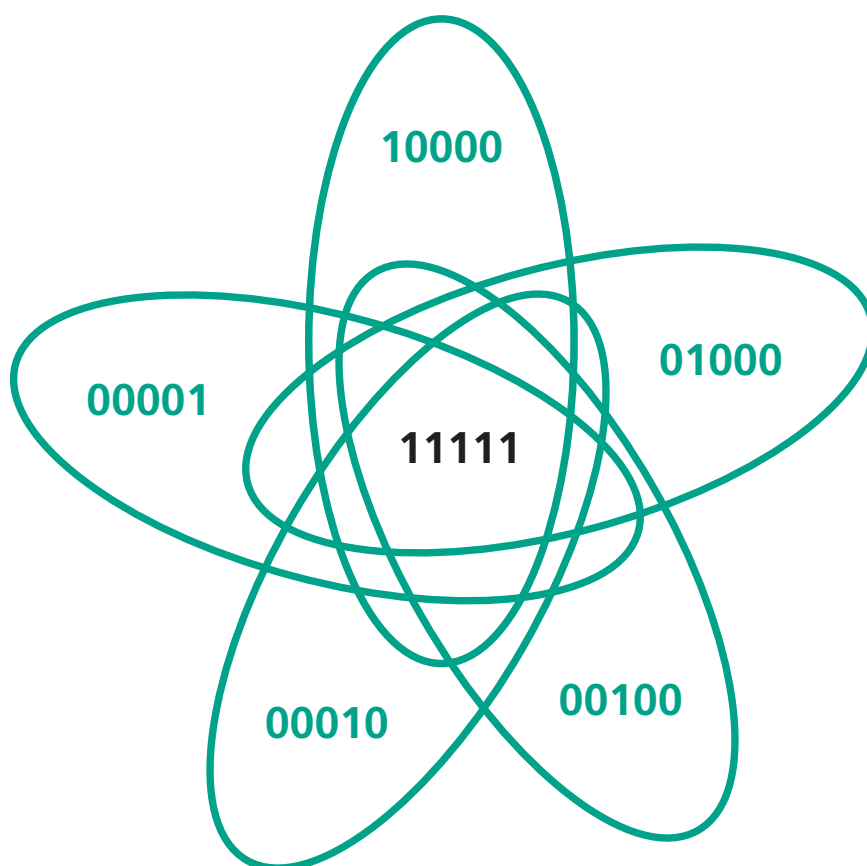


Slawomir Duszynski

## Analyzing Similarity of Cloned Software Variants using Hierarchical Set Models



Editor-in-Chief: Prof. Dr. Dieter Rombach

Editorial Board: Prof. Dr. Frank Bomarius

Prof. Dr. Peter Liggesmeyer

Prof. Dr. Dieter Rombach

FRAUNHOFER VERLAG

# **PhD Theses in Experimental Software Engineering**

## **Volume 51**

Editor-in-Chief: Prof. Dr. Dieter Rombach

Editorial Board: Prof. Dr. Frank Bomarius  
Prof. Dr. Peter Liggesmeyer  
Prof. Dr. Dieter Rombach

Zugl.: Kaiserslautern, Univ., Diss., 2015

Printing:  
Mediendienstleistungen des  
Fraunhofer-Informationszentrum Raum und Bau IRB, Stuttgart

Printed on acid-free and chlorine-free bleached paper.

All rights reserved; no part of this publication may be translated, reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. The quotation of those designations in whatever way does not imply the conclusion that the use of those designations is legal without the consent of the owner of the trademark.

© by **Fraunhofer Verlag**, 2015  
ISBN (Print): 978-3-8396-0860-9  
Fraunhofer-Informationszentrum Raum und Bau IRB  
Postfach 800469, 70504 Stuttgart  
Nobelstraße 12, 70569 Stuttgart  
Telefon +49 711 970-2500  
Telefax +49 711 970-2508  
E-Mail [verlag@fraunhofer.de](mailto:verlag@fraunhofer.de)  
URL <http://verlag.fraunhofer.de>

# Analyzing Similarity of Cloned Software Variants using Hierarchical Set Models

Beim Fachbereich Informatik  
der Technischen Universität Kaiserslautern  
zur Verleihung des akademischen Grades

**Doktor der Ingenieurwissenschaften (Dr.-Ing.)**

genehmigte Dissertation  
von

**M. Sc. Slawomir Duszynski**

Fraunhofer-Institut für Experimentelles Software Engineering  
(Fraunhofer IESE)  
Kaiserslautern

Berichterstatter:

Prof. Dr. Dr. h.c. H. Dieter Rombach  
Prof. Dr. rer. nat. habil. Gunter Saake

Dekan:

Prof. Dr. rer. nat. Klaus Schneider

Tag der Wissenschaftlichen Aussprache:

09.01.2015

**D 386**



To my wife Basia and daughter Hania



# Acknowledgements

A doctoral thesis has usually one author. However, a thesis is created not by the work of just one person, but rather bases on the support and collaboration of many people, provided on many different levels. I would like to thank everyone who in any way contributed to this thesis.

I thank Prof. Dieter Rombach for the guidance and feedback he provided along the whole thesis period, and for creating the applied research environment of Fraunhofer IESE. I thank Prof. Gunter Saake for his insightful questions and feedback, which provided a different perspective on the thesis and improved it in many ways. I thank Prof. Reinhard Gotzhein for chairing the dissertation committee.

During the thesis research, I received much support from the colleagues at Fraunhofer IESE. I especially thank Dirk Muthig for pushing me in the right direction in the very beginning, Martin Becker for his ongoing support both as a discussion partner and as a department head, and Jens Knodel for many in-depth discussions on the details of the approach. During my time at IESE, I also much benefited from countless discussions with Michalis Anastasopoulos, Thiago Burgos, Isabel John, Ralf Kalmar, Thomas Patzke, Daniel Schneider, Adeline Silva Schäfer and Bo Zhang.

I thank Andreas Jedlitschka and Jessica Jung for their support in the preparation of the empirical studies. I thank the students of the Software Product Lines course who participated in the controlled experiment. I thank Yael Dubinsky, Julia Rubin, Thorsten Berger and Krzysztof Czarnecki for the joint work on the industrial cloning survey. I thank Chanchal Roy and Jeffrey Svajlenko from the University of Saskatchewan for the discussions and the analytical evaluation support. I also thank Andreas Feldges and Jörg Wleklik for sharing their industrial experiences with the practical approach use.

I thank the students who performed parts of the implementation work: Selcuk Imal, Benjamin Precht, and Fatimah Zahra. I especially thank Vasil Tenev for the substantial improvements and extensions of the tool, the mapping algorithm, and the many algorithmic discussions. I thank Thomas Forster, Dominik Rost and Balthasar Weitzel for their technical support.

Finally, I thank my wife Barbara for continuously supporting me during the time of the thesis. She is the person who believed in me the most. I also thank my daughter Hanna for reminding me that there are things in life which are more important than work.





# Abstract

Software reuse approaches are known to enable considerable effort and cost savings during the development of a group of software systems with a significant overlap in functionality. In practice, however, the need for systematic reuse often becomes apparent only after a number of product variants have already been delivered. The existing literature and an industry survey performed in the context of this dissertation indicate that in practice, new product variants are often created by cloning the code of an existing product and changing it according to the new requirements. In a long-term perspective, this practice often leads to significant maintenance problems.

To counteract such maintenance problems, a systematic reuse approach can be introduced afterwards by transforming the implementation of the cloned product variants. However, successful transformation is a challenging task because it requires precise and detailed information about the distribution of implementation similarity between the product variants. This information is usually not available, as the product variants were modified independent of each other. The motivation for this dissertation is hence to provide the needed similarity information and thus support the migration of existing system variants towards software reuse.

The main contribution of this dissertation is a reverse engineering approach for obtaining information about the source code similarity of existing product variants. Compared to existing approaches, it delivers more detailed similarity information, reduces the analysis effort, and allows for improved correctness of similarity information understanding. The approach models the variant products as hierarchical, intersecting sets of uniquely identifiable elements, and expresses the similarity of the variants using set algebra. The resulting similarity information is available on any abstraction level, from a single code line to a whole product. The approach proposes a generic analysis framework, which can be used for diverse system representations, diverse similarity detection algorithms, and diverse definitions of element similarity. Hence, the approach can be instantiated in various contexts and adapted to a specific analysis goal.

The contributed approach supports simultaneous analysis of multiple source code variants and proposes visualization concepts that enable easy interpretation of the analysis results even for large systems and a high number of variants. The benefits of the approach are evaluated empirically by means of a controlled experiment and an industrial case study, and analytically on a reference set of cloned system variants. Furthermore, practical applications of the approach in an industrial context are briefly presented.



# Table of Contents

<b>1</b>	<b>Introduction .....</b>	<b>1</b>
1.1	Research Approach .....	3
1.2	Research Problems .....	4
1.3	Scope and Contributions.....	9
1.4	Outline.....	14
<b>2</b>	<b>Context and Related Work .....</b>	<b>17</b>
2.1	Software Reuse .....	17
2.1.1	Software Product Line Engineering .....	19
2.1.2	Software Product Line Adoption Strategies .....	22
2.2	Reverse Engineering .....	23
2.3	Similarity Analysis Approaches for Software Variants.....	27
2.3.1	Comparison and Differencing Algorithms .....	28
2.3.2	Clone Detection.....	29
2.3.3	Other Approaches .....	31
2.4	Summary .....	32
<b>3</b>	<b>Towards an Approach for Variant Similarity Analysis .....</b>	<b>33</b>
3.1	Cloning in Industrial Software Product Lines – An Exploratory Survey .....	33
3.1.1	Survey Results .....	35
3.1.2	Discussion.....	37
3.2	Application Scenarios and Analysis Goals for Code Similarity Analysis.....	38
3.3	Shortcomings of the Existing Approaches.....	41
3.4	Research Hypotheses.....	44
3.5	Summary .....	48
<b>4</b>	<b>Investigation and Formalization of the Variant Similarity Analysis Problem .....</b>	<b>49</b>
4.1	Software Variants.....	49
4.1.1	A Discussion on the Lack of Objective Variant Ordering .....	52
4.2	The Construction Requirements for Techniques Analyzing Variant Similarity .....	55
4.2.1	Consequences of the Lack of Variant Ordering .....	56
4.2.2	Providing Detailed Result Information .....	57
4.2.3	Result Presentation and Interpretation .....	59
4.2.4	Other Requirements.....	60
4.3	Assumptions Resulting from the Application Scenarios.....	60
4.4	A Formal Definition of Variant Similarity Analysis.....	63

4.5	Evaluating Quality of the Variant Similarity Analysis Results .....	66
4.5.1	Evaluating Results of Information Retrieval Problems.....	66
4.5.2	Definition of Precision and Recall Measures for Similarity Analysis Results.....	67
4.6	The Conceptual Model of Variant Similarity Analysis .....	71
4.6.1	Software Asset Structure.....	71
4.6.2	Software Asset Similarity.....	73
4.6.3	Similarity Analysis of Software Asset Variants.....	75
4.7	Summary.....	76
<b>5</b>	<b>Variant Similarity Analysis with Hierarchical Set Similarity Models.....</b>	<b>77</b>
5.1	The Set Similarity Model.....	78
5.2	Hierarchical Set Similarity Model.....	84
5.2.1	Asset Content Trees and Set Similarity Models .....	84
5.2.2	A Data Model for the Set Model Based Similarity Analysis .....	88
5.2.3	Representation of Equivalence Between Transformed Tree Structures .....	89
5.2.4	Further Remarks on the Hierarchical Set Similarity Model .....	92
5.3	A Process for Hierarchical Set Model Construction and Usage ..	92
5.4	Approaches for Set Similarity Model Construction.....	95
5.4.1	An Instantiation of Asset Content Decomposition .....	97
5.4.2	Mapping Correspondences in Structure Hierarchies .....	98
5.4.3	Definition of Atomic Content Element Equivalence Relation .....	103
5.4.4	Discussion .....	107
5.5	Visualization.....	110
5.5.1	Set Bar Diagrams: Visualizing the Similarity of Multiple Intersecting Sets.....	111
5.5.2	Visualization of Set Similarity in the Asset Structure Hierarchy .....	114
5.5.3	Visualization of Similarity Distribution .....	116
5.5.4	Size-Preserving Set Intersection Visualization.....	119
5.5.5	Similarity Visualization with Phylogenetic Trees .....	121
5.5.6	Discussion .....	125
5.6	Metrics.....	127
5.7	Discussion .....	131
5.8	Summary.....	134
<b>6</b>	<b>Analysis Tool Implementation Techniques .....</b>	<b>135</b>
6.1	Supporting Performance Optimization with Data Redundancy .....	135
6.2	Efficient Evaluation of Subset Calculations .....	136
6.3	Ensuring Repeatable Results for Multiple Optimal Solutions...	140
6.4	Summary.....	141

<b>7 Evaluation .....</b>	<b>143</b>
7.1 Analytical Evaluation .....	144
7.1.1 Performance and Scalability .....	146
7.1.2 Input and Result Transitivity .....	148
7.1.3 Approach Instantiation Correctness: Precision and Recall .....	151
7.2 Controlled Experiment .....	153
7.2.1 Experiment Goal and Hypotheses.....	153
7.2.2 Experiment Design and Operationalization.....	155
7.2.3 Experiment Results.....	158
7.2.4 Threats to Validity .....	164
7.3 Industrial Case Study.....	166
7.4 Industrial Application Experiences .....	170
7.5 Summary .....	172
<b>8 Summary and Outlook.....</b>	<b>173</b>
8.1 Results and Contributions .....	173
8.1.1 Understanding Large-Scale Cloning: Reasons, Consequences and Solutions .....	173
8.1.2 A Set Model Based Approach to Variant Similarity Analysis .....	174
8.1.3 Empiricism and Evaluation .....	175
8.1.4 Further Approach Benefits .....	177
8.2 Limitations .....	177
8.3 Future Work.....	179
8.3.1 Extending the Analysis Approach .....	179
8.3.2 Further Evaluation.....	180
8.3.3 Open Research Questions .....	180
8.4 Concluding Remarks .....	182
<b>References.....</b>	<b>183</b>
<b>Appendix A Experiment Material .....</b>	<b>199</b>
A.1 Experiment Infrastructure Setup .....	199
A.1.1 Analyzed Software Systems .....	199
A.1.2 Adaptations to the Variant Analysis Tool.....	199
A.1.3 Computing Infrastructure .....	200
A.2 Experiment Documents .....	201
A.2.1 Tool Tutorials .....	202
A.2.2 The Main Experiment Document.....	205
A.3 Raw Experiment Results .....	216
A.3.1 Briefing Questionnaire Results.....	216
A.3.2 Task Answers and Time Measurement .....	218
A.3.3 Debriefing Questionnaire Results .....	220

<b>Appendix B</b>	<b>Application Guidance .....</b>	<b>221</b>
B.1	Reuse Potential Assessment and Software Consolidation.....	221
B.1.1	Similarity Properties Supporting the Scenarios.....	222
B.1.2	Prioritization of Consolidation Activities .....	222
B.1.3	Further Activities Supporting the Scenarios .....	223
B.2	Parallel Variant Maintenance .....	224

# List of Figures

Figure 1	The research approach followed in this thesis .....	3
Figure 2	Example of system cloning: the history of BSD-based operating systems [Yamamoto 2005] .....	4
Figure 3	The research context of the Variant Analysis approach .....	10
Figure 4	The role of code similarity analysis in an example reuse migration process .....	11
Figure 5	Thesis contributions .....	13
Figure 6	The empirical investigations along the research cycle .....	14
Figure 7	Thesis chapters and contributions mapped to the research approach structure .....	15
Figure 8	A schema of product line engineering (adapted from [Muthig 2002]) .....	20
Figure 9	Forward engineering, reverse engineering, and reengineering (adapted from [Chikofsky 1990]) .....	24
Figure 10	The relationships between data, information and knowledge (from [Liew 2007]) .....	25
Figure 11	A generic reverse engineering process and its relation to data, information and knowledge .....	26
Figure 12	Example presentation of multi-system similarity analysis results in the form of pairwise similarity matrix (left; the values indicate the degree of similarity) and a multi-system scatterplot (right; the similarity is indicated by the cell color) .....	31
Figure 13	Overview of the practical and scientific hypotheses and their relations to the application scenarios .....	47
Figure 14	Versions and variants of a software asset .....	50
Figure 15	The basic analysis schema for software versions and variants .....	51
Figure 16	A schematic visualization of example content change across a group of related asset variants .....	53
Figure 17	Two example intersecting sets A and B .....	56
Figure 18	The inadequacy of pairwise result presentation: identical results are provided (middle) although the analyzed situations (left, right) strongly differ .....	58
Figure 19	A schematic presentations of the similarity analysis input (left) and the analysis result (right) .....	65
Figure 20	A model of an information retrieval problem and the four possible result categories .....	66
Figure 21	An example of an incorrect analysis result .....	67



Figure 22 The incorrect result from Figure 21 (left) and its interpretation according to the merge-based analysis result definition (middle, with legend to the right).....	69
Figure 23 The conceptual model of variant similarity analysis .....	72
Figure 24 Equivalence sets construction: the elements of the input asset content sets are assigned to the equivalence classes (left). The resulting equivalence sets, containing these classes, intersect with each other (right). .....	79
Figure 25 The input sets (left) and the analysis result from Figure 19 (middle) represented with the set similarity model (right). ....	80
Figure 26 Example subset calculations. ....	81
Figure 27 A metamodel of the tree-based structure of asset content hierarchy.....	84
Figure 28 A set of asset content elements and an asset content tree. ...	85
Figure 29 Three assets are decomposed into asset content trees, with atomic content elements stored in the tree leaves (left, middle). A unified asset content tree is constructed from the asset content trees, and its elements reference the set similarity models of atomic content elements (right).....	86
Figure 30 The construction of set models for parent elements of the unified asset content tree based on the child element set models. ....	87
Figure 31 A data metamodel for the similarity analysis: the content structure of each asset (left) is associated with two set models (right), which store the asset content similarity information...	89
Figure 32 Three assets content trees from Figure 29 (left), with the node $\Pi$ moved inside the parent node $\Phi$ in the second variant, are used to construct a unified asset content tree. The tree (right) contains hard links in all original locations of the node $\Pi$ , which reference the same set similarity model.....	90
Figure 33 The construction of set similarity models for the parent node $\Phi$ (left) and for the root node $\Omega$ (right) of the unified asset content tree from Figure 32. ....	91
Figure 34 The process for the set model based similarity analysis. ....	93
Figure 35 The similarity analysis input (left), the internal analysis process using the customizable definitions of analysis mechanisms (middle), and the resulting analysis output (right).....	94
Figure 36 Abstract forms of input similarity data: gradual pairwise similarity (left) and binary pairwise similarity (right).....	95
Figure 37 The iterative Center Star method: in each iteration the star center variant is determined, and its elements are mapped using the center's pairwise alignments. ....	102
Figure 38 Examples of non-transitive relation graphs constructed from the diff results. ....	104

Figure 39 Example relation graphs and their alternative solutions. ....	106
Figure 40 Solutions provided for the relation graphs from Figure 38...	107
Figure 41 A Venn diagram for five intersecting sets (left) and the construction of a set bar diagram for these sets: the intermediate form (top right) and the final diagram form (bottom right). ....	112
Figure 42 The visualization of an example subset calculation in the set bar diagram.....	113
Figure 43 Hierarchy structure visualization showing similarity information for each displayed element. ....	114
Figure 44 Visualization of code-level similarity with line background coloring, category icons, and on-demand details.....	116
Figure 45 Distribution diagram visualization: the illustration of the construction principle, created for the <i>io</i> folder from Figure 43 (left), and a diagram screenshot for a large industrial system group (right). ....	117
Figure 46 Tree map visualization: the illustration of the construction principle, created for the <i>src</i> folder from Figure 43 (left), and a screenshot for a code folder from three BSD systems (right). The color intensity shows the proportion of core code in the set union of a given element.....	118
Figure 47 A Venn diagram for four industrial system variants, indicating the intersection sizes (left). The same intersecting sets visualized using a tree map set diagram (right).....	119
Figure 48 Example hierarchy structures, created for four sets, which can be used in a tree map set diagram.....	120
Figure 49 Dendrogram constructed for the full source code of six BSD systems [Tenev 2012]. The location of branching points corresponds to the similarity of the respective tree branches. ..	122
Figure 50 The similarity of a group of versions (left) as compared to a group of variants (right). The drawn Venn diagrams are area-proportional, i.e. the size of an area indicates its cardinality.....	124
Figure 51 Cladogram constructed for the full source code of six BSD systems depicted in Figure 49 [Tenev 2012]. The length of branch sections is proportional to the amount of common or unique code.....	124
Figure 52 A data metamodel from Figure 31, with the additional attributes storing redundant, performance-relevant set model information.	136
Figure 53 A screenshot of the user interface for specifying subset calculation condition (called a “query” in the tool).....	137
Figure 54 The overview of the practical and scientific hypotheses and the used evaluation means. ....	143
Figure 55 The transitivity measurements for subgroups of Ind_Medium variants. ....	150

Figure 56 The three measurement series for the evaluation of approach precision and recall. ....	152
Figure 57 The experiment results: task time (left) and task errors (right)...	160
Figure 58 The experiment results: cognitive load. ....	161

# List of Tables

Table 1	The results of an industry survey concerning product line adoption strategies [Berger 2013] .....	4
Table 2	Two dimensions of product line adoption (based on [Bosch 2002] and [Krueger 2002]).....	22
Table 3	The properties of the existing approaches: + stands for "supported", (+) for "partially supported", "-" for "not supported" .....	43
Table 4	The practical hypotheses .....	44
Table 5	The scientific hypotheses.....	46
Table 6	The calculation formulas and example values for the measures used in the merge-based precision and recall definition .....	70
Table 7	The metrics characterizing a group of intersecting sets.....	128
Table 8	The metrics characterizing the distribution of the variant sets in the structure hierarchy .....	129
Table 9	The metrics characterizing similarity distribution in the textual file content.....	130
Table 10	Truth table evaluating the correctness of the quick evaluation expressions .....	139
Table 11	Example calculations using the quick evaluation expressions .	139
Table 12	The analyzed variant system groups .....	145
Table 13	A short description of the analyzed systems' similarity.....	146
Table 14	The performance and scalability measurements .....	147
Table 15	The transitivity measurements for all variant groups .....	148
Table 16	The transitivity measurements for subgroups of Ind_Medium variants .....	150
Table 17	Approach precision and recall, measured on the five generated system variant groups.....	152
Table 18	The main metrics collected during the controlled experiment and the associated hypotheses.....	155
Table 19	The experiment process: steps for controlled experiment execution .....	157
Table 20	The briefing questionnaire results: participant background..	158
Table 21	The briefing questionnaire results: participant experience....	159
Table 22	The briefing questionnaire results: participant motivation....	159
Table 23	Statistical testing of the experimental hypotheses .....	161
Table 24	The debriefing questionnaire results.....	163

Table 25	The case study questionnaire results.....	168
Table 26	Industrial applications of the analysis approach .....	171
Table 27	The experiment results: the briefing questionnaire .....	217
Table 28	The experiment results: time measurement and task answers..	219
Table 29	The experiment results: the debriefing questionnaire .....	220

# List of Definitions

Definition 1	Software reuse.....	17
Definition 2	(Reuse) asset.....	18
Definition 3	Systematic software reuse.....	18
Definition 4	Software product line .....	19
Definition 5	Reverse engineering.....	24
Definition 6	Reengineering.....	24
Definition 7	Data .....	25
Definition 8	Information.....	25
Definition 9	Knowledge .....	25
Definition 10	Software clones.....	29
Definition 11	Software version .....	50
Definition 12	Software variant.....	50
Definition 13	Precision.....	67
Definition 14	Recall .....	67
Definition 15	Similarity .....	73
Definition 16	Equivalence set.....	79
Definition 17	Equivalence set intersection.....	79
Definition 18	Equivalence set union.....	79
Definition 19	Set similarity model .....	80
Definition 20	Set model based similarity analysis .....	80
Definition 21	Asset content tree.....	85
Definition 22	Unified asset content tree.....	85
Definition 23	Hierarchical set similarity model.....	88



# 1 Introduction

Software customization	Software systems frequently need to fulfill the requirements of various user groups and work in diverse technical environments on disparate hardware platforms. An inevitable consequence of this variety of requirements is that software systems are often developed not as a singular instance, but rather as a family of similar system variants which provide functionality customized for particular user groups and environments. Software system customization is “unavoidable and purposeful” [Parnas 1976], and it is currently being practiced in a broad range of software-intensive industries [SPLC 2014].
Software reuse benefits customization	As the system variants usually remain considerably similar despite the customization, adopting large-scale reuse of software assets frequently brings benefits in their development. First, as the software assets are reused in multiple software projects, less code needs to be developed anew, which results in a reduction of project development effort and cost. Second, this reduction in effort helps to shorten project development time. And third, the reused assets have already been verified in past projects – hence they usually have higher quality than freshly developed code, which leads to higher quality of the final system. From the multitude of existing software reuse approaches, the software product lines approach is especially being advocated for the development of a group of similar software systems, and its adoption is known to enable the achievement of the reuse benefits listed above [Gacek 2001] [Clements 2002a] [Bass 2003] [Steger 2004] [SPLC 2014].
Customization via system cloning	The adoption of a systematic reuse approach needs advance planning as well as initial investment in the reusable asset base, and might also require restructuring of the software-developing organization and redefinition of its processes [Lim 1998] [Clements 2002a]. Hence, adopting software reuse requires time, money and management commitment. In industrial practice, however, factors such as lack of planning certainty, development resource constraints, and tight deadlines in many cases prevent a software-developing organization from adopting a systematic reuse approach. In such situations, the existing system variants are frequently just cloned and modified to create the next system variant [Dubinsky 2013]. The cloned system variants typically undergo further parallel development, and reuse approaches are often not introduced until after the variants have matured. As a consequence, the developing organization misses the benefits of software reuse. Additionally, the organization faces increased maintenance effort, as many tasks need to be duplicated between the system variants, and each duplicated task needs to be carefully verified due to a potentially different context in each of the systems [Ray 2012].



Migration to software reuse	<p>In a system cloning situation such as described above, the adoption of a systematic reuse approach remains a tempting proposition – particularly as the development of further system variants and functionalities escalates the maintenance challenges. When striving for reuse, the developing organization faces a choice between building reuse-based systems from scratch or migrating the existing system variants towards a reusable form. This “rebuild or migrate” decision depends on many factors, such as longevity of the systems, quality of the existing code, and the degree of requirement and code similarity between the migrated systems. In case the similarity and the quality of existing system variants are high enough, and the maintenance is planned to continue in the future, migration of the systems or at least their selected parts is frequently the better option, because complete rebuilding would mean a loss of the past investments already made into the development of the systems [Simon 2002]. However, such a migration is a complex and effort-intensive undertaking, as several variants of each migrated software asset need to be understood and to be transformed correctly into the new reusable form.</p>
Thesis motivation and content	<p>The motivation for this thesis is to support the migration of existing system variants towards software reuse. In many cases, reuse migration is impeded by the lack of sufficient similarity information about the source code of the migrated software assets (see Section 1.2). In an industrial survey, we observed that the similarity information tends to be quickly lost during the development of cloned system variants (Section 3.1). Moreover, the existing approaches for recovering that information exhibit deficiencies (Section 3.3). Hence, this thesis contributes a reverse engineering approach, named Variant Analysis, for obtaining the similarity information from the source code of the migrated system variants. Compared with the existing approaches, Variant Analysis delivers more detailed similarity information, reduces the analysis effort, and improves the correctness of similarity information understanding. The approach definition is based on a formalization of the variant similarity analysis problem (Chapter 4), and introduces a generic analysis framework based on modeling the analyzed system variants as hierarchical sets of uniquely identifiable elements having known sizes (Chapter 5). The approach supports simultaneous analysis of multiple source code variants and proposes visualization concepts that enable easy interpretation of the analysis results, even for large systems and a high number of variants. The benefits of the approach are evaluated empirically by means of a controlled experiment and an industrial case study, and analytically on a reference set of cloned system variants (Chapter 7).</p>
Introduction content	<p>Before delving into the detailed content of the thesis, in this introduction we discuss the followed research approach, present the addressed research problems, describe the scope of this thesis, and outline its contributions and structure.</p>

## 1.1 Research Approach

This thesis follows the experimental software engineering paradigm [Basili 1993] and the Fraunhofer method of addressing research problems with industrial relevance [Rombach 2000]. The stages of the resulting research approach, illustrated in Figure 1, are:

- In the **Practical Problem Identification** stage, the state of the industrial practice is analyzed, for example through observation of organizations developing software or with the help of a literature review, in order to identify existing problems.
- The **Scientific Problem Identification** stage concerns an investigation of the background of the practical problem to discover the underlying reasons and formulate research questions. The improvement hypotheses related to the identified practical and scientific problems are stated.
- In the **Solution Development** stage, a new approach, intended to solve or mitigate the identified scientific and practical problems, is researched.
- The **Scientific Benefit Evaluation** stage is an (at best empirical) assessment of whether the scientific hypotheses concerning the developed solution can be confirmed.
- The **Practical Benefit Evaluation** stage is likewise an (at best empirical) evaluation of the practical hypotheses concerning the solution.

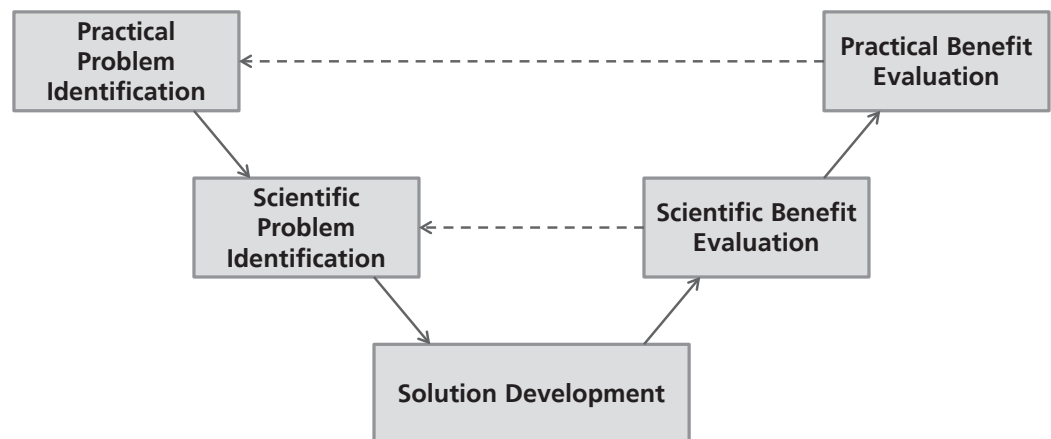


Figure 1

The research approach followed in this thesis

In the following sections, we discuss the content and background of this thesis according to the described approach stages. Afterwards, we map the thesis contributions and chapters to the approach structure.

## 1.2 Research Problems

System cloning is frequent

Software customization realized via system cloning is frequently reported in the literature. System cloning is applied both for open-source systems, where it is known as forking [Ernst 2010] [Robles 2012], and for commercial software in various industries [Dubinsky 2013]. We also observed cases of system cloning in our industrial consultancy projects [Duszynski 2008] [Duszynski 2011a]. Figure 2 presents a well-known example of an open-source system fork, which resulted in the creation of the BSD-based operating systems family.

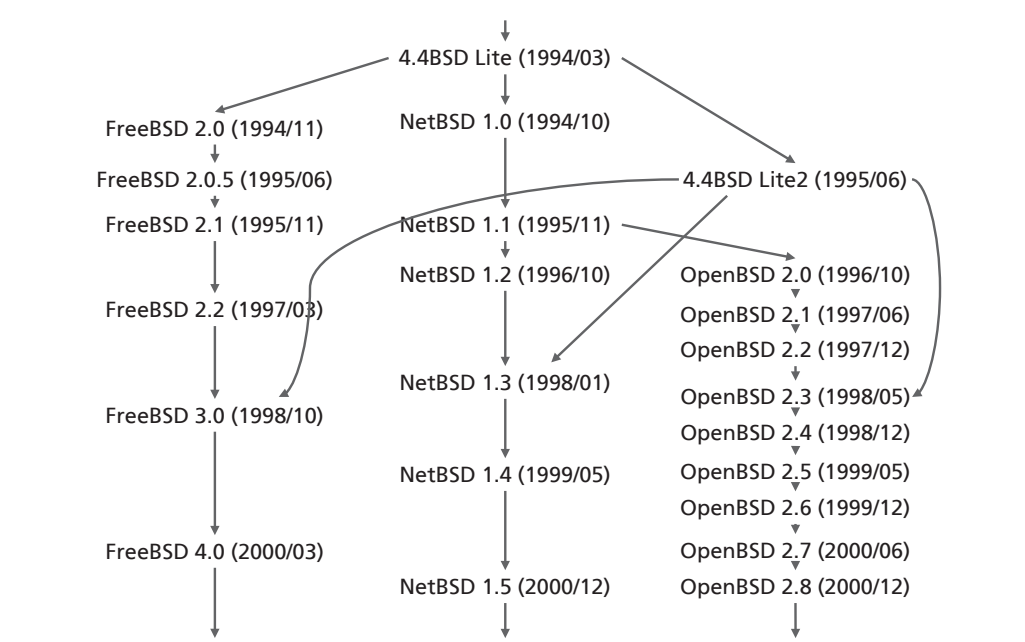


Figure 2 Example of system cloning: the history of BSD-based operating systems [Yamamoto 2005]

Reuse migrations are frequent

Several successful migrations of cloned systems to software reuse have been reported [Faust 2003] [Riva 2003] [Staples 2004] [Jepsen 2007]. In a recent industrial survey [Berger 2013], reuse migration of independent products was reported as the most frequent way of software product line adoption: 50% of the 42 participants, each of whom was involved in developing software product lines, stated that they created at least one product line using this strategy (Table 1).

Product line adoption strategy	Proportion of participants who applied the strategy
Proactive: product line was developed before any product was derived	35.30 %
Reactive: a single product was evolved into a product line	47.10 %
Extractive: multiple existing products were reengineered into a product line	50.00 %
Any combination of the strategies above	26.50 %
Other	20.60 %

Table 1 The results of an industry survey concerning product line adoption strategies [Berger 2013]

Practical migration problems are severe

However, reuse migration is usually a complex and effort-intensive undertaking, which requires extensive restructuring of the system assets and affects the organization's structure and processes [Clements 2002a]. In addition to the high effort and complexity, many reported reuse migrations fail to fully exploit the reuse potential resulting from the similarity of the migrated system variants – they miss the existing reuse opportunities. In some cases, the migration of system variants, although potentially beneficial, is not even attempted. As a consequence, the organization continues to face the high maintenance effort resulting from many duplicated tasks, which have increased difficulty as each of the involved code locations in the variant systems might be slightly different due to past customizations [Ray 2012]. Hence, the practical problems related to reuse migration are:

- missing reuse opportunities due to a risk-averse migration process, where only the best-understood assets and systems are migrated, and the remaining cloned variants continue to be maintained in separate code bases [Jepsen 2007],
- missing reuse opportunities due to a lack of knowledge about whether assets similar to a given one exist [Dubinsky 2013],
- incorrect assessment of the achievable degree of reuse, leading to an overly optimistic migration plan [Yoshimura 2006] [Kolb 2006a],
- a loss of the past investments made into the existing products through rejection of migration plans and implementation of new reusable products from scratch [Beyer 2008],
- continued maintenance challenges and a deteriorating code base resulting from the postponement or rejection of reuse migration [Dubinsky 2013].

Practical problems

Hence, the lack of sufficiently detailed and dependable code similarity information contributes to increased maintenance effort for cloned system variants, and reuse migration, if attempted, might require more restructuring effort and achieve a lower reuse rate than if this similarity information were available. Therefore, we identified the following **practical problems**:

Migration of cloned software variants towards software reuse is effort-intensive, and is likely to miss some of the existing reuse opportunities.

For the cloned asset variants that are not migrated, their continued maintenance is also effort-intensive due to repetitive tasks applied in varying contexts of different system variants.

Industrial survey: cloning can be a valid strategy

To better characterize the initial situation related to software system cloning, with a group of researchers we performed an exploratory survey on six industrial system families developed with the use of cloning [Dubinsky 2013] (see Section 3.1). One of our main findings was that there are several justified reasons for cloning a software system, even though cloning later causes the maintenance problems discussed above. The initial effort investment is perceived by the survey participants to be

significantly lower for cloning than for a systematic reuse approach, which makes cloning a preferred development approach in case the available resources (effort, time) are scarce in the short term. Furthermore, cloning increases planning independence and flexibility by eliminating the need to coordinate the development across a group of software variants, which would be necessary for reusable assets. And finally, the lack of planning certainty, caused for example by unpredictable market developments, may make it impossible to recognize the reuse potential of the software system variants upfront. Only as the first, experimental products turn out to be a success, and requirements for further variants emerge, does the longer-term perspective of introducing a software reuse approach become viable. Hence, the use of system cloning is not a fault of the developers, but rather a pragmatic response to the specific situation in which they find themselves. Therefore, it is reasonable to expect that cloning will continue to be practiced as a software customization approach for future software systems, and that the need to support software-developing companies with suitable responses to the stated practical problems will persist.

Industrial survey: Furthermore, we found out in the survey that as the cloned software variants are modified during evolution, their developers quickly lose the overview of the similarities existing in the variant code. Naturally, each of the clones is modified in a different way, as each of them realizes a different functionality. However, we discovered that even those changes that should be applied to all the clones are not always propagated consistently. Also, the knowledge about the cloning activities and subsequent changes was not managed in the surveyed organizations and was therefore lost quickly. As a consequence, the developers were not able to assess the degree of similarity between the code of different software assets. For example, they could not determine which variants are relevant for a specific code change, or had problems selecting a suitable initial code variant that could be cloned to develop a new variant with the lowest possible effort. Hence, we identified that the surveyed organizations lack sufficient information on the similarity of their variant code – a finding that we also observed in our industrial consultancy projects.

The importance of similarity information Dependable assessment of the degree of similarity between different asset variants is crucial for reuse migration. Reuse migration typically requires extensive restructuring of the system assets. Hence, in migration planning it is essential to characterize with sufficient detail the starting point – the state of the software at present – and the target state in which the reuse approach is to be operational. Among other inputs, the asset similarity information is crucial in a range of significant migration decisions, for example:

- the selection of assets to be migrated,
- the choice of specific variants of the assets to be migrated,
- the assessment of reuse potential, that is the achievable degree of reuse,

- the assessment of migration difficulty and effort for the particular asset,
- the prioritization of the migration tasks,
- the selection of the implementation-level migration approach, e.g., whether a group of asset variants is merged or whether a single variant is extended to cover the functionality required by all systems.

Even if reuse migration is not attempted, code similarity information still provides much help in maintenance activities for the cloned variants, as it helps to classify the similar assets and supports correct change propagation between the clones [Toomim 2004] [Nguyen 2012].

#### Scientific problems

Based on these findings, our hypothesis is that the lack of sufficient code similarity information contributes to the identified practical migration problems such as high effort, missing reuse opportunities, or deferral of the migration. Therefore, recovering the code similarity information is a scientific problem with practical significance. In particular, the recovered information should be sufficient for the practical needs of reuse migration – it should dependably support the developers in the migration decisions listed above. The similarity information should be sufficiently detailed, should be available on any level of abstraction (from small code chunks to whole systems), and should be available for any subgroup of the analyzed system variant family. Moreover, the developers should be able to understand the delivered information efficiently and correctly. Hence, the **scientific problems** addressed by this thesis are:

How to recover similarity information from the source code of multiple similar software asset variants, in sufficient quality to support reuse migration or parallel maintenance of these assets?

How to structure and present the recovered information in a way that enables humans to understand it efficiently and correctly?

#### Existing similarity analysis approaches

The current approaches for recovering asset similarity information can be divided into two categories:

- Top-down similarity analysis involves an examination of the high-level descriptions and representations of the software, and assesses whether the functionality of the asset variants is identical or at least similar enough to enable reuse. An example of a top-down functionality-based similarity analysis approach is product line scoping [Schmid 2002a].
- Bottom-up similarity analysis involves an examination of the low-level implementation assets, most frequently the source code, in order to determine if the variant implementations are similar enough to be replaced by a single-copy, generic and reusable asset. Bottom-up similarity analysis approaches are realized with reverse engineering techniques [Chikofsky 1990] and are the focus of this thesis.



### Difference between function and code similarity

In most cases, the functionality of the migrated software systems is well known to its architects and developers. Hence, they have enough information to perform a top-down functionality-based similarity analysis. However, a top-down analysis is frequently not detailed enough to account for minor, but purposeful differences in the asset functions, e.g. due to differences in supported hardware platforms or the realized non-functional requirements. As a result, the implementation similarity of the analyzed assets might be significantly different than the similarity of their functionality. For example, Yoshimura et al. analyzed two variants of an automotive engine control system, and found that “the portion of functional commonality among two products is about 60-75%; their implementations, however, share as little as around 30% of code” [Yoshimura 2006]. We also experienced a similar case when a top-down functional analysis overlooked important differences between system variants [Wleklik 2011]. Although understanding of the functional similarity is necessary for the successful adoption of a software reuse approach, it is not sufficient if the existing implementation assets need to be migrated to a reusable form. The result difference between the two similarity analysis approaches indicates that obtaining the implementation-level similarity information is crucial for correctly planning a reuse migration. However, this is a difficult task, as software systems are frequently implemented using hundreds of thousands or even millions of source code lines, and that amount of code needs to be further multiplied by the number of analyzed system variants. The large amount of code-level information cannot be comprehended and analyzed directly by a human. As a result, software architects and developers are often unable to assess the code similarity of the developed system variants or their parts, as indicated by our industrial survey.

### Shortcomings of existing approaches

The large amount of analyzed source code calls for automated reverse engineering approaches developed to structure, abstract, and analyze the code similarity information and to allow humans to understand the analysis results correctly and efficiently. However, the results delivered by the existing reverse engineering approaches are lacking important details. For example, calculating similarity metrics on software system variants [Yamamoto 2005] is not sufficient as there is no information about the locations of code parts recognized as similar or different. On the other hand, recovering detailed variant code similarity with the use of clone detection techniques [Roy 2009a] creates a large number of unstructured results for any non-trivial variant set [Svajlenko 2013]. Although these results technically contain all the relevant similarity data, they require effort-intensive manual analysis, and the large amount of data makes it impossible for a human to fully comprehend the analyzed situation within a realistic period of time. Furthermore, the current approaches for structuring and abstracting clone detection information [Yoshimura 2006] [Mende 2008] are not satisfactory, as they only

provide results for any selected variant pair, but do not aggregate them for larger groups of three or more variants. Finally, clone detection techniques provide results that are recognized as similar enough according to the specified threshold, but may still be different enough for a developer to discard the possibility of reuse [Roy 2007]. This makes clone detection results not fully dependable as the code identified as similar cannot be classified as a reuse candidate with complete certainty without manual verification. In Chapter 3, we discuss the shortcomings of the existing approaches in more detail.

### 1.3 Scope and Contributions

Hierarchical set models	To address the research problems described above, in this thesis we contribute a reverse engineering approach for obtaining the similarity information from the source code of software asset variants. In the approach, we propose a generic analysis framework based on modeling the analyzed system variants as hierarchical sets of uniquely identifiable elements having known sizes. The hierarchical set similarity models provide a data structure that, to a large extent, does not exhibit the outlined deficiencies of other approaches. The set models can structure a large amount of code similarity data containing the necessary degree of detail, while the proposed abstraction and visualization concepts enable easy interpretation of the analysis results with low manual effort – even for large software systems (1 MLOC and more) and a high number of variants (10 and more). The proposed approach is named “Variant Analysis”.
Focus on similarity between variants	As the main purpose of the approach is to support the consolidation of software variants in reuse migration, it focuses on detecting the similarity <u>between</u> software asset variants. The similarity existing within a particular variant is not addressed, as the detection of such similarity presents a different kind of research problem (see Chapter 2). Also, the approach assumes that a relatively high structural similarity exists between the analyzed asset variants, for example due to their common origin in a cloning process. Hence, it is less suitable for analyzing functionally similar, but structurally different systems developed independent of each other.
Scope of the approach	Figure 3 depicts the relation of our approach to the complementary research concerns. Our approach focuses on structuring, abstracting, and visualizing the cross-asset similarity information in order to enable developers to understand this information efficiently and correctly. The use of a similarity detection algorithm working with the detailed asset content is needed to create the input similarity data. Hence, we provide generic means for accommodating a range of such existing algorithms depending on developer needs.



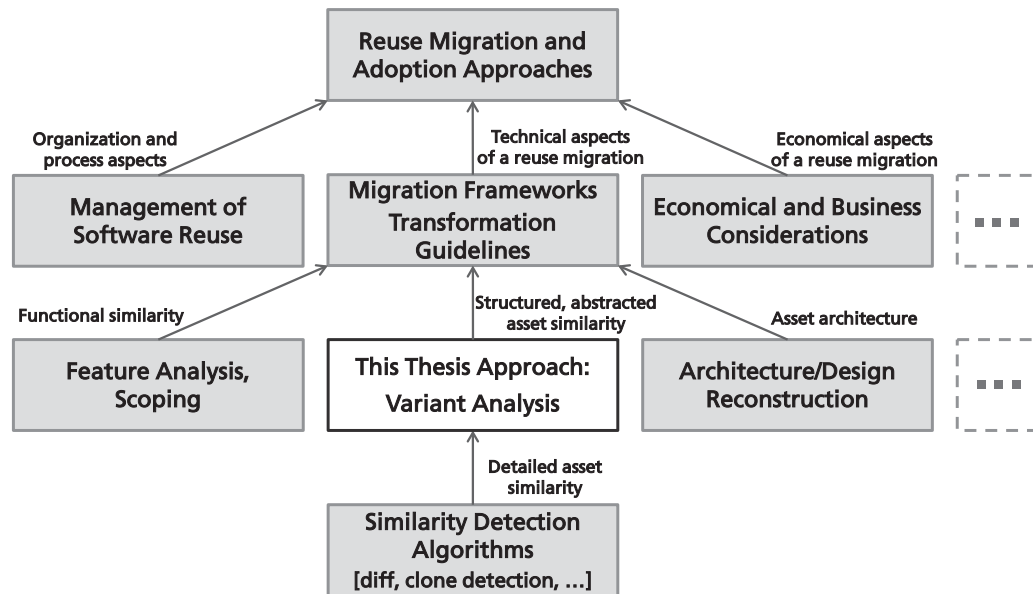


Figure 3 The research context of the Variant Analysis approach

### Role of the approach

The output of the approach is the structured, abstracted and visualized similarity information. Although we discuss the interpretation of this information and suggest possible resulting migration decisions, we do not define a general migration methodology or guidance. The reason for that is that the code similarity information, reflecting the available reuse potential, is just one of many criteria influencing migration decisions [Schmid 2005]. For example, Koskinen et al. [Koskinen 2005] identified and empirically validated 45 different criteria influencing decisions on software modernization, and still stated that their list is incomplete. The decision criteria for reuse migration can be of a technical nature (code quality [Wleklik 2011], functional similarity), but they may also concern the organizational structures and processes supporting software reuse, the economics of reuse decisions (future product development plans, available resources, scheduling of reuse migration activities), and others. The role of our approach is hence to provide similarity information as input for the higher-level migration methods and frameworks integrating the various technical aspects, such as the framework of Rubin et al. [Rubin 2013], which in turn provide just the technical perspective to holistic reuse adoption approaches such as the Carnegie Mellon Software Engineering Institute's Adoption Factory [Clements 2002a] [Northrop 2004].

In our approach, we concentrate on the similarity of the source code, and see approaches such as feature-based similarity analyses (e.g., scoping [Schmid 2002a]) and software family architecture reconstruction [Kang 2005] [Koschke 2009] as complementary, but not overlapping with our approach. This view is consistent with several existing reuse reengineering approaches that advocate the use of multiple information sources, including the analysis of functionalities, architecture, and asset implementation details [DeBaud 1998] [Bayer 1999] [Knodel 2005] [Kolb 2006b]. Figure 4 depicts the role of our approach in an example software reuse migration process. The Analysis and Evaluation process

phases can be repeated iteratively in case the evaluation uncovers new information needs requiring an extension of the previously performed analyses. Similarly, the migration process can be iterative itself – for example, it can be used to periodically reassess the state of the managed product portfolio and perform corrective migration actions if needed.

#### Practical hypotheses

The purpose of our approach is to support reuse migration and parallel maintenance of a group of similar, possibly cloned, software system variants by delivering detailed code similarity information. As discussed above, we found out in an industrial survey that the code similarity information is usually not available for the migration stakeholders – although its availability is crucial for the quality of migration decisions. Our hypotheses concerning the identified practical migration problems are therefore that the availability of detailed code similarity information has the following effects:

- it reduces migration effort,
- it increases the degree of reuse achieved in the migration,
- in case the migration is not attempted, it reduces the effort for further parallel maintenance of system variants.

#### Scientific hypotheses

Consequently, a similarity analysis approach should support the migration stakeholders by obtaining the needed information efficiently and correctly. Hence, the hypotheses related to the identified scientific problems state that our approach fulfills this purpose better than the other related approaches:

- it reduces the effort for analyzing and understanding the similarity information,
- the degree of effort reduction increases with an increasing number of analyzed variants (i.e., for a higher number of variants, the contributed improvement is greater),
- it allows the migration stakeholders to understand the implemented similarity with a higher degree of correctness.

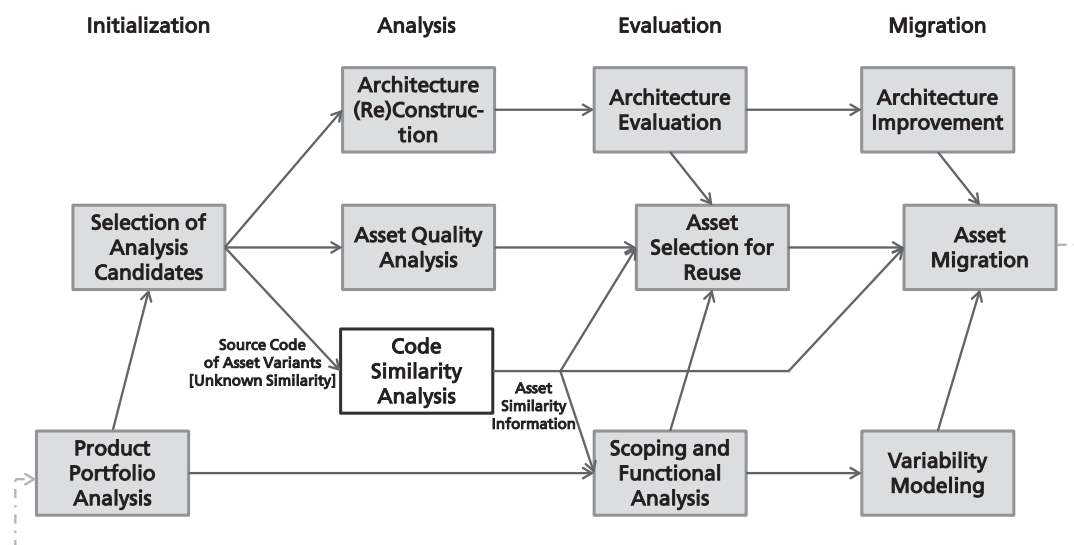


Figure 4

The role of code similarity analysis in an example reuse migration process

Analytical evaluation	Naturally, the basic prerequisite for applying any analysis approach is that the obtained results are technically correct and complete. In Chapter 7, we describe the analytical evaluation of the correctness and completeness of our approach results.
Empirical evaluation	We evaluated a subset of our practical and scientific hypotheses empirically. In a controlled experiment, we investigated the effect of using the set model on the effort and correctness of source code similarity analysis. In an industrial case study, we asked the participating developers to assess the effect of information delivered by the approach on reuse migration. In both evaluations, the collected empirical results supported our hypotheses. As the performed empirical evaluations provide just singular data points, a further, more extensive evaluation of the stated hypotheses remains to be performed as interesting future work. We describe our hypotheses in more detail in Chapter 3, and provide the details of the controlled experiment and of the case study in Chapter 7.
Approach use beyond reuse migration	Although the main motivation for our approach is to support the migration of cloned software variants towards a reuse approach, for example software product lines, we do not assume or distinguish any such specific reuse approach. The provided code similarity information can be used for migration to any approach, as well as for other informative or analytical purposes. For example, an analysis of an old, discontinued system variant group that will not be maintained or migrated anymore can still be helpful for the planning of its successor, as similarities in the new variants are most likely to occur in the same functional areas.
Set models for generic similarity analysis	Taking a more general view, the set models provide a general-purpose approach to structuring and presenting the results of any kind of comparison, performed on hierarchical structures composed from any kind of comparable atomic elements. Given suitable comparison functions, the set models can be constructed and visualized not only for code, but also for software models and even for non-software assets. In Chapters 4 and 5, we specify and discuss the requirements on such functions that are necessary and sufficient for defining a set model based similarity analysis in a generic case.
Contributions	<p>Figure 5 presents the detailed contributions of this thesis and assigns them to four main contribution categories: formalization, methodology, instantiation, and evaluation and empiricism. In this thesis, we make the following contributions:</p> <ul style="list-style-type: none"><li>• <b>Formalization of the variant similarity analysis.</b> We define a conceptual model that classifies and relates the concepts associated with the variant similarity analysis problem. We discuss the general properties of software variants and derive from them a group of formalized requirements concerning the construction of the analysis technique.</li></ul>

Based on the requirements and the scope of our application scenarios, we formally define a variant similarity analysis technique and a method for evaluating the quality of its results. Although the conceptual model, the requirements, and the definitions serve as a theoretical foundation for our approach, we believe they are useful for defining and evaluating any kind of multi-system similarity analysis technique.

- **Variant similarity analysis method.** We define a generic similarity analysis method based on the hierarchical set similarity models. We propose visualization concepts that help the interpretation of the model information, and define metrics that provide additional information to support migration decisions. The generic method can be used to structure and present the results of various similarity analysis algorithms applied to various types of content such as source code, models, and non-software assets.
  - The core idea of the method is the use of **hierarchical set similarity models**. The set models structure the similarity information in a way that is both technically viable and easy to understand for humans, even for a large number of analyzed variants, and make it available on any abstraction level, from a single code line to a whole system. We discuss the algorithms and activities needed for set model construction. In a controlled experiment, we show the benefits of the set models: reduced effort and improved correctness of source code similarity analysis. Furthermore, we analytically evaluate the high degree of correctness and completeness of the provided analysis results, and collect measures related to the potential drawbacks of set model use, such as the proportion of original similarity data ignored due to the requirement of result transitivity. In conclusion, the proportion of ignored original similarity data is not significant.

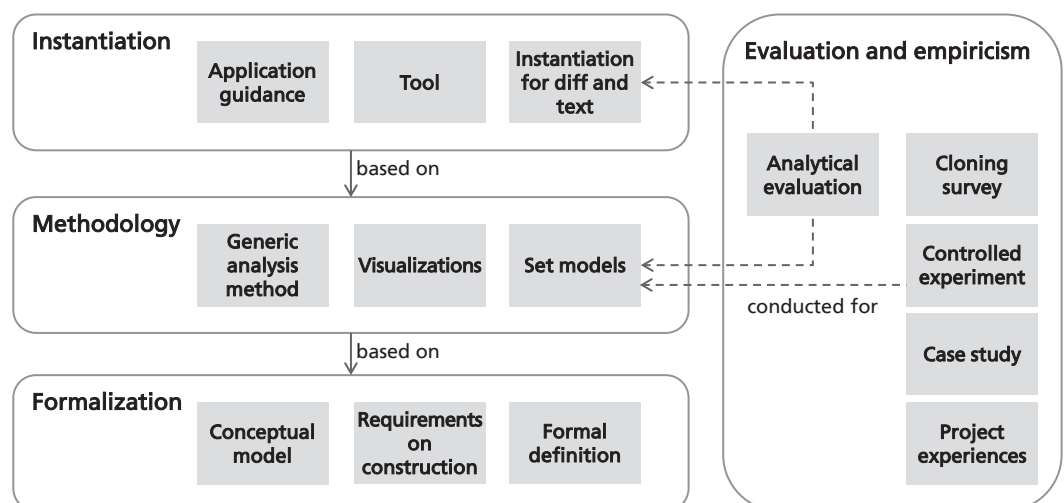


Figure 5

Thesis contributions

- **Instantiation of the similarity analysis method.** We instantiate the defined similarity analysis method for the longest common subsequence (diff) algorithm applied to the textual representation of software source code. Using a tool implementation, we perform the analytical set model evaluation described above. Finally, we provide a set of guidelines on performant and scalable set model implementation techniques, on the result interpretation, and on the practical application of our approach.
- **Empirical contributions.** By performing a survey on the cloning practices in industry, we contribute to a better understanding of the origins of the practical problems: the large-scale cloning practices, their benefits and drawbacks, and their consequences for reuse migration. Hence, we characterize the problem and provide the empirical basis for formulating related research hypotheses. Furthermore, we evaluate the benefits of the core idea of our solution, the set similarity model, in a controlled experiment. Finally, we describe and evaluate the practical application of our similarity analysis method in an industrial case study and in the consultancy projects that used the implemented analysis tool. Hence, using these three types of empirical investigations, we empirically support the complete iteration of the research cycle, as depicted in Figure 6.

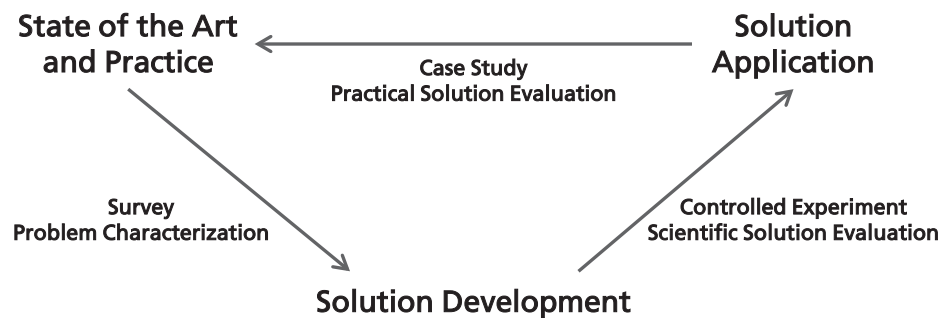


Figure 6 The empirical investigations along the research cycle

## 1.4 Outline

Figure 7 maps the chapters of this thesis and the particular contributions to the research approach structure, described in Section 1.1. To complement the chapters, we provide further information on specific topics related to our approach in the appendices of this thesis.

- The current, introduction chapter provides an overview of the scope of this thesis, the addressed problems, the followed research approach, the proposed solution ideas and the resulting contributions.
- In Chapter 2, we describe the research context of this thesis: the basic ideas of software reuse and of the product line approach, product line adoption strategies, the fundamental concepts of reverse engineering, and related approaches for similarity analysis of software variants.

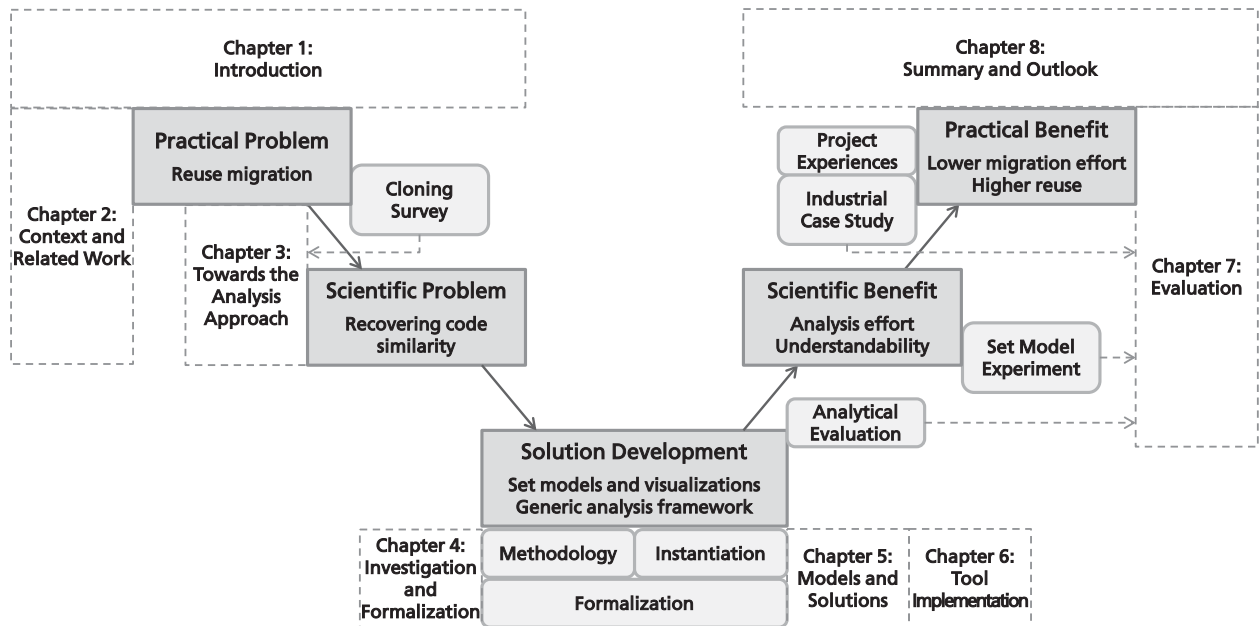


Figure 7

Thesis chapters and contributions mapped to the research approach structure

- In Chapter 3, we focus on the practical and scientific problems addressed by this thesis: we present the industrial survey of code cloning practices, define the application scenarios and analysis goals for our approach, discuss the shortcomings of existing related approaches, and derive the practical and scientific hypotheses.
- In Chapter 4, we describe the foundation of our approach: the basic properties of software variants, the conceptual model of variant similarity analysis, the requirements on the construction of variant similarity analysis techniques, and the formal definitions of a variant similarity analysis technique and of the quality of its results.
- Chapter 5 contains the core of our solution: the definition of the hierarchical set similarity models, the algorithms and activities used in their construction, the visualization concepts, and the similarity metrics.
- Chapter 6 describes techniques for the performant and scalable implementation of our approach, with the main focus on the set model.
- Chapter 7 describes the analytical evaluation of the developed approach, the evaluation of the scientific hypotheses in a controlled experiment, and practical application experiences regarding our approach, including the industrial case study.
- In Chapter 8, we conclude our thesis, summarize the contributions and limitations of our approach, and outline future work.
- Appendix A contains the full participant material used during the execution of the controlled experiment described in Chapter 7 and reports the raw data collected in the experiment.
- Appendix B discusses our application guidance for the practical use of the defined similarity analysis approach.



## 2 Context and Related Work

This thesis contributes a reverse engineering<sup>1</sup> approach for analyzing the similarity of a group of cloned software system variants. In this chapter we provide a short overview on the two main research areas constituting the background of this work:

- In Section 2.1 we introduce the basic concepts of software reuse and describe the software product lines approach, which is a systematic reuse-based approach used for development of a group of software system variants.
- In Section 2.2 we address the fundamental principles of reverse engineering, and in Section 2.3 we present the reverse engineering approaches used in the context of introducing software reuse in the development of software system variants.

In the research background description we provide definitions of the fundamental concepts of software reuse and reverse engineering, which we will refer to in the remainder of the thesis. For space reasons, we do not provide a complete presentation of the state of the art in these topics – instead, we rather concentrate on their aspects which are the nearest to the focus of this thesis. For interested readers, the provided references lead to more comprehensive literature sources.

### 2.1 Software Reuse

**Basics of reuse** The reuse of software assets has been proposed by McIlroy as early as in 1968 [McIlroy 1969], and that idea accompanied the software engineering research ever since. The basic rationale behind software reuse is that if a software asset solving a specific problem already exists, and a related problem needs to be solved, it should be easier and faster to use (and potentially adapt) the existing asset than to “reinvent the wheel” by developing the new solution from scratch. Software reuse is defined as:

**Definition 1** Software reuse

*The use of an asset in the solution of different problems [IEEE 2010].*

In particular, software reuse means that the same software asset is used during the development of many other assets or software systems. As software development creates a range of different asset types, reuse is possible for any of these types.

---

<sup>1</sup> As this thesis is related to software, it refers to software concepts unless explicitly stated otherwise. Hence, the term “reverse engineering” means “software reverse engineering”, “product lines” means “software product lines”, etc.



Definition 2      (Reuse) asset

*An item, such as design, specification, source code, documentation, test suites, manual procedures, etc., that has been designed for use in multiple contexts [IEEE 2010].*

Development  
for and with  
reuse

The above definitions reflect two important properties of software reuse. First of all, reuse involves two types of activity: the provision of assets, i.e. their development for reuse, and the subsequent development of software solving a particular problem, with reuse of the provided assets. The distinction between development for reuse and development with reuse is frequently applied in reuse approaches. Typically, development for reuse requires an investment of additional effort, as generic assets are more costly to develop than non-generic ones. Subsequently, the reuse of the generic assets is expected to provide savings that outweigh the initial investment [Barns 1991].

Reuse needs  
planning and  
management

Second, the definitions reflect the fact that in most cases reuse is not likely to occur as a matter of coincidence, but rather needs to be a result of a plan. In the development for reuse, a software asset needs to be generalized, and its interfaces carefully structured, to enable its use in more than one context. Hence, the asset needs to be explicitly developed for reuse. Furthermore, a reuse of an already existing asset also benefits from a systematic plan. Development with reuse involves finding an appropriate asset, understanding and evaluating it, and optionally adapting the asset to the target context. If planned and structured support is not available, the effort required for these steps can outweigh the reuse savings: the appropriate asset can be hard to find [Henninger 1994], understanding the asset can be difficult without appropriate documentation [Bayer 2004], and adaptation efforts can be significant despite a seemingly minor mismatch between the provided and required functionality [Thomas 1997]. Consequently, performing software reuse in an unplanned and ad-hoc way might fail to achieve the promised benefits. Hence, several systematic approaches, addressing the problems of planning, structuring, managing, and financing a reuse program, have been developed [Jacobson 1997] [Lim 1998] [Clements 2002a].

Definition 3      Systematic software reuse

*Systematic software reuse is the purposeful creation, management, support, and reuse of assets [Jacobson 1997].*

Reusing software influences not only the technical development activities, but has also implications for development processes, organization structure, and even for the way the organization offers its products on the market. The systematic software approaches address these issues, as neglecting them can ultimately lead to a failure of the reuse program [Sherif 2003]. Hence, the organizational and process issues need to be carefully considered when reuse adoption is planned. A migration of existing software assets towards a reusable form needs to unify the technical and organizational aspects, and is therefore a complex undertaking.

Granularity of reuse	In the 1960's and 1970's, reuse of source code was practiced mainly on the level of algorithms and small routines, for example performing calculations of mathematical functions. With a growing maturity of reuse approaches, the typical size of a reusable asset increased: in the 1990's, component-based software engineering advocated the reuse of components, which encapsulated semantically related functionalities of a software system and were composed to form the final software products [Jacobson 1997]. Finally, the software product line engineering approach raises the granularity of reuse to the level of complete software systems.
----------------------	--

### 2.1.1 Software Product Line Engineering

Software product line	Software product line engineering is a systematic reuse approach for the development of multiple similar software systems. The developed group of similar software systems is called a software product line.
-----------------------	---

Definition 4	<p>Software product line</p> <p><i>A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [Clements 2002a].</i></p>
--------------	---

Mass customization of software systems	The software product line engineering approach can be seen as a further specialization of component-based software engineering. In both approaches, the software systems are developed by reusing a <i>set of core assets</i> . However, product line engineering is adapted for the situation in which the developed software products exhibit a high degree of similarity as they <i>satisfy the specific needs of a particular market segment</i> . Each of the products is tailored to best fit the needs of a particular customer group in that segment, but at the same time, the high similarity makes it possible to compose a major share of product functionalities from reusable assets. Consequently, large-scale reuse enables development of a potentially large family of individualized products in a cost-effective way, a principle known in many industries as mass customization [Tseng 2007].
--	--

Management of features and assets	The high similarity of the products is actively promoted by <i>managing the set of features</i> provided by the product line: for example, development of a feature beneficial to just one product might be rejected if it compromises any quality attribute of the other products. The development is optimized for achieving the global goals of the whole product line, and the local goals of particular products can be sacrificed if necessary. The management is also performed on the technical level: the reusable assets are developed only if sufficient need for them, motivated by the product features, exists. The assets are developed and composed <i>in a prescribed way</i> , so that the asset interfaces and the architecture of the product line
-----------------------------------	--

provide a “building plan” optimized for quick product assembly. Finally, each asset is generic enough to support all products where it is reused. This is assured by managing the variability of the assets, which is the set of all characteristics which may vary from one product to another.

### Structuring product line engineering

In product line engineering, the distinction between development for reuse and development with reuse is reflected by the definition of two main development activities: family engineering, which develops the reusable assets and stores them in an asset base, and application engineering, which reuses the assets to derive end-user products (see Figure 8). The ultimate goal of both development activities is to satisfy the requirements placed on the developed software products. The requirements are managed in the scoping process in order to harmonize their alignment between products and hence maximize the benefits of reuse [Schmid 2002a]. Additionally, product line engineering establishes an explicit feedback process, where application engineering provides information on new product requirements, asset usage, and product-specific asset adaptations back to family engineering. The feedback process, frequently missing in the component-based approaches, ensures that the product line evolves consistently with the newest product requirements.

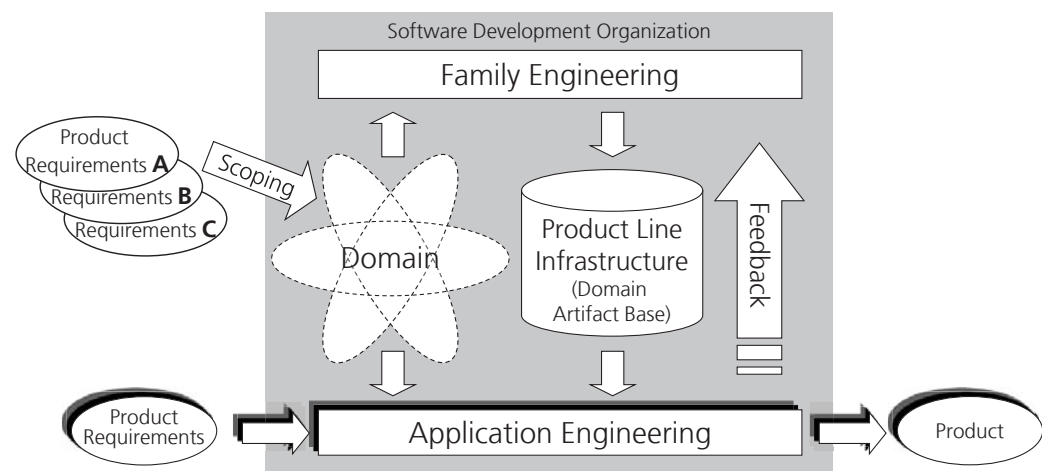


Figure 8 A schema of product line engineering (adapted from [Muthig 2002])

### Product line economics

The family engineering activity concerns the development of assets for reuse. As in any other reuse approach, the development of reusable assets requires an initial investment, which is paid back by the savings resulting from reusing the assets in application engineering. In case the complete process of establishing a product line is performed before the delivery of the first product (a proactive adoption approach, see Section 2.1.2), the initial investment ranges between one and two times the cost which would be required for developing a single software product without reuse. Typically, the investment in product line engineering is paid back after the third software system is delivered [Clements 2002a]. Hence, product line engineering is only justified and beneficial if a sufficiently large number of sufficiently similar systems should be developed [Böckle 2004] [Ganesan 2006].

Implementation approaches for similar systems Even if a group of similar systems is delivered, satisfying the market demand for individually customized solutions, these systems do not necessarily have to be developed as a product line internally. Depending on the similarity of demanded features and the technical organization of the assets, the systems can be developed using a range of techniques with varying approach to asset reuse: from a standardized platform, where just the underlying infrastructure components are reused, up to a fully configurable generic product base where a complete product is composed automatically from the reusable assets based on provided feature selection. Several classifications of these approaches exist [Bosch 2002][Riva 2003][Krueger 2004]. In particular, the similar systems can be developed in completely independent software projects, with no application of software reuse – this is considered to be the most immature approach to implementation of a group of similar software systems. In our thesis, we focus on the similarity analysis of such independent software projects in order to support creation of reusable assets, but we do not assume any specific target approach for implementation of the restructured reuse-based systems.

Implementation approaches for reusable assets Similarly, the variability supported by reusable assets can also be realized in many ways with the use of many different technologies [Anastasopoulos 2001]. Basically, the approaches for variant derivation, that is for instantiating the generic asset for the use in a specific context, can be classified into the following categories [Kästner 2010]:

- **Compositional approaches** create the content of the asset by composing it from a number of smaller content pieces such as files, classes, modules or code fragments. The composition is usually performed at the build time or during deployment. The possible implementation technologies range from simple file selection to advanced mechanisms such as aspects [Kiczales 1997] or feature-oriented programming [Apel 2013].
- **Annotative approaches** process a generic asset, which contains annotated content sufficient to derive all intended asset variants, and remove all content fragments except for those that correspond to the single selected variant. The removal is typically performed at the build time. An example of an annotative approach is the C preprocessor [ISO/IEC 2011].
- In **duplication-based approaches** the content of each asset variant is stored separately in a permanent way, and new variants are created by duplication of other already existing variant during the development time and subsequent modification of that content. The most popular duplication-based approaches are configuration management branching [Conradi 1998] and cloning [Dubinsky 2013].
- **Other approaches** include techniques not falling into the above categories, such as generators and model-driven development [Beydeda 2005] where the content of the asset is created automatically based on a higher-level specification.

In the compositional and annotative approaches, customized asset instance is typically created in an automated process which reads a correct configuration (i.e. a parameter or feature selection) as input and produces the intended asset variant content as output. Hence, the information concerning the parameters and features is explicitly maintained. Moreover, the content of an asset variant only exists after the automated process was run. In the duplication-based approaches, the variant derivation process is typically not automated, as the changes to the asset variant content, which eventually differentiate it from other variants, are added in a human-based development process. Hence, the information on possible parameters and features is not required for variant derivation and in the practice is often not explicitly documented.

2.1.2 Software Product Line Adoption Strategies

Adoption strategy classification

An organization planning the development of a product line can be situated in a variety of circumstances, and can accordingly select from a variety of product line adoption strategies. A general classification of these strategies is provided in Table 2.

		Future product prediction approach	
		Revolutionary (proactive)	Evolutionary (reactive)
Development starting point	New product line (green field)	New assets are developed to match all expected products. Known as the proactive approach. [Clements 2002a]	New assets match the current products and are evolved as further products emerge. Known as the reactive approach. [Clements 2002a]
	Using existing set of products (extractive)	New product line is developed from existing assets and matches all existing and expected products. Known as the extractive approach. [Krueger 2002]	Existing assets are adapted for reuse in existing products, and evolved as further products emerge. Known as retroactive [Staples 2004] or extractive approach.

Table 2 Two dimensions of product line adoption (based on [Bosch 2002] and [Krueger 2002])

Adoption in the green field scenario

In many cases, the product line is developed in a green field scenario, as no comparable products exist in the organization yet. Hence, the development scope of the product line and the reusable assets needs to be defined first. Depending on the market prediction possibilities and domain stability, the development of reusable assets might encompass all products foreseeable over the lifetime of the product line (revolutionary, proactive approach), or it might just include the already ordered ones and assume that products which would emerge later will be addressed by the respective adaptation of the assets (evolutionary, reactive approach).

Adoption using existing products and assets	In case one or more projects addressing the market segment of the product line already exist, these projects can be reengineered to form (a part of) the new product line. Hence, the suitable assets are extracted from the existing products and adapted for reusability [Schmid 2002b]. Again, the development might follow the revolutionary route and encompass all foreseeable products, or limit itself to the currently provided ones, assuming evolutionary adaptation to further requirements which are not yet known or certain at the moment of product line adoption.
Green field vs. migration	As product line engineering requires an initial investment into the reusable asset base, its adoption in the green field scenario requires sufficient confidence that the reuse will pay off. Hence, it is applicable in case when there is sufficient certainty that a number of similar products will be developed, and the resources necessary for its initialization are available. However, in many situations one or both of these requirements are not fulfilled. The lack of resources (especially time) or uncertainty of market development might justify the development of products with little or no reuse [Dubinsky 2013] (see also Section 3.1). Only in the longer perspective, if sufficient products proved successful and their maintenance could be optimized by reuse adoption, the organization can justify the introduction of a reuse approach and restructure the existing products accordingly. In such situation, reverse engineering can play an important role by delivering information supporting the migration decisions [Hall 1992].
Proactive vs. reactive evolution	The choice of proactive or reactive approach to product line evolution depends mainly on economical and risk management considerations. Although the proactive approach is thought to enable a higher ultimate payoff from reuse, it also requires a larger initial investment and sufficient certainty of future product development [Clements 2002b]. If the certainty is not given, or the resources available for product line adoption are significantly limited, the reactive approach is preferred [Simon 2002].

## 2.2 Reverse Engineering

Reverse engineering fundamentals	This thesis focuses on the analysis of asset similarity across a group of software variants. Such an analysis is an example of a reverse engineering task. In contrast to the regular, forward engineering process, where high-level abstractions (requirements, design) are transformed into low-level implementation of a system, reverse engineering proceeds in the opposite direction in order to gain knowledge of higher-level concepts from the lower-level implementation assets, as depicted in Figure 9.
----------------------------------	---



## Definition 5

## Reverse engineering

*Reverse engineering is the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction [Chikofsky 1990].*

Reverse engineering is a recommended practice if the knowledge and documentation of the system is not available or is insufficient, and the source code remains the only reliable representation of the software system [IEEE 1998]. The result of reverse engineering is the recovered information and knowledge concerning the analyzed software system. During this process, the analyzed assets are not altered – hence, reverse engineering is a purely analytical activity. If the knowledge gained by reverse engineering is subsequently used to plan and perform changes to the subject system, starting a forward engineering process, the resulting cycle is a reengineering process (see Figure 9), defined as:

## Definition 6

## Reengineering

*Reengineering (...) is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form. Reengineering generally includes some form of reverse engineering (to achieve a more abstract description) followed by some form of forward engineering or restructuring [Chikofsky 1990].*

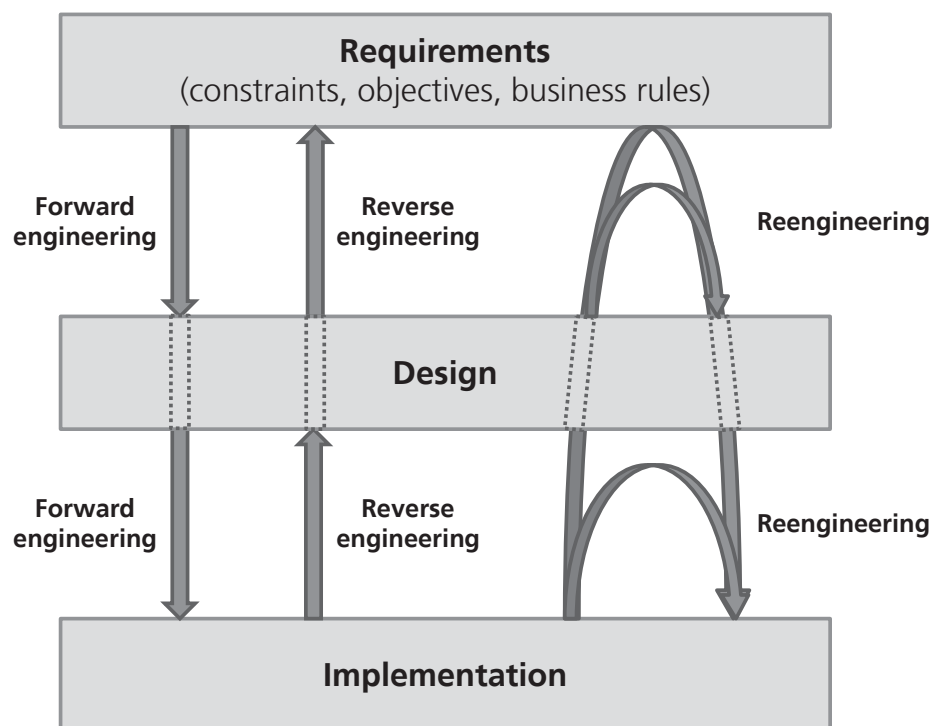


Figure 9

Forward engineering, reverse engineering, and reengineering (adapted from [Chikofsky 1990])

Data,  
information,  
knowledge

Reverse engineering aims at gaining knowledge from the existing assets. However, this process cannot be fully automated, as knowledge is a capacity of a human being. Therefore, knowledge has to be obtained by a human through analysis and interpretation of information concerning the analyzed system. The information, in turn, can be generated automatically based on the available data.

Definition 7

Data

*Data are discrete, objective facts or observations, which are unorganized and unprocessed, and do not convey any specific meaning [Awad 2004].*

Definition 8

Information

*Information is data that have been shaped into a form that is meaningful and useful to human beings [Laudon 2006, p. 13].*

*Information is an aggregation of data that makes decision making easier [Awad 2004, p. 36].*

Definition 9

Knowledge

*Knowledge is data and/or information that have been organized and processed to convey understanding, experience, accumulated learning, and expertise as they apply to a current problem or activity [Turban 2005, p. 38].*

*Knowledge is information combined with understanding and capability; it lives in the minds of people [Laudon 2006, p. 2].*

Data are raw facts collected from the subject system. Information is created by processing and structuring the data for a specific analysis purpose. Data and information are not human-dependent and can be created and processed automatically. Finally, knowledge is built by humans interpreting the available information based on their goals and experience (Figure 10).

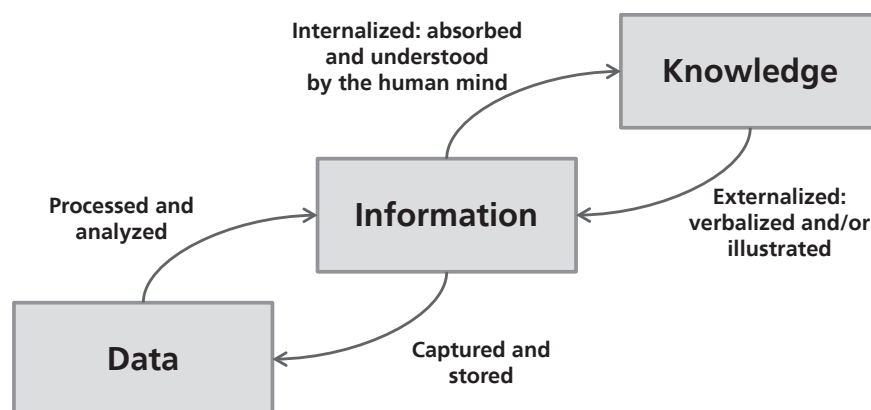


Figure 10

The relationships between data, information and knowledge (from [Liew 2007])



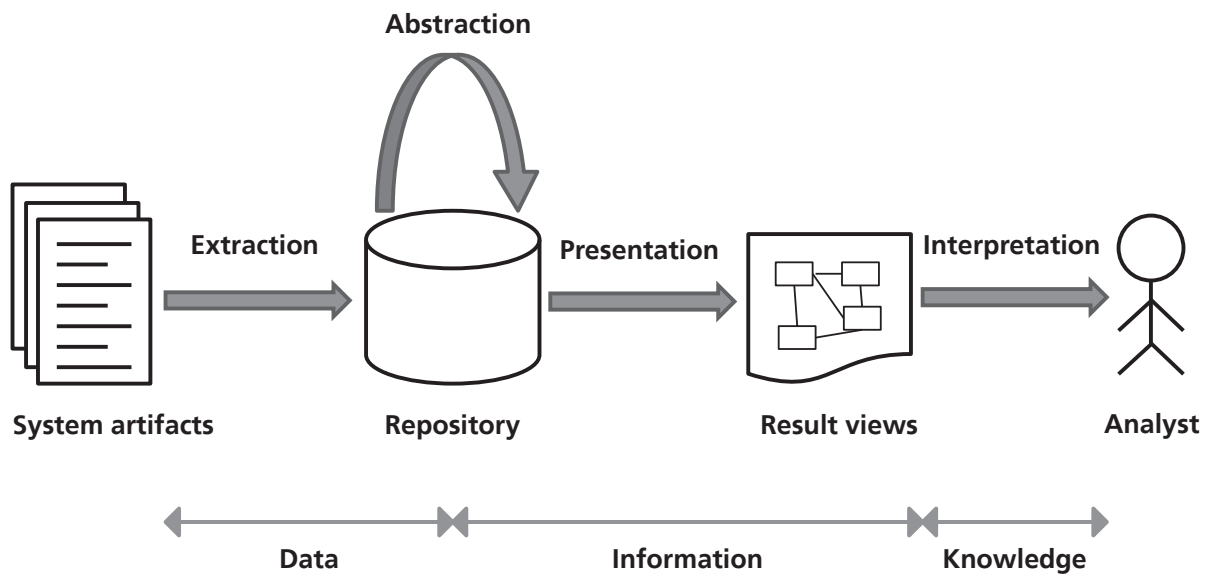


Figure 11 A generic reverse engineering process and its relation to data, information and knowledge

#### Reverse engineering phases

The hierarchy of data, information and knowledge shapes the structure of any reverse engineering process. Typically, a reverse engineering process (e.g. [Jarzabek 1998], [Mueller 2000]) consists of the following generic phases, depicted in Figure 11:

- **Extraction:** collects the raw data from system assets and stores them in a repository or model. The data can be collected using a multitude of automated techniques, such as for example parsing [Aho 2006], and also by manual inspection [Demeyer 2008].
- **Abstraction:** processes the collected data in order to derive information. For that purpose, the data can be organized (e.g. structured, filtered, aggregated), contextualized (e.g. categorized, linked) and analyzed (e.g. to derive further information). Since many different abstractions can be derived from the same data, it is necessary for an efficient analysis to define the concrete analysis goals and the intended users of the information. The abstraction phase can be performed iteratively, as the information derived in the recent abstraction step can be combined with preexisting data to create new information.
- **Presentation:** concerns the display of the created information to the human analyst in a suitable form. Different views and visualization techniques might be used to facilitate navigation and understanding of the facts and correlations obtained in the reverse engineering process [Eick 2002].
- **Interpretation:** is performed by a human analyst based on the analysis goals, available information, previous knowledge and personal experience. In the result, new knowledge on the subject system is derived. Depending on the analysis goals, the interpretation of the same provided information can lead to creation of different,

goal-specific knowledge. Therefore, the quality of the gained knowledge needs to be evaluated with respect to the analysis goal. Subsequently, the knowledge can be used for [Knodel 2011]:

- **Refinement of reverse engineering analyses**, in case the analysis results raises further questions regarding the subject system.
- **Verification of existing assumptions** with regard to the subject system, resulting in confirmation of the status quo or identification of discrepancies.
- **Synthesis with the results of other analyses**, in case the reverse engineering analysis provides only one of many possible viewpoints on the underlying problem.
- **Definition of action items**, based on the analysis goals and the problems or risks identified during the analysis.

Typically, the phases of extraction, abstraction and presentation are automated, while the interpretation phase necessarily remains a human-based task. However, the interpretation of the information provided by the reverse engineering approach might be supported by appropriate guidance, for example in the form of rules, patterns, or heuristics [Demeyer 2008].

## 2.3 Similarity Analysis Approaches for Software Variants

Focus on the variants

Reverse engineering techniques can be used to recover the information about source code similarity of the system variants, which in turn is required to support the decisions on reuse introduction. A large number of reverse engineering techniques have been developed [Canfora 2007]. However, as each technique addresses one of a variety of specific analysis goals, most of them cannot be directly applied to the variant similarity analysis problem. A large proportion of reverse engineering techniques is targeted at an analysis of only a single instance of a software asset or system [Canfora 2007], for example to redocument that system or asset [Benedusi 1992], recover its architecture [Koschke 2000], detect design violations [Murphy 2001][Knodel 2011] or find code smells [van Emden 2002]. Moreover, the techniques which do aim at analyzing a group of system instances are frequently developed with the goal of analyzing system versions, for example to detect development trends or recover information on system evolution [Kagdi 2007][D'Ambros 2008]. However, the analysis of versions is based on a number of assumptions, which are not fulfilled for system variants. Hence, in this section we provide an overview of only these reverse engineering algorithms and techniques which can be used for the analysis of variants. Subsequently, in Section 3.3 we analyze the drawbacks of these techniques in the context of a system cloning scenario. The inherent differences between software versions and variants are discussed in Section 4.1.

### 2.3.1 Comparison and Differencing Algorithms

The basic prerequisite for analyzing similarity of any group of objects is the ability to compare at least two objects and recognize the differences between them. Several algorithms for differencing various types of data structures exist:

- A popular algorithm for comparison of text files is the Longest Common Subsequence algorithm, also known as diff [Hunt 1976]. The algorithm compares two text files, treated as lists of text lines, and determines the longest list containing text lines which are identical in both files and occur in the same order. The remaining lines in both files are considered to be different. An extended variant of the algorithm can detect the differences between 3 files at once [Khanna 2007].
- Apart from text representation, source code can also be represented as an abstract syntax tree (AST). Several algorithms for differencing tree structures exist: for example, the Change Distiller algorithm [Fluri 2007] compares two ASTs, matches the corresponding nodes, and computes a minimal edit script transforming one tree into the other.
- Source code can be represented as a model, i.e. a typed graph. Multiple algorithms for finding isomorphic subgraphs in two input graphs exist. Examples of such algorithms directed at software models are JDiff [Apiwattanapong 2007] and UMLDiff [Xing 2005].

Detailed results, only little abstraction

Typically the differencing algorithms compare just two objects (files, trees, models), and provide a list containing every difference they found. Hence, they are suitable for an analysis of relatively small amount of code, where the amount of found differences is low enough to be comprehended by a human without the use of structuring, filtering, and other abstraction mechanisms. Even though the advanced frameworks using these algorithms, such as Beyond Compare [BeyondCompare 2014], do utilize an abstraction mechanism, based on the system folder hierarchy, the current form of that mechanism still provides only little abstraction. Only the existence of an unspecified difference inside the system hierarchy is indicated, and no further information about the size and nature of that difference is provided.

No comparison of a larger number of objects

The differencing algorithms do not directly address simultaneous comparisons of a larger number of objects – such a comparison would need to be performed as a series of pairwise comparisons. A few file differencing tools, e.g. Diffuse [Diffuse 2014], attempt such comparison by selecting one file and comparing every other file with it – a so-called “star comparison”. However, file pairs not involving the selected star center are not compared.

### 2.3.2 Clone Detection

Definition of a software clone Software cloning, that is duplication of software assets and their use in other context with or without modifications, has been extensively researched [Koschke 2008]. However, depending on the intended usage of the cloning information, various different notions of a software clone are used. Clone detection experts frequently have varying opinions whether a given code sample should be considered a clone or not [Kapser 2007]. Hence, any definition of a software clone which is general enough must necessarily be a vague one. We use the following definition of a software clone, attributed to Ira Baxter [Koschke 2008]:

Definition 10 Software clones

*Clones are segments of code that are similar according to some definition of similarity.*

The notion of a clone is therefore defined by referring to the concept of similarity – which again can be defined in a multitude of ways. We define and further discuss the concept of similarity in Section 4.6.2.

Cloning versus reuse It is important to distinguish asset cloning from asset reuse. Although both these activities result in the usage of an asset in more than one context, in the case of reuse the same asset is used at all locations. Even if adaptations and configurations of a local asset instance are needed, conceptually all instances of the reused asset evolve together. In contrast to that, software cloning is an activity of duplication: two or more (initially identical) copies of the asset are created, which in their further evolution are treated as different assets.

Basics of cloning Software cloning is mostly studied as a small-scale phenomenon, where an asset fragment such as a method is copied to a new location in the same or different asset. Several approaches for small-scale clone detection [Bellon 2007], removal [Rieger 1999], prevention [Lague 1997], and management [de Wit 2009] have been proposed. A typical size of a code clone ranges from a few code lines up to several hundreds of lines. The search space of clone detection algorithms is large for any non-trivial software system, as any code fragment can be potentially judged as a clone of any other code fragment. Typically, code fragments are reported as clones if the measured value of their similarity exceeds a specified threshold [Mende 2008]. The clones can be detected as pairs as well as groups of similar code fragments – such groups are called clone classes. Hence, clone detection can be used to simultaneously analyze a larger number of software systems.

Clone detection results need filtering In an analysis of a group of similar software systems, the extensive search strategy of clone detection approaches results in reporting a large amount of detailed clone data, which leads to two analysis problems. First, the reported clones might concern very diverse configurations of

similar code fragments, including clones found inside one system, clones between unrelated fragments of the various systems, and finally the clones in software assets which come from different systems and semantically relate to each other. However, only the last category is relevant to the later reuse migration activities. In an example analysis, Svajlenko et al. generated and analyzed five similar software systems containing 28 860 relevant function clone pairs. However, due to the extensive search, a clone detection tool reported, in addition to the expected clone pairs, over 2 million further clone pairs in these systems. Hence, only 1.46% of reported function clone pairs corresponded to similarities which were relevant from the reuse migration point of view [Svajlenko 2013]. Although these results were technically correct from the clone detection point of view, they need to be further filtered for their use in a reuse migration.

Clone detection results need abstraction	A second analysis problem is caused by the fact that clone detection results are represented as a list of code locations where the similar code fragments were found. However, these detailed results are not suitable for a human to directly estimate the degree of total similarity between particular assets or the whole systems. Hence, abstracting the results by calculating similarities of larger code structures is necessary.
Abstraction and filtering techniques	The clone detection results can be abstracted by using the system hierarchy structure (e.g. code files and folders) and calculating clone coverage metrics, i.e. the proportion of total code of a given system part (folder, file) covered by the detected clones. Furthermore, for determining the similarity of two selected system parts the clone detection results can be filtered – only the clones occurring between two selected system parts need to be considered, that is, the clones where all similar fragments are found inside only one system part can be discarded. A system structure browser, based on these abstraction and filtering techniques, can display several coverage metrics for clones found between the selected system part and either the remainder of the system or any other system part [Kapser 2006] [Jiang 2007]. The browsing of clones in the system structure can also be facilitated by interactive visualizations and user-specified filters [Zhang 2008] [Asaduzzaman 2011].
Multi-system similarity analyses	The described abstraction and filtering techniques are also used in similarity analyzes for a group of software systems. Yoshimura et al. used clone coverage metrics to assess similarity of two software systems [Yoshimura 2006] – here, the detected clones were filtered by only considering these clones which had a counterpart in the other system. In an analysis of a larger group of systems, Yamamoto et al. and Mende et al. computed the similarity metrics for each pair of the systems and presented them as a square matrix (see Figure 12 left) [Yamamoto 2005] [Mende 2008]. Hemel et al. used clone detection to perform a “star comparison” of a group of system branches and estimate their deviation

from the main development branch [Hemel 2012]. For visualization of similarities between many systems, Kamiya et al. propose using scatterplots (see Figure 12 right) [Kamiya 2002]. Cordy extends that idea by proposing live scatterplots, which can aggregate several rows or columns in the scatterplot based on the system hierarchy, and on user demand provide detailed data for each scatterplot cell [Cordy 2011].

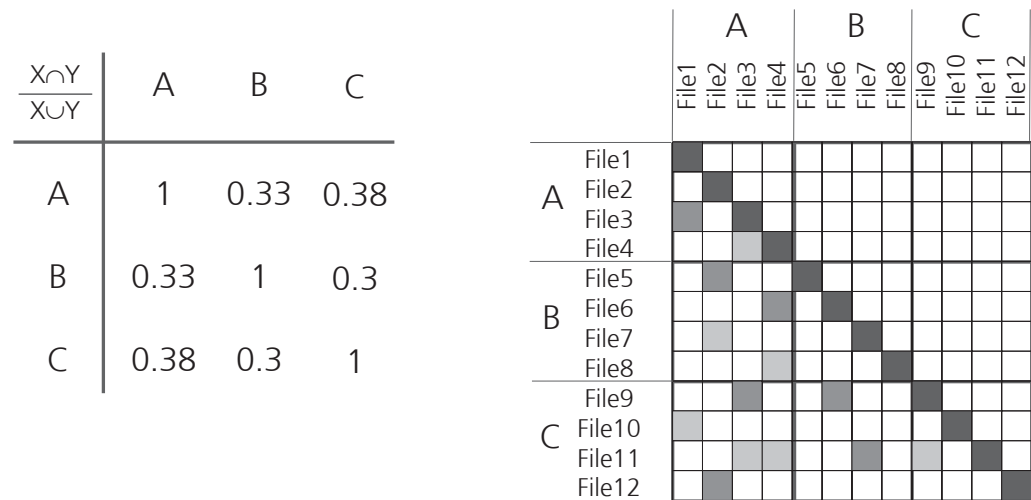


Figure 12 Example presentation of multi-system similarity analysis results in the form of pairwise similarity matrix (left; the values indicate the degree of similarity) and a multi-system scatterplot (right; the similarity is indicated by the cell color)

### 2.3.3 Other Approaches

Several other techniques related to assessing similarity of software system or asset variants, based on reverse engineering, have been proposed. However, as they have a different analysis focus than our technique, i.e. they do not directly aim at assessing the similarity of source code, we only categorize these approaches and provide example references:

- Techniques for reverse engineering of product line architecture recover the architectures of the particular similar systems and then match the architectural elements between the systems in order to identify common and variable architectural components [Kang 2005] [Koschke 2009] [Wu 2011]. However, their focus remains at the abstraction level of architectural components.
- Several techniques for detecting features in the source code have been proposed. Features are units of program functionality, and the goal of feature location techniques is to establish a correspondence between the features and the source code locations responsible for their implementation [Kästner 2014]. These techniques can also be applied to a group of cloned software systems in order to detect optional features, i.e. functionalities supported by only a subset of the systems [Rubin 2012].

- Several approaches for recovery and evaluation of potentially reusable assets exist [Bayer 1999] [Knodel 2005] [Kolb 2006b]. The basic idea of these approaches is to select an asset from a single product and identify the functionalities which are missing for achieving full reusability of the asset across the product line.
- There are two approaches which propose expressing and measuring the similarity of software system variants by using a set model:
  - Peterson discusses sets constructed from requirements which are posed on products in a product line [Peterson 2004]. These sets typically intersect as some requirements are relevant to several products. He measures the reuse potential of the product line based on the degree to which the requirement sets overlap. However, the requirements are specified manually and no reverse engineering is involved.
  - Berger et al. identify components in Simulink models which have similar interface descriptions, mark them as variants, and model the variant products as intersecting component sets [Berger 2010]. However, only interfaces of components are considered, and components are identified as variants if the interface similarity exceeds a specified threshold. Hence, the identified component variants can still significantly differ in the implementation details. Also, there is no support for abstracting larger models (e.g. in the form of model structure hierarchy), nor there are any visualization concepts defined.

## 2.4 Summary

In this chapter we presented the context of our work: we discussed the fundamental concepts of software reuse, including software product line engineering (Section 2.1), and outlined the basics of reverse engineering (Section 2.2). Subsequently, we presented the related approaches analyzing similarity of software variants (Section 2.3).

We frequently refer to the fundamental concepts presented here in the further chapters of this thesis. Moreover, in Section 3.3 we discuss the deficiencies of presented existing analysis approaches in the context of analysis goals related to reuse migration, and we motivate the need for developing a solution overcoming these deficiencies.



### 3 Towards an Approach for Variant Similarity Analysis

In the Introduction we presented the existing literature reports which document the existence of cloned software system variants, the reasons for their creation, and the resulting maintenance challenges. As most of these reports are punctual observations made in just one organization, together with a group of other authors we contributed a broader perspective by performing an exploratory survey, in which we investigated the cloning practices in industrial software product lines [Dubinsky 2013]. In Section 3.1 we describe a subset of the observations we made in that survey, concerning the benefits and drawbacks of cloning and the role of code similarity information, and derive respective conclusions.

The results of the industrial survey motivate the practical need for delivering the necessary similarity information, for example by performing source code analysis. Hence, in Section 3.2 we define three application scenarios where code similarity analysis is applicable and derive from them the specific analysis goals which are addressed by our approach. Subsequently, in Section 3.3 we discuss the shortcomings of existing similarity analysis approaches which prevent them from fully addressing the information needs of a reuse migration. Finally, the expected benefits of using our approach in the defined application scenarios are formulated as hypotheses in Section 3.4.

#### 3.1 Cloning in Industrial Software Product Lines – An Exploratory Survey

Singular reports on cloned systems	The practice of cloning complete software systems in order to provide variants of their functionality was reported by many researchers [DeBaud 1998] [Schmid 2002b] [Faust 2003] [Riva 2003] [Staples 2004] [Yoshimura 2006] [Jepsen 2007] [Koschke 2009] [Duszynski 2011a]. Usually, these systems were cloned by either directly copying the source code, or by creating separate branches in a configuration management repository. Regardless of the technical mechanism however, the duplication of the similar code ultimately resulted in increased maintenance effort and a need to consolidate the variant systems into a reusable code base.
Exploratory survey goals	Despite the literature reports on maintenance problems, system cloning is still a frequently used mechanism for implementation of system variants. However, until now no systematic study has been conducted to investigate the reasons of that contradiction. Hence, together with a



group of other researchers we performed a survey of industrial organizations which clone large artifacts or complete systems to develop new system variants [Dubinsky 2013]. The survey had an exploratory, theory-building nature – we did not search to confirm or refute any specific hypothesis. Instead, we aimed at characterizing the context of the clone-based system development, the rationale behind cloning, and its perceived positive and negative consequences. For space reasons, in this section we only concentrate on the study findings directly related to the context of this thesis. We omit other aspects investigated in the survey, e.g. these related to organizational roles and processes, as well as some details of the survey setup, as these aspects are described in the conference paper [Dubinsky 2013].

Surveyed participants and system groups

In the survey we interviewed eleven participants, involved in developing six groups of similar system variants realized with the use of cloning. The participants were employed in three different software-developing organizations, each belonging to a different industry: aerospace and defense, data storage management, and automotive. The selection of surveyed organizations and cloned system groups was limited to those we had access to – we did not perform any further filtering of the surveyed systems. Most of the participants were fulfilling senior technical roles in the software development process: five of them described their role as “software leader/technical leader”, three as “architect”, two as “developer” and one as “integrator and QA engineer”. Among the surveyed cloned system groups, the oldest was initially developed about 10 years before the time of our study, while the youngest emerged one and a half years before the study. However, all of the system groups are composed of products which are still actively offered on the market. The teams responsible for the development of system groups numbered between 26 and 100 people.

Data collection and analysis

We collected the survey data using a questionnaire, followed by a structured interview with predefined open-ended questions. The questionnaire contained questions about the general setting of the system group and the extent of the cloning practices. In the interview, we first asked the interviewees to describe the system group and used processes and tools in more detail. Then, we investigated the way the cloned systems were created and maintained: e.g. who decides to create a clone, which reasons are used to motivate a cloning case, and how the information about existing clones is subsequently maintained. We also asked the participants about their perception of advantages and disadvantages of cloning in their specific situation. Finally, we analyzed the findings using the grounded theory approach [Corbin 2008] to detect and describe repeating concepts and link them to the collected evidence. In the analysis, we only used the questionnaire and interview data – we were given no access to the source code of surveyed system groups, and hence we could not perform own measurements of the extent and nature of the existing clones.

### 3.1.1 Survey Results

Short-term  
efficiency

In the opinion of the survey participants **cloning saves time and reduces cost of the initial development** of a new system variant. By cloning an existing solution, a first code base version already supporting many of the required features can be quickly created. Moreover, the original code, which is cloned to obtain the new variant, is already **trusted and validated** and can hence be assumed to have sufficient quality. Note that the same two reasons, i.e. development speed and code quality, are also provided in the case of proper software reuse, and constitute the main reuse benefits. In fact, some study participants considered cloning to be a form of reuse and wished to increase the amount and scope of the artifacts they cloned.

*"It is easier to start with something. Cloning gives [us] an initial basis."*

*"It saves time. These components were already used, tested, closed. A kind of an off-the-shelf software."*

*"We did something. It is 'old' and for most cases it is stable. The amount of time to bring [new code] to the required level of quality is not easily estimated."*

*"We clone code and should do better with cloning requirements and design."*

Advantages of  
cloning  
compared to  
reuse

In contrast to reuse, **cloning has a low entrance barrier**, as no special skills or development methodologies are required. Moreover, the development of the cloned variant is initially easier as in the case of reuse: since no assets are shared with other existing systems, the **dependencies to these systems do not need to be considered**. Hence, there is no need to inform the other projects about code changes, to refine the common code, or to consider the lifecycles, schedules and development goals of the other systems. One participant also indicated that as the cloned code is only used in the context of one system, it can be **more readable and understandable** than a highly generic reusable code. Consequently, his team decided to introduce file-level clones with the purpose of improving code readability and maintainability.

*"It gives freedom to change, [when cloning] there is no damage to existing products."*

*"[In the past,] a new variant (...) was integrated back into the mainstream by using preprocessor switches. This has made the code very unreadable, so we wanted to go away from that and we started to branch off the files that differ among variants."*

Rationale for  
cloning

In the opinion of survey participants, the low initial effort makes cloning to a suitable development approach when there is a strong pressure to **deliver the new software system variant quickly**. Similarly, cloning might be chosen if the **additional resources necessary to set up the reuse infrastructure are not available**. The use of cloning can be

therefore justified or even imposed by the circumstances of the development project, or might represent a “lesser evil” compared to the consequences of delayed project completion. Moreover, in some cases the **knowledge of the number and required functionality of the demanded system variants is not available upfront**, but rather emerges progressively as time passes. Hence, the scope of the system group might only look suitable for reuse in the hindsight.

*“When a new customer came, we needed to decide how to implement his requirements in the fastest way. We do not have time to think thoroughly about generic approaches.”*

*“Maybe we can [think about reuse] from the beginning. Still this is easy to say now, when we know that the first product is a success. At the beginning, the other risks are more important.”*

*“At the beginning we did not know that we will have to support all the controllers that we support now – this emerged over time.”*

#### Short-term thinking

In the surveyed projects, cloning can also occur in an unplanned and unorganized way, as a consequence of **short-term thinking** and **unawareness of reuse-based development approaches**. In some organizations, the lack of resources for setting up a systematic reuse approach resulted not from the tight deadlines, but rather from the **missing organizational focus on reuse**. As these organizations provided no incentives or supporting structures for recognizing reuse opportunities, and no funding scheme covering the initial costs of reuse was available, the particular variant projects used cloning to optimize the development costs in the short term.

*“There is a lack in resources for an organized work and methodology with respect to the product line engineering.”*

*“There is no place or procedure that asks to search for existing assets.”*

*“No one [is responsible for reuse]. One who requires an asset, takes it.”*

#### Maintenance problems

The existence of many cloned systems and assets led to a longer time to additional work and significant maintenance problems in the surveyed projects. **Repetitive maintenance tasks**, for example propagating a bug fix or a requirement change, need to be performed on each cloned copy. Moreover, each task duplicate still requires a careful analysis, as each clone has been modified for a specific context. Finally, the degree of change required to make the cloned code compliant with new variant requirements, and hence **the adaptation effort, is sometimes much higher than initially estimated**.

*“We need to perform many activities several times: for each variant, we have to check the code and implement the change or fix. Then, the design and documentation documents, as well as the test specification need to be adapted for each variant. Tests need to be run.”*

*"If we find a bug then many times it can be here and also in other places. The new product contains code that exists also in the old product. So, if we fix the old one then we also fix the new or vice versa."*

*"It is usually not possible to port without making changes to the code."*

*"It is a copy and a lot of adaptation."*

Lack of governance and of cloning information

A common characteristic of the surveyed system groups was the lack of sufficient reuse-oriented development governance. On the technical level, this manifested as the **lack of management of reuse-related and cloning-related information**. First, the information about **cloned artifact origin was not tracked** or stored, and instead existed mainly in team member's minds. Hence, the provenance of particular asset clones might be forgotten due to passing time and staff turnover, and the later bug fixes and feature changes could fail to address all relevant asset clones. Second, **the changes applied to particular clones were not tracked** or managed. As a consequence, assessing the similarity of two clones or judging their suitability as a basis for a new variant implementation was difficult without detailed code analysis. And third, **no measurements related to reuse or cloning were performed**. Hence, the organizations were not able to reliably assess the reuse opportunities, nor were they objectively informed about the technical benefits and drawbacks of their cloning choices.

*"No one is in charge of the cloning knowledge – in practice, it is the one who implements [a functionality] and the architect who is in charge of the work item."*

*"(...) code that we cloned loses connection with the product which it is cloned from, and then there is no sharing of new insights and innovations."*

*"Sometimes, we find the same bug again in a different variant that nobody thought about before."*

### 3.1.2 Discussion

Cloning as a development strategy

The results of the survey indicate that the use of cloning, even at the level of complete software systems, might be a justified development strategy. Some of the surveyed software systems were cloned because the pressure to deliver new system variants quickly at a low cost was stronger than the incentives to optimize the development for longer-term goals such as maintainability and reusability. As the tradeoff between speed and reusability occurs frequently [Kolb 2010], many companies might be tempted to clone and, after some time, consolidate only these assets and products which proved to be successful on the market while abandoning the rest (the grow-and-prune approach [Faust 2003]). Cloning can for a certain period of time be beneficial, providing development speed and flexibility [Riva 2003], or at least constitute a "lesser evil" compared to other early development risks. Hence, cloning has to be considered as one

of possible and justified strategies for development of multiple similar software systems – even if it sometimes also results from the unawareness of other, reuse-based development approaches. Interestingly, similar conclusions were also formulated by Kapser and Godfrey with regard to small-scale code clones [Kapser 2008].

The need for information management and recovery

However, although cloning might sometimes be considered to be “good”, and it will likely be further practiced, the consolidation of cloned systems should be performed early enough to prevent the long-term maintenance problems resulting from code duplication. Moreover, the collected results suggest that the maintenance problems occurring to cloned systems are intensified because the reuse-related and cloning-related information tends to be lost in the evolution process. Similarly to the maintenance tasks, the consolidation of cloned systems also requires a modification of their code, but on a larger scale. Hence, both the maintenance and the consolidation of cloned systems would benefit from the existence of current and accurate cloning information. Therefore, the cloning organizations need to be supported by approaches and tools for management (e.g. using documentation) and recovery (e.g. using code analysis) of the code similarity information. Also, methodical approaches utilizing that information for both clone consolidation and cloned code maintenance are needed.

Threats to validity

The surveyed system groups and their developers were selected because of their availability to the survey authors. Hence, the survey results and the derived conclusions need to be interpreted in due consideration of a range of validity threats, especially of external nature (i.e., with regard to the result generalizability). First, the number of surveyed subjects is limited, and they can potentially be not representative of the general software industry. Hence, the estimation of prevalence or significance of the identified facts is not possible. Second, our only data sources were the subjective answers provided in the questionnaires and the interviews. We were not able to cross-check these answers by measuring the artifacts belonging to the surveyed system groups, which leaves a possibility that some of the answers could be inaccurate. To conclude, the survey results can be mainly treated as a data point in the investigation of industrial cloning practices, but not as their complete picture.

### **3.2 Application Scenarios and Analysis Goals for Code Similarity Analysis**

Scenario background

In the previous section we described the practical situations where software products are cloned and separately maintained, and discussed the resulting long-term maintenance problems. Furthermore, we motivated the need for performing code-level similarity analysis on the created software product variants. To complete the description of the practical context of the variant similarity analysis problem, in this section we list



three concrete application scenarios for which the variant similarity analysis technique provided in this thesis is intended, and derive from them the goals that the analysis technique should fulfill. We define the application scenarios on the basis of the industrial survey, a review of the related literature, and our experience in industrial technology transfer projects at Fraunhofer IESE (see Section 7.4). The application scenarios are:

**[AS1] Reuse potential assessment:** A group of software system or software asset variants, maintained in parallel, is analyzed in order to assess whether introduction of a systematic reuse approach is appropriate for the analyzed variants or a subset of them. For that goal, the parts of the analyzed software assets suitable for transformation into a reusable form should be identified. The information delivered by similarity analysis is used to guide the selection of system variants and their constituent software assets for performing the transformation activities, and enables discussion on implementation alternatives (e.g. use the asset as is, modify it, or write a new version from scratch) [Yoshimura 2006] [Koschke 2009] [Duszynski 2011a].

From the economical and risk management perspective, the migration of existing system variants towards reuse frequently constitutes a better choice compared to the development of the target reusable systems from scratch [Simon 2002]. However, in some cases the organization might decide to start a completely new development of the reusable system variants and replace the old cloned products. The reason for such a decision might be, for example, a wish to abandon outdated implementation technologies or an insufficient general code quality of the old product implementation [Wleklik 2011]. In this situation, a similarity analysis performed on the old products still delivers important information for the new implementation planning, as the similarities between the new products, and hence their reuse potential, will likely be analogous to the similarity found in the old analyzed products covering the same markets and functionalities.

**[AS2] Consolidation of existing reusable software:** Even if a software system uses a structured reuse approach, e.g. the software product line approach, new functionalities of the particular system variants can still emerge in various ways, including cloning [Staples 2004] [Mende 2008] [Schulze 2013]. Therefore, the software system variants are periodically analyzed in order to check whether new candidates for reusable assets emerged after some evolution period. If such candidates are identified, a merge can be performed to reconsolidate the reusable implementation and assure achieving a high reuse rate. This scenario is also known as the grow-and-prune approach [Faust 2003]: the explicitly allowed uncontrolled growth of the software allows for quickly satisfying customer demands, as discussed in Section 3.1, while the later pruning phase consolidates the newly implemented assets and creates a generic solution, counteracting the long-term maintenance problems.

**[AS3] Support for parallel variant maintenance:** An organization developing cloned variants in parallel might decide to not introduce software reuse despite favorable similarity analysis results [Rubin 2013]. This can happen for a number of valid reasons, such as e.g. a high cost of an already performed safety certification for the products which would need to be repeated after reuse migration. However, the organization can still regularly analyze the system variants for code similarity in order to use the derived information for reduction of maintenance effort. For example, similarity information is useful to identify whether a specific code change (e.g. a bug fix) is relevant to other system variants. Also, it helps reduce code inspection effort by avoiding assessing the same code again in another variant. Finally, the similarity information can be used at the planning and management level, for example for verifying assumptions regarding similarity distribution or for detecting development trends such as a growing dissimilarity of a specific asset in a specific variant.

Requirements  
resulting from  
the scenarios

Although the described application scenarios target different practical situations, they share a number of common characteristics:

- In each scenario, the analysis users are interested in the similarity of software assets *between* the analyzed system variants, while the similarity *inside* the particular variants (e.g. code clones) is not relevant.
- In each scenario, the information retrieved by the analysis concerns software assets of different sizes: starting from the small scale, e.g. single methods, up to whole potentially large software systems. Because of that, providing both code-level similarity details as well as suitable abstractions for similarity of large asset structures is an important requirement for a similarity analysis technique satisfying these scenarios.
- In most cases a fairly high similarity among the analyzed software system variants can be expected – otherwise, the intention of the development organization to capitalize on the similarity existing in the software assets, and the resulting wish to perform the similarity analysis, would not emerge in the first place. The expectation of a relatively high similarity is particularly reasonable if the analyzed variants were developed in a cloning process.
- The human effort for performing the analysis and interpreting its results should not be overly high. As all the described application scenarios occur in the context of development effort reduction measures, this strived for effort reduction should not be canceled out by an effort-costly analysis process, as this would nullify the analysis purpose.

Analysis goals Moreover, it is important to note that while the application scenarios differ in their intended use of the similarity information, they share common requirements regarding the form and scope of that information. Hence, common analysis goals with regard to the retrieved similarity information can be derived from each application scenario. These goals, specifying the variant similarity analysis problem addressed by this thesis, are:

- Identify software assets of any size, belonging to the analyzed software system variants, which exhibit similarity across some or all of the variants.
- Characterize the found similarity with regard to the properties that enable well-founded decisions on further activities concerning the similar assets (e.g. their transformation into a reusable form). These properties are:
  - the degree to which the assets are similar,
  - the variants where assets similar to a given one are found,
  - the distribution of the similarity in the system or in a particular software asset,
  - the deviations in the found similarity, describing which asset elements are dissimilar, how they are dissimilar and where the dissimilarities are located.

The commonality of the analysis goals, shared by the described application scenarios, justifies the possibility to apply the same reverse analysis technique for retrieving similarity information in all these different practical cases. Although the interpretation of the retrieved information certainly differs for each application scenario, the requirements on the form and scope of the input information are essentially the same.

### 3.3 Shortcomings of the Existing Approaches

None of the existing reverse engineering techniques for analysis of variant similarity (see Section 2.3) can fully address the requirements resulting from the application scenarios described above. In particular, every of the existing approaches exhibits one or more of the following shortcomings:

- **No abstraction mechanism for large systems.** If a group of large software systems is analyzed, suitable abstractions of the similarity information are required. However, the comparison and differencing algorithms, as well as some of the clone detection approaches, only provide a detailed list of low-level analysis results [Hunt 1976] [Roy 2009a]. Even the advanced differencing frameworks merely indicate an existence of an unspecified difference inside a structure hierarchy, without providing any further information about the size and nature of that difference [BeyondCompare 2014]. Hence, deriving any statement on the similarity of large code structures is not directly possible.



- **No detailed code information available.** Some approaches, especially these providing similarity metrics aggregated on the level of whole systems [Yamamoto 2005], exhibit a deficiency opposite to the one described above. Although the aggregated metrics provide instant similarity information even for large systems, the low-level information on the particular similar code locations which contributed to the calculated metrics values are not available. Hence, this information is not sufficient to identify concrete reengineering tasks for a reuse migration on the code level.
- **Imprecise similarity information.** Clone detection techniques frequently use a similarity threshold to decide whether two code fragments are similar enough to be considered as clones [Mende 2008]. On a similarity scale ranging from 0 (no similarity) to 1 (identity), frequently a threshold value of 0.7 is selected. The use of a similarity threshold is helpful for finding cloned code fragments despite their subsequent modification. However, in the context of reuse potential assessment it introduces a significant imprecision in the analysis results. First, a code asset which is in 50% covered by clones of other asset code, assuming the 0.7 threshold value, can actually contain any proportion of similar code between 35% and 50%. Second, the code assets assessed as clones cannot be merged into a reusable form without a manual code review, as their residual differences can be large and meaningful enough to make them unsuitable for reuse migration. A similar problem occurs also for model-based code similarity analyses, e.g. using UML models, as two code fragments having an identical model can still substantially differ on the code level, e.g. due to the peculiarities of different hardware platforms.
- **No abstraction mechanism for a large number of asset variants.** Most of the existing approaches deliver similarity information calculated for each pair of asset variants [Yamamoto 2005] [Mende 2008]. However, for  $n$  variants there exist  $n(n-1)/2$  different variant pairs. Hence, the similarity of  $n$  asset variants is reported as  $n(n-1)/2$  partial results, which still need to be aggregated together in order to understand the complete similarity distribution. Already for 10 variants, 45 partial results are calculated. The lack of suitable abstraction makes it difficult to analyze such complex similarity analysis result.
- **No information on all variant combinations available.** This shortcoming is related to the previous one, as it is a consequence of the pairwise similarity result presentation. Since only the information about the pairs of compared variants is provided, the analysis questions concerning larger groups of variants (three and more) cannot be answered without further result processing. For example, a simple question such as "what is the amount of code which is identical across all the variants" cannot be answered based on the pairwise similarity only, as each pair of variants can potentially share a different selection of similar code fragments.

Shortcomings prevent the fulfillment of analysis goals

As discussed in the previous section, a similarity analysis technique should facilitate a quick understanding of the provided results. The result understanding should be supported for any level of abstraction in two scalability dimensions: the asset size dimension, ranging from small code fragments up to large systems, and the number of variants dimension, ranging from two up to several tens of asset variants. At the same time, the delivered information should be precise and accurate in both dimensions, regardless of the asset size and the amount of its variants. In Table 3, we map these requirements to the existing analysis approaches presented in Section 2.3. Due to the discussed shortcomings, none of the existing approaches fulfills all the requirements. Moreover, the last two requirements, particularly relevant for a large number of analyzed variants, are not adequately addressed by any of the approaches. Hence, the objective of this thesis is to provide a similarity analysis approach which does not exhibit the listed deficiencies and fulfills all requirements.

We further refer to the above discussion in Section 4.2, where we define construction requirements for techniques analyzing variant similarity. As the review of related approach shortcomings contributed to the requirement definition, we provide there a deeper discussion of some of the listed problems, in particular these related to pairwise analysis result presentation.

<b>Requirements</b> <b>State of the art approaches</b>	<b>Abstraction for large systems</b>	<b>Detailed code-level information</b>	<b>Precise similarity information</b>	<b>Abstraction for a large number of variants</b>	<b>Information on all variant combinations</b>
Comparison and differencing algorithms [Hunt 1976] [Fluri 2007] [Xing 2005] [Diffuse 2014]	–	+ / – (model based)	+ / – (model based)	–	–
Advanced differencing frameworks [BeyondCompare 2014]	(+) qualitative only: amount of change not visible	+	+	–	–
“Bare” clone detection [Koschke 2008] [Roy 2009a]	–	+	–	(+) local only: clone classes	(+) local only: clone classes
Clone detection with hierarchical abstraction [Kapsner 2006] [Jiang 2007]	+	+	–	(+) local only: clone classes	(+) local only: clone classes
Clone coverage metrics [Yoshimura 2006] [Mende 2008] [Hemel 2012]	+	+	–	–	–
System similarity metrics [Yamamoto 2005]	+	–	+	–	–
Cross-system scatterplots [Kamiya 2002] [Cordy 2011]	+	+	–	–	–

Table 3

The properties of the existing approaches: + stands for “supported”, (+) for “partially supported”, “–” for “not supported”

### 3.4 Research Hypotheses

**Hypotheses context** The main goal of our approach is to counteract the identified practical problems discussed in the thesis introduction: the migration of a group of cloned system variants towards reuse is effort-intensive and is likely to miss some of the reuse opportunities, while their continued parallel maintenance, without the migration, also requires a high effort due to the many repetitive tasks. Furthermore, in previous sections we discussed that the information on code similarity is frequently missing or insufficient in the organizations developing cloned system variants, which is detrimental to cloned variant maintenance and migration. Hence, our approach concentrates on providing the information on code similarity in order to support the migration and maintenance activities. In that context, we defined three application scenarios where code similarity analysis can contribute (Section 3.2).

**Practical hypotheses** Based on these discussions, we postulate three practical hypotheses concerning the role of code similarity information in the migration and maintenance of cloned system variants (Table 4). In the following, we discuss our understanding of these hypotheses.

Hypothesis name	Hypothesis text	Postulated improvement measure
<b>HP1:</b> Migration Effort Reduction	Availability of detailed code similarity information reduces the effort for migration to reuse.	20% less migration effort
<b>HP2:</b> Higher Degree of Reuse	Availability of detailed code similarity information allows for achieving a higher degree of reuse in the migration.	80% less missed reuse opportunities
<b>HP3:</b> Effort Reduction in Parallel Variant Maintenance	Availability of detailed code similarity information reduces the maintenance effort for variants developed in parallel.	10% less maintenance effort

Table 4 The practical hypotheses

**Detailed similarity information** The practical hypotheses formulate the postulated improvements in the migration and maintenance of cloned system variants, achieved by providing the detailed code similarity information, as compared to the situation where that information is not (or not sufficiently) available. As discussed in the Section 1.2 and 3.2, the code similarity information should be available on any level of abstraction (from small code chunks up to whole systems), be available for any subgroup of the analyzed system variant family, and should be sufficient to dependably support the developers in the migration decisions (e.g. these listed in Section 1.2). Certainly, such information could in the practice encompass many interesting categories: the syntactic similarity of code, the semantic similarity of the program behavior, or even the similarity of created runtime structures such as call graphs, function pointer hierarchies, and data structures. Moreover, the usefulness of each such category could

strongly vary depending on the context: the system domain, code-level implementation mechanisms, and even the used programming language. It remains an open research question to determine which kind of information provides the best support for the defined application scenarios and what kind of context dependencies for that information exist. In this thesis, we narrow our focus on the detailed code similarity information to the syntactic similarity.

#### Postulated practical improvements

Consequently, we formulate the hypothesized improvements, which can be achieved by providing the code similarity information, with relation to the syntactic similarity only. The postulated practical improvement measures, given in Table 4, indicate our intuition regarding the type and significance of the improvement provided by our approach as compared to the current state of the art (Section 2.3). Still, it needs to be recognized that a reuse migration is a complex undertaking, involving many activities and influenced by many context factors. The migration effort can be divided into multiple constituent parts, for example such as:

- the effort for planning the migration,
- the effort for performing the analyses supporting the planning,
- the effort for setting up the reuse infrastructure and implementing the necessary changes in the assets,
- unnecessary additional effort spent due to incorrect reuse decisions, e.g. introducing reuse where it does not provide benefits,
- other engineering effort (testing, verification, training),
- management effort.

While for some of the listed categories no meaningful influence of a code similarity analysis can be expected, we hypothesize that the effort for supportive analyses and the unnecessary effort due to incorrect reuse decisions can indeed be significantly reduced. Hence, in the hypothesis HP1 (Migration Effort Reduction), we postulate that a migration effort reduction by 20% can be achieved by using our approach. The possible reduction is, however, heavily depended on the proportion of the different effort categories and possibly on several other context factors. Therefore we postulate that the 20% improvement should be achievable in most cases, but can possibly fail to materialize in a particularly unfavorable context. Naturally, the postulated improvement relates to a situation when the availability of the similarity information is provided, but all other factors influencing the improvement measure remain constant. The same conditions apply likewise to the other hypothesized improvements.

#### Scientific hypotheses

The scientific problems addressed by this thesis are the recovery and structuring of code similarity information, as well as the subsequent presentation of that information to enable efficient and correct similarity information understanding by human analysts (Section 1.2). Hence, we hypothesize that the Variant Analysis approach provides an improvement

in these areas. Naturally, the proposed similarity analysis approach should also provide technically correct results and be suitable for practical use in the context of defined application scenarios. Our scientific hypotheses, listed in Table 5, reflect these conditions:

Hypothesis name	Hypothesis text	Postulated improvement measure
<b>HS1:</b> Correctness	The Variant Analysis approach provides correct results.	Precision > 0.99 Recall > 0.99
<b>HS2:</b> Analysis Effort Reduction	The Variant Analysis approach reduces the effort for analyzing the similarity information compared to other approaches.	Up to 4 variants: 30% less analysis effort 5 and more variants: 50% less analysis effort
<b>HS3:</b> Analysis Effort Scalability	The effort for analyzing the similarity information using the Variant Analysis approach grows slower with an increasing number of analyzed variants compared to other approaches.	For any $m, n \in \mathbb{N}$ , where $m > n \geq 2$ , the effort fulfills: $VA(m)/VA(n) < OTHER(m)/OTHER(n)$
<b>HS4:</b> Understandability	The Variant Analysis approach allows for understanding the implemented similarity with a higher correctness compared to other approaches.	50% less false statements
<b>HS5:</b> Practicability	The Variant Analysis approach can be successfully used by practitioners.	Successful industrial applications: > 90% positive feedback

Table 5 The scientific hypotheses

Details of the scientific hypotheses

While the hypotheses HS1 (Correctness) and HS5 (Practicability) concern the properties of the Variant Analysis approach only, the remaining three hypotheses have a comparative character. Hence, the hypotheses HS2, HS3 and HS4 concern the improvements provided by the Variant Analysis approach compared to the currently existing analysis techniques – such as the use of comparison algorithms or clone detection approaches combined with their respective structural abstraction mechanisms. Below, we further discuss the details of the particular scientific hypotheses:

- The hypothesis HS1 (Correctness) concerns only the technical correctness of the provided results and, in contrast to the other four hypotheses, does not include any human-based factors.
- The hypothesis HS2 (Analysis Effort Reduction) specifies two values of the improvement measure, depending on the number of analyzed variants. This is a result of the hypothesis HS3 (Analysis Effort Scalability), which states that the effort savings achieved by the use of the Variant Analysis approach increase with the increasing number of analyzed variants. In other words, we postulate that with a growing number of variants, our approach is more scalable in terms of the involved human analysis effort than the other current approaches. The hypothesis HS3 is based on the observation that the

current approaches do not provide sufficient abstraction mechanisms for a larger number of variants (see Section 3.3). Hence, the positive effect of such mechanisms defined in the Variant Analysis approach should be increasingly visible when the number of variants grows.

- The hypothesis HS4 (Understandability) concerns humans who analyze the information provided by an analysis approach and derive from it higher-level statements concerning the similarity of analyzed assets. With this hypothesis, we postulate that the results provided by the other approaches are harder to understand (for example due to their ambiguity – see Section 4.2.2, especially Figure 18) and might hence lead to false statements concerning similarity. False statement is a statement about code similarity which is formulated by a human as a result of using a given analysis technique, and which is incorrect because the real situation in the code is different.
- The hypothesis HS5 (Practicability) implies not only that the practitioners are technically able to understand and apply the Variant Analysis approach, but also that they are satisfied with the use of the obtained results in their migration and maintenance decisions.

As in the case of the practical hypotheses, the postulated improvement measures are not absolute, but rather indicate our intuition regarding the type and significance of the expected improvement. We postulate that the specified improvements should be in general achievable, unless a very unfavorable combination of other influencing context factors occurs.

#### Hypotheses summary

In Figure 13, we provide a summarizing overview of the formulated hypotheses and their relations to the defined application scenarios. While some practical hypotheses are specific to a given application scenario, the fulfillment of any scientific hypothesis contributes to the fulfillment of every practical hypothesis.

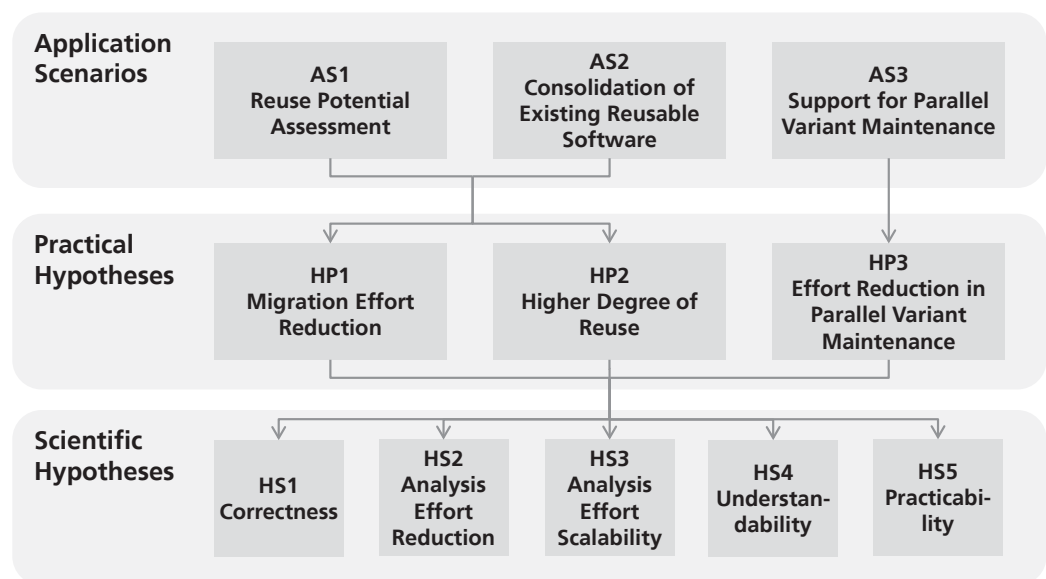


Figure 13

Overview of the practical and scientific hypotheses and their relations to the application scenarios

## 3.5 Summary

In this chapter we provided a consolidated description of the problem area addressed by this thesis. First, in Section 3.1 we presented the results of an industrial survey of code cloning practices performed in the context of variant software system development. The study results indicate that the existence of detailed information on the cloned software asset similarity would be beneficial for solving the occurring maintenance challenges – however, that information is frequently missing in the practice. Subsequently, in Section 3.2 we defined three application scenarios for a technique analyzing variant similarity, which we derive from the study results, a literature review and own practical experiences. By characterizing the information needs resulting from the application scenarios, we defined the analysis goals for our asset variant similarity analysis technique. In Section 3.3 we argument that the existing similarity analysis approaches do not address the analysis goals adequately, as they exhibit a range of shortcomings. Finally, in Section 3.4 we provide the improvement hypotheses, related to the application scenarios, which describe the benefits we expect from the solution described in this thesis. Hence, this chapter completed the description of the addressed research problem and its scientific and practical context, and set the scene for the following chapters which provide our solution.



## 4 Investigation and Formalization of the Variant Similarity Analysis Problem

In the previous chapters, we motivated the practical need for analyzing variant source code for similarity and defined application scenarios where that need occurs. In general, the purpose of performing variant code similarity analysis is to:

- recover the similarity information from the existing assets,
- support the human in understanding that information,
- and finally provide a fact base for decisions aimed at solving a practical problem defined by the given application scenario.

However, different analysis techniques might fulfill that purpose to a different degree. Therefore, this chapter provides a foundation to reason about such techniques and to evaluate them. We start by characterizing the specific properties of software variants, especially those which distinguish them from software versions (Section 4.1). Based on these properties, we define and motivate a group of construction requirements that a variant similarity analysis approach should fulfill in order to achieve its purpose (Section 4.2). The requirements provide means to compare and evaluate variant similarity analysis techniques, and can serve as guidance when defining a new technique.

Furthermore, in Section 4.3 we extend the defined construction requirements by deriving a group of assumptions which concern the nature of information needed in the specified application scenarios. These assumptions motivate the design decisions structuring the analysis approach described in this thesis. In Section 4.4 we formally define the similarity analysis of software variants based on the stated construction requirements and assumptions. In Section 4.5 we define an approach to measure the result quality of a similarity analysis structured according to our definition. Finally, in Section 4.6 we systematize the introduced concepts by defining a conceptual model of variant similarity analysis. Hence, the basis for reasoning about the analysis problem, the possible solutions, and the quality of results provided by these solutions is defined.

### 4.1 Software Variants

Versions and variants

Although software versions as well as software variants represent distinguished states of a given software asset, they have a fundamentally different nature. Basically, a version refers to an identified state assumed by an asset at a specific point in time, while a variant refers to one of multiple asset states which exist (or potentially exist, e.g. can be automatically generated) at the same point of time (Figure 14).



Definition 11      Software version

*A version of a software asset is an identifiable, unique state of the content of that asset (e.g. of its source code) created at a given point in time. A change to the asset content results in a creation of a subsequent version.*

Definition 12      Software variant

*A variant of a software asset is one from a group of identifiable, unique states of the content of that asset (e.g. of its source code) which exist or can potentially exist at the same point in time.*

Time ordering of versions      A group of software versions represent different states of a given software asset that it assumed in time (Figure 14). Therefore, the versions can be treated as points placed on the axis of time as they represent the temporal change of the asset content. The time axis defines an order on the group of versions, and allows treating that group as a list, unambiguously ordered by creation time. This order has a clear objective meaning as it represents a linear flow of changes performed on the asset during its development history. Typically, two asset versions which according to the time order are neighboring or placed near each other (i.e. they are separated by only a low number of other versions) are much stronger related or similar to each other than two versions which are distant (i.e. separated by many other versions).

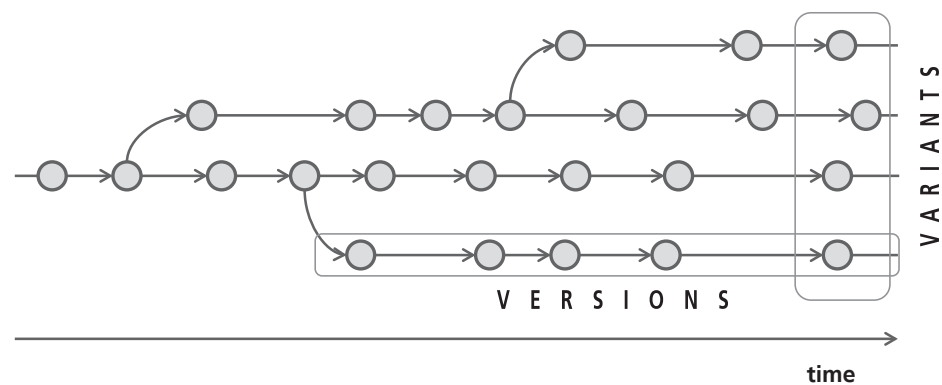


Figure 14      Versions and variants of a software asset

No objective ordering of variants      In contrast to that, every element of a group of software variants exists and is valid simultaneously at the same point in time (Figure 14). Variants represent the spatial variability of the asset content, due to the fact that different forms of the asset content are used simultaneously at different logical locations – for example in different products of a product line. However, as the usage locations are of purely logical nature, there is no objective way to define an order on the variants in a general case (see Section 4.1.1). Hence, the group of variants needs to be treated as an unordered set.

### Implications of version ordering

The reverse engineering techniques analyzing asset versions take advantage of the linearly ordered nature of the analyzed data. Because the versions represent a flow of changes made to the asset, it is valid to assume that a change made to a given asset version is present in all the subsequent versions of the asset unless another change modifies or removes the affected content. Therefore, for the analysis techniques it is sufficient to relate each asset version to its immediate predecessor to fully characterize the differences across a group of analyzed asset versions (see the left part of Figure 15). Hence, for  $n$  analyzed versions the analysis needs to consider  $n-1$  relations between versions. Since such analysis depends on the defined order of analyzed versions, any modification of that order can lead to a different analysis result.

Furthermore, the focus of the version analysis techniques is to characterize the changes performed on the asset according to the direction defined by time. Therefore, the time direction is relevant when comparing two asset versions, and the time ordering of versions is used in the presentation of comparison results. For example, a comparison of versions V1 (predecessor) and V2 (successor) could produce a result such as “15 code lines added, 2 deleted”. In this case, the presence of 15 code lines in V2 which are absent in V1 is interpreted as addition due to the fact that the version V2 is the successor of V1 according to the time ordering. Given an opposite ordering of these versions (i.e. if V1 would be a successor of V2), the same comparison would produce a result with an opposite interpretation of the found change: “2 code lines added, 15 deleted”. Hence, the direction of the version ordering is relevant in the interpretation of version analysis results, as symbolized by directed relation edges in Figure 15.

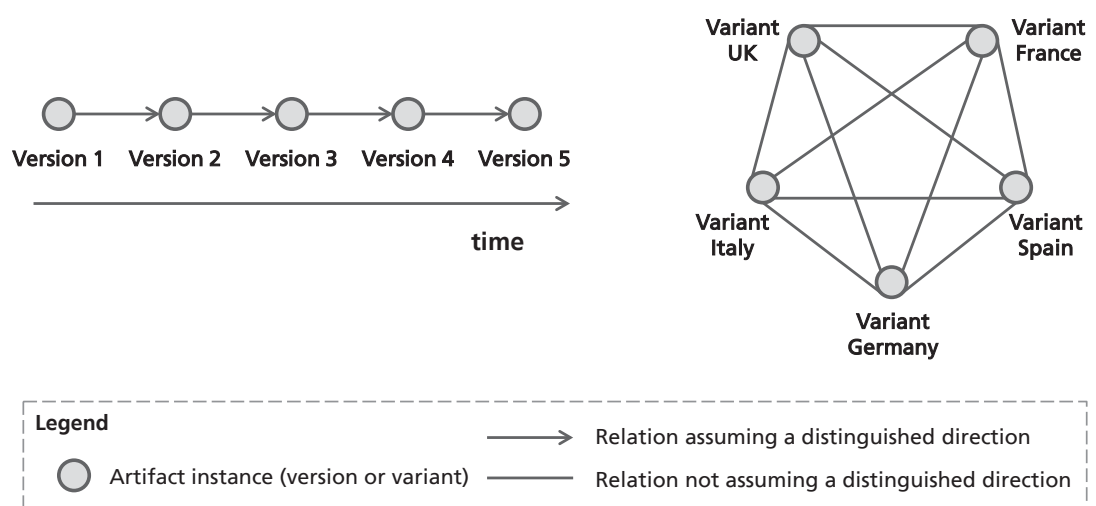


Figure 15

The basic analysis schema for software versions and variants

Implications of the lack of variant ordering

Software variants have no objectively defined order that could be used by a respective analysis technique. Therefore, a variant analysis technique should not assume or depend on any variant order – such dependence would mean that using different input orders of the same variants could lead to different analysis outputs. Since none of the orders is distinguished or correct, it would not be decidable which of the many different analysis outputs is correct. Moreover, since all the analyzed variants exist simultaneously at the same point in time, no assumption of the content of a specific variant can be derived by only looking at the contents of any other variant or group of variants. To fully characterize a group of software variants, it is necessary to relate each asset variant to every other analyzed variant. Hence, for  $n$  analyzed variants the analysis technique needs to consider  $\frac{n(n-1)}{2}$  relations (Figure 15 right).

Moreover, the lack of order defined on the analyzed variants means that there is no objective reason to interpret the differences between any two of the variants by assuming a specific direction of the difference, as done in the case of versions. For variants, both possible difference directions are equally relevant. This is symbolized by undirected relation edges connecting the variant nodes in Figure 15.

#### 4.1.1 A Discussion on the Lack of Objective Variant Ordering

Examples of incorrect variant ordering

In some cases, the properties of the existing variant derivation approaches (see Section 2.1.1) might suggest that an unambiguous order on the derivable software variants can be defined. In this section, we discuss and eventually refute the possibility of defining such order. The possible criteria that could deceptively be proposed as a basis for defining an order on the variants are:

- **The size of variant asset content**, calculated in bytes or as the amount of contained lower-level or higher-level elements (e.g. lines of code or modules). This measure can be calculated for all variant derivation approaches. However, the size of a variant does not necessarily correspond to its functionality or other properties, so that it is possible that variants having similar sizes might strongly differ from each other due to implementing very different features. Ordering variants by their size would in most cases not place the more related or more similar variants in the neighborhood of each other, as typically expected in the case of time-ordered software versions. Finally, some variants could have an identical size, despite different content, so that the decision on their ordering would need to be subjective. Hence, the content size cannot be used as an ordering criterion in the general case. The arguments provided above apply analogically to the idea of ordering the variants by **the number of selected parameters or features**.

- **The value of a certain configuration parameter**, for example a numeric parameter. This measure could be calculated for compositional and annotative approaches, as they explicitly maintain the configuration information. However, as the space of potential variant assets is multi-dimensional due to the existence of many parameters and many values for each parameter, there could be more than one variant having the same value of the selected configuration parameter. This would again result in the necessity of a subjective ordering decision. An analogical case occurs in mathematics, where there is no natural ordering definable on the set of complex numbers. Furthermore, in a general case it is not guaranteed that the parameter selected as an ordering criterion corresponds or correlates to the properties analyzed by a given reverse engineering approach.

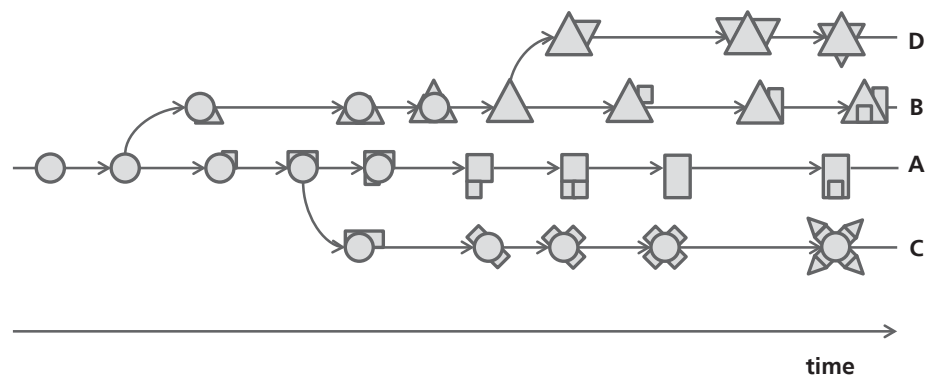


Figure 16

A schematic visualization of example content change across a group of related asset variants

- **Variant creation time**, which could be especially relevant in the context of duplication-based approaches. For example, one could argue that a variant that was derived (branched, cloned) from the main development line at an earlier point in time should be less related or similar to the main line than another variant derived from it at a later point in time. Hence, the initial derivation time of the variants would correlate with their similarity to the main line. According to that ordering proposition, variant B in Figure 16 should be the least related or similar to variant A, since it was derived from it first, while variant C should be a little more similar and variant D should be the most similar. However, there are several observations that make this ordering idea invalid:
  - Variants might evolve at different speeds, so that the content of a younger variant might be changed faster than the content of an older variant. Eventually, the amount of changes performed on the younger variant might be greater than the amount of changes performed on the older variant. Hence, the amount of change relative to the first version of the given variant does not need to correlate with its age. In Figure 16, variant B is more similar to variant A than variant C, despite the fact that B is older than C.

- New variants might be derived not only from the main line, but also from any other variant. For example, in Figure 16 variant D was derived from variant B. In case a new variant is derived from an existing variant which is the least similar to the main line, it will, at least initially, also be one of the least similar variants, despite being the youngest. In Figure 16, variant D is less similar to variant A than variant B, despite being younger than variant B.
- If a new variant is derived from a non-mainline variant, and in a short period of time a second new variant is derived from another non-mainline variant, these new variants might be very dissimilar to each other (depending on the similarity of the original parent variants) despite having a proximate creation time.
- Finally, the derived variants might develop in different “directions”, that is, each of them might be extended by different new features as illustrated in Figure 16. Moreover, the similarity between the parent variant and the derived variant does not only depend on the changes performed on the derived variant, as the parent variant is typically developed further after the derivation too (for example, in Figure 16 compare the latest state of variant A to the state it had at the moment when variant B was derived from it). Also, some variants which were initially dissimilar might in the course of their evolution be extended by the same or analogous features and become more similar (as in the case of variants C and D in Figure 16). Some content parts might also be exchanged at a later time between any groups of the cloned variants (an activity known as porting). Hence, in a general case the similarity of a group of variants does not need to correspond to the order in which they were initially created. As a consequence, also the approaches going in the opposite direction by attempting to deduct the evolution history of a group of software variants based on their code similarity ([Yamamoto 2005] [Tenev 2012] [Kanda 2013], see also Section 5.5.5) will not always produce a result identical to the actual evolution history in the general case.

Possible  
meaningful  
variant  
ordering

In some specific cases it might be possible to define a meaningful order on a group of software variants. A precondition for such an order is that there exists a known correspondence between the defined order and the property analyzed by the reverse engineering technique. For example, a software product can have a “Minimal”, “Standard” and “Extended” variant, where each larger variant is created by only adding features to a smaller one. In such a case, ordering the variants by size measured as number of used features can be justified if the analyzed properties are for example code similarity or memory footprint. A good understanding of the analyzed products is necessary to recognize such a case. However, in the general case techniques analyzing software variants cannot assume that any specific order is definable.

## 4.2 The Construction Requirements for Techniques Analyzing Variant Similarity

Symmetry of a variant analysis	<p>In the previous section we characterized a fundamental difference between software versions and variants: while the versions are ordered by time, and can therefore be treated as an ordered list of asset states, the variants cannot be ordered in a general case and need therefore to be treated as an unordered set. Because of that, a technique analyzing software variants should not make any assumptions regarding the order of variants and should treat each variant symmetrically: <b>no variant should be in any way distinguished in the analysis or in result interpretation.</b> In the following subsection, we elaborate on the consequences of this important property by defining construction requirements for any general-purpose technique analyzing similarity of software variants. In the later subsections, we discuss further general requirements, from which any such analysis technique would benefit. The requirements provide means to compare and evaluate variant similarity analysis techniques, and can serve as guidance when defining a new technique.</p>
Origin of the construction requirements	<p>The defined construction requirements are derived from theoretical reflections on the variant similarity analysis problem, a review of the existing approaches (see Sections 2.3 and 3.3), as well as from our practical experiences in applying earlier versions of our Variant Analysis technique in the industry ([Duszynski 2008] [Duszynski 2011a], see also Section 7.4). In our experience, software architects and developers need to be very well informed before they make a decision on a code transformation as significant as it is needed for introducing software reuse. Hence, the information provided by an analysis technique needs to be sufficiently detailed – as the source code is their ultimate mean to specify system functionality, the code-level facts as well as code-level consequences of transformation are very important to the developers. Naturally, the provided information needs to be highly trustable and dependable – optimally the architects and developers should understand or trust the analysis algorithm creating the information and be able to verify the result manually. Finally, the developers demand a high control over the process of code transformation – hence, the code transformation activities should be performed or at least controlled by a human, as a code automatically transformed to a reusable form might be not trusted by the developers and might look unfamiliar to them, hence inhibiting the attempted maintainability improvement. Therefore, the code similarity information needs to be easily understandable by a human to facilitate the manual code modification activities.</p>
Non-technical factors in the analysis	<p>The above characteristics of technical stakeholder needs are not entirely of a technical nature, but also include psychological and cultural factors such as the tendency to risk avoidance. In a practical experience report from industrial reengineering projects in the financial domain, Cordy</p>



provides several observations from the perspective of an analysis tool provider that we'll agree with our above experiences [Cordy 2003]. He further notes that providing correct analysis answers are very important, and having partial but correct information is better than no information or incorrect information. Although Cordy stresses that his observations might not be generalizable, which also applies to our experience, we believe that accounting for the specific needs and attitudes of stakeholders who use the provided information is important for any reverse engineering technique.

#### 4.2.1 Consequences of the Lack of Variant Ordering

The lack of defined order in the set of analyzed variants makes it necessary to make the analysis and its results order-agnostic. Therefore, the analysis needs to use:

**Commutativity [C1] Commutative analysis operations:** consider two software variants, abstractly represented by two intersecting sets A, B (Figure 17). The sets contain respectively 5 and 7 elements, and three of these elements are common to both sets. An example analysis operation, denoted as  $\nabla$ , is commutative when for two analyzed sets A, B, the analysis result is the same for  $A \nabla B$  and for  $B \nabla A$ , and non-commutative otherwise. Since a non-commutative operation assumes a certain order on the input sets, which is not defined for software variants, such an operation would be unsuitable for an analysis of variants.

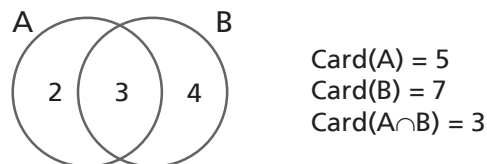


Figure 17 Two example intersecting sets A and B

**Commutative result presentation** The commutativity of operation  $\nabla$  concerns also the presentation of the created result. For example, if  $\nabla$  were a comparison operation, a non-commutative result presentation for sets A and B could be "60% of set A elements also belong to set B", because for the other order of compared sets, we get a different result of "42.9% of set B elements also belong to set A". A commutative result presentation for these sets is "3 elements of set A also belong to set B", "33,3% (3 out of 9) elements that belong to at least one of the sets A, B, belong to both these sets", or "the intersection of sets A, B contains 3 elements".

A non-commutative result presentation is problematic because it can potentially be different for any permutation of the analyzed sets. Since none of the orders defined by such permutations is distinguished, it would not be decidable which of the many different analysis outputs is

correct. Using only one of the different results can be misleading, while using all of them makes the result interpretation complex and ambiguous. In contrast to that, using a commutative presentation produces just one analysis result, which helps interpretation.

Associativity

**[C2] Associative analysis operations:** in the implementation of the analysis technique, it might be necessary to perform some operations on selected subgroups of the analyzed variant set first, and aggregate the partial results later. Also, for some data parallel operations (i.e. operations performed multiple times on different data, such as the pairwise comparison of each pair of variants), the implementation might require that these operations are performed in a sequence – for example if a single program thread is used. However, the final result of the analysis should not depend on the selection of such groupings or on the order of the performed operations. The associativity property details the previously stated condition that the result of the analysis must not depend on the order in which the variants are provided in the analysis input.

#### 4.2.2 Providing Detailed Result Information

The construction requirements described in this subsection support the goal of providing result information in a possibly complete, detailed and user-verifiable form.

Availability of  
all variant  
combinations

**[C3] Information on all possible variant combinations** should be provided in the analysis results. Several comparison approaches performed on variants, for example [Yamamoto 2005] and [Mende 2008], present the analysis results as a square matrix of variant-to-variant similarity metrics (see the example in Figure 18). Although this result presentation might seem to be natural, as the comparison of many variants is usually performed pairwise with each of the variants compared to each other, it hides important information when used for three or more variants, such as for example the size of the common parts shared by all analyzed variants.

Incompleteness  
of pairwise  
result  
presentation

Consider the two situations depicted in Figure 18, where three variants abstractly represented by sets  $A$ ,  $B$ ,  $C$  are analyzed for similarity. On the left side of Figure 18, all the commonality between the three sets is located in the subset  $A \cap B \cap C$ , while the other set intersections  $A \cap B \cap C'$ ,  $A \cap B' \cap C$ ,  $A' \cap B \cap C$  are empty ( $A'$  denotes the complement of set  $A$ , that is the set's negation). On the right figure side, each of these other set intersections has a cardinality of 2, which is a sizable part of the sets  $A$ ,  $B$ ,  $C$ , and the size of the subset  $A \cap B \cap C$  is reduced to one element. However, for both situations the cardinalities of the three sets  $A$ ,  $B$ ,  $C$  are identical, and the cardinalities of the two-set intersections  $A \cap B$ ,  $A \cap C$  and  $B \cap C$  are also identical. Hence, despite the strong difference between the two analyzed situations, for both of them identical pairwise



comparison results and identical square similarity matrix are created (the middle of Figure 18), which is of course undesirable. For more than three variants, examples of much stronger differences which still lead to identical results in the pairwise presentation can be constructed.

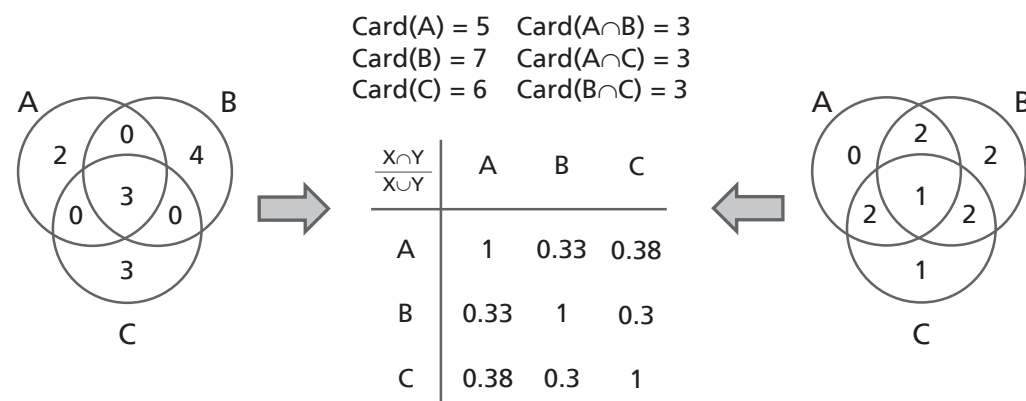


Figure 18 The inadequacy of pairwise result presentation: identical results are provided (middle) although the analyzed situations (left, right) strongly differ

The above example illustrates the fact that pairwise result presentation does not deliver full information about the analyzed variants. However, result information about any possible combination of the variants, and not only about the pairs, is needed. For example, a similarity analysis might need to calculate the size of the asset content shared by all variants – this information is not available in the square similarity matrix. Moreover, providing the information on all possible variant combinations is necessary to distinguish different analyzed situations such as those depicted in Figure 18.

Highly detailed results

**[C4] Information detail level** should be sufficient with regard to the analysis goals. For example, a similarity analysis technique that provides input for planning code reengineering should be detailed enough to detect even small, but meaningful, code-level differences between the analyzed variants. In case the variants differ in any aspect that is relevant to the developer, they should not be recognized as identical.

Some analysis techniques provide only low-detail results in the form of metrics or abstracted code representation such as UML models. Such result form is not detailed enough for use in implementation level activities, as it is possible to define two assets or asset part variants with identical metric values or model representations which still implement very different functionalities. In such a case, a developer could not depend on the analysis result and could not assume that the analyzed asset variants are also identical on the code level. Thus, such low-detail representations are not suitable for use in the planning and performing of implementation level activities.

Traceable  
results

**[C5] Traceability to implementation** is another aspect of the information detail level. For each part of the analyzed asset implementation, it should be possible to determine how that particular part contributes to the analysis result. For example, in a similarity analysis it is not sufficient to deliver a single similarity metric on the input asset variants (such as, e.g., the Levenshtein distance [Levenshtein 1966] of their textual representations). A similarity metric does not allow to determine how exactly a given variant differs from the others and which parts of the content contributed in which way to the calculated similarity value. Optimally, it should be possible not only to determine the amount or proportion of content identified as being similar or different in the analyzed variants, but also to trace down which of the elementary content elements are recognized as similar. Consequently, for any content element the information about its similarity should be known, reflecting the information detail level requirement.

#### 4.2.3 Result Presentation and Interpretation

The construction requirements described in this subsection reflect the requirements regarding the ease of result interpretation.

Abstraction  
mechanisms

**[C6] Proper result abstraction:** although sufficient details should be available in the analysis result, the user should not be overwhelmed with these details unless they are demanded. A proper abstraction, allowing uniform result presentation regardless of the analyzed asset size and the number of analyzed asset variants, is needed. The human effort for result preparation and abstract result interpretation should not grow with increasing asset size, and should grow only moderately with increasing number of analyzed asset variants.

For example, a result presentation in the form of square similarity matrix (Figure 18) does not provide a good abstraction mechanism and is complex to interpret. Even though the matrix is symmetric and its diagonal can be ignored, for  $n$  compared variants it still contains  $\frac{n(n-1)}{2}$  significant numbers characterizing the overall similarity. Thus, a human needs to relate all these numbers to each other in order to understand the overall situation. However, even for low values of  $n$  the amount of the numbers can be large – for example for 10 variants the matrix already contains 45 significant values.

The need for  
size infor-  
mation

**[C7] Indication of the relative size of variants and their elements** is helpful as the analyzed asset variants usually have different sizes. Also, the sizes of their content parts (subsystems, files) are likely to vary strongly. Because of this, the information about the degree of similarity found in a particular asset or its part should be accompanied by an indication of the size of that part. This enables better interpretation of the result since it is, e.g., easier to understand how the differently sized parts contribute to the total similarity of the asset.

#### 4.2.4 Other Requirements

General analysis requirements	<p>The above list of requirements focuses on the aspects relevant for similarity analysis of software variants. However, it does not cover further, general principles which are relevant for any reverse engineering technique, such as:</p> <ul style="list-style-type: none"><li>• result completeness, i.e. the extent to which all information relevant to the goal is retrieved,</li><li>• result correctness,</li><li>• robustness, e.g. against unexpected or non-standard input data,</li><li>• efficiency (also covering scalability of the technique and even the possibility to automate its steps).</li></ul>
Realization for an analysis of variants	<p>Quality evaluation of a variant similarity analysis technique should incorporate both the specific as well as the general construction principles. For example, result correctness and completeness can be evaluated in terms of precision and recall (see Section 4.5). Robustness concerns the behavior of the analysis in untypical situations such as processing an asset variant strongly dissimilar from the others or receiving multiple identical variants in the analysis input. The scalability of a similarity analysis means in the practice that it should be possible to perform the analysis for many variants of a software asset (10 or more), and for assets as large as software systems having millions lines of code. Efficiency means, among others, that the analysis should be automatable and that the user should be able to access result details or to validate hypotheses formulated upon result investigation (e.g., by on-demand calculations) with low response times.</p>

### 4.3 Assumptions Resulting from the Application Scenarios

The focus of our analysis technique	<p>The construction requirements described in the previous section provide a general framework for discussing and evaluating any similarity analysis technique for software variants. However, in the context of this thesis we further assume that our specific similarity analysis technique should be mainly used for the application scenarios defined in Section 3.2, which are:</p> <ul style="list-style-type: none"><li>• [AS1] Reuse potential assessment,</li><li>• [AS2] Consolidation of existing reusable software,</li><li>• [AS3] Support for parallel variant maintenance.</li></ul>
-------------------------------------	--

In Section 3.2 we characterized the common characteristics of these three application scenarios and derived from them general goals concerning the information provided by the similarity analysis. In this section, we elaborate on the consequences of the selected application scenarios by defining a group of construction assumptions for our analysis technique. The assumptions play a complementary role to the construction requirements described above, and together with them

form the basis on which our analysis technique is defined. In contrast to the construction requirements, we do not postulate that the below assumptions need to apply for other analysis techniques.

Focus on the similarity between asset variants

**[A1] The analysis results should only concern similarities between the variants of software assets,** as this is the main interest of the analysis users in the specified application scenarios. A search for similarity inside the same asset variant is not necessary, and can be sufficiently covered by already existing techniques applied to a single software asset variant, such as clone detection.

Focus on highly similar assets

**[A2] A relatively high structural similarity of the analyzed variants can be assumed.** Since the ultimate goal of two application scenarios AS1 and AS2 is to merge the analyzed software assets into a common reusable form, the similarity found by the analysis technique needs to be sufficiently high in order to allow such merge activities. Also for the third scenario AS3, the intended maintenance tasks can only be repeated on code parts having a high enough similarity. However, the similarity reported by the analysis technique can only be as high as the similarity actually existing in the analyzed assets. Therefore, while the analysis has to provide high quality results in the scope of the application scenarios, the asset groups with low similarity are not in that scope. Consequently, it is acceptable for the analysis to exhibit reduced performance for input assets having low similarity.

One-to-one correspondences

**[A3] Use one-to-one correspondences between the parts of asset variants recognized as similar.** As the analyzed variants are in most cases created in a cloning process, for many of their assets there exists a single counterpart asset in another variant that shares a common ancestor with the given asset in the cloning history. Even if the properties of asset evolution history not always correspond to the similarity of asset variants (see Section 4.1.1), the asset variants sharing a common history are in most cases still more similar to each other than assets developed independently. Moreover, identification of a single counterpart to an analyzed asset is helpful for planning the cross-variant code merging activities in the context of defined application scenarios, while code merging inside one variant can be addressed using information provided by clone detection techniques. Finally, this one-to-one correspondence assumption is a direct consequence of the previous assumptions A1 and A2. Since identifying similar assets inside one variant is not necessary, also matching all these assets to some asset in another variant is of limited use and can actually complicate the interpretation of analysis results. Also, the assumed high structural similarity makes it likely, for all but the smallest assets, that at most one appropriate matching candidate exists in the other variant, even if its content and the location in the structure hierarchy were modified in the course of variant evolution.

High result  
certainty

**[A4] Make the analysis results dependable – prefer higher certainty results over providing more results.** As the information provided by the analysis greatly impacts decisions concerning code transformation activities of a potentially large scale, it is crucial that this information is as correct as possible. As discussed in Section 4.2, the risk of making an incorrect reuse decision or of otherwise reducing code quality has to be minimized. At the same time, the demand to minimize the human effort for information analysis, resulting from the defined application scenarios, implies that the human should be able to trust the provided result and should not need to manually inspect the code to verify the result correctness. Therefore, there should be a very high certainty that the provided results accurately characterize the real state of the analyzed assets, i.e. that the identified similar assets are indeed similar. It is acceptable that the analysis uses techniques that maximize the result certainty even at the cost of missing some relevant results: it is better to miss some similarity than to report dissimilar assets as being similar. Hence, although both the certainty and the quantity of result information are important, the certainty takes precedence over the quantity.

The described concepts of result certainty and quantity are analogous to, but more general than the information retrieval measures of precision and recall (see Section 4.5). In information retrieval terms, the analysis technique should strive for high result precision, and sacrifice the recall if necessary. In addition to that, result certainty also means that providing information which is known to potentially contain incorrect values, or which requires further validation, should be avoided. For example, many clone detection approaches such as the one of Mende et al. recognize any two functions having a Levenshtein distance based similarity of 70% or more as similar, regardless of the type of remaining dissimilarities [Mende 2008]. Such a result is not certain enough as the remaining 30% of code can potentially contain functionalities that still differ enough to prevent a transformation of the analyzed functions to a reusable form.

Transitivity

**[A5] Provide transitive similarity results.** In case an asset A is recognized by the analysis as similar to asset B, and asset B is recognized as similar to asset C, the similarity of A to C should also be given. Transitive similarity results are much easier to interpret than non-transitive ones, as similarity of an asset to any element from a given group implies that the asset is similar to every element of that group. Hence, only the members of the group, and not the topology of the recognized pairwise similarity relations among these group members, need to be known in order to fully understand the reported similarity result. As in the practice analyses of 20 and more asset variants are possible, and hence groups of similar elements having 20 and more elements can be found, an interpretation of a complex graph of up to 190 possible pairwise similarity relations among such 20 elements would be a very complex task – which would need to be repeated several times during a single analysis result interpretation as each variant would likely contain several analyzable content elements. In contrast to that, a

transitive result is simple to understand as it only needs to name the similar elements and state the nature of similarity relation that bounds any two of them. Hence, transitivity well supports the need to provide easily interpretable analysis results.

Because of the assumption A4, demanding high result correctness, the similarity reported by the analysis cannot be constructed as a transitive closure of the actually found similarity relations, as this might result in incorrectly reporting dissimilar asset pairs as similar. Hence, the analysis should report a transitive form of the found similarity by either constructing a transitive subset of the result, skipping the pairwise relations that do not contribute to a transitive result form, or by relaxing the assumed similarity definition, so that it fully and transitively applies to the transitive closure of the previously found result. However, relaxing the similarity definition should not conflict with the result correctness assumption A4 – hence, skipping some of the detected non-transitive relations is preferred. As there are several possible algorithms to perform that step, which provide results optimized according to various criteria, we defer further discussion on the possible alternatives to Chapter 5.

There are several advantages in processing and interpreting transitive results of a similarity analysis. Apart from the already described easy interpretation of the results, transitivity enables a stronger and more focused definition of the analysis approach described in this thesis. At the same time, the information loss due to the creation of a transitive result subset is in the practice minor for our instantiation of the approach (see Section 7.1).

## 4.4 A Formal Definition of Variant Similarity Analysis

Scope of the analysis definition	Based on the considered application scenarios for variant similarity analysis and on the construction assumptions derived from them, in this section we formally define the notion of similarity analysis presented in this thesis. The definition provided here is formal and abstract, as it does not specify yet the exact form of the provided result, the quality of that result, further information derivable from the result, or any other technical aspects of the analysis. The details of the analysis approach conforming to the given definition are described in Chapter 5.
Definition prerequisites	The analysis approach definition is based on the construction requirement C1 (commutative analysis operations), on the assumptions A1 (only similarities between variants are considered), A3 (one-to-one correspondences between similar assets), and A5 (result transitivity), and it also assumes that any object is similar to itself (see the discussion in Subsection 4.6.2). The construction assumption A3 implies that for a given element of an analyzed asset variant, at most one corresponding similar element can be identified in each other variant. Moreover, the identified elements are transitively similar to each other (assumption A5),



and they are not identified as similar to any other element of any variant (consequence of assumption A1). Hence, for each element of any analyzed asset variant, a similarity analysis should identify zero or one similar elements respectively from each other variant. Finally, it is important to note that the similarity relation between the identified elements is an equivalence relation, as it is reflexive (any object is similar to itself, see Subsection 4.6.2), symmetric (requirement C1) and transitive (assumption A5).

**Analysis input** Let's define  $S_V$  as the set of all analyzed software asset variants, and  $S_V \neq \emptyset$ .  $S_V$  is the input data delivered to the variant similarity analysis. Furthermore, each asset variant contains some internal elements, which can be tested for similarity with the elements of any other variant (see Section 4.6). Therefore, without further defining the nature and properties of the elements, let's assume that the asset variant can be treated as a set containing its content elements. The assumed set can be correctly and unambiguously constructed, as the asset variant content contains no two elements recognized by the analysis as identical (assumption A1). Hence,  $S_V$  is a set containing non-empty sets of content elements from the analyzed variants:

$$S_V = \{S_1, S_2, \dots, S_N\}, \text{ where } S_i = \{e_1, e_2, \dots, e_K\} \text{ and } N > 0 \text{ and } \forall i \in (1..N): K_i > 0$$

Equation 1 Analysis input: a set containing non-empty sets of asset variant content elements

**Similarity selection** Let's define  $\underline{U}S_V$  as the union of all sets  $\{S_1, S_2, \dots, S_N\}$ , and let's select an element  $e$  from one of these sets, that is,  $e \in \underline{U}S_V$ . For any given element  $e$ , belonging to an analyzed asset variant, the similarity analysis should identify zero or one similar elements respectively from each other variant. To represent that selection of similar elements, we define a function  $f_{SIMSEL}$  on the input sets  $S_V$ . For a given element  $e \in \underline{U}S_V$ , the function  $f_{SIMSEL}$  assigns to each set  $S_i$  in  $S_V$  zero or one elements of that set  $S_i$  which are recognized as similar to  $e$ . Hence, the result of the function  $f_{SIMSEL}$ , returned for the sets  $S_V$  and the element  $e \in \underline{U}S_V$ , is a set of  $\underline{U}S_V$  elements which has the following properties:

$$f_{SIMSEL}(S_V, e) = \{elem: elem \in \underline{U}S_V\},$$

$$\text{where } e \in f_{SIMSEL}(S_V, e) \text{ and } \forall S_i \in \{S_1, S_2, \dots, S_N\}: (card(S_i \cap f_{SIMSEL}(S_V, e)) \leq 1).$$

Equation 2 The selection function identifying variant content elements similar to a given element  $e$

Obviously, the result of the function  $f_{SIMSEL}$  always contains the input element  $e$ . In a special case, the returned set might contain only the element  $e$  – such a result means that no element from other asset variants was identified as similar to  $e$ . Moreover, note that the function  $f_{SIMSEL}$  can also be defined as a function returning the equivalence class of the input element  $e$ . In this case, the equivalence class of  $e$  is specified by a similarity relation defined on  $\underline{U}S_V$ , which is an equivalence relation as discussed above. In the remainder of this chapter, we use the term “similarity selection” to refer to any result set returned by the function  $f_{SIMSEL}$ .

The similarity selection, which is the output element set of  $f_{SIMSEL}$ , should not be confused with a tuple that lists similar elements from each set  $S_i$ . The mathematical definition of a tuple states that tuple elements are ordered and that it contains no empty elements. However, we don't define any specific order on the sets of  $S_v$ , and for some of these sets there might be no identifiable element similar to the input element  $e$ .

**Analysis result** Finally, we define the result of a variant asset similarity analysis  $Sim\_Analysis(S_v)$  as the set of all outputs of the similarity selection function  $f_{SIMSEL}$  obtained for all content elements of all asset variants, that is for all elements of the sets  $S_i$ .

$$Sim\_Analysis(S_v) = \{ f_{SIMSEL}(S_v, e) : e \in \underline{US}_v \}$$

**Equation 3** Result of the similarity analysis for the input set of non-empty variant content sets  $S_v$

**Properties of the analysis result** Because the similarity relation constructed by the analysis is an equivalence relation, and hence the particular similarity function result sets  $f_{SIMSEL}(S_v, e)$  represent equivalence classes containing elements of  $\underline{US}_v$ , the analysis result has the following properties:

1. For any two similar elements, the function  $f_{SIMSEL}$  returns the same result:

$$\forall e_1 \in \underline{US}_v : \forall e_2 \in \underline{US}_v \wedge e_2 \in f_{SIMSEL}(S_v, e_1) : f_{SIMSEL}(S_v, e_1) = f_{SIMSEL}(S_v, e_2)$$

2. The different  $f_{SIMSEL}(S_v, e)$  sets are pairwise disjoint:

$$\forall e_1 \in \underline{US}_v : \forall e_2 \in \underline{US}_v \wedge e_2 \notin f_{SIMSEL}(S_v, e_1) : f_{SIMSEL}(S_v, e_1) \cap f_{SIMSEL}(S_v, e_2) = \emptyset$$

3. All elements of  $\underline{US}_v$  are covered:

$$\underline{U} \{ f_{SIMSEL}(S_v, e) : e \in \underline{US}_v \} = \underline{US}_v$$

**Equation 4** Properties of the similarity analysis result

Consequently, every input element  $e \in \underline{US}_v$  belongs to exactly one similarity function result set  $f_{SIMSEL_j}$ . Figure 19 schematically depicts the analysis input as defined by Equation 1, and the analysis result as defined by Equation 3. In the figure, the similar elements from each of the asset variants are displayed as similar geometrical shapes.

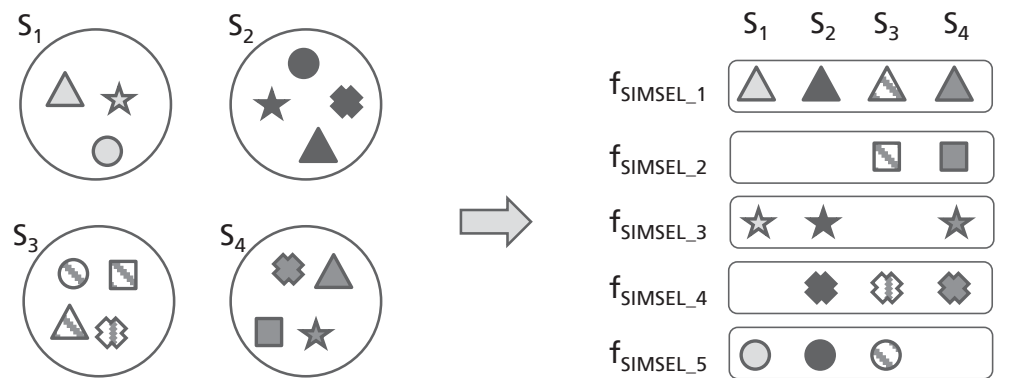


Figure 19

A schematic presentations of the similarity analysis input (left) and the analysis result (right)



### 4.5 Evaluating Quality of the Variant Similarity Analysis Results

The need for result quality evaluation

The definitions introduced in the previous section specify that the similarity analysis result is a set of similarity selection sets. These definitions describe the form of the analysis result, but do not define its quality. In an ideal case, the similarity relation constructed by the analysis should truly and fully correspond to the actual similarities of the analyzed asset content elements. However, in a practical case the similarity analysis, as well as many other reverse engineering approaches, can produce information different from the ideal solution, for example due to inaccuracies between the analyzed reality and the model of that reality assumed by the analysis algorithm. Hence, a means for evaluating the correctness and completeness of analysis algorithm result is needed in order to assess the quality of a given algorithm in the practice.

#### 4.5.1 Evaluating Results of Information Retrieval Problems

Precision and recall measures

The variant similarity analysis problem is an example of an information retrieval problem. Hence, in this subsection we briefly introduce the information retrieval measures of precision and recall, which are used to measure the correctness and completeness of analysis results.

The left part of Figure 20 describes an abstract model of an information retrieval problem and of the result of an analysis solving that problem. The analysis solves the problem by retrieving all the elements from a given group that fulfill a specified criterion: in Figure 20, all circles which are filled should be retrieved. The elements retrieved by the analysis are called *positives*, and the not retrieved elements are called *negatives*.

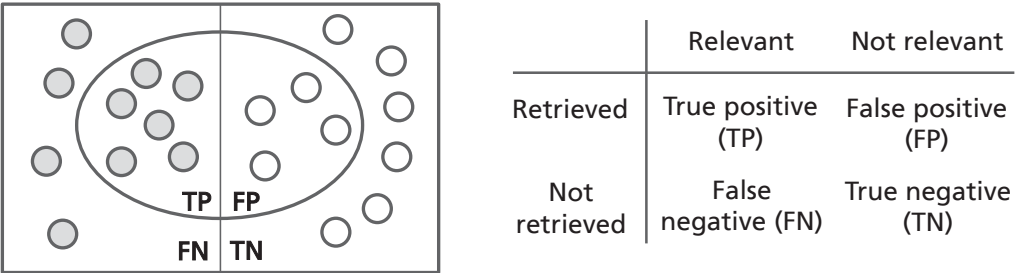


Figure 20 A model of an information retrieval problem and the four possible result categories

To depict the most general case, the analysis result symbolized by the large circle in Figure 20 missed some of the expected elements and also returned some other elements not belonging to the correct result. This illustrates that the input elements can be classified based on the analysis result into four categories: true positive (TP), false positive (FP), false negative (FN), or true negative (TN), as defined in the right part of Figure 20. The measures of precision and recall, defining respectively the correctness and completeness of the retrieved result, are based on these four element categories and are defined as follows [Manning 2008]:

Definition 13 Precision

*Precision (P) is the fraction of retrieved documents that are relevant:*

$$P = \frac{TP}{TP + FP}$$

Definition 14 Recall

*Recall (R) is the fraction of relevant documents that are retrieved:*

$$R = \frac{TP}{TP + FN}$$

Hence, high recall means that most of the existing relevant results are found, while high precision means that most of the found results are relevant. In an ideal case, all the relevant results are returned, and none of the returned results is irrelevant, resulting in the maximal possible recall and precision values of 1.

#### 4.5.2 Definition of Precision and Recall Measures for Similarity Analysis Results

Result construction by set merging and splitting

Let's assume that the analysis result depicted in Figure 19 presents the ideal, correct solution of the similarity analysis problem. Consequently, any other result provided by the analysis for the given input sets, such as the result provided in Figure 21 (below), is not fully correct. Note that any analysis result being different from the ideal one can be constructed from the ideal result by applying, if necessary several times, one or both of the following two operations:

- Some of the original similarity selection sets could be split into two or more sets, incorrectly implying that there is no similarity between the element of these new partial similarity selection sets,
- Some of the (partial or original) similarity selection sets could be merged into larger sets, incorrectly implying that there is similarity between the elements of the previously separate selection sets.

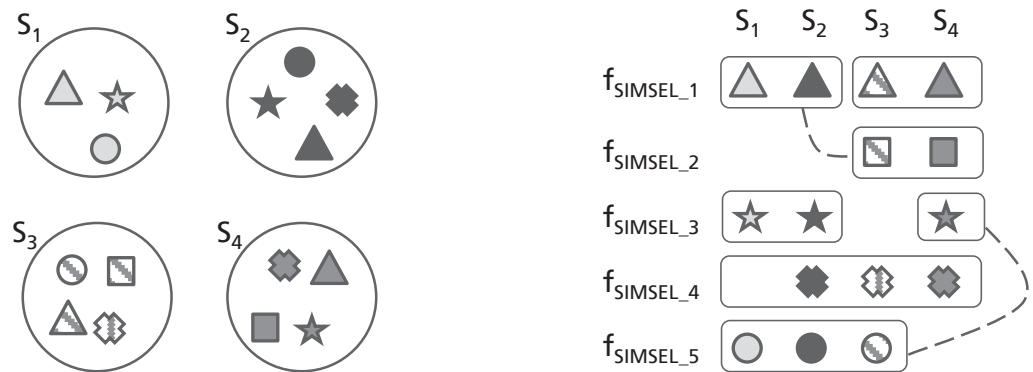


Figure 21

An example of an incorrect analysis result

For example, in Figure 21 the sets  $f_{\text{SIMSEL}_1}$  and  $f_{\text{SIMSEL}_3}$  were split in two parts each, implying that there is no similarity between the elements of these parts. Subsequently, the first part of  $f_{\text{SIMSEL}_1}$  was merged with  $f_{\text{SIMSEL}_2}$ , implying transitive similarity between the contained elements: the light-gray triangle, the black triangle, the white square and the dark-gray square. Finally, the second part of  $f_{\text{SIMSEL}_3}$  was merged with  $f_{\text{SIMSEL}_5}$ . To evaluate the degree to which the constructed result is incorrect, we discuss now the definition of the measures of precision and recall for the given form of the analysis result.

Problems of  
evaluating  
pairwise simi-  
larity results

Because the analysis result is transitive, an existence of a similarity selection set containing  $n$  elements means that there exist  $\frac{n(n-1)}{2}$  pairwise similarity relations between the elements of that set. It seems intuitive to define the result correctness based on these pairwise similarity relations: an existing and found pairwise relation is a true positive, an existing but not found relation is a false negative, and so on. However, this intuitive definition is problematic in use, as it leads to multiple undesired effects. These effects can produce precision and recall values that are lower (worse) for some solutions although these solutions are intuitively better than solutions which have higher (better) measure values. The undesired effects are:

- Incorrectly assigning a dissimilar element to an existing group of  $n$  other elements creates  $n$  false positive relations (one for each element in the group). As a result:
  - For a group of 2 elements 2 false positives are created, while for a group of 10 elements the addition of one incorrect element creates 10 false positives. Hence, the second case is measured as being much worse than the first, although in both cases there is only one element which is placed incorrectly.
  - Incorrectly matching 9 pairs of dissimilar elements produces 9 false positives, while incorrectly adding one element to a group of 10 elements produces 10 false positives. Hence, the second case is measured as being worse than the first, although intuitively the opposite should be true.
  - Merging two groups, having respectively  $n$  and  $k$  similar elements, produces  $nk$  false positives. Very different numbers of false positives can result from a single decision to merge two groups. Again, a single merge of two large groups is valued as much worse as multiple merges of small groups (e.g. consider one merge for  $n=k=10$  and 90 merges for  $n=k=1$ ).
- Analogically, incorrectly excluding one element from a group of  $n$  elements creates  $n-1$  false negative relations (one for each element remaining in the group). Again, the number of false negatives

depends on the size of the initial group, which leads to counterintuitive effects and measure values for the cases of element removing and group splitting. We omit the discussion of these cases here, as they are analogical to the cases occurring for false positives.

- Moreover, it is very hard to determine the number of true negatives, which is needed for calculating further result correctness measures beyond precision and recall [Manning 2008]. Theoretically, the sum of false positives and true negatives should be constant, and the worst possible analysis result should return the maximum possible number of non-relevant elements. Hence, the sum of false positives and true negatives should equal to the maximal possible number of false positives. However, calculating that number is a complex task – especially as the analysis can in the practice be used on more than 20 asset variants having millions of content elements each.

Because of the discussed problems, we don't use the above definition of positive and negative result elements based on pairwise similarity. Instead, we introduce another, simpler definition that does not cause the listed negative effects.

Set merge  
based result  
quality  
evaluation

We define the result of the similarity analysis algorithm as a series of set merge operations performed on the elements of the input systems. Consequently,  $n-1$  correct merge operations performed on single elements construct a correct similarity selection set of size  $n$ , which represents a result containing  $n-1$  true positives. If a merge incorrectly connects two sets, it is considered a false positive. If a correct merge is not present in the analysis result, it is considered a false negative. Figure 22 depicts the true positives, false positives and false negatives existing in the incorrect analysis result introduced in Figure 21. In the figure, any two elements connected through one or more similarity relations (i.e. a true positive or a false positive) are considered similar due to the transitivity of the result.

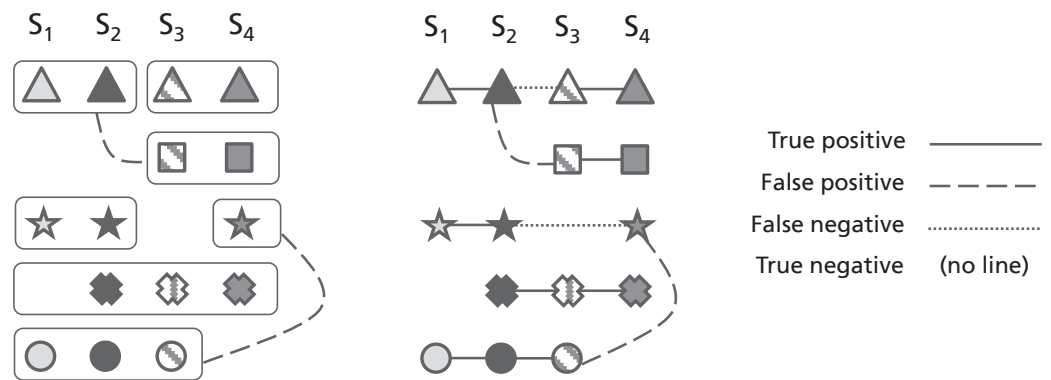


Figure 22

The incorrect result from Figure 21 (left) and its interpretation according to the merge-based analysis result definition (middle, with legend to the right).

As discussed above, any incorrect analysis result can be constructed from the correct similarity selection sets by a series of merge and split operations. Hence, as a split is just an absence of merge, these operations result in the creation of respectively one false positive (merge) or false negative (split). The number of false positives and false negatives can be therefore calculated by finding the minimal number of merges and splits necessary to convert the ideal result into the evaluated one.

The merge-based analysis result definition does not introduce the negative effects described above, and the precision and recall values calculated for comparatively worse analysis results are consistently lower (worse) than those obtained for better analysis results. Finally, the number of true positives, false positives, true negatives and false negatives is easy to calculate for any, even large, analysis result. Below, we define the respective calculation formulas. For space reasons, we omit the mathematical proofs of formula correctness.

Calculation of  
merge-based  
evaluation  
measures

Let's denote  $R$  as the number of all relevant elements, and  $NR$  as the number of all not relevant elements. Obviously,  $R = TP + FN$  and  $NR = FP + TN$  (see Figure 20). Furthermore, let's denote  $U$  as the cardinality of the union of all analyzed input sets  $\underline{US}_v$ , and  $S_{REF}$  as the number of similarity selection set in the ideal, reference solution. Finally, let's denote  $S_{SOL}$  as the number of created similarity sets in the tested solution, and  $P_{SOL}$  as the number of all selection set parts from the ideal solution that are found in the tested solution.

Measure	Description	Calculation Formula	Value in Figure 22
$U$	Cardinality of $\underline{US}_v$	Count set elements	15
$S_{REF}$	Sim. sets in reference	Count similarity sets	5
$S_{SOL}$	Sim. sets in solution	Count similarity sets	5
$P_{SOL}$	Ref. set parts in solution	Count set parts	7
$R$	Relevant elements	$R = U - S_{REF}$	10
$TP$	True positives	$TP = U - P_{SOL}$	8
$FN$	False negatives	$FN = R - TP = P_{SOL} - S_{REF}$	2
$FP$	False positives	$FP = P_{SOL} - S_{SOL}$	2
$MAX\_VAR$	Maximum size of a set representing a variant	Count set elements, select the largest set size	4
$MAX\_SIMREF$	Maximum size of a selection set from the reference solution	Count set elements, select the largest set size	4
$NR$	Not relevant elements	$U - MAX(MAX\_VAR, MAX\_SIMREF)$	11
$TN$	True negatives	$TN = NR - FP$	9
Precision in Figure 19		See Definition 13	1.0
Recall in Figure 19		See Definition 14	1.0
Precision in Figure 22		See Definition 13	0.8
Recall in Figure 22		See Definition 14	0.8

Table 6

The calculation formulas and example values for the measures used in the merge-based precision and recall definition

All these values are easy to determine by simply counting the number of respective objects in the ideal or tested solution, and they are sufficient for calculation of the precision and recall measures. Table 6 lists the calculation formulas for the introduced values, provides their values for the example presented in Figure 22, and lists further calculations of true negatives as well as the precision and recall measures for the ideal and the example solution. As the merge-based values of positive and negative analysis results are easier to interpret and easier to calculate than the values based on the pairwise similarity definition, we use the merge-based definition in the evaluation of our approach discussed in Chapter 7.

## 4.6 The Conceptual Model of Variant Similarity Analysis

The role of the conceptual model In the previous sections we characterized the specific properties of software variants and created the basis for defining and evaluating reverse engineering techniques for analyzing variant similarity. In this section, we describe the conceptual model that systematizes and interrelates the concepts used in the similarity analysis of software variants. The purpose of the model is to provide a “big picture” overview of the variant similarity analysis problem.

The presented conceptual model is general, as it applies to various analyzes of similarity performed on various asset types. Hence, for many concepts described in the model an inheritance hierarchy is defined, listing some of the specific manifestations of the given concept. The lists of specific manifestations are not intended to be exhaustive – their role is to present the breadth of the spectrum of possible choices.

We describe the conceptual model in three subsections, related to the **Structure** of the analyzed software asset, the concept of **Similarity** defined on this structure, and to the **Analysis** itself. The complete conceptual model is depicted in Figure 23.

### 4.6.1 Software Asset Structure

Software asset A *Software Asset* can exist in many *Variants* (see Definition 12), which are the object of the variant similarity analysis. A *Software Asset* consists of *Asset Elements*, which might in turn be composed from further *Asset Elements*. The *Software Asset* can in a specific case be a *Code Asset* (e.g. an implementation of a software system), a *Design Asset* (e.g. a model of that system) or a *Specification Asset* (e.g. a description of system requirements). Depending on the type of the *Software Asset*, the contained *Asset Elements* can be implemented as *Code Elements* (*Packages*, *Compilation Units*, *Classes*, *Methods*, and *Tokens*), *Textual Elements* (*Folders*, *Files*, *Line Blocks*, and *Lines*) or *Model Elements* (*Model Nodes*, *Model Links* and *Model Attributes*).





It is important to note here that the type assumed for a specific *Asset Element* depends not only on its intrinsic properties, but is also a function of the performed analysis – for example, in some cases the same asset can be treated as *Code Element* (e.g. a *Method*) or a *Textual Element* (e.g. a *Line Block*), depending on the specific analysis goal.

Assets in the structure hierarchy	A <i>Structure Hierarchy</i> provides a mechanism for grouping the <i>Asset Elements</i> . Frequently, the <i>Structure Hierarchy</i> is defined based on the containment hierarchy of the <i>Asset Elements</i> – however, definitions based on their other properties are possible. If for a given <i>Asset Element</i> there exist some subordinate <i>Asset Elements</i> in the <i>Structure Hierarchy</i> , we call such an element <i>structural</i> – consequently, an element having no subordinates is <i>atomic</i> . The place that an element occupies in the <i>Structure Hierarchy</i> is characterized by its <i>Location</i> .
Asset content	Finally, an <i>Asset Element</i> contains its <i>Content</i> . The specific form of the content strongly depends on the type of <i>Asset Element</i> and its role in the <i>Structure Hierarchy</i> : for <i>atomic</i> elements the <i>Content</i> can frequently be characterized as one or a group of primitive data type values (e.g. a textual string for a <i>Line</i> , a numeric value for a <i>Model Attribute</i> ), while for <i>structural</i> elements the <i>Content</i> is usually composed from the subordinate <i>Asset Elements</i> and, recursively, their own <i>Content</i> .

#### 4.6.2 Software Asset Similarity

Similarity	The <i>Similarity</i> between a group of <i>Software Assets</i> relates to the <i>Content</i> and <i>Structure Hierarchy</i> of the <i>Asset Elements</i> placed in these assets. <i>Similarity</i> is an abstract concept which we define as:
------------	--

Definition 15	<p>Similarity</p> <p><i>Similarity between two or more objects is a relation of sharing common properties. Similarity is gradual and relates to commonalities and differences between objects: the more commonality the objects share, the higher their similarity; the more differences the objects have, the lower their similarity.</i></p>
---------------	--

The above definition is necessarily abstract – in a concrete case, *Similarity* needs to be further defined for the purpose of a particular *Similarity Analysis*. Specifically, a *Similarity Analysis* needs to define which objects it can analyze, which properties of these objects are relevant for characterizing their *Similarity*, and what corresponding values or value classes of these properties are accounted for as a commonality or a difference.



Identity	<p><i>Identity</i> is the maximal <i>Similarity</i> between a group of objects: the objects are considered identical if all their relevant properties are common and none of the properties is different. Consequently, an object is always identical to itself. Depending on the choice of the object properties relevant for the analysis, the <i>Identity</i> can be defined as <i>Complete Identity</i> (all object properties are analyzed), <i>Identity except Formatting</i> (the syntactic formatting and visual layout of the asset content is not considered), <i>Identity of Formal Definition</i> (only the properties directly relevant to the purpose of the asset are considered, and others, e.g. comments or notes, are omitted), <i>Identity of Result</i> (e.g. the behavior specified by the asset content is identical, even if the form of its specification is not), or others.</p>
Similarity with deviation	<p><i>Similarity with Deviation</i> is a kind of <i>Similarity</i> weaker than <i>Identity</i>, as differences in the analyzable properties are to a certain degree allowed. Usually, a similarity threshold is defined that specifies the maximal degree of difference that is still acceptable for the objects to be considered similar. The differences might be allowed to exist for all types of object properties (resulting in <i>Similarity with Non-Systematic Deviation</i>) or only for certain property types or for properties fulfilling specific conditions (resulting in <i>Similarity with Systematic Deviation</i>).</p> <p>A <i>Similarity Analysis</i> can use different similarity definitions for different <i>Asset Elements</i>. For example, <i>Identity</i> can in the practice only be expected for rather small <i>Asset Elements</i>, such as e.g. <i>Tokens</i> and <i>Methods</i>, while larger elements such as <i>Packages</i> and whole <i>Software Assets</i> are rarely identical, but still can exhibit <i>Similarity with Deviation</i>.</p>
Similarity measures	<p><i>Similarity</i> can be quantified by a <i>Similarity Measure</i> which is a numeric value assigned to the analyzed objects. If the measure only applies to two objects, we call it a <i>Pairwise Similarity Measure</i>. Frequently, such measure is calculated by a distance function which places the analyzed objects in a metric space [Santini 1999]. Other <i>Pairwise Similarity Measures</i> are confidence and correlation measures (e.g. Pearson correlation [Rodgers 1988]). If more than two objects are considered, we call such measure a <i>Multi-Object Similarity Measure</i>. Example <i>Multi-Object Similarity Measures</i> are dispersion metrics (e.g. standard deviation) and commonality metrics (quantifying the properties common to all considered objects).</p>
Similarity computation	<p>While a <i>Similarity Measure</i> represents a value quantifying the <i>Similarity</i>, a <i>Similarity Computation</i> is an algorithm or formula used to calculate that value. An <i>Elementary Similarity Computation</i> calculates the <i>Similarity</i> based on the immediate properties of the analyzed <i>Asset Elements</i> derived from its <i>Content</i> and <i>Location</i>. An <i>Aggregating Similarity</i></p>

*Computation* operates additionally on the *Structure Hierarchy* of the *Asset Elements* to calculate *Similarity* for structural elements by aggregating the similarity values of their subordinate elements. For example, a *Similarity Analysis* might use the Levenshtein distance as an *Elementary Similarity Computation* to calculate *Similarity* of any two *Files*. However, as Levenshtein distance is not defined for *Folders*, a further *Aggregating Similarity Computation* needs to be defined that specifies which *File* similarities should be used for *Folder* similarity calculation (i.e. if all the possible pairs of *Files*, or only some of them, should be considered), which formula should be used to calculate the *Folder* similarity from the selected *File* similarity values, and how to calculate similarity for nested hierarchies of *Folders*.

#### 4.6.3 Similarity Analysis of Software Asset Variants

The role of similarity analysis	As described in the introduction to this Chapter, the purpose of performing a <i>Similarity Analysis</i> on <i>Variants</i> of a <i>Software Asset</i> is to recover the <i>Similarity Information</i> from these assets. For that purpose, the <i>Similarity Analysis</i> uses the <i>Similarity Measures</i> calculated for <i>Asset Elements</i> to create the <i>Similarity Information</i> , for example in the form of <i>Numerals</i> (e.g. values of a particular <i>Similarity Measure</i> ), <i>Aggregations</i> (created by grouping of the <i>Asset Elements</i> or of the <i>Information</i> units describing them), <i>Associations</i> (relating some <i>Asset Elements</i> or <i>Information</i> units to each other) and <i>Classifications</i> (assigning the <i>Asset Elements</i> or the <i>Information</i> units describing them to predefined abstract categories). As the <i>Similarity Analysis</i> is defined with respect to certain <i>Analysis Goals</i> , derived from an <i>Application Scenario</i> , the resulting <i>Similarity Information</i> is also interpreted with regard to these goals.
Consequences of application scenario selection	The <i>Application Scenario</i> concerning the analyzed <i>Variants</i> plays a key role in the similarity analysis process: by providing the <i>Analysis Goals</i> for the <i>Similarity Analysis</i> , it defines the form and scope of the required <i>Similarity Information</i> . Consequently, the type of the analyzed <i>Similarity</i> and the proper <i>Similarity Computations</i> and <i>Similarity Measures</i> are selected and defined accordingly to provide that <i>Information</i> . The selection of the type of sought <i>Similarity</i> might also result in processing the same analyzed <i>Asset Elements</i> in different ways, for example as <i>Textual Elements</i> or <i>Code Elements</i> . Hence, two <i>Similarity Analyses</i> performed on the same <i>Software Assets</i> but targeting different <i>Application Scenarios</i> might differ in their use of the particular <i>Asset Elements</i> , in the provided <i>Similarity Information</i> , as well as in the definition of any of the underlying concepts of <i>Similarity</i> , <i>Similarity Computations</i> and <i>Similarity Measures</i> .

## 4.7 Summary

In this chapter we defined the foundations for the variant similarity analysis approach described in this thesis. We started by discussing the inherent properties of software variants, such as the lack of an objectively definable variant ordering. Based on these properties and the application scenarios described in Section 3.2, we defined the construction requirements applicable to any variant similarity analysis technique, which specify the necessary order independence of the analysis, the information content of the results, and the need for user-supporting result presentation. These requirements form the foundation of our approach, further defined in Chapter 5, and can also be used to reason about other, related analysis approaches. Subsequently, we derived a number of further assumptions applying specifically to our approach, such as the preference for high result certainty and result transitivity, which we motivated by the stated application scenarios.

Based on the construction requirements and the defined assumptions, in Section 4.4 we provided a formal definition of a similarity analysis approach. Subsequently, in Section 4.5 we discussed the method for evaluating the result quality of the approach by using the information retrieval measures of precision and recall and introducing a set merge based definition of the correct analysis result. Finally, in Section 4.6 we systematized the concepts used in similarity analysis of software variants by providing a conceptual model of software assets, their similarity, and the analysis of that similarity. Hence, this chapter provided the foundation for reasoning about the similarity analysis problem and for defining and evaluating its possible solutions in the scope of the selected application scenarios.

## 5 Variant Similarity Analysis with Hierarchical Set Similarity Models

As motivated in the previous chapters, an analysis of a potentially large number of system variants, each of them having a potentially large code basis, is frequently required in the practice. Despite the analysis difficulty, resulting from the ordering independence of variants, the similarity analysis results should provide a sufficient detail of information and be available on any granularity level in the system structure hierarchy. Hence, the defined application scenarios pose significant requirements on detailed data collection, suitable result model structuring and understandable and supportive result presentation. In this chapter we describe the concepts, models, algorithms and visualizations constituting our variant code similarity analysis approach, aimed at satisfying these requirements. The approach is based on the idea of hierarchical set similarity models and is defined in compliance with the construction requirements, assumptions and formalizations presented in Chapter 4.

Using the analysis result formalization defined in Chapter 4, we derive in Section 5.1 the basic idea of the set similarity model and discuss its properties, such as the availability of similarity information for any combination of the input variants. In Section 5.2 we integrate the set model with the concept of a structure hierarchy, which together define the hierarchical set similarity model and enable the provision of similarity information of various granularity levels. On this basis, in Section 5.3 we discuss the activities needed for hierarchical set model construction and motivate the analysis process followed in our approach. Afterwards, in Section 5.4 we proceed to the description of analysis algorithms used for mapping corresponding elements between the structure hierarchies and for the construction of set model based on input similarity data. In Section 5.5, we introduce the visualization concepts defined for presenting the set model information across the available variant combinations and the system structure hierarchy, and discuss the properties of these visualizations. In Section 5.6 we describe various metrics calculated on the created set model, which further detail the analysis results. Finally, in Section 5.7 we summarize the complete analysis results, present the advantages and limitations of our approach, and discuss the fulfillment of the previously stated requirements. Subsequently, in Chapter 6 we describe a few implementation mechanisms for the analysis approach, and in Chapter 7 we evaluate a subset of the approach contributions.

## 5.1 The Set Similarity Model

Properties of asset variants content	<p>In Chapter 4, we introduced several properties characterizing the content of the asset variants for the purpose of our similarity analysis approach. In particular, we postulated that:</p> <ul style="list-style-type: none"><li>• An asset variant can be treated as a set containing the asset content elements, which are the atomic units of the analysis.</li><li>• (Assumption A1) The analysis only considers the similarities between different asset variants, while the similarities inside a variant are ignored. Hence, the set of variant content elements is a proper set and not a multiset, because any two elements of that set are considered to be different.</li><li>• (Assumption A3) An asset element should have at most one counterpart element, recognized as similar, in every of the other variants. The analysis should therefore establish one-to-one correspondences between the elements of any analyzed variant pair. In case there are more than one suitable candidate elements in the counterpart variant, the analysis should provide criteria to select the most suitable candidate and discard the others.</li><li>• (Assumption A5) If an asset element is recognized as similar to any two (or more) other elements, these other elements should also be recognized as similar to each other. In other words, the analysis should deliver transitive similarity results.</li><li>• In result of the above statements, the similarity analysis should, for any given element of any variant, identify zero or one similar elements respectively from each other variant.</li><li>• The relation of similarity, defined on the asset content elements, is reflexive (an element is similar to itself) and symmetric (which results from the requirement C1). As it is also transitive, it is consequently an equivalence relation.</li><li>• The group of identified similar elements, which are in equivalence relation to each other, is an equivalence class. Hence, the analysis assigns the content elements of the input variants to equivalence classes based on the defined similarity relation, so that every element belongs to exactly one such class.</li></ul>
Interpretation of an equivalence class	<p>All elements assigned to a given equivalence class, constructed by the analysis, are identified as similar to each other. Furthermore, each of these elements belongs to a different input asset variant. Hence, all member elements belonging to the same equivalence class can be treated as variant manifestations of a single, abstract content element, which is possibly instantiated with a slightly different content in the respective asset variants. In other words, the proposed interpretation of the analysis result is that the asset variants containing the elements of the equivalence class actually contain variants of a single, abstract content element. The abstract element and the variants are equivalent, that is, they can only differ up to the degree allowed by the used similarity equivalence relation.</p>

**Equivalence sets** Consider a group of sets  $S_1, S_2, \dots, S_N$ , containing the atomic content elements of  $N$  input variant assets, and an equivalence relation  $\equiv$ , defined on the elements of the sets  $S_1, S_2, \dots, S_N$  according to the above discussion, i.e., such that no two elements belonging to the same set are equivalent. For each set  $S_i$ , we define a corresponding equivalence set  $S_i^E$  as follows:

**Definition 16** Equivalence set

*An equivalence set  $S_i^E$  corresponding to a set  $S_i$ , constructed using the relation  $\equiv$ , is the set of all equivalence classes of  $\equiv$  which contain an element of the input set  $S_i$ .*

The construction of equivalence sets, using shape similarity as the similarity equivalence relation, is depicted in the Figure 24. As some of the equivalence classes contain elements from more than one of the input sets  $S_1, S_2, \dots, S_N$ , and hence they belong to more than one equivalence set, the constructed equivalence sets intersect with each other. For any subgroup of the sets  $S_1, S_2, \dots, S_N$ , all content elements identified by  $\equiv$  as similar across all the subgroup sets are placed into the equivalence classes located inside the intersection of the corresponding equivalence sets (see Figure 24 right). Consequently, the remaining content elements are placed into equivalence classes located outside of that intersection.

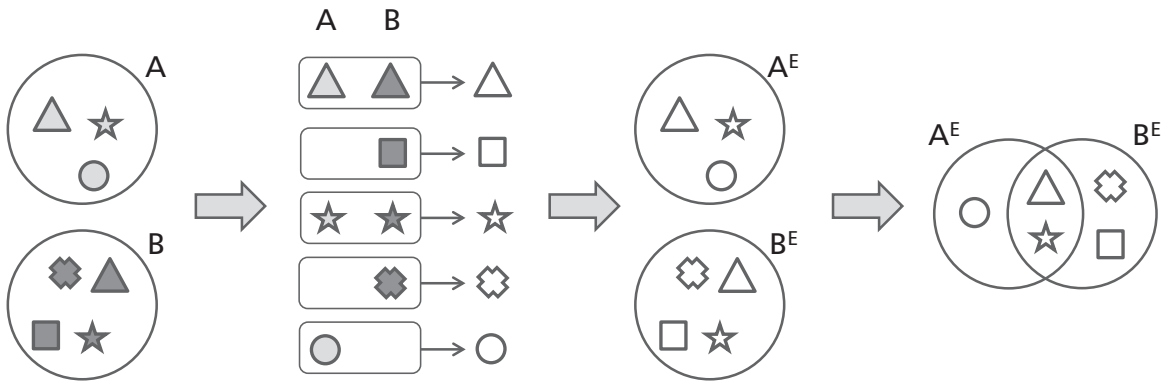


Figure 24

Equivalence sets construction: the elements of the input asset content sets are assigned to the equivalence classes (left). The resulting equivalence sets, containing these classes, intersect with each other (right).

Hence, the analysis can express the similarity of the input sets by using intersections and unions of the corresponding equivalence sets. The equivalence set intersection and union are standard set operations. Based on the sets  $S_1, S_2, \dots, S_N$  and the relation  $\equiv$ , we define them as:

**Definition 17** Equivalence set intersection

*An equivalence set intersection, constructed for any subgroup (proper or not) of the sets  $S_1, S_2, \dots, S_N$  using the relation  $\equiv$ , is the set of all equivalence classes of  $\equiv$  which for every subgroup set contain an element of that set.*

**Definition 18** Equivalence set union

*An equivalence set union, constructed for any subgroup (proper or not) of the sets  $S_1, S_2, \dots, S_N$  using the relation  $\equiv$ , is the set of all equivalence classes of  $\equiv$  which contain an element of any subgroup set.*

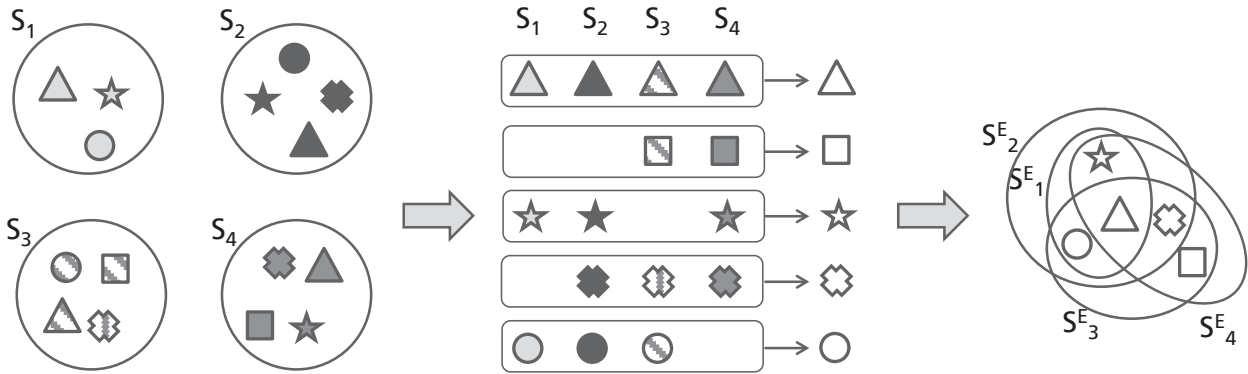


**Set similarity model** The union of equivalence sets  $S_1^E, S_2^E, \dots, S_N^E$  contains, through the equivalence classes, all content elements of the input sets  $S_1, S_2, \dots, S_N$ . Moreover, each content element is contained exactly once in the equivalence set union. The equivalence set union models the similarity of the input sets by storing the equivalence classes and presenting the intersections of any subgroup of the equivalence sets. We call this model, being the result of the similarity analysis, a set similarity model and define it as<sup>2</sup>:

**Definition 19** Set similarity model

*A set similarity model, built for the sets  $S_1, S_2, \dots, S_N$  and the equivalence relation  $\equiv$ , is the union of the equivalence sets  $S_1^E, S_2^E, \dots, S_N^E$  corresponding to the sets  $S_1, S_2, \dots, S_N$  and constructed using the relation  $\equiv$ . The set model expresses the similarity of the input sets using the equivalence classes of  $\equiv$  and the intersections of the equivalence sets.*

In Figure 25 we provide an example of a set similarity model constructed for the analysis result presented previously in Figure 19. Again, the analysis uses the shape similarity as the similarity equivalence relation. Note that the right part of Figure 24 (on the previous page) also presents a set similarity model.



**Figure 25** The input sets (left) and the analysis result from Figure 19 (middle) represented with the set similarity model (right).

Consequently, the task of the set model based similarity analysis, using the provided similarity equivalence relation, is to form the equivalence classes from elements of the input content sets and to build the set similarity model by precisely determining how the resulting equivalence sets intersect:

**Definition 20** Set model based similarity analysis

*A set model based similarity analysis treats the input variant assets as sets of comparable, atomic content elements. The analysis uses an equivalence relation, defined on the atomic content elements, to construct the equivalence sets corresponding to the input variant sets and to determine their intersections, consequently building a set similarity model.*

<sup>2</sup> In the text of selected definitions in this chapter, we underline the concepts introduced previously in earlier definitions to indicate that the respective previous definition is necessary to fully understand the currently defined concept.

The role of the equivalence relation	<p>Note that the scope of element similarities recognized by the similarity analysis fully depends on the provided equivalence relation. The equivalence relation defines which properties of the analyzed elements are relevant for their similarity, while their remaining properties are ignored. For a different relation used on the same input sets, a different analysis result might be constructed. For example, using the similarity of shape <u>and</u> color on the sets from Figure 25 would lead to zero recognized similarities. Hence, the similarity equivalence relation selected for the specific analysis should always be known during the interpretation of the analysis result.</p>
Properties of the set similarity model	<p>The described concept of a set similarity model has the following properties, relevant in the processing and interpretation of similarity analysis results, which distinguish it from the other related approaches:</p> <ul style="list-style-type: none"> <li>• For any content element of any input set, it is directly known which other sets contain equivalent content elements, and which elements these are. This information can be used to locate the elements corresponding to each other among input sets, or to determine the degree to which a given element exhibits reuse potential (e.g. by counting the sets containing the element variants).</li> <li>• For any intersection of the equivalence sets, all the content elements belonging to that intersection are directly known. For example, it is straightforward to determine the content elements which are recognized as common among all the input sets, or the content elements which are unique to a given set.</li> <li>• The cardinality of any equivalence set and of any intersection can be simply calculated by counting the contained elements. The cardinality metric can be then used to define various metrics expressing the similarity of the sets or their subsets – for instance, the similarity of two sets <math>A, B</math> can be defined as <math>card(A \cap B) / card(A \cup B)</math>.</li> <li>• The set model describes a set union of the code bases of all analyzed variant products. Hence, it enables analysis and measurement of the unified code base as a whole, without the need to assume the point of view of a concrete product variant.</li> <li>• The set model can be further analyzed by retrieving all elements fulfilling a given logical condition. The condition can be specified directly, e.g. <math>S_1 \cap S_2 \cap S_3'</math>, or indirectly, e.g. "all set intersections shared by exactly <math>k</math> sets". Such retrieval operation, which we call a <i>subset calculation</i>, has the same properties as a set intersection: it provides a group of elements which can be unambiguously identified and counted.</li> </ul>

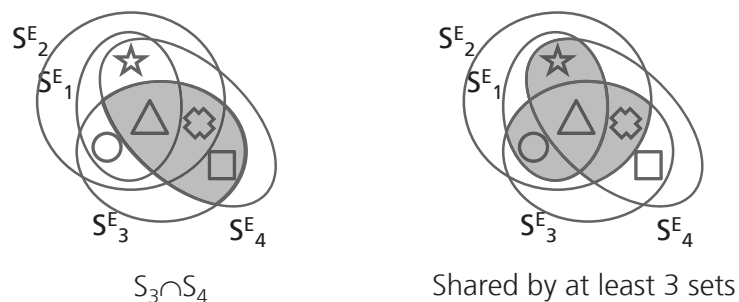


Figure 26

Example subset calculations.



- Further logical conditions, defining new subset calculations, can take as an input the results of already existing calculations and, for instance, merge or intersect the provided element groups. The subset calculations can be used to identify the elements fulfilling a specific analysis goal – for example, in the right part of Figure 26 the equivalence classes containing the elements shared by at least 3 sets, and hence possibly having a high reuse potential, are selected.
- Moreover, the results of several subset calculations can be automatically processed to answer more complex questions arising in specific analysis settings – such as finding the most similar set pairs, finding sets completely contained in other sets, clustering the input sets into strongly similar groups, finding a minimal number of sets covering almost the complete code, and further. The set model provides a convenient basis for performing such calculations – they only require the model itself, and no reanalysis of the underlying code assets is needed. In contrast to that, the results of other approaches do not directly allow for such follow-up analyses.
- The set model is an abstraction of the underlying variant products. However, it contains more information than the other existing abstraction mechanisms such as similarity metrics and pairwise matrices. Consequently, the set model provides more support in distinguishing different arrangements of similarity among the input products, as it is much less likely that different analysis inputs will result in the same analysis output (see the example in the Figure 18).

Applicability to  
diverse asset  
types

The set similarity model is simple to understand, and it has several properties, discussed above, which are supportive for a similarity analysis. It is defined in an abstract way, without specifying the exact type of analyzed variant systems and their elements. Hence, the concept of set similarity model, and the resulting further analysis mechanisms described in the remainder of this chapter, can in theory be applied to any asset type – the source code treated as text or as parse trees, software models, other software-related assets such as documentation and test cases, abstract asset descriptions such as requirement repositories, and also to any other non-software assets. However, the prerequisite to defining a set similarity model, and hence also to using the further analysis concepts, is the conformance to the stated analysis assumptions. Consequently, there are two basic analysis mechanisms which need to be thoroughly defined before the set model can be constructed for a given asset type:

- **The decomposition of the asset into atomic, comparable elements.** Optimally, the elements should represent roughly equally sized or important pieces of the asset content. Defining elements of varying importance (e.g. some elements represent single code methods, while others represent single import statements) complicates the interpretation of the analysis result, as e.g. counting the elements belonging to a given set intersection does not provide a fully reliable indication of the relative importance of that intersection anymore. If the definition of unequally important elements cannot be avoided, the

elements might be assigned positive integer weights denoting their relative importance to ensure that recognized element similarity is proportionally reflected in the resulting similarity of the whole assets.

- **The definition of an equivalence relation on these elements**, as previously discussed in this section. The relation definition might be supported by the rules for unambiguous assignment of equivalent elements to the reported equivalence classes. In the most cases, a definition of an equivalence relation which only considers the element content (e.g. element content equality) results in equivalence classes containing more than one element in some asset variants – violating the assumptions A1 (only similarities between variants are considered) and A3 (one-to-one correspondences). Hence, it is necessary to extend that basic definition of an equivalence relation with further rules, for example based on the location of the elements in the system structure, in order to decide which of the potentially equivalent element candidates should be finally reported in the analysis result. In the Section 5.4 we provide an example of such rules, which we defined for our implemented analysis instantiation.

Open problems  
for some asset  
types

Admittedly, the difficulty of defining the three stated analysis mechanisms, as well as the quality of achievable analysis results, varies strongly depending on the analyzed asset type. In Section 5.4 we define the similarity analysis on textual assets, which are decomposed to atomic lines of text and analyzed for syntactic equivalence. Analyses of more complex asset types require solving various research problems: for example, finding corresponding elements is more difficult for software models than for text files. Furthermore, different asset types may require different processing of input data in order to achieve the transitivity of similarity relation (see Section 5.4 for discussion). Finally, analysis of asset types which should be compared based on the semantics rather than syntax, such as e.g. requirement descriptions, faces the difficulty of defining an automatically evaluable equivalence function. Consequently, instantiating the set model based similarity analysis for further asset types might require prior research, and is part of the planned future work.

Further  
discussions on  
the set model

In the following sections, we define and discuss further concepts of our analysis which are based on the set similarity model and extend the possibilities of its use. In Section 5.2 we define the application of a set model in hierarchical system structures, which enables a set-based similarity assessment on any level of the system hierarchy. In Section 5.4 we present a concrete instantiation of the set model based analysis for textual software artifacts and discuss the necessary steps for input data processing and model construction. In Chapter 6 we describe ideas for performant set model implementation. Finally, the understandability of the set models, especially for a large number of sets and a large number of set elements, is discussed in Section 5.5, where we propose the respective visualization concepts. Set model understandability is also evaluated in a controlled experiment in Section 7.2.

## 5.2 Hierarchical Set Similarity Model

Tree-based  
asset content  
hierarchy

In this section we combine the idea of the set similarity model with the use of trees representing the internal hierarchical structure of the asset content. In the most cases, the internal structure of a software system or asset can be represented as a tree-based hierarchy of its constituent parts. For example, the source code is physically stored in files and organized in a tree structure of folders. Several programming languages use a logical structure of a package or namespace tree. Also in software models, a tree package structure is frequently used. Finally, a tree hierarchy of assets is constructed by many reverse engineering approaches [Lethbridge 2004]. The tree hierarchy supports decomposition of the asset into smaller parts, grouping and categorization of these parts, and easy navigation in the asset content. Depending on the specific asset type, many concrete decompositions of the asset into the structure tree and the atomic content elements can be defined. As discussed in the Section 5.1, this decomposition constitutes an important analysis mechanism, and should hence be defined to appropriately support the analysis goals. In this section we define the data model expressing the decomposition and the similarity of asset content, while in Section 5.4 we provide an example decomposition for source code directories containing textual code files.

### 5.2.1 Asset Content Trees and Set Similarity Models

A metamodel  
of the asset  
content tree

A metamodel of the asset content tree is presented in Figure 27. The metamodel defines the decomposition of the asset into a tree of *Structure Tree Elements*. In the tree structure, we distinguish between *Structural Containers* (e.g. folders, packages), which can store other *Structure Tree Elements*, and *Content-Filled Elements* (e.g. files, classes) which can only be located at the tree leaves. A *Content-Filled Element* can store a set of *Atomic Content Elements*, i.e. the basic analyzable elements the asset is decomposed to.

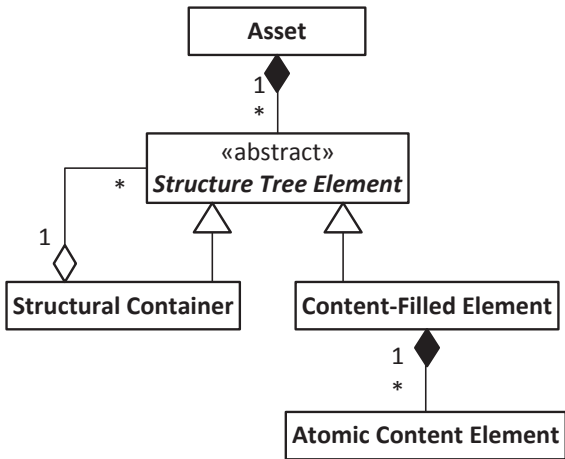


Figure 27

A metamodel of the tree-based structure of asset content hierarchy.

Asset content  
tree

According to the definition of set model based similarity analysis (Definition 20), an asset is treated as a set of atomic content elements. Consequently, for a given set of asset content elements  $S$  we define the asset content tree  $T$  as:

Definition 21 Asset content tree

An asset content tree for a given set  $S$  is a tree graph  $T$ , in which the tree leaves reference non-overlapping subsets of the set  $S$ , and the union of all these subsets is equal to the set  $S$ .

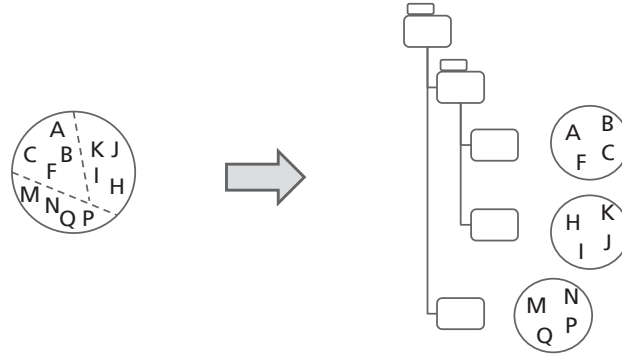


Figure 28 A set of asset content elements and an asset content tree.

An example asset content tree is presented in Figure 28. The asset content tree specifies the internal asset structure, defines a partition of the asset content set, and assigns the content subsets created by the partition to the tree leaves (*Content-Filled Elements*). Moreover, by using the principle of hierarchical aggregation, any parent node in the tree (i.e., a *Structural Container*) can also be associated with a subset of the original asset content set. The parent subset is simply the union of all leaf subsets existing in the subtree rooted by the selected parent node.

Equivalence relation on asset content tree elements

The analysis assumptions, described in the Section 4.3, prescribe that the similarity of the Structure Tree Elements located in the different analyzed asset variants should be expressed using one-to-one element correspondences. Furthermore, these correspondences are subject to the same conditions as in the case of Atomic Content Elements: no two elements of the same tree are similar to each other, and their similarity relations are transitive. Hence, the correspondence identification defines an equivalence relation on the Structural Tree Elements and assigns the elements of the input trees to equivalence classes.

Unified asset content tree

For a group of input asset content trees  $T_1, T_2, \dots, T_N$ , and an equivalence relation  $\approx$  defined on the elements of these trees such that no two elements of the same tree are equivalent, we construct an unified asset content tree. The unified asset content tree represents the structure of all input content trees, expresses the correspondences between the tree elements, but does not yet contain the asset content set elements:

Definition 22 Unified asset content tree

An unified asset content tree built for the input trees  $T_1, T_2, \dots, T_N$  and the equivalence relation  $\approx$ , is a tree structure containing the equivalence classes of the input tree elements constructed using the relation  $\approx$ .

The construction principle of the unified asset content tree is analogical to the creation of a set model from the asset content sets. The Structure Tree Elements of the input asset variants are placed into the equivalence classes of the unified asset content tree in the same way as the Atomic Content Elements of these variants are placed into the equivalence classes of the content set similarity model.

Unified asset content tree construction

Let’s analyze three example variants of a software asset, represented in Figure 29 by their asset content trees. The example equivalence relation of the analyzed Structure Tree Elements is defined according to their location in the asset structure tree, and indicated by the respective Greek symbols. Note that as the variant structure trees are different, not all Structure Tree Elements have a correspondence in each other variant (see for example the element  $\Sigma$ ). Subsequently, a unified asset content tree, covering all Structural Tree Elements from all asset variants, is constructed. The unified asset content tree contains an equivalence class for any Structure Tree Element which exists in at least one input tree variant.

Assigning set models to the leaf unified tree elements

Subsequently, for each identified group of corresponding Content-Filled Elements, a set model is constructed from their Atomic Content Elements, as depicted in the right part of Figure 29. In that process, the content element equivalence relation  $\equiv$  (introduced in the Section 5.1) constructs equivalence classes from elements of content sets which are assigned to the elements of the same tree element equivalence class. After the set model is constructed, each element of the unified asset content tree representing the Content-Filled Elements references a related set model built from Atomic Content Elements. Consequently, the unified asset content tree and the set models represent the full analysis result, providing an overview of all asset variants, their elements, and the identified similarity.

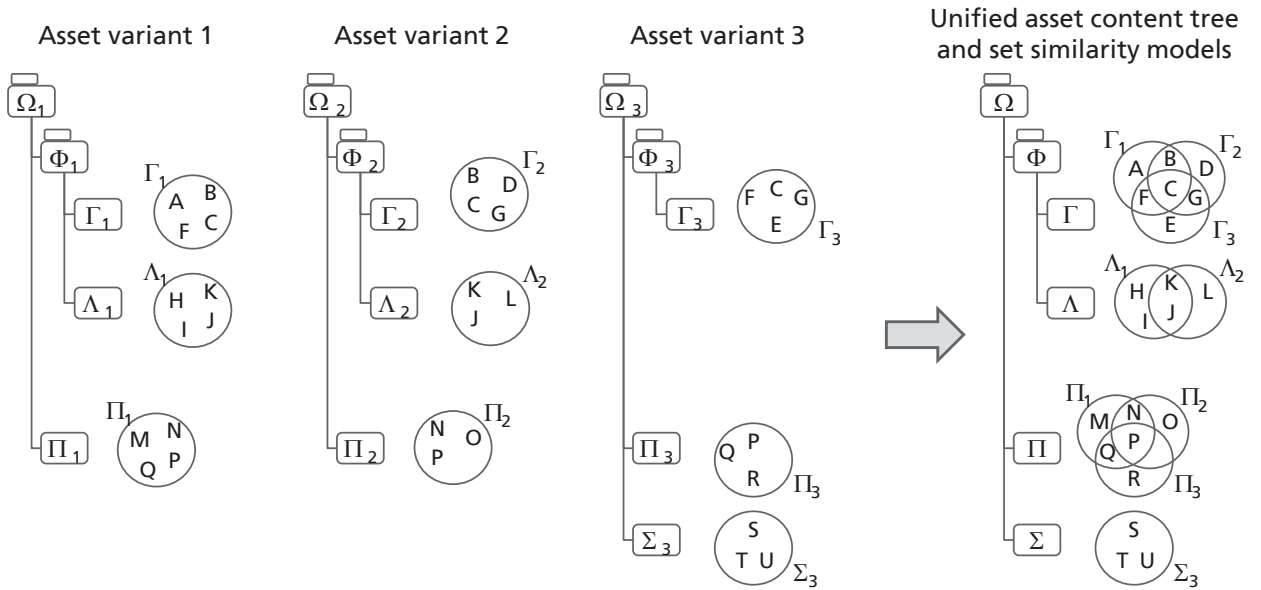


Figure 29 Three assets are decomposed into asset content trees, with atomic content elements stored in the tree leaves (left, middle). A unified asset content tree is constructed from the asset content trees, and its elements reference the set similarity models of atomic content elements (right).

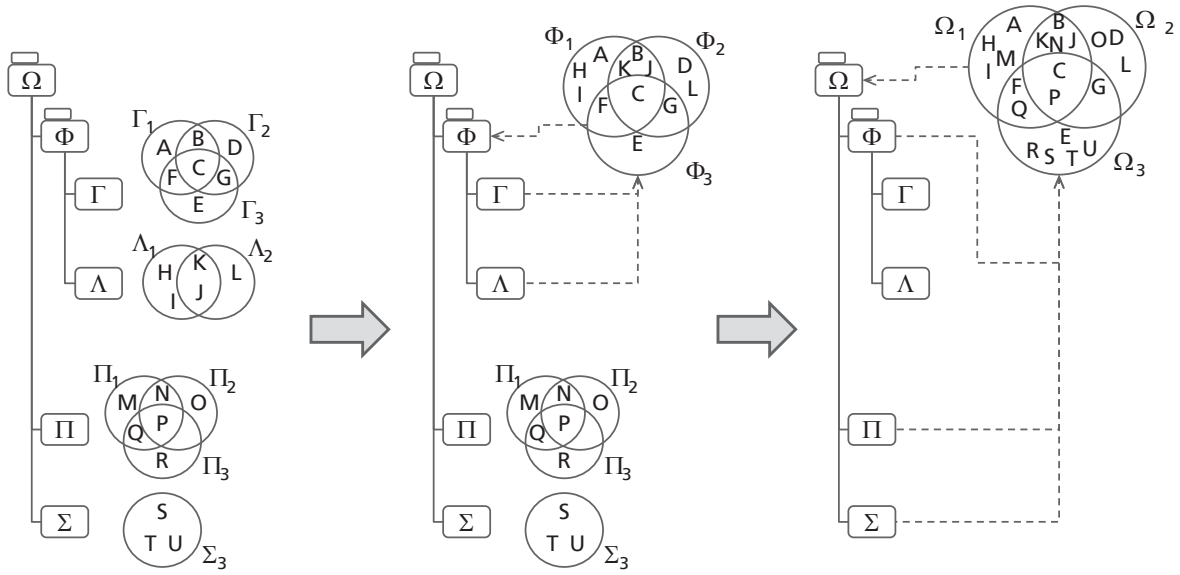


Figure 30 The construction of set models for parent elements of the unified asset content tree based on the child element set models.

Constructing set models for parent tree elements

Finally, the set models of the unified asset content tree elements representing the Structural Containers can be constructed based on the set models of their contained Content-Filled Elements. The construction process, depicted in Figure 30, follows two simple principles. First, the set model of a given Structural Container contains all Atomic Content Elements from the Content-Filled Elements located inside it. Second, the membership of any Atomic Element in the respective set intersection is permanent, that is, in the parent sets representing the variant Structural Containers the Element still belongs to the same set intersection (i.e., the intersection expressing the same logical condition on asset membership, e.g.  $S_1 \cap S_2 \cap S_3$ ). Hence, the logically equivalent set intersections created for the child tree elements can be simply added to obtain the respective intersection for the parent tree element.

Figure 30 presents the construction of the parent set models for the intermediate Structural Container and for the root Structural Container, hence building the set model for the tree root representing the complete analyzed asset. Consequently, for each element of the unified asset content tree, representing either an equivalence class of Content-Filled Elements or of Structural Containers, a set similarity model expressing the similarity of all contained Atomic Content Elements is always available.

Set model for Structure Tree Elements

Another interesting property of the unified asset content tree is that the tree fulfills all conditions necessary to create a set similarity model from its elements (nodes). Hence, given a group of sets  $S_{T1}, S_{T2}, \dots, S_{TN}$  containing the elements of the trees  $T_1, T_2, \dots, T_N$  and the equivalence relation  $\sim$ , a second type of a set similarity model can be constructed. The new model expresses the similarity of the given trees, but not the similarity of the contained atomic content elements.



The same set model construction principle can be applied to any group of corresponding subtrees of the given trees  $T_1, T_2, \dots, T_N$ , hence creating a set similarity model for these subtrees. Consequently, for each element of the unified asset content tree a set similarity model expressing the similarity of the contained Structure Tree Elements is available. A set similarity model of Structure Tree Elements can be analyzed and processed in the same way as a set model of Atomic Content Elements, including the computation of subset calculations and the use of set visualizations.

**Hierarchical set similarity model** With all the above observations, we finally define the hierarchical set similarity model. That model is a fundamental structure used by our similarity analysis approach and enables the definition of the analysis mechanisms described in the remainder of this Chapter:

**Definition 23** Hierarchical set similarity model

*Given a group of analyzed assets, a hierarchical set similarity model for these assets is the unified asset content tree, where each node of the tree references two set similarity models. These two set similarity models express respectively the similarity of the Structure Tree Elements and of the Atomic Content Elements located in the subtree of the unified asset content tree rooted by the given node.*

Accordingly, the term “hierarchical set similarity model” refers to the presence of a hierarchy of set similarity models on the different granularity levels of the asset content decomposition, as exemplified in Figure 30. The construction of the hierarchical set similarity model for a given group of assets requires that the asset content trees, the tree element equivalence relation  $\approx$ , and the atomic content element equivalence relation  $\equiv$  are defined.

### 5.2.2 A Data Model for the Set Model Based Similarity Analysis

**Associating the asset structure model with set models** Based on the association between the unified content tree elements and the set models constructed for the Atomic Content Elements and the Structure Tree Elements, we extend the metamodel of asset structure hierarchy, defined previously in Figure 27, by adding the asset similarity information. Figure 31 presents the resulting metamodel of the data used in our analysis.

The metamodel is subdivided into the asset content structure metamodel (left) and the asset content element similarity metamodel in the form of two *Set Models* (right): one for the *Structure Tree Elements* of the variant assets, and another one for their *Atomic Content Elements*. Naturally, a *Set Model* is associated with all analyzed asset variants. The similarity information is stored in the *Equivalence Class* objects, which associate the content elements recognized by the analysis as equivalent. Consequently, the metamodel specifies the data structure of the asset content tree and the two respective equivalence relations, which is sufficient for the construction of the unified asset content tree and the set similarity models for each tree element. Hence, the presented metamodel fully specifies the hierarchical set similarity model.



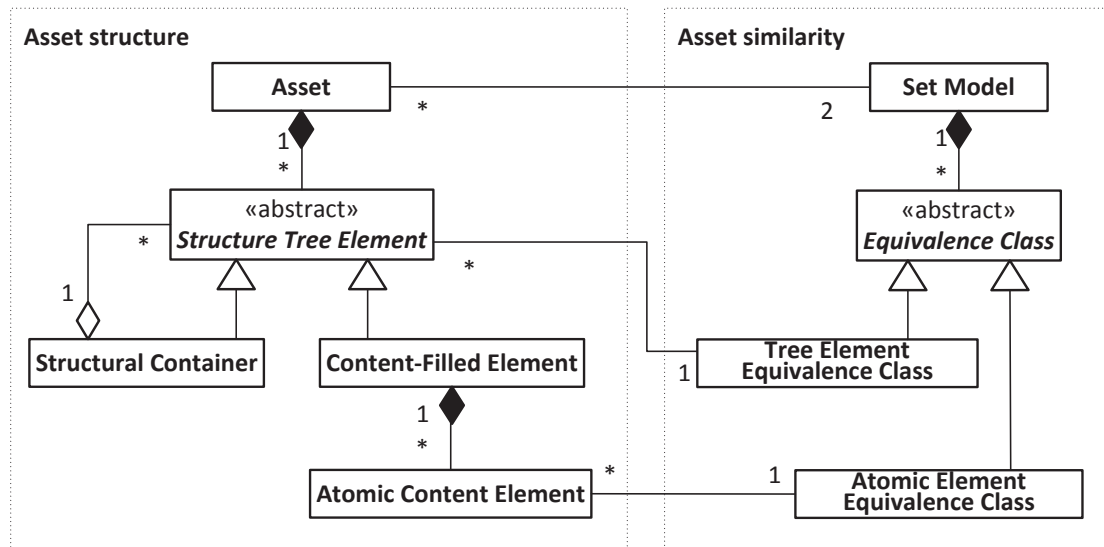


Figure 31 A data metamodel for the similarity analysis: the content structure of each asset (left) is associated with two set models (right), which store the asset content similarity information.

### 5.2.3 Representation of Equivalence Between Transformed Tree Structures

**Correspondence identification between tree structures** The construction of the unified asset content tree and the hierarchical set similarity model, described above, can be performed for any arrangement of identified equivalences between the variant Content-Filled Elements. For description simplicity, the above example used the tree location equivalence – however, in the practice the asset content parts represented by the Content-Filled Elements can be renamed, moved to another Structural Container, and otherwise transformed during the evolution of the particular asset variant. Despite the transformations, the equivalence of the respective elements should still be recognized. In this section, we describe the representation of such equivalences in the hierarchical set similarity model, while in Section 5.4 we discuss the possible approaches to the equivalence identification.

**Unified tree construction for moved tree elements** The construction of the unified asset content tree for the location identity approach is simple, as the unified tree is just a supertree of the input asset content trees. In the other case, when a group of differently located elements is recognized as equivalent, each of their respective parent elements existing in the unified asset content tree should still be able to contain all its child elements. To preserve that property, we use the concept of hard links, commonly known in the file systems: a given equivalence class of Structure Tree Elements can be referenced by more than one location in the tree, and each of the referencing locations is treated as a full representation of the given class. Hence, for the equivalent elements having different tree locations, a hard link is inserted in the unified tree at each location where one of the original elements existed, and all these links refer to the same equivalence class of the Structure Tree Elements and to the same Atomic Content Element set similarity model. Consequently, also in the general case the unified asset content tree is constructed as a supertree of the input asset content trees – however, some of the supertree elements are replaced by hard links in order to appropriately reflect the constructed equivalence relation.

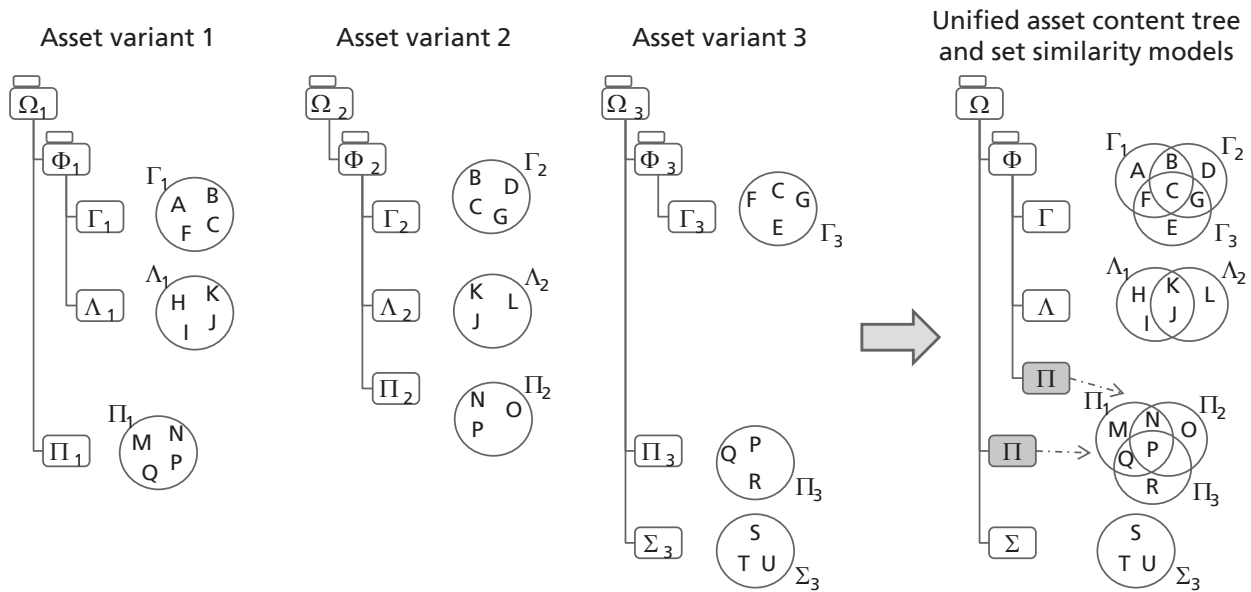


Figure 32 Three assets content trees from Figure 29 (left), with the node  $\Pi$  moved inside the parent node  $\Phi$  in the second variant, are used to construct a unified asset content tree. The tree (right) contains hard links in all original locations of the node  $\Pi$ , which reference the same set similarity model.

An example unified asset content tree using hard links is presented in the right part of Figure 32 (above). The figure shows the construction of the unified tree for the asset content trees which were introduced before in Figure 29 – however, the second asset variant tree is modified as the Content-Filled Element  $\Pi$  is moved into the Structural Container  $\Phi$ . The element  $\Pi$  exists in two different locations in the original asset content trees. Consequently, two hard links are created in the resulting unified asset content tree, and both those links represent the same equivalence class of Content Filled Elements and reference the same set similarity model.

Structural  
Container  
equivalence

When the equivalent Content-Filled Elements are hard-linked with each other, the hard links are not needed for Structural Containers, as it is sufficient to express the equivalence of the Structural Containers based on their tree location identity:

- In case a more complex equivalence relation of the differently located Structural Containers was used, the identified containers would need to be represented as hard links in each of their original locations in the unified asset content tree. However, as only the Content-Filled Elements can contain the Atomic Content Elements of the analyzed asset, the same effect of Container representation can be achieved by accordingly creating the hard links of the child Content-Filled Elements wherever necessary, and leaving the non-linked Containers in each of their original locations.
- Moreover, avoiding hard Container links eliminates the possibility to introduce cycles to the tree hierarchy structure, which would complicate the processing of the hierarchical set similarity model.

- Finally, using the Container equivalence relation based on the tree location identity simplifies the respective equivalence algorithms, as only the Content-Filled Element equivalence needs to be found.

Hierarchical aggregation in the presence of hard links

To conclude this discussion on the tree element equivalence, let's consider the hierarchical aggregation of the set similarity models in a unified asset content tree containing hard links. To obtain a correct set model, the aggregation can follow one of two strategies, presented in Figure 33:

- **The exclusive aggregation strategy** builds the set similarity model of the parent tree element only from the models of these child elements which are a child of the given parent node in each of their variants. Hence, only non-linked elements and these of the hard linked elements which only occur in the subtree rooted by the given node are considered. The remaining hard linked elements, whose instances also occur outside of the considered subtree, are excluded.
- **The inclusive aggregation strategy** differs from the exclusive strategy by including the set models referenced by the hard linked elements having further instances outside of the considered subtree. Hence, in the inclusive strategy the set similarity model is built from the models of these child elements which are a child of the given parent node in at least one of their variants. Consequently, the models of every contained non-linked element and every contained hard linked element are included.

Naturally, for each group of hard links referencing the same set similarity model the given model is considered in the aggregation just once. Furthermore, note that the result of both strategies is identical for the root tree element, because as the root element is the parent of every other tree element, no further hard link instances exist outside of its subtree.

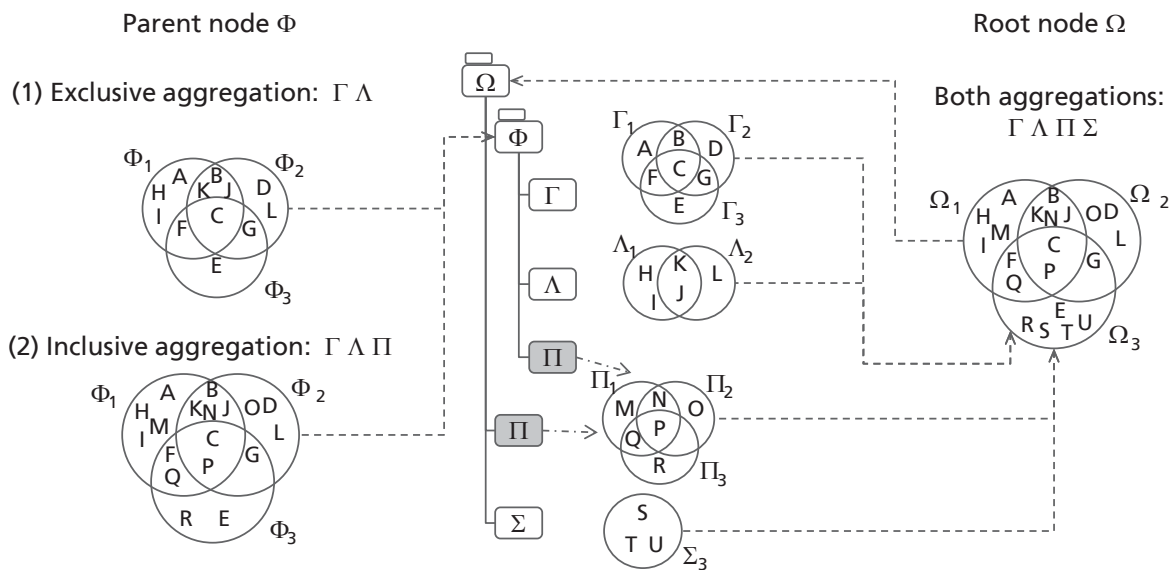


Figure 33

The construction of set similarity models for the parent node  $\Phi$  (left) and for the root node  $\Omega$  (right) of the unified asset content tree from Figure 32.

### 5.2.4 Further Remarks on the Hierarchical Set Similarity Model

The use of the mathematical concepts	The hierarchical set similarity model bases on several concepts commonly used in the mathematics and its applications, such as element sets, equivalence relations, tree graph structures, and tree-based hierarchical data aggregation. Furthermore, the concept of an equivalence set, defined in Section 5.1, bases on a well-known concept of a quotient set and extends it for the case of a specific type of equivalence relation defined on many element sets instead of just one set.
Originality of the contribution	However, the introduced composition of the existing mathematical concepts, resulting in the definition of the hierarchical set similarity model, is to the best of our knowledge an original contribution of this thesis, as no similar model structure was defined or used in other published research works. Based on a survey of related approaches, we are confident that no analogical structure was defined or used for the purpose of reverse engineering software similarity, which is the application area of this thesis. We are also not aware of the existence of such structure in other areas of computer science. Finally, the existing data structures which use similar names, such as “hierarchical sets” or “nested set model” [Kamfonas 1992], describe in fact a different, much simpler structure of an element set and its contained subsets, analogical to the asset content tree.
Implications of the model for the analysis process	As discussed in this section, the construction of the hierarchical set similarity model requires inputs provided by three other activities: the decomposition of the assets into the asset content trees, the construction of unified asset content tree using Structure Tree Element equivalence relation, and the construction of the Atomic Content Element set similarity models. Hence, the process of performing the similarity analysis needs to include these three analysis stages. With this observation, we continue to Section 5.3, where we define the analysis process of our approach.

## 5.3 A Process for Hierarchical Set Model Construction and Usage

The process for set model based similarity analysis	As discussed in Section 2.2, the purpose of reverse engineering is to construct higher-level abstractions of the input data stored in the analyzed assets, and to visualize these abstractions to support a human in building asset-related knowledge. Hence, the most reverse engineering approaches follow a typical analysis process consisting of four main phases: extraction, abstraction, presentation, and interpretation (see Figure 11 in Section 2.2). The above typical reverse engineering process applies as well to the similarity analysis approach defined in this thesis. Accordingly, in the previous section we already hinted at the necessity of introducing separate analysis phases, which cover various aspects of the analysis and build the respective parts of the defined data model. Hence, the set model based similarity analysis process, depicted in Figure 34, consists of the following phases:
---	---

- **Structure extraction phase** constructs an asset structure model for each input asset variant. As discussed in the previous sections, the structural decomposition of an asset can include an identification of internal content hierarchy structure, if applicable, and it results in the creation of element sets which contain the basic analyzable Atomic Content Elements for the further analysis. For now, the constructed structure models of the asset variants are not yet related to each other.
- **Structure mapping phase** takes as its input the created structure models and uses the defined structure tree element equivalence relation to determine the asset structure hierarchy elements which correspond to each other across the compared asset variants. In the result, a unified hierarchy structure is constructed, and the correspondence information is stored in the structure tree set model.
- **Content set model construction phase** analyzes the corresponding sets of Atomic Content Elements, identified by the previous phases, and in accordance to Definition 20 determines their set intersections. Hence, by using the defined equivalence relation, the set model of Atomic Content Elements is built. The constructed model expresses the similarity of the input asset variants and is the basis for result visualization and interpretation. Furthermore, on the analyst request the computation of **subset calculations and metrics** can, at any point in time, be performed on the both set models and used to enrich the information basis for the subsequent analysis phases.

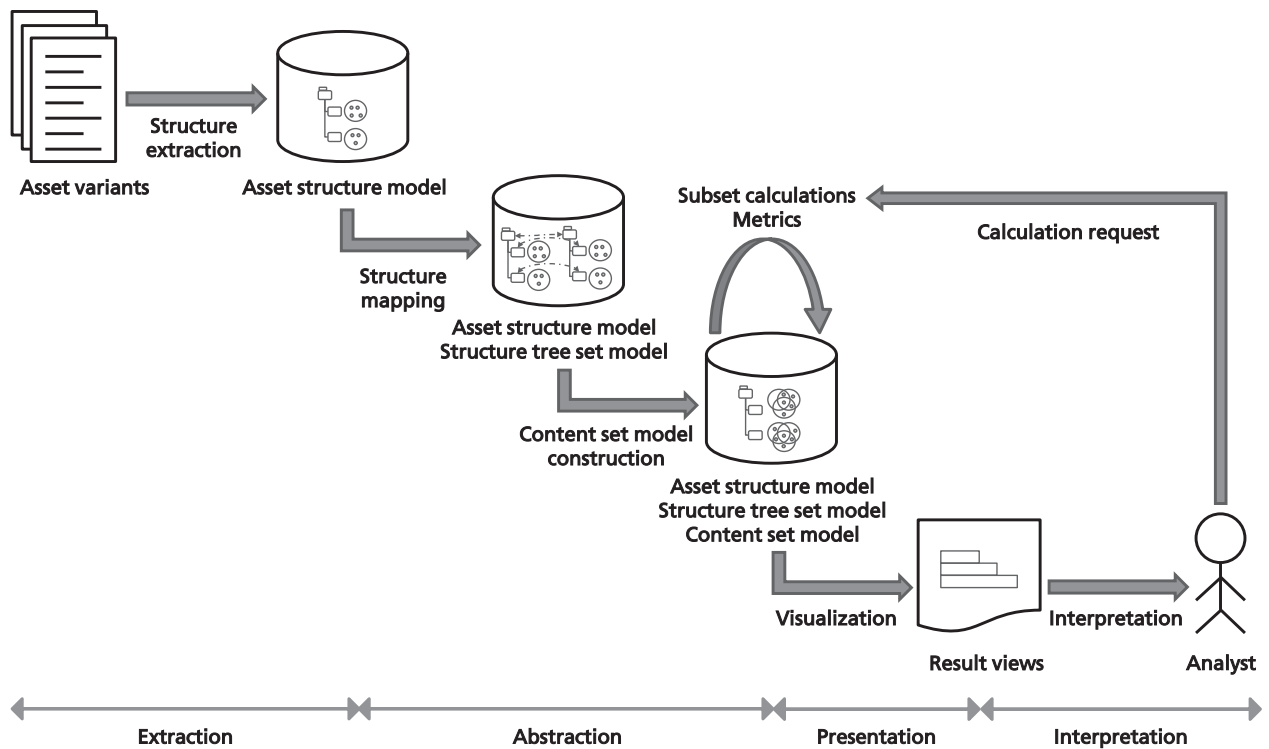


Figure 34

The process for the set model based similarity analysis.

- **Visualization phase** presents the calculated information in the form of diagrams and tables. We defined several views, enabling the user to explore and navigate through the created results. The user can retrieve the information about the degree of asset similarity, the distribution of similar elements in the asset content, and other considerations for any structural and atomic element of the asset content.
- **Interpretation phase** concerns the activities of a human analyst, performed using the visualized results and related to the analysis goals. As described in Section 2.2, these activities include analysis refinement (e.g. by starting **calculation requests** or re-running the analysis using a different definition of the equivalence relation), verification of previous assumptions, synthesis with results of other analyses, and finally definition of action items. These manual activities are not in the explicit focus of this thesis – however, we partially support them by providing the respective guidance in Appendix B.

Generic and customizable analysis process

Analogically to the other concepts of the set model based similarity analysis, the described analysis process is generic and applicable to any kind of software asset, as long as the basic analysis mechanisms listed in Sections 5.1 and 5.2 are defined. Hence, Figure 35 presents the analysis process on a higher abstraction level, and explicitly depicts the analysis input, the analysis results, and the three mechanisms where the generic analysis is customized for a specific asset type and analysis goal: the definition of asset content decomposition, the definition of structure element equivalence relation, and the definition of atomic element equivalence relation.

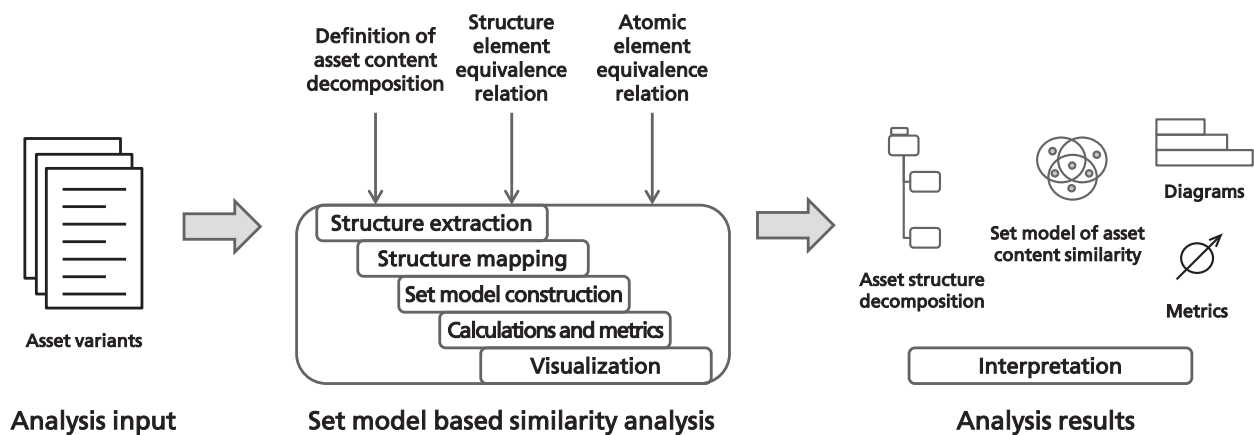


Figure 35

The similarity analysis input (left), the internal analysis process using the customizable definitions of analysis mechanisms (middle), and the resulting analysis output (right).

Details of the analysis phases

In the following sections, we discuss in more detail the concepts and algorithms specific to each of the defined analysis phases. As the structure extraction was already discussed in Section 5.2, we proceed now to the topics of structural correspondence mapping and set model construction.

## 5.4 Approaches for Set Similarity Model Construction

The need for  
equivalence  
relation

As discussed in the previous sections, a similarity analysis performed for a group of hierarchically decomposed asset variants constructs two set models: the first one for the Structure Tree Elements, expressing the correspondences between the elements of different asset content hierarchy trees, and the second one for the Atomic Content Elements, expressing the content similarity of the asset or its parts. In both these cases, for the construction of a set model it is necessary to define a respective equivalence relation on the asset content elements. The provided definition of the equivalence relation should, in particular, allow for unambiguous selection of the best match for a given content element, and guarantee the required result properties such as transitivity. In this section, we discuss the concepts and algorithms used in the construction of an equivalence relation on different forms of input similarity data and present their example instantiation.

Input element  
similarity  
measure

In a general case, the similarity between any two elements of the input asset content can be calculated by applying a pairwise similarity measure (as discussed in Section 4.6), for example calculated by a distance function. Without loss of generality, we can assume that the resulting similarity measure value is a real number from the  $[0;1]$  interval, where the value of 0 expresses the minimal or non-existing similarity, and the value of 1 expresses the maximal possible similarity. Naturally, the similarity measure can only be defined for element pairs which are comparable, e.g. when both elements are of the same type. However, this condition does not reduce the generality of the above similarity measure, as any pair of non-comparable elements can be assigned the similarity measure value of 0.

Forms of the  
input similarity  
data

The set model based similarity analysis does not consider the similarity between comparable elements located inside the same asset variant. Consequently, all such elements are treated as being different, and only the similarity between elements of different asset variants is considered. As our approach does not introduce any new similarity identification algorithm, the input similarity data is provided by an external algorithm. For example, the Levenshtein distance algorithm can be used to measure similarity of two files treated as lists of text lines. The input similarity data, being the basis for the construction of element equivalence classes, is typically provided in one of the two following, abstracted data forms, illustrated in Figure 36:

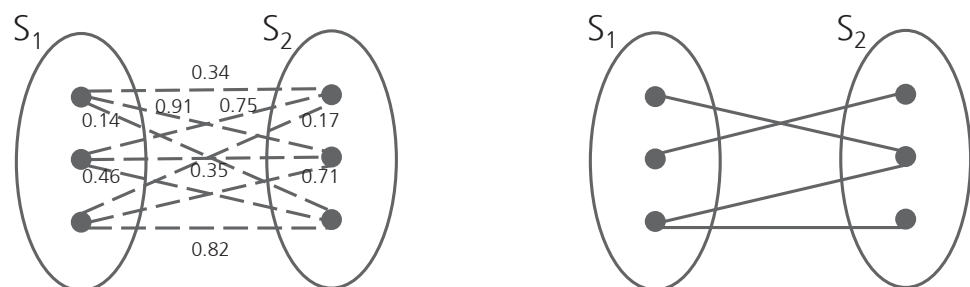


Figure 36

Abstract forms of input similarity data: gradual pairwise similarity (left) and binary pairwise similarity (right).



- **Gradual pairwise similarity** provides a similarity measure value in the  $[0;1]$  interval for any possible pair of input elements coming from different variants. Hence, for a given content element all elements from a different variant are initially considered as potential matches, but the degree of their similarity varies. This data form is typically produced for content elements having further internal structure, where the simple similarity definitions such as identity are not practically useful. Example similarity identification approaches calculating such similarity measures are Levenshtein distance applied to text files and common token coverage applied to code clones.
- **Binary pairwise similarity** lists pairs of similar elements. The element similarity is binary, as only the existence of similarity is reported: a given pair of elements is either similar or not. Hence, for a given content element a low number of potential matches, all having the same degree of similarity, is proposed. This data form is typically produced for simple content elements, having no meaningful internal structure, where the similarity definition of identity can be used. For example, the longest common subsequence (diff) algorithm identifies matching lines from two analyzed text files, but does not further investigate the similarity of non-identical lines.

Transformation of the data similarity form

Some similarity identification approaches initially collect the similarity data in the gradual form, but subsequently process them to report a result of the binary nature. This is done for example by many clone detection approaches, which apply a similarity threshold value to report the higher gradual similarity values as clone pairs and discard the remaining ones. An example of a transformation of gradual pairwise similarity to binary pairwise similarity is depicted in Figure 36, where the gradual similarity data in the left part of the figure is processed using a similarity threshold of 0.7 to receive the binary pairwise similarity data in the right figure part. In a general case, the resulting similarity relation is reflexive and symmetric, but not transitive. Hence, the construction of an equivalence relation, specifying the set model, requires further processing of the similarity data.

Algorithms and examples

In our analysis we receive input similarity data of both gradual and binary nature. Consequently, in the following subsections we discuss the concepts used in the processing of both these data forms. However, to support that discussion with respective analysis examples, in the subsection 5.4.1 we first present the asset content decomposition instantiated in our analysis implementation. Subsequently, in subsection 5.4.2 we describe the algorithms used for structure correspondence identification, working on gradual similarity data, and in subsection 5.4.3 we present the atomic element equivalence algorithms, using binary similarity data. Finally, in subsection 5.4.4 we discuss the consequences of the selected asset decomposition and equivalence identification approaches.

### 5.4.1 An Instantiation of Asset Content Decomposition

A tradeoff in source code analysis	<p>The objective of this thesis is to support the migration of variant software assets towards software reuse. In the most cases, the functionality of software assets is expressed in the form of source code. Hence, it is worthwhile for a similarity analysis approach to support an analysis on the source code level. However, multiple programming languages exist, and the followed programming paradigms and defined language grammars vary strongly. Consequently, an analysis which involves parsing the source code faces a tradeoff between broad applicability of the analysis and the technical difficulty of supporting the structural and syntactic forms of various source code languages.</p>
Language-independent textual decomposition	<p>In the context of this thesis, we strive for both generality and simplicity of the analysis instantiation. The provided instance of similarity analysis should be applicable to a sufficiently broad range of software assets, while its implementation should mainly focus on the concepts related to the set similarity model and not on the peculiarities of a given programming language or asset-derived data structure. For these reasons, we define the following, language-independent decomposition of the asset source code to structural and atomic content elements:</p> <ul style="list-style-type: none"> <li>• The source code of the asset is decomposed according to the structure of its file system tree. Hence, the source code folders assume the role of Structural Containers, and the source code files are treated as Content-Filled Elements.</li> <li>• The content of each source file is treated as text, without considering the used programming language. Hence, a file is decomposed into text lines, which are the Atomic Content Elements of the analysis.</li> <li>• Furthermore, the decomposition can filter out the asset content parts which conform to the above decomposition rules but are not considered relevant for the analysis. For example, the non-source files located in the source folders, or empty text lines in the source files, can be ignored and hence not treated as analyzable content elements.</li> </ul>
Equivalence functions and interpretations	<p>The presented textual and file system based asset decomposition determines the nature of applicable equivalence relation definitions, as well as the scope of possible interpretations of the analysis result. Consequently, the construction of the structure tree set model involves the search for code files which correspond to each other across the asset variants. The construction of atomic element set model involves the search for corresponding text lines across the mapped source files. Finally, the results of similarity analysis, applied to the given asset decomposition, measure the textual similarity of asset implementation. The atomic content element set model of an asset contains all of the asset's text lines, and the intersections of the asset element sets contain the text lines which are identified as common across the respective asset variants. In subsection 5.4.4 we provide a deeper discussion on the benefits and drawbacks of the above asset content decomposition and the equivalence relations defined on top of it.</p>

### 5.4.2 Mapping Correspondences in Structure Hierarchies

Three categories of Structure Tree Elements	<p>The construction of Structure Tree Element set model requires an identification of the tree elements which correspond to each other in the asset content structure hierarchies. Hence, the role of the Structure Tree Element equivalence relation is to construct a mapping of the variant structure trees onto each other. In an analyzed group of cloned asset variants, where each variant might further evolve after the cloning, a given Structure Tree Element can be assigned to one of the following three groups:</p> <ul style="list-style-type: none"><li>• The elements which existed in the original asset implementation and were <b>not moved</b> in the structure hierarchy after the cloning.</li><li>• The elements which existed in the original asset implementation, but were <b>moved</b> to other location in the structure hierarchy after the cloning.</li><li>• The <b>newly created</b> elements, which did not exist in the original asset implementation and were added after the cloning.</li></ul>
Use of location and content similarity	<p>Due to the cloning of the asset variants in the past, the corresponding Structure Tree Elements which were not moved are still located at the same asset root relative paths in their asset structure trees. Naturally, the content of any Structure Tree Element might also be modified, possibly to a significant degree, during the asset evolution. Hence, the equivalence relation on the Structure Tree Elements can be constructed based on two data sources. First, the similarity of the element location in the structure tree needs to be considered. In particular, the Structure Tree Elements having the same relative locations in the structure hierarchies are likely the elements which were not moved, and hence should correspond to each other – except if their content similarity is very low and other, significantly more similar elements exist. Second, the similarity of element content is also relevant for the equivalence relation: the mapping should identify tree element groups having maximal possible content similarity, and two cloned and subsequently modified elements are likely to be much more similar to each other than two unrelated elements. Hence, content similarity can be used to determine the corresponding elements which were moved in the structure hierarchy, or such newly created elements which are related despite having no common ancestor element.</p>
Element similarity versus their evolution history	<p>It is important to note that the input similarity of the variant structures and the classification of Structure Tree Elements based on their location similarity might not fully correspond to the original correspondences of tree elements resulting from their evolution history. First, two variant elements placed at the same tree locations could both represent the not moved elements, but it is also possible that one or both of these elements were moved or newly created and did not originally exist in their current tree locations. As these cases cannot be distinguished based on the tree structure alone, the content similarity can be used to justify whether such</p>

elements should indeed be reported as related to each other. But, second, also the content of a tree element could be significantly modified during its evolution. Hence, corresponding files might be not recognized because of strong content modification, while unrelated files might be strongly similar due to e.g. using boilerplate code. In our analysis approach we are interested in finding asset parts currently exhibiting reuse potential, regardless of their evolution history. We accept the discrepancies listed above, such as e.g. not recognizing two dissimilar elements having shared history, since their low similarity would likely not allow for reuse introduction anyway. Consequently, in the context of our approach the mapping should represent the best possible result calculated solely on the basis of the currently existing similarity of asset element location and content.

Use of gradual content similarity	The Structure Tree Elements contain a potentially large number of Atomic Content Elements, which can be modified during asset evolution. Hence, a search for related tree elements having identical content would typically not return many results – it is much more likely that the content of the related elements is slightly different. Consequently, in case the content similarity is used for the identification of corresponding tree elements, their similarity should be expressed in the gradual pairwise form.
Mapping for file system trees	In the file system based asset structure decomposition, used in our analysis instantiation, the role of the structure tree mapping is to verify the similarity of files located at the same locations in the file system tree, i.e. having the same paths and names, and to reconstruct the file movement operations performed in the evolving asset variants. In result, the mapping identifies the groups of similar files, which are likely to represent cloned variants traceable to a single code file from the primary asset implementation. Furthermore, strongly similar newly created files should also be identified – in the practice such files might be created by a subsequent small-scale cloning or they can otherwise represent potentially reusable functionality.
Mapping approaches	<p>Based on the above observations, our analysis instantiates the following three approaches to the identification of corresponding Structure Tree Elements across asset structure hierarchies:</p> <ul style="list-style-type: none"><li>• <b>Location identity</b> approach does not consider the content of the tree elements, but only their location. The elements placed at the identical tree paths are reported as equivalent to each other – for example, for the file system trees the approach would report files having the same tree root relative paths and file names. This simple approach can only provide good results when the analyzed asset structures have not been significantly modified after the cloning. However, in the practice it was suitable for about 70% of software system groups we analyzed.</li><li>• <b>Manual mapping</b> approach can be used when the structure element correspondences are known to the analyst or system developers. As an automatic approach cannot provide a better result in that case,</li></ul>

the corresponding elements are marked by a human. Alternatively, to reduce the effort, a human can review and modify the preexisting results of an automatic approach. In the practice, we applied the manual approach to about 10% of analyzed software system groups.

- **Algorithmic** approach involves the use of a formal algorithm. Several concrete algorithms are imaginable here. In the context of our analysis approach, two algorithms were developed:
  - An algorithm matching isomorphic subtrees between given tree structures [Valiente 2001] was adapted to the file system structure by Zahra [Zahra 2010]. Hence, the algorithm analyzes only the input tree structures, where the tree elements are labeled using file system names. The rationale for using the tree structure based algorithm is especially to detect folder-level rename and move operations, and to perform the mapping without potentially time-consuming element content comparison. However, this algorithm did not provide sufficiently good results and was therefore not used in the practical analyses.
  - Tenev developed an algorithm for mapping multiple variants of a connected graph (not necessarily a tree), which uses a predefined distance function calculating the similarity of any two graph elements [Tenev 2011]. In the case of file system structure, the Levenshtein distance was used to evaluate the content similarity of any two files. We applied the algorithm to about 20% of software system groups we analyzed, and received very good results (see Chapter 7 for more information on the algorithm's evaluation). In the following paragraphs, we briefly discuss the details of the algorithm.

Algorithm result optimization criteria For the reasons discussed above, an algorithm for tree element mapping which relies on element content similarity needs to use gradual pairwise similarity. Furthermore, as the algorithm maps many variants of a tree structure, containing many tree elements, it should optimize its result in the following way:

- Local optimum for a given tree element in a pairwise mapping: when mapping two tree structure variants, a given structure element should be assigned to its most similar counterpart in the other variant, unless the content similarity of even the best candidate is too low to be meaningful.
- Global optimum in a pairwise mapping: the achievement of the local optimum for every tree structure element might be impossible, as the proposed mapping choices might conflict with each other (e.g. two elements might have the same most similar counterpart). Hence, the globally optimal pairwise mapping should optimize the similarity of the complete identified structure match, and might sacrifice the local optimums of some elements, i.e. provide for them alternative less similar counterparts, to achieve that goal.

- Global optimum in mapping many structure variants: the identified pairwise element matches, calculated for each tree variant pair, might conflict with each other when building the transitive equivalence classes over all tree variants. For example, the two elements in a given pairwise match might be, based on the other pairwise matches involving these elements, optimally placed in two different equivalence classes. Hence, it might again be necessary to discard or correct some of the provided results, optimal from the point of view of a given variant pair, in order to achieve the global optimum similarity identified over all analyzed variants.

Tenev's  
mapping  
algorithm

Tenev provides an algorithm following the above optimization strategy, inspired by multiple alignment algorithms used in bioinformatics [Gusfield 1993]. The algorithm has a time complexity of  $O(n^3k^2\log(k))$  and requires  $O(n^2k^2)$  memory space, where  $n$  is the number of analyzed graphs (e.g. asset variants) and  $k$  is the number of graph elements (e.g. source files).

In the first step, the algorithm computes the pairwise mapping for each pair of input asset variants. To achieve that, the similarity of all possible element pairs is computed using the defined distance function. Then, the element pairs are selected for the pairwise mapping using a greedy choice, starting from the most similar element pair. Naturally, every next element pair can only be selected for the result if it does not contain any of the elements already selected before. The selection process terminates if there are no more candidate element pairs, having similarity above the defined threshold, which can be added to the mapping result. In the result, an approximation of the globally optimal pairwise mapping is constructed.

In the second step, the algorithm uses the prepared pairwise mappings to construct the final mapping for all input variants. This is done by iterative use of the Center Star method [Gusfield 1993], illustrated in Figure 37 on the next page. For each input graph variant, a cost function is calculated on all pairwise mappings involving that variant. In this way, a single variant having the strongest connections to all remaining variants is determined. Subsequently, the selected variant is used as the star center, that is, for each of its elements an equivalence class is built which includes the given element and all the elements from the other variants which were mapped to it in the pairwise mappings (even if these elements were not transitively mapped to each other). Hence, for  $n$  input variants only the  $n-1$  pairwise mappings involving the star center variant are used to construct the equivalence classes. In this way, the use of the Center Star method is a form of global optimization, as it selects the pairwise mappings which lead to a better global mapping result.



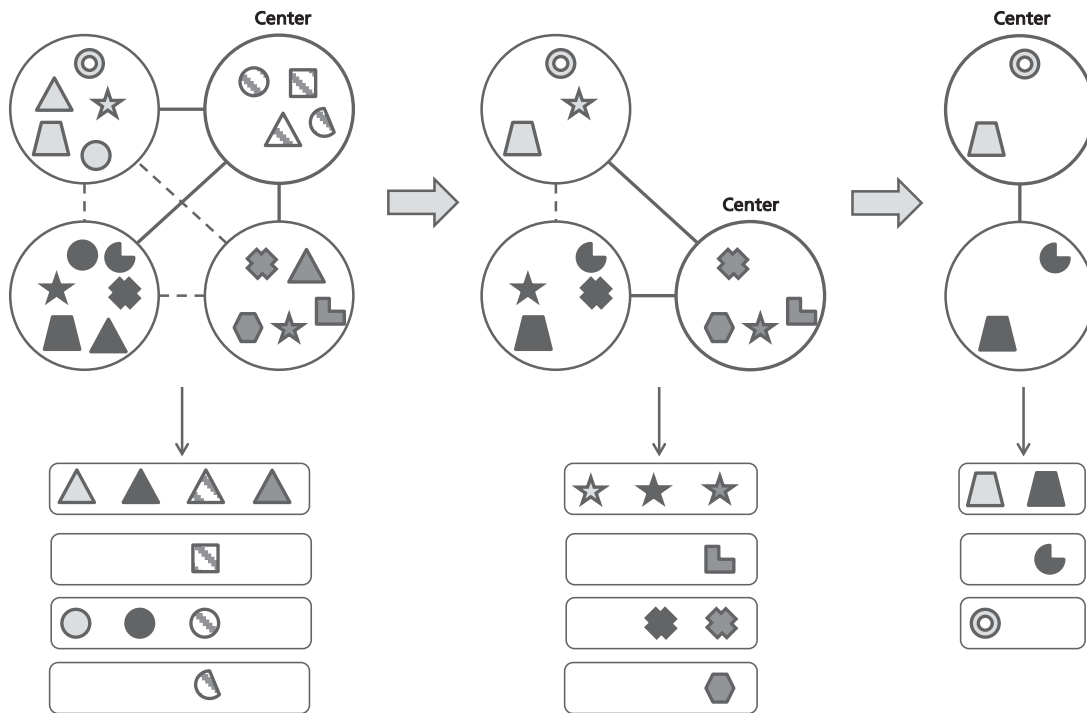


Figure 37

The iterative Center Star method: in each iteration the star center variant is determined, and its elements are mapped using the center's pairwise alignments.

After every element from the star center was mapped, the process is iteratively repeated: the pairwise mappings between the remaining variants, reduced by removing the elements already mapped to the previous star center, are again input to a cost function which determines the next star center. After  $n-1$  iterations all input variants are processed and the mapping result is complete.

Note that the created equivalence classes can contain element pairs not included in the pairwise mappings (due to a possible transformation of a non-transitive relation graph into an equivalence class), while they might also miss some of the previously identified element pairs. However, as the element pairs are also constructed as a result of the tradeoff between the local and global pairwise mapping optimums, some of the pairs might also include two non-related elements or miss the related ones. To measure the consequent result quality, the algorithm should be therefore evaluated based on its final results, as discussed in Section 4.5. We discuss the algorithm evaluation in Chapter 7, using the text-based instantiation of the analysis approach.

As defined in Section 5.2, the mapping of  $n$  input structure trees onto each other constructs a set model on the Structure Tree Elements, and consequently determines the form of the unified content tree representing the union of all input asset variants. The mapping only determines the equivalences of Content-Filled Elements, as the Structural Containers are always mapped to each other using the location identity approach. Consequently, the mapping algorithms described above only construct the correspondences for Content-Filled Elements, i.e. the code files.



### 5.4.3 Definition of Atomic Content Element Equivalence Relation

<p>Analogies to the tree structure mapping</p>	<p>After the structure tree elements are mapped to each other across the analyzed variants, the atomic element set models are built. For reasons analogical as in the case of tree elements, the identification of corresponding atomic content elements can be performed based on the similarity of their location and content. As the atomic elements represent small pieces of asset content, which usually do not have a meaningful internal structure, their similarity is frequently expressed in the binary pairwise form. This observation applies also to the atomic element analysis of our textual asset content decomposition, as described in the next paragraph. Hence, in this subsection we discuss the algorithms and approaches dealing with the construction of a set model based on binary pairwise similarity provided for the asset content elements.</p>
<p>Algorithms for the textual content decomposition</p>	<p>The file system based asset structure decomposition, used in our analysis, treats the source code files as text and divides the file content into text lines, which are the atomic content elements. Hence, the construction of the set model requires finding the text lines which correspond to each other. In the case of two analyzed files, the Longest Common Subsequence (diff) algorithm is most frequently used for this task in the practice. The text lines inside a file are ordered, and the result of diff is the longest list of lines which exist in both given files and occur there in the same order. Hence, we decided to create the text line set model based on the results of the diff algorithm. Using other algorithms, such as e.g. searching for similar text lines without considering their order, would also be possible. We prefer diff however, as its results, such as e.g. “these lines occur one after another in the second file, potentially with gaps”, are easier to interpret and more meaningful for developers than the results of line order independent algorithms, e.g. “each of these lines exists at some unspecified place in the second file”. In subsection 5.4.4 we present an approach for improving the results of diff with regard to the detection of non-identical, but still similar, text lines.</p>
<p>Properties of diff</p>	<p>Diff creates the text line similarity results in the binary pairwise form – that is, it identifies the line pairs containing the same text, while the remaining line pairs are considered dissimilar. The use of line ordering information allows for an unambiguous assignment of lines to each other, as required in our approach: even if multiple text lines identical to the sought line exist in the counterpart file, only one of these lines, determined by its location, is reported by diff in the resulting similar line pair. Moreover, no similarities between the lines of the same file are identified. Consequently, diff results constructed for a pair of files can be interpreted as a group of equivalence classes defined on the input text lines, and directly used to construct a set model for the input file pair. Finally, the results of diff contain no false positives, as no two different text lines are reported as similar.</p>

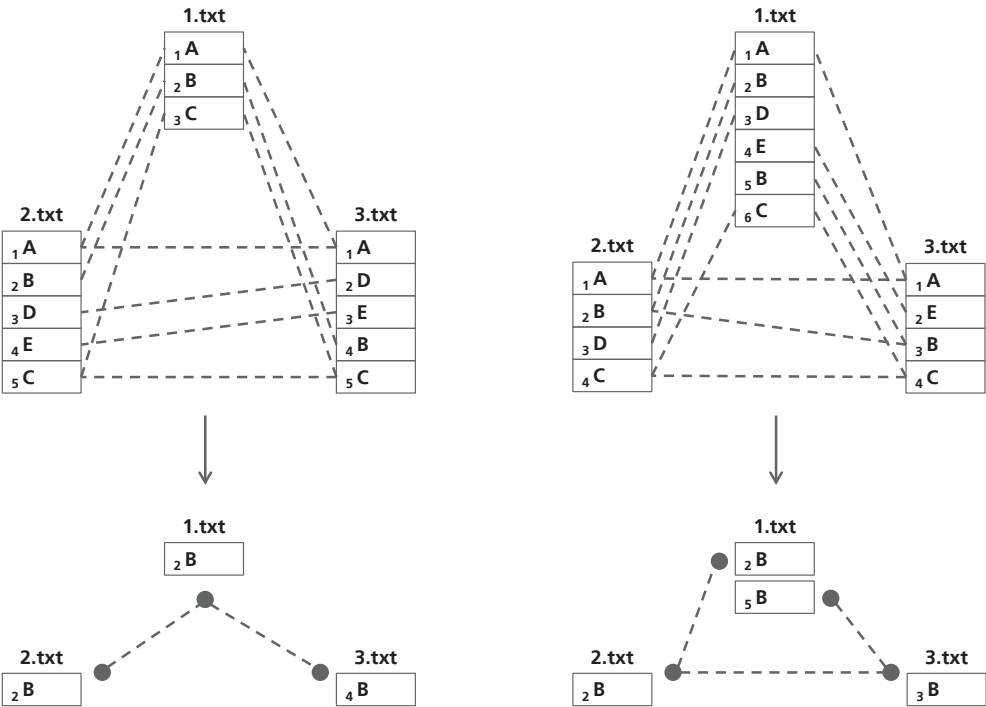


Figure 38 Examples of non-transitive relation graphs constructed from the diff results.

Result  
optimization  
criteria

In the general case of  $n$  input files, the relation graphs created by  $\frac{n(n-1)}{2}$  pairwise diffs performed on these files are in over 99% cases transitive (see Chapter 7). The transitive graphs can be directly reported as equivalence classes. The remaining graphs of identified pairwise similarities are not only non-transitive, but they can also include more than one line originating from the same file (see Figure 38 for examples). These kinds of input irregularities can also occur for other similarity identification algorithms besides diff. Hence, in the general case an algorithm constructing equivalence classes out of non-transitive relation graphs is needed. The algorithm can optimize its result according to at least the following four, partially conflicting criteria:

- [OC1] Maximizing the amount of identified element pairs included in the equivalence classes. For example, each line pair identified by diff is correct, so they are all worth to be included in the result. However, including all element pairs requires a transformation of non-transitive relation graphs into equivalence classes, adding element pairs not occurring in the input to the result. Also, the set model construction principles forbid such transformation for graphs containing more than one element originating from the same asset variant.
- [OC2] Minimizing the amount of element pairs, which were not included in the input, in the reported result. In particular, it is possible to not include such pairs at all – in such a case, the equivalence classes need to be built from transitive subgraphs of the input relation

graph. Below, we discuss various possible solutions to that problem, following to a different degree the other optimization criteria.

- [OC3] Maximizing the size of identified equivalence classes. Large equivalence classes are interesting in the context of reuse migration, as the code parts recognized as shared by many asset variants indicate more reuse potential than the code parts recognized as shared by only few variants. However, defining possibly large equivalence classes might conflict with the other criteria.
- [OC4] Minimizing the amount of identified equivalence classes might also help express the reuse potential. The minimal size of the constructed equivalence set union can be used to estimate the maximal reuse potential of the input assets, achieved when each equivalence class is transformed to a reusable content element. Again, this criterion in some cases conflicts with the others, as discussed below.

Transitivity  
algorithms  
for relation  
graphs

Simultaneous optimization for the criteria OC1 and OC2 is addressed by existing algorithms for correlation clustering [Bansal 2004]. However, the assumption A4, discussed in Section 4.3, states that high certainty results should be preferred. Consequently, we decided to focus on the criterion OC2, and only include these element pairs in the result which were provided in the input while not allowing any other element pairs. Hence, we deal with the problem of covering the input relation graph with disjoint transitive subgraphs (cliques). Note that with that problem definition, the further problem of only creating equivalence classes that do not contain two elements from the same variant can be ignored – such two elements are not connected with a similarity relation, so they are never included in the same clique anyway. The clique coverage problem can be solved in the following ways:

- To optimize the criterion OC3, an algorithm searching for the maximum clique can be applied iteratively on the relation graph. Hence, the maximum clique possible for the given graph is found, and the remainder of the graph is then covered with further disjoint maximum cliques. Our implementation uses two variants of that algorithm: a faster, approximated one and an exact one based on the work of Tomita et al. [Tomita 2006].
- Algorithms optimizing the criterion OC4, that is partitioning the input graph into a minimal number of cliques, also exist [Tseng 1986].
- The optimization of criterion OC1 involves determining a minimum graph cut created by partitioning the graph into disjoint cliques. An algorithm which can be used for that goal was developed by Ji and Mitchell [Ji 2007].

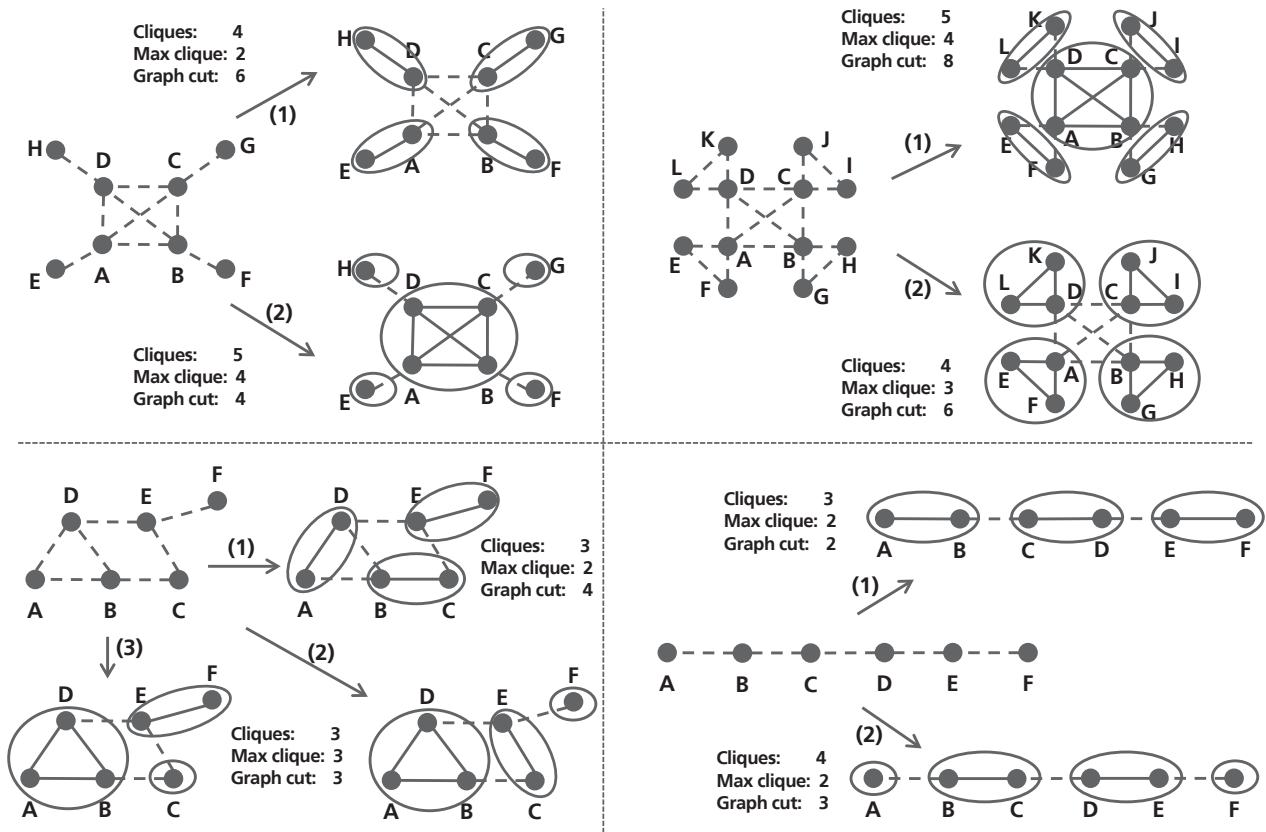


Figure 39

Example relation graphs and their alternative solutions.

- Simultaneous optimization for multiple stated criteria is harder than for a single one. The listed algorithms can produce several solutions, which are equivalent from their criterion point of view but are not equivalent according to the other criteria. Listing all solutions optimal with regard to one criterion, and then selecting from them the best choice with regard to another criterion, might be in some cases a suitable strategy:
  - All solutions provided in the bottom left corner of Figure 39 contain the minimum possible amount of 3 tuples, but the solutions denoted with (2) and (3) contain a larger maximum tuple and have a lower graph cut.
  - Both solutions provided in the bottom right corner of Figure 39 contain the maximal possible clique of size 2, but only the solution denoted with (1) is optimal with regard to the amount of tuples and to the graph cut.
- However, there exist cases where achieving a solution satisfying all three criteria OC1, OC3 and OC4 is not possible. Hence, a prioritization of the criteria is necessary:
  - Consider the upper left corner of Figure 39, where either the two criteria OC1 and OC3 or the single criterion OC4 can be fully optimized. The solution denoted with (1) contains the minimal number of equivalence classes, as specified by the criterion OC4, while the different solution denoted with (2) includes the maximum sized equivalence class and has the minimum graph cut.

- In the upper right corner of Figure 39, the criteria OC1 and OC4 can only be achieved for non-maximal clique size (hence missing the criterion OC3), while satisfying the criterion OC3 by creating a maximal clique of size 4 leads to non-optimal solutions for the other two criteria.
- Finally, in some cases multiple solutions, optimal with regard to all three criteria, can exist. For example, consider the solutions denoted with (2) and (3) in the bottom left corner of Figure 39. In such a case, a further criterion is necessary so that only one of these solutions is consistently selected each time the given graph appears. We use a prioritization of graph vertices, hence determining an order on them, and select solutions where the larger tuples are created for vertices having a higher priority. For example, if the vertex priority is noted using the alphabetical order, solution (2) is preferred over (3) as vertex C appears in the order before vertex F. In Chapter 6 we describe the possibilities to provide a graph vertex order, solving the presented choice problem, without violating the variant order independence mandated by the construction requirements of our approach.

Applying the similarity data processing approach described above, in Figure 40 we provide solutions for the non-transitive diff relation graphs presented in Figure 38. The transitivity of diff-based input and the result quality of the maximum clique algorithm are evaluated in Chapter 7.

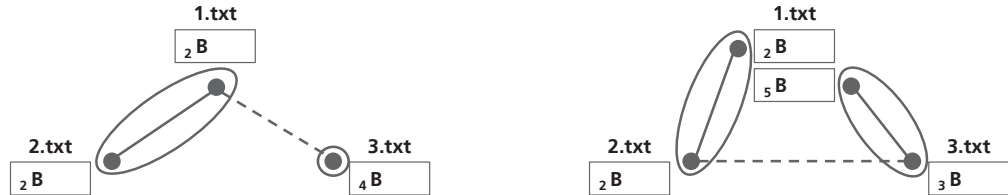


Figure 40

Solutions provided for the relation graphs from Figure 38.

#### 5.4.4 Discussion

Context  
factors in  
transitivity  
identification

In the previous subsections we presented a range of possible approaches and algorithms for constructing an equivalence relation out of initially non-transitive similarity data. The resulting equivalence relation differs from the input data: edges are removed or added to the input similarity relation graphs, and in the case of gradual similarity the input similarity values are not part of the result. Naturally, the difference between the originally detected similarity and the returned equivalence relation should be minimized. The degree of that difference depends primarily on the transitivity of the input data, and, to a lesser extent, is influenced by the applied transitivity algorithm. Hence, the type of the analyzed asset, and the definition of its structure decomposition which determines the applicable similarity detection approaches providing the input data, significantly influences the generally achievable quality of analysis result.

In a specific case, the degree of difference can also be higher for strongly dissimilar input asset variants, for example extensively modified during a long parallel evolution. In that case, the mapping algorithm might miss some corresponding structure element pairs (e.g. if their similarity degree falls below the acceptance threshold), and the corresponding atomic content elements might also be more difficult to find. This might cause an input relation graph to become non-transitive, despite an existence of a corresponding element group, and consequently lead to further element pair omissions during equivalence class construction. In any case, even if the provided input is highly transitive, it is still necessary to consciously select a transitivity algorithm suitable for the defined analysis goals. As discussed in the previous subsection, the form of the created equivalence classes and their difference to the input data can be optimized towards various criteria, affecting the meaning and interpretation possibilities of the analysis result.

The role of analysis assumptions	The form of presented mapping and transitivity algorithms is strongly influenced by the analysis assumptions listed in Section 4.3. First, the assumptions A1 (only considering similarity between the variants), A3 (one-to-one correspondences), and A5 (transitivity), fundamental for the definition of the set similarity model, create the need for the presented algorithms and specify the form of their results. Second, the assumptions A2 (focus on high similarity input) and A4 (high result certainty) drive the design choices made while defining the algorithm details. Because the motivation of analysis assumptions, described in Section 4.3, also applies to the consequent set model algorithm choices, we do not repeat the respective discussion here.
Focus on the textual system decomposition	In our opinion, the disadvantages of ignoring a reasonably minor part of the similarity input during the construction of equivalence classes are far outweighed by the benefits of using the set model in result analysis. We provide a deeper discussion of that topic in Section 5.7 at the end of this chapter, when the full benefits of the set model usage are already described. Hence, in the remainder of this subsection we concentrate on the properties of the textual, file system based asset decomposition, and on its consequences for the similarity analysis.
Advantages of textual decomposition	The textual asset content decomposition, and the choice of the diff algorithm as the basis for text line equivalence function, is motivated by its generality and simplicity. Regarding generality, the content of a broad range of asset types is physically stored in textual files, and can hence be processed by the analysis instantiation. Diff is frequently sufficient in the practice for comparing text-based files, even if their internal syntax and semantics is complex: a notable example is the use of diff by most configuration management systems for comparing versions and variants of source code files regardless of their programming language. Regarding simplicity, uniform processing of text line lists is much simpler than dealing with various structures of asset type specific abstract syntax



trees, for example depending on the programming language. Consequently, the analysis instantiation can focus on the aspects peculiar to the set model construction.

Analogically, the decision to use a file system based asset structure hierarchy is also motivated by the concerns of generality and simplicity: file system hierarchies are commonly used, and the construction of other possible structures (e.g. namespace hierarchies) would require deriving semantic information from the file content, which is bound to a specific asset type. To provide results consistent with the text-based definition of file content similarity, the provided mapping algorithms, identifying corresponding similar files, also use text-based file similarity measures such as the Levenshtein distance.

Finally, the simplicity of the defined decomposition, the high detail level of text analysis, and the use of well-known diff algorithm for similarity detection are likely to help technical stakeholders in understanding the analysis process and trusting its results. As discussed in Section 4.2, the factors of understandability and trustability are important for every reverse engineering technique.

Use of clone  
detection  
results

Alternatively, clone detection could be used for the similarity analysis of textual assets, as several text-based clone detection approaches exist [Roy 2007]. In that case, the mapping could measure file similarity using common clone coverage, and the atomic element similarity could be built using code clones expressed as blocks of corresponding lines or tokens. Furthermore, some clone detection approaches directly report clone classes instead of pairs, eliminating the need for transitivity algorithms. However, the clone classes cannot be used directly in the construction of a set model, as they might overlap (i.e. two classes can partially cover the same lines or tokens). Moreover, the overlapping clone classes might cover different file variant groups, and overlap only in some of the involved variants. Hence, compatible non-overlapping clone class coverage needs to be constructed. To retain a possibly high proportion of the input similarity data, the detected clone classes might be cut into smaller parts, possibly having different similarity relation graphs. Solving that problem is possible [Tenev 2013], but the involved algorithms and the interpretation of the constructed result are more complex compared to diff.

Disadvantages  
of textual  
decomposition

The main disadvantage of text-based processing, ignoring asset content syntax and semantics, is the sensitivity to non-meaningful content changes. First, the content can be textually modified without syntactic changes, for example by adding whitespace characters or by changing the location or amount of line breaks. Second, the content might be modified to a syntactically equal form by renaming identifiers, changing variable types, or moving a line group to a different location in the same file. And third, a modification can replace the old content with a functionally equivalent, but syntactically and textually different form.



Extending diff  
with code  
normalization

We support diff in detecting part of the above changes by performing user-configurable in-memory normalization of the input text. The simplest form of normalization performs line content modifications such as removing unnecessary whitespaces, character case normalization, string replacement (e.g. based on user-provided equivalent identifier list, such as compiler-specific keywords), and other rule-based replacements such as comment removal. Hence, we extend the pairwise equivalence relation implemented by diff, as the non-identical text lines which are equivalent with regard to the normalization can also be found. Further normalization includes filtering out unnecessary text lines such as empty lines or identifiable code blocks (e.g. multi-line comments). Finally, advanced normalization of the input code, involving parsing and syntax-based transformations, can be performed by an external tool before running the diff algorithm [Roy 2009b]. The advanced normalizations used by Roy include pretty printing (nullifying the formatting differences), normalization of identifier names and data types, normalization of constants and literal values, and normalization of expressions (e.g. arithmetic operator changes). In this way, most categories of local code changes detectable by clone detection can be neutralized. In result, after normalization the code affected by these categories of changes is recognizable as similar by diff.

There are two prominent remaining change categories not addressed by the normalization approach. First, due to the line order dependence of diff, it is still not possible to recognize an unmodified code part which was moved to a different location in the file. This problem could be mitigated by the use of a differencing algorithm detecting block moves, such as the algorithm of Tichy [Tichy 1984]. Second, the detection of code parts which remain semantically equivalent despite modification is not possible with the normalization approach, as it requires building a detailed, asset type specific model of the analyzed source code functionality [Gabel 2008]. Despite these deficiencies, a diff-based similarity analysis provides practically useful results, as reported in Chapter 7.

## 5.5 Visualization

Proceeding to the next phase of the analysis process, in this section we define various visualizations of the information stored in a hierarchical set similarity model. The defined visualizations are intended to support the human analyst in navigating and understanding the provided similarity information. As noted by Eick et al., visualizations of abstract data (i.e. non-physical objects) require effective visual metaphors [Eick 2002]. Consequently, for the defined visualization concepts we discuss the analysis support rationale that led to their presented form. Accompanying the graphical visualizations, we also briefly describe the tables and data exports which provide a textual view on asset similarity.

The visualization of a hierarchical set model needs to include at least two basic views: a view presenting the similarity of a group of intersecting sets, and a view on the system structure hierarchy. We describe these views and their coupling in the following two subsections. Afterwards we describe further visualizations, which are aimed at the distribution analysis of similarity – either with the goal of understanding the distribution in general, or supporting the identification of specific asset elements interesting in the context of analysis goals. Finally, we present two views on the set model information answering specific analysis questions and visualized with phylogenetic trees: a dendrogram, which clusters the variant asset sets according to their similarity, and a cladogram which reconstructs their probable evolution history.

### 5.5.1 Set Bar Diagrams: Visualizing the Similarity of Multiple Intersecting Sets

Venn diagrams and their properties	The visualization of a set model should present the information about all analyzed sets and their similarity (i.e. their existing intersections) in a compact and understandable form. The most common method for visualization of a group of intersecting sets is a Venn diagram [Venn 1880], where the sets are presented as partially overlapping shapes (see the left part of Figure 41). A Venn diagram for $n$ sets contains $2^n - 1$ areas, corresponding one-to-one to all possible set intersections. The diagram displays hence the complete information about set similarity. Moreover, the set intersections which differ only by adding or removing one set are adjacent on the diagram, so that all intersections fulfilling a simple logical statement such as $S_1 \cap S_2$ are conveniently grouped.
Disadvantages of Venn diagrams	Venn diagrams are intuitive and easy to understand for up to five sets, but for a higher number of sets the exponentially growing number of displayed diagram areas, and the need to use complex irregular shapes instead of ellipses [Ruskey 2005], makes them complex and difficult in interpretation. Furthermore, area-proportional Venn diagrams, visually indicating the relative sizes of the set intersections, were only demonstrated to exist for a low number of sets [Ruskey 2005], and the presented intersections still have varying shapes which hinders the visual comparison of their size. Finally, the fixed layout of adjacent set intersections prevents visual grouping of the intersections according to arbitrary criteria (consider e.g. a group of all intersections belonging to exactly two sets). Euler diagrams, which are a variant of Venn diagrams not displaying empty intersections, are slightly less complex for suitable data sets, but otherwise share the listed disadvantages. As in the practice an analysis of 20 or more cloned asset variants might be required, we do not consider Venn diagrams and Euler diagrams to be suitable for the defined application scenarios.

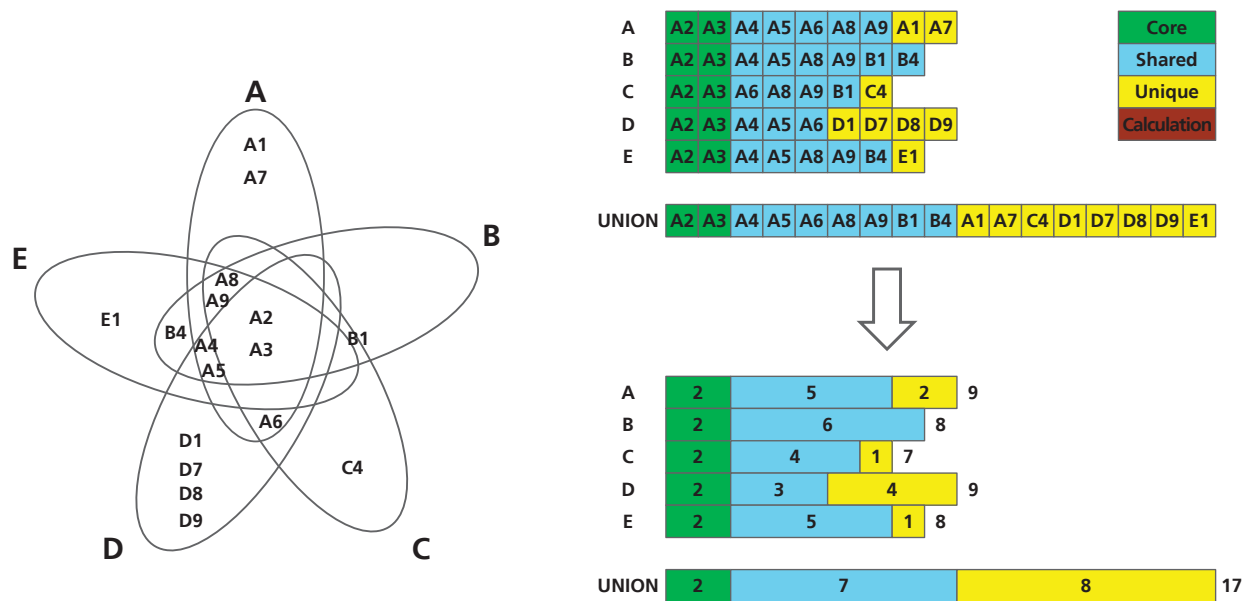


Figure 41 A Venn diagram for five intersecting sets (left) and the construction of a set bar diagram for these sets: the intermediate form (top right) and the final diagram form (bottom right).

Set bar  
diagram  
visualization

Therefore, we define a visualization of a group of intersecting sets in the form of a bar diagram, which eliminates most of the above disadvantages of Venn diagrams at the cost of displaying only a part of the available information at a time [Duszynski 2010a]. The construction principle of a set bar diagram is presented in Figure 41. A set bar diagram contains a single bar for each of the  $n$  sets, which represents all set elements, and one additional bar displaying the elements of the complete set union. All the bars in the diagram have equal width, and the length of each bar is proportional to the number of elements in the respective set. We categorize the set intersections (or, equivalently, the set elements) into three groups: *core* (belonging to all sets), *shared* (belonging to not all, but more than one set) and *unique* (belonging to exactly one set). Each element category is displayed in an own section of the bar, sized in proportion to its cardinality, and indicated by a distinct color. In the figure, the cardinality of each set or element category is indicated by a number placed on or near the bar section. These numbers can also be hidden in the diagram, and displayed in a tabular form instead.

Due to the size-proportional visualization, a set bar diagram provides a quick overview of the amount of set elements falling into each of the defined categories, as well as over the sizes of the analyzed sets relative to each other and to the set union. The relative proportions of the three element categories allow for an initial estimation of the reuse potential of the asset parts represented by the displayed sets: a bar with a high core element proportion indicates a high reuse potential, while a bar with a high unique element proportion does not. The shared element category can be suitable for reuse to a different degree, as it contains elements shared by 2 sets as well as by  $n-1$  sets. Hence, the use of subset calculations is necessary to retrieve more information about these elements.

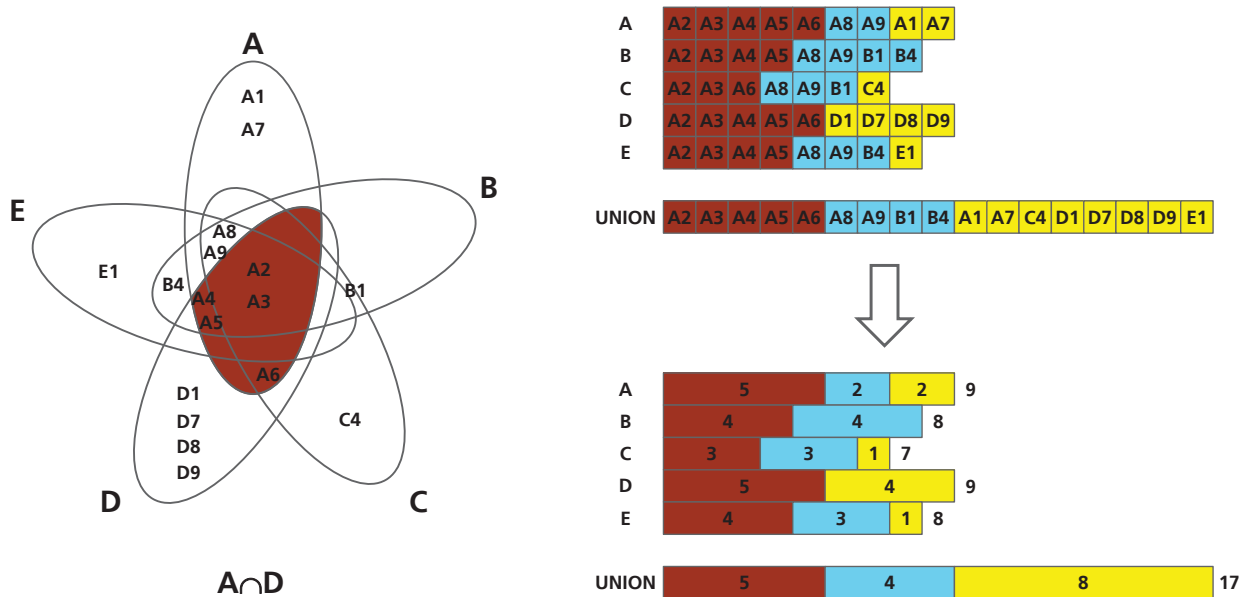


Figure 42 The visualization of an example subset calculation in the set bar diagram.

Visualization of subset calculations

Consequently, a fourth category of *subset calculation* elements can be built from the elements of the above three categories fulfilling a provided logical condition, and displayed on the diagram on demand. The visualization of a subset calculation, showing the set intersections specified by a logical condition, can be dynamically overlaid over the bar diagram as presented in Figure 42. Note that the subset calculation bar section is contiguous, except for a case when it is split into two parts for a calculation including core and unique elements, but only some or none of shared elements. Furthermore, it is possible to use different colors to visualize a group of subset calculations on the same diagram, as long as they are disjoint. This can be done for example to indicate the categories of elements shared by a different number of sets.

Comparison to the Venn diagrams

In the practice, gaining an overview of the similarity of a group of sets typically requires displaying a group of bar diagrams, as each of the diagrams contains only partial information about the sets – unlike a Venn diagram, which shows the complete set similarity. However, a bar diagram can be also constructed and understandably visualized for a high number of variants, including a graphical indication of the relative sizes of the sets and their intersections, which is not possible for Venn diagrams.

Availability for any set model

As the set bar diagram is based on the set similarity information only, it can be displayed for any set model constructed by the analysis. Hence, for any Structure Tree Element a bar diagram displaying the atomic content model of the element, as well as a bar diagram of the structure tree set model for the subtree rooted in the selected element, are available.

5.5.2 Visualization of Set Similarity in the Asset Structure Hierarchy

UML-like  
tree structure  
presentation

During a similarity analysis, the visualization of the tree-based asset structure hierarchy should enable the analyst to explore the structure tree, recognize the elements which might be interesting in the context of the analysis goals, and request details for these elements. To provide the diagram space needed for displaying summarized similarity information for each currently visible tree element, we use a simple tree structure visualization inspired by UML package diagrams, which presents the tree elements in the form of nested rectangles (see Figure 43). In the visualization, each element rectangle is divided into up to three areas: the name area displaying the element type and name, the similarity information area described in more detail below, and, for Structural Containers, the content area displaying the child tree elements. As typical for tree-based visualizations, the non-leaf element rectangles can be expanded, displaying their immediate children at the next tree level, or collapsed, hiding all their children.

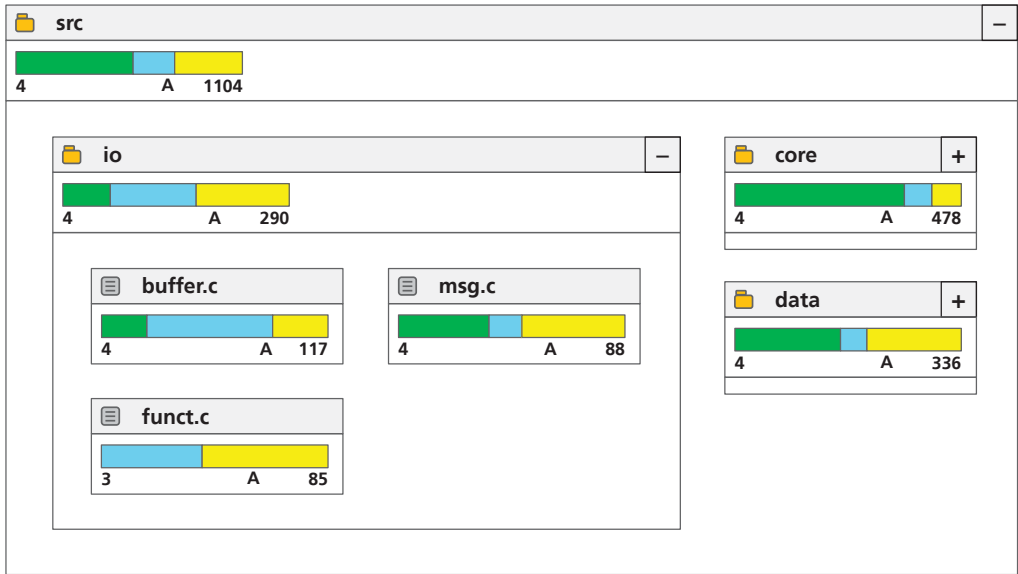


Figure 43 Hierarchy structure visualization showing similarity information for each displayed element.

Similarity  
information  
area

The similarity information area provides a condensed view on the set model of the given structure element. It displays a single similarity bar, which is identical with the union similarity bar of the respective set bar diagram. Hence, the proportion of core, shared and unique elements indicates the overall similarity of the contained sets. As in the case of bar diagrams, the similarity bar can also display the proportion of set elements returned by a similarity calculation. Although the bar section proportions are preserved in each similarity bar, the bars belonging to different elements are drawn with the same size. The reason for that is that vast differences in possible element set sizes can exist on the same diagram, conflicting with the decision to use a fixed-size information area. Hence, additional textual information is provided below the

similarity bar: starting from the left side, the number of analyzed sets (i.e. element variants), the type of the shown set model (atomic or structural), and the size of the contained set union are displayed. Hence, an indication of the relative size of each element is provided, albeit not in a graphical form.

Visualization  
approach  
properties

The UML package tree structure visualization supports a top-down result exploration approach, where the user can start at the highest structure level and proceed stepwise to the details by expanding the elements that exhibit interesting concentrations of similarity. For each element, further similarity details are available on demand: all other types of diagrams defined in this section, for example the bar diagram, can be displayed for the selected element in a separate view. Moreover, the structure and similarity information can be filtered and otherwise configured before visualization. First, the displayed tree can include only elements fulfilling a predefined condition, for example the elements existing in a certain variant, having large sizes, or associated with a metric value above certain threshold. Second, the displayed similarity information can be based not on the set union, but on a specific variant set, and show the similarity bars corresponding to that set. Finally, the provided area can be used for displaying other information, for example the values of element-specific metrics defined in Section 5.6. Hence, the information display can be adapted to consider a specific analysis goal.

The chosen visualization approach, providing an equally sized information area for each displayed element, allows for presenting a group of metric values or a small diagram for each displayed element. However, it causes the relative importance of the elements to be not directly recognizable in the visualization, preventing a quick analysis of similarity distribution. Furthermore, the structure diagram is less understandable when showing a large number of elements. In the subsection 5.5.3, we present two visualization approaches having the opposite properties – they graphically show relative element importance, also for a large number of elements, at the cost of providing only a limited information for each single element.

## Visualization of Code-Level Similarity

Code level  
visualization

For the analysis of text files, a text editor displaying the similarity information of individual atomic content elements (i.e. text lines) is provided in addition to the diagrams (see Figure 44). The editor can be started by selecting a file in the system structure diagram. In the editor, the similarity category of each text line (core, shared, unique, calculation) is indicated with line background coloring and a category icon – except for lines which were ignored during the analysis and consequently do not represent set model elements. For each line, a tooltip showing similarity information details is provided on user demand. Hence, the defined visualization mechanisms allow for accessing the set model



based similarity information on any level of system hierarchy – from the system structure root, representing the complete asset, down to every single atomic content element.

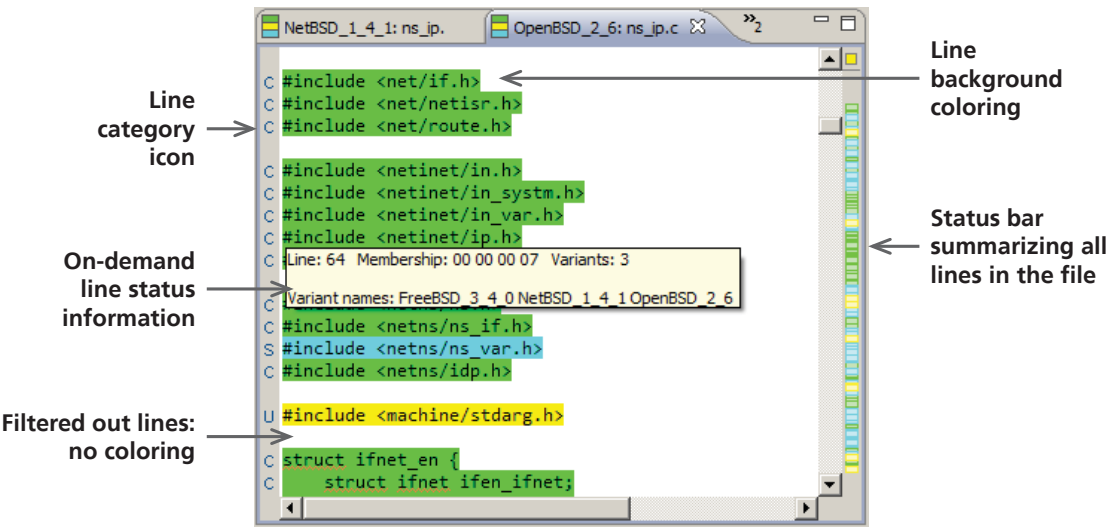


Figure 44 Visualization of code-level similarity with line background coloring, category icons, and on-demand details.

5.5.3 Visualization of Similarity Distribution

Assessing similarity distribution using package view

The UML package tree structure visualization allows for navigating the system hierarchy towards the elements that exhibit interesting concentrations of similarity. Particularly, a non-leaf tree element can only contain a high proportion of similar code if the majority of its child elements are likewise composed of predominantly similar code. For example, the *core* source folder displayed in Figure 43 only contains about 20% of non-core code. Hence, any of its child elements can only contain a low amount of such code, and the proportion of non-core code can only be high for small elements. However, the interactive exploration approach requires the user to traverse the structure tree, collect the similarity information, and reason about the relative element importance. In the process, some high similarity elements located together with groups of low similarity siblings can be overlooked, as their parent tree element exhibits a low overall similarity. Furthermore, estimation of similarity distribution inside a yet unopened tree branch is not possible. Hence, it is for example not known whether the contained atomic core elements are distributed proportionally among all child elements, or strongly concentrated inside just a few of them. To counteract the above deficiencies, we provide two visualizations of similarity distribution which do not require hierarchy traversing: a distribution diagram and a tree map.

Similarity distribution diagram

A distribution diagram for a given Structure Tree Element is constructed using the union similarity bars of all Content-Filled Elements located in the selected subtree (see Figure 45). The similarity bars are drawn vertically and placed next to each other. All bars are displayed with equal



height, still keeping the relative proportion of each similarity category in the bar, while the size of the respective element set is indicated using its width. Hence, the total width of the distribution diagram corresponds to the set union size of the selected Structure Tree Element whose children are displayed, and the area occupied by each element category on the whole diagram is proportional to the cardinality of that category. The bars building the similarity diagram can be sorted from left to right according to various criteria, e.g. the proportion of contained core code or the size of their element set. The sizes of presented elements can be either displayed individually, or, as in Figure 45, successively added to indicate the cumulative code size of all elements between the left diagram edge and the current position.

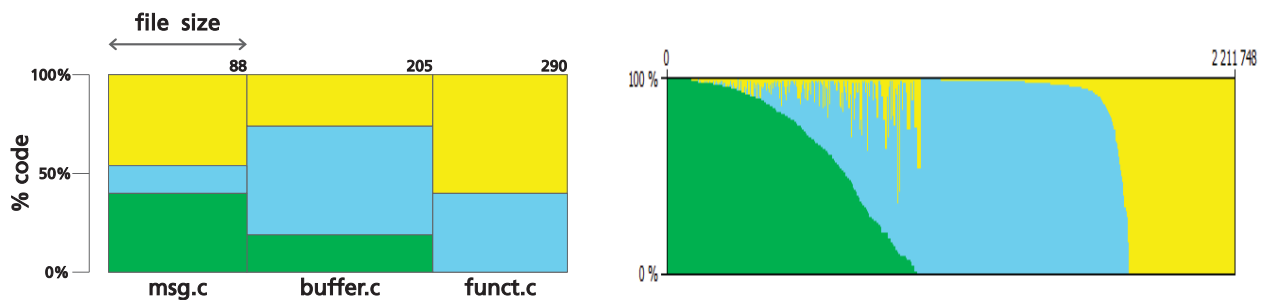


Figure 45

Distribution diagram visualization: the illustration of the construction principle, created for the *io* folder from Figure 43 (left), and a diagram screenshot for a large industrial system group (right).

The distribution diagram has two basic usage scenarios. First, large structure elements containing e.g. strongly similar or dissimilar code can be visually identified based on their width, without the need to browse structure hierarchies. Second, the border lines separating the bars on the diagram can be hidden (see the right part of Figure 45). In that case, the similar code areas shown in the diagram present curves of total similarity distribution inside the selected structure element. The analysis of these curves might help in estimation of the reuse potential and in the size assessment of element groups having certain degree of similarity. For example, in the right part of Figure 45 the files containing 100% core code, grouped near the left edge of the diagram, constitute about 5% of the total code base, and the files containing 90% or more core code constitute about 15% of the shown code. Hence, the example analyzed systems contain about 300 KLOC of code which can be transformed to a reusable form with a likely low effort.

#### Tree map diagram

Another visualization technique supporting a quick identification of characteristic tree structure elements is a tree map [Bederson 2002]. A tree map displays all elements of a tree at once (see Figure 46). The structure elements are visualized as rectangles, nested according to the tree hierarchy, and having areas proportional to their cumulative element size. Hence, a non-leaf element occupies exactly the same area as all its children, which are overlaid over it. Furthermore, each rectangle can be colored according to a metric value associated with the tree element.

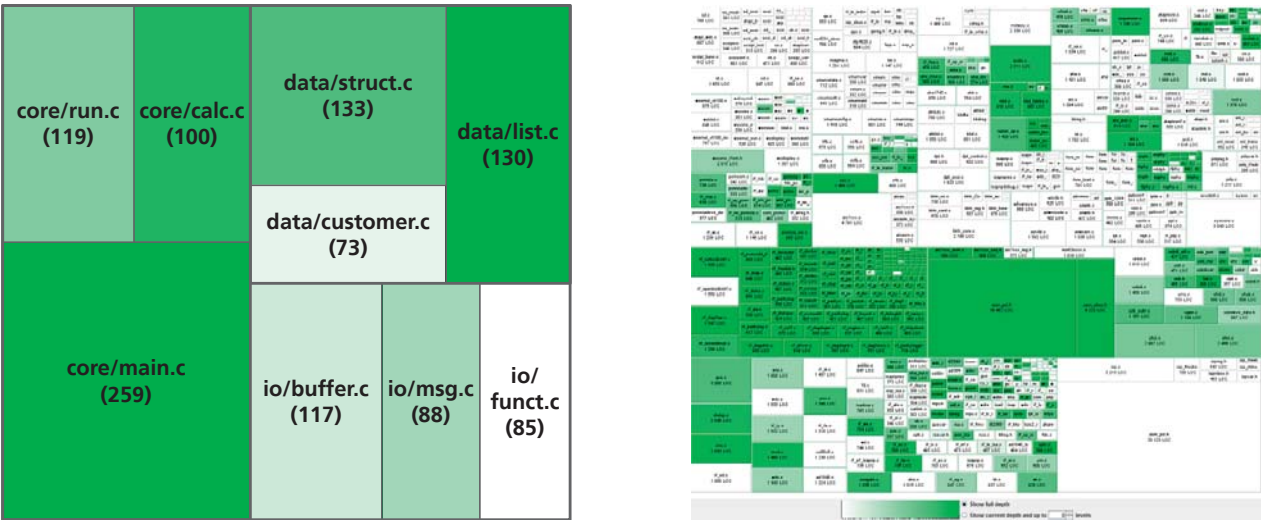


Figure 46 Tree map visualization: the illustration of the construction principle, created for the `src` folder from Figure 43 (left), and a screenshot for a code folder from three BSD systems (right). The color intensity shows the proportion of core code in the set union of a given element.

Hence, a tree map integrates the display of the system structure hierarchy, the relative element sizes, and an additional metric value shown for each element. In this way, it supports quick identification of elements with unusual sizes, unusual metric values, or both, while also indicating whether these elements are located nearby in the system structure.

Using tree map for similarity analysis

In our visualization, the size of the set union is used as displayed element size, and the elements are grouped according to the system hierarchy. The third category of visualized information, i.e. the metric, allows just for color-coding single values of enumerative or numeric type. Hence, the displayed element color can indicate either the proportion of a certain code category (core, shared, unique, calculation) in the total union code of the element, or it can be used for displaying other metrics such as those defined in Section 5.6. However, the tree map is not suitable for presenting values of more than one metric for the structure elements [Bederson 2002], which is a disadvantage when compared to the fixed-size package visualization.

As in the case of package system hierarchy visualization described in subsection 5.5.2, the input similarity information displayed in the distribution diagram and the tree map can be configured to reflect a specific analysis goal. Hence, these visualizations support presenting filtered or variant-specific perspectives on the analysis models.

Properties of different diagram types

A distribution diagram presents a compact high-level abstraction of the overall similarity distribution inside the selected subtree of the system hierarchy. Hence, it is suitable for gaining an initial overview of the distribution. The tree maps, on the other hand, are suitable for finding elements or element groups having a high value of the displayed metrics, e.g. reflecting the element variant similarity. However, both these

visualizations, due to displaying all hierarchy elements at once, can only present low amount of information for each element. Consequently, we see these diagrams as complementary to the UML-like package hierarchy visualization, as each of them provides a different perspective on the hierarchical set similarity model. Finally, presenting the similarity information of all elements in a tabular form, filtered and sorted according to user-specified criteria such as the proportion of core code, provides yet another perspective on the code similarity. While the tabular form misses most of the graphical visualization advantages, it can be used in a fashion similar to a database and quickly provide lists of sought elements, such as e.g. the files with the highest core code proportion.

### 5.5.4 Size-Preserving Set Intersection Visualization

Advantages of tree maps compared to Venn diagrams

The tree map visualization exhibits three properties which are not sufficiently supported by Venn and Euler diagrams. First, it presents a high number of diagram areas in an understandable way. Second, it graphically indicates the relative area sizes, which allows for quick visual identification of the largest areas. And third, it uses similar shapes for the shown areas, which facilitates their visual comparison.

Tree map set diagram

During a set model based similarity analysis, it is interesting to identify and compare the size proportions of the content element sets and their intersections. Hence, to support that analysis, we exploit the tree map advantages listed above by defining a tree map set diagram (see Figure 47). In the diagram, the tree map areas are used to display set intersections instead of showing structure hierarchy elements. The name of each area indicates its membership in the input sets (in Figure 47 we use binary name coding for readability), and the area size corresponds to the cardinality of the associated set intersection. Consequently, the empty intersections are not displayed, analogically as in an Euler diagram.

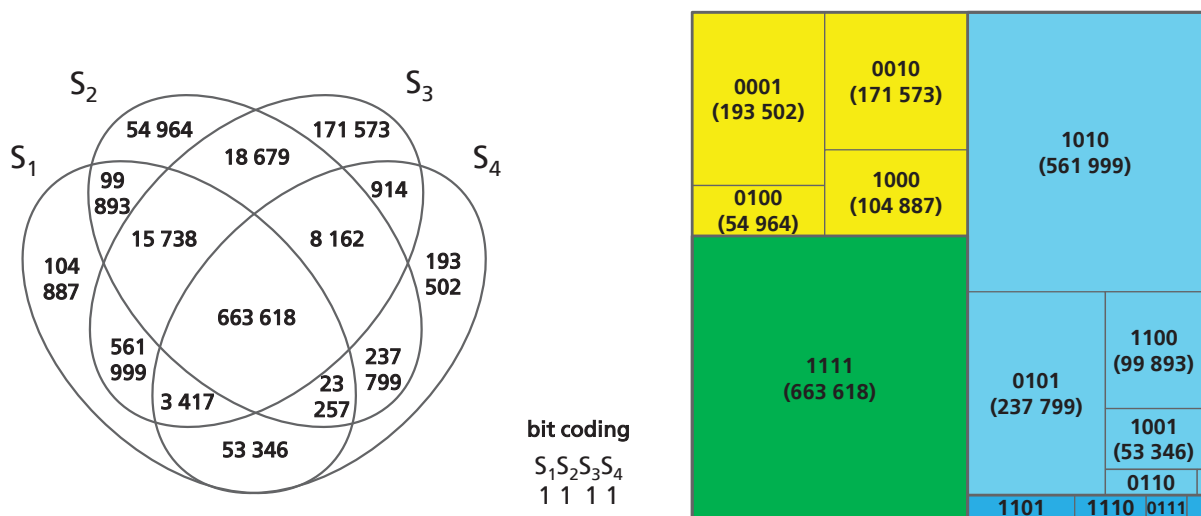


Figure 47

A Venn diagram for four industrial system variants, indicating the intersection sizes (left). The same intersecting sets visualized using a tree map set diagram (right).

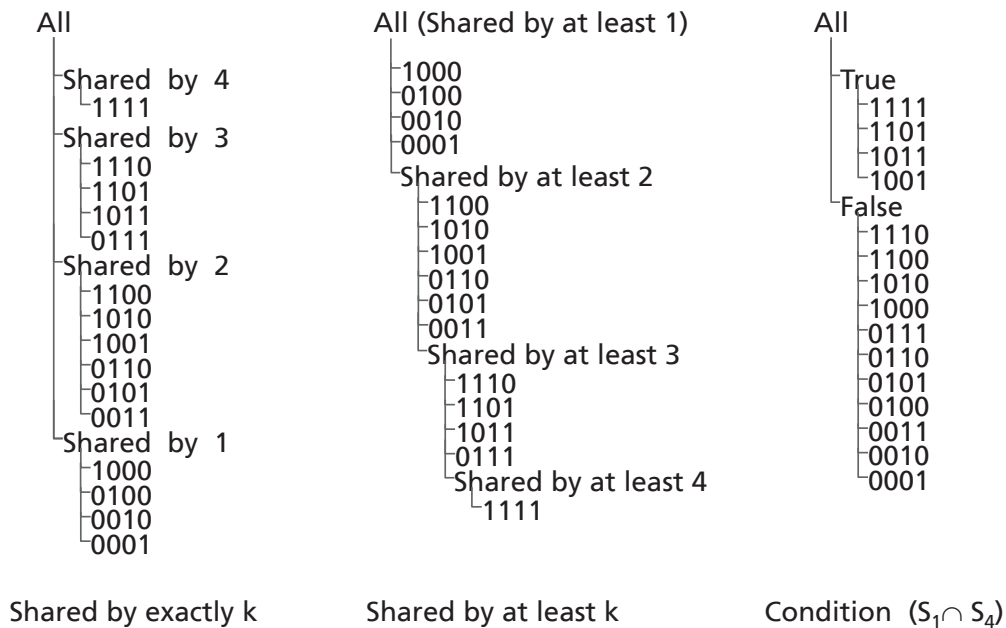


Figure 48 Example hierarchy structures, created for four sets, which can be used in a tree map set diagram.

Grouping set intersections with the tree structure

The standard tree map visualization utilizes the system structure hierarchy to graphically group the elements which are located in the same structural container. This property can also be used to facilitate the understanding of the tree map set diagram. Naturally, a group of intersecting sets does not specify a hierarchical structure. However, a tree map cannot preserve the adjacency-based layout of a Venn diagram, which groups all intersections belonging to a given set in a contiguous area. To partially compensate for this disadvantage, an artificial structure can be defined on the set intersections: for example, all intersections fulfilling a specified condition can be grouped together in a structural container. Figure 48 presents three example definitions of a tree map set structure hierarchy. The first structure, grouping the intersections shared by a specific amount of sets, is used in Figure 47.

Use of diagram area color

Furthermore, the tree map area color can also be used to support the diagram understanding. In Figure 47, we reuse the set element category colors to emphasize the structure hierarchy and create a visual reference to the coloring used in other diagram types. Naturally, as in the case of the typical tree map visualization, the element color can also be used to indicate the value of any other metric.

Visualization of a large amount of sets

The tree map set diagram is in the practice suitable for visualizing code similarity of even 20 to 30 asset variants. Although 30 sets can in theory create over  $10^9$  intersections, the amount of non-empty intersections cannot be larger than the cardinality of the set union, as each such intersection contains at least one union element. Moreover, the existence of set intersections having high cardinality, frequent in the case of cloned asset variants, further limits the possible amount of remaining intersections.

In our experience, the analyzed asset element sets rarely create more than 100 000 set intersections, and the visualization of that amount of areas is possible with the available tree map drawing tools. Finally, due to the size proportional visualization of a tree map, the small set intersections containing just a few elements are displayed as a single point or completely disappear from the diagram, and large groups of such small intersections are shown in the tree map as “boxes of sand” – areas densely filled with point-sized fragments, with visually recognizable summary size. Hence, typically a few hundreds of the largest set intersections remain identifiable on the diagram, which is sufficient for the analysis goal.

Relation to the bar diagram	The relation between the bar diagram and the tree map set diagram is analogical as the relation between the similarity distribution diagram and the structure tree map. While the bar diagram shows a compact high-level abstraction of the set similarity, providing an initial overview, the tree map set diagram delivers a detailed indication of the most relevant set intersections. Hence, these two diagrams are based on the same similarity information, but respond to different analysis needs.
The set-based and structure-based perspective	The tree map set diagram allows the human to identify the most relevant set intersections and subsequently formulate subset calculations to verify the assumptions about overall set similarity. The results of the specified subset calculations can be in turn visualized in other diagrams, such as the package hierarchy diagram and the structure tree map, to provide the information about relative distribution of the calculated element categories in the system structure hierarchy. In this way, the visualizations defined up to the present point provide two complementary perspectives on the asset similarity, focusing either on the similarity of element sets, or on the projection of that similarity onto the asset structure.

### 5.5.5 Similarity Visualization with Phylogenetic Trees

The information stored in the set model can be used to answer a range of questions regarding the relative similarity of the input asset variants. For the frequent questions, it is worthwhile to automate the typical user steps leading to an answer and define suitable answer visualization. We define two such visualizations, based on phylogenetic trees, in a joint work with Vasil Tenev [Tenev 2012].

#### Similarity Clustering of Element Sets

A frequent analysis concern is the identification of input variant pairs or groups which exhibit particularly high similarity. To automate the search for similar variant groups, similarity based clustering of the variant sets needs to be constructed and visualized. Dendrograms are a form of phylogenetic trees frequently used for this purpose. Hence, we construct a dendrogram using the set similarity information as input.

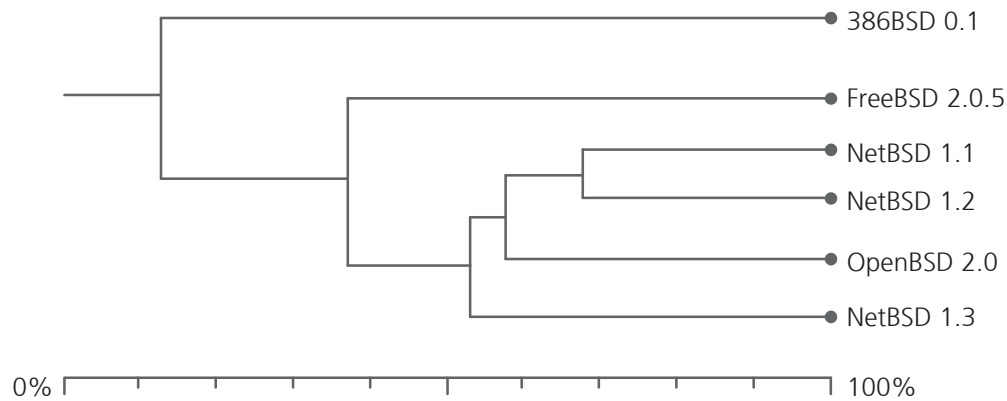


Figure 49 Dendrogram constructed for the full source code of six BSD systems [Tenev 2012]. The location of branching points corresponds to the similarity of the respective tree branches.

**Dendrogram construction** A dendrogram visualizes the input asset variants as branches of a tree (see Figure 49). The construction algorithm starts from the top of the tree, where all the branches exist separately. In each iteration the algorithm finds two most similar branches and merges them, indicating their relative similarity with the location of the drawn merge point. Finally, only the tree trunk remains and the clustering is complete.

**Interpretation** The constructed tree visualizes the identified variant groups, using the branch layout, and indicates the relative similarities between and within the groups, using the branch length. Hence, an analyst can intuitively identify the similar variant groups, as well as the outlier variants with low similarity (for example, see the outlier 386BSD 0.1 system in Figure 49). Moreover, it is simple to assess whether the similarity of a variant group is high enough for the analysis goal. A dendrogram can also be a convenient starting point for formulating subset calculations, subsequently visualized in other diagrams.

**Possible inconsistencies** The dendrogram construction algorithm is based on the pairwise similarity distance between the tree branches. Hence, the information stored in a set model is sufficient for tree construction, and the dendrogram can be drawn for any set model, available on any structure hierarchy level in the analysis results. However, as discussed in Section 4.2.2, the pairwise similarity information does not adequately reflect all details of the similarity arrangement between the input sets. Moreover, several algorithms for branch weighting exist. The calculated weighted similarity of two groups can hide much higher or lower similarity values existing between some of member variants, depending on the algorithm. Finally, two variants with a minimally lower similarity value can be assigned to two different clusters, if only other variants having minimally higher similarity to these variants exist. Therefore, a dendrogram provides a helpful indication of the general clustering tendency between the input variants, but the gained insights need to be verified using more precise analysis means such as the quantitative results of the respective subset calculations.



## Similarity-Based Reconstruction of Probable Evolution History

The role of evolution history information

Another analysis concern, particularly relevant for asset variant groups having a long evolution history, is the identification of evolutionary relationships between the variants. Reconstruction of the past history of variant cloning can provide important indications for subsequent reuse migration decisions, regardless of the current variant similarity:

- Estimation of the degree of change, experienced by a given variant after the cloning, indicates the intensity of past maintenance activities. Reuse introduction is more profitable for variants experiencing intensive maintenance, as the reuse reengineering investment pays off faster in their case.
- It is helpful to identify whether the input assets group contains only variants, developed in parallel, or if some of them are in fact asset versions, developed one after another. In the embedded systems domain, many versions of the same asset variant might need to be maintained, as the customers using the earlier versions might demand bug fixes while rejecting major version updates. However, in most cases only the latest version of a given variant is relevant for reuse migration – the maintenance of the earlier versions is typically either planned to cease in the near future, or requires only a low effort compared to the new version development.

Differences between version and variant similarity

As discussed in Section 4.1, the versions of an asset form a time-ordered list, while the variants are developed in parallel and cannot be ordered using an objective criterion. This property is also reflected in the similarity existing in a group of versions as compared to a group of variants: versions are similar to each other in a different way than variants. The last existing version  $k$  is typically the most similar to version  $k-1$ , a little less similar to version  $k-2$ , and so forth – the time ordering of versions can be hence reconstructed based on their similarity. In contrast to that, the similarity of variants is much more symmetrical and cannot be reduced to a linear form.

Figure 50 depicts an example of a set similarity model for three variants and three versions, using area-proportional Venn diagrams. The version  $S_1$  was developed as the first, and the version  $S_3$  as the last. Consequently, they are less similar to each other than to the intermediate version  $S_2$ . Furthermore, their set model is also characteristic, as explained by the evolution-based interpretation of the particular set intersections:

- The intersection  $S_1 \cap S_2' \cap S_3$  represents code that existed in version  $S_1$ , was removed in  $S_2$ , but was again added in  $S_3$ . This is not typical, as the majority of code which was once removed does not appear again in subsequent versions. Consequently, this set intersection is very small.
- Similarly, the intersection  $S_1' \cap S_2 \cap S_3'$  represents temporary code, added in  $S_2$  but removed in  $S_3$ . Again, this intersection is relatively small.
- In contrast to that, the intersection  $S_1' \cap S_2 \cap S_3$  is much larger, as it represents the code added in version  $S_2$  which remained in use in  $S_3$ .



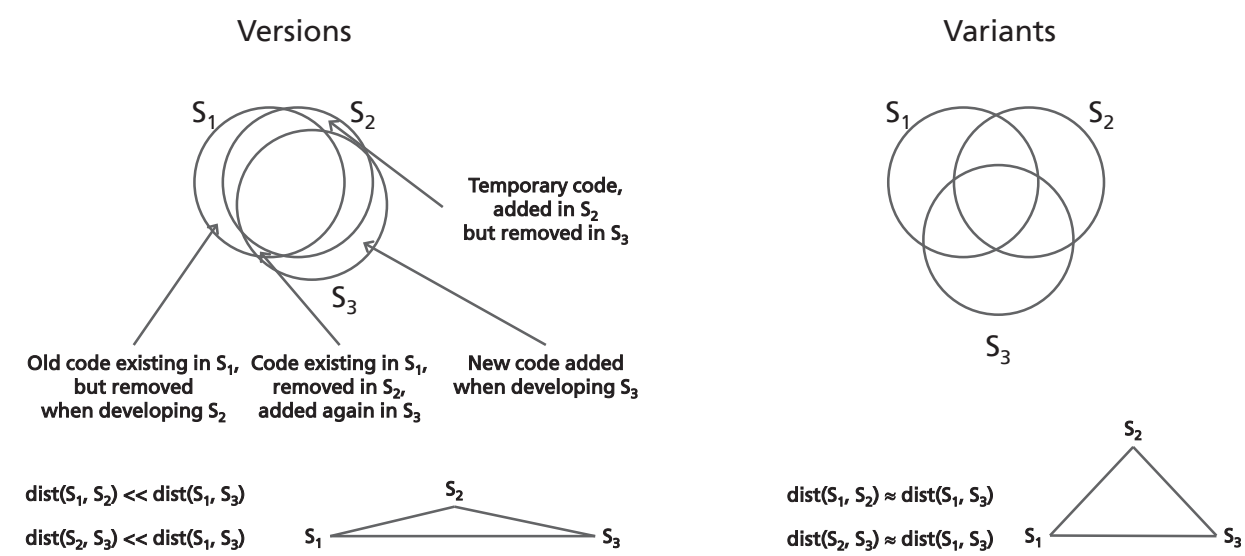


Figure 50 The similarity of a group of versions (left) as compared to a group of variants (right). The drawn Venn diagrams are area-proportional, i.e. the size of an area indicates its cardinality.

In contrast to the set model of versions, the model constructed for variants is much more symmetrical due to their parallel evolution. For example, in Figure 50 the set intersections  $S_1 \cap S_2 \cap S_3$ ,  $S_1 \cap S_2' \cap S_3$  and  $S_1' \cap S_2 \cap S_3$  have comparable sizes. While the set model of versions can be linearized by filtering out the set intersections having negligible sizes, this is not possible for the set model of variants. Hence, this property can be used to distinguish asset variants from versions.

Cladogram construction

We use the set model information to identify the evolutionary relationships among a group of input asset variants (or versions) by constructing a cladogram, which is a type of phylogenetic tree designed to visualize this type of information (see Figure 51). We start by filtering out all set intersections with cardinality falling below a defined threshold – by default, 1% of the union code size. Then, we construct a Hasse diagram of the remaining intersections and lay out the diagram as a tree, with branch lengths proportional to the sizes of particular intersections.

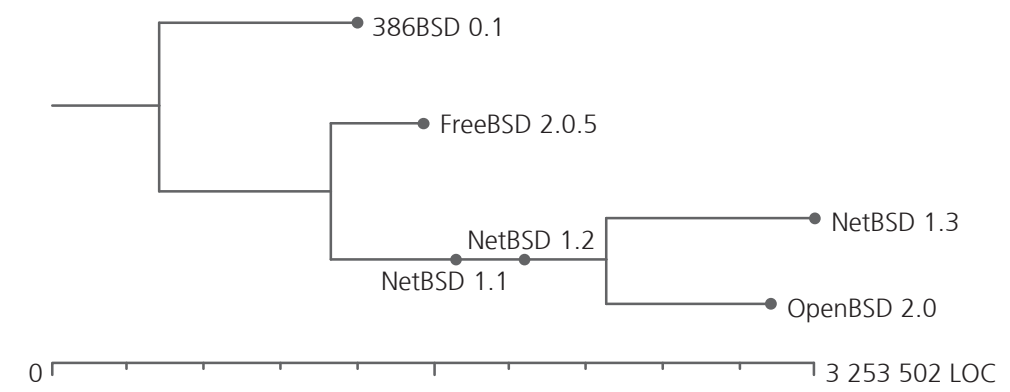


Figure 51 Cladogram constructed for the full source code of six BSD systems depicted in Figure 49 [Tenev 2012]. The length of branch sections is proportional to the amount of common or unique code.

In most cases, the Hasse diagram can be unambiguously reduced to a tree, although sometimes a higher value of the filtering threshold might be needed to remove further intersections. If the transformation is not possible, the alternative solutions can be displayed as parallel, alternative tree branches, hence proposing more than one tree location for a given variant. The existence of alternative tree branches indicates that the involved assets are variants which contain large code parts shared by some, but not all of the parallelized variants. This can happen for example if significant code parts were exchanged between the variants not having a direct common cloning ancestor.

The cladogram construction algorithm assumes that the software grows with time, i.e. a later software version contains more code than an earlier one. In most cases, this assumption is correct (see for example the three consecutive versions of the NetBSD system in Figure 51), although the opposite case of shrinking software can also exist. Moreover, in some cases the difference between two analyzed variants or versions can be smaller than the defined set intersection filtering threshold. In such a situation, these assets are represented by a single tree location, labelled accordingly with both asset names.

**Interpretation** The cladogram helps to distinguish asset versions from variants, and indicates the relative changes between them. However, as already discussed in Sections 4.1.1 and 5.4.2, the similarity of asset variants and versions might not always correspond to their real evolution history. For example, the system OpenBSD 2.0 branched off from NetBSD 1.1 (see Figure 2), and not from NetBSD 1.2 as indicated in Figure 51. Hence, the information displayed by a cladogram should be mainly used as an approximation of the evolution history when a reliable history record cannot be provided from other sources. Analogically, the similarity distance between two variants should not be confused with their maintenance intensity, as different variants might have different evolution paces. Instead, the maintenance intensity of a given variant should be estimated by comparing the degree of change with the length of time period a given variant exists.

### 5.5.6 Discussion

**Visualization categories** The visualizations presented in the current Section can be assigned to three categories: visualizations of set similarity (set bar diagram, tree map set diagram), projections of the set similarity onto the system structure (package hierarchy diagram, code similarity view, distribution diagram, tree map), and visualizations peculiar to a specific analysis question (dendrogram and cladogram). The visualizations present the hierarchical set model from various complementary perspectives, and are available for any set model existing at any level of the system structure hierarchy.

Customizable and interactive diagrams	The set similarity and system structure visualizations can be modified by the use of subset calculations, visually emphasizing the set elements fulfilling a specified logical condition. Furthermore, the similarity information used in the system structure visualizations can be filtered, displaying e.g. only the structure elements having specified sizes, similarity properties, or variant membership. Hence, a specific diagram type can be used many times, also in an interactive way, to display different perspectives on the input asset variant similarity.
Tabular information export	The visualized information can be alternatively provided in a textual, tabular form. For example, the bar diagram section cardinalities, lists of hierarchy elements with associated similarity information and metric values, and even the complete table of all identified element equivalence classes can be displayed, exported, and processed with external tools such as spreadsheets and databases. This enables the user to filter and sort the information according to further criteria, also these which are not supported in the defined visualizations.
Realization of the construction requirements	<p>Visualization of analysis results needs to comply with the same principles and construction requirements as the analysis algorithms themselves. In particular, the discussed diagrams were defined according to the analysis construction requirements described in Section 4.2:</p> <ul style="list-style-type: none"><li>• Complying with the requirements C1 (commutativity) and C2 (associativity), the visualizations do not distinguish any of the input variants and treat all of them with equal importance, unless specified otherwise by the user.</li><li>• Information on all possible combinations of the variant intersections is available (requirement C3), and it is provided on any system hierarchy level down to the single atomic content elements (requirement C5 – traceability).</li><li>• The requirement C4 (information detail level) is achievable if the asset structure decomposition distinguishes sufficiently detailed atomic content elements. With this condition fulfilled, the defined visualizations comply with the requirement C4.</li><li>• The requirement C6, i.e. a result abstraction defined in a uniform and scalable way regardless of the analyzed asset size and the number of analyzed asset variants, is supported.</li><li>• Indication of the relative size of variant sets and their intersections (requirement C7) is supported.</li></ul>
Further diagram types for set models	Further visualizations of the hierarchical set similarity model information, for example using statistical graphics techniques such as histograms or scatterplots, can be defined. These diagram types could extend the available perspectives on the set similarity, supporting the identification of tendencies and correlations or the search for outlier elements. In the future work we intend to investigate the further possible visualizations and evaluate their resulting usefulness for the similarity analysis.

## 5.6 Metrics

The need for model metrics	<p>The set model provides detailed information on input asset variant similarity, which can be used in the context of the defined application scenarios. The provided information is quantitative, and calculations performed on that information such as e.g. the calculation of core content proportion can be used to select the asset elements suitable for the analysis goal. However, in some cases the set model information needs to be further examined to provide a more precise basis for result interpretation and reuse migration decisions, or to otherwise characterize the set model and the system structure. Consider the following examples:</p> <ul style="list-style-type: none"> <li>• For a given source code file, containing e.g. 50% core code, the estimated migration difficulty might depend on the distribution of that code inside the file. A code composed of a single block, covering 50% of the file, is likely easier to migrate than strongly fragmented core code located in every second line of the file.</li> <li>• Analogically, a file where the code belongs to many different set intersections might be more difficult in the migration than a file containing a low amount of intersections, despite the same degree of overall similarity existing in both files.</li> <li>• The same observations apply to the higher structure level of source folders and whole assets. For example, an asset containing 50% core code, strongly concentrated in just a few files, might be easier to migrate than an asset where the core code is distributed proportionally and fills 50% of each code file.</li> </ul>
Three metrics categories	<p>Consequently, we define a group of metrics, characterizing the fragmentation and distribution properties of the hierarchical set model, which are intended to support the reuse decisions by providing additional relevant information. Based on the metric calculation basis, we divide the metrics into three categories: metrics characterizing a group of intersecting sets, metrics characterizing the distribution of these sets in the structure hierarchy, and metrics specific to the textual file system based asset decomposition. For brevity, we only catalogue and discuss the primary metrics, i.e. the metrics calculated directly on the model information. Naturally, in the analysis the metric values can be further combined, and their values can be aggregated (e.g. as the average or extreme values) and related (e.g. as values relative to element size or other metrics). The values of all three metric categories can be visualized on the structure diagrams described in the previous Section. The metric values can also be exported in a tabular form, processed (sorted, filtered) with external tools, and visualized with other diagram types such as histograms and correlation charts.</p>
The set metrics	<p>Let's denote the amount of analyzed variants as <math>N</math>, the variant sets as <math>S_i</math> (where <math>1 \leq i \leq N</math>), the equivalence set union for the sets <math>S_i</math> as <math>U</math>, the set of all equivalence set intersections for the sets <math>S_i</math> as <math>P</math>, and the set of all elements of unified structure tree as <math>T</math>. Using that notation, in Table 7 we define four metrics characterizing a group of intersecting sets.</p>

Metric	Calculation Formula	Value Range
Number of Non-Empty Set Intersections (NNEI)	Count the intersections: $\text{card}(\{p \in P : \text{card}(p) > 0\})$	$1..2^N-1$
Size of the Largest Intersection (SLI)	Select the largest intersection: $\text{MAX}(\text{card}(p \in P))$	$1.. \text{MIN}(\text{card}(S_i))$
Entropy of the Set Intersection Sizes (ESIS)	$\sum [p \in P] [ - (\text{card}(p)/\text{card}(U)) / \log_2(\text{card}(p)/\text{card}(U)) ]$ Normalized: $\text{ESIS} / \log_2(2^N-1)$	$0.. \log_2(2^N-1)$ Normalized: $0..1$
Relative Set Union Size (RSUS)	$(\sum [i=1..N] \text{card}(S_i)) / \text{card}(U)$ Normalized: $(\text{RSUS}-1) / (N-1)$	$1..N$ Normalized: $0..1$

Table 7

The metrics characterizing a group of intersecting sets

- Number of Non-Empty Set Intersections expresses the amount of different intersections which need to be considered to fully cover all content elements. A high number of intersections indicates a relatively higher complexity of the similarity arrangement and a higher migration effort, as each intersection potentially needs to be dealt with separately.
- Size of the Largest Intersection is also relevant for the estimation of migration effort. In case even the largest set intersection is relatively small, the variant code is strongly fragmented between different intersections and hence more complex from the reuse point of view.
- Entropy of the Set Intersection Sizes measures the relative distribution of content elements among the set intersections using the Shannon entropy [Shannon 1948]. The calculated entropy value equals 0 if all elements belong to just one intersection, and achieves the maximal value if the elements are distributed evenly, i.e. every set intersection has the same size. The metric can consider either all  $2^N-1$  set intersections, or only the non-empty ones – naturally, this needs to be indicated when reporting the metric value. A variant code concentrated in just a few set intersections, i.e. having a low entropy, is likely easier to transform into a reusable form.
- Relative Set Union Size (RSUS) describes the theoretical reuse potential of the analyzed sets, achievable when each equivalence class is replaced by a single, reusable content element. The metric expresses the maximal factor of the input content size reduction achievable through reuse – hence, the higher values indicate more reuse potential. The minimal RSUS value of 1 is only achievable for completely disjoint input sets, containing only *unique* elements, while the value of N indicates that each element of the N input sets is a *core* element. Note that the RSUS metric is also influenced by the *shared* content elements and the amount of sets these elements belong to.

The set metrics defined above can be applied to any group of intersecting sets – for example, the metrics values can be listed for the content sets of all hierarchy structure elements in order to support the identification of the elements interesting for further analysis. The above metrics are equally applicable to the atomic content element sets as well as to the sets containing the structure tree elements.

Metric	Calculation Formula	Value Range
Structure Elements Containing the Intersection (SECI)	Given $p \in P$ , count the structure elements containing $p$ : $\text{card}(\{t \in T : \text{card}(p \text{ in } t) > 0\})$	$0.. \text{card}(T)$
Intersection Entropy in the Tree Structure (IETS)	Given $p \in P$ and denoting $(p:t)$ as the proportion of $p$ in the content of $t$ : $\sum_{t \in T} [ - ((p:t) / \sum_{t \in T} (p:t)) / \log_2((p:t) / \sum_{t \in T} (p:t)) ]$ Normalized: $\text{IETS} / \log_2(\text{card}(T))$	$0.. \log_2(\text{card}(T))$ Normalized: $0..1$

Table 8

The metrics characterizing the distribution of the variant sets in the structure hierarchy

#### Set distribution metrics

In Table 8 we define two metrics characterizing the distribution of the intersections of atomic content sets in the structure hierarchy. Both these metrics are calculated for a specified set intersection – it can be a single intersection, as well as a group of such intersections selected by a subset calculation. Because only Content Filled Elements contain the atomic content elements, the Structural Containers are not included in the calculation of the metrics value. The distribution metrics can be calculated for all Content Filled Elements in a tree, or for a specific selection of them, for example for the elements located in a given tree branch.

- Structure Elements Containing the Set Intersection counts the Content Filled Elements in which at least one atomic content element belonging to the given intersection is contained.
- Intersection Entropy in the Tree Structure measures the relative distribution of the intersection elements in the structure tree. The entropy value equals 0 if the intersection only exists in one Content Filled Element, and achieves the maximal value if the intersection is distributed evenly, i.e. the atomic intersection elements proportionally fill the content of all tree elements. The calculation uses the relative proportions of the intersection size  $(p:t) / \sum_{t \in T} (p:t)$ , and not the absolute cardinalities  $\text{card}(p \text{ in } t) / \text{card}(p)$ . The reason for that is that Content Filled Elements have different sizes themselves. In case the specified intersection has equal absolute cardinalities in the Content Filled Elements having different sizes, these elements are filled with the intersection content to a different proportion, and the intersection is hence not distributed evenly.

Generally, set intersections having lower SECI and IETS values are concentrated and located in only a part of the asset, and might be therefore easier to manage in reuse migration.

#### Textual decomposition metrics

Finally, in Table 9 we define three metrics which are specific to the textual file system based asset content decomposition. In that decomposition, the content lines in text files are ordered, and their order is meaningful for the application scenarios as the small functionality fragments (e.g. methods) are implemented using consecutive code lines. The fragmentation of the text line list with regard to the similarity information is therefore relevant for the assessment of reuse migration or parallel maintenance difficulty.



Metric	Calculation Formula	Value Range
Number of Fragments (NF)	Count the fragments: $\text{card}(F_i)$	$1..\text{card}(S_i)$
Size of the Largest Fragment (SLF)	Select the largest fragment: $\text{MAX}(\text{card}(f \in F_i))$	$1..\text{card}(S_i)$
Fragment Entropy (FE)	$\sum [f \in F_i] ( - (\text{card}(f)/\text{card}(S_i)) / \log_2(\text{card}(f)/\text{card}(S_i)) )$ Normalized: $\text{FE} / \log_2(\text{card}(S_i))$	$0..\log_2(\text{card}(S_i))$ Normalized: $0..1$

Table 9

The metrics characterizing similarity distribution in the textual file content

Consequently, in the defined metrics we use the concept of a fragment, which is a group of consecutive lines belonging to the same set intersection. Each file can be unambiguously divided into such fragments, and there exists at least one fragment in each file. In the case of the set union, the file fragments do not exist physically. Hence, the above metrics are calculated separately for the elements of each variant tree, and their values for the union tree are provided as the maximum or minimum of the variant values, depending on the analysis goal. In Table 9, we denote the set of all fragments located in the files belonging to the considered variant structure tree or a tree branch as  $F_i$ . The defined metrics are:

- Number of Fragments counts the fragments existing in a given file. For a folder, it reports the sum of fragment counts of the contained files.
- Size of the Largest Fragment reports the largest fragment in a file. For a folder, it returns the sum of largest fragment sizes of the contained files.
- Fragment Entropy measures the relative distribution of the file content in the fragments: the entropy equals 0 if the code is concentrated in one fragment, and achieves the maximal value if the code is composed entirely from single-line fragments. Hence, it expresses the overall fragmentation of the code. The metrics calculation is identical for files and for folders.

The above fragmentation metrics are intended to support the estimation of relative difficulty of the reuse migration and parallel maintenance tasks. A lower code fragmentation and a larger fragment size might indicate a lower transformation or maintenance effort, as the arrangement of existing code similarities is in that case relatively less complex and easier to understand and manage.

#### Interpretation

In general, the software metrics provide a convenient way to identify locations or elements in a software system exhibiting a sought property (e.g. complexity, maintainability). However, as the given property is usually influenced by more context factors than these included in the metric calculation, the elevated metric values might in the practice not always correlate with the elevated occurrence of the sought software property [Lanza 2006]. Hence, a review of the identified locations is still needed to confirm their suitability for the analysis goal. This applies likewise to the metrics defined in this section. Nevertheless, it is an interesting future work to evaluate in the practice the predictive power of these metrics.



## 5.7 Discussion

**Benefits: generality** The generic similarity analysis approach, discussed in this Chapter, is instantiated for a specific asset type by defining three analysis mechanisms: a decomposition of the asset content into structural and atomic content elements, an equivalence relation on the structural elements, and an equivalence relation on the atomic elements. The resulting analysis instantiation decomposes the content of input asset variants and expresses their similarity in the form of a hierarchical set similarity model. Accompanying the model, a group of defined visualizations and metrics supports the navigation and interpretation of the analysis results. The defined analysis and visualization concepts are generic, i.e. independent of the analyzed asset type. The generic nature of the approach is its strong advantage, since it broadens the possible approach application scope and allows for customization of the analysis according to the peculiarities of a given asset type.

**Benefits: abstraction and detail level** The similarity analysis approach is defined in conformance with the construction requirements discussed in Section 4.2. In particular, the approach and the constructed set model do not distinguish any of the analyzed variants, nor do they assume a specific variant order. Furthermore, the set model based similarity analysis fulfills all the criteria concerning the nature of the provided similarity information, discussed in Section 3.3, which are not addressed in their entirety by the other related approaches. In short, the stated criteria demand that full information details, as well as suitable abstraction mechanisms, are provided for the two dimensions of analysis problem complexity: the asset size (which can range to millions of code lines) and the amount of asset variants (where many tens of variants are possible):

- Regardless of the size of a structural asset or asset element, and of its location in the structure hierarchy, the similarity of the asset variants can be uniformly presented in the form of intersecting content element sets. At the same time, the similarity information is traceable to each individual asset content element. Given a sufficiently restrictive equivalence function, the stored similarity information is also precise in the sense used in Section 3.3 and the construction requirement C4, i.e. it enables distinguishing even small, but meaningful differences between the asset variants.
- The set model expresses the similarity of even a large number of input variants in an understandable way. At the same time, the similarity information concerning any subgroup of the analyzed variants is readily accessible, and is available on any system hierarchy level down to the single content element.

Benefits  
described by  
the hypotheses

Consequently, we consider the hierarchical set model and the visualizations defined on top of it to be the main factor which enables the achievement of a scalable abstraction of the analysis result, despite preserving the full information detail, for both dimensions of the system size and the amount of variants. Furthermore, the scalable abstraction and the fulfillment of the remaining similarity information criteria discussed in Section 3.3 constitute the core reason for the benefits of our analysis approach as compared to the current state of the art. Hence, the set model is the main factor contributing to the fulfillment of practical and scientific improvements specified in our hypotheses in Section 3.4: reduced migration and maintenance effort, less missed reuse opportunities, and a faster and more correct assessment of asset variant similarity. In Chapter 7, we continue the discussion on the approach benefits by presenting the evaluations of the research hypotheses.

Limitations:  
divergence  
from the input  
similarity

The construction of the similarity model in the form of intersecting content element sets is only possible due to the restriction of the analysis scope, as defined by the analysis assumptions in Section 4.3. In particular, the set model exhibits two properties which limit the type of the storable similarity information: it does not consider similarities existing within a single set, and it requires that the similarity relation between the content elements is transitive. Therefore, the similarity expressed in the set model can differ from the original similarity existing in the input assets in two ways: some of the input similarity might be not reflected in the set model, while some additional similarity might be artificially added to the model (e.g. by constructing a transitive closure of the input similarity relation).

The divergence between the set model similarity and the input similarity constitutes in our opinion an acceptable price paid in exchange for the benefits of using the set model. Especially in the context of the application scenarios intended for the analysis approach, which motivate the analysis assumptions and the resulting limitation of the model usage scope, the similarity divergence is either occurring mainly for non-relevant information, or it can be influenced to reduce its effect on the analysis goals:

- As discussed in Section 4.3, the similarities existing inside the same set are not in the focus of the application scenarios – therefore their absence in the set model is acceptable.
- The amount of artificial similarity added to the result can be minimized when choosing the analysis algorithms, at a possible cost of ignoring a part of the original similarity information. The optimization towards minimal or none artificial similarity leads to a creation of analysis result which has a high certainty, which saves the human effort as there is no need to verify the result manually. At the same time, the possible omission of a small amount of similarity relations, indicating potentially reusable element pairs, constitutes a comparatively much lesser concern: although the reuse potential of these elements is not exploited, overlooking their similarity does not induce additional analysis or migration effort.

We consider the benefits of the set model to be much more significant than its drawback of the restricted similarity information form, unless the resulting divergence from the input similarity is large enough to hinder the achievement of analysis goals. In the case of textual file system based analysis instantiation, using the diff algorithm, the divergence is relatively small (see Section 7.1) and the analysis goals are achieved in the practice (Section 7.3 and 7.4). However, an analysis instantiation for any asset type needs to be evaluated separately: the overall utility of the set model might vary for different asset types, depending on the degree to which the input similarity of these types is transitive. Furthermore, the properties of the used content decomposition and equivalence relations need to be known during result interpretation, as they likewise influence the utility of the approach instantiation.

The role of transitivity

The property of transitivity, required in the construction of the set model, significantly influences both the benefits and drawbacks of the presented approach, and hence deserves a further discussion. The divergence between the set model and the original similarity is, to a large degree, created in the process of finding the transitive similarity relation most closely resembling the provided input. Hence, the negative consequences of that divergence can be to a large extent attributed to the transitivity requirement. However, the construction of a set model is not possible without transitive similarity, so that the set model benefits are also only achievable due to the transitivity.

Transitivity simplifies result interpretation

The main advantage of the transitive similarity form is that the interpretation of a transitive similarity analysis result is much easier than of a non-transitive one. Since any element of a transitively similar group is similar to any other element from that group, knowing the group members is sufficient to fully understand their similarity. In contrast to that, interpretation of a result which is not known to be transitive requires analyzing the topology of pairwise element similarity relations – a tedious task, which is complex and likely to induce mistakes when the group contains many tens of elements. In a controlled experiment, described in Section 7.2, we evaluated the effect of using the set model for presenting the similarity information of five source file variants as compared to presenting the same information in the pairwise form. The set model group performed the experimental tasks over twice as fast on average, while making over 90% fewer mistakes (see Section 7.2 for details and discussion). Between the two groups, the strategy of solving the (identical) experimental tasks mainly differed in the usage of result transitivity. Hence, in our opinion the experiment result supports the above claim of better understandability of the transitive similarity results. Admittedly, the experimental variable varied between the participant groups had a broader scope, as the complete underlying similarity model (set or pairwise) was varied. However, transitivity plays a central role in the interpretation of the set model information, and it is likely that it substantially contributed to the measured result.

Transitivity  
enables  
scalable  
abstraction

Furthermore, note that even for a large number of content elements the similarity information stored in the set model is still available in its entirety. In contrast to that, the existing approaches using non-transitive similarity information, e.g. provided by clone detection, face the tradeoff between presenting the complete input information, which cannot be understood by a human in reasonable time when the number of elements is large, and abstracting that information in the form of pairwise similarity matrices or scatterplots, which by necessity ignore the topology of the similarity relation graphs. Hence, in the abstracted pairwise result it is not possible to recognize how the elements similar between variants A and B relate to the elements similar between B and C, and whether in both cases the same or different elements of variant B are reported (see Section 4.2.2). Hence, although the information provided by the non-transitive approaches is technically fully correct, their abstraction mechanisms cannot express an important part of that information.

Transitivity of  
maintenance  
activities

Finally, the further activities performed on the identified similar asset part variants, such as merging their implementations to a single-copy reusable asset, are in most cases transitive in their nature. Hence, the transitive form of the analysis result corresponds to the nature of similarity relations sought in the context of the given application scenarios, and can be directly utilized in the consequent maintenance tasks.

## 5.8 Summary

Based on the formalization of variant similarity analysis described in Chapter 4, in this chapter we presented the main contribution of this thesis: the definition of the similarity analysis approach based on the idea of hierarchical set similarity models. We described the construction of a set similarity model with the use of an equivalence relation, and listed three analysis mechanisms which need to be defined in order to make the analysis applicable for a specific asset type. Subsequently, we integrated the set model with a tree-based model of a system structure hierarchy, hence creating the hierarchical set similarity model. The model uses set algebra to express the similarity of both atomic content elements as well as the system tree structures, and allows for evaluating the analyzed variant asset similarity at any level of the structure hierarchy.

Using the properties of hierarchical set model, such as the need to establish variant structure correspondences before analyzing the atomic content elements, we motivated a generic similarity analysis process. In turn, we described the concepts and algorithms used in the respective process phases. We devoted particular focus to the set model construction, using two forms of input similarity data, and to the visualization of intersecting set similarity, allowing for understandable presentation of a high number of intersecting sets and a high number of structure hierarchy elements. Furthermore, we defined several metrics calculated on the set model information, intended to support the interpretation of the analysis results in the context of the application scenarios. Finally, we discussed the benefits and drawbacks of the presented analysis approach.

## 6 Analysis Tool Implementation Techniques

The purpose of this short Chapter is to present selected implementation techniques for the Variant Analysis approach. As previously discussed, it is required that the capabilities of the analysis tool scale according to the problem dimensions occurring in the practice: the input asset variants can contain millions of code lines, and many tens of variants can exist. At that problem scale, the structure and processing of the analysis data model need to be optimized to avoid performance problems. Hence, we address the optimization of the data structures storing the set model in Section 6.1, and the processing of these structures in Section 6.2. Furthermore, in Section 6.3 we discuss a technique to ensure a repeatable and consistent selection of analysis result despite the presence of multiple optimal solutions to a given analysis problem.

### 6.1 Supporting Performance Optimization with Data Redundancy

The construction of the set model requires that the asset content elements identified as similar are assigned to the same equivalence class. In the further processing, that assignment information is accessed in various ways. For a detailed analysis of the asset similarity, the exact locations of all elements equivalent to a given one are needed. Visualizations mainly use the statistical information on each equivalence class, i.e. its cardinality and the set membership of contained elements. Finally, subset calculations are also based on the set membership information. The structure of forward and backward references between the elements and their equivalence class, specified by the data metamodel in Section 5.2, needs to be supported with additional, redundant information to prevent performance problems:

- A unique integer identifier from the 0..N-1 range is assigned to each of the N input asset variants. This *storage order* of variants does not influence the analysis result, but is only used to determine which object in a given list refers to which input asset variant. For example, the equivalence class object maintains a resizable array, storing pointers to the member elements at the positions referring to their asset identifiers (or null pointers if there is no equivalent element in a given content set). Hence, the pointer structure has a size of N and allows for constant-time access to any of its elements.
- The membership information, listing the content sets where the equivalence class elements belong to, can be efficiently processed when stored in a bit vector. The vector abstracts the pointer array described above: at the respective bit positions, only ones and zeroes remain to represent the element pointers and null pointers. In our tool we use a 32-bit integer to encode the bit vector, as supporting the amount of up to 32 variants was in the practice sufficient for all performed analyses.

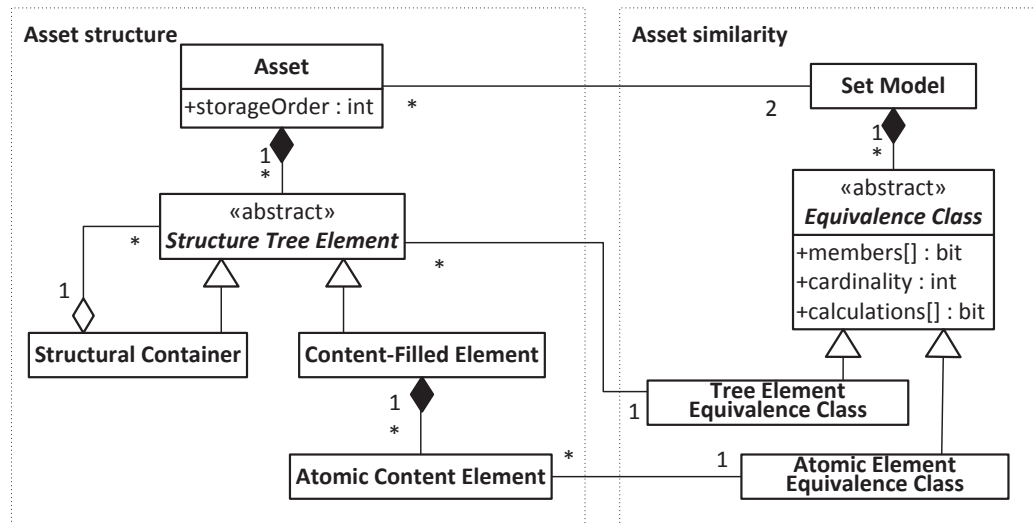


Figure 52

A data metamodel from Figure 31, with the additional attributes storing redundant, performance-relevant set model information.

- Furthermore, each equivalence class explicitly stores its cardinality value. Although that value is redundant, as it can be calculated using the pointer array or the bit vector, the cardinality of an equivalence class is constant and only needs to be calculated once. Consequently, the acquisition of data is optimized by reading the cardinality value instead of calculating it again.
- Finally, every equivalence class object contains a second bit vector storing the results of subset calculations. The inclusion or exclusion of the equivalence class in the result of a given calculation is represented by the value of one bit. The result needs to be stored for performance reasons, as it is subsequently used in visualizations, metrics, and further calculations. Again, our implementation uses a 32-bit integer to store that bit vector.

In total, each equivalence class stores three additional integers, using 12 bytes of memory. In other words, the additional information occupies about 12 MB of memory for each million lines of set union code, which is an acceptable overhead. In Figure 52 we show the data metamodel containing the additional, redundant information needed for the optimized set model processing.

## 6.2 Efficient Evaluation of Subset Calculations

The need for fast iterations over the content

The construction of the analysis data model is performed once, and it is acceptable if it runs for a few minutes for large asset groups. However, the subsequent browsing of the model should not involve noticeable delays even for large data structures. Hence, the data acquisitions for various diagrams and the subset calculations need to be performed in a fraction of a second. Such an operation might need to visit each asset content element from every variant, each equivalence class, or both, iterating over millions of elements.



Figure 53 A screenshot of the user interface for specifying subset calculation condition (called a “query” in the tool).

### Specification of subset calculations

In our tool implementation, the data acquisition operations involving only reading a group of variable values comfortably fit within the specified time limit. However, the subset calculations needed further optimization, described below, as they require that the membership and cardinality information is evaluated with the use of a user-specified logical condition. To retrieve any possible combination of the set intersections, our implementation provides the following possibilities to specify the logical condition (see Figure 53):

- Primary calculations are specified using the input set names. For each set, the condition can affirm the set, negate it, or ignore it. For example, for input sets  $S_1$ ,  $S_2$ ,  $S_3$  the condition  $S_1 \cap S_2'$  affirms the set  $S_1$ , negates  $S_2$ , and ignores  $S_3$ .
- Optionally, a criterion on the equivalence class cardinality SHARED BY FROM ... TO ... can also be specified as a part of primary calculation.
- In a given calculation, all atomic formulas are connected with the same logical operator, which can be either AND or OR.
- Secondary calculations can be used to construct more complex logical conditions by using other, already existing primary or secondary calculations. Again, each used calculation can be affirmed, negated, or ignored, and all non-ignored calculations are connected using the AND or OR operator.

The use of negation and the AND and OR operators is sufficient to construct any combination of the input set intersections. The logical conditions using both AND and OR operators need to be constructed as secondary calculations, using the intermediate step of primary calculations to specify the single-operator formulas. The use of only one operator type in a given calculation has two advantages. First, it allows for unambiguous recognition of the operator precedence, as the operators of the previously existing calculations are evaluated first. Second, it enables the performance optimizations described below.



Bit vectors  
supporting  
subset  
calculations

The evaluation of the SHARED BY condition is quick, as the cardinality of a given equivalence relation, stored as its attribute, needs to be simply compared to the numbers specified in the condition. For the evaluation of the remaining calculation parts, note that the relevant information is in each case fully available in a single bit vector: the set membership vector for primary calculations, and the past calculations vector for the secondary calculations. To use that fact, we define further three helper bit vectors based on the specified logical condition. The bit positions in the defined vectors are analogical to the bit positions used in the original membership or calculation vectors. In the calculations on the bit vectors, we use the symbol  $|$  to denote bitwise OR operator, and the symbol  $\&$  to denote bitwise AND operator. The helper vectors need to be calculated just once before iterating over the equivalence classes:

- AFFIRM vector bits are set to one if and only if the respective set (or calculation) is affirmed by the specified condition.
- Likewise, the NEGATE vector bits are set to one if and only if the respective set (or calculation) is negated by the condition.
- The USED vector bits are set to one if and only if the set (or calculation) was not ignored by the query. Hence,  $USED = AFFIRM | NEGATE$ . As the condition needs to reference at least one set, we always have  $USED \neq 0$ .
- Furthermore, we assume that a single condition does not affirm and negate the same set or calculation, i.e. that  $AFFIRM \& NEGATE = 0$ . This assumption is not restrictive, as the calculations not fulfilling it are only those which return all set union elements (for OR operator) or no elements (for AND operator).

Quick  
evaluation  
expressions

Using the above three bit vectors, the logical condition specified by the calculation can be quickly evaluated on the respective bit vector `EC_VECTOR` of each equivalence class in the following way:

- The equivalence class fulfills the condition using the AND operator if and only if  $EC\_VECTOR \& USED = AFFIRM$ .
- The equivalence class fulfills the condition using the OR operator if and only if  $EC\_VECTOR \& USED \neq NEGATE$ .

Correctness

The `EC_VECTOR` bits which are not selected by the `USED` bit mask vector do not influence the calculation result. Hence, to verify the correctness of the above logical statements we only need to consider the `EC_VECTOR` bits for which exactly one from the respective `AFFIRM` and `NEGATE` bits are selected.

- First, let's consider a single `EC_VECTOR` bit: in Table 10 we present the evaluation of all possible single bit value combinations. The evaluated conditions, listed in the two rightmost columns of the table, provide correct results in all four cases. When the bit should be `AFFIRMED`, they return the value of 0 for the 0 bit and 1 for the 1 bit. When the bit should be `NEGATED`, they return the value of 1 for the 0 bit and 0 for the 1 bit.

EC_VECTOR	AFFIRM	NEGATE	USED	EC_VECTOR & USED	EC_VECTOR & USED = AFFIRM	EC_VECTOR & USED != NEGATE
0	0	0	Not evaluated: bit not used			
0	0	1	1	0	1	1
0	1	0	1	0	0	0
0	1	1	Not evaluated: AFFIRM & NEGATE != 0			
1	0	0	Not evaluated: bit not used			
1	0	1	1	1	0	0
1	1	0	1	1	1	1
1	1	1	Not evaluated: AFFIRM & NEGATE != 0			

Table 10

Truth table evaluating the correctness of the quick evaluation expressions

- Now, let's consider a multi-bit EC\_VECTOR and a logical condition using the AND operator. Every considered bit needs to fulfill the condition. Hence, the vectors AFFIRM and EC\_VECTOR & USED need to be identical at every bit, and hence equal, as specified by the quick evaluation expression.
- For a multi-bit EC\_VECTOR and a logical condition using the OR operator, at least one bit needs to fulfill the specified condition. Hence, there must be at least one bit where EC\_VECTOR & USED differs from the NEGATE vector. In other words, it is sufficient if these two vectors are not identical, as specified by the inequality relation in the quick evaluation expression.

Calculation  
properties

Consequently, the fulfillment of a primary or secondary subset calculation condition for a given equivalence class can be evaluated with two simple operations, bitwise AND and equality testing, each of which is performed in a single processor cycle. The resulting calculations are very fast even for million lines of input code, as described in Section 7.1. Moreover, the calculation time is not influenced by the complexity of the used logical expression (e.g. the amount of used variables). In Table 11 we provide an example calculation of the conditions  $S_1 \cap S_2'$  and  $S_1 \cup S_2'$ , performed on the input consisting of three asset variant content sets.

EC_VECTOR ( $S_1; S_2; S_3$ )	AFFIRM	NEGATE	USED	EC_VECTOR & USED	EC_VECTOR & USED = AFFIRM ( $S_1 \cap S_2'$ )	EC_VECTOR & USED != NEGATE ( $S_1 \cup S_2'$ )
000	100	010	110	000	0	1
001	100	010	110	000	0	1
010	100	010	110	010	0	0
011	100	010	110	010	0	0
100	100	010	110	100	1	1
101	100	010	110	100	1	1
110	100	010	110	110	0	1
111	100	010	110	110	0	1

Table 11

Example calculations using the quick evaluation expressions

### 6.3 Ensuring Repeatable Results for Multiple Optimal Solutions

**Problem:** As discussed in Section 5.4.3, some of the analysis problems have many optimal solutions. For example, the transitivity algorithms described in Section 5.4 might find several equally good partitionings of the input similarity graph. Likewise, many longest common subsequences might exist for two input element lists. As only one of the optimal solutions can be stored in the analysis result, the tool implementation faces the problem of selecting these results in a consistent and repeatable way: for the same analysis input and parameters, the analysis should always return the same result.

**Input order dependence in the used algorithms** Frequently, an algorithm selects the first found optimal result and then terminates. However, the selected result might depend on the order of the input data or on other factors which are not directly related to the solved problem. For example, we observed that the most implementations of the diff algorithm might return a slightly different (but still optimal) result when the order of the two compared files is switched: they tend to maximize the length of contiguous line blocks reported for the first input file. Hence, comparing the two sequences "ABC" and "ACB", these implementations return "AB" as the longest common subsequence, while for the reversed order of the sequences the result "AC" is reported. Hence, the use of the diff algorithm in variant similarity analysis enforces a decision on the relative order of each input variant pair. At the same time, as discussed in Section 4.1, the analysis result should not depend on the order of the input variants.

**The canonical variant order** To consistently return an identical analysis result despite input changes which are not relevant from the similarity point of view (e.g. input data order, names of assets or asset parts, etc.), we define a *canonical order* on the input variants. Hence, the asset or asset parts which are provided to an order-sensitive analysis algorithm are consistently ordered before. To eliminate the influence of external factors, the ordering criteria are based solely on the analysis-relevant information. For example, a group of files is sorted based on the amount of relevant content lines, and for equally sized files the lexicographical order of the first differing line pair is used. If no differing line pair can be found, the files are identical from the analysis algorithm point of view, and hence their relative order is irrelevant. Using the canonical order for the above example of two string sequences, the sequence "ABC" is ordered as the first, and the result "AB" is always returned.

Using the canonical order, the analysis always returns the same result for the given input file group, as long as no analysis-relevant information is changed. The result is also identical for an identical file group which is found elsewhere, e.g. at a different location or in a different selection of input asset variants, as only the analysis-relevant file content is used when determining the order. Note that the canonical order is determined separately for each input file group belonging to the input asset variants.

Consequences	<p>The use of a canonical order solves both problems described above. First, in the case of externally implemented analysis algorithms it eliminates the influences of input element order on the provided result, occurring when the algorithms implicitly use the order to select one of the possible optimal solutions. Second, in the case of an own algorithm implementation the canonical order helps in an explicit selection of the single optimal result, as discussed in Section 5.4.3. At the same time, the canonical order does not violate the variant order independence mandated by the construction requirements of our approach: the provided analysis result is identical for any order of the input variants, and the canonical order has no influence on the result interpretation. Actually, the role of the canonical order is a positive one: it constitutes a mechanism which enables the analysis to achieve the order independence despite the use of order-dependent external algorithms and the existence of multiple solution optimums. Furthermore, as the provided analysis result is selected as one of a group of solutions optimal with regard to the specified analysis criteria, there is no other analysis problem solution better than the provided one, although there are many solutions which are equally good and differ in some details.</p>
Further implementation considerations	<p>On the implementation level, the repeatability of the provided analysis result can be further influenced by the nondeterministic behavior of the used data structures. For example, in the Java programming language the iteration order over elements stored in a HashSet or a HashMap might be different for two program executions run on the same data. Consequently, the resulting differences in element iteration order might override the canonical order, influence the order-dependent algorithms, and cause them to choose a different optimal solution. To prevent that influence, we avoid data structures with nondeterministic iteration order and use their deterministic variants instead (in the case of Java, these are the LinkedHashSet and the LinkedHashMap).</p>

## 6.4 Summary

In this Chapter we discussed three selected implementation mechanisms for our analysis approach: the use of data redundancy for improving the data acquisition performance, the use of bitwise operations for a performant computation of subset calculation results, and the definition of a canonical order on the analysis elements which allows for a consistent and repeatable selection of a single analysis result from multiple optimal solutions. With these descriptions, we argue that the set model based similarity analysis can deliver its results in a performant, scalable and consistent way. Furthermore, in Chapter 7 we support that statement with performance measurements of our analysis implementation, performed for data sets of various sizes.



## 7 Evaluation

The set model based similarity analysis approach, defined in the previous Chapters, is intended for the analysis of a group of cloned and modified software asset variants. Depending on the application scenario, the analysis results are subsequently used to plan a reuse migration of the input variants, or to support their further parallel maintenance. In contrast to the state of the art approaches, the hierarchical set model provides a scalable abstraction of the analyzed similarity information for both large software assets and a large amount of variants. Hence, we hypothesize that the use of the approach reduces the analysis effort and allows for better understanding of the similarity information. Furthermore, these analysis-time benefits should allow for a reduction of the overall migration or parallel maintenance effort, and lead to a higher degree of reuse achieved in the migration. Finally, we postulate that the results provided by the approach are correct and that the approach can be successfully used by the software development practitioners. A discussion of the above hypotheses was presented in Section 3.4.

To substantiate the stated hypotheses, we use a range of evaluation means (see Figure 54). In an analytical evaluation (Section 7.1), performed on industrial, open source, and artificial software systems, we check the result correctness and collect measures concerning the input data transitivity and analysis performance. With a controlled experiment (Section 7.2), we investigate the differences in human analysis effort and in human understanding correctness between two forms of presented similarity information: pairwise and set model based. In an industrial case study (Section 7.3), we mainly look at the fulfillment of the practical hypotheses concerning the effort reduction and achievable reuse degree. Finally, in Section 7.4 we report on five industrial application experiences, where the approach was used by the author, other researchers, and software practitioners on software system groups from various domains.

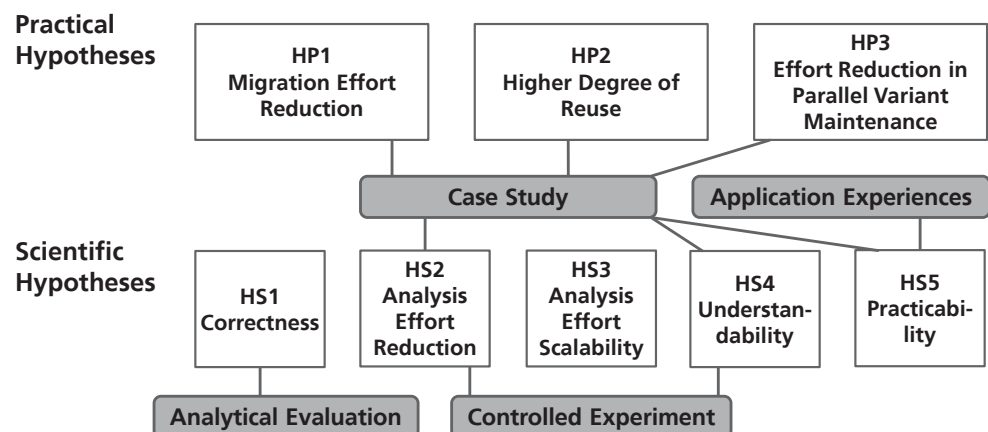


Figure 54

The overview of the practical and scientific hypotheses and the used evaluation means.

While the performed evaluations investigate the approach contributions from a variety of viewpoints, they are all performed based on a realistically available data input. Hence, the provided positive evaluation results are not a decisive proof of the approach benefits, but rather constitute data points which indicate that the hypotheses were confirmed in the example encountered context. The positive evaluation results increase the confidence that the defined approach indeed provides the stated benefits. Nevertheless, we consider a further approach evaluation to be a necessary part of the future work.

## 7.1 Analytical Evaluation

The analyzed system types      In this Section we report and discuss the results of a group of measurements, targeting the transitivity of the similarity data provided by the diff algorithm, the correctness of the overall approach result (hypothesis HS1), and the tool performance. We collected the measurements on three types of input software variant groups, having varying sizes and varying amount of member variants:

- **Industrial software systems** created with the use of cloning. The selection of the system groups is limited to these for which we had a sufficient source code access during the practical application projects at Fraunhofer IESE. For anonymity reasons, we only provide the resulting measurement data, but do not disclose the company names, product names or other sensitive details.
- **Open source systems** where multiple cloned variants exist and are actively maintained. As the BSD Unix system family is the most prominent case of a large-scale cloning of long-living systems, we perform the measurements on selected groups of the BSD variants.
- **Artificial software systems** generated by ForkSim, a cloned system generation framework [Svajlenko 2013]. The sample data sets, provided by the main ForkSim author Jeffrey Svajlenko, were created as source clones of the JHotDraw 5.4 graphics framework and extended by random injections of files and functions from the Java 7 SDK code. The added code was modified in some variants and injected at same or different locations. While the properties of artificial data sets might differ from real system clones, their advantage is that the origin and location of each code modification, and hence the information on all code similarities, is exactly known. Using a group of artificially created systems, the similarity analysis results can be evaluated for correctness against the known similarity – which is not possible for real system clones.

Technical configuration      The measurements were performed on a computer equipped with a 2.53 GHz Intel Core2 Duo processor, 2 GB RAM and 32-bit Windows 7 operating system. Our tool implementation, based on the Fraunhofer SAVE framework [Duszynski 2009], is implemented in the Java 6 programming language, uses Eclipse Modeling Framework for data model management, and runs as a set of Eclipse plugins under Eclipse 3.7. All analysis procedures, except for the multiple alignments mapping algorithm, use a single processing thread.



Analyzed system groups In Table 12, we present the analyzed variant system groups in more detail. We analyzed three industrial system groups of various sizes: the group Ind\_Small represents one of the smallest real-world analysis problems, while the other two groups are typical for the analysis problem dimensions occurring in the industry. Subsequently, we analyzed four BSD Unix variants released in a similar time. Before that time, OpenBSD did not yet exist, while the development of the “standard” BSD Unix ceased after the 4.4 Lite 2 release. Hence, only during the selected time as many as four large BSD variants existed simultaneously. In addition, we analyzed the BSD\_15 system group – although these systems are both variants as well as versions of each other, we analyzed that large group to test the approach scalability. Finally, we analyzed five artificial system groups, created using different generation settings (see Table 12 for details). The reported code size does not include empty (whitespace-only) lines, while all remaining file content (import declarations, comments, etc.) is included. The file count considers only code files: configuration files and the remaining textual files were ignored in the analysis.

System Group Id	Description	Number of Systems	LOC (Min-Max)	Files (Min-Max)
Ind_Small	Industrial embedded software: 2 variants of a machine controller driver for various hardware, C	2	31,685 to 35,340	38
Ind_Medium	Industrial embedded software: automotive controller with variants for various customers, C/C++	12	31,402 to 179,299	139 to 534
Ind_Large	Industrial embedded software: 4 variants of a machine control system, various functionality, C/C++	4	1,122,110 to 1,526,155	3,540 to 4,555
BSD_4	The full <i>usr/src</i> code of four BSD Unix variants released in 1995 and 1996: 4.4 BSD Lite 2, FreeBSD 2.1.5, NetBSD 1.2, OpenBSD 2.0. C/C++	4	2,275,711 to 3,604,446	6,921 to 11,193
BSD_15	The full <i>usr/src</i> code of fifteen BSD Unix releases, developed between 1992 and 1999, belonging to four BSD variants: BSD (4.4, 4.4 Lite 2), FreeBSD (2.0.5, 2.1.5, 2.2.5, 3.0), NetBSD (1.0, 1.1, 1.2, 1.3, 1.4.1), OpenBSD (2.0, 2.1, 2.3, 2.4). C/C++	15	1,885,612 to 4,808,638	6,302 to 15,730
Gen_4	4 generated variants based on JHotDraw (original system size: 285 files and 34,513 lines). The new code was injected into up to 3 variants and modified with 50% probability. Java	4	55,371 to 111,713	361 to 636
Gen_5A	5 generated JHotDraw variants; new code injected into up to 4 variants, 50% modification probability. Java	5	45,313 to 82,681	311 to 393
Gen_5B	5 generated JHotDraw variants; new code injected into up to 4 variants, 50% modification probability, increased number of injected files and folders. Java	5	64,051 to 110,322	413 to 671
Gen_5C	5 generated JHotDraw variants; new code injected into up to 5 variants, 50% modification probability, increased number of injected files and folders. Java	5	98,145 to 114,258	489 to 574
Gen_6	6 generated JHotDraw variants; new code injected into up to 5 variants, 50% modification probability, increased number of injected functions. Java	6	57,352 to 79,273	380 to 444

Table 12 The analyzed variant system groups

System Group Id	Set Union Size	Core Code Size	Unique Code in the Union	Union Similarity Bar	Number of Non-Empty Intersections	Relative Set Union Size
Ind_Small	38,203	28,822	9,381		3	1.754
Ind_Medium	205,134	24,112	20,388		273	6.527
Ind_Large	2,215,140	663,618	534,853		15	2.382
BSD_4	5,687,835	932,767	2,846,787		15	1.954
BSD_15	16,240,391	292,789	6,744,827		3,251	2.921
Gen_4	136,421	34,616	31,170		15	2.524
Gen_5A	101,223	34,371	38,347		31	2.972
Gen_5B	151,175	34,936	36,036		31	3.172
Gen_5C	155,410	72,387	35,148		31	3.414
Gen_6	104,167	34,272	21,022		63	3.905

Table 13 A short description of the analyzed systems' similarity

**Analysis settings** Table 13 presents a shortened description of the analyzed systems' similarity. The system files were first matched to each other using the mapping algorithm specified in Table 14, and subsequently the file content was compared using diff. For the content comparison, the leading and trailing white spaces were removed from content text lines to nullify their influence of the reported similarity. Apart from that, no other text filtering or normalization operations were used.

**High industrial system similarity** In the created results, note the high similarity of all three industrial systems groups. The proportion of unique code is low in each industrial group. In the Ind\_Small group, the core code constitutes at least 80% of each variant's code. In the Ind\_Medium group, the set union size is 6.5 times smaller than the sum of variant sizes – despite a relatively small core, there exist sizable code fragments shared by many variants. Finally, the core of the Ind\_Large group constitutes at least 43% of each variant's code. The high similarity of the presented system groups is in our experience typical for the industrial cloned system variants, which motivates the design decisions of our approach discussed in the previous Chapters.

### 7.1.1 Performance and Scalability

**Measurement interpretation** Table 14 presents the performance and scalability measurements of the approach implementation. The run times and memory use were measured by system function calls embedded in the tool code. Due to the use of the Eclipse Modeling Framework data model, all measurements do not concern the pure analysis algorithms measured in isolation, but rather by necessity include the overhead of creating and iterating over the EMF object structures. Hence, the measured run times are longer and the memory use is larger than the values necessary for the analysis alone (for example, see the mapping run time of the BSD\_15 systems and the memory use of the Ind\_Small systems). Hence, the measurements should be interpreted as the practical time and memory requirements of the analysis tool realized in the given technology.

System Group Id	Mapping Algorithm	Mapping Time	Total Analysis Time	Total Memory Use	Subset Calc. Time (Min-Max)
Ind_Small	Location Identity	0.046 s	1.48 s	223 MB	1-4 ms
Ind_Medium	Alignments	1,204 s	1,270 s	731 MB	8-12 ms
Ind_Large	Location Identity	5 s	263 s	1,152 MB	94-102 ms
BSD_4*	Alignments	13,069 s	13,611 s	65,101 MB	204-216 ms
BSD_15*	Location Identity	189 s	3,374 s	43,822 MB	486-578 ms
Gen_4	Alignments	71 s	81 s	413 MB	7-11 ms
Gen_5A	Alignments	63 s	71 s	320 MB	4-9 ms
Gen_5B	Alignments	132 s	145 s	478 MB	6-12 ms
Gen_5C	Alignments	138 s	150 s	425 MB	6-12 ms
Gen_6	Alignments	119 s	129 s	382 MB	4-9 ms

Table 14 The performance and scalability measurements (\* used different hardware, see below)

Selection of the mapping algorithm

In the analysis, we used either the location identity or the multiple alignments mapping algorithms. For industrial system groups, we used the multiple alignments algorithm only if it resulted in a significantly higher found similarity, which was the case for Ind\_Medium systems. For the artificial system groups, we always used the multiple alignments algorithm as we also needed it for the evaluation of approach correctness, described later in this Section. Finally, for BSD\_4 systems the multiple alignments algorithm resulted in a higher found similarity, but we were unable to use that algorithm for BSD\_15 systems due to its memory requirements.

Different hardware for the BSD systems

The 32-bit Java virtual machine is only able to allocate about 1.5 GB of memory, which is not sufficient for analyzing the BSD\_4 and BSD\_15 system groups. Hence, we analyzed these groups using a different computer equipped with 64-bit Windows 2008 operating system, 24 processor cores running at 2.80 GHz, and 64 GB of memory. Note that as only the multiple alignments mapping algorithm is parallelized, the high amount of cores does not affect the comparability of the remaining time measurements. In Table 14, we indicated the use of the different hardware with a star symbol (\*) placed next to the BSD system group names.

Good implementation performance

The measured results show that using a relatively modest contemporary hardware configuration, an analysis of all except the largest system groups is possible. Except for the alignment mapping or the large BSD system groups, the similarity analysis and the construction of the hierarchical set model is performed within few minutes. In case the multiple alignments algorithm is used, it dominates the time and memory requirements of the implemented tool. In the subsequent result interpretation phase, for all system groups no user-noticeable delays were observed during result browsing and diagram construction. This is exemplified by the measured subset calculation times. For each system group, we performed 10 measurements, formulating different logical conditions and using both primary and secondary calculations. The calculation time did not exceed 12 ms for medium-sized system groups, and the longest time of 578 ms was measured for the BSD\_15 system group having 16 MLOC of set union code.

### 7.1.2 Input and Result Transitivity

**Measurement procedure** In Table 15 we report the transitivity-related measurements performed on the input similarity data provided by the diff algorithm. First, we counted the provided similarity graphs, with nodes representing text lines and edges representing their binary similarity relations identified by diff. Second, we distinguished between the transitive and non-transitive graphs: as each analyzed graph is contiguous, the transitive graphs are complete and can be directly reported as equivalence classes. Third, we processed the non-transitive graphs (see Section 5.4.3 for details) to split them into transitive subgraphs. Consequently, the final amount of graphs reported in the result (i.e. the amount of the equivalence classes, equal to the size of the set union) is higher than the amount of input graphs. Finally, we counted the edges belonging to each graph category. For the measured systems, at least 99.56% of input graphs, containing at least 97.52% of input edges, are transitive. The final analysis result contained at least 99.25% of input edges. As discussed in Section 5.4.3, no artificial edges were added to the result.

System Group Id	Graphs				Edges			
	All Input	Input Transitive	Input Non-Transitive	Result	All Input	In Transitive Graphs	In Non-Transitive Graphs	Removed from the Result
Ind_Small	38,203	38,203 100%	0	38,203 +0%	28,822	28,822 100%	0	0
Ind_Medium	200,319	199,750 99.72%	569 0.28%	205,134 +2.40%	5,542,092	5,472,550 98.75%	69,542 1.25%	23,561 0.43%
Ind_Large	2,184,673	2,175,073 99.56%	9,600 0.44%	2,215,140 +1.39%	5,144,634	5,063,006 98.41%	81,628 1.59%	38,823 0.75%
BSD_4	5,658,176	5,639,487 99.67%	18,689 0.33%	5,687,835 +0.52%	8,984,972	8,875,337 98.78%	109,635 1.22%	39,415 0.44%
BSD_15	16,103,634	16,049,242 99.66%	54,392 0.34%	16,240,391 +0.85%	122,905,536	119,856,941 97.52%	3,048,595 2.48%	654,267 0.53%
Gen_4	136,333	136,268 99.95%	65 0.05%	136,421 +0.06%	345,351	345,007 99.90%	344 0.10%	142 0.04%
Gen_5A	101,005	100,858 99.85%	147 0.15%	101,223 +0.22%	455,040	453,669 99.70%	1,371 0.30%	478 0.11%
Gen_5B	151,003	150,879 99.92%	124 0.08%	151,175 +0.11%	686,227	685,129 99.84%	1,098 0.16%	381 0.06%
Gen_5C	155,035	154,799 99.85%	236 0.15%	155,410 +0.24%	863,365	861,286 99.76%	2,079 0.24%	745 0.09%
Gen_6	103,678	103,372 99.70%	306 0.30%	104,167 +0.47%	803,648	799,728 99.51%	3,920 0.49%	1,183 0.15%

Result

Table 15 The transitivity measurements for all variant groups

Consequences of the high measured transitivity	Considering the high proportion of transitive input graphs, the set similarity model is in our opinion suitable for storing and analyzing the diff-based similarity information. Furthermore, the high proportion of input edges which are included in the final analysis result, and the lack of artificially added edges, make the analysis result trustable and dependable for the defined application scenarios. Naturally, the input system groups provide just example data points and do not allow for a statistically significant analysis.
A search for factors influencing transitivity	<p>At this point, it is interesting to ask whether, and under which conditions, the similarity provided by diff can be strongly non-transitive and hence unsuitable for the analysis goals. Based on the data in Table 15, there seems to be no single measured factor correlating with lower transitivity:</p> <ul style="list-style-type: none"> <li>• The proportion of non-transitive graphs is similar in all real-world system groups (0.28% to 0.44%), regardless of their size and the amount of member systems. The artificial systems contain less non-transitive graphs (0.05% to 0.30%), which might be a side effect of the generation process.</li> <li>• Naturally, the graphs created for a higher amount of systems contain a larger number of edges (e.g. on average there are 5.9 edges for BSD_4 graphs, but 122.2 edges for Ind_Medium graphs). However, the data suggests that there is no strong correlation between the amount of non-transitive graph edges and the amount of removed edges. In Table 15, between 21% and 48% of the input non-transitive graph edges needed to be removed. The proportion of removed edges varies strongly for groups containing a similar number of systems. Interestingly, the proportion of removed edges is the highest for the Ind_Large group, containing just 4 systems, and the lowest for BSD_15 group containing 15 systems.</li> <li>• Furthermore, there seems to be no clear correlation between the amount of analyzed systems and the transitivity of the result. Although lower transitivity could be expected for a higher number of variants, the lowest result transitivity value was measured for the Ind_Large group.</li> <li>• Consequently, we initially conclude that the individual content topology of the analyzed system groups, defining the layout of the non-transitive graphs, is a factor having much more influence on the result transitivity than the amount of grouped variants or the proportion of non-transitive edges. Hence, the result transitivity and the suitability of the system group for a set model based similarity analysis cannot be estimated in a simple way before the actual analysis.</li> </ul>
Reducing the topology influence	To reduce the influence of the individual content topology on the measurements of result transitivity, we performed a further analysis on a single system group, the Ind_Medium systems. We selected that group as it contains a high number of real-world variants. From the initial group of 12 systems, in each iteration we removed one system, performed a new analysis, and measured the result transitivity, until two systems remained. For consistency, we always preserved the initial analysis settings and we reused the mapping previously constructed for 12 systems.



The use of two variant orders

When removing the single systems in the analysis iterations, we effectively needed to define a removal order on the variants. However, as no variant order can be distinguished (see Section 4.1), and there exist  $12!$  possible orders for Ind\_Medium systems, we decided to at least run the analysis with two removal orders: based on the variant names (the alphabetically last variant is removed) and on the variant code sizes (the largest variant is removed). Table 16 contains the analysis results and the transitivity measurements, which are visualized in Figure 55.

Systems	Name Ordered Ind_Medium Variants					Size Ordered Ind_Medium Variants				
	Graphs		Edges			Graphs		Edges		
	All	Trans	All	Trans	Result	All	Trans	All	Trans	Result
2	128,609	128,609 100%	79,376	79,376 100%	79,376 100%	98,020	98,020 100%	24,286	24,286 100%	24,286 100%
3	193,521	193,135 99.80%	272,494	271,133 99.50%	271,752 99.73%	159,870	159,838 99.98%	78,198	78,104 99.88%	78,144 99.93%
4	194,962	194,468 99.75%	579,671	575,484 99.28%	577,489 99.62%	189,595	189,563 99.98%	219,383	219,185 99.91%	219,301 99.96%
5	195,540	194,984 99.72%	1,005,035	996,097 99.11%	1,000,856 99.58%	189,621	189,589 99.98%	477,613	477,273 99.93%	477,503 99.98%
6	196,886	196,340 99.72%	1,544,935	1,529,727 99.02%	1,538,591 99.59%	190,094	190,043 99.97%	851,827	851,103 99.92%	851,620 99.98%
7	196,911	196,365 99.72%	2,200,539	2,178,030 98.98%	2,192,513 99.64%	191,785	191,358 99.78%	1,335,553	1,325,989 99.28%	1,332,886 99.80%
8	197,595	197,043 99.72%	2,972,069	2,940,218 98.93%	2,961,241 99.64%	192,430	192,046 99.80%	1,936,884	1,921,677 99.22%	1,932,897 99.79%
9	199,738	199,181 99.72%	3,853,182	3,809,426 98.86%	3,836,304 99.56%	193,711	193,307 99.79%	2,655,336	2,629,896 99.04%	2,648,592 99.75%
10	199,996	199,431 99.72%	4,852,784	4,796,342 98.84%	4,833,009 99.59%	194,325	193,852 99.76%	3,492,048	3,453,078 98.88%	3,481,628 99.70%
11	200,301	199,732 99.72%	5,212,881	5,150,654 98.81%	5,190,836 99.58%	197,047	196,514 99.73%	4,432,191	4,376,255 98.74%	4,414,509 99.60%
12	200,319	199,750 99.72%	5,542,092	5,472,550 98.75%	5,518,531 99.58%	200,319	199,750 99.72%	5,542,092	5,472,550 98.75%	5,518,531 99.58%

Table 16 The transitivity measurements for subgroups of Ind\_Medium variants

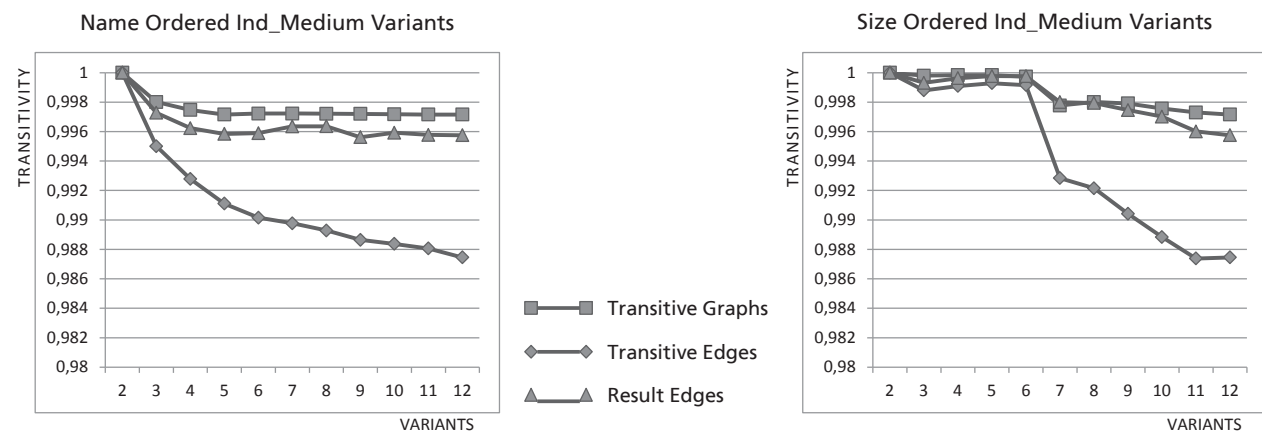


Figure 55 The transitivity measurements for subgroups of Ind\_Medium variants.

Observations: two counteracting trends	<p>For the two variant orders, two different results are created: while for the name order the transitivity gently decreases with increasing amount of variants, for the size order a sudden change occurs between the six-variant and the seven-variant groups. In our opinion these two different results, created by selecting different variants out of the same group, again indicate that the individual topology of the variant content is the strongest factor influencing the transitivity of both the input and the analysis result. Nevertheless, based on the measurement data we observe that:</p> <ul style="list-style-type: none"> <li>• The proportion of non-transitive input graphs increases, but only slowly, with the increasing size of the selected variant group.</li> <li>• The proportion of edges belonging to the non-transitive input graphs increases moderately with the increasing size of the variant group. Apparently, within the same system group selection the smaller input graphs tend to be more frequently transitive than the larger ones.</li> <li>• The proportion of the input edges removed in the construction of the analysis result increases, but only slowly, with the increasing size of the variant group. Given the much faster growing proportion of the non-transitive input edges, this means that with the increasing graph size the transitivity algorithms are able to preserve an increasing proportion of the input edges for the final result. This property, effectively counteracting the growing proportion of non-transitive input edges, is intuitively correct: for the smallest non-transitive graph, having 3 nodes and 2 edges, 50% of the edges need to be removed, while for the larger graphs having many tens of edges a removal of just a few edges (and hence their lower proportion) might be sufficient when the graph is dense.</li> </ul>
Conclusion: no transitivity estimators	<p>As we have no access to a system group having more than 12 real variants, we cannot measure how far the two observed counteracting trends, i.e. the increasing proportion of non-transitive graphs and edges in the input, and the decreasing proportion of the initially non-transitive edges which are removed from the result, balance each other for a higher number of variants. Hence, we cannot say whether the diff-created input can be in a general case less transitive for a higher number of variants – answering this question, and providing more generalizable measurements, remains a future work. Nevertheless, the individual content topology of a variant group seems to dominate the other factors influencing the result transitivity: adding a new variant can drastically reduce the transitivity, as well as improve it (e.g. consider the result transitivity of groups having three, four and five variants in the size-ordered analysis). Hence, the proportion of edges removed from the result cannot be estimated upfront – instead, it needs to be measured individually for each analyzed variant group.</p>

### 7.1.3 Approach Instantiation Correctness: Precision and Recall

In the last part of analytical evaluation, we measure the precision and recall of our approach according to the method defined in Section 4.5. As the evaluation can only be performed when the correct analysis result is exactly known, we conducted the measurements on the generated system groups.



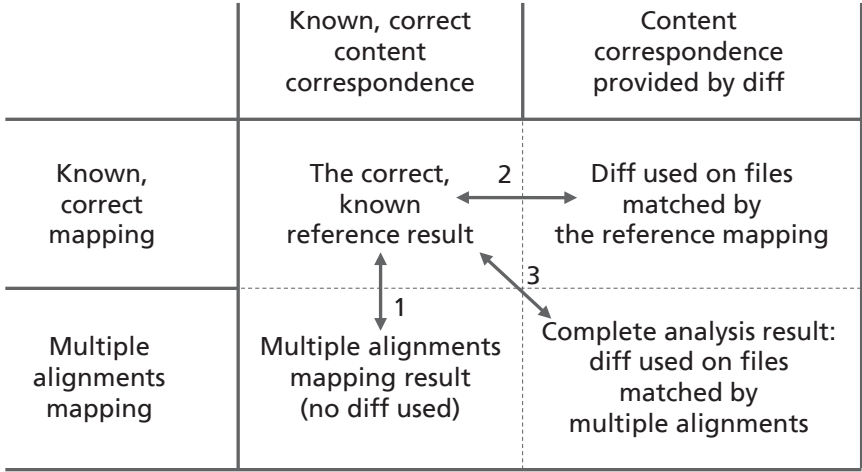


Figure 56 The three measurement series for the evaluation of approach precision and recall.

Three measurement series

We performed three measurement series, illustrated in Figure 56. The first series evaluated the result correctness of the multiple alignments algorithm, considering only the corresponding files across the variants. The second series used the reference, fully correct mapping to match the files and then evaluated only the correctness of matching the corresponding content lines using diff. Hence, the first two series tested in isolation the results of respectively the structure element and the content element equivalence functions. Finally, the third series compared the matched content lines of the reference result to the analysis result constructed using alignments mapping and diff – hence evaluating the combined correctness of both functions. The measured precision and recall values for all series are provided in Table 17.

System Group Id	Series 1: Alignments Mapping			Series 2: Reference Mapping Diff			Series 3: Alignments Mapping Diff		
	True Positives	False Positives [Precision]	False Negatives [Recall]	True Positives	False Positives [Precision]	False Negatives [Recall]	True Positives	False Positives [Precision]	False Negatives [Recall]
Gen_4	1,462	2 99.86%	2 99.86%	207,768	178 99.91%	122 99.94%	207,598	324 99.84%	292 99.86%
Gen_5A	1,297	3 99.77%	2 99.85%	199,230	181 99.91%	96 99.95%	198,964	604 99.70%	362 99.82%
Gen_5B	2,213	20 99.10%	15 99.33%	327,975	298 99.91%	158 99.95%	326,588	1,703 99.48%	1,545 99.53%
Gen_5C	1959	9 99.54%	7 99.64%	374,605	578 99.85%	224 99.94%	373,181	2,032 99.46%	1,648 99.56%
Gen_6	2050	2 99.90%	1 99.95%	301,794	784 99.74%	483 99.84%	301,773	823 99.73%	504 99.83%

Table 17 Approach precision and recall, measured on the five generated system variant groups

High values of precision and recall	The measured diff result has a very high precision ( $\geq 0.9974$ ) and recall ( $\geq 0.9984$ ), while the multiple alignments result has lower measured values (precision $\geq 0.9910$ , recall $\geq 0.9933$ ). Consequently, the combined result constructed by the both equivalence functions has values slightly lower than these of diff, but higher than these of multiple alignments mapping (precision $\geq 0.9946$ , recall $\geq 0.9953$ ). All measured precision and recall values are larger than 0.99 – hence, the measured data supports the hypothesis HS1 (Correctness).
Threats to validity	However, there are two main threats to the validity of this conclusion. First, the measured input systems are artificially generated, and might differ from real-world cloned variants – the use of generated systems was necessary though, as only for them the reference result is known. Second, the measurement was performed on just five systems, having the same origin, which limits the conclusion generalizability. While the construction of difficult analysis examples, resulting in precision and recall values well under 0.99, is possible, we do not know how often such difficult cases occur in the practice. Hence, as discussed before in Section 3.4, we assume that the stated hypotheses are true in the most cases, but might be false in particularly unfavorable conditions.

## 7.2 Controlled Experiment

Hypotheses addressing human-based effects	The scientific hypotheses HS2 (Analysis Effort Reduction), HS3 (Analysis Effort Scalability) and HS4 (Understandability) address not the technical side of the analysis approach, but rather the human-based effects resulting from its use, i.e. the human effort and the similarity understanding. Hence, they need to be evaluated based on the human-related measures collected during the approach execution – either in a controlled experiment, or in a case study. In the current and the next Section, we report the respective results we collected by using both these evaluation means. For brevity, we describe an aggregated view on experiment setup and results here, while the further details of the experiment, including the materials used by the participants, are provided in Appendix A.
---	--

### 7.2.1 Experiment Goal and Hypotheses

Isolation of the influencing factors	A definition of a controlled experiment, evaluating the benefits of the presented approach, faces the difficulty of a proper isolation of the factors influencing the experiment outcome. A precise identification of the causes of an analysis effort reduction, achieved by using the complete approach, is difficult as several analysis mechanisms, differentiating the approach from the state of the art, are introduced (e.g. the set model, the abstractions, the visualizations). Furthermore, some types of information are automatically collected and quickly available in the set model based approach, while in some other approaches they need to be manually gathered. Additionally, the use of different tools, instantiating the compared approaches, adds a further influencing factor in the form of potentially different user interface concepts.
--------------------------------------	---

Finally, a realistic analysis of a group of software systems necessarily involves tasks of a very different granularity: from the analysis of single code lines, up to the similarity assessment of whole systems having many thousands of code lines. Hence, due to the differences in the used abstraction mechanisms, the measured analysis effort reduction could vary strongly depending on the task granularity. Summing up, an experimental evaluation of the complete analysis approach would not provide the means to determine whether, and to which degree, the measured difference in the analysis effort was caused by the use of the set model, the abstractions, the visualizations, or the different user interface, but would only provide results for a combination of those.

Experiment  
focus:  
the set model

Consequently, we decided to limit the scope of the experiment and evaluate the core contribution of our approach, the set similarity model, in isolation from the other factors such as the hierarchy abstractions and visualizations. In this context, it is interesting to ask how far the similarity abstraction in the form of a set model is easier for humans to analyze and understand as compared to the most frequently used state of the art similarity abstraction, provided by the pairwise comparison. The experiment goal [Briand 1996], explained in more detail in the next subsections, is therefore to:

<b>Analyze for the purpose of with respect to from the viewpoint of in the context of</b>	the pairwise and set-based similarity models comparison analysis efficiency, correctness and cognitive load a software developer Software Product Lines course, with students analyzing file variants for code similarity.
---	---

Experimental  
hypotheses

To address the defined goal, we derived new experimental hypotheses, targeting only the set model, by restating the original hypotheses HS2 and HS4. We decided to not evaluate the hypothesis HS3 for effort and participant availability reasons – such evaluation would require a much more extensive experiment, performing a series of measurements for different numbers of analyzed variants. The experimental hypotheses are therefore:

- **HSet1 Efficiency.** The use of the set similarity model reduces the effort for analyzing similarity information as compared to the use of pairwise comparison model (30% time reduction for up to 4 variants, 50% time reduction for 5 and more variants).
- **HSet2 Correctness.** The use of the set similarity model allows for understanding the similarity information with a higher correctness compared to the use of pairwise comparison model (50% less false statements).
- **HSet3 Cognitive Load.** The use of the set similarity model allows for analyzing the similarity information with a lower cognitive load compared to the use of pairwise comparison model (cognitive load lower by at least one category on the SMEQ scale).

The rationale and measurement of HSet3

The hypotheses HSet1 and HSet2 directly restate the hypotheses HS2 and HS4, with the scope reduced to the set model only. The intention of the last hypothesis HSet3 is to provide more evaluation support for the both original hypotheses HS2 and HS4. To evaluate HSet3, we use the Subjective Mental Effort Question (SMEQ) scale [Zijlstra 1993], validated in usability research, which is frequently used for cognitive load measurement [Albers 2012]. The SMEQ presents a continuous scale, labeled in nine locations with categories ranging from “absolutely no effort” to “extreme effort” (see the experiment material in Appendix A). The respondents indicate their subjectively felt cognitive load, experienced during the similarity analysis, by placing a mark anywhere on the scale. The mark location is subsequently converted to an integer value between 0 and 150. The SMEQ measurements are provided on an interval scale, as the category locations were psychometrically calibrated. This allows a convenient response analysis, as the calculation of averages and distances is meaningful for interval scale data.

We added the hypothesis HSet3 to the experiment, as its positive evaluation has two effects. First, the cognitive load measured by SMEQ for a group of participants solving the same tasks was reported to highly correlate with task time ( $r = -0.82$ ,  $p < 0.01$ ) and task errors ( $r = -0.72$ ,  $p < 0.01$ ) [Sauro 2009]. Hence, the measurement of cognitive load reduction provides another indication supporting the original hypotheses HS2 and HS4. And second, we consider the cognitive load reduction to be a further benefit provided by the set model. In Table 18 we list the metrics which we collect in the experiment to evaluate the three stated hypotheses.

Metric	Metric Name	Associated Hypothesis	Null Hypothesis
M1	Analysis time [min]	HSet1: $M1(\text{Set}) < M1(\text{Pair})$	HSet1 <sub>0</sub> : $M1(\text{Set}) \geq M1(\text{Pair})$
M2	Number of correctly solved tasks		
M3	Number of all tasks (constant)		
M4	Incorrect answer ratio: $M4 = \frac{M3 - M2}{M3}$	HSet2: $M4(\text{Set}) < M4(\text{Pair})$	HSet2 <sub>0</sub> : $M4(\text{Set}) \geq M4(\text{Pair})$
M5	Cognitive load (measured using SMEQ)	HSet3: $M5(\text{Set}) < M5(\text{Pair})$	HSet3 <sub>0</sub> : $M5(\text{Set}) \geq M5(\text{Pair})$

Table 18 The main metrics collected during the controlled experiment and the associated hypotheses

### 7.2.2 Experiment Design and Operationalization

Experiment design

To compare the effects of using the two evaluated similarity models, we selected the between-subjects experiment design. Hence, the participants were assigned to one of the two groups: the treatment group, using only the set similarity model, or the control group, using only the pairwise similarity model. Consequently, in the experiment we compared the performance of the both groups.

Student participants

The experiment was performed in January 2013. The experiment participants were 23 students attending the Software Product Lines course at the Technical University of Kaiserslautern: 19 master-level and 4 bachelor-level students, studying computer science or software engineering. None of the students had a prior contact with the Variant Analysis approach or tool. The experiment was performed during the regular lecture hours, and the use of lecture time for the experiment was announced in advance. The participation in the experiment was voluntary, and almost all course attendants appeared for the experiment.

Assigning participants to the groups	<p>The 23 participants were assigned to the two experimental groups in a random way. Before the experiment, the participants gathered in the lecture room, and took the seats which were lined in rows. We assigned every of 11 participants sitting on an even-numbered seat to the treatment group, and the remaining 12 participants to the control group. The intention of this randomization method was to split and evenly distribute the groups of students having similar background, which would be most probably sitting near each other in the lecture room. As presented in the next subsection, this resulted in balanced experimental groups.</p>
Experimental tasks	<p>The experimental task, identical for both groups, was to answer 16 questions concerning the code similarity in the files belonging to five software system variants. The questions were printed in the experiment documents, distributed to each participant, and needed only a short answer, such as stating the names of variants fulfilling a given condition. As in the case of the analytical evaluation, the system variants analyzed by the participants were generated by the ForkSim tool. The same five variants, based on the Java code of the JHotDraw tool, were provided to both groups. The use of the generated variants guaranteed that the correct answers to the experimental questions, used to evaluate the participant results, were determined without the need to use any of the analysis methods investigated in the experiment.</p>
Similarity viewed with a tool	<p>The participants from both groups viewed the provided code files using a variant of the Variant Analysis tool specifically adapted for the experiment. In the tool, all visualizations and analysis mechanisms except for the system hierarchy navigation and the code view were disabled (see the Appendix A for details). Hence, the students were only able to locate the files in the system structure diagram, identical for both groups, and to view the code of file variants in the code editor. In the editor, the background of the displayed code lines was colored according to the similarity information provided by one of the similarity models: the pairwise model for the control group and the set model for the treatment group. The textual information provided for each line by the respective category icons was also model-dependent. Except for these differences, all other user interface mechanisms were identical for both groups. The participants were asked to not use any other tools (e.g. operating system tools) during the experiment, but they could take any necessary notes on the provided paper sheets. The tool use was periodically controlled by experiment supervisors walking around the laboratory – no deviations occurred.</p> <p>The actual automatic analysis of the code similarity was not part of the experiment – the students only viewed the similarity information, provided by a previously performed analysis, and used that information to answer the given questions. As the same system variants were analyzed by both groups, the used similarity information was technically the same – we verified that all the pairwise similarity relations were included in the constructed set model.</p>

Independent and dependent variables

As discussed above, the only independent variable varied between the experimental groups was the similarity model used for accessing the similarity information: the pairwise model or the set model. Every other difference between the groups was removed as far as possible: the groups solved identical analysis tasks, used identical tool to access the similarity information (except for the underlying model), were given identical experiment documents, and worked in parallel in two equivalent computer laboratories. The dependent variables investigated in the experiment are the analysis effort, the answer correctness, and the cognitive load, as specified in Table 18 on the earlier page. Apart from collecting the metrics related to the dependent variables, we also asked each participant a range of identical briefing and debriefing questions to characterize the participant background and check the impressions they had immediately after the experiment completion.

The experiment process

Table 19 presents the experimental process, followed by both groups, and lists the differences in particular process steps and artifacts caused by the use of the respective similarity model. Each participant received an identical main experiment document, containing the introductory information, the briefing questionnaire, the experimental tasks, and the debriefing questionnaire. Moreover, they received a printed tool tutorial, which was also presented to them as a slide show. The tutorial was identical for both groups, except for the part concerning the similarity model (see the Appendix A for the complete experiment document and the tutorials). After the tutorial, the participants familiarized themselves with the tool and answered two sample warm-up questions, which had an identical form as the actual experiment questions but concerned the different example software variants, loaded into the tool for the tutorial purposes.

Step	Step Description	Difference Between the Groups
1.	The experiment procedure is presented.	None
2.	The participants are split into two groups.	None
3.	The participants receive and read the introductory material.	None
4.	The participants fill out the briefing questionnaire.	None
5.	The participants listen to a tool tutorial and receive it in a printed form.	Tutorial is identical except for the similarity model parts.
6.	To better understand the tool and the type of the tasks, the participants use the tool on an example and answer sample questions.	Identical example and questions. Different similarity models used.
7.	The main experiment part: participants in each group receive five software system variants and answer a set of questions concerning their similarity.	Identical software variants analyzed. Identical questions. Different similarity models used.
8.	The participants fill out the debriefing questionnaire.	None
9.	On a later day, the full information on experiment setup and results is presented to all participants.	None

Table 19

The experiment process: steps for controlled experiment execution



In this way, the participants could clarify any doubts before the actual experiment, and the influence of the first learning effects on the experiment result was reduced. The experiment, including the introduction and the tutorial, lasted in total about 90 minutes. The participants were not interrupted, i.e. they worked on the given tasks until completion.

7.2.3 Experiment Results

Removal of one participant During the initial viewing of the experiment results, we decided to remove the answers given by one participant of the control group from the further analysis. The reason for that is that the participant achieved an exceptionally bad result (11 incorrect answers), and stated in the debriefing questionnaire that he/she did not understand the tasks, nor did he/she use the provided analysis tool to gather the information required to solve them. This constitutes a high contrast to the answers of all other participants, who achieved much better results and provided consistently positive feedback on the task understanding and tool usage. Hence, as the answers of the one untypical participant were not included in the further analysis, the both experimental groups analyzed below were equally sized and counted 11 member participants.

The Briefing Questionnaire

Participant background In the briefing questionnaire, we first asked about participant background: the field of study, study level (bachelor, master, or other), study semester, and color blindness. The answers are summarized in Table 20: both groups had similar backgrounds, and the differences between them were not statistically significant (all briefing and debriefing answer difference significances were tested with two-tailed Mann-Whitney U test at  $p = 0.05$ ). In the treatment group, two participants indicated they were color blind – however, in the debriefing questionnaire they both “strongly agreed” that they could “easily see” the color differences in the code editor. Hence, we assumed that the color blindness had no influence on their results.

Question (shortened form)	Treatment Group (Set Model) N = 11	Control Group (Pairwise) N = 11
Field of study	8 x Computer Science 2 x Software Engineering 1 x Business	10 x Computer Science 1 x Software Engineering
Study level	8 x Master, 1 x Diploma, 2 x Bachelor	9 x Master, 2 x Bachelor
Study semester	Average: 9.91 (10 <sup>th</sup> semester)	Average: 9.00 (9 <sup>th</sup> semester)
Color blindness	9 x No 2 x Yes	11 x No

Table 20 The briefing questionnaire results: participant background

Participant experience We further asked the participants about their experience in programming and in the use of methods and tools similar to the evaluated ones. For all questions, the responses were indicated on a five-point Likert scale, as described in the legends in Table 21 and Table 22. The differences between the groups, summarized in Table 21, were not statistically



significant except for one question: the control group had more experience in using diff tools (median: 3, "medium experience") than the treatment group (median: 1, "no experience"). Hence, the control group was more experienced in a method similar to the one they used in the experiment. However, as reported in the next subsection, the control group achieved consistently worse task results. Hence, we consider the different experience to not influence the hypothesis evaluation, as the result of the control group would be probably even worse if its members had less experience, like the treatment group.

Question (shortened form)	Treatment Group (Set Model) N = 11	U Value Significance	Control Group (Pairwise) N = 11
General programming experience		60 No	
Java language experience		61.5 No	
Eclipse environment experience		74 No	
Code comparison using a diff tool (any kind)		96.5 <b>Yes</b>	
Code comparison using the Eclipse Diff tool		75.5 No	
Using the Variant Analysis tool		55 No	
Experience Likert scale: No-Little-Medium-Significant-Professional		U+U'=121	Experience Likert scale: No-Little-Medium-Significant-Professional

Table 21

The briefing questionnaire results: participant experience

Participant motivation

Finally, we asked the participants about their motivation to perform well in the experiment, and received from both groups a response with no statistically significant difference, as shown in Table 22.

Question (shortened form)	Treatment Group (Set Model) N = 11	U Value Significance	Control Group (Pairwise) N = 11
Motivation to perform well in the experiment		70.5 No	
Motivation Likert scale: Highly unmotivated – Unmotivated – Neither motivated nor unmotivated – Motivated – Highly motivated			

Table 22

The briefing questionnaire results: participant motivation

Hypothesis Testing

Symbols and notation	In Figure 57, we present the results of time and answer correctness measurements. We use boxplots to visualize the data distribution and additionally draw a circle representing the average value with error bars showing the 95% confidence interval. In the result description, we use the symbol A for the average (provided with the confidence interval), M for the median, and $\sigma$ for the standard deviation.
Task time results	All participants from the treatment group, using the set model, finished their tasks faster (maximum time: 18 minutes) than the fastest participant from the control group (minimum time: 25 minutes). The treatment group participants needed on average $A=14.0\pm1.35$ minutes to complete the tasks ( $M=14.0$ min, $\sigma=2.28$ min), while the control group participants needed on average $A=33.7\pm4.57$ minutes ( $M=32.0$ min, $\sigma=7.38$ min). In the control group, one participant did not provide the finishing time. However, according to the group supervisor, that participant was neither the fastest, nor the slowest in the group. As the total task time for this person is not known, we report and analyze the time measurement results for a control group size of 10.
Task correctness results	The task correctness was higher in the set group: one participant from that group made two errors, while all others provided fully correct answers ( $A=0.18\pm0.35$ , $M=0.0$ , $\sigma=0.60$ ). In contrast to that, only three participants from the control group provided correct answers for all 16 questions, while the others made between 1 and 6 errors ( $A=2.27\pm1.21$ , $M=2.0$ , $\sigma=2.05$ ).
Cognitive load results	In Figure 58, we present the cognitive load results provided by the participants in the debriefing questionnaire. The treatment group cognitive load ( $A=19.0\pm5.35$ , $M=15.0$ , $\sigma=9.06$ ) was much lower than the control group load ( $A=50.0\pm22.04$ , $M=48.5$ , $\sigma=35.56$ ). In the control group, one participant did not provide a numeric answer. Consequently, we report and analyze the data for the remaining 10 group participants.

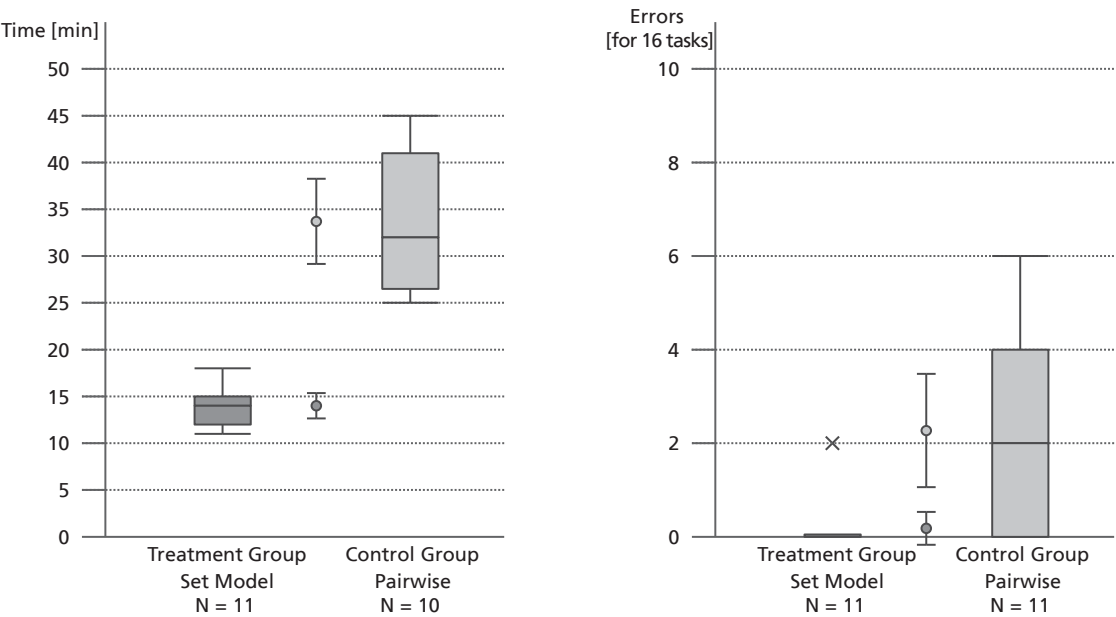


Figure 57 The experiment results: task time (left) and task errors (right).

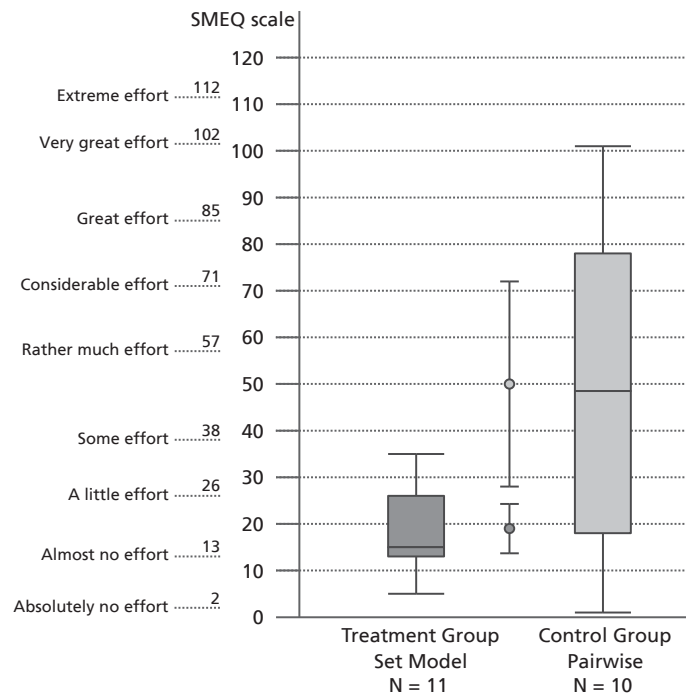


Figure 58

## Hypothesis testing

The experiment results: cognitive load.

From the six above data series, only the task error series of the treatment group does not pass the Shapiro-Wilk normality test. The time and error count values are measured on a ratio scale, while the cognitive load data is provided on an interval scale. Hence, we tested the task time and cognitive load data series with the independent samples Student's t-test, as they fulfill the test requirements: normality and at least interval scale. The task error series, containing not normally distributed ratio scale data, were tested using the Mann-Whitney U test. As all three tested hypotheses are directional, we used one-tailed tests. The results of hypothesis testing are provided in Table 23. Furthermore, to estimate the measured effect size, we provide for all three hypotheses the p value (calculated with the t-test for normally distributed series pairs and with the U test for the task error series) and the Cohen's d [Cohen 1992]. Note that for the task error data series, which is not normally distributed, the value of Cohen's d might be unreliable. For this reason, we additionally calculated the values of Cliff's delta [Cliff 1993], as that parameter is intended for effect size estimation on non-parametric data. Finally, we quantified the observed improvement in two ways: by comparing the averages for the both groups, as well as by comparing the maximum value within the 95% confidence interval of the treatment group to the minimum value within the 95% confidence interval of the control group.

Hypothesis	Accepted at $p < 0.05$	p	Effect size: Cohen's d	Effect size: Cliff's delta	Observed improvement
<b>HSet1</b> Efficiency	Yes (t-test)	3.7e-08	3.30 Large	-1.0 Large	Avg.: 14.0 to 33.7 → 58.5% Conf.: 15.35 to 29.13 → 47.3%
<b>HSet2</b> Correctness	Yes (U test)	0.0048	1.32* Large	-0.66 Large	Avg.: 0.18 to 2.27 → 92.1% Conf.: 0.53 to 1.06 → 50.0%
<b>HSet3</b> Cognitive load	Yes (t-test)	0.0057	1.29 Large	-0.52 Large	Avg.: 19.0 to 50.0 → over 1 cat. Conf.: 24.35 to 27.96 → small

Table 23

Statistical testing of the experimental hypotheses

Result interpretation	All three evaluated hypotheses were accepted in the experiment, and the measured effect sizes were in all cases large. The observed improvement, calculated based on the measured average values, was in all cases larger than initially stated in the hypotheses: the reduction in the task time and task errors between the pairwise group and the set model group amounted to respectively 58.5% and 92.1%, while the cognitive load was lower in the set model group by more than one SMEQ category. The improvement calculated with the more conservative method, by comparing the minimum and maximum edges of the confidence interval, was still substantial for time reduction (47.3%) and task errors reduction (50.0%), and small, but positive for the cognitive load.
Conclusion	We observed in the experiment that the use of the set model indeed results in the previously stated benefits: the similarity analysis which uses the set model is faster, induces less cognitive load, and leads to a better similarity information understanding as compared to the analysis based on the pairwise similarity model. Hence, the original research hypotheses HS2 (Analysis Effort Reduction) and HS4 (Understandability) are already fulfilled when only one of the analysis mechanisms provided by our approach, i.e. the set similarity model, is used. We expect that the other, not yet evaluated approach mechanisms provide further positive contributions to the research hypotheses. Consequently, the evaluation of the improvements provided by the hierarchical abstraction and the defined visualizations is needed to solidify the empirical evidence of the benefits provided by the complete analysis approach.

## The Debriefing Questionnaire

Debriefing questions	In the debriefing questionnaire, we first asked the participants about the experienced cognitive load (analyzed in the previous subsection). Then, we asked a number of control questions concerning the participant views on the experiment tasks and procedures – whether the participants understood these, used them as intended, and had sufficient time. Furthermore, we asked the participants to evaluate the subjectively felt correctness of their answers, as well as the support provided by the used tool for a quick and correct task solution. The answers to the control, correctness, and support questions, summarized in Table 24, were indicated on a five-point Likert scale described in the table legend. Finally, we asked two open questions concerning the participant feedback on the experiment, the tasks, and the used analysis tool.
Control questions results	The majority of participants from both groups indicated a “strong agreement” with the control questions statements, and there was no statistically significant difference between the groups (tested with two-tailed Mann-Whitney U test at $p = 0.05$ ). Hence, we conclude that the participants performed the experiment tasks as intended, having a good understanding of the tasks, the tool, and the provided similarity information – which limits the respective validity threats.

Question (shortened form)	Treatment Group (Set Model) N = 11	U Value Significance (U+U'=121)	Control Group (Pairwise) N = 11
I understood the description of the experimental tasks		49.5 No	
I understood how to use the analysis tool		41 No	
I understood the meaning of the code diff. inform.		53.5 No	
I could easily see the difference between colors		49 No	
I had enough time for solving the tasks		44 No	
I only used the specified tool for solving the tasks		55 No	
I think my answers were correct		38.5 No	
I think the tool supports solving the tasks quickly		20 <b>Yes</b>	
I think the tool supports solving the tasks correctly		33.5 No	
Agreement Likert scale: Strongly disagree – Disagree – Neither agree nor disagree – Agree – Strongly agree			

Table 24

The debriefing questionnaire results

Correctness and support questions results

In the questions concerning the subjectively evaluated answer correctness and tool support, the treatment group indicated a higher agreement with the question statements (mostly “strongly agree”) than the control group (mostly “agree”) – hence giving a more positive evaluation of the set model based analysis method. The difference between the groups was statistically significant for the question concerning the support for quick task solution. We interpret that result as another indication of the set model benefits.

Feedback

The open feedback question answers mainly contained various suggestions concerning the tool improvement. However, all these ideas were already covered by the full Variant Analysis tool, not known to the participants.

### 7.2.4 Threats to Validity

Threat classification	<p>For every experiment, threats to the validity of its results exist. We describe the validity threats of our experiment according to the framework of Cook and Campbell [Cook 1979]. The four distinguished validity types correspond to different stages of an experiment [Wohlin 2000] [Trochim 2006]:</p> <ul style="list-style-type: none"> <li>• Conclusion validity addresses the result analysis, and concerns the existence of statistical relation between the treatment and the outcome.</li> <li>• Internal validity targets the experiment design. It concerns the degree to which the experiment outcome was caused by the treatment.</li> <li>• Construct validity addresses the experiment measurements, i.e. whether the hypothesized cause and its effect were adequately represented and measured in the experiment by the treatment and the outcome.</li> <li>• External validity targets the sampling of experimental objects from the general population, and hence concerns the generalizability of the result.</li> </ul>
Conclusion validity	<p>The identified threats to the <b>conclusion validity</b> are:</p> <ul style="list-style-type: none"> <li>• The moderate size of both experimental groups (11 participants). Performing the experiment with larger groups would result in more statistical support for the conclusions and would reduce the size of the calculated confidence intervals. Nevertheless, the results of hypotheses testing are already statistically significant, with very low p values and non-overlapping confidence intervals. Recruiting a larger, but still relatively homogenous group of participants was not possible.</li> <li>• The analyzed number of errors does not consider for which tasks the errors were made. Although the experimental tasks were not identical, each incorrect answer was assigned the same importance and counted as one task error. We mitigated this threat by formulating tasks having a similar difficulty. Furthermore, this threat does not influence the result evaluation for participants which made no errors: as 10 out of 11 treatment group participants provided fully correct answers, the difference observed between the two groups remains strongly significant.</li> </ul>
Internal validity	<p>The threats to the <b>internal validity</b> of the performed experiment include:</p> <ul style="list-style-type: none"> <li>• Experimental groups potentially unbalanced with regard to factors influencing the outcome. In order to mitigate that threat, we used a random assignment of participants to groups. In the consequence, the randomly created groups were balanced (no statistically significant differences), except for the experience in using diff tools which was higher in the control group. Nevertheless, we consider that difference to not endanger the validity, as the result of the control group, already weaker than the result of the treatment group, would be most probably worse if the group members had less experience.</li> <li>• After splitting the participants in two groups, the groups had different supervisors. The approach author supervised the treatment group, while the control group was supervised by another Fraunhofer IESE researcher. This assignment was selected to avoid the possibility of unintentional negative influence of the control group result by the</li> </ul>

approach author. Although there were no explicit or observed differences in the supervision of the groups, a theoretical possibility of an experiment result influence remains.

- The tool used in the experiment was new to the participants. Moreover, the used similarity analysis method was also new to some participants: four control group participants had “none” or “little” experience in using diff-like code comparison tools, while the set model based comparison was new to all treatment group participants. We mitigated that threat by the use of a pre-experiment tutorial and the example analysis tasks, which familiarized the participants with the used methods and reduced the influence of the ease of method learning on the experiment result (as the method learning occurred mostly before, and not during the experiment).

The other differences between the analysis tasks, methods, or documents, which were not caused by the choice of the similarity model, were eliminated – as discussed previously in this Section. Moreover, no special events, breaks, or other disturbing factors occurred during the experiment.

#### Construct validity

The identified threats to **construct validity**, affecting the appropriateness of the experimental representations constructed for real-world similarity analysis, the analysis effort, and analysis correctness, are:

- Hypothesis guessing by the experiment participants. We mitigated this threat by not mentioning the hypotheses, the tasks and tools of the other group, nor the role of a given group to the participants. Furthermore, the groups worked in separate rooms and hence could not communicate. On the other hand, the participants knew that they participate in an experiment and could easily guess that their results will be evaluated at least for solution time and correctness.
- The participant reaction to the situation of being evaluated. For some people, a test situation might lead to stress and an increase or decrease of individual performance. We mitigated that threat by making the experiment participation voluntary, assuring participant evaluation anonymity, stating that their results are only used for the experimental purpose and not for individual evaluation, and performing the supervision in a relaxed, non-intrusive way. On the other hand, participation in a test is certainly a psychologically different situation than performing a regular workplace task of code analysis.
- Mono-method bias caused by measuring the outcomes with single metrics only. Especially, it can be argued that the correctness of similarity information understanding, being a result of a complex mental process, should be measured with more metrics than just the task errors. Furthermore, the task answers were evaluated in a binary way, i.e. they could only be correct or not. However, in a real-world similarity analysis a partial understanding of similarity information, resulting in an answer which is incomplete but otherwise not incorrect, might be already of value. A form of mitigation of this threat is the use of cognitive load measurement in addition to the task errors.



External validity	<p>Finally, the threats to <b>external validity</b>, affecting the generalizability of the experiment result, are:</p> <ul style="list-style-type: none"> <li>• The experiment participants, i.e. computer science students, might be not representative for a general population of software developers, especially as contrasted to experienced industry practitioners. Furthermore, the familiarity of the participants with the analyzed system variants, and their motivation to solve the analysis tasks, are different than in the industrial case. The industry developers frequently know the analyzed system code well, and the analysis results are highly relevant for their other work tasks. In contrast to that, the students did not know the experimental system variants, and the analysis results were not relevant for their other assignments.</li> <li>• The analyzed software systems might be not representative for a general population of cloned system variants. First, the systems were generated, so that the resulting code similarities might be not typical for the general case. Second, the analyzed systems were written in the Java programming language, while other languages, especially C/C++, are also used in the industry. And third, the size and number of analyzed system variants might influence the scale of the evaluated improvement – which is not addressed in the experiment, as the evaluated system variants provide only a single data point.</li> <li>• Finally, the experimental tasks might be not representative for the general population of similarity analysis tasks. Especially, a real similarity analysis includes tasks of different granularity, addressing single code lines as well as large modules composed of thousands or even millions of code lines. In contrast to that, the experimental tasks had a similar granularity and difficulty, as we wanted to assure their comparability.</li> </ul>
-------------------	--

In the future work, the listed external validity threats can be mitigated by a replication of the presented experiment in different settings involving other types of the participants, the analyzed system variants, and tasks.

### 7.3 Industrial Case Study

Evaluated hypotheses	<p>Although a controlled experiment provides a high degree of control over the execution environment and the factors influencing the evaluated result, the disadvantage of that artificial setting is that the generalizability of the achieved result to real-world situations is necessarily limited. Hence, we performed a case study, described in this Section, to provide additional, practical evaluation input for the hypotheses HS2 (Analysis Effort Reduction) and HS4 (Understandability). Moreover, the case study was an opportunity for evaluating the practical hypotheses HP1 (Migration Effort Reduction), HP2 (Higher Degree of Reuse) and HP3 (Effort Reduction in Parallel Variant Maintenance). Finally, the practical approach use allowed for evaluation of the hypothesis HS5 (Practicability), also including the user satisfaction with the provided results – a concern which is influenced by, but still orthogonal to, the analytical correctness parameters such as recall and precision.</p>
----------------------	--

Use of expert estimations	A frequent issue in case study design is the difficulty of comparing the newly introduced approach to the baseline state. In our case study, we did not have baseline data concerning the similarity analysis efforts spent without the use of our approach, as that task was subsumed as a part of larger development activities. Hence, the differences between the previously used analysis approach, based on pairwise file comparison, and the introduced Variant Analysis approach were estimated by developers who performed similarity analyses with both approaches during their regular work activities. Furthermore, in the description of the case study we deliberately use approximate numbers to protect sensitive information.
Approach application context	The Variant Analysis approach was applied in one software development team at the Diesel Gasoline Systems unit of Robert Bosch GmbH, a large automotive and electronics company. Bosch Diesel Gasoline Systems develops a family of complex embedded software systems, realized in C and C++, which is described in more detail by Tischer et al. [Tischer 2011] [Tischer 2012]. The systems are composed from about ten thousands of components realizing the particular system functions. Each component, typically having a few thousands of code lines, is in turn realized in multiple variants. A decision process is used to determine the realization mechanism for component variants: preprocessor use, branching, or a mixture of these two is possible. However, after some evolution time the similarity properties of the component variant implementation, and the resulting need for maintenance efforts, might lead the team to change the decision concerning the preferred variant implementation mechanism. Moreover, the size and complexity of the system make it difficult to assess the implementation-level component similarity based on the domain knowledge alone – hence creating a need for code analysis approach. Consequently, each of the thousands of components potentially presents a separate similarity analysis case for our approach. Within the whole system, all three application scenarios of our approach occur (Reuse Potential Assessment, Consolidation of Existing Reusable Software, and Parallel Variant Maintenance) – however, in the performed analyses the development team mainly concentrated on the first scenario, targeting the reuse potential assessment among cloned variants.
Data collection	At the time of case study data collection, the development team already used the Variant Analysis approach for about three months. In a questionnaire, we asked the team members to anonymously describe their background and to assess the properties of similarity analyses, performed on the typical analysis problems occurring in their daily work, with two approaches: the baseline one, using pairwise comparison, and the Variant Analysis approach. The questions were mainly targeted at the evaluation of our hypotheses and at the satisfaction of the users with the approach results. For some questions, we first explained the terms we asked for, such as e.g. the “incorrectly understood similarity facts”. The questions provided equal possibility to evaluate any approach as better or equivalent to the other. From five approach users, four returned the questionnaires. The participants indicated they had between 5 and 17 years of industrial experience, and worked on the current project since 1 to 8 years.

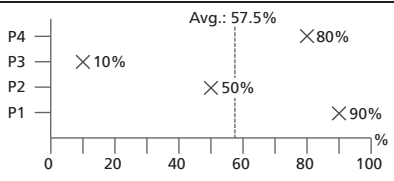
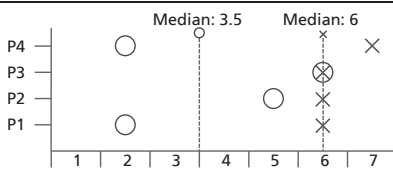
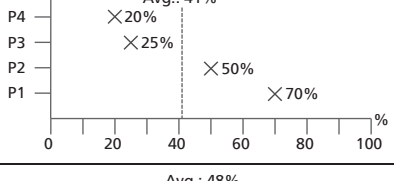

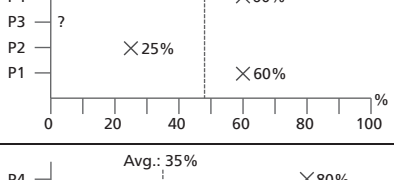
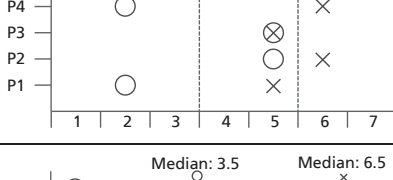
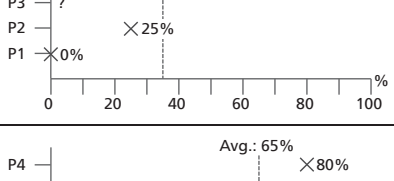
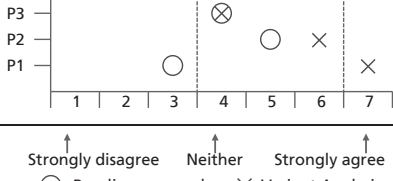
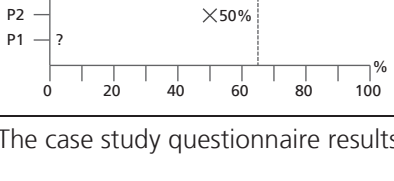
Hypothesis Questions	Participant Answers N = 4	Satisfaction Questions	Participant Answers N = 4
<b>HS2</b> Analysis effort for a typical similarity analysis task (VA reduction %)		The analysis is able to fully satisfy the analysis goals	
<b>HS4</b> Incorrectly understood similarity facts (VA reduction %)		The analysis is able to retrieve the inf. 100% correctly	
<b>HP1</b> Reuse migration effort (VA reduction %)		The analysis is able to retrieve the inf. 100% completely	
<b>HP2</b> Achieved reuse degree (VA increase %)		The analysis is a simple and uncomplicated task	
<b>HP3</b> Effort for parallel variant maintenance (VA reduction %)		<b>Legend</b> Agreement scale: Strongly disagree (1) – Disagree (2) – Slightly disagree (3) – Neither agree nor disagree (4) – Slightly agree (5) – Agree (6) – Strongly Agree (7)	

Table 25

The case study questionnaire results

## Hypothesis questions

The participant answers are summarized in Table 25, with individual participants identified by the codes P1 to P4. For all hypothesis questions, the developers indicated that the Variant Analysis approach provides in their opinion an improvement over the previously used similarity analysis method based on pairwise comparison. Hence, in Table 25 we visualize their estimations as a percentage improvement brought by Variant Analysis over the baseline approach. The developers estimated an average of 57.5% reduction in analysis effort (hypothesis HS2) and 41% reduction in the incorrectly understood similarity facts (hypothesis HS4). Furthermore, they estimated that the total reuse migration effort for their specific cases could be reduced by average 48% (hypothesis HP1). In the case of hypothesis HP2, targeting the missed opportunities of reuse, the theoretically achievable reuse degree is hard to assess in the practice. Hence, we formulated the question differently and asked for the actually achieved reuse degree. The participant estimation was that on average 35% more code can be made reusable when the similarity information provided by Variant Analysis is used. Finally, the participants estimated an average 65% reduction in parallel variant maintenance effort achievable thanks to the Variant Analysis information (hypothesis HP2). Note that not all participants answered the practical hypothesis questions: the missing answers are indicated in Table 25 with question marks.

Satisfaction questions	<p>In the satisfaction questions, we related the two evaluated approaches to an imaginably ideal one, indicating that with respective phrasing of the questions (e.g. “fully satisfy the analysis goals”, “retrieve the information 100% completely”). Rating the two approaches, the developers indicated their agreement with the provided question statements using a seven-point Likert scale described in the legend of Table 25. The questions addressed four types of practical approach benefits: the full satisfaction of analysis goals, the 100% correct information retrieval (which targeted the practically perceived equivalent of the theoretical parameter of precision), the 100% complete information retrieval (i.e. the practically perceived equivalent of recall), and the ease of approach use (i.e. whether the approach use is a “simple and uncomplicated task”). The answers provided by the developers were consistently much more positive for the Variant Analysis approach than for the baseline approach, which supports the hypothesis HS5. The median ratings of Variant Analysis concentrated around the “agree” answer, while the median ratings of the baseline approach were located between “slightly disagree” and “neither” (see Table 25 for details).</p>
Threats to validity	<p>The provided answers, giving a consistently positive indication for Variant Analysis in both hypothesis evaluation and use satisfaction areas, suggest that the approach indeed provides the hypothesized benefits. However, the performed case study has a limited validity. First, the provided improvement numbers are not measured, but only estimated by the developers – albeit based on their practical experience in the use of both approaches. Second, the answers were provided by just four developers – which limits the generalizability of the result. Finally, the developers might interpret the questions in a different way than intended in the hypotheses. This might be the case for participant P3: he/she did not provide practical hypothesis estimations, and indicated identical satisfaction with both approaches, but in the open feedback question he/she stated instead that “the total effort is the same, but the benefit is in the improved quality and less variants”. Our understanding of that answer is that the information provided by Variant Analysis is regarded by that participant as more useful than in the case of the baseline approach, but the resulting practical benefits are in his/her opinion different than these mentioned in our hypotheses.</p>
Further experiences	<p>Currently, after about a year of approach usage, the team evaluated the similarity of over 300 component groups having from 2 to 18 variants. From these groups, about 130 contained variants suitable for merging – this was decided, among other criteria, based on the detected code similarity exceeding 90%. According to the internal measurements, the analysis and code restructuring effort definitely pays itself off in the form of reduced component maintenance needs. The yearly net effort savings, calculated for combined maintenance and development efforts of the addressed components, are estimated to exceed the worth of 100 000 euro. The team members indicated that the similarity analysis with the Variant Analysis approach is convenient and very fast to perform. They also stated that conducting the similarity analyses to the current extent, and hence achieving the maintenance effort savings, would not be possible with the other known approaches due to the prohibitive analysis effort needed.</p>

## 7.4 Industrial Application Experiences

Origin of the presented system groups	<p>The similarity analysis approach developed in this thesis was used in the context of a range of industrial consultancy projects conducted by Fraunhofer IESE. In this Section, we briefly report on five analyses performed in these projects. The presented analyses were all performed on industrial software system variants created with the use of cloning, and were conducted for a concrete project purpose. Hence, the list does not include analyses which were performed for demonstrative or experimental reasons. Furthermore, the list does not include the analyses performed in our case study, as they are already described in the previous Section.</p>
Presented information	<p>For each analysis, Table 26 lists the analysis application scenario, the basic characteristics of the analyzed system group such as the domain, size and number of variants, and their similarity, and it specifies the person who performed the analysis. The five listed analysis cases cover all three application scenarios, with scenario AS1 Reuse Potential Assessment being the most frequent (three occurrences). The analyzed system groups include medium-sized and large systems (group average system sizes from 112 KLOC to 1319 KLOC), and contain from 4 to 14 cloned variants. All systems are realized in the C and C++ programming languages, and are deployed as embedded software. In the listed cases, the analyses were performed by the approach author, other IESE researchers, as well as by the customer employees. This shows, as in the case study, that the approach can be successfully applied by software practitioners, supporting the hypothesis HS5 (Practicability).</p>
Application details	<p>In all cases, the performed analysis provided a new view on the customer code, allowing for assessment of the variant system similarity which was, according to customer feedback, not possible with other means:</p> <ul style="list-style-type: none"><li>• In company J, the general high similarity of the four system variants was suspected, but no respective measurements existed. The performed analysis identified groups of strongly similar components which were suitable for unification across the four variants with low effort, and a range of further component groups with potentially sufficient similarity. An initial planning for reengineering activities was started – however, the company decided instead to develop a new generation of products, not code-compatible with the current one, and discontinued the analyzed variants.</li><li>• Company D develops a product line of power electronics systems since several years. The product line architect was interested how far the existing code components, which are similar to each other across the product variants, correspond to the planned reusable components as documented by the software architecture. The similarity analysis revealed that the components intended to be reusable were indeed highly similar, and the other similar components found in the code constituted exceptions previously known to the architect. Hence, no further actions were performed after the analysis.</li></ul>



Company name (anonymized)	Domain	Application scenario	Analysis done by	Number of variants	Average variant system size	Core code size	Average unique code size
J	Machine construction	Reuse assessment	Approach author	4	1,319 KLOC	664 KLOC	131 KLOC
D	Power electronics	Consolidation	Customer	10	427 KLOC	161 KLOC	152 KLOC
H	Automotive	Parallel maintenance	Other IESE researcher	14	186 KLOC	132 KLOC	2 KLOC
C	Automotive	Reuse assessment	Approach author	12	112 KLOC	24 KLOC	2 KLOC
U	Telecommunication	Reuse assessment	Other IESE researcher	6	202 KLOC	145 KLOC	36 KLOC

Table 26 Industrial applications of the analysis approach

- Company H develops a group of 14 cloned, similar software systems. However, no code unification is intended, as the reuse introduction was judged to conflict with system safety considerations. Nevertheless, the company was interested in identifying groups of highly similar components, and in identifying variants which nearly fully cover the code of other, smaller variants. For example, it was found that 3 of the 14 system variants cover over 99% of the complete set union code. The provided similarity information was used to improve the planning of code inspection efforts.
- Company C develops several product lines of controllers and drivers used in automotive parts. The analyzed product line was intended to be replaced by a new one. The similarity information provided in the analysis was used in two further activities. First, in the context of scoping, the similarity information supported the assessment of reuse potential for newly developed components, functionally analogical to the old ones. Second, a detailed review of code differences helped the developers of selected components to understand fine-grained implementation-level peculiarities of each variant, not large enough to be visible at the level of whole features or functions.
- Company U intends to introduce the software product line approach to its family of similar, cloned software variants. The similarity analysis was performed to help assess whether the product line approach is suitable for the system group. The analysis indicated a significant reuse potential and identified groups of highly similar components. The next reengineering steps are currently under consideration.

Further experiences

Generalizing the more detailed experiences gained in the above industrial application cases, we also observed that:

- Typically the performed analyses confirm the fragments of similarity information which are already known, and provide much more information which was previously unknown to the participating customer employees. This is consistent with the observations we made in the industrial survey in Section 3.1, regarding the generally low availability of code similarity information.

- The high information detail level provided by the analysis due to the use of the diff algorithm was considered by the developers to be very helpful. In many cases, the presented differences in single code lines, not sufficiently relevant for a high-level similarity assessment such as scoping, were rated by the developers as highly important as they contained e.g. hardware-specific mechanisms for component functionality realization.
- In all cases the customers provided a positive feedback on the analysis and its results, stating that it delivers an added value compared to the previous state of the practice. The information provided by the analyses was assessed by the customers as correct and useful in their maintenance decisions. Hence, the customer feedback supports the hypothesis HS5 (Practicability).

## 7.5 Summary

In this Chapter we evaluated the practical and scientific hypotheses concerning the Variant Analysis approach. The performed evaluations provided consistently positive outcomes, supporting the scientific and practical hypotheses. The described results were all created based on a realistically available data input – hence, the measured improvement values are not absolute, but rather indicate the benefit achieved through the approach use in the concrete context.

In the analytical correctness evaluation we measured that, for the analyzed cases, the precision and recall of the approach results are higher than 0.99. We also discussed that the transitivity of the input provided by the diff algorithm is sufficient for the purposes of set model construction, with at least 99.56% of input graphs, containing at least 97.52% of input edges, being transitive, and at least 99.25% of the input edges being included in the finally constructed set model. Furthermore, we have demonstrated that the approach implementation is scalable and performant.

In the controlled experiment, we addressed the benefits of the set similarity model relative to the most frequently used state of the art approach, the pairwise comparison. We observed that the set model group solved the similarity analysis tasks on average 58.5% faster, while making 92.1% less errors than the pairwise comparison group. Furthermore, the set model group reported a lower cognitive load when solving the tasks and a higher satisfaction with the used analysis approach.

Finally, a case study demonstrated that the analysis approach can be successfully integrated in the software maintenance activities in the industrial context. The industrial approach users estimated that the hypothesized practical improvements, such as migration effort reduction and the increase of achieved reuse degree, do occur in the practice. The use of the analysis results enabled net effort savings exceeding 100 000 euro yearly. The usefulness of the approach results was further confirmed by the described industrial application experiences in five different companies.



## 8 Summary and Outlook

In this last Chapter we describe the concluding reflections over the thesis content and the defined similarity analysis approach. We discuss the provided contributions (Section 8.1) and the approach limitations (Section 8.2), and we outline the possible future work areas (Section 8.3).

### 8.1 Results and Contributions

Addressed problems	<p>In the industrial practice, groups of functionally similar software systems are frequently developed without considering software reuse approaches. Instead, the code of existing systems is repeatedly cloned and adapted to the specific customer's needs, creating new system variants. This approach results in short-term advantages such as reduced first development effort and shortened time to market, but its long-term disadvantage is the significantly increased maintenance effort. Hence, even if reuse introduction was not intended or not possible in the short term, in the longer term a consolidation of the variants into a reusable form is beneficial. However, the consolidation is difficult: the practically occurring problems are the high required effort and the possibility to miss the reuse opportunities. One of the reasons for these problems is that the cross-variant code similarity information, necessary in the consolidation process, is not available. Hence, a related scientific problem is the recovery, structuring and presentation of code similarity information, in quality and detail sufficient to support the migration and maintenance activities. This problem is open as the existing reverse engineering approaches, which can be used to recover that information, exhibit deficiencies which prevent them from fully addressing the respective information needs.</p>
--------------------	---

#### 8.1.1 Understanding Large-Scale Cloning: Reasons, Consequences and Solutions

Contributions to problem understanding	<p>In this thesis, we contribute both to the understanding of the mentioned practical and scientific problems, as well as to their solution. Our contributions to the problem understanding are:</p> <ul style="list-style-type: none"> <li>• <b>A survey of the large-scale cloning practices</b> in the industry, performed on six groups of similar system variants. In the survey, we investigated the reasons for cloning, the practically perceived benefits and drawbacks, and the cloning consequences. The survey results show that cloning causes long term-maintenance problems, and leads to the loss of information on the similar assets – which hinders reuse adoption. On the other hand, cloning is in many situations a justified or even preferred approach, as its short-term advantages over software reuse are a low entrance barrier, reduced first development effort,</li> </ul>
--	---

possibility of a quick reaction to unexpected market demands, reduced cross-project synchronization requirements, and freedom to perform experimental changes. We conclude that cloning will remain a practically used development approach, and hence the need to counteract the resulting maintenance problems will persist.

- **A characterization of the state of the art approaches** to similarity analysis of multiple cloned variants. First, we discussed that the missing similarity information needs to be provided by a reverse engineering approach, as the other alternative, a top-down human-based analysis, overlooks detailed differences relevant to the asset functionality implementation. Second, we discussed the properties of the existing reverse engineering approaches with respect to provided abstraction mechanisms and information detail level on two dimensions of analysis problem complexity: the size of the analyzed software assets, and the amount of their variants. We showed that while the approaches deliver good support for the asset size dimension, the information provided for the variant dimension is incomplete, and suitable abstraction mechanisms supporting understanding and interpretation of the variant similarity information are missing.

### 8.1.2 A Set Model Based Approach to Variant Similarity Analysis

Scientific contributions to problem solution

Consequently, we analyzed the identified practical and scientific problems and formulated application scenarios specifying the problem scope addressed in the thesis. We address the scenarios of the reuse potential assessment across the variants, the consolidation of already existing, partially reusable software, and of the code similarity based support for parallel variant maintenance. For the defined application scope, we developed a reverse engineering approach to variant similarity information recovery, which is the main contribution of this thesis. The core idea of the approach is the use of hierarchical set similarity model to represent the similarity of analyzed system variants. The set model is in turn constructed and used by a generic analysis framework, specifying the analysis process and the data model, and it is presented by a range of defined visualizations. Hence, our scientific contributions to the problem solution are:

- **A definition of a conceptual similarity model**, which classifies and relates the concepts associated with the variant similarity analysis problem and provides the basis for reasoning on the solutions.
- **A definition of construction requirements** for variant similarity analysis approaches, which are derived from the properties of software variants – especially from the lack of objectively definable variant order. The requirements are applicable to any general-purpose approach analyzing variant similarity, provide means to compare and evaluate such approaches, and can serve as guidance when defining a new approach.
- **A definition of the hierarchical set similarity model**, based on a formalization of the variant similarity analysis and its results. The set similarity model represents the analyzed variants as sets of atomic, comparable content elements. The sets intersect with each other,

expressing the variant similarity: the elements similar across a group of variants are placed inside the respective set intersection, while the other elements remain outside. The set model is combined with the use of tree structures, representing the asset content hierarchy, in result defining the hierarchical set similarity model. This enables the use of set model based similarity analyses, measurements and visualizations on any granularity level in the asset structure. The hierarchical set similarity model allows for understandable analysis and presentation of similarity information for both large software assets and a high amount of variants, while at the same time providing the access to the lowest detail level of single content elements. The model definition is generic, which makes it applicable for similarity analysis of many asset types, also non-software ones.

- **A definition of a generic analysis framework** using the hierarchical set similarity model. The framework includes the data model, the analysis process, and the requirements concerning the definition of three basic analysis mechanisms used in our approach: an asset content decomposition, a structure element equivalence relation, and an atomic content element equivalence relation. These three basic analysis mechanisms need to be defined for each specific asset content type and analysis goal – hence, they constitute the customization points of the generic framework and enable the use of the, likewise generic, hierarchical set similarity model and its metrics and visualizations. Consequently, the analysis framework can be used for diverse system representations and diverse similarity detection algorithms. In the thesis we provide an example instantiation of the framework for file system based structures, with file content treated as text and analyzed with the diff algorithm. That instantiation is suitable for analysis of assets developed in the most of the currently existing programming languages.
- **A definition of hierarchical set model visualizations**, presenting the similarity of intersecting variant content sets, the distribution of the similarity in the asset structure hierarchy, and the status of the particular structure and content elements. Most of the defined visualizations are applicable for any group of intersecting sets, regardless of their origin – hence, they can be applied beyond the context of this thesis, and even beyond the area of computer science.

### 8.1.3 Empiricism and Evaluation

Empirical contributions

We hypothesize that the defined similarity analysis approach provides a range of benefits compared to the state of the art. We evaluated these benefits empirically through a controlled experiment, a case study, and industrial applications of our approach, with consistently positive results:

- In **the controlled experiment**, we observed that the use of the set model reduces the similarity analysis effort (experimental result: 58.5% shorter analysis time) and leads to a better similarity information understanding (experimental result: 92.1% less task errors) as compared to the analysis based on the pairwise similarity model.

Although the experiment targeted the set model only, we hypothesize that the improvement provided by the whole approach, compared to the state of the art approaches, has a similar scale. Moreover, we hypothesize that the scale of the improvement increases with the increasing amount of the analyzed variants, due to the use of the abstraction mechanisms defined in the approach.

- In **the case study**, the approach was applied for over a year by one development team in a large electronics company. The approach use allowed for development and maintenance net effort savings exceeding 100 000 euro yearly. After initial three months of approach use, the developers indicated that they were much more satisfied with our approach than with the previously used approach based on pairwise comparison. On average, they estimated that the practical analysis effort is lower by 57.5% and the amount of incorrectly understood similarity facts is reduced by 41% – providing a further support for the hypotheses evaluated in the experiment. They also estimated that the use of approach results allows for reducing the reuse migration effort (average estimation: 48%), increasing the amount of reused code (35%) and reducing the effort for parallel maintenance of non-migrated variants (65%) – hence confirming all our practical benefit hypotheses.
- **The industrial application experiences** demonstrated that the approach results are useful in the practice in a variety of contexts, and they showed that the defined approach can be successfully used by software practitioners.

Analytical  
approach  
evaluation

Additionally, we performed an **analytical evaluation** of the correctness and performance of our approach. For the analyzed cases, the measured precision and recall of the approach were very high (precision  $\geq 0.9946$ , recall  $\geq 0.9953$ ). Furthermore, performing the similarity analysis required at most a few minutes, except for the largest system groups or the system groups containing many renamed files.

Additionally, in the analytical evaluation we measured the transitivity of the input similarity data provided by the diff algorithm and the proportion of the input similarity data included in the resulting set model. As the similarity represented by the set model is transitive, but the input data might be not transitively similar, ensuring a minimal divergence between the input similarity and the set model is necessary. In the measured examples, at least 99.56% of the input graphs, containing at least 97.52% of input edges, were already transitive, and the processing of the remaining graphs allowed for including at least 99.25% of the input edges into the set model. No additional, artificial edges were included – hence, the modeled similarity information is 100% correct, but not 100% complete. We discussed that the small divergence in the represented similarity information constitutes a low cost paid in exchange for the several benefits of the set model, discussed above. This was confirmed by the industrial developers using our approach, who were satisfied with the correctness and completeness of the analysis results.

### 8.1.4 Further Approach Benefits

Set model benefits	We consider the hierarchical set model and the visualizations defined on top of it to be the main factor which enables the achievement of the approach benefits. The hierarchical set model allows for <b>a scalable abstraction of the analysis result</b> , while also preserving the full information detail, for both dimensions of the system size and of the amount of variants. The provided similarity visualizations and measurements are easily understandable for both small and large code structures, up to millions code lines and tens of variants. Consequently, the identification of similar variant groups, similar component variants and similar code fragments can be performed with low effort. Moreover, the information stored in the set model is detailed and can be further processed, for example by using subset calculations, metrics, and aggregative visualizations such as phylogenetic trees.
Textual instantiation benefits	The described instantiation of the approach for textual asset content, and the choice of the diff algorithm as the content equivalence function, have the advantages of <b>generality and simplicity</b> . The content of a broad range of asset types is physically stored in textual files, and can hence be processed by the analysis instantiation. At the same time, the use of well-known and simple diff algorithm helps technical stakeholders in understanding the analysis process and trusting its results.
Enabling and extending software reuse benefits	Finally, the benefits of the approach application extend beyond the performed analysis and its direct use in the migration tasks. In the case study, as well as in the industrial experiences, the introduction of the approach enabled performing the similarity analyses to a much larger extent than realistically possible before, as the baseline approach effort was previously considered to be prohibitively high. The new analysis approach enabled the case study organization to not only achieve better effort and reuse results in the particular analyses, but also to perform more analyses and hence <b>increase the scope of reuse migration</b> . This in turn is likely to strengthen the benefits which are typically associated with software reuse introduction, such as maintenance effort reduction, shorter development time, and better software quality – as already expressed by the effort savings estimated in the case study organization.

## 8.2 Limitations

Scope of the approach	Before the construction of the analysis approach, we formulated a group of assumptions concerning the form of the similarity information needed in the context of <b>the defined application scenarios</b> . First of all, we explicitly excluded from the analysis scope the similar code fragments located within the same software asset. Consequently, only the similarity between the variants is reported. Furthermore, we assumed that the similar asset elements from different variants should be related to each other with one-to-one and not many-to-many correspondences. In result, instead of many potential matches, only a single most similar counterpart
-----------------------	---

is reported for a given element. Finally, we assumed a high structural similarity of the analyzed variants, motivated by the observation that low-similarity code cannot be effectively merged into a reusable form. The consequent design decisions might cause the approach to provide reduced result quality for low-similarity analysis input. The defined analysis approach is therefore particularly suitable for similarity analysis of variant assets which were created via cloning and, due to the common origin, still share structural similarity. Outside of the application scenario scope, the defined approach likely does not provide the stated benefits. Particularly if no assumptions concerning the properties of the sought similarity can be made, and hence a search for many similar code fragments within and across the variants is needed, the construction of a respective hierarchical set model storing the analysis result is not possible.

#### Result similarity divergence

A further limitation of the approach resulting from the analysis assumptions is the potential **divergence** between the complete similarity of the analyzed asset variants and the similarity analysis result stored in the set model. As discussed in the previous Section, the input similarity of analyzed variants might be not transitive, but the transitivity is required in the set model construction, and a respective transformation of the input similarity creates the stated divergence. Although in the case of the diff algorithm that divergence is small, it can be potentially much larger for other similarity detection functions or other asset types. Hence, the size of the divergence needs to be measured in order to estimate how far the consequent incompleteness of the analysis result reduces the approach benefits for the concrete asset type.

#### Limitations of syntactic similarity

Finally, any instantiation of the approach needs to consider that the most functions recognizing the similarity of software assets target the **syntactic similarity**, as its detection is much easier to automate than the detection of semantic similarity. However, two asset content fragments which realize the same functionality but were implemented in different ways mostly do not look syntactically alike. In the context of clone detection in program code, Juergens et al. discuss that semantically similar code fragments occur frequently, but the current clone detection approaches cannot be improved to detect them [Juergens 2010]. Hence, the difference between the detectable syntactic similarity and the reuse-relevant semantic similarity causes some potentially reusable asset fragments to remain undetected. Although the syntax-based processing of program code, performed by the described textual approach instantiation, leads to the demonstrated practically useful results, the difficulty in recognizing meaningfully similar asset content might be significantly higher for more complex asset representations such as models. In such a case, the defined approach would not be able to provide its benefits, as it depends on the quality of the used equivalence relations, fulfilling the similarity detection task. On the other hand, if a semantic equivalence relation would be constructed in the future, it could be easily incorporated into the existing generic analysis framework of our approach.



## 8.3 Future Work

We discussed and demonstrated that the similarity analysis approach defined in this thesis provides a contribution to the state of the art and of the practice. Nevertheless, there are still many open points in the approach development and evaluation which need to be addressed in the future. Moreover, during the thesis research many new questions, related to the research and practical context of the approach, emerged. We summarize the open points and the research questions in the following subsections.

### 8.3.1 Extending the Analysis Approach

Definition and evaluation of further instantiations	We instantiated the defined analysis approach for a textual, file system based representation of the asset content, and constructed the asset content equivalence relation by using the diff algorithm. In the future, it is interesting to <b>address other asset types and other equivalence relations</b> . Set model based analyses of code abstract syntax trees, models (including executable models defined by visual programming languages such as Simulink), and even non-software assets are imaginable. To define these analyses, additional work is needed in the area of analysis mechanism definition (content decomposition, equivalence relations), measurement of analysis result properties (especially related to correctness and transitivity), and evaluation of instantiation-specific approach benefits.
Improvements of analysis mechanisms	Moreover, the defined <b>generic analysis mechanisms could be further extended</b> . On the one hand, the extensions could improve and automate the support for addressing more specific analysis concerns. For example, new visualizations such as histograms and scatterplots could provide another view on the measured similarity values and the calculated metrics, while new calculations could automatically provide answers to the most frequent questions (e.g. finding top $m$ similar variant groups of size $k$ ). On the other hand, also the generic analysis mechanisms can be extended: for example, it might be helpful for the users to offer size-proportional Venn diagram visualizations when up to five variants are presented.
Use of context information	<b>More information sources could be used during the analysis.</b> Especially the information provided by configuration management systems, such as file cloning and renaming history, file maintenance intensity, and co-change analysis, could be used either for improving the analysis result quality, or for supporting the migration decision making.
Improvements of user support	Finally, <b>the application guidance for the approach can be extended</b> . With more application experience, the guidance for analysis result interpretation could be improved by adding further suitability criteria and providing more decision suggestions supported by practical use cases. The guidance could also address the migration process itself by helping the user to decompose the high-level migration goals into finer-grained migration activities and matching them with best practices and reengineering patterns similar to these of Demeyer et al. [Demeyer 2008].



### 8.3.2 Further Evaluation

Evaluation topics	<p>The performed evaluations of the approach provide initial indications regarding the contributed benefits. However, a <b>further empirical and analytical evaluation</b> is needed to strengthen the existing evidence and provide more understanding of the influencing context factors:</p> <ul style="list-style-type: none"><li>• The performed experiment provided just a single data point for evaluating the scientific hypotheses – hence, its replication for different types of participants, variants and tasks is of much value.</li><li>• The experiment addressed just the benefits of the set model, while other approach mechanisms such as visualizations should also be evaluated.</li><li>• Further experimental evaluations should also provide input for validating the hypothesis HS3 Analysis Effort Scalability, stating that the effort savings increase with the amount of analyzed variants.</li><li>• Performing further case studies is important to extend the validation of practical hypotheses. It would be also interesting to gather experiences regarding approach utility for a very high number of variants (30 and more).</li><li>• The usefulness of the defined similarity metrics in the estimation of reuse potential is likewise worth evaluating.</li><li>• The correctness and transitivity measurements should be performed for more systems, especially for real-world software variants.</li></ul>
Experience collection	<p>Finally, <b>collecting further practical experience</b> will help to improve the approach by providing a better understanding of the details of the possible analysis goals, the usefulness of different similarity information types in addressing these goals, the role of the similarity information in the process of software reuse migration, and the suitable migration strategies.</p>

### 8.3.3 Open Research Questions

	<p>The developed approach addresses different computer science areas, such as software maintenance, reverse engineering and reengineering, program understanding, and software reuse. Accordingly, the open research questions address all these areas.</p>
Understanding maintenance problems	<p>First of all, <b>understanding of the large-scale cloning practices</b> can be improved by a comprehensive study of evolving cloned system variants. In the context of our approach, the study results could be used especially to extend the interpretation and migration guidance. The questions which could be addressed are:</p> <ul style="list-style-type: none"><li>• How do the differences between variants develop with time, regarding their size, distribution, and granularity?</li><li>• Do systems cloned by copying look differently than the systems cloned using configuration management branches? Do industrial cloned variants look differently than open source variants?</li></ul>

- How long do the benefits of cloning prevail over its disadvantages? When is it beneficial to consolidate the cloned systems, and when is it better to keep them separated? What are the strategies to manage the interplay of cloning and reuse, as attempted by some of the industrial survey and case study organizations?
- How fast is the original similarity information being lost?
- Is there a point when is it too late for code consolidation, due to the extent and complexity of the differences between the variants?

Combining  
reverse  
engineering  
techniques

Furthermore, **the reverse engineering of code similarity** could be potentially improved by using not only content comparison, but also including the results of other analysis techniques for mutual benefit. For example, techniques such as feature location could help in locating similar content and in result interpretation, while the similarity information could in turn help to better identify feature differences between the variants. It is also interesting to investigate how the semantic information about the analyzed systems (e.g. provided by scoping or an existing feature model) could be used to better guide the automatized analysis process.

Similarity  
visualization  
and  
understanding

Third, **the visualization and understanding support** for the created similarity information should be deeper researched. We are convinced that many more visualizations of the hierarchical set model information can be defined. The existing visualizations can also be improved – for example, further supportive hierarchy structures for the tree map set diagram should be proposed and evaluated. It would also be interesting to investigate the benefits of different similarity visualizations from the psychology and program understanding point of view. These investigations should include an evaluation of the color use in the diagrams, and could propose other colors which better support similarity understanding.

Reuse  
adoption

Fourth, **the reuse migration and the parallel variant maintenance**, including the role of the provided similarity information, need to be better understood. From our point of view, the interesting questions are:

- What are the estimators for migration difficulty and for migration benefits which could be derived from the assets?
- How should they influence the migration, considering that there are many other factors influencing the migration decisions?
- What would be, from the technical point of view, the best way to merge the asset variants in a given specific case?
- What other information could be reverse engineered to support reuse migration and parallel variant maintenance?

Suitability for  
further  
applications

Finally, the defined analysis approach is generic, and the hierarchical set model can be used to represent the similarity of any group of elements. In the future, it would be very interesting to **investigate the applications and benefits of the approach for a broader range of assets** and element representations, also beyond the scope of computer science.

Collections of overlapping hierarchical data, potentially suitable for our approach, exist for example in biology (genome comparisons [Kestler 2004] [Fouts 2005] [Argout 2011] [Madak-Erdogan 2013]) and medicine (disease prevalence [Mapel 2004], comparison of different diagnostic methods [Walline 2013]). Potentially suitable data collections exist also in further contexts, where a population of persons, hierarchically structured according to location, age, occupation, or other partitioning attribute, is analyzed with the use of multiple binary attributes describing their genetic [Willer 2013], economic [Noack 2011], demographic [USCB 2014], behavioral [Utter 2007], medical [Viegi 2004] or social [Zammit 2012] characteristics.

## 8.4 Concluding Remarks

The research topic of this thesis, that is the similarity analysis of cloned asset variants, originated from our practical observations of industrial problem cases which we encountered in the Fraunhofer IESE consultancy projects. The first solution was already able to provide useful results to the supported company [Duszynski 2008] – however, it also had several drawbacks. Actually, we learned about the most of the approach construction requirements listed in Section 4.2 by investigating the reasons for the deficiencies of the first solution. Consequently, a next version of the analysis approach was created, and refinements resulting from further experiences were added successively.

This cyclical pattern of problem identification, solution development, practical solution application and post-application reflection shaped this thesis research. Hence, we consider the thesis to be an example of empirically supported applied research: the thesis was influenced by practical problems (investigated in the industrial survey), it developed theoretical research results (evaluated in the controlled experiment), and included practical result application (presented in the case study). In our opinion, the possibility to apply the research ideas in a practical context greatly contributed to the maturation of the defined approach, and the successively added improvements resulting from the practical feedback enabled the achievement of the approach benefits.

The positive outcomes of performed empirical evaluations, as well as the successful experiences of industrial approach application, make us convinced that this thesis provides a good contribution to the area of software similarity analysis in the context of the defined application scenarios. Moreover, the generic contributions of the approach, especially the hierarchical set similarity model and its visualizations, can be used in similarity analysis of any other objects fulfilling the set model assumptions. Hence, investigating the suitability of the generic approach contributions outside of their original, software-based context remains an interesting open problem.

## References

- [Aho 2006] Aho, A.; Lam, M.; Sethi, R.; Ullman, J.; *Compilers: Principles, Techniques, and Tools* (2<sup>nd</sup> Edition). Addison Wesley, 2006.
- [Albers 2012] Albers, M.; *Human-Information Interaction and Technical Communication. Concepts and Frameworks*. IGI Global, 2012.
- [Anastasopoulos 2001] Anastasopoulos, M.; Gacek, C.; *Implementing Product Line Variabilities*. In *Proceedings of the 2001 Symposium on Software Reusability (SSR '01)*, 2001, Toronto, Canada, pp. 109—117.
- [Apel 2013] Apel, S.; Kästner, C.; Lengauer, C.; *Language-Independent and Automated Software Composition: The FeatureHouse Experience*. In *IEEE Transactions on Software Engineering*, Vol. 39, Issue 1, January 2013, pp. 63—79.
- [Apiwattanapong 2007] Apiwattanapong, T.; Orso, A.; Harrold, M.J.; *JDiff: A Differencing Technique and Tool for Object-Oriented Programs*. In *International Journal Automated Software Engineering*, Vol. 14, Issue 1, March 2007, pp. 3—36.
- [Argout 2011] Argout, X.; Salse, J.; Aury, J.M.; Guiltinan, M.J.; Droc, G.; Gouzy, J.; Allegre, M. et al.; *The Genome of Theobroma Cacao*. In *Nature Genetics*, Vol. 43, 2011, pp. 101—108.
- [Asaduzzaman 2011] Asaduzzaman, M.; Roy, C.K.; Schneider, K.; *VisCad: Flexible Code Clone Analysis Support for NiCad*. In *Proceedings of the 5<sup>th</sup> International Workshop on Software Clones (IWSC 2011)*, 2011.
- [Awad 2004] Awad, E.M.; Ghaziri, H.M.; *Knowledge Management*. Pearson Education International, Upper Saddle River, NJ, 2004.
- [Bansal 2004] Bansal, N.; Blum, A.; Chawla, S.; *Correlation Clustering*. In *Machine Learning Journal*, Vol. 56, Issue 1-3, July 2004, pp. 89—113.
- [Barns 1991] Barns, B.H.; Bollinger, T.B.; *Making Reuse Cost-Effective*. In *IEEE Software*, Vol. 8, Issue 1, January 1991, pp. 13—24.
- [Basili 1993] Basili, V.R.; *The Experimental Paradigm in Software Engineering*. In: Rombach, H.D. (Ed.); Basili, V.R. (Ed.); Selby, R. (Ed.); *Experimental Software Engineering Issues: Critical Assessment and Future Directives*. LNCS Volume 706, Springer-Verlag, Berlin Heidelberg, 1993.
- [Bass 2003] Bass, L.; Clements, P.; Kazman, R.; *Software Architecture in Practice*. Second Edition. Addison-Wesley, 2003.

- [Bayer 1999] Bayer, J.; Girard, J.F.; Würthner, M.; DeBaud, J.M.; Apel, M.; Transitioning Legacy Assets to a Product Line Architecture. In Proceedings of the 7<sup>th</sup> European Software Engineering Conference (ESEC/FSE '99), LNCS Volume 1687, 1999, pp. 446—463.
- [Bayer 2004] Bayer, J.; View-Based Software Documentation. Dissertation, PhD Theses in Experimental Software Engineering, Vol. 15, Fraunhofer Verlag, Stuttgart, Germany, 2004.
- [Bederson 2002] Bederson, B.B.; Shneiderman, B.; Wattenberg, M.; Ordered and Quantum Treemaps: Making Effective Use of 2D Space to Display Hierarchies. In ACM Transactions on Graphics, Vol. 21, Issue 4, October 2002, pp. 833—854.
- [Bellon 2007] Bellon, S.; Koschke, R.; Antoniol, G.; Krinke, J.; Merlo, E.; Comparison and Evaluation of Clone Detection Tools. In IEEE Transactions on Software Engineering, Vol. 33, No. 9, 2007, pp. 577—591.
- [Benedusi 1992] Benedusi, P.; Cimitile, A.; de Carlini, U.; Reverse Engineering Processes, Design Document Production, and Structure Charts. Journal of Systems and Software, Vol. 19, No. 3, 1992, pp. 225—245.
- [Berger 2010] Berger, C.; Rendel, H.; Rumpe, B.; Busse, C.; Jablonski, T.; Wolf, F.; Product Line Metrics for Legacy Software in Practice. In Proceedings of the 14<sup>th</sup> International Software Product Line Conference (SPLC 2010), Volume 2, 2010, pp. 247—250.
- [Berger 2013] Berger, T.; Rublack, R.; Nair, D.; Atlee, J.M.; Becker, M.; Czarnecki, K.; Wasowski, A.; A Survey of Variability Modeling in Industrial Practice. In Proceedings of the Seventh International Workshop on Variability Modeling of Software-intensive Systems (VaMoS '13), 2013, pp. 1—8.
- [Beydeda 2005] Beydeda, S. (Ed.); Book, M. (Ed.); Gruhn, V. (Ed.); Model-Driven Software Development, Springer-Verlag, Berlin Heidelberg, 2005.
- [Beyer 2008] Beyer, H.; Hein, D.; Schitter, C.; Knodel, J.; Muthig, D.; Naab, M.; Introducing Architecture-centric Reuse into a Small Development Organization. In Proceedings of the 10<sup>th</sup> International Conference on Software Reuse (ICSR 2008), 2008, pp. 1—13.
- [BeyondCompare 2014] Beyond Compare – a Commercial Tool for Comparing Files and Folders. Information available under <http://www.scootersoftware.com/> (accessed on 4 May 2014).
- [Böckle 2004] Böckle, G.; Clements, P.; McGregor, J. D.; Muthig, D.; Schmid, K.; Calculating ROI for Software Product Lines. In IEEE Software, Vol. 21, Issue 3, May-June 2004, pp. 23—31.
- [Bosch 2002] Bosch, J.; Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization. In Proceedings of the 2<sup>nd</sup> International Software Product Line Conference, LNCS Volume 2379, Springer Verlag, 2002, pp. 257—271.

- 
- [Briand 1996] Briand, L.C.; Differding, C.M.; Rombach, H.D.; Practical Guidelines for Measurement-Based Process Improvement. In *Software Process Improvement and Practice*, Vol. 2, Issue 4, Dec. 1996, pp. 253—280.
- [Canfora 2007] Canfora, G.; Di Penta, M.; New Frontiers of Reverse Engineering. In *Proceedings of the 2007 Workshop on the Future of Software Engineering (FOSE 2007)*, 2007, Minneapolis, MN, USA.
- [Chikofsky 1990] Chikofsky, E.; Cross, J.; Reverse Engineering and Design Recovery: a Taxonomy. In *IEEE Software*, Vol. 7, No. 1, Jan. 1990, pp. 13—17.
- [Clements 2002a] Clements, P.; Northrop, L.; *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.
- [Clements 2002b] Clements, P.; Krueger, C.W.; Point/Counterpoint: Being Proactive Pays Off – Eliminating the Adoption Barrier. In *IEEE Software*, Vol. 19, No. 4, July-August 2002, pp. 28—31.
- [Cliff 1993] Cliff, N.; Dominance Statistics: Ordinal Analyses to Answer Ordinal Questions. In *Psychological Bulletin*, Vol. 114, 1993, pp. 494—509.
- [Cohen 1992] Cohen, J.; A Power Primer. In *Psychological Bulletin*, Vol. 112, No. 1, July 1992, pp. 155—159.
- [Conradi 1998] Conradi, R.; Westfechtel, B.; Version Models for Software Configuration Management. In *ACM Computing Surveys*, Vol. 30, Issue 2, June 1998, pp. 232—282.
- [Cook 1979] Cook, T.D.; Campbell, D.T.; *Quasi-Experimentation – Design and Analysis Issues for Field Settings*. Houghton Mifflin, 1979.
- [Corbin 2008] Corbin, J.; Strauss, A.; *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. 3<sup>rd</sup> Edition. Sage Publications, 2008.
- [Cordy 2003] Cordy, J.R.; Comprehending Reality - Practical Barriers to Industrial Adoption of Software Maintenance Automation. In *Proceedings of the 11<sup>th</sup> IEEE International Workshop on Program Comprehension (WPC 2003)*, 2003.
- [Cordy 2011] Cordy, J.R.; Exploring Large-scale System Similarity Using Incremental Clone Detection and Live Scatterplots. In *Proceedings of the 19<sup>th</sup> International Conference on Program Comprehension (ICPC 2011)*, 2011, pp. 151—160.
- [D'Ambros 2008] D'Ambros, M.; Gall, C.; Lanza, M.; Pinzger, M.; Analysing Software Repositories to Understand Software Evolution. In: Mens, T. (Ed.); Demeyer, S. (Ed.), *Software Evolution*, Springer-Verlag, Berlin Heidelberg, 2008, pp. 37—68.



- [DeBaud 1998] DeBaud, J.M.; Girard, J.F.; The Relation between the Product Line Development Entry Points and Reengineering. In Proceedings of the 2<sup>nd</sup> International ESPRIT ARES Workshop on Development and Evolution of Software Architectures for Product Families, LNCS Volume 1429, Springer Verlag, 1998, pp. 132—139.
- [Demeyer 2008] Demeyer, S.; Ducasse, S.; Nierstrasz, O.; Object-oriented Reengineering Patterns. Square Bracket Associates, Switzerland, 2008.
- [de Wit 2009] de Wit, M.; Zaidman, A.; van Deursen, A.; Managing Code Clones using Dynamic Change Tracking and Resolution. In Proceedings of the International Conference on Software Maintenance (ICSM 2009), 2009, pp. 169—178.
- [Diffuse 2014] Diffuse – Graphical Tool for Merging and Comparing Text Files. Available under <http://diffuse.sourceforge.net/> (accessed on 4 May 2014).
- [Dubinsky 2013] Dubinsky, Y.; Rubin, J.; Berger, T.; Duszynski, S.; Becker, M.; Czarnecki, K.; An Exploratory Study of Cloning in Industrial Software Product Lines. In Proceedings of the 2013 17<sup>th</sup> European Conference on Software Maintenance and Reengineering (CSMR 2013), 2013, Genova, Italy, pp. 25—34, Best Paper Award.
- [Duszynski 2008] Duszynski, S.; Knodel, J.; Naab, M.; Hein, D.; Schitter, C.; Variant Comparison – A Technique for Visualizing Software Variants. In Proceedings of the 15<sup>th</sup> Working Conference on Reverse Engineering (WCRE 2008), 2008, Antwerp, Belgium, pp. 229—233.
- [Duszynski 2009] Duszynski, S.; Knodel, J.; Lindvall, M.; SAVE: Software Architecture Visualization and Evaluation. In Proceedings of the 13<sup>th</sup> European Conference on Software Maintenance and Reengineering (CSMR 2009), 2009, pp. 323—324.
- [Duszynski 2010a] Duszynski, S.; Visualizing and Analyzing Software Variability with Bar Diagrams and Occurrence Matrices. In Proceedings of the 14<sup>th</sup> International Software Product Line Conference (SPLC 2010), 2010, pp. 481—485.
- [Duszynski 2010b] Duszynski, S.; John, I.; Variability Introduction Strategies. Fraunhofer IESE Report 026.10/E, 2010, Kaiserslautern, Germany.
- [Duszynski 2011a] Duszynski, S.; Knodel, J.; Becker, M.; Analyzing the Source Code of Multiple Software Variants for Reuse Potential. In Proceedings of the 18<sup>th</sup> Working Conference on Reverse Engineering (WCRE 2011), 2011, Limerick, Ireland, pp. 303—307.
- [Duszynski 2011b] Duszynski, S.; A Scalable Goal-Oriented Approach to Software Variability Recovery. In Proceedings of the 15<sup>th</sup> International Software Product Line Conference (SPLC 2011), 2011, Vol. 2.



- [Duszyński 2012a] Duszyński, S.; Becker, M.; Recovering Variability Information from the Source Code of Similar Software Products. In Proceedings of the 3<sup>rd</sup> International Workshop on Product Line Approaches in Software Engineering (PLEASE 2012), 2012, pp. 37—40.
- [Duszyński 2012b] Duszyński, S.; Becker, M.; Kalmar, R.; Variantenanalyse – Wiederverwendungspotenzial auf Basis einer Quellcodeanalyse. In ATZ Elektronik Vol. 7 (2012), Issue 6, 2012, pp. 440—445.
- English variant: Variant Analysis – Reuse Potential Based on Source Code Analysis. ATZ Elektronik worldwide, Edition: 2012-06.
- [Eick 2002] Eick, S.; Graves, T.; Karr, A.; Mockus, A.; Schuster, P.; Visualizing Software Changes. In IEEE Transactions on Software Engineering, Vol. 28, No. 4, April 2002, pp. 396—412.
- [Ernst 2010] Ernst, N.A.; Easterbrook, S.M.; Mylopoulos, J.; Code Forking in Open-Source Software: A Requirements Perspective. eprint arXiv:1004.2889, 2010.
- [Faust 2003] Faust, D.; Verhoef, C.; Software Product Line Migration and Deployment. In Software: Practice and Experience, Vol. 33, Issue 10, August 2003, pp. 933—955.
- [Fluri 2007] Fluri, B.; Wursch, M.; Pinzger, M.; Gall, H.C.; Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. In IEEE Transactions on Software Engineering, Vol. 33, Issue 11, November 2007, pp. 725—743.
- [Fouts 2005] Fouts, D.E.; Mongodin, E.F.; Mandrell, R.E.; Miller, W.G.; Rasko, D.A.; Ravel, J.; Brinkac, L.M. et al.; Major Structural Differences and Novel Potential Virulence Mechanisms from the Genomes of Multiple *Campylobacter* Species. In PLOS Biology, Vol. 3, No. 1, 2005.
- [Fowler 1999] Fowler, M.; Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, 1999.
- [Gabel 2008] Gabel, M.; Jiang, L.; Su, Z.; Scalable Detection of Semantic Clones. In Proceedings of the 30<sup>th</sup> International Conference on Software Engineering (ICSE 2008), 2008, pp. 321—330.
- [Gacek 2001] Gacek, C.; Knauber, P.; Schmid, K.; Clements, P.; Successful Software Product Line Development in a Small Organization. A Case Study. Fraunhofer Institute for Experimental Software Engineering (IESE), Technical Report 013.01/E, 2001.
- [Ganesan 2006] Ganesan, D.; Muthig, D.; Yoshimura, K.; Predicting Return-on-Investment for Product Line Generations. In Proceedings of the 10<sup>th</sup> International Software Product Line Conference (SPLC 2006), 2006, pp. 13—24.

- [Gusfield 1993] Gusfield, D.; Efficient Methods for Multiple Sequence Alignment with Guaranteed Error Bounds. In *Bulletin of Mathematical Biology*, Vol. 55, Issue 1, January 1993, pp. 141—154.
- [Hall 1992] Hall, P.A.V.; *Software Reuse and Reverse Engineering in Practice*. Chapman & Hall, London, 1992.
- [Hemel 2012] Hemel, A.; Koschke, R.; Reverse Engineering Variability in Source Code Using Clone Detection: A Case Study for Linux Variants of Consumer Electronic Devices. In *Proceedings of the 19<sup>th</sup> Working Conference on Reverse Engineering (WCRE 2012)*, 2012, pp. 357—366.
- [Henninger 1994] Henninger, S.; Using Iterative Refinement to Find Reusable Software. In *IEEE Software*, Vol. 11, Issue 5, September 1994, pp. 48—59.
- [Hunt 1976] Hunt, J.W.; McIlroy, M.D.; An Algorithm for Differential File Comparison. *Computing Science Technical Report 41*, Bell Laboratories, June 1976.
- [IEEE 1998] IEEE Standard 1219-1998: IEEE Standard for Software Maintenance. IEEE Computer Society Press, Los Alamitos, CA, USA, 1998.
- [IEEE 2010] IEEE Standard 1517-2010: IEEE Standard for Information Technology – System and Software Life Cycle Processes – Reuse Processes. IEEE, New York, USA, 2010.
- [ISO/IEC 2011] ISO/IEC 9899:2011 Programming Languages – C (C11). ISO/IEC, 2011.
- [Jacobson 1997] Jacobson, I.; Griss, M.; Jonsson, P.; *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley Professional, 1997.
- [Jarzabek 1998] Jarzabek, S.; Wang, G.; Model-based Design of Reverse Engineering Tools. In *Journal of Software Maintenance: Research and Practice*, No. 10, 1998, pp. 353—380.
- [Jepsen 2007] Jepsen, H.P.; Dall, J.G.; Beuche, D.; Minimally Invasive Migration to Software Product Lines. In *Proceedings of the 11<sup>th</sup> International Software Product Line Conference (SPLC 2007)*, 2007, pp. 203—211.
- [JHotDraw 2014] JHotDraw – a Java GUI Framework for Technical and Structured Graphics. Available under <http://www.jhotdraw.org/> (accessed on 4 May 2014).
- [Ji 2007] Ji, X.; Mitchell, J.E.; Branch-and-Price-and-Cut on the Clique Partitioning Problem with Minimum Clique Size Requirement. In *Discrete Optimization*, Vol. 4, Issue 1, March 2007, pp. 87—102.
- [Jiang 2007] Jiang, Z.M.; Hassan, A.E.; A Framework for Studying Clones In Large Software Systems. In *Proceedings of the 7<sup>th</sup> IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, 2007, pp. 203—212.

- 
- [Juergens 2010] Juergens, E.; Deissenboeck, F.; Hummel, B.; Code Similarities Beyond Copy & Paste. In Proceedings of the 14<sup>th</sup> European Conference on Software Maintenance and Reengineering (CSMR 2010), 2010, pp. 78—87.
- [Kagdi 2007] Kagdi, H.; Collard, M.; Maletic, J.; A Survey and Taxonomy of Approaches for Mining Software Repositories in the Context of Software Evolution. In Journal of Software Maintenance and Evolution: Research and Practice, Vol. 19, Issue 2, March/April 2007, pp. 77—131.
- [Kamfonas 1992] Kamfonas, M.J.; Recursive Hierarchies – The Relational Taboo. In The Relational Journal, October-November 1992.
- [Kamiya 2002] Kamiya, T.; Kusumoto, S.; Inoue, K.; CCFinder: a Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code. In IEEE Transactions on Software Engineering, Vol. 28, No. 7, July 2002, pp. 654—670.
- [Kanda 2013] Kanda, T.; Ishio, T.; Inoue, K.; Extraction of Product Evolution Tree from Source Code of Product Variants. In Proceedings of the 17<sup>th</sup> International Software Product Line Conference (SPLC 2013), 2013, pp. 141—150.
- [Kang 2005] Kang, K.C.; Kim, M.; Lee, J.; Kim, B.; Feature-oriented Re-engineering of Legacy Systems into Product Line Assets – A Case Study. In Proceedings of the 9<sup>th</sup> International Software Product Line Conference (SPLC 2005), 2005.
- [Kapser 2006] Kapser, C.J.; Godfrey, M.W.; Supporting the Analysis of Clones in Software Systems: A Case Study. In Journal of Software Maintenance and Evolution: Research and Practice, Vol. 18, Issue 2, March-April 2006, pp. 61—82.
- [Kapser 2007] Kapser, C.J.; Anderson, P.; Godfrey, M.W.; Koschke, R.; Rieger, M.; van Rysselberghe, F.; Weißgerbe, P.; Subjectivity in Clone Judgment: Can We Ever Agree? In Duplication, Redundancy, and Similarity in Software, Dagstuhl Seminar Proceedings 06301, 2007.
- [Kapser 2008] Kapser, C.J.; Godfrey, M.W.; “Cloning Considered Harmful” Considered Harmful: Patterns of Cloning in Software. In Empirical Software Engineering, Vol. 13, Issue 6, December 2008, pp. 645—692.
- [Kästner 2010] Kästner, C.; Virtual Separation of Concerns: Toward Preprocessors 2.0. Dissertation, Otto-von-Guericke-Universität Magdeburg, Germany, 2010.
- [Kästner 2014] Kästner, C.; Dreiling, A.; Ostermann, K.; Variability Mining: Consistent Semi-automatic Detection of Product-Line Features. In IEEE Transactions on Software Engineering, Vol. 40, Issue 1, January 2014, pp. 67—82.

- [Kestler 2004] Kestler, H.A.; Müller, A.; Gress, T.M.; Buchholz, M.; Generalized Venn Diagrams: a New Method of Visualizing Complex Genetic Set Relations. In *Bioinformatics*, Vol. 21, Issue 8, pp. 1592—1595.
- [Khanna 2007] Khanna, S.; Kunal, K.; Pierce, B.; A Formal Investigation of Diff3. *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science*, LNCS Volume 4855, 2007, pp. 485—496.
- [Kiczales 1997] Kiczales, G.; Lamping, J.; Menhdhekar, A.; Maeda, C.; Lopes, C.; Loingtier, J.-M.; Irwin, J.; Aspect-oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP 1997)*, 1997, pp. 220—242.
- [Knodel 2005] Knodel, J.; John, I.; Ganesan, D.; Pinzger, M.; Usero, F.; Arciniegas, J.; Riva, C.; Asset Recovery and Their Incorporation into Product Lines. In *Proceedings of the 12<sup>th</sup> Working Conference on Reverse Engineering (WCRE 2005)*, 2005, pp. 120—129.
- [Knodel 2011] Knodel, J.; Sustainable Structures in Software Implementations by Live Compliance Checking. Dissertation, PhD Theses in Experimental Software Engineering, Vol. 35, Fraunhofer Verlag, Stuttgart, Germany, 2011.
- [Kolb 2006a] Kolb, R.; John, I.; Knodel, J.; Muthig, D.; Haury, U.; Meier, G.; Experiences with Product Line Development of Embedded Systems at Testo AG. In *Proceedings of the 10<sup>th</sup> International Software Product Line Conference (SPLC 2006)*, 2006, pp. 172—181.
- [Kolb 2006b] Kolb, R.; Muthig, D.; Patzke, T.; Yamauchi, K.; Refactoring a Legacy Component for Reuse in a Software Product Line: a Case Study. In *Journal of Software Maintenance and Evolution: Research and Practice*, Vol. 18, Issue 2, March-April 2006, pp. 109—132.
- [Kolb 2010] Kolb, R.; van der Linden, F.; Point/Counterpoint: The Need for Speed – Why Do We Do Product Lines? In *IEEE Software*, Vol. 27, No. 3, May-June 2010, pp. 56—59.
- [Koschke 2000] Koschke, R.; Atomic Architectural Component Recovery for Program Understanding and Evolution. PhD Thesis, University of Stuttgart, Germany, 2000.
- [Koschke 2008] Koschke, R.; Frontiers of Software Clone Management. *Frontiers of Software Maintenance (FoSM 2008)*, 2008, pp. 119—128.
- [Koschke 2009] Koschke, R.; Frenzel, P.; Breu, A.; Angstmann, K.; Extending the Reflexion Method for Consolidating Software Variants into Product Lines. *Software Quality Journal*, Vol. 17, Nr. 4, December 2009, pp. 331—366.

- 
- [Koskinen 2005] Koskinen, J.; Ahonen, J. J.; Sivula, H.; Tilus, T.; Lintinen, H.; Kankaanpää, I.; Software Modernization Decision Criteria: An Empirical Study. In Proceedings of the 9<sup>th</sup> European Conference on Software Maintenance and Reengineering (CSMR 2005), pp. 324—331.
- [Krueger 2002] Krueger, C.W.; Easing the Transition to Software Mass Customization. In Proceedings of the 4<sup>th</sup> International Workshop on Software Product Family Engineering, LNCS Volume 2290, Springer Verlag, 2002, pp. 282—293.
- [Krueger 2004] Krueger, C.W.; Towards a Taxonomy for Software Product Lines. In Proceedings of the 5<sup>th</sup> International Workshop on Software Product Family Engineering (PFE 2003), LNCS Volume 3014, 2004, pp. 323—331.
- [Lague 1997] Lague, B.; Proulx, D.; Mayrand, J.; Merlo, E.; Hudepohl, J.; Assessing the Benefits of Incorporating Function Clone Detection in a Development Process. In Proceedings of the International Conference on Software Maintenance (ICSM 1997), 1997, pp. 314—321.
- [Lanza 2006] Lanza, M.; Marinescu, R.; Object-Oriented Metrics in Practice. Springer Verlag, 2006.
- [Laudon 2006] Laudon, K.C.; Laudon, J.P.; Management Information Systems: Managing the Digital Firm. 9th edn., Pearson Prentice Hall, Upper Saddle River, NJ, 2006.
- [Lethbridge 2004] Lethbridge, T.; Tichelaar, S.; Ploedereder, E.; The Dagstuhl Middle Metamodel: A Schema For Reverse Engineering. In Electronic Notes in Theoretical Computer Science, Volume 94, May 2004, pp. 7—18.
- [Levenshtein 1966] Levenshtein, V.I.; Binary Codes Capable of Correcting Deletions, Insertions and Reversals. In Soviet Physics Doklady, Vol. 10, No. 8, February 1966, pp. 707—710.
- [Liew 2007] Liew, A.; Understanding Data, Information, Knowledge And Their Inter-Relationships. In Journal of Knowledge Management Practice, Vol. 8, No. 2, June 2007.
- [Lim 1998] Lim, W.C.; Managing Software Reuse. Pearson Prentice Hall, Upper Saddle River, NJ, 1998.
- [Madak-Erdogan 2013] Madak-Erdogan, Z.; Charn, T.H.; Jiang, Y.; Liu, E.T.; Katzenellenbogen, J.A.; Katzenellenbogen, B.S.; Integrative Genomics of Gene and Metabolic Regulation by Estrogen Receptors  $\alpha$  and  $\beta$ , and Their Coregulators. In Molecular Systems Biology, Vol. 9, No. 1, 2013.
- [Manning 2008] Manning, C.D.; Raghavan, P.; Schütze, H.; Introduction to Information Retrieval. Cambridge University Press, 2008.
- [Mapel 2004] Mapel, D.W.; Treatment Implications on Morbidity and Mortality in COPD. In Chest Journal, Vol. 126, No. 2\_suppl\_1, 2004, pp. 150—158.

- [McIlroy 1969] McIlroy, M.D.; Mass-Produced Software Components. In: Naur, P. (Ed.), Randell, B. (Ed.), *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, October 1968*, pp. 138—155, January 1969.
- [Mende 2008] Mende, T.; Beckwermert, F.; Koschke, R.; Meier, G.; Supporting the Grow-and-Prune Model in Software Product Lines Evolution Using Clone Detection. In *Proceedings of the 12<sup>th</sup> European Conference on Software Maintenance and Reengineering (CSMR 2008)*, pp. 163—172.
- [Mueller 2000] Müller, H.; Jahnke, J.; Smith, D.; Storey, M.A.; Tilley, S.; Wong, K.; Reverse Engineering: A Roadmap. In *Proceedings of the Conference on The Future of Software Engineering (ICSE 2000)*, Limerick, Ireland.
- [Murphy 2001] Murphy, G.; Notkin, D.; Sullivan, K.; Software Reflexion Models: Bridging the Gap between Design and Implementation. In *IEEE Transactions on Software Engineering*, Vol. 27, No. 4, April 2001.
- [Muthig 2002] Muthig, D.; A Light-weight Approach Facilitating an Evolutionary Transition Towards Software Product Lines. Dissertation, PhD Theses in Experimental Software Engineering, Vol. 11, Fraunhofer Verlag, Stuttgart, Germany, 2002.
- [Nguyen 2012] Nguyen, H.A.; Nguyen, T.T.; Pham, N.H.; Al-Kofahi, J.; Nguyen, T.N.; Clone Management for Evolving Software. In *IEEE Transactions on Software Engineering*, Vol. 38, No. 5, Sept.-Oct. 2012, pp. 1008—1026.
- [Noack 2011] Noack, A.M.; Vosko, L.F.; Precarious Jobs in Ontario: Mapping Dimensions of Labour Market Insecurity by Workers' Social Location and Context. Commissioned by the Law Commission of Ontario, November 2011.
- [Northrop 2004] Northrop, L.; Software Product Line Adoption Roadmap. Technical Report CMU/SEI-2004-TR-022, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA, 2004.
- [Parnas 1976] Parnas, D.L.; On the Design and Development of Program Families. In *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 1, March 1976.
- [Peterson 2004] Peterson, D.; Economics of Software Product Lines. In *Proceedings of the 5<sup>th</sup> International Workshop on Software Product-Family Engineering (PFE 2003)*, LNCS Volume 3014, 2004, pp. 381—402.
- [Ray 2012] Ray, B.; Kim, M.; A Case Study of Cross-System Porting in Forked Projects. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*, article no. 53.



- [Rieger 1999] Rieger, M.; Ducasse, S.; Golomingi, G.; Tool Support for Refactoring Duplicated OO Code. In Proceedings of the Workshop on Object-Oriented Technology (ECOOP '99), 1999. LNCS Volume 1743, Springer Verlag, 1999.
- [Riva 2003] Riva, C.; Del Rosso, C.; Experiences with Software Product Family Evolution. In Proceedings of the 6<sup>th</sup> International Workshop on Principles of Software Evolution (IWPSE '03), 2003.
- [Robles 2012] Robles, G.; Gonzalez-Barahona, J.M.; A Comprehensive Study of Software Forks: Dates, Reasons and Outcomes. In Proceedings of the 8<sup>th</sup> IFIP WG 2.13 International Conference on Open Source Systems, IFIP Advances in Information and Communication Technology Volume 378, 2012, pp. 1—14.
- [Rodgers 1988] Rodgers, J.L.; Nicewander, W.A.; Thirteen Ways to Look at the Correlation Coefficient. In The American Statistician, Vol. 42, No. 1, February 1988, pp. 59—66.
- [Rombach 2000] Rombach, D.; Fraunhofer: The German Model for Applied Research and Technology Transfer. In Proceedings of the 22<sup>nd</sup> International Conference on Software Engineering (ICSE '00), 2000, Limerick, Ireland, pp. 531—537.
- [Roy 2007] Roy, C.K.; Cordy, J.R.; A Survey on Software Clone Detection Research. Technical Report 2007-541, School of Computing, Queen's University, Canada, September 2007, 115 pp.
- [Roy 2009a] Roy, C.K.; Cordy, J.R.; Koschke, R.; Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. In Science of Computer Programming, Vol. 74, Issue 7, May 2009, pp. 470—495.
- [Roy 2009b] Roy, C.K.; Detection and Analysis of Near-Miss Software Clones. Dissertation, Queen's University, Kingston, Ontario, Canada, August 2009.
- [Rubin 2012] Rubin, J.; Chechik, M.; Locating Distinguishing Features using Diff Sets. In Proceedings of the 27<sup>th</sup> IEEE/ACM International Conference on Automated Software Engineering (ASE 2012), 2012, pp. 242—245.
- [Rubin 2013] Rubin, J.; Czarnecki, K.; Chechik, M.; Managing Cloned Variants: A Framework and Experience. In Proceedings of the 17<sup>th</sup> International Software Product Lines Conference (SPLC 2013), 2013.
- [Ruskey 2005] Ruskey, F.; Weston, M.; A Survey of Venn Diagrams. The Electronic Journal of Combinatorics, Dynamic Surveys, DS#5, June 2005, accessed on 4 May 2014, available under <http://www.combinatorics.org/ojs/index.php/eljc/article/view/DS5>.



- [Santini 1999] Santini, S.; Jain, R.; Similarity Measures. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 21, No. 9, September 1999, pp. 871—883.
- [Sauro 2009] Sauro, J.; Dumas, J.S.; Comparison of Three One-Question, Post-Task Usability Questionnaires. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '09)*, 2009, pp. 1599—1608.
- [Schmid 2002a] Schmid, K.; A Comprehensive Product Line Scoping Approach and its Validation. In *Proceedings of the 24<sup>th</sup> International Conference on Software Engineering (ICSE '02)*, 2002, pp. 593—603.
- [Schmid 2002b] Schmid, K.; Verlage, M.; The Economic Impact of Product Line Adoption and Evolution. In *IEEE Software*, Volume 19, Issue 4, July/August 2002, pp. 50—57.
- [Schmid 2005] Schmid, K.; John, I.; Kolb, R.; Meier, G.; Introducing the PuLSE Approach to an Embedded System Population at Testo AG. In *Proceedings of the 27<sup>th</sup> International Conference on Software Engineering (ICSE'05)*, 2005, pp. 544—552.
- [Schulze 2013] Schulze, S.; Analysis and Removal of Code Clones in Software Product Lines. Dissertation, Otto-von-Guericke-Universität Magdeburg, Germany, 2013.
- [Shannon 1948] Shannon, C.E.; A Mathematical Theory of Communication. In *The Bell System Technical Journal*, Volume 27, No. 3, July 1948, pp. 379—423.
- [Sherif 2003] Sherif, K.; Vinze, A.; Barriers to Adoption of Software Reuse: A Qualitative Study. In *Journal of Information and Management*, Vol. 41, Issue 2, December 2003, pp. 159—175.
- [Simon 2002] Simon, D.; Eisenbarth, T.; Evolutionary Introduction of Software Product Lines. In *Proceedings of the 2002 Software Product Lines Conference, LNCS Volume 2379*. Springer-Verlag, Berlin Heidelberg, pp. 272—283.
- [SPLC 2014] Software Product Line Conferences: Product Line Hall of Fame. Available under <http://splc.net/fame.html> (accessed on 4 May 2014).
- [Staples 2004] Staples, M.; Hill, D.; Experiences Adopting Software Product Line Development without a Product Line Architecture. In *Proceedings of the 11<sup>th</sup> Asia-Pacific Software Engineering Conference*, 2004, pp. 176—183.
- [Steger 2004] Steger, M.; Tischer, C.; Boss, B.; Müller, A.; Pertler, O.; Stolz, W.; Ferber, S.; Introducing PLA at Bosch Gasoline Systems: Experiences and Practices. In *Proceedings of the 2004 Software Product Lines Conference, LNCS Volume 3154*. Springer-Verlag, Berlin Heidelberg, pp. 34—50.

- [Svajlenko 2013] Svajlenko, J.; Roy, C.K.; Duszynski, S.; ForkSim: Generating Software Forks for Evaluating Cross-Project Similarity Analysis Tools. In Proceedings of the 13th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2013), 2013.
- [Tenev 2011] Tenev, V.; Nebel, M. (Sup.); Duszynski, S. (Sup.); Directed Coloured Multigraph Alignments for Variant Analysis of Software Systems. Bachelor Thesis, Fraunhofer IESE Report 112.11/E, 2011, Kaiserslautern, Germany.
- [Tenev 2012] Tenev, V.; Duszynski, S.; Applying Bioinformatics in the Analysis of Software Variants. In Proceedings of the 20<sup>th</sup> IEEE International Conference on Program Comprehension (ICPC 2012), 2012, Best Poster Paper Award.
- [Tenev 2013] Tenev, V.; Duszynski, S.; Similarity Normalization of Clone Detection Results for the Set Similarity Model. Fraunhofer IESE Report 029.13/E, 2013, Kaiserslautern, Germany.
- [Thomas 1997] Thomas, W.M.; Delis, A.; Basili, V.R.; An Analysis of Errors in a Reuse-Oriented Development Environment. In Journal of Systems and Software, Volume 38, Issue 3, September 1997, pp. 211—224.
- [Tichy 1984] Tichy, W. F.; The String-to-String Correction Problem with Block Moves. In ACM Transactions on Computer Systems (TOCS), Vol. 2, Issue 4, November 1984, pp. 309—321.
- [Tischer 2011] Tischer, C.; Müller, A.; Mandl, T.; Krause, R.; Experiences from a Large Scale Software Product Line Merger in the Automotive Domain. In Proceedings of the 15<sup>th</sup> International Software Product Lines Conference (SPLC 2011), 2011, pp. 267—276.
- [Tischer 2012] Tischer, C.; Boss, B.; Müller, A.; Thums, A.; Acharya, R.; Schmid, K.; Developing Long-Term Stable Product Line Architectures. In Proceedings of the 16<sup>th</sup> International Software Product Lines Conference (SPLC 2012), 2012, pp. 86—95.
- [Tomita 2006] Tomita, E.; Tanaka, A.; Takahashi, H.; The Worst-Case Time Complexity for Generating All Maximal Cliques and Computational Experiments. In Journal Theoretical Computer Science – Computing and Combinatorics, Vol. 363, Issue 1, October 2006, pp. 28—42.
- [Toomim 2004] Toomim, M.; Begel, A.; Graham, S.L.; Managing Duplicated Code with Linked Editing. In Proceedings of the 2004 IEEE International Symposium on Visual Languages and Human-Centric Computing (VLHCC '04), 2004, pp. 173—180.
- [Trochim 2006] Trochim, W.; Donnelly, J.P.; The Research Methods Knowledge Base. 3<sup>rd</sup> Edition, Atomic Dog Publishing, 2006.

- [Tseng 1986] Tseng, C.J.; Siewiorek, D.P.; Automated Synthesis of Data Paths in Digital Systems. In IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 5, Issue 3, July 1986, pp. 379—395.
- [Tseng 2007] Tseng, M.M.; Jiao, J.; Mass Customization. In: Salvendy, G. (Ed.), Handbook of Industrial Engineering: Technology and Operations Management. 3<sup>rd</sup> Edition, Wiley, New York, 2007, pp. 684—709.
- [Turban 2005] Turban, E.; Rainer, R.K.; Potter, R.E.; Introduction to Information Technology. 3<sup>rd</sup> Edition, Wiley, New York, 2005.
- [USCB 2014] United States Census Bureau. Survey and census data available under <http://www.census.gov/> (accessed on 4 May 2014).
- [Utter 2007] Utter, J.; Schaaf, D.; Mhurchu, C.N.; Scragg, R.; Food Choices Among Students Using the School Food Service in New Zealand. In Journal of the New Zealand Medical Association, Vol. 120, No. 1248, 2007.
- [Valiente 2001] Valiente, G.; An Efficient Bottom-Up Distance between Trees. In Proceedings of the 8<sup>th</sup> International Symposium of String Processing and Information Retrieval (SPIRE 2001), 2001, pp. 212—219.
- [van Emden 2002] van Emden, E.; Moonen, L.; Java Quality Assurance by Detecting Code Smells. In Proceedings of the 9th Working Conference on Reverse Engineering (WCRE 2002), 2002, Richmond, VA, USA, pp. 97—107.
- [Venn 1880] Venn, J.; On the Diagrammatic and Mechanical Representation of Propositions and Reasonings. In The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science, Year 10, No. 59, 1880, pp. 1—18.
- [Viegi 2004] Viegi, G.; Matteelli, G.; Angino, A.; Scognamiglio, A.; Baldacci, S.; Soriano, J.B.; Carrozzi, L.; The Proportional Venn Diagram of Obstructive Lung Disease in the Italian General Population. In Chest Journal, Vol. 126, No. 4, 2004, pp. 1093—1101.
- [Walline 2013] Walline, H.M.; Komarck, C.; McHugh, J.B.; Byrd, S.A.; Spector, M.E.; Hauff, S.J. et al.; High-Risk Human Papillomavirus Detection in Oropharyngeal, Nasopharyngeal, and Oral Cavity Cancers - Comparison of Multiple Methods. In JAMA Otolaryngology Head and Neck Surgery, Vol. 139, No. 12, 2013, pp. 1320—1327.
- [Willer 2013] Willer, C.J.; Schmidt, E.M.; Sengupta, S. et al.; Discovery and Refinement of Loci Associated with Lipid Levels. In Nature Genetics, Vol. 45, 2013, pp. 1274—1283.
- [Wleklik 2011] Wleklik, J.; Rombach, H.D. (Sup.); Duszynski, S. (Sup.); Analysis of Migration Strategies from Individual Development to Product Line Development on the Example of the Envisiontec Machine Control Software. Master Thesis, Fraunhofer IESE Report 113.11/E, 2011, Kaiserslautern, Germany.

- 
- [Wohlin 2000] Wohlin, C.; Runeson, P.; Höst, M.; Ohlsson, M.C.; Regnell, B.; Wesslen, A.; Experimentation in Software Engineering – An Introduction. Kluwer Academic Publishers, 2000.
- [Wu 2011] Wu, Y.; Yang, Y.; Peng, X.; Qiu, C.; Zhao, W.; Recovering Object-Oriented Framework for Software Product Line Reengineering. In Proceedings of the 12<sup>th</sup> International Conference on Software Reuse (ICSR 2011), LNCS Volume 6727, 2011, pp. 119—134.
- [Xing 2005] Xing, Z.; Stroulia, E.; UMLDiff: an Algorithm for Object-Oriented Design Differencing. In Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE '05), 2005, pp. 54—65.
- [Yamamoto 2005] Yamamoto, T.; Matsushita, M.; Kamiya, T.; Inoue, K.; Measuring Similarity of Large Software Systems based on Source Code Correspondence. In Proceedings of the 6<sup>th</sup> International Conference on Product Focused Software Process Improvement (PROFES 2005), Oulu, Finland, pp. 530—544.
- [Yoshimura 2006] Yoshimura, K.; Ganesan, D.; Muthig, D.; Assessing Merge Potential of Existing Engine Control Systems into a Product Line. In Proceedings of the 2006 International Workshop on Software Engineering for Automotive Systems (SEAS '06), 2006, pp. 61—67.
- [Zahra 2010] Zahra, F.; Rombach, H.D. (Sup.); Dodero, G. (Sup.); Duszynski, S. (Sup.); Correspondence Identification Techniques for Multiple Similar Software Systems. Master Thesis, Fraunhofer IESE Report 093.10/E, 2010, Kaiserslautern, Germany.
- [Zammit 2012] Zammit, A.R.; Starr, J.M.; Johnson, W.; Deary, I.J.; Profiles of Physical, Emotional and Psychosocial Wellbeing in the Lothian Birth Cohort 1936. In BMC Geriatrics, Vol. 12, 2012.
- [Zhang 2008] Zhang, Y.; Basit, H.A.; Jarzabek, S.; Anh, D.; Low, M.; Query-based Filtering and Graphical View Generation for Clone Analysis. In Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2008), 2008, pp. 376—385.
- [Zijlstra 1993] Zijlstra, F.; Efficiency in Work Behavior. A Design Approach for Modern Tools. Dissertation, Delft University of Technology, Delft, The Netherlands, 1993.



## Appendix A Experiment Material

The information and documents provided in this appendix can be used to analyze the details of the controlled experiment performed for this thesis and to replicate that experiment. For the description of experiment goals, hypotheses, and design please refer to Chapter 7.

### A.1 Experiment Infrastructure Setup

#### A.1.1 Analyzed Software Systems

The software systems analyzed by experiment participants were based on the source code of JHotDraw, an open-source middle-sized Java software system [JHotDraw 2014]. Five different variant systems were generated from the original JHotDraw code by the ForkSim tool [Svajlenko 2013] in a randomized generation process. In that process, syntactically correct code parts were injected at random, but syntactically correct locations into the original JHotDraw code, sometimes with and sometimes without repetitions across the different variants. The inserted code parts were in some cases modified for a subgroup of the target variants, for example by changing the code formatting, removing some code lines, renaming variables and other identifiers, or changing the assigned variable values. The insertion and modification operations were logged, so that the similarity of the generated system variants was exactly known. Thus, correct answers to the experimental questions were known without the need to use any of the analysis techniques investigated in the experiment.

All files which were not relevant for the experiment were deleted from the analyzed system variants in order to ease the code navigation and thus reduce the amount of time the subjects needed to spend on locating the respective files.

#### A.1.2 Adaptations to the Variant Analysis Tool

The experiment goal was not to compare any specific software tools, but rather to compare two techniques for analyzing code similarity. However, using software tools was necessary as they create and provide access to the similarity data. Although the common pairwise comparison tools and the Variant Analysis tool differ in several places, e.g. with regard to user interface and data visualization, all these differences had to be removed in the experiment in order to eliminate their influence on the experiment

result. Therefore, to isolate the effect of using a different similarity model, the tools used by both groups were made equal in all aspects – except the similarity model used. Depending on the similarity model, the similarity visualized in the tool was based either on pairwise comparison results or on the set-based comparison results. Apart from that difference, both tools used the same algorithm for pairwise comparison of code files, as well as the same code navigation and visualization means. This was achieved by providing adapted variants of the Variant Analysis tool to both groups. Moreover, advanced functionalities of the Variant Analysis tool, which could be used by the participants as an alternative way to obtain the needed results, were disabled.

For the experimental group working with the set similarity model, the Variant Analysis tool was adapted in the following way:

- The Variant Analysis views were disabled and hidden. The only interface elements the participants could interact with were the system structure diagram and the colored code editor.
- The similarity information was removed from the system structure diagram elements. Hence, the similarity information was only displayed in the code editor.
- The model analysis menus were disabled and hidden.
- The comparison result dialog, normally used for configuring and opening the code editor, was reduced to only contain the basic settings relevant for the experiment.

For the other group, working with pairwise similarity, the above modifications were applied too. The tool for that group was further adapted as follows:

- The comparison result dialog was modified to enable the user to specify a pair of file variants for analysis.
- The textual line information and the code coloring, displayed in the code editor, were adapted to reflect pairwise similarity results instead of the set-based results.

### **A.1.3 Computing Infrastructure**

Each participant received a workplace equipped with standard Fraunhofer IESE student workstation hardware. Hence, all students worked on identical or very similar hardware configurations. On each computer, the Eclipse environment containing the group-specific adaptation of the Variant Analysis tool was installed and started. The systems for analysis were stored on the hard disk.



The automated analysis phase, necessary to create the similarity information for the analysis, was not part of the experiment. Instead, the students received the results of an already completed analysis, and their only task was to answer similarity questions based on the presented information.

## **A.2 Experiment Documents**

In the course of the experiment, each participant received two printed documents:

- The tool tutorial slides (specific for the given group).
- Experiment document (identical for both groups), containing:
  - Introductory information
  - The briefing questionnaire
  - The description of the tasks to be solved
  - The debriefing questionnaire

These documents are stored in the following subsections.

## A.2.1 Tool Tutorials

The following tutorial slides were common for both experimental groups.

### The Similarity Analysis Experiment

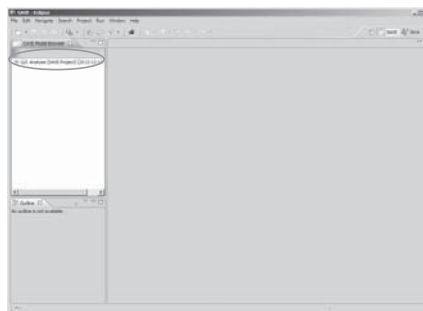
#### Tool Tutorial

1

#### Welcome to the Tool Tutorial!

This tutorial shows how to use the similarity analysis tool.

The analyzed software systems are already imported into the tool.

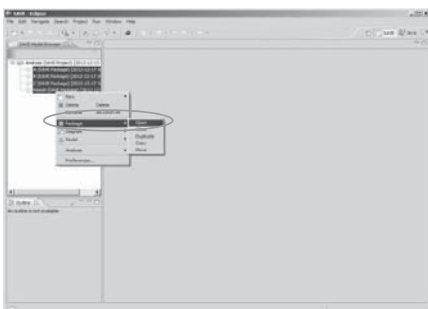


- To start, please expand the Analysis project tree

2

#### Basics: Loading Packages

There are three systems in the analysis: A, B and C.  
The analysis results are stored in the Result package.  
You need to load the packages in order to access the analysis results.

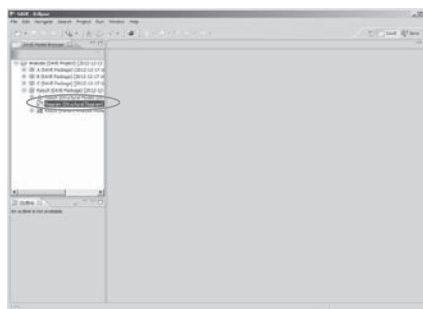


- Select all packages
- Open the right-click menu
- Select the Package -> Open menu item
- The package icons change color: the packages are loaded now

3

#### Basics: Opening the Diagram

The analyzed files will be displayed on a diagram. Please open the diagram to see the files.



- Open the „Result“ package
- Double-click on the diagram to open it

4

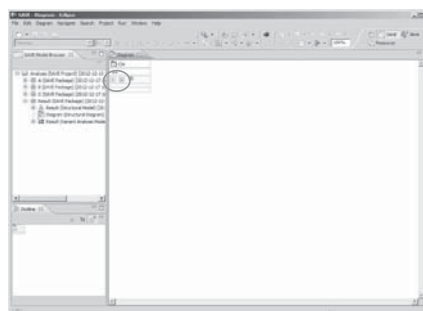
### Navigating the Diagram

5

#### Navigating the Diagram (1): Expand

The diagram visualizes the directory structure of the analyzed systems.

You can expand and collapse the directories to see the files inside.

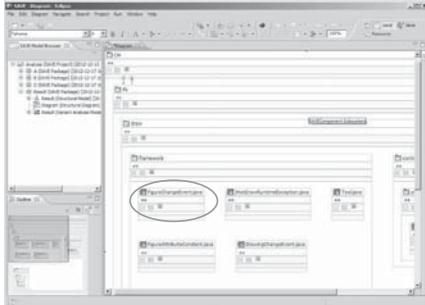


- The two plus icons on the CH component mean „Expand“ and „Expand all“
- Please click „Expand all“ to see the whole directory structure

6

### Navigating the Diagram (2): Folders and Files

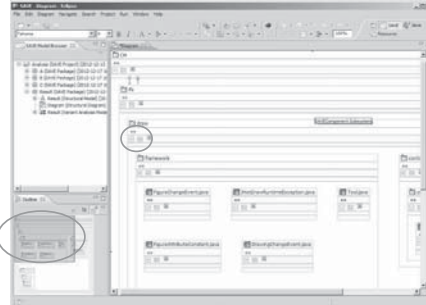
The files are shown as yellow rectangles, contained in the respective parent folders.  
For example, the file in the circle is *CH/ifa/draw/framework/FigureChangeEvent.java*



7

### Navigating the Diagram (3): Scrolling and Outline

You can scroll through the diagram using the scroll bars or the Outline view (bottom left).



8

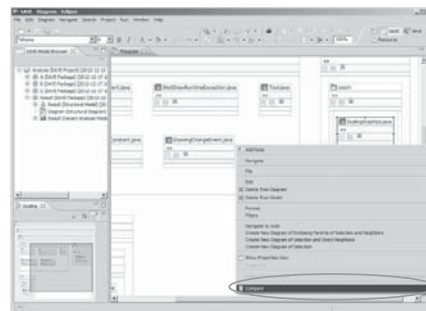
- You can also collapse some of the folders by using one of the minus icons „Collapse“ and „Collapse all“

### Analyzing the Similarity

9

### Analyzing the Similarity (1): Compare Menu

Open the code similarity results by right-clicking on a file and selecting the „Compare“ menu item. Let's use the file *CH/ifa/draw/contrib/zoom/ScalingGraphics.java* as an example.



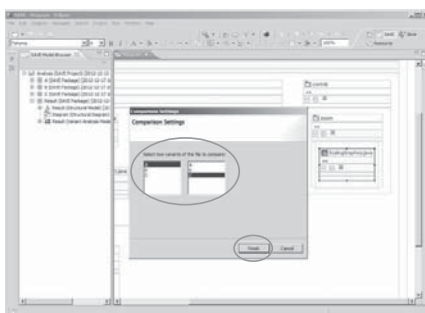
10

- Please find the file *ScalingGraphics*
- Right-click on it and select „Compare“

The following two slides were used only in the pairwise similarity group.

### Analyzing the Similarity (2): Wizard

A wizard appears. Please select two of the existing file variants to be compared.

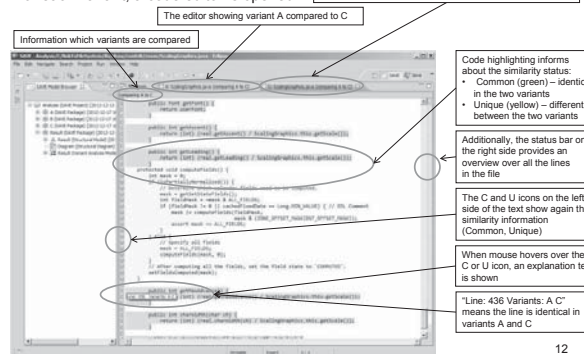


11

- Select two file variants in the lists (for example A and C)
- Click the „Finish“ button to open the selected file variants

### Analyzing the similarity (3): Editor

For each variant, a code editor is opened.



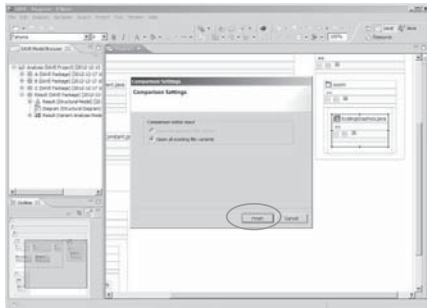
12

- The editor showing variant C of the *ScalingGraphics.java* file, compared to its variant A
- Information which variants are compared
- Code highlighting informs about the similarity status:
  - Common (green) – identical in the two variants
  - Unique (yellow) – different between the two variants
- Additionally, the status bar on the right side provides an overview over all the lines in the file
- The C and U icons on the left side of the text show again the similarity information (Common, Unique)
- When mouse hovers over the C or U icon, an explanation text is shown
- „Line: 436 Variants: A C“ means the line is identical in variants A and C

The following four slides were used only in the set model similarity group.

### Analyzing the Similarity (2): Wizard

A wizard appears, informing that all existing file variants will be opened.

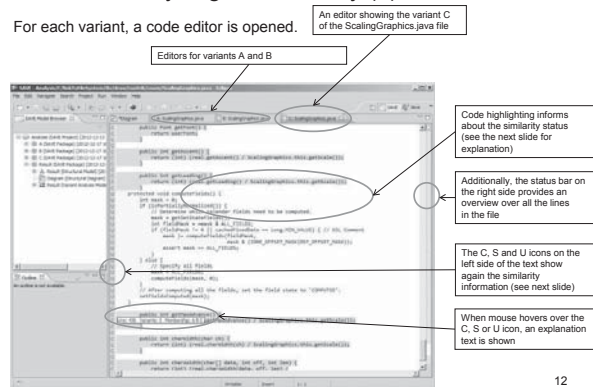


- Click the „Finish“ button to open the file variants

11

### Analyzing the similarity (3): Editor

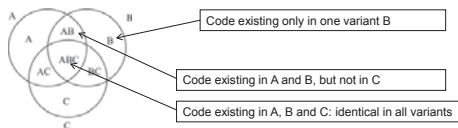
For each variant, a code editor is opened.



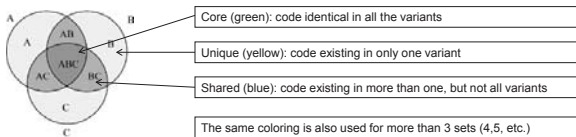
12

### The Set Similarity Model (1)

The information in the editor is based on a set similarity model.



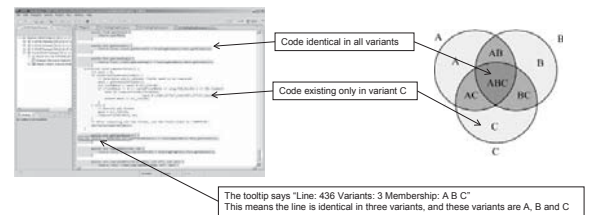
The code is colored according to the set information



13

### The Set Similarity Model (2)

This is how the editor visualizes the set model for the code:



The same type of information is shown in the editors for the variant A and B

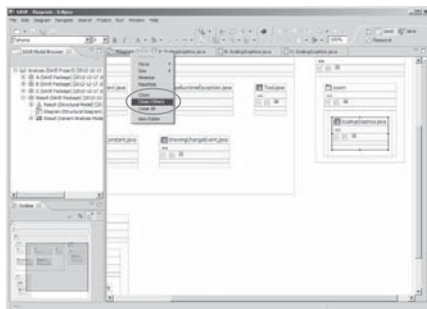
Of course, the code existing only in variant A will only be shown in the editor A

14

The remaining tutorial slides were common for both experimental groups, except for their slide numbers which differed due to a different number of preceding slides.

### After Analysis: Close the Files

When you finish analyzing a group of files, you can clean up your tool by closing all the editors and leaving only the diagram open.



- You can close each editor's window individually
- Or, you can go to the diagram tab, right-click on it, and select the „Close others“ option

13

### Example Tasks

14

<h3 style="text-align: center;">Example Tasks</h3> <p>Please try to determine the correct answers to the following example tasks. The next slide contains the answers, but try to solve the example tasks first! In case you have any question about the tool or the tasks, please ask it now!</p> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr style="background-color: #f2f2f2;"> <th style="text-align: left; padding: 5px;">Example Question</th> <th style="text-align: left; padding: 5px;">Answer</th> </tr> </thead> <tbody> <tr> <td style="padding: 5px;"> <b>QEx1</b> Which two variants of the file <code>CH\ifa\draw\samples\Animator.java</code> are the most similar to each other? </td> <td style="padding: 5px; text-align: right;"> <input type="checkbox"/> I don't know </td> </tr> <tr> <td style="padding: 5px;"> <b>QEx2</b> Which variants of the file <code>CH\ifa\draw\framework\Tool.java</code> have identical code? </td> <td style="padding: 5px; text-align: right;"> <input type="checkbox"/> I don't know </td> </tr> <tr> <td style="padding: 5px;"> <b>QEx3</b> Which variant of the file <code>CH\ifa\draw\framework\FigureChangeEvent.java</code> contains <b>only</b> code which also exists in all other variants? </td> <td style="padding: 5px; text-align: right;"> <input type="checkbox"/> I don't know </td> </tr> </tbody> </table> <p style="text-align: right;">15</p>	Example Question	Answer	<b>QEx1</b> Which two variants of the file <code>CH\ifa\draw\samples\Animator.java</code> are the most similar to each other?	<input type="checkbox"/> I don't know	<b>QEx2</b> Which variants of the file <code>CH\ifa\draw\framework\Tool.java</code> have identical code?	<input type="checkbox"/> I don't know	<b>QEx3</b> Which variant of the file <code>CH\ifa\draw\framework\FigureChangeEvent.java</code> contains <b>only</b> code which also exists in all other variants?	<input type="checkbox"/> I don't know	<p>(intentionally left blank)</p> <p style="text-align: right;">16</p>
Example Question	Answer								
<b>QEx1</b> Which two variants of the file <code>CH\ifa\draw\samples\Animator.java</code> are the most similar to each other?	<input type="checkbox"/> I don't know								
<b>QEx2</b> Which variants of the file <code>CH\ifa\draw\framework\Tool.java</code> have identical code?	<input type="checkbox"/> I don't know								
<b>QEx3</b> Which variant of the file <code>CH\ifa\draw\framework\FigureChangeEvent.java</code> contains <b>only</b> code which also exists in all other variants?	<input type="checkbox"/> I don't know								

The number of blank slides, placed after the example tasks slide, varied between the two tutorial variants. The reason for that was that in both variants the example task answers were placed on slide 19 to prevent the participants from seeing the answers prematurely.

<h3 style="text-align: center;">Example Tasks – Answers</h3> <p>In case you have any question about the tool or the tasks, please ask it now!</p> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr style="background-color: #f2f2f2;"> <th style="text-align: left; padding: 5px;">Example Question</th> <th style="text-align: left; padding: 5px;">Answer</th> </tr> </thead> <tbody> <tr> <td style="padding: 5px;"> <b>QEx1</b> Which two variants of the file <code>CH\ifa\draw\samples\Animator.java</code> are the most similar to each other? </td> <td style="padding: 5px;"> A, B <input type="checkbox"/> I don't know </td> </tr> <tr> <td style="padding: 5px;"> <b>QEx2</b> Which variants of the file <code>CH\ifa\draw\framework\Tool.java</code> have identical code? </td> <td style="padding: 5px;"> A, B, C <input type="checkbox"/> I don't know </td> </tr> <tr> <td style="padding: 5px;"> <b>QEx3</b> Which variant of the file <code>CH\ifa\draw\framework\FigureChangeEvent.java</code> contains <b>only</b> code which also exists in all other variants? </td> <td style="padding: 5px;"> B <input type="checkbox"/> I don't know </td> </tr> </tbody> </table> <p style="text-align: right;">19</p>	Example Question	Answer	<b>QEx1</b> Which two variants of the file <code>CH\ifa\draw\samples\Animator.java</code> are the most similar to each other?	A, B <input type="checkbox"/> I don't know	<b>QEx2</b> Which variants of the file <code>CH\ifa\draw\framework\Tool.java</code> have identical code?	A, B, C <input type="checkbox"/> I don't know	<b>QEx3</b> Which variant of the file <code>CH\ifa\draw\framework\FigureChangeEvent.java</code> contains <b>only</b> code which also exists in all other variants?	B <input type="checkbox"/> I don't know
Example Question	Answer							
<b>QEx1</b> Which two variants of the file <code>CH\ifa\draw\samples\Animator.java</code> are the most similar to each other?	A, B <input type="checkbox"/> I don't know							
<b>QEx2</b> Which variants of the file <code>CH\ifa\draw\framework\Tool.java</code> have identical code?	A, B, C <input type="checkbox"/> I don't know							
<b>QEx3</b> Which variant of the file <code>CH\ifa\draw\framework\FigureChangeEvent.java</code> contains <b>only</b> code which also exists in all other variants?	B <input type="checkbox"/> I don't know							

## A.2.2 The Main Experiment Document

The main experiment document was identical for both groups.

The questions asked in the document, except for the analysis task questions, were not numbered in any way visible to the participants. However, we numbered all questions internally in order to refer to them in other documents such as result tables. Below, we indicate the respective question numbers by using notes placed on the left document margin. The same question numbers are used in Appendix A.3, where the participant answers are reported.

.11.01.2013,

## **The Similarity Analysis Experiment**

Please read the experiment documents carefully. In case you have any problems in understanding the given information, the experiment tasks or the tool usage, please contact the experiment supervisor.

The experiment is scheduled to last about 90 minutes.

Thank you for participating in the experiment, and good luck!

### **Experiment Goal**

The goal of the experiment is to evaluate two methods which are used for analyzing the similarity of software system implementation. The time you spend on solving the analysis tasks and the correctness of your answers are measured. Please try to solve the tasks quickly, but also make sure you solve them all correctly, as correctness is important here.

Please note that the experiment is evaluated anonymously. The goal is not to evaluate your personal performance, but to evaluate the similarity analysis methods.

### **Your Role in the Experiment**

In the experiment, you take the role of a software architect who investigates a few variants of a software product for similarity – for example, to determine the possibility of transforming the variants to a software product line. The knowledge of software architecture or software product lines concepts is not required for the experiment. Because the experiment has a limited time, only a few selected locations in the software will be investigated.

### **Definition of Code Similarity used in the Experiment**

The similarity of two source files is defined as the similarity of their source code:

- two files where only a few code lines differ have a high similarity,
- two files where a lot of code lines differ have a low similarity.

In the experiment, **the whole content of the Java file**, including comments and import statements, is treated as source code. Therefore, differences in imports and comments affect the similarity in the same way as the differences in Java code statements.

.11.01.2013,

### **The Experiment Procedure**

The analysis tool and the investigated systems are already installed on your computer. Additionally, the tool documentation is attached to this document. Before the analysis starts, you will receive a short training on the tool usage. During the experiment, you will use the tool to analyze the provided systems and answer the provided questions.

#### **Preparation Phase**

- Please read the experiment description (Page 1 and 2).
- Please fill out the Briefing Questionnaire (Page 3).
- Read the task solution guidance (Page 4), and wait for the tool tutorial.
- Listen to the tool tutorial and perform the presented tool usage steps.
- Try out the tool on the example workspace. Make sure you understand how to use the tool.
- When you are ready to start, tell it to the experiment supervisor. The execution phase will start when all group members are ready.

#### **Execution Phase**

- All group members receive the name of the experiment workspace.
- Please switch Eclipse to the experiment workspace. Make sure that the system packages and the diagram are opened.
- Write down the execution start time on Page 4.
- Conduct the experiment tasks (Pages 5 to 6).
- When you finish all the tasks, write down the execution stop time on Page 4.

#### **Finalization Phase**

- Please fill out the Debriefing Questionnaire (Pages 7 and 8).



.11.01.2013,

### Briefing Questionnaire

The purpose of this questionnaire is to characterize the background of the participants. The information you provide will help in the analysis of experiment results. The answers will be treated anonymously.

Please answer the questions as complete and as honest as possible. Try to answer every question – if you are not sure about the answer, just select the one you feel is the most likely. Thank you for your support!

#### Background information

- What is your major field of study? ☐ Computer Science  
☐ Business Informatics  
☐ Mathematics  
☐ Other, please specify \_\_\_\_\_
- What is the degree you are currently studying for? ☐ Bachelor  
☐ Master  
☐ Other, please specify \_\_\_\_\_

- In which semester of your study are you at the moment (counting from the beginning of the bachelor studies)? \_\_\_\_\_

- Are you color blind? ☐ Yes (which colors?) \_\_\_\_\_  
☐ No

- Please rate your experience in the following categories. For each category, please select only one option.

How much experience do you have in ...	No experience 1	Little experience 2	Medium experience 3	Significant experience 4	Professional experience 5
Programming in general	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
The Java programming language	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Using the Eclipse environment	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Comparing source code using diff tools (any kind)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Comparing source code using the Eclipse Diff tool	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Using the Variant Analysis tool	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

	Highly unmotivated 1	Unmotivated 2	Neither motivated nor unmotivated 3	Motivated 4	Highly motivated 5
How motivated are you to perform well in the experiment?	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

.11.01.2013,

### Task Solution Guidance

You take the role of a software architect who investigates a few variants of a software product for similarity. Please use the specified tool installed on your computer to analyze the five variants of the JHotDraw system, named A, B, C, D and E. During the analysis, please answer the questions below. In case you are unable to determine the correct answer, please instead mark the “☐ I don't know” field. If there is no answer at all, the “I don't know” explanation will be assumed, too. The questions answered with “I don't know” will be counted as answered incorrectly.

When answering the questions, it is enough that you write down the variant names or the number being the correct answer. “A and B” or “5” are sufficient, for example:

Example Question	Example Answer
<b>QEx1</b> Which two variants of the file <i>package/File.java</i> are the <b>most similar</b> to each other?	A, B <input type="checkbox"/> I don't know
<b>QEx2</b> Which methods in the file <i>package/test/Test.java</i> <b>exist only in variant A</b> of that file?	runOnce() <input type="checkbox"/> I don't know

Please remember to only use the provided tool to analyze the similarity. Do not use other tools such as Notepad or Excel. If you need to take down notes, please use the empty sheet of paper attached at the end of this document.

#### The Experiment

Please notify the experiment supervisor when you are ready to start. The execution phase will start when all group members are ready.

After receiving the name of the experiment workspace, please switch Eclipse to that workspace. Make sure that the system packages and the diagram are opened.

Then, write down the start time and turn the page to start working on the tasks. Please try to solve the tasks quickly, but also make sure you solve them all correctly, as correctness is important here.

Write down the execution **start time** here (e.g., 15:50) \_\_\_\_\_

Write down the execution **stop time** here (e.g., 16:50) \_\_\_\_\_

.11.01.2013,

## Tasks 1 – 12

Question	Answer
Q1. Which two variants of the file <i>CH/ifa/draw/contrib/CommandMenuItem.java</i> are <b>the most similar</b> to each other?	<input type="checkbox"/> I don't know
Q2. Which two variants of the file <i>CH/ifa/draw/contrib/CustomSelectionTool.java</i> are <b>the most similar</b> to each other?	<input type="checkbox"/> I don't know
Q3. Which variants of the file <i>CH/ifa/draw/contrib/Helper.java</i> have <b>identical</b> code?	<input type="checkbox"/> I don't know
Q4. Which variants of the file <i>CH/ifa/draw/contrib/AutoscrollHelper.java</i> have <b>identical</b> code?	<input type="checkbox"/> I don't know
Q5. Which variant of the file <i>CH/ifa/draw/util/CommandButton.java</i> is <b>strongly dissimilar</b> from the others?	<input type="checkbox"/> I don't know
Q6. Which variant of the file <i>CH/ifa/draw/util/UndoableAdapter.java</i> is <b>strongly dissimilar</b> from the others?	<input type="checkbox"/> I don't know
Q7. Which variants of the file <i>CH/ifa/draw/util/UndoableTool.java</i> have <b>identical</b> code?	<input type="checkbox"/> I don't know
Q8. Which variant of the file <i>CH/ifa/draw/standard/BoxHandleKit.java</i> contains the most <b>unique code</b> (code which <b>doesn't exist in any other variant</b> )?	<input type="checkbox"/> I don't know
Q9. Which variant of the file <i>CH/ifa/draw/standard/HandleTracker.java</i> contains <b>only</b> code which also <b>exists in all other variants</b> ?	<input type="checkbox"/> I don't know
Q10. Which variant of the file <i>CH/ifa/draw/samples/minimap/MiniMapDesktop.java</i> is <b>strongly dissimilar</b> from the others?	<input type="checkbox"/> I don't know
Q11. Which variant of the file <i>CH/ifa/draw/application/DrawApplication.java</i> contains the most <b>unique code</b> (code which <b>doesn't exist in any other variant</b> )?	<input type="checkbox"/> I don't know
Q12. Which variants of the file <i>CH/ifa/draw/contrib/SVGDrawApp.java</i> have <b>identical</b> code?	<input type="checkbox"/> I don't know

(continued on the next page)

.11.01.2013,

**Tasks 13 – 16**

	Question	Answer
<b>Q13.</b>	Which variant of the file <i>CH/ifa/draw/standard/FigureChangeEventMulticaster.java</i> contains the most <b>unique code</b> (code which <b>doesn't exist in any other variant</b> )?	<input type="checkbox"/> I don't know
<b>Q14.</b>	Which methods in the file <i>CH/ifa/draw/figures/FontSizeHandle.java</i> from variant C <b>exist only in that variant</b> of the file?	<input type="checkbox"/> I don't know
<b>Q15.</b>	Which two variants of the file <i>CH/ifa/draw/figures/EllipseFigure.java</i> are the <b>most similar</b> to each other?	<input type="checkbox"/> I don't know
<b>Q16.</b>	Which two variants of the file <i>CH/ifa/draw/util/StandardVersionControlStrategy.java</i> are the <b>most similar</b> to each other?	<input type="checkbox"/> I don't know

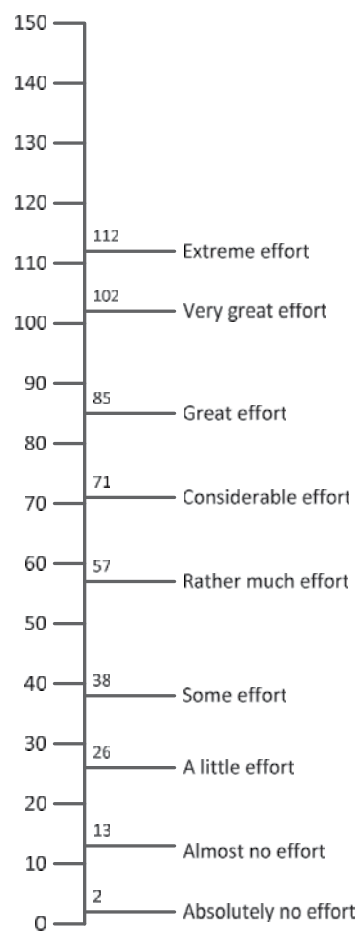
Please remember to write down the stop time on Page 4 when you finish!

.11.01.2013,

**Debriefing Questionnaire**

How **difficult** did you perceive the tasks? How much effort did it require to solve them?

Please draw a horizontal line or mark a cross on the following vertical scale. Select a location on the scale that characterizes in the best way how you perceived the task difficulty. Then, write down the number that corresponds to the location you selected. You can select any number between 0 and 150.



Task difficulty: \_\_\_\_\_

(continued on the next page)

.11.01.2013,

**Debriefing Questionnaire (continued)**

Please answer the following questions about the conducted experiment. For each question, please select only one option.

To what degree do you agree with the following statements?	Strongly disagree	Disagree	Neither agree nor disagree	Agree	Strongly agree
	1	2	3	4	5
DB2 I understood the description of <b>the experiment tasks</b> .	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
DB3 I understood how I should use <b>the analysis tool</b> to receive the source code difference information.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
DB4 I understood the meaning of <b>the source code difference information</b> presented to me.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
DB5 I could easily see <b>the difference between the code highlighting colors</b> in the code editor.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
DB6 I had <b>enough time</b> for solving the tasks.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
DB7 I only used <b>the specified tool</b> for solving the tasks.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
DB8 I think my <b>answers</b> were correct.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
DB9 I think the tool I used provides a good support for <b>solving the tasks quickly</b> .	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
DB10 I think the tool I used provides a good support for <b>solving the tasks correctly</b> .	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
DB11 - What would you change to <b>improve</b> the experiment, the task description, or the tool you used?	<input type="text"/> <input type="text"/> <input type="text"/> <input type="text"/>				
DB12 - Do you have any additional <b>comments</b> or suggestions?	<input type="text"/> <input type="text"/> <input type="text"/> <input type="text"/>				

Thank you for participating in the experiment!

.11.01.2013,

**Notes**



.11.01.2013,

**Notes**

### A.3 Raw Experiment Results

The following tables report the raw results collected in the experiment. The participants are sorted in the tables according to the order in which they returned the filled experiment material in their groups, except for participants P10 and P13, for whom that order could not be determined. The participants identified as *S {number}* were assigned to the set model group, and the participants identified as *P {number}* were assigned to the pairwise comparison group.

In the initial result analysis, some participant answers were recognized as implausible or incomplete. These answers are marked in the result tables using gray cell background. For each table, the procedure applied to these answers in the later analysis, as well as the answer abbreviations used in the table, are explained.

The results of participant P4 were excluded from the experiment for reasons discussed in Chapter 7. These results are retained here for completeness, and are also marked using gray cell background.

#### A.3.1 Briefing Questionnaire Results

Table 27 presents the participant answers given to the briefing questionnaire. The abbreviations used in the table are: CS – Computer Science. Soft. Eng. – Software Engineering. M – Master. B – Bachelor.

**Implausible answers:** when answering questions B2 and B3, the participants indicated their study level (bachelor, master) and their current study semester, counted from the beginning of the bachelor studies. However, four participants indicated that although they were already in master studies, they currently studied in the 1<sup>st</sup> or 3<sup>rd</sup> semester. This is not possible, as the Technical University of Kaiserslautern only offers master studies to persons who hold the bachelor degree (and hence already studied for about 8 semesters). These persons were therefore either in bachelor studies at the moment, or they indicated by mistake their current semester of the master study.

As the Product Line course, attended by all the participants, is an advanced course offered mainly to master-level students, we assumed that the marked participants indicated the current semester of their master studies alone. Hence, in the further result processing, we added 8 semesters to each of their answers to estimate the duration of their bachelor studies.

Answering the question B10, participant S14 indicated he/she had “little experience” with using the Variant Analysis tool. This is implausible as, according to our knowledge, none of the participants could have seen the tool before. As we had no possibility to ask the participant for the reasons of that answer, we assume it was given due to a misunderstanding.

Participant ID	B1	B2	B3	B4	B4a	B5	B6	B7	B8	B9	B10	B11
S4	CS	M	10	No		3	3	3	1	1	1	4
S6	CS	M	11	No		5	5	5	2	1	1	4
S7	CS	M	9	No		3	2	2	2	2	1	4
S15	Soft. Eng.	M	3	No		5	3	2	2	1	1	4
S14	CS	M	9	No		4	3	2	2	1	2	3
S8	CS	M	9	No		4	4	3	1	1	1	5
S12	CS	B	7	No		2	2	2	1	1	1	3
S10	CS	M	11	Yes	Green-red	5	4	4	3	3	1	3
S13	Business	Diplo ma	13	No		2	2	1	1	1	1	3
S9	CS	B	8	No		2	3	3	1	1	1	3
S16	Soft. Eng.	M	3	Yes	Green and red, brown	3	3	3	1	1	1	3
P1	CS	M	9	No		5	3	3	3	2	1	4
P16	CS	M	8	No		4	4	4	3	2	1	4
P15	CS	B	8	No		4	3	3	1	1	1	3
P11	CS	M	9	No		4	1	2	4	1	1	4
P12	Soft. Eng.	M	10	No		3	4	4	4	3	1	3
P6	CS	M	1	No		3	3	2	2	1	1	5
P9	CS	M	10	No		3	2	3	1	1	1	4
P3	CS	M	1	No		3	3	3	2	1	1	4
P5	CS	B	1	No		3	3	3	3	1	1	4
P4	CS	M	2	No		3	3	3	1	1	1	4
P10	CS	M	14	No		1	4	3	3	2	1	3
P13	CS	M	12	No		4	3	3	3	2	1	3

Table 27 The experiment results: the briefing questionnaire

### A.3.2 Task Answers and Time Measurement

Participant ID	T1	T2	T [T2-T1]	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8
S4	16:17	16:28	11	BE	BC	ACDE	AD	A	D	DE	D
S6	16:17	16:31	14	BE	BC	ACDE	AD	A	D	DE	D
S7	16:17	16:32	15	BE	BC	ACDE	AD	A	D	DE	D
S15	16:17	16:30	13	BE	BC	ACDE	AD	A	D	DE	D
S14	16:17	16:28	11	BE	BC	ACDE	AD	A	D	DE	D
S8	16:18	16:31	13	BE	BC	ACDE	AD	A	D	DE	D
S12	16:17	16:32	15	BE	BC	ACDE	AD	A	D	DE	D
S10	16:17	16:32	15	BE	BC	ACDE	AD	A	D	DE	D
S13	16:15	16:33	18	BE	BC	ACDE	AD	A	D	DE	D
S9	16:17	16:29	12	BE	BC	ADE	AD	A	D	DE	D
S16	16:19	16:36	17	BE	BC	ACDE	AD	A	D	DE	D
P1	16:13	16:40	27	BE	BC	ACDE	AD	A	D	DE	D
P16	16:15	16:46	31	BE	BC	ACDE	AD	A	D	DE	D
P15	16:15	16:46	31	BE	BC	ACDE	AD	A	D	DE	D
P11	16:15	16:40	25	BE	BC	ACDE	AD	A	D	DE	D
P12	16:18	16:43	25	BE	BC	AC	AD	A	D	DE	D
P6	16:21	17:01	40	BE	BC	ACDE	AD	A	D	DE	D
P9	16:17	16:53	36		BC	ACDE	AD	A	D	DE	
P3	16:13			BE	X	ACDE	AD	A	D	DE	D
P5	16:14	16:59	45	BE	BC	ACDE	AD	A	D	DE	X
P4	16:00	17:03	63	BE, CE	BC	AC, AD, AE, CD, DE	AD	AB, AC, AE	BD	DE	BD, X
P10	16:19	17:03	44	BE	BC	ACDE	AD	A	D	DE	X
P13	16:14	16:47	33	BE	AE	ACDE	AD	A	D	DE	D

Table 28 presents the participant answers provided for the experiment tasks and the time measurements. The abbreviations used in the table are: X – the “I don’t know” answer was selected. Empty cell – no answer was given.

**Incomplete or implausible answers:** participant P3 did not specify the completion time for his/her tasks. Hence, the time measurement for that participant could not be performed, and this time result could not be considered in the further analysis. However, according to the group supervisor, the participant P3 was neither the slowest, nor the fastest person in the group.

Participant P4 stated that he/she measured the time difference accurately, but did not record the actual starting time, and hence he/she provided artificial results with correct time difference. As participant P4 was the last to finish the tasks in that group, the stated time difference is considered plausible.

Participant ID	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Correct [Q1...Q16]	Errors [Q1...Q16]
S4	C	C	D	BDE	D	paramString()	AD	BC	16	0
S6	C	C	D	BDE	D	paramString()	AD	BC	16	0
S7	C	C	D	BDE	D	paramString()	AD	BC	16	0
S15	C	C	D	BDE	D	paramString()	AD	BC	16	0
S14	C	C	D	BDE	D	paramString()	AD	BC	16	0
S8	C	C	D	BDE	D	paramString()	AD	BC	16	0
S12	C	C	D	BDE	D	paramString()	AD	BC	16	0
S10	C	C	D	BDE	D	paramString()	AD	BC	16	0
S13	C	C	D	BDE	D	paramString()	AD	BC	16	0
S9	C	C	D	BD	D	paramString()	AD	BC	14	2
S16	C	C	D	BDE	D	paramString()	AD	BC	16	0
P1	C	C	D	BDE	D	paramString()	AD	BC	16	0
P16	C	C	D	BDE	D	paramString()	AD	BC	16	0
P15	C	C		BDE	D	paramString()	AD	BE	14	2
P11	C	C	D	BDE	D	paramString()	AD	BC	16	0
P12	C	C	D	BD	D	paramString()	AD	BC	14	2
P6	E	C	D	BDE	B	paramString()	AE	BC	13	3
P9	C	C	D	BDE	D	paramString()	AD	BC	14	2
P3	D	None	X	BDE	D	X	AD	AE	10	6
P5	C	X	C	BDE	B	X	AD	BC	11	5
P4	No	X	No	No	X	paramString()	X	X	5	11
P10	C	C	X	BDE	X	X	AD	BC	12	4
P13	C	C	D	BDE	D	paramString()	AD	BC	15	1

Table 28

The experiment results: time measurement and task answers

### A.3.3 Debriefing Questionnaire Results

Table 29 presents the participant answers provided for the debriefing questionnaire.

Participant ID	DB1	DB2	DB3	DB4	DB5	DB6	DB7	DB8	DB9	DB10
S4	15	5	5	5	5	5	5	5	5	4
S6	10	5	5	4	5	5	5	4	5	5
S7	13	5	5	4	5	5	5	5	5	5
S15	26	5	5	5	5	5	5	5	5	5
S14	25	4	5	4	5	5	5	5	4	4
S8	15	5	5	5	4	5	5	4	4	4
S12	5	5	5	5	5	5	5	5	5	5
S10	26	5	5	5	5	5	5	5	4	4
S13	35	4	3	3	5	5	5	5	5	5
S9	13	4	5	5	5	5	5	5	5	4
S16	26	5	5	5	5	5	5	5	5	5
P1	101	5	5	5	5	5	5	5	2	4
P16	20	5	5	5	5	5	5	5	4	4
P15	1	5	4	3	5	5	5	5	5	5
P11	13	5	5	5	5	5	5	4	3	4
P12		5	5	4	5	5	5	4	5	5
P6	71	4	4	4	4	4	4	4	2	4
P9	57	4	5	5	5	5	5	4	4	4
P3	40	4	4	5	5	5	5	4	4	4
P5	71	4	4	4	4	4	5	5	3	3
P4	72	1	1	1	1	2	1	3	5	3
P10	100	4	4	3	3	2	5	4	2	2
P13	26	5	5	5	5	5	5	5	4	4

Table 29 The experiment results: the debriefing questionnaire

**Incomplete answers:** participant P12 did not provide any number when answering the question DB1. Instead, he/she wrote that “the task is easy, but time consuming”. This answer could not be used in the further analysis.

The answers to questions DB11 and DB12, not provided in the above table for space reasons, contained various suggestions for improvements of the provided tool functionalities. However, these suggested functionalities were already covered by the full Variant Analysis tool, which was not known to the experiment participants.

## Appendix B      Application Guidance

The information provided in this appendix concerns the practical use of the results of the defined similarity analysis approach in the context of its application scenarios. As discussed in Section 1.3, there are several context factors influencing the software migration decisions. The concerns such as functional similarity, future plans regarding the product variants [Schmid 2005], the overall code quality [Wleklik 2011], and resource availability play an at least as important role as the code similarity analysis results. Furthermore, many different reverse engineering techniques can be used on the analyzed asset variants to recover information not necessarily concerning code similarity, but still relevant for the migration [Knodel 2005] [Duszynski 2010b]. Hence, as our approach focuses on the code similarity only, the application guidance described below is necessarily rather an advice than a precise algorithm, and is intended to rather support reuse decisions than to prescribe them. For each group of similar software asset variants, the unique combination of context factors might require a case-specific use of the analysis results, overriding the application guidance.

The existing guidance concerning software reengineering, such as the refactoring methods [Fowler 1999], reengineering patterns [Demeyer 2008], or clone detection and removal approaches [Rieger 1999] [Schulze 2013], applies also in the context of our application scenarios. Hence, the application guidance discussed below complements these approaches and does not override them.

### B.1      Reuse Potential Assessment and Software Consolidation

In this section, we discuss the guidance concerning the application scenarios AS1 (reuse potential assessment) and AS2 (consolidation of existing reusable software). In both these scenarios, the analysis purpose is to identify the variants of assets or their constituent parts which are suitable for reuse introduction. Hence, from the similarity analysis point of view the foremost concern is the identification of assets having high similarity. For each asset or asset part, the following decisions can be made:

- The asset can be consolidated and made reusable for all its variants.
- The asset can be consolidated and made reusable for a subgroup of its variants.
- The asset can be left unconsolidated, and all its variants can be further maintained in parallel.
- The asset can be rewritten in a reusable form, abandoning the content of existing variants but using the information on their similarity.



### B.1.1 Similarity Properties Supporting the Scenarios

From the code similarity point of view, a group of some or all variants of the analyzed assets is particularly suitable for consolidation when the following criteria are met:

- **High content similarity:** the union of the content sets contains at least 80% of core content, or 80% of such content which is common (core or shared) within the selected variant subgroup.
- **High concentration of similar and unique content:** the content common to the selected variants, as well as the unique content, is concentrated to over 90% in the Content Filled Elements belonging to the analyzed asset. A high proportion of unique content is not detrimental to reuse migration if whole Content Filled Elements are unique. In such a case, these elements can be conveniently handled by the compositional variability mechanisms regardless of their size. In contrast to that, a migration of asset variants containing highly dispersed unique content likely requires a higher effort.
- **Large fragments of similar code:** the content fragments having the same similarity are large, and only a low number of different set intersections are represented in the Content Filled Elements (10 or less in an element, while each element can contain a different group of intersections).

Naturally, the consolidation is also possible for asset variants which do not fulfill some or all of the above criteria – however, in such a case a relatively higher consolidation effort is likely needed. Furthermore, in case the specified criteria are met only for a specific part of the analyzed asset, the analyst can decide to perform the consolidation only for that asset part, achieving for it the reuse benefits, and leave the remaining asset content unaffected. Finally, the asset variant consolidation might be not attempted despite the fulfillment of the stated criteria: for example, due to a low code quality the asset might be instead rewritten in a reusable form.

### B.1.2 Prioritization of Consolidation Activities

The consolidation planning frequently needs to prioritize the activities performed on the identified asset variants. In particular, the consolidation activities should quickly provide a benefit in the form of reduced asset maintenance effort – the saved effort can be then in turn used for further extension of the consolidation scope. The following criteria support the prioritization of consolidation activities:

- **Overall maintenance intensity:** the assets experiencing heavy maintenance should be prioritized in the consolidation activities for two reasons. First, performing further changes on the candidate asset variants before the consolidation might increase their complexity and reduce their reuse potential. On the other hand, the changes included after the consolidation might require less effort, due to the

already existing reuse, and are likely implemented in a more reuse-supportive way. Second, the introduction of reuse to the heavily maintained variants provides a faster return on the consolidation investment. In contrast to that, the consolidation of low-maintenance asset variants might never pay off, as the achieved overall maintenance savings might be low despite a high content similarity.

- **Near-time maintenance intensity:** for the reasons analogical as discussed in the previous point, an asset might be prioritized for consolidation if a significant maintenance activity is planned for its variants soon. Even if the average maintenance intensity of the asset is low, the summary effort needed for the consolidation and the subsequent phase of intensive maintenance, performed on the already reusable content, might be lower than the effort required for repetitive maintenance tasks performed on the non-consolidated variants. Hence, this situation potentially provides a quick pay off for the consolidation effort.
- **Very high code similarity:** the consolidation of nearly-identical asset variants requires likely a low effort. However, it provides a quick benefit even if the average asset variants maintenance intensity is low: the activities related to the management and quality assurance of the variants, which are frequently effort-intensive, can afterwards only be performed once for all previously independent asset variants.
- **The neighborhood of reusable content:** it might be beneficial to concentrate the consolidation activities on a few areas of the asset content, and create a low number of relatively large “reuse islands” instead of a high number of small, scattered reusable elements. The larger reusable asset parts are easier to manage, and are more likely to represent a semantically coherent subsets of asset functionality. Hence, the candidate asset parts which contribute to increasing the size of already existing commonalities should be prioritized.

### B.1.3 Further Activities Supporting the Scenarios

The reuse consolidation can be performed using different approaches – for example, the group of identified asset variants can be merged together, or a single variant can be extended to cover the functionality required by all other variants. In the case of merging, the following complementary implementation level activities might be performed before the actual consolidation:

- **Removal of non-significant differences:** during the parallel variant code maintenance, minor implementation differences not significantly influencing the asset functionality can emerge. For example, a refactoring or code commenting activity might be performed only on some of the variants, or a propagation of a code change might not be propagated consistently between the relevant variants. Furthermore, the variant code might differ in the used file and identifier names.

Because an unification of such differences is relatively simple, and there is no reason to preserve and manage them in the resulting reusable assets, the removal of non-significant differences can be performed as one of the first consolidation steps, helping in the isolation of more significant variant-specific asset content differences.

- **Dead code removal:** particularly in the older asset groups the amount of dead code can be relatively high. The creation of dead code can be intensified due to the variant cloning, especially in case the ancestor variant is not fully understood by the developer and the new clone is used in a different context. Obviously, dead code is not reusable and the effort spent on its removal pays off during the consolidation.

## B.2 Parallel Variant Maintenance

In the application scenario AS3 (parallel variant maintenance) the similar content is not intended to be consolidated. Instead of identifying potentially reusable content parts, the analysis supports the developers in understanding the similarity distribution in order to exploit that similarity during the parallel maintenance activities. The main activities utilizing the similarity information are hence the planning and verification of parallel content changes as well as the content inspections. A prior **removal of non-significant content differences**, as described in the previous section, is beneficial for both these activities. Subsequently, the most significant effort reduction in the parallel maintenance activities can be expected for asset variants having the following properties:

- **Nearly identical content** of asset or Content Filled Element variants (98% or more common code). In that case, the further parallel content changes implementing analogical functionality are likely to be performed in a syntactically similar way for these nearly identical variants. The similarity of both base code and the changes is also supportive for change verification, which aims to ensure that the changes were performed consistently in all variants, and all differences between the particular change implementations are intended and not accidental. Finally, a significant share of the content inspection effort can be saved, as the reviewed content is already known and only needs to be verified for the possibility of a yet unconsidered usage context.
- **Asset or Element variant fully covering the content of another variant**, possibly with the exception of minor unique content parts. In that case, the benefits occurring for nearly identical content, i.e. the high syntactical similarity of functionally identical changes, can also be expected. Hence, the results of maintenance activities performed on the larger variant can be frequently transferred to the smaller one after a short review of its usage context.
- **Large fragments of similar content**, i.e. content fragments belonging to the same set intersection, exhibit for the involved variant content sets the same benefits as discussed above.

The benefits of understanding the similarity distribution are not limited to the nearly identical asset variants – however, they are the most significant and visible for these assets. Reduction of content change and inspection effort can also be achieved in the case if only specific asset parts, affected by the given maintenance activity, exhibit a sufficiently high content similarity. Admittedly, in the parallel maintenance scenario the content similarity which results in a sufficient benefit needs to be higher compared to the other two scenarios, which are focusing on reuse introduction. While two asset variants with medium similarity (50-70% of common content) can still be transformed to a reusable form in many cases, the difference in their functionality is likely high enough to require a separate change analysis and inspection performed for each of the asset variants.



---

## Curriculum Vitae

<b>Name</b>	Sławomir Duszyński	
<b>Address</b>	Konrad-Adenauer-Strasse 43 67663 Kaiserslautern	
<b>Date of Birth</b>	16 September 1981	
<b>Place of Birth</b>	Wrocław, Poland	
<b>Marital Status</b>	Married, 1 child	
<b>Education</b>	1988–1996	Primary school
	1996–2000	Adam Mickiewicz Lyceum in Wrocław
	2000–2005	Wrocław University of Technology Computer Science / Software Engineering Degree: Master of Science
<b>Professional Experience</b>	2002–2004	Co-founder of a small software company Jawor, Poland
	2004–2005	Software developer Siemens sp. z o.o., Wrocław
	2005–2007	Software developer sympat GmbH / evosoft GmbH, Nürnberg (located at Siemens AG, Fürth)
	2007–today	Researcher Fraunhofer Institute for Experimental Software Engineering (IESE), Kaiserslautern

Kaiserslautern, 9<sup>th</sup> January 2015





# PhD Theses in Experimental Software Engineering

- Volume 1**      **Oliver Laitenberger** (2000), *Cost-Effective Detection of Software Defects Through Perspective-based Inspections*
- Volume 2**      **Christian Bunse** (2000), *Pattern-Based Refinement and Translation of Object-Oriented Models to Code*
- Volume 3**      **Andreas Birk** (2000), *A Knowledge Management Infrastructure for Systematic Improvement in Software Engineering*
- Volume 4**      **Carsten Tautz** (2000), *Customizing Software Engineering Experience Management Systems to Organizational Needs*
- Volume 5**      **Erik Kamsties** (2001), *Surfacing Ambiguity in Natural Language Requirements*
- Volume 6**      **Christiane Differding** (2001), *Adaptive Measurement Plans for Software Development*
- Volume 7**      **Isabella Wieczorek** (2001), *Improved Software Cost Estimation A Robust and Interpretable Modeling Method and a Comprehensive Empirical Investigation*
- Volume 8**      **Dietmar Pfahl** (2001), *An Integrated Approach to Simulation-Based Learning in Support of Strategic and Project Management in Software Organisations*
- Volume 9**      **Antje von Knethen** (2001), *Change-Oriented Requirements Traceability Support for Evolution of Embedded Systems*
- Volume 10**    **Jürgen Münch** (2001), *Muster-basierte Erstellung von Software-Projektplänen*
- Volume 11**    **Dirk Muthig** (2002), *A Light-weight Approach Facilitating an Evolutionary Transition Towards Software Product Lines*
- Volume 12**    **Klaus Schmid** (2003), *Planning Software Reuse – A Disciplined Scoping Approach for Software Product Lines*
- Volume 13**    **Jörg Zettel** (2003), *Anpassbare Methodenassistenz in CASE-Werkzeugen*
- Volume 14**    **Ulrike Becker-Kornstaedt** (2004), *Prospect: a Method for Systematic Elicitation of Software Processes*
- Volume 15**    **Joachim Bayer** (2004), *View-Based Software Documentation*
- Volume 16**    **Markus Nick** (2005), *Experience Maintenance through Closed-Loop Feedback*

- Volume 17**     **Jean-François Girard** (2005), *ADORE-AR: Software Architecture Reconstruction with Partitioning and Clustering*
- Volume 18**     **Ramin Tavakoli Kolagari** (2006), *Requirements Engineering für Software-Produktlinien eingebetteter, technischer Systeme*
- Volume 19**     **Dirk Hamann** (2006), *Towards an Integrated Approach for Software Process Improvement: Combining Software Process Assessment and Software Process Modeling*
- Volume 20**     **Bernd Freimut** (2006), *MAGIC: A Hybrid Modeling Approach for Optimizing Inspection Cost-Effectiveness*
- Volume 21**     **Mark Müller** (2006), *Analyzing Software Quality Assurance Strategies through Simulation. Development and Empirical Validation of a Simulation Model in an Industrial Software Product Line Organization*
- Volume 22**     **Holger Diekmann** (2008), *Software Resource Consumption Engineering for Mass Produced Embedded System Families*
- Volume 23**     **Adam Trendowicz** (2008), *Software Effort Estimation with Well-Founded Causal Models*
- Volume 24**     **Jens Heidrich** (2008), *Goal-oriented Quantitative Software Project Control*
- Volume 25**     **Alexis Ocampo** (2008), *The REMIS Approach to Rationale-based Support for Process Model Evolution*
- Volume 26**     **Marcus Trapp** (2008), *Generating User Interfaces for Ambient Intelligence Systems; Introducing Client Types as Adaptation Factor*
- Volume 27**     **Christian Denger** (2009), *SafeSpection – A Framework for Systematization and Customization of Software Hazard Identification by Applying Inspection Concepts*
- Volume 28**     **Andreas Jedlitschka** (2009), *An Empirical Model of Software Managers' Information Needs for Software Engineering Technology Selection  
A Framework to Support Experimentally-based Software Engineering Technology Selection*
- Volume 29**     **Eric Ras** (2009), *Learning Spaces: Automatic Context-Aware Enrichment of Software Engineering Experience*
- Volume 30**     **Isabel John** (2009), *Pattern-based Documentation Analysis for Software Product Lines*
- Volume 31**     **Martín Soto** (2009), *The DeltaProcess Approach to Systematic Software Process Change Management*
- Volume 32**     **Ove Armbrust** (2010), *The SCOPE Approach for Scoping Software Processes*

- Volume 33**     **Thorsten Keuler** (2010), *An Aspect-Oriented Approach for Improving Architecture Design Efficiency*
- Volume 34**     **Jörg Dörr** (2010), *Elicitation of a Complete Set of Non-Functional Requirements*
- Volume 35**     **Jens Knodel** (2010), *Sustainable Structures in Software Implementations by Live Compliance Checking*
- Volume 36**     **Thomas Patzke** (2011), *Sustainable Evolution of Product Line Infrastructure Code*
- Volume 37**     **Ansgar Lamersdorf** (2011), *Model-based Decision Support of Task Allocation in Global Software Development*
- Volume 38**     **Ralf Carbon** (2011), *Architecture-Centric Software Producibility Analysis*
- Volume 39**     **Florian Schmidt** (2012), *Funktionale Absicherung kamerabasierter Aktiver Fahrerassistenzsysteme durch Hardware-in the-Loop-Tests*
- Volume 40**     **Frank Elberzhager** (2012), *A Systematic Integration of Inspection and Testing Processes for Focusing Testing Activities*
- Volume 41**     **Matthias Naab** (2012), *Enhancing Architecture Design Methods for Improved Flexibility in Long-Living Information Systems*
- Volume 42**     **Marcus Ciolkowski** (2012), *An Approach for Quantitative Aggregation of Evidence from Controlled Experiments in Software Engineering*
- Volume 43**     **Igor Menzel** (2012), *Optimizing the Completeness of Textual Requirements Documents in Practice*
- Volume 44**     **Sebastian Adam** (2012), *Incorporating Software Product Line Knowledge into Requirements Processes*
- Volume 45**     **Kai Höfig** (2012), *Failure-Dependent Timing Analysis – A New Methodology for Probabilistic Worst-Case Execution Time Analysis*
- Volume 46**     **Kai Breiner** (2013), *AssistU – A framework for user interaction forensics*
- Volume 47**     **Rasmus Adler** (2013), *A model-based approach for exploring the space of adaptation behaviors of safety-related embedded systems*
- Volume 48**     **Daniel Schneider** (2014), *Conditional Safety Certification for Open Adaptive Systems*
- Volume 49**     **Michail Anastasopoulos** (2013), *Evolution Control for Software Product Lines: An Automation Layer over Configuration Management*
- Volume 50**     **Bastian Zimmer** (2014), *Efficiently Deploying Safety-Critical Applications onto Open Integrated Architectures*

**Volume 51**

**Slawomir Duszynski** (2015), *Analyzing Similarity of Cloned Software Variants using Hierarchical Set Models*

Software Engineering has become one of the major foci of Computer Science research in Kaiserslautern, Germany. Both the University of Kaiserslautern's Computer Science Department and the Fraunhofer Institute for Experimental Software Engineering (IESE) conduct research that subscribes to the development of complex software applications based on engineering principles. This requires system and process models for managing complexity, methods and techniques for ensuring product and process quality, and scalable formal methods for modeling and simulating system behavior. To understand the potential and limitations of these technologies, experiments need to be conducted for quantitative and qualitative evaluation and improvement. This line of software engineering research, which is based on the experimental scientific paradigm, is referred to as 'Experimental Software Engineering'.

In this series, we publish PhD theses from the Fraunhofer Institute for Experimental Software Engineering (IESE) and from the Software Engineering Research Groups of the Computer Science Department at the University of Kaiserslautern. PhD theses that originate elsewhere can be included, if accepted by the Editorial Board.

Editor-in-Chief: Prof. Dr. Dieter Rombach

Director Business Development of Fraunhofer IESE and Head of the AGSE Group of the Computer Science Department, University of Kaiserslautern

Editorial Board Member: Prof. Dr. Peter Liggesmeyer

Executive Director of Fraunhofer IESE and Head of the SEDA Group of the Computer Science Department, University of Kaiserslautern

Editorial Board Member: Prof. Dr. Frank Bomarius

Deputy Director of Fraunhofer IESE and Professor for Computer Science at the Department of Engineering, University of Applied Sciences, Kaiserslautern

ISBN 978-3-8396-0860-9

