Minimal-invasive provenance integration into data-intensive systems

Dissertation

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

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

von Dipl.-Inform. Martin Schäler
geboren am 31.März 1985 in Havelberg

Gutachter:

Prof. Dr. Gunter Saake
Prof. Dr. Wilhelm Hasselbring
Prof. Dr. Klaus Schmid

Ort und Datum des Promotionskolloquiums: Magdeburg, 05.12.2014

Schäler, Martin:
Minimal-invasive provenance integration into data-intensive systems
Dissertation, Otto-von-Guericke-Universität Magdeburg, 2014.

Abstract

The purpose of provenance is to determine origin and derivation history of data. Thus, provenance is used, for instance, to validate and explain computation results. Due to the digitalization of previously analog processes that consume data from heterogeneous sources and increasing complexity of respective systems, it is a challenging task to validate computation results. To face this challenge, there has been plenty of research resulting in solutions that allow for capturing of provenance data. However, all these approaches have in common that they are tailored for their specific use case. Consequently, provenance is considered as an integral part of these approaches that can hardly be adjusted for new requirements. We envision that provenance, which highly needs to be adjusted to the needs of specific use cases, should be a *cross-cutting concern* that can *seamlessly be integrated* without interference with the original system.

The goal of this thesis is to analyze in how far techniques, known from modern software engineering, such as feature-oriented programming, aspect-oriented programming, or advanced preprocessor techniques are sufficient to integrate a totally new variability dimension into a monolithic systems. In particular, we empirically explore benefits and drawbacks of applying these techniques in contrast of using an intuitive approach. In consequence, the key task is to design, implement, and evaluate a *Provenance solution* in form of an application programming interface (API). That API addresses generality as well as customizability, and minimally affects properties of the original system (minimal invasiveness). Our results indicate that the nature of the provenance concern changes based on the granularity of the captured provenance data. Thus, we need different implementation techniques for different application scenarios.

Zusammenfassung

Der aus dem Englischen entlehnte Begriff *Provenance* zielt auf die Bestimmung der originalen Herkunft und der Transformationshistorie von Daten und realen Objekten. Aus diesem Grund wird Provenance beispielsweise dazu verwendet Resultate digitaler Berechnungsvorgänge zu erklären und zu validieren. Aufgrund der zunehmenden Durchdringung der Gesellschaft mit digitalen Prozessen, die zuvor analog bearbeitet wurden, ist die Bestimmung der Validität der digitalen Gegenstücke zum Einen schwierig, zum Anderen von essentieller Bedeutung. Von besonderer Herausforderung ist hierbei, dass die digitalen Prozesse oft Daten aus verschiedenen Quellen miteinander kombinieren und aggregieren, sowie diese Systeme selbst immer komplexer werden. Aus diesem Grund existiert eine Vielzahl von Ansätzen, die sich mit dem Sammeln und Auswerten von Provenance-Daten beschäftigt. Allerdings sind diese Systeme bisher als Einzelfallanwendung konzipiert. Das bedeutet, dass die Provenance-Funktionalität fest und unveränderlich von Anfang an in diese Systeme integriert ist. Somit kann sie nicht oder nur schwer auf sich ändernde Anforderungen angepasst werden. Im Gegensatz dazu zielt diese Arbeit darauf Provenance-Funktionalität, welche schnell an sich ändernde Gegebenheiten angepasst werden muss, als *quer-schneidenden* Belang anzusehen. Dieser soll *unmittelbar* und *ohne negative Beeinflussung* in bestehende Systeme integriert werden können.

Die Zielstellung dieser Dissertation ist es zu analysieren inwiefern Ansätze und Techniken aus der derzeitigen Forschung im Bereich Software-Engineering dazu verwendet werden können, eine neue Variabilitätsdimension in bestehende Systeme einzubetten. Hierbei stehen Techniken, wie feature- und aspektorientierte Programmierung sowie erweiterte Präprozessoren im Vordergrund, da für diese bereits bekannt ist, dass sie sich für ähnliche Anwendungsfälle eignen. Die Grundidee besteht darin Vor- und Nachteile dieser Techniken im Vergleich zu intuitiven Techniken explorativ zu bestimmen. Daher besteht eine der Kernaufgaben darin eine möglichst generelle und flexible Provenance-Lösung zu designen, zu implementieren und zu evaluieren. Die Integration dieser Lösung soll dabei *minimal-invasiv* erfolgen. Das bedeutet, dass beispielsweise die Laufzeit des Originalsystems möglichst wenig verändert wird, aber auch, dass der Integrationsaufwand möglichst gering ausfällt. Unsere Ergebnisse zeigen, dass die Art der zu integrierenden Provenance-Funktionalität sich mit zunehmender Granularität der gesammelten Provenance-Daten ändert und somit unterschiedliche Implementierungstechniken benötigt werden, um das Ziel einer minimal-invasiven Integration zu erreichen.

Acknowledgements

Not starting, not defining the concept, or conducting the implementation, but finding the point when to finish a dissertation is the real challenge. The long road from starting such a project to its actual end, should and cannot be taken in isolation. In this sense, I like to express my gratitude to several people in the following.

First and foremost, I would like to thank my advisors Gunter Saake and Thomas Leich for supporting me, starting from my Bachelor thesis when I was still a student assistant researcher at the METOP until finishing my dissertation. Similarly, I would like to express my gratitude to my external reviewers Prof. Hasselbring and Prof. Schmid. During the past years as a researcher, many people supported me as teacher, mentor, (competitive) colleague, or friend. To this end, I want to express my gratitude to the (current and former) members of the Research Group on Databases and Software Engineering at the University of Magdeburg, in particular, Christian Kästner, Marko Rosenmüller, Martin Kuhlemann, Norbert and Janet Siegmund, Thomas Thüm, Mario Pukall, Andreas Lübcke, Sandro Schulze, Alexander Grebhahn, Reimar Schröter (also making the best coffee), David Broneske, Sebastian Breß, Ingolf Geist, and Veit Köppen who accompanied me all the way. You helped me to understand research in its entirety, by controversial discussions, constructive criticism, and by showing me different perspectives on certain research topics. In that, you made me a lot the researcher that I am today. Similarly, many members of the AMSL team, especially those sitting with me in Room 001 supported me in the same sense. Therefore, I would like express my gratitude to Mario Hildebrandt, Ronny Merkel, Stefan Kiltz, and Christian Krätzer as well. A special thank goes to the METOP research institute, namely Matthias Ritter, Andy Kenner, Dan Klingenberg, Andreas Holstein, Katja Gündel, and Mario Pape. Project work in Digi-Dak and some industrial inside views often help me to see challenges from a different perspective. Sometimes, taking a break and doing something entirely different helps to clear your head resulting in new inspiration and motivation. Thus, I would like to thank all people from my athletics club SC Magdeburg. Special thanks hereby go to my former trainer Klaus Wübbenhorst, my former and current training partners, as well as my current athletes. In particular, I would like to mention Birk Lösche, Benjamin Wegner, Susan Wolf, and Anne Metzlaff. Finally, I would like to issue special thanks the Mensa, as research makes hungry.

Contents

List of Figures xv

List of Tables xvii

List of Acronyms xix

1 Introduction 1

1.1 Contribution . . . 3
1.2 Outline . . . 4

2 On the nature of provenance 7

2.1 What is provenance? . . . 7
2.2 Background and notation . . . 8
2.2.1 The open provenance model . . . 9
2.2.2 The relationship between lineage, why, how, and where provenance 9
2.3 A hierarchical provenance framework . . . 11
2.3.1 Provenance systems . . . 12
2.3.2 Necessity for abstraction layers . . . 12
2.4 Workflow layer . . . 15
2.4.1 Extended open provenance model . . . 15
2.4.2 Graph refinements . . . 15
2.4.3 Usage and limitations . . . 17
2.4.4 Related approaches . . . 20
2.5 Existence layer . . . 20
2.5.1 Sub levels of the existence layer . . . 21
2.5.2 Limitations of this Layer . . . 25
2.6 Value origin layer . . . 25
2.6.1 Structure of artifacts . . . 25
2.6.2 Value origin with existing approaches . . . 26
2.6.3 Relationship to the previous layer . . . 28
2.7 Cross-cutting reliability layer . . . 29
2.7.1 Reliability: A different dimension of provenance . . . 29
2.7.2 Current research on reliable provenance . . . 30
2.8 The question of identity . . . 30

2.8.1 Determining artifact identity . . . 31
2.8.2 A flexible notion of identity . . . 32
2.9 Insights gained . . . 32

3 Background on software product lines 35
3.1 Domain and application engineering . . . 36
3.1.1 Domain engineering . . . 36
3.1.2 Application engineering . . . 38
3.2 Implementation techniques . . . 39
3.2.1 Intuitive techniques as reference . . . 39
3.2.2 Preprocessor-based techniques . . . 40
3.2.3 Aspect-oriented programming . . . 41
3.2.4 Feature-oriented programming . . . 42
3.2.5 Additional techniques and tool support . . . 43
3.3 Advanced topics of relevance for this thesis . . . 43
3.3.1 Composition of features structure trees . . . 43
3.3.2 Homogeneous and heterogeneous cross-cutting concerns . . . 44
3.4 Summary . . . 46

4 Toward a general provenance- capturing solution for existing systems 47
4.1 Reasoning about the necessity for provenance integration in existing systems 47
4.1.1 The need for customizable solutions . . . 48
4.1.2 Are current solutions feasible? . . . 51
4.1.3 The necessity for an own solution . . . 52
4.2 The goal of minimal-invasive and efficient provenance integration . . . 52
4.2.1 A notion of invasiveness . . . 53
4.2.2 Measuring invasiveness of different implementation techniques . . 56
4.3 Conceptual design . . . 57
4.3.1 Vision - A universe of provenance . . . 58
4.3.2 Architecture - The neuron analogy . . . 58
4.3.3 Parts of special interest and missing basic technologies . . . 61
4.3.4 Derived research agenda . . . 62
4.4 Summary . . . 63

5 Tailoring database schemas of provenance data stores 65
5.1 Generalization of the basic problem . . . 65
5.2 Requirements for tailored database schemas . . . 66
5.3 Limitations of currently used approaches . . . 67
5.3.1 Global schema . . . 67
5.3.2 View-based approaches . . . 67
5.3.3 Framework solutions . . . 68
5.4 Our solution . . . 68
5.4.1 Basic idea of our approach . . . 68
5.4.2 Relationship between features and database schema elements . . . 68

5.4.3 Composing a tailored schema variant . . . 70
5.4.4 Structure of features at implementation level . . . 70
5.5 Evaluation of our approach . . . 70
5.5.1 An industrial-size case study . . . 71
5.5.2 Feasibility of the approach . . . 72
5.5.3 Improving maintenance and further development . . . 76
5.5.4 Improving data integrity . . . 78
5.5.5 Comparison to existing approaches . . . 78
5.6 Role and contribution of the approach for this thesis . . . 79

6 Database-centric chain-of-custody **81**
6.1 Analysis goal and resulting methodology . . . 81
6.1.1 A holistic approach . . . 82
6.1.2 Assumptions and architecture . . . 85
6.2 Preliminary considerations . . . 91
6.2.1 What is the initial situation? . . . 91
6.2.2 What do we want? - Required functionality . . . 92
6.3 The provenance feature tree . . . 94
6.3.1 The provenance feature . . . 94
6.3.2 Initial linking the ProveSet to the artifact . . . 95
6.3.3 The Security feature . . . 97
6.3.4 A short intro to (invertible) watermarks . . . 98
6.3.5 Feature content of the Watermarking feature . . . 100
6.3.6 A short intro to forensic file formats . . . 101
6.3.7 Re-computation feature . . . 103
6.4 Summary . . . 105

7 First exploratory case studies **107**
7.1 Coarse-grained provenance integration on tool side . . . 108
7.1.1 Objectives . . . 108
7.1.2 Implementation concept . . . 109
7.1.3 Provenance integration . . . 111
7.1.4 Observations . . . 112
7.1.5 Lessons learned . . . 120
7.2 Coarse-grained provenance for databases . . . 121
7.2.1 A concept for coarse-grained provenance integration on database side . . . 121
7.2.2 Provenance integration . . . 122
7.2.3 Observations . . . 124
7.2.4 Lessons learned . . . 126
7.3 Scientific data management for QuEval . . . 126
7.3.1 Contribution of QuEval for the goal of minimal-invasive provenance integration . . . 127
7.3.2 Scientific-data management . . . 128

7.3.3 New features for tailored index structure implementations 130
7.3.4 Results and lessons learned 131
7.4 Contributions and conclusions 134

8 Fine-grained provenance integration into complex systems 137
8.1 Granularity refinement on tool side 137
8.1.1 Expected insights 138
8.1.2 Provenance integration 139
8.1.3 Initial integration 141
8.1.4 Extracted provenance data 143
8.1.5 Performance considerations 145
8.1.6 Lessons learned 147
8.2 Fine-grained provenance for databases 148
8.2.1 Expected insights 148
8.2.2 Motivating scenario 149
8.2.3 Feature model and resulting changes 150
8.3 Provenance integration for databases 151
8.3.1 Case study selection 151
8.3.2 Exploratory implementation 153
8.3.3 Evaluation of non-functional properties 155
8.3.4 Alternative implementation techniques 157
8.4 Adaption for different database systems 159
8.4.1 Premises for generalization 159
8.4.2 Versions of HyperSQL 160
8.4.3 H2 Database 162
8.5 Contributions and conclusions 163

9 Conclusion and future work 165

A Appendix 169

Bibliography 175

List of Figures

1.1 Tailored provenance integration into an existing system 2

2.1 Provenance graph 9
2.2 Relationship between provenance terms 10
2.3 Provenance system 12
2.4 Hierarchical provenance framework 14
2.5 Complex artifacts and processes 16
2.6 Visualization of implicit dependencies 17
2.7 Provenance graph containing complex artifacts and processes 18
2.8 Exemplary backtracking levels 18
2.9 Process refinement 19
2.10 Sub levels of the existence layer 21
2.11 Computation with provenance 27

3.1 Phases of SPL engineering 36
3.2 Feature model of a database SPL 37
3.3 SPL implementation with conditional statements 40
3.4 Conditional compilation with preprocessors 41
3.5 SPL implementation with AOP 41
3.6 Feature-oriented SPL implementation using AHEAD 42
3.7 Excerpt of a features structure tree 44
3.8 Homogeneous and heterogeneous cross-cutting concerns 45

4.1 Vision - A universe of provenance 58

4.2 The neuron analogy . . . 59
4.3 Architecture mapping in program code using the static intuitive implementation technique . . . 61
4.4 Derived research agenda . . . 63

5.1 Basic idea to create a particular variant . . . 69
5.2 Composition of feature structure trees . . . 71
5.3 Procedure to map schema elements to features based on client implementation . . . 73
5.4 Tool support: Screen shot of mapping matrix . . . 74
5.5 Derivatives: Size of the Archiving features in different feature combinations 76
5.6 Complexity reduction of the schema variants in different versions of the case study . . . 79

6.1 Digi-Dak's chain-of-custody excerpt . . . 82
6.2 Analysis of the existing infrastructure . . . 84
6.3 Concept database-centric chain-of-custody . . . 86
6.4 Create a new intermediate artifact . . . 89
6.5 Infrastructure classes . . . 90
6.6 Artifact model without provenance . . . 93
6.7 Feature content . . . 94
6.8 Independence of the provenance feature . . . 94
6.9 Excerpt of the provenance feature tree . . . 95
6.10 The ProveSet relations . . . 95
6.11 Interactions caused by provenance feature on concept and implementation level . . . 97
6.12 Variability for the security feature . . . 97
6.13 Watermarking options . . . 100
6.14 Effects adding the Watermark feature to database schema generation . . . 102
6.15 Forensic file formats feature . . . 102
6.16 Provenance derivative . . . 105

7.1 Refinement of the neuron analogy for the first case study 109
7.2 Change of the provenance feature tree 113
7.3 Code to integrate using static if-approach with activated hashing feature 114
7.4 Average response time per feature and implementation technique 119
7.5 Average main-memory consumption after five minutes per feature and implementation technique 120
7.6 Exemplary usage of pgcrypto library 122
7.7 Average response time per feature on database side 125
7.8 Median maximum main-memory consumption 126
7.9 Query execution in QuEval 129
7.10 Exact-match query speed up for low-populated spaces 132
7.11 Epsilon-distance query speed up for low-populated spaces 133
7.12 k nearest-neighbor query speed up for low-populated spaces 134

8.1 Expected extracted provenance data 139
8.2 Architecture of the equalization tool 141
8.3 Extracted provenance data 144
8.4 Initial and optimized non-functional properties 146
8.5 Initial and optimized performance 147
8.6 Evidence containing four potential regions of interest 150
8.7 Extensions for the provenance API 152

A.1 Artifact validation procedure 171
A.2 Example implementation pg/plSQL - Coltuc watermark 172
A.3 Expected extracted provenance data 173

List of Tables

2.1 Refinement with fragmentary knowledge 24

2.2 Region of interest relation 26

4.1 Categorization of effects on functional behavior 54

5.1 Correlation of size on database and client side 75

5.2 Evaluation of the variable schema approach compared to currently used approaches 79

7.1 Effects on functional level 118

8.1 Necessary modifications in HyperSQL per feature using 154

8.2 Evaluation queries 156

8.3 Non-functional properties per feature and query 156

8.4 Dendrite and Soma reusabilty for HyperSQL versions 161

List of Acronyms

AFM aspectual feature modules

AOP aspect-oriented programming

API application programming interface

CIDE colored integrated development environment

DBMS database management system

FOP feature-oriented programming

FST feature structure tree

IDE integrated development environment

JDBC Java database connectivity

ODBC open database connectivity

OPM open provenance model

QuEval query evaluation framework

ROI region of interest

SPJU selection, projection, join, and union - as subset of the relational operators

SPL software product line

SQL structured query language

1. Introduction

Do you know where your data's been? This is the provoking title of a technical report from 2009 [ZCL09] describing one of the most challenging questions in our linked world. Data that is used for our own computations often stems from multiple sources, was transformed multiple times[1], and the original data sources are unknown. In addition, we use aggregated data, for that we often do not know the original data items nor their composition to the aggregated values(s). However, in case we want to understand complex computations (e.g., climate models [CBSA11]), determine the influence of a single data item, or have doubts on the correctness of specific computation result, we need detailed information on the composition of data items and their origin. These, in fact, are issues related to *provenance*. Moreover, if we have provenance information, we may have doubts on the reliability of the provenance information itself, because it may be tampered in multiple ways. Hence, we could even ask: What is the provenance of the provenance information? Indeed, sometimes having no provenance at all is considered to be better than having (partly) wrong provenance [BSS08].

Anyway, with the collection of (reliable) provenance data, we want to raise transparency of complex computations, which in summary addresses trust in these results. Often, there are already existing systems or grown infrastructure for that we need to capture provenance data instead of re-designing the whole infrastructure from scratch. Therefore, we require for a solution that allows us to integrate a totally new functionality (a concern) into that system. The challenge then is that this concern is potentially scattered across the source code of the whole system and that existing decomposition (e.g., into classes and methods for object-oriented programs) is not designed for this new concern. In addition to the aforementioned challenges, our insights reviewing current research on provenance reveal that there is no one-size-fits-it-all provenance functionality. In fact, based on the intended objective, different solutions require different *tailored* provenance integration as illustrated in Figure 1.1. Consequently, we need an approach to enhance

[1] Chinese whispers

an existing monolithic system with new functionality. However, we also need to limit the interference on functional level to the desired effects by keeping non-functional behavior as close as possible to the original system. In the remainder of this thesis, we refer to such an approach as *minimal-invasive* integration.

In software engineering, software product line development is a well-known approach that focuses on the creation of multiple variants of a program from one common code base. For software product lines, several implementation techniques are proposed to develop the single concerns and automatically compose the desired ones into one program. However, decomposing or designing software in terms of features and integrating a new variable concern into an existing monolithic system is a different, but related topic. Therefore, from our point of view, it is a valid proposition that knowledge from software product line research can be transferred here. The basic questions of this thesis therefore are: Can we adopt design and implementation techniques in order to model provenance and generate program variants with tailored provenance support? How do software product line implementation techniques allow to integrate the resulting provenance code fragments into an existing system in a minimal-invasive way? How do properties of the considered case studies and the nature of the provenance concern interact with our attempt to use software product lines to integrate provenance?

Methodology of this thesis

The goal of this thesis is to enrich existing systems, with additional tailored provenance capturing and storing capability (cf. Figure 1.1). The basic idea is to re-use the already existing infrastructure to minimize the amount of changes necessary to integrate the provenance functionality. To answer the aforementioned questions, this thesis focuses on using empirical methods. In particular, after an analysis of explicit limitations of current provenance solutions, we design several concepts contributing to our overall goal. To evaluate these concepts, we use a series of exploratory case studies to collect observations and gain insights that, in summary, help to find a sound answer regarding the aforementioned questions.

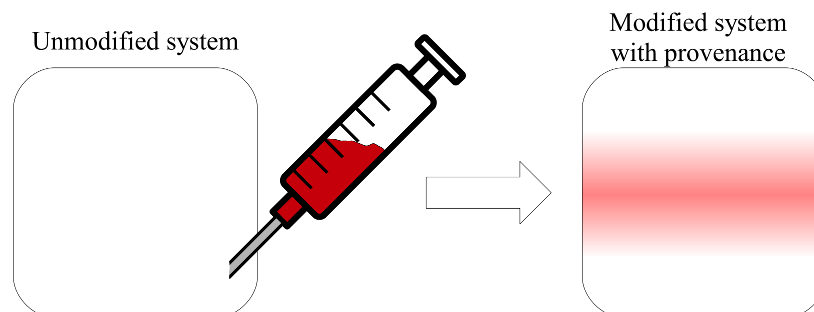


Figure 1.1: Tailored provenance integration into an existing system

1.1 Contribution

The contribution of this thesis shall mainly attract readers from two research communities. First, there are contributions on the nature of provenance and the relationship of different approaches on model and implementation level. Second, we contribute to software engineering, specifically to the feature-oriented software development community, due to the application and evaluation.

Contribution to the provenance community

Current provenance research focuses on three main directions. First there are formal approaches for relational databases, frameworks for scientific workflows, and (operation) system-based approaches.

However, focusing on addressing a broad variety of these approaches offers views from different directions. It includes the modeling and implementing provenance integration in a software product line. This, we argue, results into interesting new research directions for provenance research in order to augment solutions with provenance functionality.

Contribution to the software engineering community

As stated before, provenance is a variable concern. Due to user requirements, a system may need highly differing provenance extensions. Since we want to provide tailored provenance solutions, the contribution to the software product line community is to open an additional field for well-established methods as well as to reveal benefits and limitation of currently used approaches from a practical point of view.

Specific contributions

Besides the contribution for these scientific communities, we reveal properties of such software systems that should be avoided in order to easily integrate provenance and recommend alternative solutions that we favor for provenance integration.

Besides these high-level contributions, the most prominent specific contributions are:

1. We clearly motivate the necessity for minimal-invasive provenance integration and show short comings of currently used approaches resulting in the need for an own approach based upon the definition of the term minimal-invasive provenance integration.

2. We contribute a new approach to tailor relational database schemas (including user defined functions). This allows to automatically create tailored database schemas. We show the practical applicability of our approach with the help of an industrial size case study.

3. We define the database centric chain-of-custody. It is an approach to capture reliable coarse-grained provenance data. This concept is based on the infrastructure used to exchange data.

4. We propose a novel approach to exploit the object-oriented decomposition of a program in order to augment it to capture fine-grained provenance data and evaluate the applicability of this approach.
5. As a means to minimize the performance penalty after integrating provenance, we show how to use and to tailor multi-dimensional index structures. Therefore, we introduce *QuEval* as a framework to tailor and evaluate these indexes.
6. We design, implement, and evaluate a comprehensive and easy to integrate solution to track fine-grained provenance data for relational databases.

We bundle the resulting implementations and case studies in several projects. Therefore, we use the, from our point of view, most advanced and comprehensive development environment for software product line development FeatureIDE [TKB+14].

Primary result

The primary result of our investigation is that using software product lines to integrate provenance into existing monolithic systems works well, if we can exploit certain homogeneity in the system, such as the object-oriented decomposition. However, the best implementation technique depends on characteristics, especially granularity, of the the desired provenance functionality. In addition, we show that intuitive approaches are insufficient. In particular, the results for the considered implementation techniques (see Section 3.2) are:

1. An intuitive technique using conditional statements (i.e., `if`) is insufficient,
2. Preprocessors are often the only possible choice for fine-grained forms of provenance,
3. Feature-oriented programming is sometimes also well-suited for fine-grained forms provenance, especially if we consider evolution of the original programs.
4. We recommend aspect-oriented programming for coarse-grained provenance integration.

Besides this primary insights, our research and development resulted in several new approaches that are required for a comprehensive solution that allows for integrating provenance into existing systems.

1.2 Outline

The thesis is decomposed using the following structure. The primary decomposition criterion is the aforementioned contributions, which are nearly directly reflected in the outline.

Chapter 2. In Chapter 2, we analyze the concepts behind current provenance research, which is relevant in the remainder of this thesis to model the provenance concern in terms of features. In addition, this chapter reveals first short comings that are refined and analyzed in more detail in subsequent chapters and finally result in new concepts helping to reach the goal of this thesis.

Chapter 3. The third chapter contains basic background on software product lines that is later used to design, implement, and integrate the provenance capturing parts.

Chapter 4. In concept Chapter 4, we first motivate the need for integrating provenance into existing systems in form of a problem statement. Then, we analyze limitations of currently used solutions and approaches clearly resulting in the need for an own solution. Therefore, we design the overall infrastructure and reveal research gaps that need to be addressed in order to build such an infrastructure resulting in a research agenda for the remaining chapters, which address these research gaps.

Chapter 5. Chapter 5 addresses storing tailored provenance data in tailored database schemas. In this chapter, we contribute an approach to model variable relational database schemas in terms of features and evaluate this approach resulting in clear benefits using our new approach compared to alternative ones.

Chapter 6. Here, we contribute the database centric chain-of-custody as an abstract concept finally resulting in a feature model of the provenance software product line.

Chapter 7. In Chapter 7, we conduct the first exploratory case studies, where we integrate coarse-grained provenance capturing capability into object-oriented programs and as an additional layer on top of database systems.

Chapter 8. In Chapter 8, we conduct more complex case studies and integrate fine-grained provenance forms to validate the insights gained so far. To this end, we propose a concept to exploit the object-oriented decomposition to capture provenance. In addition, we contribute the evaluation of the provenance capturing approach and the provenance API for database systems.

Chapter 9. In Chapter 9, we conclude this thesis and state new research directions revealed during the work on this topic.

2. On the nature of provenance

In the following chapter, we identify relationships between existing provenance capturing approaches indicating that the ideas behind them can be unified in one general framework. As a result, this chapter is based on, but not limited to a literature review. The insights gained here are used to define features that allow us to integrate tailored provenance functionality in the remainder of this thesis. The chapter also offers a holistic view on provenance research in general. Furthermore, it contributes a general data model and an approach to determine the identity of data items. Thus, it lays required basic foundation for the remaining chapters. However, the specific features are derived in Section 6.3, based on the results presented subsequently and the general architecture. Note, the results presented here have been partly published in [SSS12a].

2.1 What is provenance?

What is provenance? As this term is used in many communities, such as relational databases [BKT01, CW00, GKT07b], (scientific) workflows [MCF+11, MBK+09, SPG08], and even to determine source code ownership [DGGH11], we cannot give an overall definition sufficient for all of these communities. Moreover, we even cannot give a clear definition to one of these communities as pointed out in [CCF+09], because of the diversity of provenance systems. But what we can say is what provenance is useful for: (1) Understanding (and possibly recomputation) of *foreign results* to validate or explain them and (2) Computation (and subsequent validation) of *own results* based on the provenance of previous results, which are used as input [CCF+09]. Moreover, several authors in the field of provenance have identified certain characteristics that seem to hold for provenance generally:

Unchangeability. Provenance describes what actually happened in the past and does not describe future alternatives. Therefore, it is unchangeable [CCF+09].

Fragmentation. This property states that the provenance information we have at each level of granularity always is fragmentary. For instance, every provenance system has some starting point. As Braun argues in a provoking manner, to be complete, we have to be able to track back the history of any data item to the Big Bang [BSS08]. Additionally, coarse-grained provenance notions (e.g., for workflows) omit possibly important details of the derivation process [CCT09]. In contrast, other forms of provenance contain a lot of very fine granular information, which might be hard to understand because the big picture is missing and there is simply too much information [ABC+10]. Thus, this is about the availability of (complete) provenance information at a certain level of granularity or in turn on missing fragments at a specific level of granularity.

Uncertainty. Provenance always contains uncertainty to some extent. Thus, in most cases no provenance is better than wrong provenance [MBM+10]. This problem also becomes visible in several papers dealing with *secure provenance*, such as [LLLS10, LM10, MBM+10], showing the necessity for reliable provenance.

Recently, provenance gained much attention [BKT01, CCF+09, GKT07b, MGH+10]. Unfortunately, it is very difficult to assign single research results to challenges in provenance, relate them to different aspects of provenance, alternative solutions or evaluate advantages and (possibly not obvious) drawbacks of a certain approach, because there is no general provenance framework covering the aforementioned aspects. To address this problem, we suggest such a general framework for provenance. To ensure generality, our framework is based on the previously introduced general characteristics. In fact, we use differences regarding *fragmentation* and *uncertainty* to describe abstraction layers that can be used for different purposes and contain several open research challenges. Knowing that our framework might be incomplete and furthermore there might be use cases that are not covered within, we explicitly encourage the reader to question and extend the results of this chapter. The basic idea is to consider new developments and research as features that can be integrated into our framework.

The final consequence of these initial considerations is that we cannot define provenance and related terms right now. We have to conduct an in depth analysis of current provenance research in the remainder of this chapter. This results in the definition of provenance terms in Section 2.9.

2.2 Background and notation

Subsequently, we introduce the notation of the well-known *Open Provenance Model* (OPM) for coarse-grained provenance and existing formalisms for fine-grained provenance which are relevant for the remainder of this thesis as our own notation and extensions are highly related to this notation.

2.2.1 The open provenance model

The OPM [Luc97, MCF+11] is an example for coarse-grained workflow provenance. According to the OPM, the causal dependencies of a *particular* data item within a provenance system are modeled as an acyclic graph (V, E). The vertices of this graph represent: (1) *artifacts* naming physical objects or their digital representation, (2) *processes* taking input artifacts to create new artifacts, or (3) *agents* that have certain effects on processes. The edges of the provenance graph show the relationships between the vertices. For instance in Figure 2.1, process *make photo* takes two input artifacts of roles *camera* and *photographic objective* to create an artifact *photo* and is triggered by a *photographer*. As a result, in Figure 2.1, we show the past derivation history of the *result artifact* `img0001.jpg`, but no possible future usage of the artifacts or alternatives in the past. Furthermore, we extended the OPM with additional semantic information. As depicted in Figure 2.1, we distinguish between three different artifact types: Initial data items, intermediate results, and final results[1]. This differentiation is important, because we do not produce initial data items in a provenance system and thus, have to rely on (provenance) information tagged to this object (*fragmentation*). In contrast, we produce and consume intermediate results in our system. Moreover, final results may be initial data items for following (future) processes.

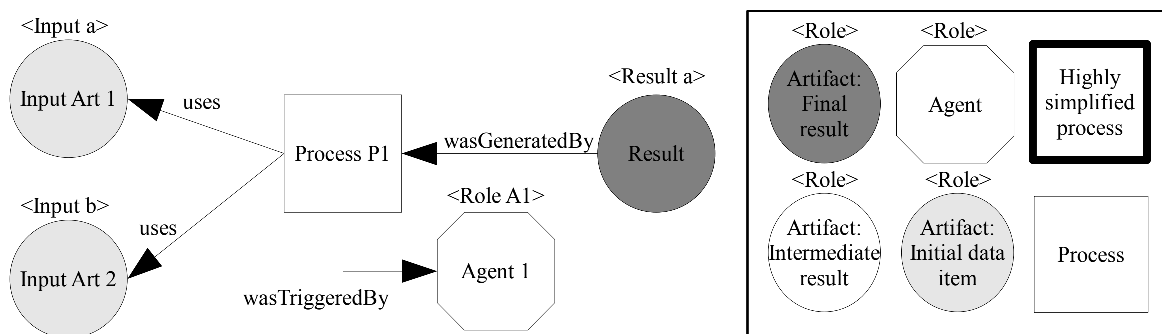


Figure 2.1: Provenance graph

2.2.2 The relationship between lineage, why, how, and where provenance

As already mentioned, we build our provenance framework based on existing formalisms. Hence, we briefly introduce Lineage [CW00], Why [BKT01], How [GKT07b] and Where [BKT01] provenance formalisms and their relationships between each other according to the formalization of Cheney et al. for the relational data model [CCT09]. To explain the relationships of these terms, we use the example in Figure 2.2. The example contains a database instance I with two binary relations $R(a, b)$ and $S(c, d)$. Furthermore, the SPJU[2] queries $q(x)$ and $r(x, y)$ create the views T and U respectively.

[1]Note, artifacts are not restricted digital items, but can also represent physical objects.
[2]Contains Selections, Projections, Joins, and Unions only.

Note that we annotated all tuples in I such as t_1 for the first tuple in R. In this example, we are interested in how t_8 was created. To do so, we track back the derivation history of t_8 to tuples in I. Particularly, we found two intentions of provenance:

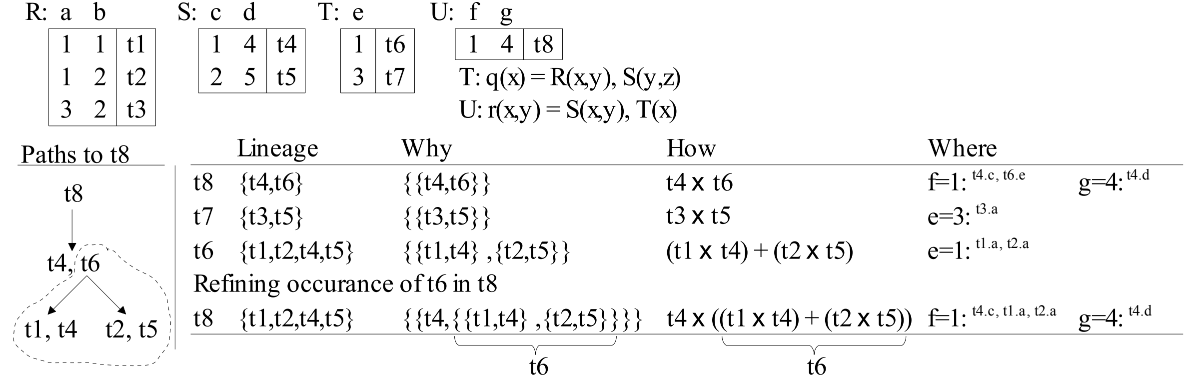


Figure 2.2: Relationship between provenance terms

(A) Existence. *Lineage, Why-*, and *How* provenance explain *why* a tuple is *part of the result* of a query (sequence). Hence, these terms are important for recomputation and validation of query results.

(B) Value origin. The terms *Where* and (partially) *How* provenance show where *attribute values* stem from and *how* they are computed if the tuple exists. Note that How provenance can explain why a tuple exists and in some cases this approach also denotes how values are computed as we explain shortly.

In the following, we explain the basic principles and formalisms of Lineage, Why, How and Where provenance with the help of the example in Figure 2.2 and discuss how the approaches relate to the two intentions.

Lineage. Formally, Lineage $Lin(Q, I, t_n)$ for a SPJU query Q, a DB I and a tuple t_n is a subset of I which is used to create t_n according to Q. For simplicity, we write $Lin(t_n)$ because we refer to the example DB instance and queries. Note that there might be several paths to compute t_n because of the set semantics of the relational model. Thus, a tuple t_k occurs only once in Lin although it might be used in several paths (e.g., as join partner). In the example, $Lin(t_8) = \{t_4, t_6\}$ expresses that these tuples are somehow used to create t_8 without showing any details such as join partners.

Why provenance. In contrast to Lineage, Why provenance $Why(t_n)$ contains a *set of sets* denoting the paths (similar to the routes in [CT06]) that indicate *why* a tuple in the output exists, which is not shown in Lineage. For t_6 in the example, we see that there are two possible paths to create t_6: Joining t_1 and t_4, or joining t_2 and t_5, because of the set semantics. With bag semantics, every $Why(t_{6_n})$

would contain exactly one path. Furthermore, Cheney et al. have shown that the provenance information contained in Lineage are totally contained in Why provenance [CCT09].

How provenance. As in Why provenance, How provenance $How(t_n)$ denotes the paths that indicate *how* a tuple was created in the notion of polynomials, containing the operations $+$ and $\times$. These operations are defined for a set K in a commutative semiring $(K, 0, 1, +, \times)$. Thus, this approach is also known as semiring model. Independent of the semantics of the operations, the polynomial shows the paths creating an output tuple such as $How(t_6) = (t_1 \times t_4) + (t_2 \times t_5)$. Dependent on the semantics of the semiring, it can be used to compute annotation values or even attribute values. For example, it is possible to compute the uncertainty value of tuples. To this end, annotations such as t_8 contain the probability of this tuple of being in a result set in a probabilistic database and the polynomial is mapped to a probability semiring using a semiring homomorphism [CCT09, GKT07b]. Recent work extended How provenance for set minuses [GIT09] and aggregate queries [ADT11]. Moreover, Cheney et al. have shown that provenance information contained in Why provenance are totally contained in How provenance [CCT09].

Where provenance. In contrast to the previous definitions, Where provenance is defined on attribute level (not on tuple level). By definition, *Where* provenance indicates where an attribute value of an output result was *copied* from. For instance, the Where provenance of attribute f in tuple t_8: $Where(t_8.f) = \{t_4.c, t_6.e\}$ denotes that the attribute in $t_8.f$ was copied from $t_4.c$ or $t_6.e$ (both containing the same value). Unfortunately, it is not possible to use Where provenance for non-copy operations (e.g., aggregate functions), which is sometimes possible in How provenance. But it is possible to use Where provenance for operations performing manipulations of the table structure at attribute level (i.e., projection). Thus, unifying them in one formal model is desirable. Although Cheney et al. could not proof that, according to the currently used formal models, *How* and *Where* provenance can be unified in one formal model [CCT09]. However, they emphasized the similarities between the two notions. An approach to unify both notions is presented in [Tan10].

The interesting question that arises for us form this consideration is: Can we exploit the similarity between the single approaches and just interpret the captured provenance data differently in order to address different provenance approaches? This would support the claim for a general provenance capturing solution.

2.3 A hierarchical provenance framework

In this section, we motivate the necessity of abstraction layers to form our hierarchical provenance framework. To this end, we introduce a running example that is used for illustrative purposes in the remainder of this thesis.

2.3.1 Provenance systems

At a very abstract level, systems capturing and evaluating provenance work as depicted in Figure 2.3. Initially, requirements for the whole system such as granularity, time when data is captured, or availability have to be defined (*preparation*). Afterward, the system *captures and stores* the provenance data (also known as annotations), which have to fulfill the previously defined requirements. Depending on how the data is stored, the system has to provide a possibility to *query* the provenance data (e.g., by the use of a query language), taking additional issues such as privacy aspects into account [BSS08].

Moreover, the provenance data are *evaluated* with respect to their validity. For instance, developers can use this information to understand how certain artifacts have been computed, e.g., how a query result evolved. Beyond that, the system can even reintroduce these results into subsequent computation steps, such as computing new query results. However, to validate such results, we may face the problem that the necessary provenance information is fragmentary or unreliable. Thus, missing information has to be somehow estimated or validation is simply impossible. In this thesis, we address the fragmentation and reliability of existing provenance information. In particular, we show that we can use different layers of abstraction for different purposes based on the fragmentation of the available provenance information. Furthermore, we show that reliability affects all of these layers and thus, can be considered as *cross-cutting* characteristics of provenance.

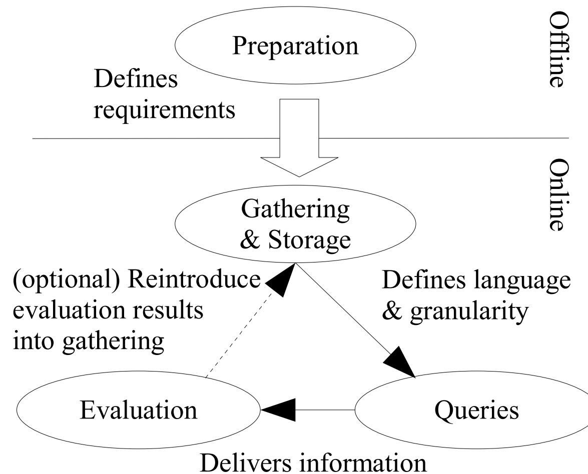


Figure 2.3: Provenance system

2.3.2 Necessity for abstraction layers

Due to the versatile characteristics of provenance systems, we build our framework on differences regarding uncertainty and fragmentation represented by layers of abstraction. We motivate the necessity for every layer shortly. Particularly, the single layers form a hierarchical granularity framework for provenance where each layer complements the preceding one in the case that additional, more fine-grained information is available.

Running example: Fingerprint recognition

For motivating such a hierarchical framework as well as clarifying our intention, we introduce a running example. In a current research project[3], we evaluate digital pattern recognition techniques for latent fingerprints based on high-dimensional sensor data. Additionally, we are interested into testing different contactless sensor devices at different materials as well as determining environmental influences. To ensure validity of the evaluation, we have to gather detailed knowledge about the derivation history of all data items originally produced by the sensor(s) and subsequent quality enhancement steps.

For this scenario, we want to be able to use the fingerprints in law enforcement proceedings. To this end, we have to provide detailed and reliable information about the whole processes from scanning the fingerprint to the final results presented at court. However, we have to take into account that the people in court do not have detailed technical background.

1. Consequently, we need a layer that *abstracts* from execution and implementation details. As a result, we can show the complete causal dependencies from input data (the original fingerprint scan) to the results presented in court and, beyond that, the (possibly certified) processes that created the respective results/data.

2. In contrast to the first layer, fingerprint experts and software designers require more detailed knowledge about the particular processing steps. Specifically, they have to be able to *recompute* and therefore *validate* foreign results (e.g., when a lawyer doubts evidences). But these people still do not need detailed information *how* exactly the single results were computed.

3. However, having detailed knowledge on *how* exactly the single results were computed is important for software engineers implementing the single computation steps. For instance, for quality enhancement of latent fingerprints, we use different filter implementations, such as Gabor filter banks. These filters modify the single pixels of the original images (scan data), which, in turn, represent the lowest level of granularity. Consequently, a developer has to know how exactly the filter modifies the pixel values to implement, improve, or choose the right filter.

4. Finally, there are additional factors, which we have to take into account. These factors are beyond mathematical descriptions and contain details of hardware architecture, storage, and transport. Hence, they complement the information of the previous layers to *improve the reliability* of both, the data itself and the respective provenance information of all preceding layers. We discuss whether this is a real layer or a different dimension in Section 2.7.

[3] https://omen.cs.uni-magdeburg.de/digi-dak

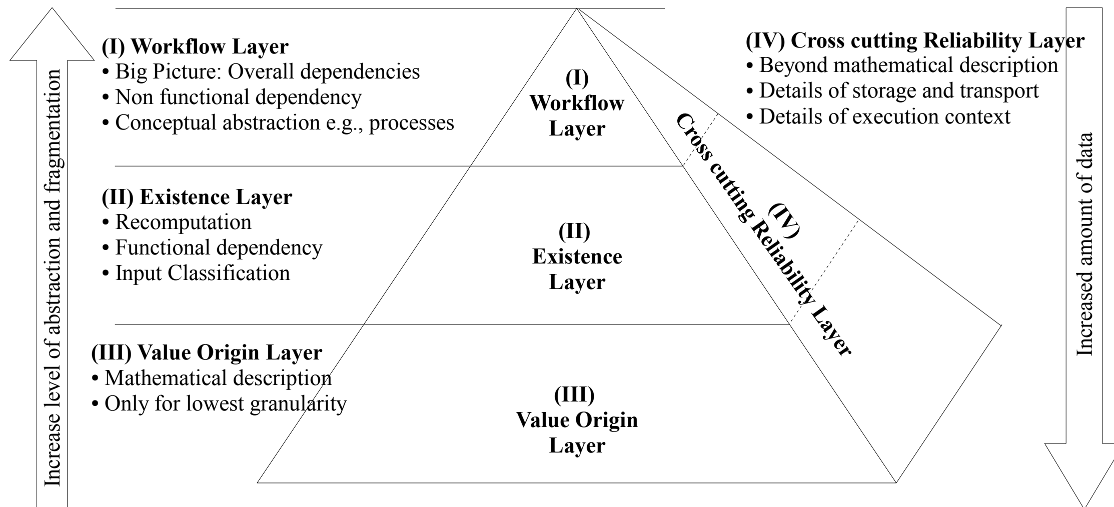


Figure 2.4: Hierarchical provenance framework - First published in [SSS12a]

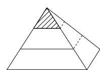
Granularity layers in the real world

To avoid overfitting of our hierarchical framework regarding the fingerprint example, we searched the literature for concepts and approaches addressing and generalizing the single layers. Additionally, we took existing approaches as base for each layer and describe the relationships among them. Then, we assembled the results in a hierarchical framework depicted in Figure 2.4, which is based on the current state of the art. The top most level contains the Workflow provenance. Several approaches, such as the OPM [MCF+11], basically rely on conceptual abstraction and thus, describe only causal dependencies independent of the implementation.

In contrast to the first layer, the second requires functional dependencies to recompute and validate results. For this layer we found Lineage [CW00], Why provenance [BKT01], How provenance [BKT01, GKT07b] and causality [MGH+10] as existing approaches. For the third layer, Buneman suggested a model explaining for copy operation, where the single attribute values of an existing tuple stem from [BKT01]. Next, the semirings in How provenance can be used for computing attribute and annotation values for different subsets of database operations. Consequently, this lays the foundation for the third layer of our framework. For layer IV, the basic idea is about increasing (and securing) the reliability of the provenance data of the previous layers such as proposed by Braun et al. [BSS08]. We give a detailed and comprehensive overview for all layers and their relation to each other in the next section.

In the next four sections, we introduce the four abstraction layers and their relationships to each other. Additionally, we point out their usage and possible refinements for each layer.

2.4 Workflow layer



This layer is a conceptual abstraction, representing what happens at coarse-grained level totally independent of implementation (if there is one). Hence, this layer shows causal dependencies while hiding details about functionality. Moreover, this is a starting point for validating, comparing, or discussing results. Finally, this layer can be used for automatic validation of derivation histories, which might be produced in distributed systems not having access to each other.

2.4.1 Extended open provenance model

In the workflow layer, we rely on the OPM [MCF+11], that is developed to be a pure conceptual abstraction of provenance information. We rely on the OPM, because it is a consensus explicitly designed for workflow provenance. Moreover, most of literature addressing workflow provenance such as [ABC+10, MBK+09, SPG08] refers to a directed acyclic graph and is therefore similar to the OPM. To bridge the gap between coarse-grained workflow provenance and fine-grained forms of provenance [CCF+09], we present an extension of the OPM in this section. Finally, we explain usage and limitations of this layer.

As already mentioned in Section 2.2.1, we use additional artifact types to have more semantics in the model. Within our extended OPM, an artifact has (1) a name, (2) a role, (3) a type showing whether this is a complex or a simple artifact and furthermore a unique id as well as a mark denoting if this is an initial or result artifact (cf. Figure 2.5). For initial artifacts, we have *no* knowledge about previous causal dependencies, but about subsequent ones. Hence, backtracking of one of these artifacts is impossible. In contrast, for result artifacts we have knowledge about previous causal dependencies, but not about subsequent ones due to the fragmentation of provenance. Thus, these are the last artifacts for which we can form the dependency graph. Similar to artifacts, each process has (1) a name, (2) a hierarchy label, and (3) a type.

2.4.2 Graph refinements

Regarding our extended OPM, the most important extension is the introduction of complex artifacts and complex processes allowing stepwise refinement of granularity. We argue that this is the key issue to bridge the semantic gap between coarse-grained workflow provenance and different fine-grained provenance models (accumulated in the subsequent layers). In fact, it is possible to refine the processes of the *workflow* layer until reaching the connection points to the *existence* layer. In the following, we introduce our extension of the OPM and how this allows for stepwise refinement.

Complex artifacts

In Figure 2.5 a, we depict our concept of stepwise refinement used for complex artifacts. A complex artifact such as the *Material DB* contains a certain number of either complex or simple artifacts. This forms a *tree structure*, where the root of the tree is the top

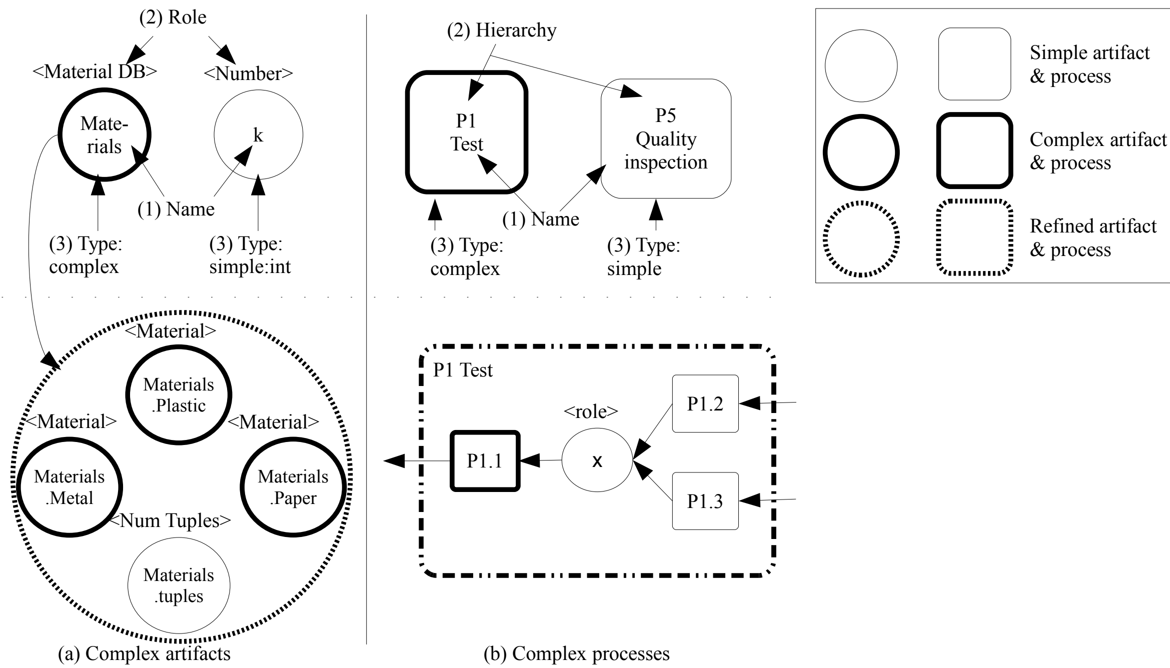


Figure 2.5: Complex artifacts and processes

most complex artifact (in this case the *Material DB*), inner nodes are again complex artifacts and the leaves are simple artifacts, directly linked to a primitive value (e.g., an integer). The position in the tree is inherited in the name of node. For instance, the number of tuples in the *Material DB* is referenced as *Materials.tuples*. This is similar to object-oriented programming where this notion is used to reference fields or methods. A complex artifact inherits its data such as tuples of a DB table or members of an object. Furthermore, such artifacts inherit *additional* annotations that contain meta data about the artifact itself. Note that due to the fragmentation of provenance or due to privacy issues [BSS08] not all data may be available (i.e., several nodes might be missing).

Complex processes

Analogously to complex artifacts, we also introduce complex processes for stepwise refinement (cf. Figure 2.5 b). The refinement of a process is an acyclic graph showing the causal dependencies of the parent process in more detail. To reference the parent process, the process contains a hierarchy label. For instance, process P1.1 is within the refinement graph of P1. Basically, the process has to collect provenance data on its own or may use integrated monitors [LM10]. Moreover, this data is always collected at the lowest level of granularity, that is, the leaf level of the tree structure. Consequently, all upper levels are aggregations of this most detailed level.

Implicit dependencies

To allow processes the usage of artifacts with different levels of granularity in one graph (i.e., only some artifacts are refined), we introduce the concept of *implicit dependencies*.

For instance in Figure 2.6, *process 1.2* uses artifact *a.2.2*. As a result of this *explicit* dependency, there are additional implicit *use dependencies* from P1.2 to every ancestor node in the *artifact tree of a*. Hence, P1.2 has implicit a *use dependency* with *a.2* and *a*. Moreover, we add additional *use dependencies* from *a.2.2* to every parent process of *P1.2*. Thus, a.2.2 has a *use dependency* with *P1*. Finally, this is repeated for every pair of parent artifact and parent process recursively, up to the lowest granularity level. Therefore, a and $P1$ have a *use dependency* too.

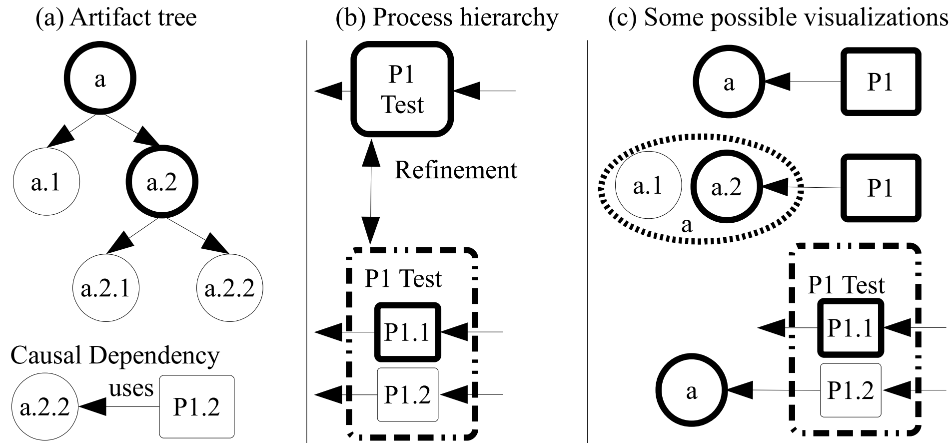


Figure 2.6: Visualization of implicit dependencies

2.4.3 Usage and limitations

To explain the usage and limitations of this layer, we refer to the motivating example from Section 2.3.2. In Figure 2.7[4], we depict the provenance graph for two result artifacts, namely quality assessment and age interval of a fingerprint. Both artifacts are final results and thus, we have no knowledge about any subsequent causal dependencies. Hence, we cannot build any graph using these artifacts as input. However, the graph reveals *where* the initial fingerprint was taken from and *how* the results have been created at a very coarse-grained level hiding lots of possibly important details.

Backtracking

In the example, the provenance graph exhibits its maximum level of backtracking. This means that we tracked back *all* dependencies to initial artifacts, where backtracking is defined as follows: First, the level of backtracking for each vertex in the graph used as starting point is defined as *Level 0*. For example in Figure 2.8, Level 0 contains only the final result of role `quality`. When increasing the backtracking level, we expand the graph for each causal dependency until we reach some process. At the same time, we add all artifacts and agents having a direct dependency with this process to the

[4]Note that we omitted labeling the causal dependencies for clarity reasons. To avoid ambiguous dependencies, we use only *use* (from process to artifact), *wasProducedBy* (artifact to process), and *wasTriggeredBy* (process or agent to process) in this example.

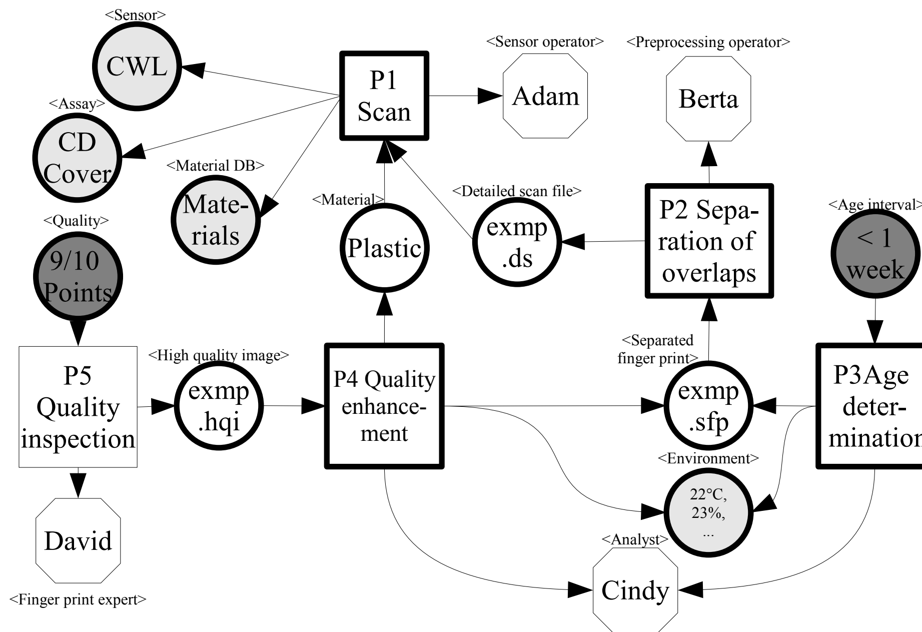


Figure 2.7: Provenance graph containing complex artifacts and processes

backtracking level. In Figure 2.8, we depict the first three backtracking levels for the *quality artifact* of the example. We argue that stepwise backtracking is important to simplify understanding of these graphs because provenance graphs can be very large and therefore hard to understand.

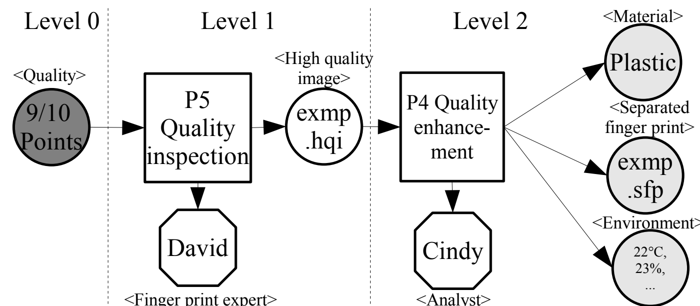


Figure 2.8: Exemplary backtracking levels

Graph merging

Unfortunately, by simple backtracking it is *impossible* to create the graph in Figure 2.7. This is the case because both result artifacts (quality and age interval) use the *separated fingerprint* artifact as input. Consequently, there is no path connecting both result artifacts in one graph. As a result, by backtracking both artifacts of our example, we obtain two provenance graphs that can be merged when backtracking reaches *Level 2* for the *quality artifact* and the *age interval* artifact. Thus, merging graphs is important to a) show whether some artifacts share a common derivation history (i.e., share a subgraph) and b) reconstruct graphs based on the derivation history of artifacts.

Granularity refinement

Except for process *P5 Quality Inspection*, all processes in our example in Figure 2.7 are of type *complex* and thus can be refined. For instance, in Figure 2.9 *P1 Scan* is refined, that is, the complex process from Figure 2.7 is replaced by its corresponding subgraph. Due to the concept of implicit dependencies, we can have both, complex and refined processes, in one provenance graph and thus show details that are not available on the coarse-grained level of granularity. For instance, our refined process in Figure 2.9 reveals that the whole scanning process performs two scans. First, *Scanner Operator Adam* triggered a coarse scan (P1.1) that was used to find *regions of interest* possibly containing a fingerprint pattern. Second, the sensor operator triggered a detailed scan (P1.4) using the information obtained from the coarse scan (P1.1). Moreover, the graph depicts that P1.1 triggers P1.2 and P1.3 automatically.

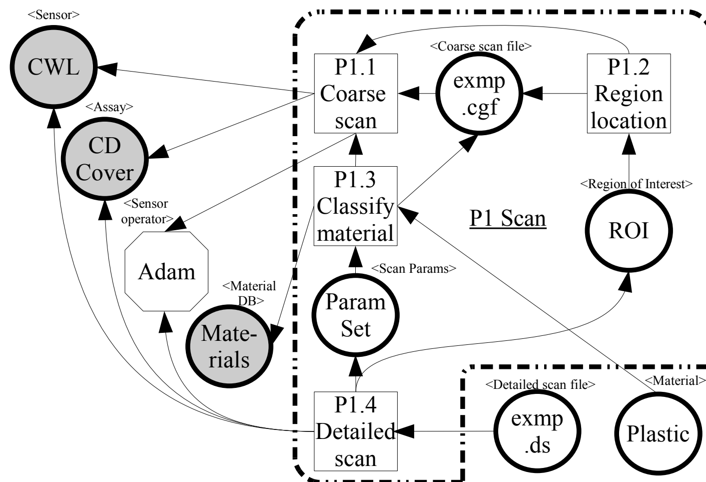


Figure 2.9: Process refinement

Automatic validation

The provenance graph can be used for automatic validation. To this end, we have to specify for each process a list for certain valid combinations of input roles and resulting output roles (or a negative list respectively). Moreover, it is possible to define certain constraints such as the number of output artifacts has to be larger than the number of input artifacts etc. Consequently, a monitoring program having access to the provenance data and to these constraints can detect anomalies.

Limitations

In contrast to the next layer, we do not restrict processes to functions. This means that providing the same input *does not necessarily* have to result in the same output[5].

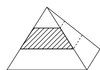
[5]Here, identical artifact means that the artifact trees have the same structure and the primitive value(s) within an artifact contain the same value(s), excluding annotations.

Reasons therefore are missing input parameters such as configuration parameters or class members. For instance, in P5 *Quality Inspection* a fingerprint expert evaluates the quality of the fingerprint image. Although we assume that he is an expert, different environmental factors such as different light settings in different laboratories might change the quality estimation result especially for borderline images. Consequently, to reproduce results we have to assume functional dependencies that we cannot express with processes as conceptual abstraction. Furthermore, in some cases, such as manual quality inspection, it may be impossible to guarantee functional dependencies. Consequently, we argue that approximating the functional dependency with refinements is practical and sufficient and better than having no provenance at all. To determine the similarity of artifacts of the same role we suggest the use of distance metrics as applied in [DGGH11].

2.4.4 Related approaches

Biton et al. use a similar concept to create *user views for arbitrary scientific workflows*. First proposed in [BCD07] and then improved to avoid to introduce loops in the workflow [BCDH08], it allows to merge composite modules (similar to our complex processes) to reduce the amount of provenance data presented to a user. As this model was developed to visualize and query previously captured provenance data, not to determine what provenance data to capture, the focus is slightly different. However, the similarity is in the way how to link different levels of granularity. In contrast to our approach, Biton et al. only refine processes (not artifacts) resulting in a complex merge procedure, which possibly forbids merging several processes. A solution therefore are our implicit dependencies allowing us to more flexibly refine processes if only parts of an artifact (e.g., a tuple in a table) is used. Moreover, to build the provenance graph, we use an inverse temporal order (from output to input, not vice versa), which is a tribute to the fragmentary nature of provenance and the application scenario in that we want to determine the past derivation history and not the future use. In fact, a result cannot know what it is used for in future, but it may know which artifacts (or respective subsets) where used to create the artifact itself. As a result our approach is more flexible, as we do not assume that in the graph there is one (complete) input node, but for instance also accepts hidden inputs such as state variables from a previous run having a certain impact on the current result building process. Finally our artifact trees may also contain the primitive values (if known) and are thus not restricted to simple id's written in the log files used to create the graph in [BCDH08].

2.5 Existence layer



While the previous layer addressed causal dependencies, this layer focuses on result validation. Therefore, we only consider artifacts a computation step created and thus *exist*. In turn, we do not consider artifacts that could have been created potentially (e.g., tuples that do not find a join partner), relevant for instance for query non-answers. To allow result verification, we assume a *functional dependency* for artifact creation: $(f : a^n \rightarrow a^m)$. Consequently, we can recompute and thus validate

results. Moreover, depending on the available information regarding the behavior of the single functions, we split the existence layer into three sub levels. In the following we explain the three sub levels, point out how refinement takes place in this layer and its limitations.

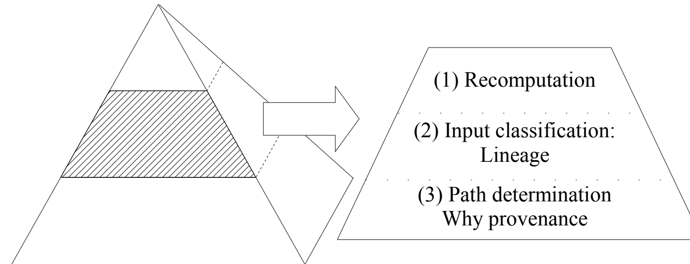


Figure 2.10: Sub levels of the existence layer

2.5.1 Sub levels of the existence layer

Subsequently, we explain the purpose, use and relationship between the three sub levels of this layer.

Sub level 1: Recomputation

The first sub level allows recomputation and thus validation of results. To recompute the result(s), we have to ensure that the input of the single functions in this layer is identical, which is not always a trivial task. Currently, most authors assume that we know the input. For example, in DB formal approaches such as How [GKT07b] and Why provenance [BKT01], the authors implicitly assume that all input (relations) are known. In the relational model, knowing the query also means knowing the input, because all input relations are part of the `FROM` clause(s). But in different data models or programming paradigms this is not as simple. While in functional programming the input are the arguments in the function call, in object-oriented programming we additionally have to take into account class members and different static variables. Consequently, knowing the input is the minimum requirement for recomputation.

Sub level 2: Classifying the input

Splitting the input into possibly necessary (*endogenous*) input and never necessary (*exogenous*) tuples can improve the understanding of what actually happened during computation. Again, we start with DB formalisms and explain which of these definitions can be adapted to different data models and programming paradigms. According to Meliou et al. [MGH+10, MGMS10], a set of input tuples I for some query $R = q(I)$ producing an output tuple t_i consists of endogenous tuples I^e (tuples used in at least one path) and exogenous tuples I^x (not used in any of the paths). Both, I^e and I^x are subsets of I, their intersection $I^e \cap I^x = \emptyset$ is empty, and their union restores $I = I^e \cup I^x$. Moreover, the authors differentiate endogenous tuples into counterfactual and actual

tuples due to the set semantics that can lead to multiple paths producing one output tuple.

Explanations for set semantic. Consider the following join example of relations R and S: $\{a\} = q(x) = R(x), S(x,z)$ over an instance (I): $R\{a,b\}, S\{(a,b),(a,c)\}$. Obviously, the tuple $R(b)$ forms I^x because there is no join partner in S. But there are two ways to compute the query result: (1) $R(a) \bowtie S(a,b)$ or (2) $R(a) \bowtie S(a,c)$. Therefore, R(a) is a counterfactual tuple $t \in I_c^e$ because removing R(a) from I would remove $\{a\}$ from the query result. According to Meliou et al. [MGH+10, MGMS10], a tuple t is part of the set of counterfactual tuples I_c^e in case there is a tuple r in the result R, which is removed from the result if t is removed from the set of input tuples. Formally, I_c^e consists of $\forall t \in I^e$ for that $\exists r \in R$ in such a way that for query q the execution of q without t resulting in $R' = q(I^e - \{t\})$ causes that r is not part of the query result $r \notin R'$. In contrast, we can remove either $S(a,b)$ or $S(a,c)$ from I^e because there is still one path creating the same result. Consequently, these tuples are actual tuples because they can be part of a certain subset $(\Gamma \subseteq I^e)$, which can be removed from the input while still producing the tuple r w.r.t to query q. Note, that Meliou et al. relate *causality* based on the definition of Halpern and Pearl [HP05] to *How provenance*. We stick to their results. But according to our framework, causality is finding all possible inputs which might have produced a result tuple r_n w.r.t. q. Hence, causality is important if we know only parts of the input and want to compute (all) possible inputs or query non-answers. By contrast, in linage we assume we know the input and want to validate what happened. Consequently, $\forall t_i \in q(I)$ I^e is equivalent to $Lin(q,I,t_i)$ and $I^x = I - Lin(q,I,t_i)$.

Adaption to different data models

When adopting these definitions to different data models without set semantics, we can omit the differentiation between actual and counterfactual input artifacts, because all actual inputs are also counterfactual. This simplifies the classification of input determined at the previous sub level, because we can simply omit this differentiation. Unfortunately, currently there are no formalisms automatically computing the input classification for different data models, such as they exist for the relational data model. Consequently, we have to manually collect these annotations in the program itself.

Nevertheless, this sub level improves understanding the relation between input data and result computation (query equivalence) as follows:

- Every input artifact in I_c^e *cannot* be removed without changing the result of the operation.
- Every input artifact in I^x *can* be removed without changing the result of the operation.
- (Only with set semantics) An input artifact $\in I^e$ but $\notin I_c^e$ can be removed from I without changing the result, because there is at least one path not using this tuple to produce the same result.

Sub level 3: Determining paths

As we stated in Section 2.2.2, a *path* is an acyclic graph showing the sequence of operation (vertices) and respective connections (edges) to input artifacts. In set semantics, multiple paths may lead to *one* result and thus paths are highly related to minimal witness bases [BKT01, MGH+10]. Furthermore, formalisms such as Why and How provenance denote the computation paths. The main difference to the previous sub level is, that an artifact may occur at different nodes in the graph (see Figure 2.2).

In summary, the former two sub levels are approximation of the path determination level. We need these approximations because of fragmentary knowledge about implementation details. For example, in API programming or for aggregate functions in databases, we only know the function name and the arguments to supply. Therefore, we have to assume that all of the input artifacts are counterfactual ($I = I^e_c$) and consequently no exogenous artifacts exist. This allows to recompute and consequently validate results. In the second sublevel, we classify the input (if possible due to our knowledge) to know which input artifacts (I^x) can be omitted without changing the result. Furthermore, we determine the existence of multiple paths ($I^e - I^e_c \neq \{\emptyset\}$), artifacts contained in every path (I^e_c) and artifacts which might be part of a certain contingency ($I^e - I^e_c$). However, in the third sublevel, we determine paths themselves. Each path p in the set of paths P ($p \in P$) w.r.t. a specific operation (e.g., query) o and Input I creating result r contains all counterfactual artifacts in I^e_c and (a possibly empty) subset of non-counterfactual artifacts $I^e_{c^{-1}} = I^e - I^e_c$. Consequently, the following equations hold: The union of all artifact sets of all paths is equivalent to the set of endogenous input artifacts $\bigcup_{\forall a \in p \in P} = I^e$. Moreover, the intersection of all artifact sets of a path is equivalent to the set of counterfactual artifacts $\bigcap_{\forall a \in p \in P} = I^e_c$. Finally, the set minus of all non-counterfactual artifacts $I^e_{c^{-1}}$ and a specific path produces a certain contingency $\Gamma = I^e_{c^{-1}} - p$. Obviously, a contingency is a set of endogenous artifacts that we can remove from I^e so that exactly one path remains for creating the result r.

Refinements

At the existence layer, there are two possible refinements at each sub level showing different results: (A) Backtracking of artifacts and (B) Implementation refinement. The limitations of both refinements are highly related to the fragmentation of our knowledge as we explain in the following.

From results to initial artifacts. For each of the sub levels it is possible to *backtrack* artifact occurrences in the same way as with the previous layer (cf. Section 2.4.2). The backtracking stops when reaching an initial artifact, which is the first element we have provenance information for. In a nutshell, backtracking operations can be summarized as follows: Union of the whole input sets for sub level one, union of endogenous input sets I^e in sub level two, and merging of directed acyclic graphs producing a new directed acyclic graph in the third sub level.

Refining implementation details. This kind of refinement requires implementation details such as additional knowledge of the internal structure of functions. For instance,

Table 2.1: Refinement with fragmentary knowledge

Fragmentary Knowledge	SubLevel 1	SubLevel 2	SubLevel 3
1 r=foo(a,b,c,d);	$I=\{a,b,c,d\}$	$I=\{a,b,c,d\}$	r ↓ a,b,c,d] foo
1 int foo(a,b,c,d) 2 { 3 ret=**foo2**(a,b); 5 ret+=c; 6 return ret; 7 }	$I=\{a,b,c,d\}$	$I^e=\{a,b,c\}$ $I^x=\{d\}$	r ↓ ret, c ↓ foo2 [a,b] foo
1 int foo(a,b,c,d) 2 { 3 ret=**min**(a,b); 5 ret+=c; 6 return ret; 7 }	$I=\{a,b,c,d\}$	$\begin{pmatrix} I^e=\{a,c\} \\ I^x=\{b,d\} \end{pmatrix}$ or $\begin{pmatrix} I^e=\{b,c\} \\ I^x=\{a,d\} \end{pmatrix}$	r ↓ ret, c ↓ a] or r ↓ ret, c ↓ min [b] foo

a function can call several functions (e.g., `foo2` and `min()` in Table 2.1) and operations such as + or -. To explain this refinement and the influence of fragmentary knowledge, we refer to the example depicted in Table 2.1. In this example, there is a function `foo()` taking four arguments as Input I (for simplicity we omit argument types) and returning one (result) artifact. In the first part of the example, we assume that we do not have detailed knowledge about this function. Hence, we cannot classify the input leading to one path, assuming that all of the input is counterfactual. In the second part of the example, we assume that we have additional knowledge about the internal structure of `foo()`. Note that we still have fragmentary knowledge, because we do not know what `foo2()` in Line 2 calculates. Hence, it is possible to *refine* implementation details by revealing the sequence of functions and inputs in the example. To keep the reference to the containing functions such as `foo()`, (sub) paths (i.e., the vertices) are annotated with the function they belong to. The refinement stops when reaching operations only or because of fragmentary knowledge about functions. In the example, we do not know what `foo2()` actually does and thus we have to approximate I^e from $I = \{a, b\}$ (i.e., we do not know whether this function actually uses the arguments). This introduces uncertainty into our provenance information. However, with additional knowledge about `foo2()`, we can calculate I^e. For instance, if it works like a min function (i.e., is equivalent to an operation), we cannot further refine the implementation, but it is possible to determine the path according to the input I.
The advantage in databases is that, for monotone queries, we have formalisms that automatically and thus without uncertainty compute the following details:

- For sub level 2, the set of necessary input tuples I^e: Lineage, Why provenance and How provenance,
- For sub level 3, the set of all paths: Why provenance and How provenance.

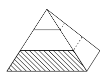
Link to parent layer. As recently introduced, a function that is called within the body of another function such as `foo2()` from `foo()` has an annotation to keep the reference to the calling function. In contrast, the function `foo()` in Table 2.1 is the top most function and thus does not contain a link to a calling function. However, such functions have a link to the abstract process in the preceding layer showing the respective part of its computation for a particular artifact. Moreover, these functions may also consume artifacts from the workflow layer.

The combination of both, refinements and the link to the workflow layer, allows us to increase our understanding of what actually happens. Note that especially the refinements require proper tool support to visualize certain excerpts from the derivation history.

2.5.2 Limitations of this Layer

In the example in Table 2.1, we did not consider whether the arguments supplied to `foo()` are primitive numeric values or complex objects. However, at this layer we do not care *how* the primitive values within an artifact or its annotations (if it is complex) are calculated. This is restricted to the semantics of the functions and operations denoting to what extent the primitive values of artifacts contributed to a result. This is in the scope of the next layer.

2.6 Value origin layer



With the previous layer, we focused on the determination of paths responsible for the existence of artifacts. In contrast, this layer is about the origin of the primitive values within existing artifacts such as attribute values in the relational data model. In the following, we explain how the origin of values can be addressed by existing approaches and applications for different data models and the relationship to the *existence* layer.

2.6.1 Structure of artifacts

As mentioned in Section 2.4.2, we differentiate between complex and simple artifacts. Simple artifacts are directly related to one primitive value such as an *int* or *char* type. In contrast, complex artifacts may contain several complex and simple artifacts forming a tree structure, where the leaf nodes are always simple artifacts. Consequently, each artifact is a container for primitive values at the lowest level of granularity. For instance, consider a relation containing *regions of interest* (ROI) from our fingerprint example. By our means, a ROI is a rectangular part of a coarse scan probably containing a fingerprint. This ROI is input for the detailed scan process. It is used to scan the physical object

again with higher resolution showing details such as sweat pores that are not available in the coarse scan. For our illustrative example we want to increase productivity. To this end, we want to scan those physical objects in detail first that carry the largest amount of fingerprints. Consequently, we need to store additional annotations representing this information.
In Table 2.2, we show the relation storing the ROIs permanently. Each tuple has a surrogate primary key from some sequence within the DBMS, two points denoting the left upper and right lower bounds of the rectangle, and a foreign key referencing the coarse scan file of the physical object. Moreover, there are general additional annotations for every artifact containing the unique artifact ID, name, role and type (cf. Section 2.4.2). Finally, for each role there are role specific annotations (cf. Table 2.2 and Figure 2.11).

Table 2.2: Region of interest relation

ROI	PK: int	l_u: Point		r_B: Point		FK: int		Role specific annotations	
		x: int	y:int	x: int	y: int			Probability	ROI Locator
t1	1	12	49	287	413	5	→	0.85	v.0.7.3a
t2	2	12543	736	12781	1052	5	→	0.91	v.0.7.3a

2.6.2 Value origin with existing approaches

Based on the structure of artifacts, we are now interested how we can determine the value origin of certain artifacts. The value origin of some simple artifact A w.r.t. an operation O for a path P shows *how* a particular subset of the primitive values within the artifacts contribute to A. For instance, consider the query Q to get all ROIs in Figure 2.11. Furthermore, assume for simplicity that there are only two ROIs for scan five. For normal computation, the result would look as follows: $t3 : \{(5, 2)\} = Q(ROI))$. In contrast, when including provenance in this scenario, the results look as depicted in Figure 2.11. The result also contains general annotations as well as role-specific annotations. The role-specific annotations are computed based on the input tuples and the respective semiring. Meanwhile, recapitulate that we want to scan those objects first carrying the greatest amount of fingerprints to increase productivity. Thus, we additionally store the probability that there is at least one fingerprint on the physical object and the ROI Locator Version. Then, the probability is computed based on the detection probability of all tuples contributing to this result and the representation in How provenance. For the example, the contributing tuples are $t1$ and $t2$. Hence, the probability is computed as follows: $p(t1)\ or\ p(t2) = (0.85+0.91)-(0.85\times 0.91) = 0.9865$. Now, we are interested into approaches explaining the value origin of the query result of Q including the respective annotations.

Where provenance

In databases, the *Where provenance* formalism is able to determine the origin for pure copy operations only [BKT01]. By definition, this works for the data such as the attribute FK, because this is what it was designed for. In contrast, when using Where provenance

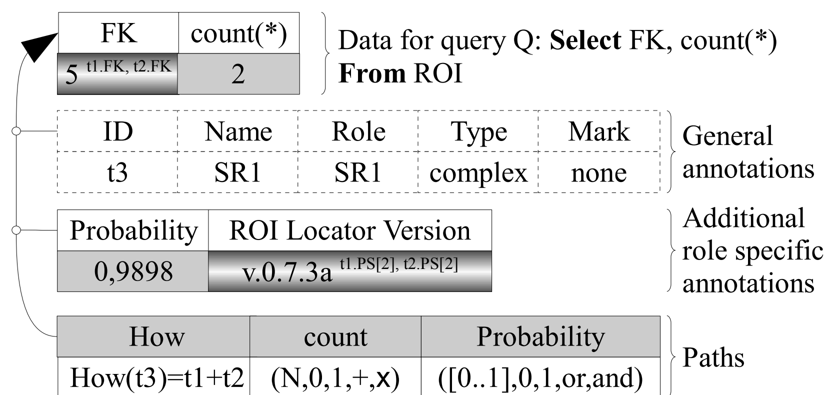


Figure 2.11: Computation with provenance

for annotations we face the problem that there may be competitive annotations. For instance, imagine that the first ROI was computed with ROI Locator Version v.0.7.3a while the second was determined by v.0.7.3. In this case, we currently add the symbol a showing that the version is ambiguous (which must not happen). Because this is a domain-specific solution, we consider different solutions allowing annotations to collect sets (both versions) or rules to determine the dominant annotations (e.g., the later version if there are no functional changes, but only an increase of performance).

Semiring model and extensions

In contrast to Where provenance, we can use the semiring model to calculate the results of queries (e.g., for attribute or annotation values). In the original version this works for SPJU queries [GKT07b] and has been extended for set minuses [GIT09]. Finally, Amsterdamer et al. extended the model for aggregation support including optional group by operations with some limitations [ADT11]. The general challenge in this model is finding the semiring with the right semantics as in the probability example [CCT09]. Moreover, it is currently not possible to determine the origin of the PK in our example, because it is linked to a sequence. To determine the origin, the model has to have access either to the sequence or to the previously created tuples (requiring an order). Finally, to the best of our knowledge it is not possible to use constants.

Usage for different data models

While *Where provenance* and the original semiring model are limited to databases, Amsterdamer et al. emphasize that the extension for aggregate queries is independent of the data model and therefore might be used to capture *automatically* provenance information for aggregate functions. However, the concepts behind Where and How provenance are also suitable for different data models.

A first formal approach for functional languages is introduced by Buneman et al. in [BCK12]. The approach also allows hierarchies in the provenance graph for refinements (zoom in and out). However, the approach currently assumes to have full knowledge on

the implementation (i.e., the semantics of the functions) and does not consider complex artifacts. As a result, it is not applicable when having *fragmentary* knowledge on the computation (black box processes). Nevertheless, the outlined provenance language aims at creating a strong formal basis addressing the first three levels of our provenance framework. Furthermore, the future provenance language shall also able to answer causality related queries (e.g., by considering if conditions) and thus very powerful. In prior work [SSS12b], we suggest to facilitate the decomposition of object-oriented programs for provenance capturing. In this approach methods and constructors are equivalent to processes and method parameters as well as class members are artifacts. Then, we create hierarchical provenance graphs, for instance by binding aspects from aspect-oriented programming to the computation steps (method executions) we are interested in. This way our approach is very flexible. For instance, we can mask input artifacts (e.g., passwords) and intermediate processing steps, allowing fine-grained tuning of the desired provenance capturing functionality. In contrast, in our approach we usually cannot provide the same semantics (Level 3) as intended by [BCK12], for instance due to invisible input. This input includes, for instance, random variables, static class members, and pointer arithmetic. This problem, however results of the challenging mapping of *all* computation steps to processes and determination *all* input artifacts, which is currently an unsolved problem. A different approach to instrument program structures for provenance capturing is using modified compilers [TAG12]. The benefit of this approach is that it is totally transparent (i.e., requires only the modified compiler). However, it does not allow fine-grained tuning of the provenance capturing nor is it currently applicable for complex artifacts (e.g., objects).

2.6.3 Relationship to the previous layer

Independent of the way how we determine the value origin, there are several relationships to the previous layer. First, for each primitive artifact in every path there is one mathematical description. In *Where* provenance these are the locations, the primitive values are copied from. In the semiring model this is a series of operations w.r.t. to a particular semiring. Second, all simple artifacts used in one mathematical description are part of the corresponding path [CCT09].

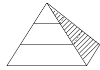
Backtracking refinement

In contrast to all previous layers, there is only one possibility for refinements, because the value origin is always defined on the lowest level of granularity. Consequently, there is no granularity refinement. However, backtracking is still possible by means of replacing a simple artifact in the mathematical description. For instance, in the initial example in Figure 2.2, the Where provenance of attribute $t8.f$ denoted as $Where(t8.f)$ are the locations $t4.c, t6.e$. Now we can replace the location $t6.e$ with $Where(t6.e) = \{t1.a, t2.a\}$ (i.e., remove $t6.e$ and add the result $\{t1.a, t2.a\}$). Limitations of backtracking are related to fragmentation. In contrast to the previous layer, we require detailed knowledge about how exactly the simple artifacts contribute to the result of the computation and cannot estimate this information. Consequently, further backtracking is not possible if we do not have this knowledge for an operation in the path.

Limitations of this Layer

As this layer works on mathematical description, it reaches its limitations for instance when details of the (hardware) dependent execution context (e.g., for timing experiments) are required or even when trying to prevent or detect possible attacks.

2.7 Cross-cutting reliability layer



As all previous layers are dealing with granularity and fragmentation, this layer addresses the reliability of artifacts itself and respective provenance data. As a result, this layer decreases the degree of *uncertainty*. Because this is possible for all previous abstraction layers, we call this layer *cross cutting*.

2.7.1 Reliability: A different dimension of provenance

Particularly, we consider this layer as a different dimension, which is partially orthogonal to all other layers, as fragmentation may cause uncertainty as well (see Section 2.5.1). Hence, we keep calling this dimension of provenance a *layer*.
The purpose of all preceding layers was to improve our understanding of what happened (e.g., for validating results) and are restricted to our fragmentary knowledge. Unfortunately, this information can be wrong for arbitrary reasons. For instance, a malicious attacker could have changed the data itself or corresponding annotations. Moreover, there can simply be errors when collecting the information. This is especially the case when there is no possibility of capturing the information automatically (e.g., by a formalism). Consequently, this layer addresses reliability and trust in provenance information [LLLS10, LM10, MBM+10].

Challenge: Increasing amount of data

Consider the example from Table 2.2. To be really sure that all ROIs are calculated with the same ROI Locator version, we furthermore annotate the version annotation with a security hash sum of the version binary. Moreover, to ensure that this artifact remains unchanged, we have to add an additional signature computed over the whole artifact (including annotations).

The cross-cutting characteristic

In the recent example, we included a signature to ensure that a specific artifact remains unchanged. Actually, we want to do this for all provenance data collected in the preceding layers such as the graphs in the Workflow Layer or the paths in the Existence Layer. Moreover, we may want to propagate that we secured the execution context for instance with TPM modules as suggested in [LM10]. But this is not only related to security issues. For instance, for some application scenarios we need a result within a certain amount of time. Therefore, we perform several timing experiments using different algorithms for computation of ROIs. To compare these results, we need to know about the hardware environment in case that the tests are not performed on the same computer. As a result, the amount of data to store increases rapidly, which is also denoted by area of the pyramid slice in the framework visualization (cf. Figure 2.4).

Annotating annotations

An interesting problem is that annotating annotations furthermore increase the amount of data. As mentioned previously, we want to include additional annotations to increase the reliability of annotations or values. Suppose we want also know which hashing algorithm was used, in which implementation, and how these additional annotations have been collected. The problem is to determine when to stop annotating, because when doing this for every piece of data (i.e., every artifact, annotation) the amount of annotation data is a multitude of the values within the simple artifacts (e.g., in Figure 2.11) [CCT09].

2.7.2 Current research on reliable provenance

Although reliability is an increasingly interesting topic, only few works exist that addresses this layer. For instance, McDaniel et al. tackle the problem where to place the software that collects provenance information (monitors). For non-formal approaches, they emphasize the need for securely deploying provenance especially in distributed systems [MBM+10]. Similarly, Tan and Lu argue that there are special problems in service oriented architectures [TGM+06] or cloud [LLLS10] environments. Consequently, the required mechanisms for reliable provenance depend on architecture and use case. For instance, Lyle et al. point out that it is applicable to protect the computation environment (i.e., processes or functions) and monitors with hardware-based methods, such as TPMs, when malicious effects (e.g., from the user) have to be considered [LM10]. Moreover, it is suggested to save reliable provenance data within multimedia data itself using invertible watermarks [SSM+11]. A similar approach is used in [CSV10], where the authors suggest to facilitate the least significant bits of sensor data for embedding a non-invertible watermark.

Basically, we identified two important points to take care of: (1) The execution context of processes or functions (e.g., usage of TPMs, hardware architecture, etc.) and monitors as well as (2) details about storing and transportation of artifacts including the detection of (malicious) modifications.

2.8 The question of identity

Another important property is that we need to be sure whether two artifacts are identical. This is apparently important for the recomputation layer. There, we focus on validating computation results. Thus, we presume functional dependency from input to output: $(f: a^n \rightarrow a^m)$. Now consider that we have a provenance data set as follows: $f_i(a,b) \rightarrow (c,d)$. That means that there was a computation somewhere in the past where a function f_i used artifacts a and b to compute artifacts (c,d). In case we want to validate the computation, we first need artifacts a and b for invocation of f_i. However, usually we have to fetch these artifacts from a database, search for them on a file system or even have to recompute them as well. Consequently, we have two new artifacts a' and b', which are copies of the original artifacts. So, we have to ensure that $identic(a,a')$ and $identic(b,b')$ holds. In such a way, we expect that invocation of $f(a',b') \rightarrow (c',d')$ where

$identic(c, c')$ and $identic(d, d')$ holds and thus we are able to validate the computation. In addition, we not only have to determine whether two artifacts are identic, but whether there is an identic artifact in large set of artifacts requesting for efficient support of such queries.

The question now is: How to define the function $identic()$. Or in other words what does identical mean for different artifacts (e.g., attributes, values, annotations)?

2.8.1 Determining artifact identity

In literature, we found two methods to determine whether two artifacts are identic.

ID-based method

Most approaches are based on sequentially assigned IDs. For instance, database formal approaches (e.g., How provenance [GKT07b]) and respective prototypes [GKT$^+$07a] use tuple identifiers. Furthermore, most all-in-one solutions for scientific workflows rely on IDs (visible in log files) as well [BCDH08]. The advantage of using IDs is its easy and fast application. Two artifacts are identic iff their IDs are identic: $identic_{id}(a, a') \leftrightarrow a.id = a'.id$.[6] However, IDs have to be globally unique and persistent. Despite the necessity for global uniqueness, IDs are not linked to the values or structure of the respective artifacts. That is, we can modify the values of primitive artifacts or even delete artifacts in the inner structure of complex artifacts and the ID-based method still considers them as identic, which is problematic for recomputation. Consequently, we have to ensure that these modifications are not possible. For instance, in PostgreSQL tuple updates are deletions and inserts resulting in new IDs.[7] Alternatively, we have to permit such modifications. The problem here is that the ID and the (data of the) artifact are not linked to each other.

For us, application of the ID-based method means that every complex artifact needs a primitive artifact carrying its ID.

Semantic-based method

A different way to determine the identity of two artifacts is using its semantics. For instance, in databases two tuples are identic if they have the same type (belong to the same table) and their attribute values are the same. We can adopt this method for our data model. For us, two artifacts are *semantically* identical if they have the same structure and all their primitive artifacts have the same role and value. However, in contrast to the previous (ID-based) method this algorithm requires much more computation effort than simply comparing two IDs. Furthermore, even unimportant changes, such as comments, last read time-stamps etc., result in non-identity of two artifacts. However, in case two artifacts are semantically identic, we can reliably use them to validate computation results.

[6] In case artifacts of different types can have the same ID, we also check the type $a.id = a'.id \cap a.type = a'.type$.

[7] PostgreSQL 9.1.9 Documentation - Chapter 55. Database Physical Storage, available at http://www.postgresql.org/docs/9.1/interactive/storage-file-layout.html

Combined approaches

The problem of sequentially assigning IDs is that there is no link between the content (e.g., values of primitive artifacts in complex artifact) and its ID. Thus, we can use techniques that are already used to ensure integrity and authenticity of data. For example, we can additionally use digital signatures and hashes (computed on some parts of the artifact) to determine identity (with a small residential risk of collisions of hash values).

A different approach is to compute feature-vectors extracted from the artifacts itself, exemplary used to determine source code ownership, named software bertillonage [DGGH11]. Then, distance metrics are used to compute similarity score and threshold to decide whether to source codes are clones of each other.

2.8.2 A flexible notion of identity

As discussed before, all known approaches have their strength and weaknesses. We do not want to restrict our framework to one of these approaches, but use a combination of them. To this end, we propose a mechanism that is based on inheritance. We define the basic semantic identity function that first checks whether both artifacts have the same role, then compare their structure, and finally determines if the values of all contained primitive artifacts are identic. All complex artifacts that are used in a system (e.g., a tuple in table or some object etc.) inherit the identity method from the basic complex artifact. Then, it is possible to define specific identity methods for each artifact role, such as $identic_{se}(Tuple\ t_1, Tuple\ t_2)$. In this way, we offer very restrictive identity functionality, required by our recomputation layer (and subsequent ones). Furthermore, a user can decide to simplify computation of identity, but has to ensure that the simplification is semantically sound.

2.9 Insights gained

The primary result of this chapter is that there cannot be a one-size-fits-it-all provenance solution due to the versatile nature of provenance. Consequently, we have to have tailored solutions collecting the data for the single layers of our provenance framework. Moreover, for fine-grained provenance capturing, data-model specific (i.e., only work for relational databases) solutions have to be provided.

Definitions

In this chapter, we revealed that there is no general definition regarding the term *provenance* as it describes an abstract goal not a particular method. Thus, we argue that it may be impossible to find such a clear definition as provenance research in computer science unifies different views from:

1. data-driven research in the area of linked data,
2. data-model research especially for relational databases,
3. process-driven research from scientific-data management
4. and (operation) system-based approaches.

Nevertheless, we are able to define several terms based on their usage in the context of this thesis. As we found no suiting definitions, we define them ourselves, based on the insights we made in this chapter.

Provenance. According to the Oxford Dictionaries[8], the term *Provenance* is taken over from the French word *provenir* meaning 'to come or stem from', which is itself based on the Latin word *provenire*, from pro- 'forth' + venire 'come'. A similar term, *provenience* has been adopted in archeology to define the location where a discovery was made, while provenance also includes a complete chain of prior owners in order to prevent forgeries. Similarly the term provenance is used in arts and for books [Pea95]. In summary, provenance helps us to judge on the validity of an assertion or the authenticity of an item. In this sense, we consider provenance, in this thesis, as the *goal* to judge on the validity of data or real-world item (retrospective aspect of provenance). In addition, it refers to the incooperation of provenance data into (future) computation steps. It helps to create more reliable results or to propagate changes in the validity of input to (all) results that were computed based on them (prospective aspect of provenance).

Provenance concern. A provenance concern refers to the requirements of an application, software system, or program with respect to provenance. In this section, we revealed that there are lots of tools and approaches to address the goal named provenance. Therefore, the provenance concern is, for instance, based on a threat model and contains respective methods to address these threats. From a software product (cf Chapter 3) line point of view, a particular provenance concern results into several features. Thus, it can be seen as a configuration of a feature model. Here, we encounter a limitation of current terms in SPL research, as a concern is, to the best of our knowledge, is not variable itself. Nevertheless, this definition fits best our purposes.

Provenance data. This term refers to data that is linked to an artifact and based on that we can judge on the validity of the artifact itself. Provenance data is variable in its form (schema), content, and interpretation, which is based on the respective Provenance concern.

Provenance functionality. Provenance functionality defines the implementation and integration that allows to capture provenance data as well as code fragments that evaluate and incooperate the captured data.

[8] Oxford Dictionaries online edition. http://www.oxforddictionaries.com/

Core challenges

In addition to the aforementioned issues, we identified two core challenges, which we address in the remainder of this thesis:

Provenance integration. Our analysis indicates that there are plenty of approaches on concept level, but mostly only the major approaches have been implemented. Only a few of them address how to capture tailored provenance data for real-world applications. Moreover, even less address the challenge of extending existing systems with provenance capturing functionality. The question that has to be issued, is how to extend an existing solution with provenance in a way that it does minimally interfere with the residual system functionality, which we call minimal-invasive integration of the Provenance concern. Moreover, we have to define what exactly does the term minimal-invasive integration mean in the context of extending an existing data-intensive system with new (tailored) functionality?

Tailored provenance store. Priorly, we revealed that the captured provenance data is very specific to the application scenario. So the question that emerges is: How to store provenance data that is application scenario specific efficiently, so that it can be queried?

In the following chapters, we refine these abstract results to a research agenda addressing the aforementioned challenges and finally resulting in the desired provenance software product line. With the help of this provenance software product line, we investigate whether minimal-invasive integration of the Provenance concern is feasible with the help of implementation techniques used to develop software product lines. However, before we can refine our objectives in Chapter 4, we have to give basic background on the implementation techniques we intend to use and software product lines in general in the next chapter.

3. Background on software product lines

In this chapter, we give basic background on software product lines (SPLs). Since one of the objectives of this thesis is to design and implement a software product line of a general provenance API, we introduce the general concept of SPLs and how to implement them. First, this encompasses domain engineering that, in our case, results in a model describing the similarities and differences amongst existing provenance capturing solutions. Second, we introduce background on application engineering, which is the automatic creation of a tailored instance of the SPL (variant) based on specific requirements of an application scenario. Finally, we provide an overview on techniques to implement SPLs. These techniques are used in the remainder of the thesis to integrate the provenance concern into our exploratory case studies.

Motivation for software product lines

Traditionally, there are two approaches in software engineering [CE00]: (1) general purpose software and (2) individual software solutions. The first approach offers a possibility to compensate the development effort with a high number of sold software (licenses). Consequently, this approach results in large-scale and, to some extent, general solutions. However, since specific requirements of an application scenario are usually unknown, this type of software often contains a lot of unnecessary functions or does not fulfill all the requirements of the application scenario. The second approach is designed to fulfill customer-specific requirements, but it is more problematic to compensate the development effort due to the smaller number of sold software (licenses). Furthermore the second type may result in developing software for similar problems each time from scratch.

In contrast to traditional software engineering approaches, the basic idea of software product lines is to offer multiple similar solutions in one domain from one common code base [PBL05, LSR07].

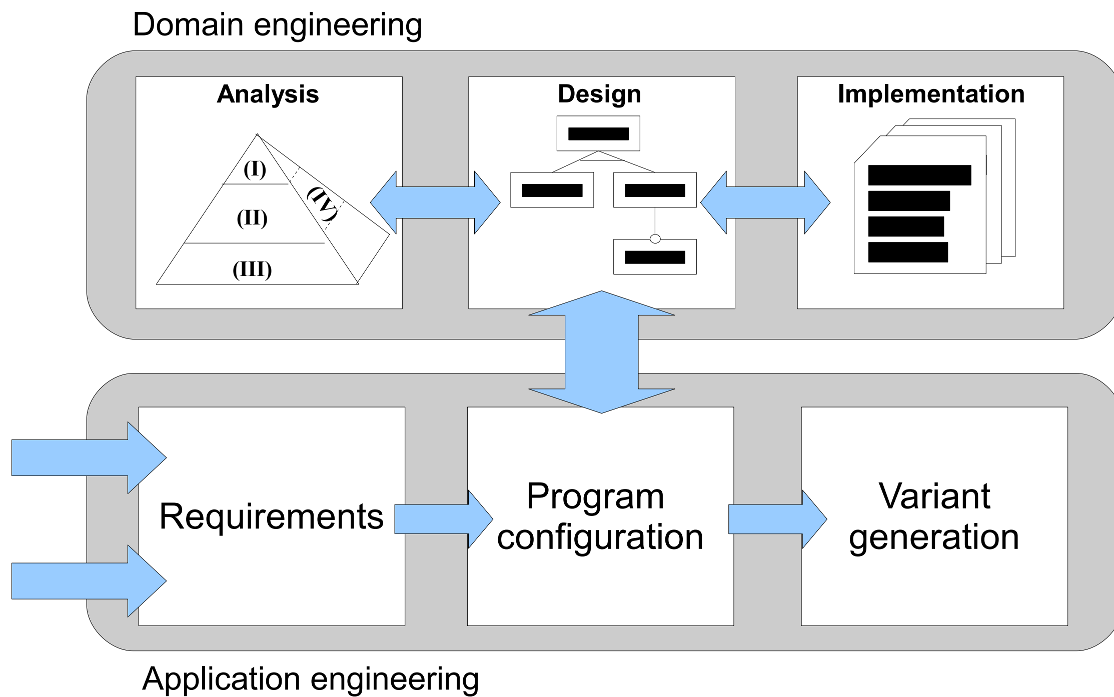


Figure 3.1: Phases of SPL Engineering adapted from [CE00]

3.1 Domain and application engineering

The concept of product lines is well-established in classic engineering disciplines, such as in the automobile industry, and has been transferred to Software Engineering. In an SPL there are several functionalities and other user visible properties (*features*). In addition, there is a description, which features can be combined to create an instance of the product line (e.g., a car), and a well-defined generation strategy that defines how the selected features are assembled to form a product. The overall SPL engineering process consists of two phases (cf. Figure 3.1) that we explain in more detail subsequently and how these phases interact with design and implementation of our provenance API.

3.1.1 Domain engineering

The primary goal of domain engineering is to model a whole domain. According to Czarnecki and Eisenecker, domain engineering consists of three steps that also apply to our provenance API [CE00].

Feature-oriented domain analysis

In the first phase, a domain expert has to identify possible features in the *domain analysis*. In feature-oriented domain analysis [KCH+90], domain experts identify requirements (e.g., based on existing solutions) that are candidates for features of the SPL. For instance, common requirements often represent features that are part of every variant of the SPL, while differing requirements usually result into optional features that can

be used to tailor variants to customer-specific needs. For our provenance API, domain analysis means that we do a comprehensive analysis of the state of the art, to identify core functionalities of current provenance capturing solutions. Here, one of the main challenges is to identify variability dimensions where we can offer differing functionality. For us, a variability dimension is highly related features that are for instance refinements of each other or alternatives. In current provenance research there are several fields (e.g., fine-grained approaches for databases [CCT09]) that may form such a dimension. Furthermore, we have to show relationships between features of one variability dimension, to offer fine-grained tailoring possibilities. Note that due to the nature of provenance it is very likely that there are gaps (i.e., missing features or approaches) that may be addressed by own approaches.

To sum up, the primary result of our provenance domain analysis should be a *conceptual framework* containing variability dimensions and classification of known approaches in provenance capturing regarding to these dimensions. This framework is, in fact, not only valuable for developing our provenance API but also contributes to the challenging question what provenance is and where are the borders to related fields of interest [CFLV12].

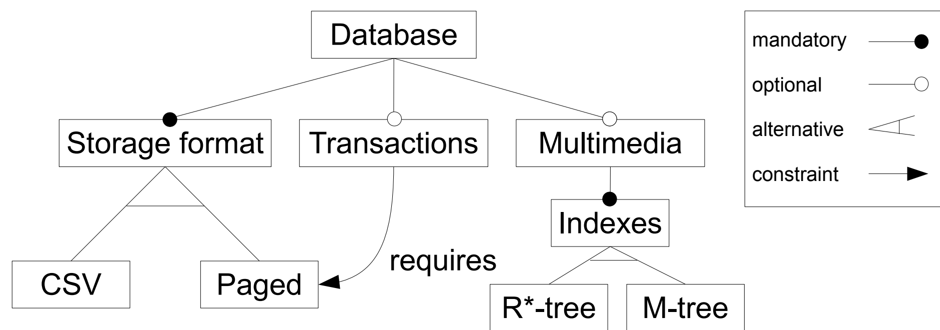


Figure 3.2: Feature model of a database SPL

Domain design

Based on the results of the feature-oriented domain analysis, SPL engineers build a *feature model* that represents the analyzed domain in terms of features and the dependencies amongst them. To this end, the feature model contains a description of the domain that states, which features can be combined to generate a valid variant. To visualize a feature model, commonly feature diagrams are used [KCH$^+$90]. In Figure 3.2, we show an exemplary feature model of a small database SPL. In this SPL, every valid variant that can be generated can store data, because the feature `Storage format` is *mandatory*. However, the user has to choose between one of the *alternative* (exclusive or) features that define how the variant stores data: Either as comma separated values choosing the `CSV` feature or on database pages in a proprietary data format by selecting the `Paged` feature. Moreover, for optional features, such as `Transactions` and `Multimedia` the user can choose any combination. In case the `Multimedia` feature is selected, the

variant contains the feature index as well, because it is a mandatory (child) feature of `Multimedia`. Consequently, this mandatory feature (tree) is only part of a variant if and only if the optional parent feature is selected. To express additional feature dependencies, it is possible to define cross-tree constraints, such as features A excludes or requires feature B. In our database SPL, the `Transaction` feature requires the paged data format (e.g., for rollbacks etc.) of the feature `Paged` and thus, implicitly excludes the `CSV` feature.

For our provenance API, we have to convert the conceptual provenance framework from the domain analysis into a feature model. This includes specifying the single features and defining their relationships. The core challenge here is to model and separate the relationships between the different variability dimensions possibly interacting with each other.

Domain implementation

The final step of domain engineering is the implementation of re-usable software artifacts (including respective documentation etc.). These artifacts represent the features on code level and are used to generate a tailored variant of the SPL based on the user selection (of features) and the generation strategy. Due to a multitude of proposed strategies that we refer to in the remainder as *implementation techniques*, we introduce them separately (cf. Section 3.2).

Usually, SPLs are not created from scratch, but decomposed from previously used monolithic applications [SLRS12]. Hence, the major purpose of these techniques is to decompose software into features, not to integrate a completely new variability dimension (here provenance) into an existing monolithic system. Consequently, we explore the general ability as well as benefits and drawbacks when applying these implementation techniques for integrating the provenance concern in the remainder of this thesis.

3.1.2 Application engineering

The aim of application engineering is to create a tailored variant of the SPL, based on available features (from domain engineering) and specific user requirements (product derivation). Consequently, as for traditional software engineering, the initial step is requirement analysis. Afterward, these requirements are mapped to available features. This results in a valid configuration of the feature model. A configuration is valid if the selection of features satisfies all dependencies in the feature model. Usually, tool supports prevents a user from selecting an invalid configuration. Yet, it is possible that, due to dependencies in the feature model or user requirements not considered in domain engineering, the SPL cannot satisfy all the requirements of a user. Then SPL engineers have to adjust the SPL, for instance in form of new features or by changing feature dependencies.

In this thesis, our aim is to provide a maximum degree of automation for the provenance API. Consequently, we focus on automated generation of tailored programs (generative programming [CE00]). Next, we introduce (implementation) techniques that allow such automated generation.

3.2 Implementation techniques

In this section, we review existing techniques for implementing the domain artifacts of an SPL and for generating tailored variants. We shortly discuss the limitations of an intuitive technique showing the need for more advanced solutions. Then, we introduce several techniques that can be used to implement the domain artifacts of our provenance SPL. For all examples that we show in the following, we use a Java-like syntax. However, respective explanations are valid for different programming languages as well, since we emphasize general concepts and do not concentrate on implementation details.

3.2.1 Intuitive techniques as reference

An intuitive technique to implement optional or alternative functionality is the usage of conditional statements, such as `if` or `switch()` in the source code of an application. Although this technique is tempting due to its easy application, it introduces several drawbacks, such as missing separation of concern or run-time overhead (e.g., for evaluation of unnecessary conditions) [ABKS13].

Dynamic if approach

In Figure 3.3.(a), we depict a *dynamic* version of the intuitive technique for our database SPL (cf. Figure 3.2). The main characteristic is that it is possible to change the value of the feature `Transaction` in Line 2 to activate this feature. However, runtime updates may rely on initialization or a specific feature selection (for our example the feature `Paged` is required, which is not checked). Hence, re-configuration of a variant at runtime faces fundamental challenges of consistent dynamic software updates, which is currently an open problem beyond the scope of this thesis [PKC+13]. Additionally, source code that belongs to deactivated features (e.g., Lines 7-8, 12-17, and 20) is still part of every variant and may introduce performance penalties or undesired feature interactions.

Static if approach

In Figure 3.3.(b), we introduce a modification of the intuitive technique that we refer to in the remainder as *static* if-technique. The primary difference is that in this technique we use *constants* instead of variables. Hence, when compiling a variant, the compiler should remove parts of the source code of deactivated features (Lines 7-8, 12-17) [Muc97, DEMD00]. In modern development environments (e.g., Eclipse[1]) this code is often marked as dead code. However, not required methods (Line 20) or classes are not removed. Due to these limitations, there has been plenty of research to offer more advanced techniques.

Meaning for this thesis

Although the aforementioned techniques exhibit several drawbacks at first hand, they are important for this thesis, because we want to investigate whether techniques used for SPL implementation are also beneficial for integration of a new variability dimension. Consequently, we need a ground truth to evaluate more advanced techniques.

[1] http://www.eclipse.org/

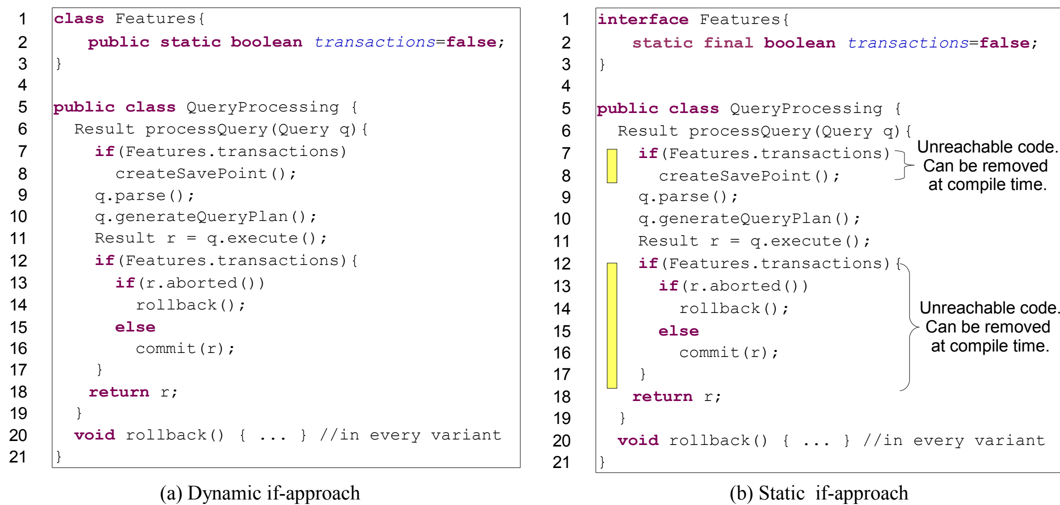


Figure 3.3: SPL implementation with conditional statements

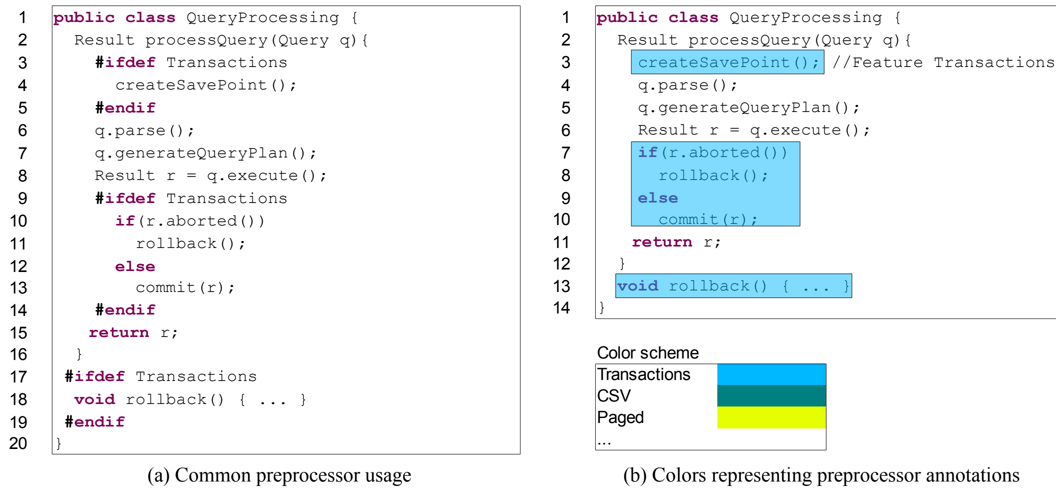
3.2.2 Preprocessor-based techniques

Conditional compilation with preprocessors, especially the C preprocessor, is a well-known technique for implementing variability [SW10]. Here, the source code contains different *annotations* that represent the features. For visualization purposes, in Figure 3.4.(a) we use the example already known from Figure 3.3, to demonstrate the application of preprocessors. In the example, feature-specific source code is annotated with #ifdef <annotated statement> #endif statements. Before compilation, the preprocessor removes feature-specific code whenever the conditional statement is not true. As a result, the respective feature combination is not selected. For instance, if feature *Transactions* is not selected by a customer, the preprocessor removes Lines 3, 7-10, and 13 from the source code in Figure 3.4.(a). Hence, in contrast to the intuitive technique, using preprocessors we can *physically* remove code fragments, such as methods, which are not required.

We can use preprocessors on a very fine-grained level, since it operates character based. For instance, we can modify names of methods, or add additional arguments to method signatures for certain variants of an SPL. Moreover, it is possible to nest #ifdefs in arbitrary ways and scatter the conditional code over the entire code base, which possibly consists of hundreds of thousands lines of code. This results in code that is hard to read and error-prone leading to assertions that introducing variability at a large scale with #ifdefs is harmful [SC92] even referred to as #ifdef-hell [LST+06].

Advanced preprocessor-based techniques

Due to the mentioned drawbacks of #ifdefs, improvements have been proposed to address the limitations of preprocessor usage. From our point of view, the most prominent and analyzed technique is Colored IDE (CIDE) also referred to as *virtual separation of concerns* [Käs10]. The idea behind CIDE is to map colored annotations



(a) Common preprocessor usage (b) Colors representing preprocessor annotations

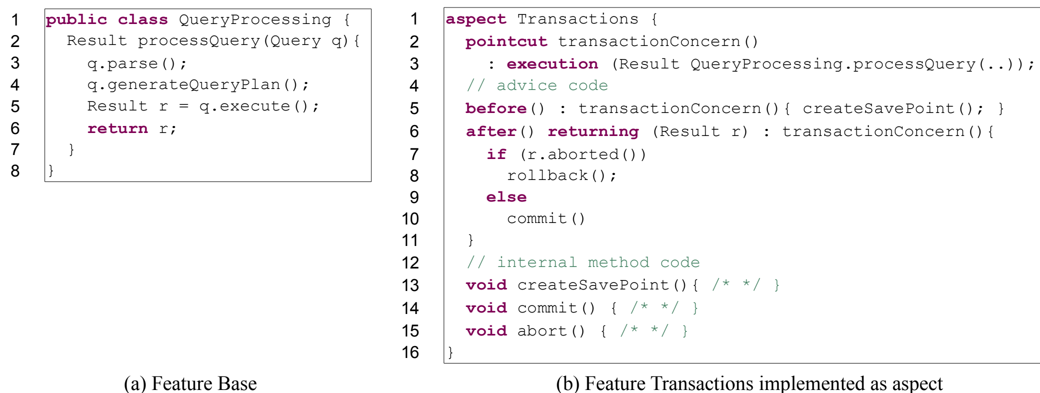
Figure 3.4: Conditional compilation with preprocessors

replacing `#ifdefs` statements to avoid the `#ifdef` hell [FKA+12]. Additionally, CIDE supports feature models etc. to provide a holistic approach for SPL engineering. Related techniques have the same goals (e.g., [BCH+10]), but do not offer the holistic approach.

Therefore, we limit our analysis for this type of techniques to CIDE representing the currently most prominent preprocessor-based technique.

3.2.3 Aspect-oriented programming

One of the basic problems of preprocessor-based techniques or the intuitive technique is missing cohesion of features. Ideally, the features should be implemented completely in a cohesive unit (e.g., an own file) containing only feature-specific artifacts [TOHS99]. All techniques introduced so far have a tendency to scatter their feature-specific code across the code base. In fact, they do not separate the particular code fragments of features *physically*, but *virtually* [KAK08].



(a) Feature Base (b) Feature Transactions implemented as aspect

Figure 3.5: SPL implementation with AOP

A technique applying physical separation is aspect-oriented programming (AOP) [Kic96]. In AOP, a feature may be defined by *multiple* aspects. A particular aspect refers to a feature that is encoded in an own file. AOP works similar to triggers (in AOP *pointcuts*) and trigger functions (*advices*) in databases. In Figure 3.5, we demonstrate SPL implementation using AOP. Part (a) of the example contains the original implementation, typically located in an own file (e.g., `QueryProcessing.java`), without any source code of the optional feature Transactions, while Figure 3.5.(b) is the implementation of feature Transactions. The definition of pointcut `transactionConcern()` in Line 2 and 3 states the context (usually a method or constructor) in which the advices linked to this pointcut are executed. In the example, the pointcut refers to every method `processQuery()` in class `QueryProcessing` that returns a class of type `Result`. Note the use of wildcards `(..)` in the signature of the referred method `processQuery` (Line 3). Using wildcards allows to refer to multiple methods with one pointcut, such as `processQuery(Query q)` or `processQuery(Query q, int priority)`. The advice definitions (cf. Figure 3.5 Line 5-10) include when to execute the advices, the linked pointcut, and the source code to execute. Consequently, the first advice in Line 5-7 is executed *before* execution of the method `processQuery()` and creates a save point in case the transactions is aborted. In contrast, the second advice (Line 8-10) is executed *after* method `processQuery()`. It organizes either rollback of the transaction in case of abortion or persistently saves the changes in the commit method.

3.2.4 Feature-oriented programming

Another technique that applies physical separation is feature-oriented programming (FOP). In FOP, all (source-code) artifacts that belong to one feature belong to one cohesive unit. This also applies to the *base feature* that contains the implementation that is the same for all variants of the SPL. To generate a particular variant of the SPL, the base implementation and the features, selected by the user, are composed incrementally [BSR03]. In the following, we explain the generation process of a variant. For details, such as algebraic properties, we refer the reader to [ALMK08].

```
1  public class QueryProcessing {
2    Result processQuery(Query q){
3      q.parse();
4      q.generateQueryPlan();
5      Result r = q.execute();
6      return r;
7    }
8  }
```

(a) Feature Base

```
1  public refines class Transactions {
2    Result processQuery(Query q){
3      createSavePoint();
4      Result r = Super().processQuery(q);//call base
5      if(r.aborted())
6        abort();
7      else
8        commit()
9    }
10   // additional methods
11   void createSavePoint(){ /* */ }
12   void commit() { /* */ }
13   void abort() { /* */ }
14 }
```

(b) Feature Transactions implemented as refinement

Figure 3.6: Feature-oriented SPL implementation using AHEAD

In Figure 3.6.(a), we depict the original implementation of class `QueryProcessing`, which contains only functionality of feature Base. Similarly to AOP, this implementation

contains no source code of the optional feature Transactions. The source code of feature Transaction is encoded in an own file that *refines* the original implementation (cf. Figure 3.6.(b)). Using FOP, an SPL engineer has to provide a refinement for every class that needs additional or modified functionality. To add new functionality, we encode an additional method in the refinement class having the same signature, such as `QueryProcessing` in Figure 3.6.(b). Generating, a variant then means using the refinement as wrapper, and in case the refinement contains the `Super()` keyword (Line 4), the original implementation is called.

3.2.5 Additional techniques and tool support

The combination of FOP and AOP is called aspectual feature modules (AFM) [ALS08]. We do not introduce AFMs, since we start using FOP and AOP as we have mature tool support for these techniques. We will only consider AFMs in case we identify the requirement for introducing a mixture of homogeneous and heterogeneous new functionality at a level of granularity beneficial for AOP and FOP in our provenance API. Moreover, in this thesis we require proper tool support to implement several case studies using different implementation techniques. To this end, we apply *FeatureIDE* [TKB+14] offering, to the best of our knowledge, the most comprehensive support for different implementation techniques and it is fully integrated into the Eclipse IDE[2].

3.3 Advanced topics of relevance for this thesis

Subsequently, we introduce advanced topics that are relevant for the remainder of this thesis. First, we explain variant composition as superimposition of feature structure trees for feature and aspect-oriented programming. Then, we introduce different types of cross-cutting concerns and explain how they are supported by the aforementioned implementation techniques.

3.3.1 Composition of features structure trees

To compose a variant of program using different features, a mechanism is required to abstract from implementation details in order to be able to compose different assets of a software product line such as source code implemented in different programming languages or the required documentation. To this end, the •-operator is introduced [AL08]. The basic idea is to have qualified paths that indicate, which assets have to be composed. Moreover, the result of a composition step of two features is again a feature containing the original implementation extended by the second feature, which then can used for additional refinement. In Figure 3.2, we explain this procedure with our database example composing the base feature `Database` with the optional extension `Transactions` denoted as: `DBwithTransactions` = `Transactions•Database`. The intended result of the composition is that the query processing of the database then supports transactions. For explanation, imagine that query processing is implemented in one source file `QueryProcessing.java`.

[2]Feature IDE is accessible via http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/

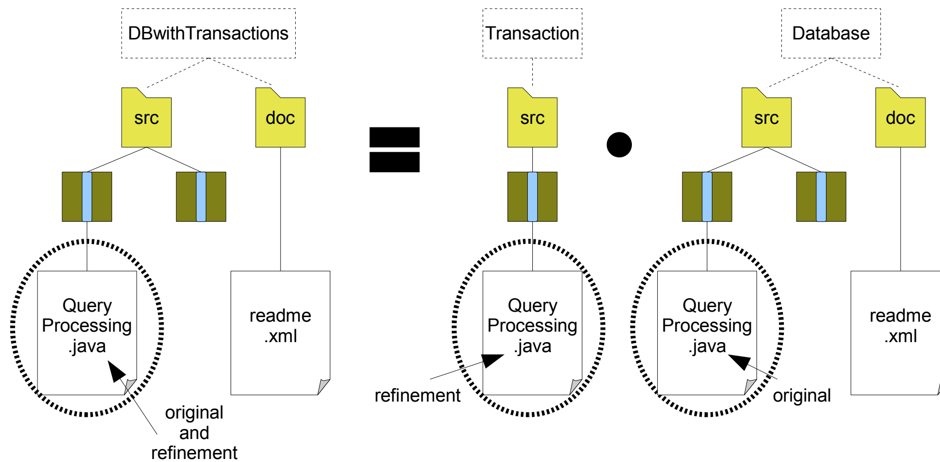


Figure 3.7: Excerpt of a features structure tree

To compose the variant, the asset from original implementation in the base features is extended with the asset sharing the same qualified path: `src.db_core.QueryProcessing`. This can be depicted using *feature structure trees* that visualize, which assets are composed as shown in Figure 3.7. How two assets are composed depends on the applied language and respective composition rules have to be provided. For instance in Figure 3.6 the `Super` keyword defines that the additional functionality wraps around the original implementation. In case this keyword is not provided the whole method is replaced. For this thesis, these implementation details are relevant as they may explain performance penalties of an applied technique. Moreover, in case we require composition support for new asset types (e.g., programming languages), we only have to provide the rules how to compose the assets in case we need to compose assets, because we can rely on an operator that is well studied and implemented [ALMK08].

3.3.2 Homogeneous and heterogeneous cross-cutting concerns

Due to one dominant decomposition strategy, the code of a certain feature may be spread over the source of the resulting program. To address this challenge the aforementioned techniques, such as feature-oriented programming and aspect-oriented programming, assist a developer to implement concerns in cohesive source-code units. However, it has been shown that there are different types of cross-cutting concerns defined upon the way how they extend the already existing code base. According to Colyer et al. [CRB04], there are homogeneous and heterogeneous cross-cutting concerns. In the following, we explain both classes of cross-cutting concerns with the help of the example in Figure 3.8. First there is an original program that is extended with additions of both concern types.

Homogeneous cross-cutting concerns. In Figure 3.8, on the right hand side a homogeneous extension of an existing source code is shown. As the locations where to extend the original program are scattered across all source files, the concern is cross-cutting. However, the basic property of a homogeneous cross-cutting concern is that the extensions are all the same. We depict this by same color in Figure 3.8.

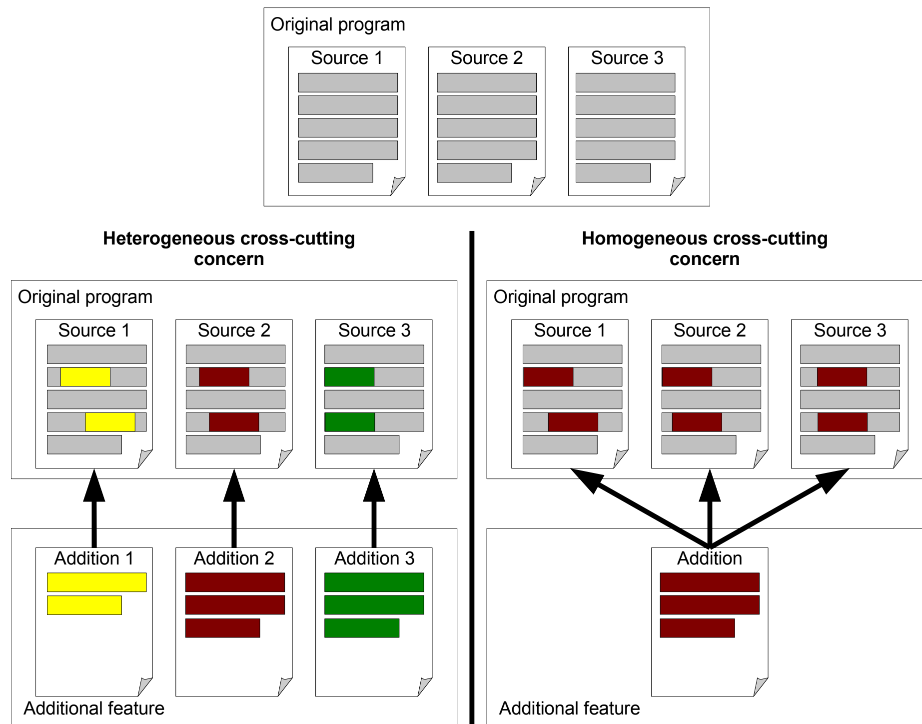


Figure 3.8: Homogeneous and heterogeneous cross-cutting concerns

Heterogeneous cross-cutting concerns. The basic difference of this type of cross-cutting concern is that there is not a single extension but multiple ones. For instance, in Figure 3.8, the concern is still cross cutting, because there are extensions in all source code files of the original program. However, for each file there are different extensions, indicated by the respective colors in the example.

The concern type is important as it is differently supported by the introduced implementation techniques, as we discuss shortly.

Applicability of the implementation techniques

From the implementation techniques we introduced, the intuitive technique as well as advanced preprocessor based techniques hardly support efficient implementation of cross-cutting concern. This is because, implementing such as concern results in copying the source code, required to implement the desired functionality, to the respective locations in the source code manually. Thus, for them the question whether a cross-cutting concern is homogeneous or heterogeneous is of minor relevance. In contrast, for aspect and feature-oriented programming the type of the concern is relevant. Aspect-oriented programming well supports homogeneous cross-cutting concern, because one advice can be bound to multiple join points, for instance using wildcards. By contrast feature-oriented programming supports heterogeneous cross cutting concerns, but has limitations with homogeneous ones. To support both concern types is the basic idea for the introduction of aspectual feature modules [ALS08].

So far, we do not know how cross cutting the provenance concern is. Therefore, we cannot estimate whether it is heterogeneous or homogeneous. Consequently, we have to implement the provenance concern into exploratory case studies our case study to answer the questions: How cross cutting is the provenance concern and whether it cuts heterogeneously or homogeneously through the original programs?

3.4 Summary

In this chapter, we introduce background on software product lines (SPLs) for our intended provenance API. To this end, we give background on the way how to design and implement SPLs and the associated procedures of domain and application engineering. Moreover, we introduced five implementation techniques. We include two intuitive techniques as reference, preprocessors, feature-oriented, as well as aspect-oriented programming. Finally, we discussed different types of cross-cutting concerns and their known correlation to implementation techniques.

4. Toward a general provenance-capturing solution for existing systems

As stated in Chapter 1, the goal of this thesis is to offer a general solution that allows for seamless integration of customized provenance capturing functionality into existing systems. Furthermore, we want to determine the influence (e.g., regarding performance) of different implementation techniques known from software product line engineering. To this end, we have to motivate the need for such a solution based on limitations of current provenance capturing solutions and requirements of systems that need provenance integration. To this end, we analyze current provenance capturing solutions in the first part of this chapter. Then, we use these results to refine the goals of this thesis and define criteria to evaluate them. Finally, we state the desired insights and course of action to achieve the previously mentioned insights.

4.1 Reasoning about the necessity for provenance integration in existing systems

In the first part of this section, we justify why we favor integrating tailored provenance rather than usage of existing solutions or why we do not want to re-implement large parts of existing infrastructure that need provenance support. For our explanations, we introduce a large-scale joined research project supporting the validity of our claims and explanations. In addition, we discuss in how far provenance challenges of this research project represent general, real-world challenges to avoid overfitting of our claims. In the remainder of this thesis, we also apply this research project to identify case studies that contain the challenges stated in this section.

In the second part of this section, we review in how far existing provenance solutions and approaches can be used to integrate tailored provenance support and discuss their

limitations and explain why they are not sufficient to fulfill our requirements. The final consequence is that current solutions are not feasible and hence the need for a new approach that we introduce in the next section.

4.1.1 The need for customizable solutions

In the remainder of this thesis, we use our experiences from the Digi-Dak project already introduced in Section 2.3.2 to justify claims and motivate design decisions. While we used this project in Section 2.3.2 to generally motivate the need for different provenance layers, in this section, we amplify it and discuss the requirements from the software engineering perspective. Furthermore, we state that these experiences and requirements are not artificial, but to some extent generalizable. As a result, this section motivates the need for a solution that allows for seamless integration of a customized provenance concern.

Motivating running example Digi-Dak project

In Digi-Dak[1], partners from academia, industry, and law enforcement agencies develop new solutions for crime-scene investigations. The original project running from 2010 to 2013 focused on non-destructive acquisition of latent finger prints at crime scenes that we named Digital Dactyloscopy[2] [MGD+12]. Then, in 2012, with DigiDak+ additional forensic evidences, such as traces on firearm cartridges and lock pins are included.

However, independent of the type of the evidence, one core challenge is keeping the chain-of-custody. The primary goal of the project is to produce evidences that may be used in court. Therefore, we have to document the whole chain from acquisition of the evidence, over quality enhancement (e.g., application of different filters [HKGV11, KHD12, KHDV12]), most likely stored in multiple large-scale databases to its final usage in court [Bis02]. Moreover, we have to prove the validity the general procedure, for instance via Daubert hearings [Cen93, DG01].

In summary, we argue that the chain-of-custody is a special form of highly reliable provenance where we have to ensure the integrity and authenticity of an evidence itself and respective provenance data. By integrity we understand that a data item is unaltered. Authenticity indicates that the data set, describing a specific forensic evidence, belongs to a real evidence found at a crime scene and is, for instance, not forged.

Our focus: Offering a bundle of solutions

The validity of overall analogue treatment of forensic evidences, including integrity and authenticity, is based on long time experiences, scientific results, and assumptions [MMJP05]. The questions now are: How to transfer this knowledge to the digital process? What new challenges arise? How to provide the same level of trust in such evidences? Or is it even possible to extend that level of trust? Our focus in this project

[1] http://www.forschung-sachsen-anhalt.de/index.php3?option=projektanzeige&pid=14049

[2] The abbreviation Digi-Dak is taken from the German translation *Digitale Daktyloskopie*.

was not the validity of the overall procedure, but enriching the parts of the overall solution, invented by our partners, with the desired provenance data. Moreover, we store the artifact data including respective provenance data and prevent forgery of these evidences including revealing for instance malicious modifications.

Basic requirements. These basic prerequisites have certain effects on our desired general provenance solution:

B1 - Research in progress. We need a complex infrastructure, where several features cannot be specified in the beginning of the project, because it is for instance simply unknown how current laws and guidelines can be transferred from the analogue processes to the digital ones. Furthermore, we cannot simply gather all data, since these evidences are person-related data affected by additional laws. Thus, the way of protecting these data and linking it to the provenance is not clear.

B2 - Nation-specific legislation. Although we focus on Germany and German legislation, we need to be aware that different countries have different requirements. Moreover these requirements may be complementary.

B3 - Privacy requirements. Currently, there is large discussion on laws regarding privacy or treatment of digital forensic evidences (e.g., [GFRD09]). Thus, we need to be aware that requirements regarding provenance capturing may change quite rapidly. Hence, we should not use inflexible solutions but adaptive ones.

B4 - Security and performance trade off. Especially securing data against forgery or capturing details of computation processes may result in large performance deficits. Moreover, we figured out that performance is a critical point. To achieve acceptance of such new solutions, the overall procedure (including the provenance concern) needs to be transparent and should not have a negative effect on the way, for instance a finger print expert works. This results in the requirement that we have to minimize the effects of provenance capturing. Thus, we need fine-grained customized solutions that offer only the desired functionality.

B5 - Flexible adaption. Finally, as we do not only want to provide a solution for certain requirements of the Digi-Dak project, but a to some extent general solution, we have provide a solution that flexibly fits into a multitude of existing systems and infrastructures.

Specific requirements requesting flexible solutions. Besides the aforementioned basic requirements there are more specific ones that we discuss subsequently. In particular, both requirements are imposed due to continuous changes of required provenance support throughout the project. This results in the need for customizable provenance solutions.

S1 - Different provenance requirements in different phases of the project. In different phases of the project, we need different types or granularity of provenance.

For instance when designing or integrating new preprocessing steps, we may want to verify the correctness of the implementation and thus, are interested into the origin of every single pixel of an image (similar to debugging). By contrast, in that phase we are simply not interested in detecting forgeries etc. Now imagine, we are convinced of the correctness of an implementation and want to benchmark that part of the infrastructure, which includes also query response times. In this case, we do not only want to know which input was used for repeatability reasons. In turn, in case we want to certify (parts) of the Digi-Dak infrastructure, we need to ensure the complete chain-of-custody with detailed provenance granularity and requirements from IT-Security, such as integrity and authenticity.

S2 - Allowing rapid independent prototyping. Especially for preprocessing of fingerprint images there are plenty of existing solutions or techniques. These solutions are modified for our purposes, enhanced, combined, or results of totally new research. For instance, in case, a new approach is found in literature an evaluation tool is rapidly implemented to benchmark this approach. The basic purpose of these tools is proof of concept of functional behavior and sometimes first feasibility studies. However, most of these approaches are not used in the final Digi-Dak infrastructure. So we have two possibilities: (1) integrate the provenance support from scratch or (2) integrate it later on, in case we want to use that approach or tool. The first approach results in large implementation and design overhead and offers little benefit in return as the basic goal is functional proof of concept. Moreover, as our provenance requirements change throughput the project (see B1 and S1), already integrated provenance functionality may not be compatible to newer ones. As a result, we have to integrate the provenance functionality later on in minimal-invasive way to allow rapid independent prototyping.

Summarily, we conclude that we need tailored provenance support for Digi-Dak. Moreover, large parts of the infrastructure require provenance support. However, for many systems of the infrastructure the specific required provenance functionality changes continuously throughout the duration of the Digi-Dak project. Consequently, we need a bundle of tailored solutions that can be seamlessly integrated into our existing systems and a solution to avoid re-implementing large parts of the infrastructure again and gain.

Generalization of the running example

The running example may inherit special requirements, due to the nature of forensic treatments. Nevertheless, in the remainder we use this example to select case studies (cf. Section 6.1.1) to address different parts of the overall infrastructure. The overall infrastructure aims at creating a holistic approach that is required to build a complete solution that acquires forensic traces and enhances the quality of raw images in multiple ways. Consequently, the chain-of-custody (i.e., provenance) plays a decisive role. This ensures comprehensiveness and soundness of the results gained in this thesis as we point out shortly.

Comprehensiveness. As we consider a complex system having different tools that require different granularity of provenance and work on the project for a time span of more than three years, we consider our results and experiences as comprehensive.

Soundness. A challenging problem when performing empirical studies is to avoid biasing the results by improper selection of the case studies. For instance, using self-implemented case studies may unintentionally contain (architecture) elements that simplify or harden the propositions that shall be shown. Alternatively, the problems contained in such a case study may not represent real-world challenges. To circumvent this threat, we select case studies that we did not implement or design by ourselves, but are taken from the Digi-Dak example and contain different complexity (mainly measured in amount of code lines). This ensures soundness of our results.

Similar experiences from practitioners. To strengthen our claims and argumentation to produce comprehensive and sound results, we talked to practitioners that have long term experiences for incorporating provenance in real-world systems. For example, MITRE[3], is a large non-profit organization that works on public projects where some of them include provenance. There results and experiences, tellingly referred to as ”provenance capturing in the wild” [ACBS10], rely on similar requirements as we worked out for Digi-Dak for totally different scenarios. These results and experiences addressing provenance for real-world systems are also published such as in [ACBS10, CBSA11, Cha08].

As a result, we argue that selecting our case studies from the context of Digi-Dak project delivers valuable insights for practitioners and scientists.

4.1.2 Are current solutions feasible?

Before we start to implement our own solutions, we review currently used approaches that could be used to implement our case studies. In our literature study, we found four classes of approaches [SSS12a]. Summarily all of them have limitations that motivate the necessity for our new solution.

Formal database-related approaches

Plenty of research focused on formal models mainly based on the relational algebra [CCT09], such as [CW00, BKT01, GKT07b, ADT11], or on related data models which may be mapped back to a nested relational calculus [ADD+11]. However, existing implementations of these formal approaches, such as Orchestra [GKT+07a], do not satisfy our need for variability, as they are built exclusively to capture their form of provenance and to the best of our knowledge do not exist as extension for existing major database systems.

Closed-world systems for provenance capturing

Especially to support scientific-data management there are domain-specific solutions that allow to design the workflow and capture provenance as well (e.g., Kepler [LAB+06]).

[3]http://www.mitre.org/

However, there are two reasons why we cannot apply such solutions. First, we need to be able to flexibly, integrate, un-integrate, or modify provenance capturing functionality. Thus, these solutions are to restricted. Second, there currently is only one approach that allows for combining coarse- and fine-grained forms of provenance [ADD+11]. However, this approach does not consider the variability requirements and the security requirements, crucial for forensic data management.

System-based approaches

Another group of provenance-aware systems, are solutions that are integrated into operating systems [MHBS06] or provide only low level information such as which files were created or read [MS12, FMS08]. Although these approaches are hard to circumvent they do not provide the amount of information nor the variability we are looking for. However, these solutions can be used as *additional* security mechanism.

Provenance capturing real-world applications via intercepting communication

In literature, there are several approaches that provide monolithic APIs [ACBS10]. However, these approaches require that the application for that we want to capture provenance is permanently changed and actively uses the API and thus, has to be aware of provenance capturing. To overcome this limitation, there are solutions that intercept provenance. In fact, these systems rely on homogeneity in the infrastructure, such as a communication between different parts of the infrastructure via services [SPG08, GMM05] or buses [CBSA11]. Consequently, the granularity of the provenance captured is restricted to the exchanged messages. Thus, we can use the same *concept* for our future case studies. Nevertheless, these solutions do not support our variability requirements and they are limited to one communication infrastructure. Finally, these approaches do not support to capture detailed provenance and thus do not fulfill our requirements.

4.1.3 The necessity for an own solution

As a result of our literature analysis, we state that we found two major problems of currently used approaches. First, none of the approaches published so far, is able to fulfill our variability requirement. Second, most solutions are limited to their application domain and cannot track fine-grained and coarse-grained provenance, necessary for our forensic scenarios. Consequently, we need to implement our own solutions.

4.2 The goal of minimal-invasive and efficient provenance integration

As motivated before, the overall goal is to integrate a customized provenance concern into an existing system. Generally, major changes of existing systems are critical, since we do not want to introduce errors or increase resource consumption (e.g., CPU or main memory) excessively. To this end, we want to modify the system in a minimal-invasive

way. This term is an analogy borrowed from surgical procedures representing the abstract goal that we want to modify the functional and non-functional behavior of system as least as possible. To evaluate the degree of invasiveness different implementation techniques (cf. Section 3.2) inflict, we define respective evaluation criteria subsequently based on our prior work [SSS12b].

4.2.1 A notion of invasiveness

While invasiveness for surgical procedures is defined on skin injuries, damage inflicted to reach the intended region of the body, and minimizing the time to recover [Paz11, Wic87], to the best of our knowledge these properties are not defined for modifications of existing software systems. Thus, we have to define what minimal invasiveness means in the context of this thesis. Generally, we have very similar goals when modifying existing systems, as surgeons in medical treatments.

1. Minimizing manual implementation and maintenance effort.
2. Non-interference with existing functionality (if not intended).
3. Robustness of non-functional properties (e.g., main-memory consumption).

Generally there is an original object-oriented program O and an extension E that introduces new fields, methods or refines methods (or constructors)[4], possibly consisting of multiple features that shall extend the existing monolithic program O. Moreover, there is a generation strategy (cf. Section 3.1) $\oplus_t$ that is used by implementation technique t (cf. Section 3.2) and the resulting program $O_E = O \oplus_t E$. Then, we qualify differences between O and O_E regarding to certain effects on functional and non-functional properties that we define shortly.

Invasiveness itself states the number and type of changes in the source code necessary to integrate the provenance concern. In addition, it states the ability to automatically remove not required parts of this concern (e.g., for customizing reasons). The basic assumption is that a less invasive integration has a lower probability of introducing non-obvious errors regarding the functional behavior of O_E. We now discuss the expected errors when introducing different levels of invasiveness.

Influences on functional behavior

Provenance features may affect the functionality of the original system differently. To qualify the level of invasiveness, we rely on a modified version of a categorization system [Kat06]. The categorization system, introduced by Katz, is used to group aspects in AOP only. Alternative categorizations (e.g., [RSB04, SdM03]) are known, but do not

[4] Note with this definition, we explicitly exclude modification that add or remove single characters, for instance to change method names, which is possible using preprocessor-based techniques. The reason is that these un-disciplined annotations introduce several severe problems and can be avoided [KKHL10].

offer all the categories or a precise definition of the single categories. In addition, we can easily extend the single groups in such a way that we cover features in general.

We introduce some modifications to the original categorization and discuss their effects and necessity shortly. First, we add an additional group to represent the ultimate goal of having no negative effect on the system at all, for instance by application of hardware-based solutions. Although these solutions are beyond the scope of this thesis, we want to mention them explicitly for (1) potential future work and (2) indicate that we are aware that every software-based solution has some (probably) negative influence. Second, we do not consider aspects but changes and group them to features. Each feature is implemented by several changes (e.g., if conditions). As a result, we are not limited to AOP but can use arbitrary implementation techniques.

Table 4.1: Categorization of effects on functional behavior based on [Kat06]

	Add to system			Modify		
	field	method	class	feature-specific variables	control flow	system variables
Non-invasive change	no	no	no	no	no	no
Spectative change	yes	yes	yes	yes	no	no
Regulative change	yes	yes	yes	yes	yes	no
Invasive change	yes	yes	yes	yes	yes	yes

In Table 4.1, we depict our version of the categorization introduced in [Kat06]. In the remainder, we explain the ideas behind the single categories and their respective restrictions.

Non-invasive change. A non-invasive change does not introduce any change to functional nor to non-functional properties. Nevertheless, we still can capture the desired provenance information. In fact, according to our definition E is empty. That is, E does not introduce fields or methods and thus $O = O_E$, which means that the binaries of both programs are the same. To be able to collect provenance information, we need hardware support that for instance logs, which addresses in the RAM are read and what data is transferred via a network etc. In this thesis, we limit ourselves to software modifications. Nevertheless, as we want to integrate the provenance concern as less invasive as possible and thus want to get as close as possible to this type of change.

Spectative change. The basic properties of these kind of changes is that it works as if we stop the program, do some independent computations and return to the program without any change of the *program states*. More formally, we have a sequence of steps of original program O denoted as $s_0 \dots s_n$. In each step s_x $status(O, s_x)$ of program O is changed where the status is defined by the all objects of O and their respective (primitive) field values (see artifacts in Section 2.8.2) and the position of the instruction pointer. A spectative change E executes its functionality between two steps s_x and s_{x+1}

in such a way that $status(O, s_x) = status(O_E, s_x)$ and there is a status $status(O_E, s_{x+k})$ so that $status(O, s_{x+1}) = status(O_E, s_{x+k})$ holds. Moreover, for each future status, where no functionality of E is executed, $status(O, s_{x+1+n}) = status(O_E, s_{x+k+n})$ holds. Consequently, E must not modify the control flow or change any objects nor their values of O. It may however change internal artifacts of E, for instance to capture provenance of objects in O.

Spectative bounded change. There are cases when even non-functional properties may have an impact on functional properties. Therefore, we introduce a special sub type of the spectative change: the *spectative bounded change*. The basic idea is as follows. Imagine a database system with very simple transaction support. To detect and resolve deadlocks each transaction has a (watch dog) timer as often used in embedded systems. This timer is reset whenever the transactions reads or writes a tuple. The basic assumption is when the timer exceeds a pre-defined value, the transaction was not able to read or write a tuple, so there probably is a deadlock and the transaction is aborted and then started again. Now, due to the provenance functionality, it may be possible that the counter is not reset due to the additional functionality and the system assumes that there is a deadlock and aborts the transaction. Similar scenarios are also possible for different non-functional properties, such as functions that monitor the RAM consumption etc. In order to prevent such effects, we can annotate the status with additional information based on non-functional properties and bound the difference between these annotations of two states. For the database example, we annotate the each status s_x with a time stamp $t(s_x)$ and define that a change is *bounded* in case for $status(O_E, s_{x+k})$: bound $< t(s_{x+k}) - t(s_x)$ holds. However, we only use this extension in case there is such a coupling of function to non-functional properties.

The spectative change is very important for our minimal invasive integration, because

1. it is the lowest invasive level that we can reach using software engineering techniques only and

2. as we do not change the state of O it is very unlikely that we introduce functional errors when integrating the provenance concern.

Nevertheless, in contrast to the previous change class, the influence on non-functional properties is hardly predictable.

Regulative change. For this change type we allow that E changes the control flow in O. In particular, among the objects of O there is a set of decision variables d that are used *only* to manipulate the control flow, such as `if or switch case` blocks. Now E may change any of these decision variables, so that O may use a different branch of an if condition or switch-case block. This way, we *limit* the amount of changes in such a way that we do not integrate totally new functionality, but use existing one. The main hypothesis is that this change is less error prone than a total invasive change where there are no limitation to what E may change.

For instance in databases, we can use such a mechanism to implement (and propagate) view deletion in materialized views if the original tuple has been deleted in the source table by extending the selection predicate with provenance information.

Invasive change. Here, there are simply no limitations what may be affected. Hence, this type of change should be avoided as it is in opposition to our goal of minimal-invasive integration.

4.2.2 Measuring invasiveness of different implementation techniques

For evaluation purposes of implementation techniques, we use groups of criteria: (1) invasiveness, (2) implementation and maintenance effort, and (3) changes to non-functional properties. To qualify (1) invasiveness, we rely on the categorization introduced in the last section. For the remaining groups, we now define respective evaluation criteria to be able to quantify invasiveness in general.

Implementation effort and maintainability

One key point is the effort to integrate the provenance concern and later on maintain it. However, qualifying the implementation effort is quite hard, as we implement the same functionality multiple times. So, after the first implementation all subsequent ones should benefit from the first one. For instance, we know in what classes in the original program we find the desired provenance information. As a result, we first use one objective measurement, namely *feature cohesion*. Coherent features are implemented separately (e.g., in own files) containing only the feature code, while non-coherent features are scattered over the whole source code. We favor coherent implementations as they are considered to provide better separations of concerns and in return are better for implementation and maintenance.

As second criterion, we apply our subjective impression, how much effort for locating the points where we need to integrate the provenance concern is required. At this point, we assume that for many case studies identifying all points in the source code of the application where we need to integrate the provenance concern is one important factor. Thus, we report on whether the implementation techniques help us to find and integrate the provenance concern. Moreover, we expect that we need to re-factor or even re-structure the source code of the original programs to be able to integrate the provenance concern, which may also be linked to the supported granularity of the implementation technique.

On a more general base, one important question is how programs and their decomposition in classes interfere with the provenance concern. That means for instance, how cuts the concern through the original program and is the integrated functionality the same at many integration points (homogeneous) or heterogeneous. This contributes to the question what properties help us or impose additional drawbacks when integrating the provenance concern.

Influences on non-functional properties

To qualify the effect of our provenance integration in the remaining sections regarding non-functional properties, we consider three commonly used criteria [SRK+12]: (a) response time, (b) main-memory consumption, and (c) (binary) footprint of the modified solutions. Furthermore, we use two ground truths to compare the effects of different implementation techniques. First, the basic ground truth is the original system O that represents the ultimate goal that we would like to reach with non-invasive changes only. Second, we use the properties of our intuitive approach using conditional statements (cf. Section 3.2.1). This is in fact, the technique that shall be outperformed or least reached by more advanced implementation techniques.

Response time. In this thesis, response time is defined as the time span from invoking a process until *all* results are available. We use this property, because for data-intensive applications, such as databases, performance in general and response time in particular is of high importance.

Main-memory consumption. Another important factor is main-memory consumption. Data-intensive systems usually require a lot of main memory and have often limited memory resources. As they require main memory for actual computation, we have to avoid that additional data has to be swapped to the hard disk (e.g. buffered pages in databases). Hence, we need to minimize the amount of additionally consumed main memory.

Footprint. In this thesis, the size of the compiled binaries (e.g., in Java the .class-files) is named footprint. The basic idea why we consider this property as one of the criteria to quantify invasiveness is the ability of the composition strategy to remove unused parts of the source code (i.e., not required functionality). In case the binaries O_E have the same size as O the composition strategy was able to fully remove not required functionality. Moreover, as closer the size of the binary of O_E is compared to O the more functionality is removed.

4.3 Conceptual design

For the general provenance capturing solution we target at, we need several infrastructure parts that we define and conceptually design in the following. In fact, our solution is not an SPL in the classic sense, but consists of several *variable* parts that need to be plugged together as well as parts that need to be integrated into the source code of the original program.

To this end, we first explain the overall vision that is, to some extent, inspired by the work of Cheney et al. in [CCF+09] and named a *universe of provenance*. Based on this vision and the insight gained so far, we derive, define, and explain the single parts of our infrastructure that we use for our exploratory case studies in Chapter 6. Finally, we justify our selection of infrastructure parts that we focus on in the remainder of this thesis, because they impose novel and, for us, interesting challenges.

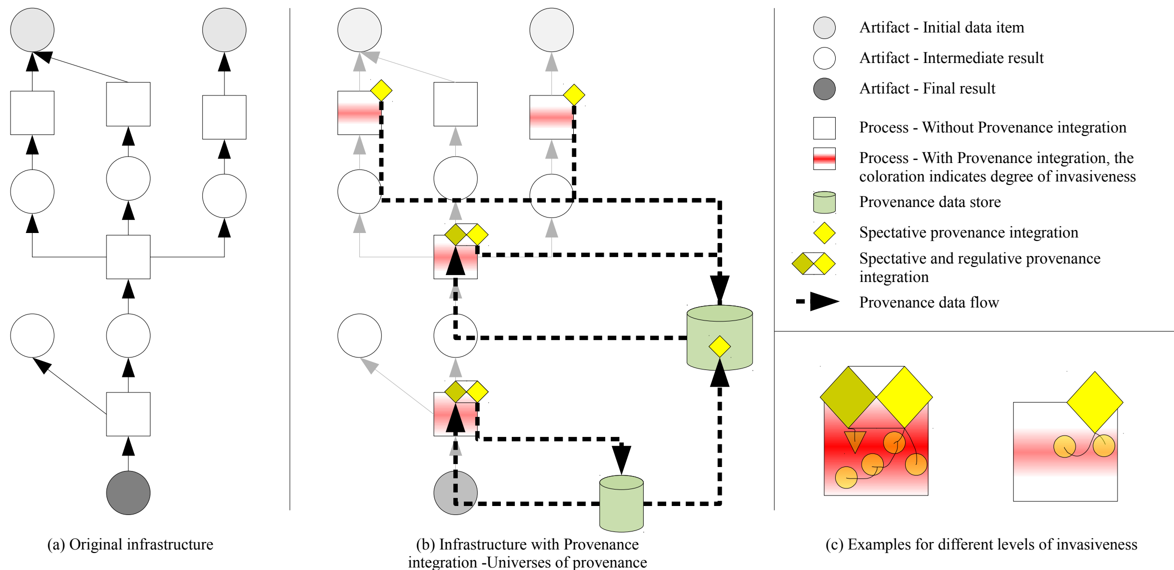


Figure 4.1: Vision - A universe of provenance

4.3.1 Vision - A universe of provenance

The overall goal or idea behind the contributions of this thesis is to improve knowledge on information represented by arbitrary data items. To this end, we want to improve trust in computation results in general or at least have additional information to judge on its reliability. Therefore, we need to have knowledge on the impact of single data items on aggregated results and be able to track back a data item to its original source. This knowledge shall, however, not result in a continuous chain of events (processes) and intermediate results (artifacts) that can be tracked back to the Big Bang as provokingly defined as the source of any information [BSS08]. More specifically, we want to be able to steer the level of fragmentation and uncertainty (cf. Section Chapter 2) that we produce and define requirements for data we consume. In addition, we also link different parts of information, probably aggregate and propagate them (cf. Figure 4.1.a). Our contribution therefore is tailoring the provenance capturing solution and offer solutions to plug different provenance capturing solutions together forming networks that are loosely coupled as depicted in Figure 4.1.b resulting in a universe of provenance.

4.3.2 Architecture - The neuron analogy

While we explained the meta level goal so far, we now concentrate on the architecture of one instance of our provenance capturing solutions that shall be integrated into one existing system to capture tailored provenance data (cf. Figure 4.1.c). To this end, we intuitively need infrastructure parts that intercept and extract the provenance data and a part that stores the intercepted data. Thus, we also need some infrastructure part that transports the extracted provenance data to the provenance data store. Altogether the infrastructure forms a network that transports information to store and eventually evaluate them, similar to neural networks in biology. In computer sciences, biologic

neurons are a famous analogy to visualize such information flows. As a neuron analogy can very easily and tellingly be used to explain the single parts of our infrastructure, we use the *neuron analogy* to visualize the single parts of the infrastructure in this section, too.

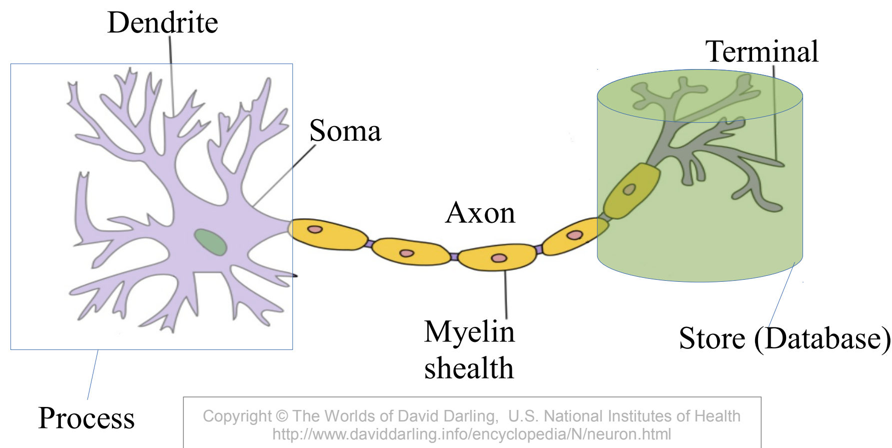


Figure 4.2: The neuron analogy

Parts of our infrastructure

Subsequently, we describe the semantics of the infrastructure parts that we depict in Figure 4.2 and how they relate to source code or other elements of our desired solutions.

Dendrite. The Dendrites are the real invasive parts that link the original program to the provenance capturing functionality. In case we use the intuitive approach, Dendrites are the if-conditions that are integrated in the original source code. The basic goal is to re-use neuron implementation between different case studies (with the help of our implementation techniques), which is challenging as we currently hypothesize that these parts are application specific. Moreover, the integration shall be as less invasive as possible.

Soma. The Soma is the code that is called by the Dendrites. So this is in fact the actual API. Furthermore, the Soma links several pieces of provenance to the data model that we described in Section 2.4 and stores intermediate results until we store them persistently. For instance, in case we integrate our provenance functionality into a database system, the provenance data of all uncommitted changes is stored in the Soma and only propagated in case the transaction is committed.

Axon. Usually we want to save the provenance data persistently. To this end, we need to transport the data to a database or some component of our infrastructure that writes the provenance data on the hard disk. Therefore, we use Axons, such as a database driver or classes that write data on the hard disk from a software point

of view. Note that hardware parts, such as networks etc. that finally transport the data, are not explicitly addressed in this thesis, as we want to focus on the software part only.

Myelin shealth. When transporting provenance data to its final storage location, it may be necessary to protect this data against possibly malicious changes and unauthorized access. Therefore, we use features of the myelin shealth, such as signatures, hashes, encryption, or watermarking techniques. Consequently, the myelin shealth contains countermeasures to the previously mentioned threats while the data is transport by the Axon.

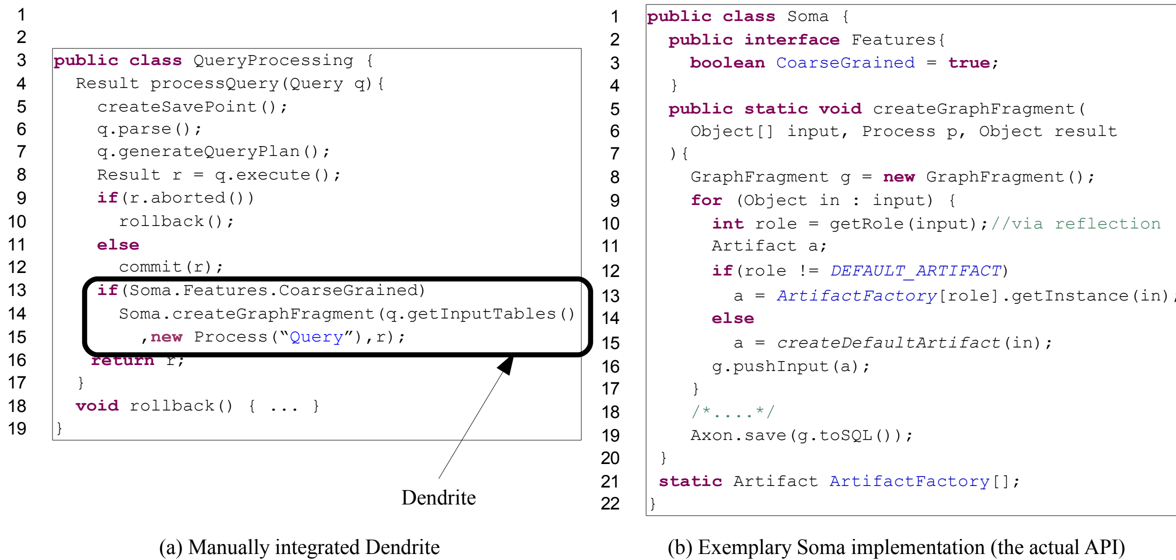
Terminal. This optional part of the infrastructure maps the data transported by an Axon to a data format that is understood by provenance data store. In case, we want to store provenance data in a database, we can either use the normal structured query language (SQL) interface or we can circumvent the SQL parsing etc. and use internal classes of the database management system to insert our data.

Provenance data store. The provenance data store persistently stores the provenance data and offers functionalities to query the collected provenance data. Generally, we intent to use relational databases, but also customized solutions as well as simple comma-separated files shall be possible.

Mapping to program code

In Figure 4.3, we depict how the previously introduced infrastructure parts map to source code tokens with the help of the already known example of a database system from Chapter 3, to give the reader a better understanding of our previous abstract explanations. For simplicity, we assume that the database system is monolithic (i.e., no SPL) itself. In part (a) Lines 11-13, there is additional source code (the *Dendrite*) that has been integrated into the source code of the original database system using the intuitive implementation technique that permanently modifies the source code. This source code, creates a graph fragment of our indented provenance graph consisting of input artifacts (the source tables of the query), the process (the query) itself and the result (again a table). The Dendrite itself only calls the functionality in the Soma in part (b) of Figure 4.3.

The important thing about the Soma is that there are *two* possibilities to treat artifacts and processes based on their role. This allows tailoring the provenance capturing functionality, refining the provenance graph as well as it improves re-use of already implemented Soma functionality. Firstly, we can create specialized artifacts having tailored identity functions (cf. Section 2.8.2) and reliability mechanisms (e.g., Signatures) mainly to reduce computation overhead or simplify identity determination or, secondly, use the default implementation which refers to the default (complex or simple) artifact.



(a) Manually integrated Dendrite

(b) Exemplary Soma implementation (the actual API)

Figure 4.3: Architecture mapping in program code using the static intuitive implementation technique

To sum up, the *Dendrite* is source code that needs to be integrated in the source code of the original system and calls the Soma, the actual provenance capturing API, which is encoded an additional package creating the provenance graph and allows tailored provenance capturing capability.

4.3.3 Parts of special interest and missing basic technologies

Although implementation of the whole infrastructure is challenging, from the research point of view, we want to focus on parts that impose special research challenges. For instance, in the beginning we stated that we want to create solutions having provenance capturing and storing capability. However, since we have structured provenance data in a form that is highly related to the Open Provenance Model, we can use existing approaches. Moreover querying provenance data has been intensively studied in [Cha08] and therefore, does not impose new challenges that we have to address in this thesis. In the remainder, we focus on how to integrate the Dendrites efficiently and how to store tailored provenance data. In the following, we justify our selection.

Variable provenance data and database schemas

A drawback of our approach to capture tailored provenance data is that not only the infrastructure parts, such as Dendrites and Soma are variable, but also the collected data itself is variable. The data may, for instance, include different signatures, additional fields, or functions to determine the identity of the artifacts having the same role. To really incorporate provenance data, we require also tailored database schemas to allow querying the provenance data for instance for indexing etc.

To the best of our knowledge, there currently is no approach that allows for tailored database schemas that are required by a client application, such as our provenance capturing infrastructure. Consequently, a major basic technology for our case studies is missing. As a result, we need to develop an own approach in Chapter 5 before we can start implementing first exploratory case studies. Furthermore, from a scientific point of view this part is especially interesting as we expect severe interactions between the different infrastructure elements.

Integration of Dendrites

The second part that we focus on, are the Dendrites. These parts are selected as they currently seem to be very application specific and re-using their implementation with the help of beneficial implementation techniques by simultaneous minimizing of their invasiveness imposes interesting challenges from a theoretical and practical Software Engineering such as sufficient decomposition of currently used approaches when integrating new functionality, degree of required granularity compared to traditional SPLs as well as the effects on non-functional properties. Moreover, from the provenance point of view, incorporating academic approaches in real-world provenance systems is one important step for realizing the ideas from Section 4.3.1.

4.3.4 Derived research agenda

Base on the information given so far, we present a research agenda for the remainder that shall lead to the desired insights of this thesis (cf. Figure 4.4). In fact, for the four major points contained in Figure 4.4, there is a chapter in the remainder of this thesis.

One of the primary goals of this thesis is to explore whether integration of a totally new and variability functionality (in our case provenance) into existing systems is feasible with current implementation techniques. However, as a first step, we need to perform research on the basic technology allowing to store the provenance data (as stated in Section 4.3.3). This is basic prerequisite to perform, as the next step, a series of first case studies having different characteristics (e.g., programming language, required granularity of provenance, complexity of the case study).

The basic idea of Chapter 6 is to integrate the required provenance capturing functionality into different parts of the infrastructure of our running example Digi-Dak, to demonstrate feasibility, generality as well comprehensiveness of our approach. Moreover, we are interested to identify pitfalls that we may encounter in more complex case studies. Specifically, we evaluate the effects of using different implementation techniques to integrate the provenance concern (cf. Section 4.2.1). In particular, we are interested in differences of the invasiveness of the newly integrated functionality.

We conduct first provenance integration in Chapter 7. Based on these insights, we perform a more advanced case study to verify our results and address limitations of the first cases studies in Chapter 8. Finally, in Chapter 9, we summarize our findings. This includes a call to software engineers to avoid properties of the programs that

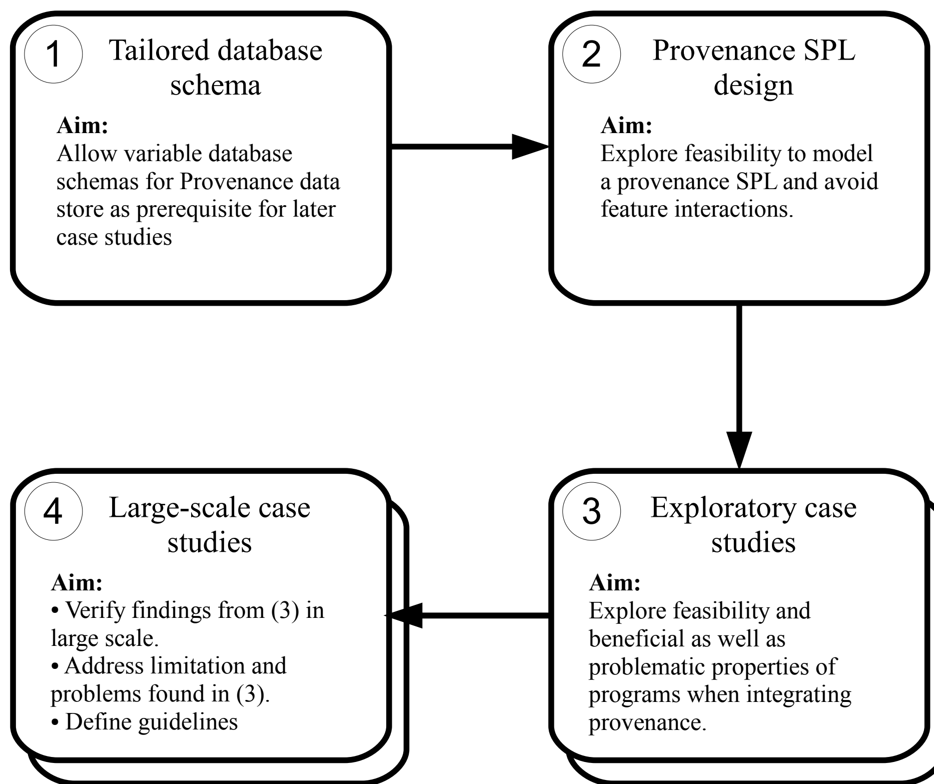


Figure 4.4: Derived research agenda

we have encountered as pitfalls and best practices as alternatives, as well as a call to use properties (and techniques) that, for instance, simplify the integration of new functionality into existing systems. As final consequence, we state research challenges based on the experiences and results gained in this thesis for future work.

4.4 Summary

In this chapter, we reason why we favor integration tailored provenance capturing functionality instead of alternative approaches. We analyze requirements regarding provenance based on a large-scale joined project Digi-Dak and results from related work and discuss limitations of current provenance capturing solutions addressing these requirements motivating the need for an own approach.

When integrating new functionality, we want to modify the characteristics of the existing system as less as possible. The integration shall be minimal invasive. To this end, we develop a notion of invasiveness to qualify differences of implementation techniques used in the remainder to integrate provenance.

Then, we develop a conceptual design of our solution including the vision to target at and architectural design. Based on that, we identify missing basic technologies and invasiveness of the infrastructure parts that really need to be integrated into the source

code of the original program as topics to focus on. To this end, we define a four-point research agenda. Each point is addressed in an own chapter in the remainder of this starting with research on variable database schemas required by the provenance data store.

5. Tailoring database schemas of provenance data stores

In this chapter, we present a novel approach to address the challenge that we need tailored database schemas to store the data of our integrated variable provenance concern. On a more abstract level, this is another important piece for holistic software product line (SPL) engineering in practice that is relevant for the whole SPL community. Our approach is based on the idea of Siegmund et al. to apply feature-oriented programming to database schemas published in [SKR+09]. We map this idea to the relational data model and demonstrate its applicability firstly in [SLS+11]. Then, we extend this approach and extensively evaluate it with an industrial-size productively-used case study with the result of significant improvements compared to traditional approaches [SLRS12]. Note, this chapter largely shares material that has been priorly published in [SLRS12].

5.1 Generalization of the basic problem

Different users of an application have different requirements to its functionality. As we want to create tailored provenance solutions, the resulting data has differing schemas. Consequently, we also need tailored database schemas to store this data. Consider as motivation a software product line of web-based solutions having a user management. On client side we have tailored implementations, while we use one global database schema for all solutions. This results in the following challenges.

Alternative and optional schema elements. There are two alternative ways (features) to perform the user management: (1) locally or (2) via an Lightweight Directory Access Protocol (LDAP) server. For the first way, we need to store the whole user profiles in the local database. This includes, for instance, login names, email addresses, and all rights of every user. For the LDAP server variant, we only need to store the user name and respective LDAP server to contact, as there

may be multiple servers within a company. Note, we store the user names locally, because not all profiles on an LDAP server necessarily shall have access to the web tool. In summary, there are optional and alternative attributes and tables. The current solution is to include all schema elements into the database schema of every variant. This results is large, highly complex database schemas. In addition, large parts of the schema possibly contain no data or dummy values and hence suggest functionality that is not included.

Constraints on optional attributes. Imagine there is a base variant where a user needs to provide a login name and an email address for authentication. For a different variant, we may also require a password. For this variant, we have to ensure that the password exists (i.e., is not null). As a result, we have to include the attribute into the database schema of *every* variant. However, we either do not define the constraint, which may lead to incorrect data, or we have to insert dummy values into this column for the base variant. Both solutions are not desirable.

Research gap. While techniques to provide variability for executable program code are intensively studied (as described in Section 3.2), knowledge on the effects at database schemas level (where we store the collected provenance data) is still fragmentary [DSC+07, Mah02, SKR+09, YPXJ09]. As a result, we need to introduce a novel approach to allow variability at database schema level in order to reach our goal of a general but tailored provenance capturing SPL.

In the following, we analyze different currently used approaches to model database schemas in SPLs. Furthermore, we discuss their limitations to justify our claim for the need of a novel approach. Therefore, we define requirements that a tailored database schema must fulfill. Note, we can create thousands of valid variants of large-scale SPLs. Thus, it is not possible to create a tailored database schema for every single SPL variant manually.

5.2 Requirements for tailored database schemas

To analyze different approaches, we consider several criteria of the process to create the single schema variants and of the resulting schema variants itself. We use two different criteria to describe the *modeling process* of a certain approach:

Modeling complexity. This criterion points out the complexity of creating the model of the (variable) database Schema, which contains the basis for the database schema variants. *Expressiveness of the model.* In traditional modeling methods, the expressiveness of the model is limited. For instance, in a global schema, an engineer cannot integrate conflicting schema elements introduced by alternative features.

Evaluation of schema variants

The requirements for the single schema variants are: *Completeness.* All database schema elements (*relations and attributes*) necessary to perform the read and write operation in

the single features, included in this variant, must be present in the variant's database schema.

Complexity of schema variants. The size of the database schema (number of *relations and attributes*) shall be reduced to a minimum. Particularly, there shall be no unused schema elements. We argue that this improves the understandability of the schema variants, which is important for maintenance and further development.

Data integrity. All integrity constraints have to be included for the schema elements in a specific schema variant, to guarantee consistent data in the data-intensive system. This includes *primary and foreign keys*, *attribute value domains*, and *not null* as well as *check* constraints.

5.3 Limitations of currently used approaches

Subsequently, we discuss three traditionally used approaches for database schemas variants and describe which problems arise when applying them for SPL development. This discussion motivates the necessity for a new approach as well as it serves as basis when analyzing benefits and drawbacks of our new approach in Section 5.5.

5.3.1 Global schema

Often, for every variant of an SPL the same global schema is applied [SKR+09]. This schema contains every schema element that is used at least in one variant. Thus, the schema contains every schema element that is necessary (*completeness*). On the other hand, major parts of the global schema can be unused in this particular variant. Consequently, the highly complex schema is unnecessarily hard to understand, which complicates maintenance and evolution of SPLs. Additionally, unused parts can impose integrity problems as stated priorly. Furthermore, the global schema does not exist in every case. As a result, conflicting schema elements introduced from alternative features cannot be defined in a global schema (expressiveness). To circumvent this problem, Ye et al. discuss that overloading schema elements (i.e., using the same schema element for different purposes instead of renaming it) is possible, but causes highly confusing database schemas [YPXJ09]. Using a global schema can be seen as the standard approach. Hence, its modeling complexity is used as reference for all other approaches [SKR+09].

5.3.2 View-based approaches

View-based approaches [BQR07] generate views on top of the global schema that emulate a schema variant for the client, which may be seen as an annotative approach [KAK08]. Thus, the global schema is still part of every schema variant. The approach inherits most of the problems of the global schema. Unfortunately, the variant's schema complexity does even increase, because the additional schema elements for the view, emulating the variant, have to be included as well. Furthermore, there is additional effort to generate

views when modeling the schema. This approach has benefits in data integrity, because the views emulating the schema variants can contain additional integrity constraints, which cannot be included into the global schema. Thus, the expressiveness of the model is also better than in the global schema approach.

5.3.3 Framework solutions

In a framework approach, the plugins implement the features of the SPL and therefore this approach is a form of physical decomposition [KAK08]. A plugin contains additional program code and an own schema. The schema variant is built from the single plugins that add the additionally required schema elements [SKR+09]. Thus, it fulfills the *completeness* requirement. Consequently, it also contains only schema elements that are needed in this variant. Unfortunately, using frameworks could lead to table partitioning, when two or more plugins use the same real world entity. Hence, this has negative impact on the complexity of the schema variants and limits the modeling expressiveness. Furthermore, consistency checking is implemented on client side, because there are no intra-plugin integrity constraints, which can lead to data integrity problems. The effort to model the schema is higher than modeling a global schema, because we have to take care of additional challenges such as naming conflicts, which are usually solved by naming conventions.

5.4 Our solution

This section contains the basic overall idea of our approach and explains how to obtain the variable database schema. Therefore, we describe the relationship between features and database-schema elements.

5.4.1 Basic idea of our approach

For explanations of our basic idea, imagine there is an SPL that consist of a client application and a database for storing data persistently. For simplicity, we again rely on the web-based solution that we already introduced in this section and its *login features*.

To create a tailored variant of an SPL, a customer first selects the desired features (see Figure 5.1(a)). Every feature contains one model for its functions in the client program and another one for the database schema (see Figure 5.1(b)). Furthermore, each of the two models of a particular feature points to different implementation artifacts (e.g., source code files). Finally, a well-defined composition strategy (see Section 5.4.3) creates the variant including the tailored client program and database schema.

5.4.2 Relationship between features and database schema elements

As defined in Section 3.1, a feature is some characteristics that is important for some stakeholder. Furthermore, the features are the units a customer selects to create a

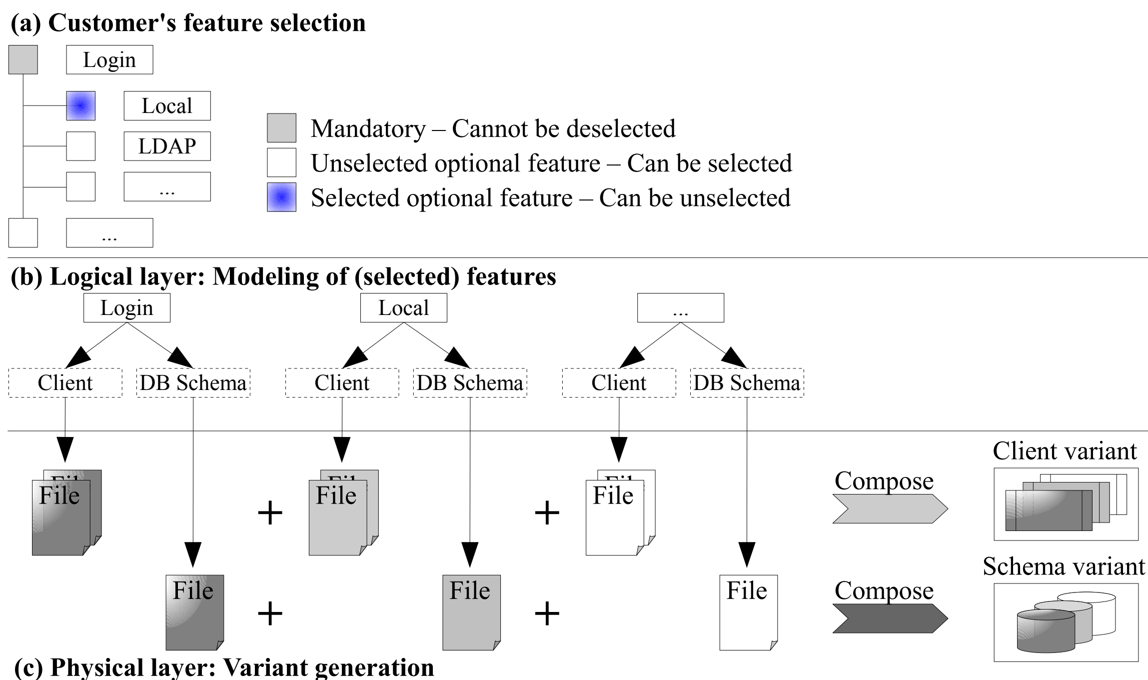


Figure 5.1: Basic idea to create a particular variant

tailored variant, which includes a database schema. Therefore, we have to define the relationship between features and schema elements to model the *variable* database schema. Moreover, we need to discuss the interaction of features on client and database-schema level. A feature at database schema level contains all schema elements (relations, attributes, and integrity constraints) that this feature on client side needs to perform the read and write operations within its specific source code on client side. Thus, we tailor the database schema with respect to the requirements of the feature on client side. As a result, there is one feature model for the whole SPL allowing us to easily generate the single variants of the SPL (see Section 5.4.3).

Note that the relationship between a feature on client and database schema level ensures *completeness* and minimizes the *complexity* of the schema variant (see Section 5.2). Every schema element that a selected feature needs is contained in the feature and therefore added to the schema variant during composition. Furthermore, the schema variant contains no unused schema elements for the same reason. Alternatively, the feature models could contain only the additionally required schema elements. This approach has the benefit that schema elements cannot be mapped to multiple features. This can lead to conflicting definitions of a schema element in *one* variant during the evolution of the SPL. The drawback of this alternative is, that we have to decide to what feature a schema element belongs, especially when they are only required by optional features. However, a high number of redundancies suggest a refactoring of the client programs source code.

Relationship to a similar approach

Meanwhile, a similar approach has been published by Brummermann et al [BKS13]. Their and our approach address the same issue as both are introduced to tailor database schemas. However, they differ in the specific application scenario and thus, also in technical details. We focus on separation of concerns and address evolution as a minor topic. In contrast, Brummermann et al. directly focus on supporting evolution in software systems. Consequently, there data model is dominated by the object-oriented decomposition not by features. Thus, from our point of view this approach is similar to priorly introduced framework solutions. Nevertheless, the existence, motivation as well a primary objective of this approach supports relevance and significance of our results.

5.4.3 Composing a tailored schema variant

We already defined that the model on database side contains a mapping of schema elements to features. Furthermore, we emphasized that the feature on database schema level contains all necessary elements for the feature on client side to perform the read and write operations of this feature.

As noted in Figure 5.1, the approach needs a well-defined composition strategy to generate a tailored SPL variant. For this reason we use *superimposition*. Superimposition is a language-independent composition strategy that has been applied successfully in software engineering [AL08] and also in view integration [BLN86]. Thus, the composition is able to handle the client implementation and the corresponding database schema. In [AKL09], Apel et al. present FeatureHouse, which offers a general support for composing software artifacts. It has been tested for several case studies on client side, written in languages such as Java, C, or Haskell.

5.4.4 Structure of features at implementation level

The $\bullet$-operator works on *feature structure trees* (FST), which represent the internal structure of the implementation of a feature. They can be seen as simplification of an abstract syntax tree [AL08]. In Figure 5.2, we show the FST on database schema level of two features of our web-based SPL that we already applied to explain the basic idea of our approach. The FST can be generated quite intuitively from the model (i.e., the feature model and the schema elements mapped to the features). The result of the composition contains all schema attributes of the input features with the desired integrity constraints. FST nodes of two features that share the same path from the root node (e.g., attribute *login_name*) are merged and thus show only once in the result as intended.

5.5 Evaluation of our approach

In the following, we apply the previously presented approach to an industrial-size productively-used case study and evaluate the results. To this end, we first justify our selection of the case study and secondly explain how we use this case study to evaluate our novel approach. Finally, we evaluate the approach.

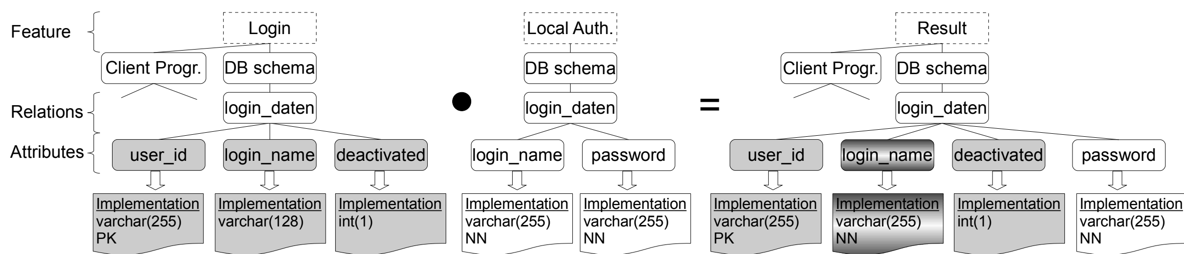


Figure 5.2: Composition of feature structure trees

5.5.1 An industrial-size case study

The ViT®-Manager is a family of industrial controlling tools for continuous improvement processes of the METOP Institute[1]. A continuous improvement process defines a structured approach with regularly (continuous) meetings to identify and solve problems within a company, which increases productivity. Because the ViT-Manager is used commercially in several companies with different size, internal structure, and portfolio, the customers have a set of equal and differing requirements regarding one variant of the application. To face these challenges, the ViT development team re-factored the application using a plug-in infrastructure to form an SPL. In the ViT framework, a customer chooses additional features according to his needs, when generating a new variant of the application. While the client program was decomposed into *features* during the re-factoring, the database schema was not. Instead, a global database schema for all variants was applied. In our prior work [SLRS12], we also defined a semi-automatic decomposition procedure to re-factor the previously used global schema into a variable one. In this thesis, we design our provenance SPL from scratch. Thus, we only briefly reflect on the properties this decomposition procedure as it shows benefits and drawbacks or our approach independent of the way to the variable schema is created.

Generally, we need to show the feasibility of our novel approach. To this end, we use the introduced case study for that we created a variable schema and evaluated the ability to integrate new features into the initial variable schema for more than a year. Moreover, we use our experiences from this case study for our future case studies to avoid non-obvious pit falls that we discover with the ViT-Manager case study.

Evaluation goals

We first analyze the feasibility of our approach before discussing its impacts on maintenance, data integrity, and further development. Finally, we evaluate the overall advantages and drawbacks of our approach compared to the traditionally used ones from Section 5.3. To ensure validity of the evaluation, we additionally rely on interviews of the ViT Development Team, which is experienced in applying global database schemas to SPLs, because the case study previously used a global schema. Furthermore, this team tested the variable schema approach for more than a year. Finally, we strengthen our conclusions with specific measurements if possible.

[1]http://www.metop.de/site/ai-angewandte-informatik/produkte/vit/

5.5.2 Feasibility of the approach

We want to evaluate whether the variable database schema approach is manageable in practice. Consequently, we have to evaluate the modeling process, the result of the modeling (features), and the variant generation process.

Creating the variable schema

To create the variable schema, there are two possibilities:

1. Refactoring a previously applied global schema into features based on the client implementation.
2. Define the features from scratch (including the respective schema definition).

In the following, assume that the client is an SPL allowing us to map code fragments to features, which is required to decompose a global schema.

Decomposing a global schema

When we created the variable schema for the case study (having at this time about 50.000 lines of code) for first time in [SLS+11], we performed the mapping of schema elements to features mainly *manual*. The basic idea is to identify database queries in feature-specific source code and identify the required schema elements. The procedure we used is as follows:

Step 0: Identify *all* database query executions. For every schema element that is required by a feature, there has to be some database query called from feature-specific code that uses this schema element. That means there is an SQL statement that contains this schema element. As executing a query requires a database driver (e.g., JDBC, ODBC, etc.) with a fixed set of methods, we automatically search the source code for these patterns. For instance, in the example in Figure 5.3 in Line 18, the method `executeSQL(String)` is called that represents such a pattern. For each of these patterns found, we then perform Steps 1 to 4.

Step 1: Get text source of SQL statement. In this step, the complete text source of the SQL statement is identified, in order to find the required database schema elements in the next step. For the example (Line 17), this identification is rather simple, similarly to the most SQL statements of our case study. Nevertheless, the SQL statement may be assembled over multiple lines, modified by optional features, or totally depend on user input making this step extremely challenging or even impossible for *some* database query executions.

Step 2: Analyze SQL statement source. In this step, we analyze what schema elements are used in the respective SQL statement. Note the definition of these schema elements (e.g., data type and constraints) are determined automatically using tool support. Consequently, in this step we only need the qualified path of every schema element consisting of the table name and the attribute name.

Step 3: Get feature context. To add the schema elements, identified in Step 2, to the correct feature(s), we determine the feature context. That means that we determine to which feature the source code fragments belong that contain the definition of the SQL statement. In Figure 5.3, the function that defines the SQL statement is called within the feature *Local Authentication*. Thus, this is the feature context. Note that there may be multiple features and determination of the feature context is a challenging task shown in [Sch12]. However, the decomposition technique simplifies this process. For instance, if we use physical decomposition, the feature context is defined by the file the source code is located in.

Step 4: Add schema elements to feature definition. In the final step the identified schema elements are added to all features of the feature context. In case a schema element is already part of the schema definition, we do not add it again.

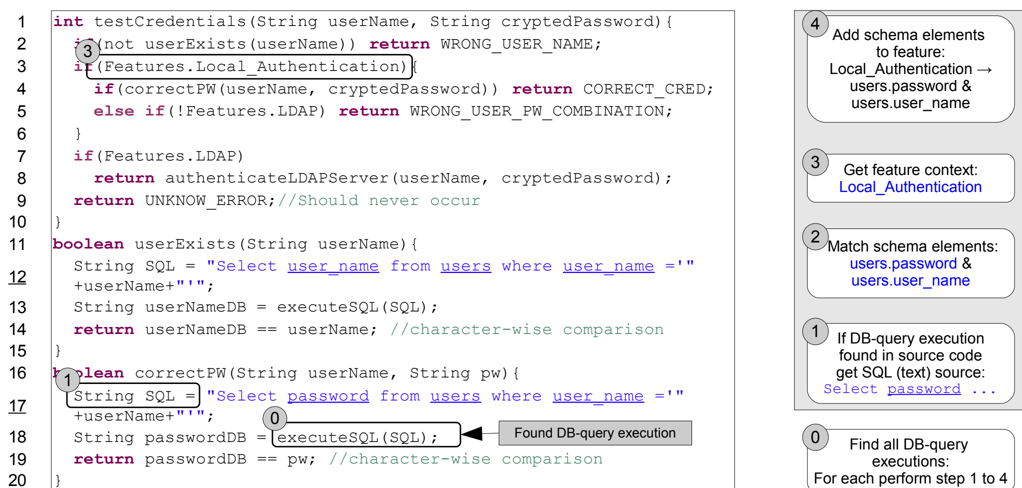


Figure 5.3: Procedure to map schema elements to features based on client implementation

Although we could use tools to identify all database query executions (step 0) and had developers with deep knowledge on the structure and semantics of the source code, the complete mapping of database schema elements took about two days of work and turned out to be laborious. Altogether there are three pattern classes and, in total, about 1.500 occurrences of these patterns. Nevertheless, even considering this effort, applying our approach in practice is manageable but currently not desirable. To address the aforementioned issue, we offered tool support and provided more automation.

Tool support to model the variable schema

To support the decomposition of a global schema, we developed a tool in our prior work [Sch10]. The tool offers the following support:

Mapping visualization. The tool visualizes the current mapping of schema elements to features in an $n \times m$ matrix (cf. Figure 5.4).

Mapping modification. The visualization matrix also allows to modify the mapping, by simply clicking on the respective check box.

Schema variant generation. Given a configuration for a particular variant, the tool is able to generate the variant's database schema in form of an SQL script. Currently the tool supports MySQL and Oracle SQL syntax.

Automatic schema element definition determination. To automatically generate a complete schema, we need the data types and integrity constraints. In Step 2 of the decomposition procedure, we only determined the qualified path, but not the integrity constraints etc. Furthermore, we need all schema elements of the previously applied global schema to map them to the features and depict this mapping. As manual insertion into our tool is laborious and error-prone, we determine this information automatically. The basic idea is that there already exists a consistent global schema. To this end, we can analyze the schema catalog of the used database system and import the desired information.

Menü Speichern

	bvw_history	Base	Workflows	BVW	ViT	FMEA	ViT-Basic	...
1	Datum			✓				
2	Geaendert_von		✓					
3	Hist_id		✓					
4	Nachher		✓					
5	Operation		✓					
6	Vorher		✓					
7	Vorschlags_id		✓					

	dyn_bvw_anlagen	Base	Workflows	BVW	ViT	FMEA	ViT-Basic	...
8	Anlagen_id		✓					
9	Geleoscht		✓					
10	Name		✓					
11	Pfad		✓					
12	Vorschlag_id		✓					

Figure 5.4: Tool support: Screen shot of mapping matrix (taken from [Sch10])

The tool support is currently based on PHP and thus, not integrated into modern IDEs, such as FeatureIDE, which is desirable, but not mandatory for our purposes.

Limitations of automizing the decomposition process

According to our experiences the most time consuming step when decomposing a global schema, is to perform the mapping of schema elements to features. To this end, we explored how to automize the manual process as much as possible [Sch12]. The major result is that we can provide a first and correct mapping, which contains a *sub set* of mappings. Moreover, we can annotate source code fragments for manual review in case one of the decomposition steps fails. Note, we also revealed it is not a problem if the automatization fails for some SQL statements to produce a quite comprehensive mapping as most schema elements are used in multiple SQL statements.

First conclusions

To sum up, with our additional improvements, namely tool support and automatically providing a first mapping, we conclude that our approach is generally feasible in practice.

As stated in Section 5.4, the modeling process, which includes the decomposition of the global schema into features, is laborious. Furthermore, the decomposition procedure is not generally automatable, because because some database queries depend on manual input or assembled using complex logic. Consequently, the procedure needs manual recall. However, according to interviews the modeling is still manageable. Moreover, the modeling effort is an investment for the automated generation of a tailored database schema variant. A customer simply chooses the desired features and the database schema (and the client) are generated automatically. Additionally, the size of the features scale in the case study (see Table 5.1), because the size on client and on database side seems to correlate. This effect does not change significantly over time. As visualized in Table 5.1, even when comparing two major releases such as v1.6 and v1.7 the correlation exists.

Table 5.1: Correlation of size on database and client side

Features v1.6	Rank Size DB/client	Attributes	Relations
ViT	1 / 1	88	16
ViT-SPO	2 / 2	80	14
BVW	4 / 3	75	12
ViT Square	3 / 4	65	12
ViT Basic	5 / 5	63	12
...			
LDAP	24 / 21	1	1

Features v1.7	Rank Size DB/client	Attributes	Relations
Archiving	1 / 4	124	17
ViT-SPO	2 / 1	105	16
ViT	3 / 2	88	16
BVW	4 / 5	75	12
ViT Square	5 / 3	66	12
...			
LDAP	30 / 26	1	1

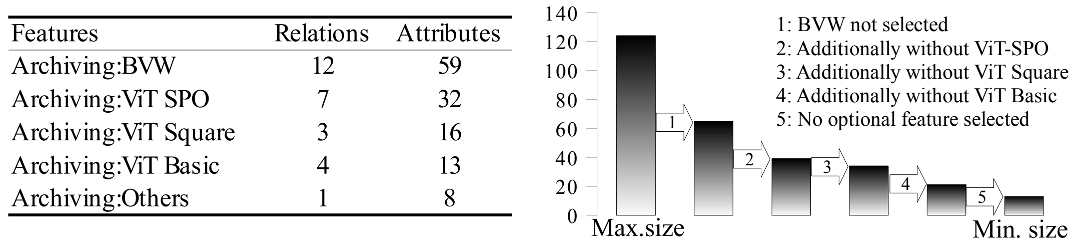
Additional challenges due to feature interactions

We identified two additional challenges of the variable schema approach that we try to avoid for our provenance solutions: (1) Redundantly defined schema elements and (2) changing feature models. Redundant schema elements increase the size of a feature on database side. Thus, the features were extracted from global schema, two features that can be present in one variant cannot contain different models of the same schema element, which would produce an error during composition. But, when modeling the features without a previously used schema this challenge has to be faced, which exists also in view integration.

The second challenge results because of the combination of optional features. The schema variant might need additional schema elements when a customer chooses two optional features to generate a variant. These schema elements are not part of the schema variant when only one of them is chosen. This effect is also known on client side (e.g., glue code), but to the best of our knowledge it has not been identified at database schema level.

In the case study, there is an optional *Archiving* feature that archives data of different *optional* features, such as *ViT basic* or *BVW*. Thus, these optional features are not part

of every variant. Therefore, the archiving feature does not need the tables to archive the *BVW* data if the *BVW* feature is not included into the variant. To minimize the complexity of the schema variant, these unneeded relations should not be included. Hence, we move the additionally needed schema elements into *derivatives*, which are included automatically when both optional features are part of the variant [LBL06]. Particularly, the composition mechanism includes *Archiving:BVW* (cf. Figure 5.5) into the schema variant, when a customer selects *BVW* and *Archiving*. Using derivatives raises the expressiveness of the model and decreases the size of the *Archiving* feature in some variants dramatically, by making the modeling process more complex.



Features	Relations	Attributes
Archiving:BVW	12	59
Archiving:ViT SPO	7	32
Archiving:ViT Square	3	16
Archiving:ViT Basic	4	13
Archiving:Others	1	8

Figure 5.5: Derivatives: Size of the Archiving features in different feature combinations

Conclusion - Trade-off

According to our interviews with the developers, the trade-off between the drawbacks of our approach, such as additional modeling overhead, and the advantages like automated variant generation are beneficial only when handling multiple variants of an application. But the specific costs, such as modeling the variable schema, related to the amount of time saved (e.g., for manually tailoring a variant or additional bug fixing if using a global schema) to the customer's needs, is imprecise for the following reasons. The complete costs for alternative solutions are unknown, because these approaches have not been implemented and the previous costs can hardly be taken into account because the tasks are highly different. Moreover, there are a lot of fine-grained costs where the specific amount of time is unknown. For instance for bug fixing in the variable-schema approach, we do not know the amount of time, as only the time from reporting the bug until fixing it is known. After all, the numbers we have and interviews with the developers suggest that for the *ViT-Manager* the variable-schema approach is beneficial when there are at least five (different) variants used productively.

5.5.3 Improving maintenance and further development

Here we argue that not having unnecessary schema elements in the variant schema is a suitable way to reduce its complexity. The effort to maintain a schema variant becomes more predictable, because we have a mapping of a particular function to schema elements (features). Thus, the estimation of possible side effects becomes easier.

The complexity of a schema variant is minimized, because it contains only necessary schema elements. Therefore, the understandability of the database schema variants

rises and is additionally supported by the mapping of schema elements to features. When comparing the complexity reduction of the schema variants in different versions (cf. Figure 5.6), the benefits of the variable schema become even more obvious for the case study. The minimum configuration (no optional feature selected) of v1.6 covered about the half of the maximum configurations schema elements (all optional features selected). Whereas, the difference in v1.7.6 is about one fourth. Furthermore, the database schema of the minimum configuration remains nearly stable. This is, because many new *optional* features have been added to the case study, but the core functions remained roughly unchanged, which is common in SPLs. Thus, we conclude that the benefit of our approach increases over time, when new optional features are added to the SPL.

According to our interviews, after introducing the variable schema the number of support requests slightly increased for a quarter of a year, especially because of problems related to the database schema. After this, there was significant decrease of support request even if introducing new features. We hypothesize that one reason therefore is the variable schema, but we cannot say to what extent.

Similar observations have been made for further development. Extending the case study with new features becomes easier and is more predictable in the variable database schema approach. First, a software engineer can simply add new features to the composition process. Furthermore, the mapping between features and schema elements is helpful when designing new features or extending existing ones. This was highly useful, when designing the *Archiving* feature and its *derivatives*, because for every feature it is known, which schema elements are necessary and thus have to be archived. Challenges arise when modifying schema elements used in different features in preserving the model's consistency.

In addition, there are naming problems when adding schema elements of new features. In particular, we identified three naming conflicts, which are well-known in schema matching and view integration [BLN86]:

1. Contradicting definition of the same schema element for instance regarding the data type or length.

2. Double definition of semantically the same schema element with different name (synonym).

3. Using erroneously the same name for semantically different schema elements (homonym).

From our experiences the first two problems are the most frequent ones. In Section 5.5.4, we already discussed how we avoid the first conflict with tool support. Similarly, we address the second conflict. In case a developer wants to add new schema elements using our decomposition tool, there is suggestion of ten *similar* schema elements even while typing the name of the new element. Similarity is defined by (1) a schema element

having the same name, (2) elements that enclose the current input (i.e., the input is a sub string of existing schema elements), and (3), in case there are less than ten similar existing schema elements found we apply the edit distance. By concept, it is also possible to extend the similarity function, for instance to use ontologies etc.

For the case study, the third conflict was not found. Thus, we do not provide a dedicated tool support and adjourn resolving this conflict to the developer.

5.5.4 Improving data integrity

One of the strongest points supporting the variable schema approach is the benefit regarding data integrity. Using the variable schema allows to reintroduce the integrity constraints that had to be dropped (see Section 5.3), because of the global schema approach. Additionally, integrity constraints that were encoded on client side can now be added to the database schema. As a result the variable schema approach *ensures integrity* of the data and is therefore beneficial here. Furthermore, tricks as inserting dummy values to circumvent Not Null constraint for attributes not used in this variant are not necessary any more. Therefore, the variable schema approach is beneficial for *data quality* as well.

Moreover, decomposition procedure guarantees that we can only introduce valid integrity constraints and that additional restrictions of the variability of the SPL (introduced by foreign keys referencing from an optional feature to a different optional feature) are visible in the feature model. Nevertheless, contradicting schema-element definitions may be introduced:

1. By intention - In case an optional feature shall override a previous definition (e.g., data type).
2. By mistake - When designing the variable schema from scratch or during evolution of the SPL.

To address this challenge, we use *tool support*. In particular, there is an additional view in our decomposition tool that shows contradicting schema-element definitions. Moreover, the current feature order is visualized and can be modified to define which shall be the dominant definition. However, in the case study overriding a schema element definition is not required, even after an evolution of more than a year.

5.5.5 Comparison to existing approaches

In Table 5.2, we resketch the result of currently used approaches of Section 5.1 and compare them to the variable schema approach. The scoring of variable schema is based on the discussion in Section 5.3. The variable schema has strong benefits due to the complexity reduction of schema variants and the improvement of data integrity (also helping to improve data quality). Furthermore, the model expressiveness increases, which

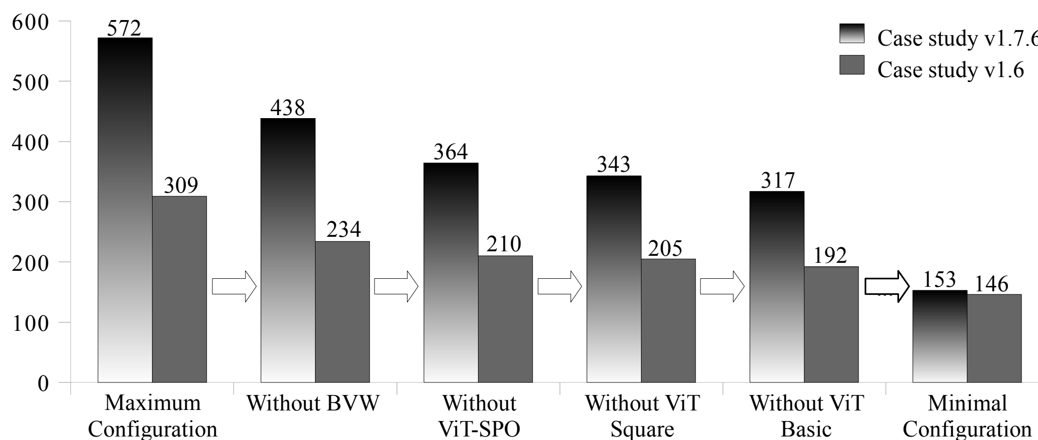


Figure 5.6: Complexity reduction of the schema variants in different versions of the case study

allows conflicting schema elements and changing feature models that is not possible with a global schema approach. By contrast, the global schema approach and frameworks have benefits regarding modeling complexity. The variable schema needs a previously used global schema, which has to be decomposed into features. Alternatively, the modeling of features instead of views, including all derivatives is also more complex than modeling one global schema. This is the trade-off our approach needs to improve maintenance and further development.

Table 5.2: Evaluation of the variable schema approach compared to currently used approaches

	Modeling		Schema Variants		
	Complexity	Expressiveness	Completeness	Data Integrity	Schema Complexity
Global DB Schema*	+	--	++	-	-
View Approach*	--	-	++	+/-	--
Framework Solutions	+/-	+/-	++	-	+
Variable Schema	-	+	++	++	++

++ very good, + good, 0 neutral, - unhandy, -- very unhandy *Approach only possible if global schema exists.

5.6 Role and contribution of the approach for this thesis

Our novel approach allows to store the data of our provenance solutions using customized database schemas. Moreover, we benefit from the experiences gained when applying the approach for the industrial case study. In particular, we need to avoid feature interactions. Whether we can avoid such interactions when designing the provenance SPL from scratch, including respective reasoning, is one question we want to explore with first case studies using the provenance SPL. Furthermore, we want to avoid problems with respect to referential integrity. Especially, foreign keys that point to schema

elements that possibly do not exist in a particular variant. With our decomposition approach to re-factor a *consistent* schema into a variable one, we have rules to avoid this problem. Whether we can (generally) avoid this problem, when designing an SPL from scratch, and whether additional, currently unknown, problems exits, needs to be determined.

6. Database-centric chain-of-custody

Before, we can start to implement the case studies, we need a proper selection of such case studies. In addition, we need to know what provenance functionality is required. To this end, in this chapter, we contribute the concept of the database-centric chain-of-custody as a general concept that finally allows a proper selection of the exploratory case studies. Secondly, we design our desired provenance SPL that defines the provenance functionality that is integrated into the case studies in the remainder of this thesis.

6.1 Analysis goal and resulting methodology

A fundamental objective of this thesis is to explore whether it is possible to integrate the provenance concern into existing systems in a minimal-invasive way using different implementation techniques. Moreover, we are interested in revealing differences between the single implementation techniques. In this chapter, we prepare the conduction of the first exploratory cases studies in the next chapter.

To reveal insights regarding this objective, we need a sound selection of case studies. In Section 4.1.1, we already justified why using case studies from the Digi-Dak project delivers sound and comprehensive results in general. However, for our first exploratory case studies, we have to consider additional properties:

Holistic approach. We need a concept that allows us to argue that the selected case studies represent all major parts of the infrastructure used in the Digi-Dak project. This prevents us from choosing an improper selection of cases studies and hence, ensures comprehensiveness of our analysis.

Technical heterogeneity. As we want to provide a (to some extend) general solution, we require case studies that use different paradigms such as object-oriented vs. relational data model or are different in complexity and decomposition approach.

Functional provenance heterogeneity. Moreover, we need case studies that require different provenance functionality, so that we gain data on the how to integrate a variable cross-cutting concern into existing solutions.

6.1.1 A holistic approach

To provide a holistic approach as requested, we suggest the concept of a database-centric chain-of-custody that we published in our prior work [SSK11]. The contribution of this concept for this chapter is to group solutions used in the Digi-Dak infrastructure in classes that share the same requirements for provenance integration. Moreover, we use these classes to define the required provenance functionality based on a threat model where provenance functionality offers respective countermeasures. Then, the basic idea is to select a case study from each group and integrate the required provenance functionality.

Infrastructure boundaries: Considered excerpt of the chain-of-custody

The future goal is to provide a system that ensures the chain-of-custody from the acquisition of a trace at a crime scene to its final usage in court. The Digi-Dak infrastructure, however, is responsible for ensuring only a part of the chain-of-custody. To this end, we first define the boundaries of the system. That means, we define the initial and final `artifacts`. Then, we take care of the intermediate results to document the chain from its initial `artifacts` to its final `artifacts` and define granularity refinements.

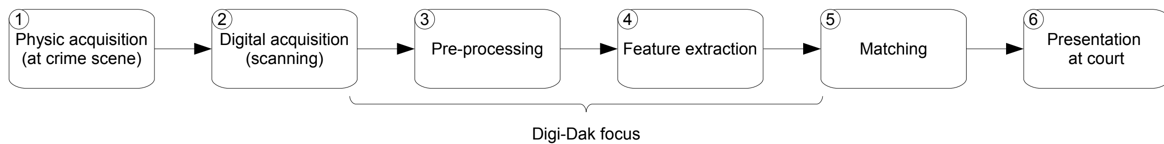


Figure 6.1: Digi-Dak's chain-of-custody excerpt

In Figure 6.1, we depict a simplification of the major steps from initial acquisition of a trace until its final deposition as well as the respective excerpt the Digi-Dak infrastructure is responsible for [HKGV11]. To clarify our intentions and to clearly mark the borders of the resulting case studies, we now explain the major steps, which are in fact highly simplified processes.

Physical acquisition. In this thesis, physic acquisition contains all necessary actions taken by law enforcement agencies at a crime scene, including transportation, in order to secure all possibly relevant assets that may contain evidences. Examples are usage of brush and powder to reveal fingerprints or removing parts or any objects (e.g., large furniture that cannot be transported) that may contain valuable evidences. These actions are performed by law enforcement agencies, using long established technical and organizational means to ensure the chain-of-custody. Hence, our infrastructure is not responsible for this part of the chain-of-custody.

Digital acquisition. The overall Digi-Dak infrastructure works on digital artifacts (primarily images). To this end, different sensors are used to create a digital image of each trace. In Digi-Dak, it is not intended to develop novel sensor technologies or devices for acquisition of fingerprints or other traces [Kom05]. Consequently, it is presumed that these sensors work correctly and thus, have limited access and knowledge on internal details of all sensors. Thus, we only capture sensor and (important) environment parameters, but treat the whole scanning process as black box. However, immediately after the creation of digital `artifact(s)`, we are responsible for ensuring the chain-of-custody.

Preprocessing. In order to increase the quality of a trace, such as a fingerprint, several preprocessing steps are performed. This includes for instance correction of optic distortions due to non-planar surfaces [KCDV12] or separation of overlapping fingerprints.

Feature extraction and quality estimation. Based on preprocessed images, there are several tools that extract features. An example is to estimate the point in time when a fingerprint was laid. In contrast to preprocessing steps, feature extraction results into data that represents a desired result, from a forensic point of view. For the Digi-Dak research project, these steps are important as they first result in final `artifact(s)` and second contain new challenges to manage research data. One of the core challenges in the project is to evaluate different well-known approaches or invent novel ones. Consequently, in the project there is plenty of scientific data that has to be managed, including respective provenance, as well. Despite the additional requirement of managing the scientific data, there is no difference between preprocessing and feature extraction from the provenance point of view, because both steps are implemented as object-oriented programs. Consequently, we treat them the same way and need to select only one case study.

Matching. The Digi-Dak infrastructure does *not* provide support for matching of fingerprints. However, the results produced are input for subsequent systems, such as the AFIS[1], or manual evaluations. To this end, we have to provide provenance information so that any subsequent system or expert trusts the delivered data. Consequently, the Digi-Dak infrastructure is responsible for ensuring the chain-of-custody until sending the final `artifact(s)`.

Presentation at court. As we are no experts in law proceedings and legal regulations differ among different countries, we do not go into detail here. Nevertheless, documentation at a level of granularity that can be presented in court must be stored in a reliable manner. Thus, any provenance data has to be stored persistently in a reliable way as well.

To summarize, the Digi-Dak infrastructure is responsible for ensuring its excerpt of the chain-of-custody. It starts with sensor images having required provenance data to

[1] http://www.bka.de/DE/ThemenABisZ/Erkennungsdienst/Daktyloskopie/AFIS/afis__node.html

ensure authenticity and integrity (initial `artifact(s)`), contains all intermediate and initial `artifact(s)` for that we capture the required provenance data, and stops at final `artifact(s)` that are send in appropriate format to subsequent systems, such as the AFIS system or manual evaluation of fingerprint experts.

A first analysis of the existing infrastructure

When looking at the already existing infrastructure of the Digi-Dak project from a software product line point of view, we discover that it can be modeled as software product line as we illustrate in Figure 6.2. The feature model mainly consists of a flat tree with several optional features that represent the single tool-chains. These tool-chains may have alternative implementations that apply different image processing filters but serve the same purpose. Moreover, nearly all feature extraction tools can directly work on the initial `artifact` (the sensor), but deliver better results with previously applied preprocessing steps. This is the case, because the result of preprocessing steps is, as the name suggests, an *image* with enhanced quality.

There are only a few dependencies. For instance, a trace is first scanned with low resolution. Then, potential regions that contain a fingerprint are marked and finally the marked regions are scanned with higher resolution. The goal is to save scan time as not the whole trace has to be scanned with high resolution. However, the result is again an image that can be processed by subsequent tool chains. Anyway, the main result for this thesis is that:

1. the tool chains can be considered independent from each other,
2. the results of the tool chains define the granularity for the chain-of-custody.

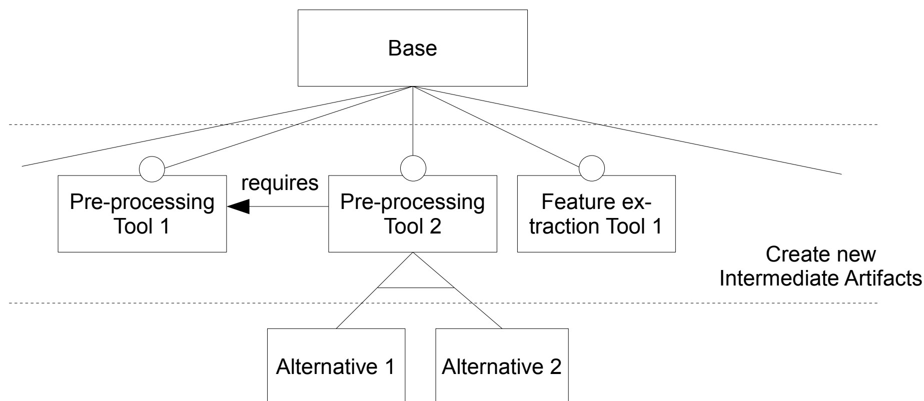


Figure 6.2: Analysis of the existing infrastructure

Granularity definition of the chain-of-custody

In our previous work [SSK11], we defined the required provenance information to ensure the chain-of-custody as follows. First, there is an initial `artifact`, in our case a sensor image, for that we rely on delivered provenance information to ensure that this artifact holds integrity and authenticity (i.e., is not faked). To produce the final `artifact(s)` of our infrastructure there are several intermediate ones. For each artifact in our infrastructure, we define the required provenance information as a *ProveSet*. The ProveSets are the links that from the chain-of-custody and are in fact fragments of a provenance graph. For the resulting graph, we have special requirements to ensure that we can reproduce any artifact (i.e., show that the chain-of-custody holds):

1. Every `artifact` with forensic relevance (defined by respective experts) itself has to be stored, for instance, to verify results by re-producing them.
2. All involved processes are in fact functions that allow to re-produce any artifact in case the same input is provided. Thus, we have to know the input `artifact(s)` as well.
3. For every intermediate or final `artifact`, we need to be able to identify the initial artifacts (sensor images), which is a starting point for any computation step for this excerpt of the chain-of-custody. This means that we can track back the provenance graph until we reach initial an `artifact`. This graph fragment then is named as Complete Prove Set *(CProveSet)*.
4. The applied transformation has to be semantically sound. However, we are not responsible for proving soundness, but provide functionality to define and check the collected provenance information.

6.1.2 Assumptions and architecture

The basic idea of the database-centric chain-of-custody is to model all transformations along the way to produce the final `artifact` as functions. Moreover, we define respective identity functions to prove identity holds for two artifacts having the same role. As a result, it allows reproduction of each transformation step. To this end, we need all input to re-invoke any process with same input that shall result in the same output. All `artifacts` that need to be stored persistently are stored in databases, such as our research prototype Fingerprint Verification Database (FiVe DB) as depicted in Figure 6.3. Moreover, any input requested from a database and newly created intermediate artifacts are stored in (a possibly differing) database as well. It is not allowed to directly pass intermediate results that shall be stored persistently, to different tool-chains.

This way, we ensure that we first store each intermediate artifact of interest persistently and second know which input artifacts are sent to a transformation, as well as the role of the expected result. This forms a first part of the desired ProveSet that is

send to the tool-chain together with input artifacts. However, to re-invoke any process expecting the same result, tool-specific configuration parameter may be required, such as thresholds for image filter functions etc. These additional input artifacts are provenance information that are added to the ProveSet by the tool-chain and are sent back to a database together with result artifact.

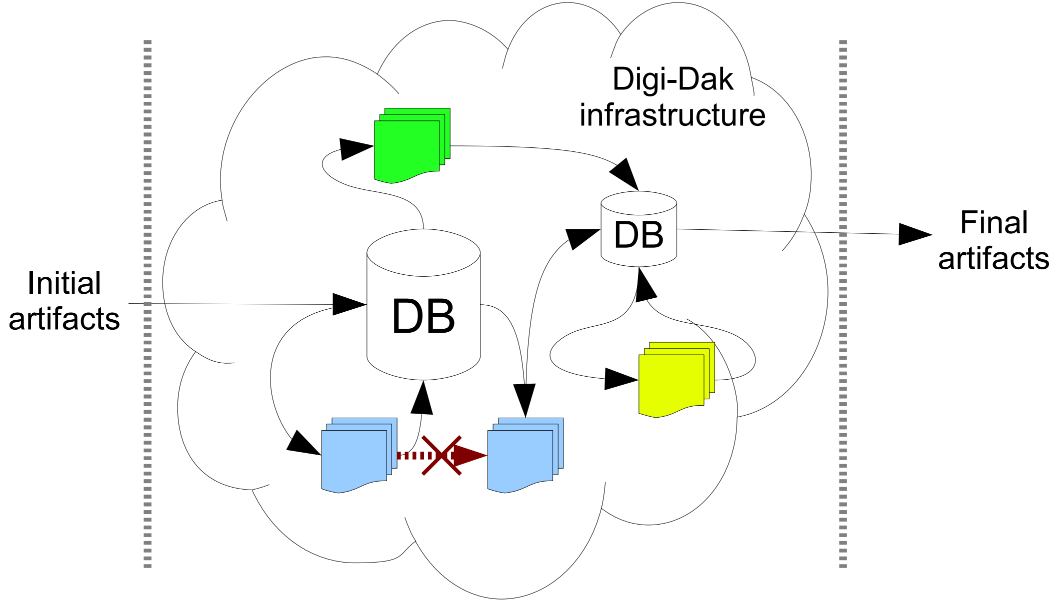


Figure 6.3: Concept database-centric chain-of-custody

ProveSet definition

As stated, a ProveSet is a single chain link for the chain-of-custody. More formally, this means that each ProveSet is linked to exactly one artifact (A_k). We write this as ProveSet(A_k). Furthermore, each ProveSet contains several attributes that we define in Equation 6.1.

$$ProveSet(A_k) := \{\{Link, Role, Process, Input, Parent, Children\}, Security\} \quad (6.1)$$

Now we explain the purpose of the single attributes and link them to the data model of our provenance framework from Section 2.4.1. Recapitulate that there are two dimensions of provenance: fragmentation and uncertainty. For the first dimension forming the provenance graph and allows granularity refinements, we use the $\{Link, Role, Process, Input, Parent, Children\}$ attributes. While for the latter dimension, we have a Security attribute that is used to reduce the amount of uncertainty. Now, we explain the semantics of all attributes.

Link. This attributes links the ProveSet to the artifact itself. This can be done in arbitrary ways, such as foreign keys in databases or pointers in object oriented programming.

Role. A role states the semantics of this artifact. This may be a table name (with respective schema) or class in object-oriented programming, but also abstract roles for real-world artifacts.

Input. This attribute contains *all* links to all input artifacts that are used to create A_k. Therefore, it is similar to the Link of the ProveSet to the associated artifact.

Process. The process that created A_k using the input artifacts that are contained in the Input attribute of the ProveSet. Each process has a name and an invocation number that links an execution of this process to the creation of A_k.

Security. To improve trust in the ProveSet and artifacts itself, it is possible to add cryptographic data, such as signatures and hashes. In contrast to the first dimension, we have no explicit structure how a security attribute shall look like. Mainly, we refer to the aforementioned techniques of signatures and hashes. However, the basic purpose of this attribute is that all data that reduce the amount of uncertainty are explicitly defined and not implicitly contained.

Derived threat model

Our architecture decisions and assumptions for the database-centric chain-of-custody result in a threat model that defines where we need provenance support and finally determines the case studies applied in the remainder of this section. Note, an instance of this threat model has been part of prior work [SSK11] and thus the model presented here is based on our explanations in the referred prior work.

Faking initial artifacts. There is a certain risk that initial `artifact(s)` are faked or a result of a sensor malfunctions. Consequently, we need provenance data that minimize this uncertainty.

Tampering the provenance data store. When an `artifact` including its provenance data is stored persistently it is possible that this data is modified for instance due to implementation errors, hardware errors, or manual interferences.

Interfering the communication channel. As any `artifact` is stored in a database and has to be send to a transformation tool as well as the result is shipped back, it is possible that it is modified in the communication channel. This includes modification of artifacts, removing, or tampering of the associated provenance data.

Malicious tool behavior. Due to implementation errors, improper usage, etc. a tool may modify artifacts or the associated provenance data in a malicious way.

Creating a new intermediate artifact.

Our recent argumentation revealed that our primary granularity focus to ensure the chain-of-custody in form of ProveSets and thus to capture provenance, are the creation of new intermediate `artifact(s)`. Consequently, we explain the procedure to create a new `artifact(s)` when using a database-centric chain-of-custody. Note, receiving initial `artifact(s)` or sending final `artifact(s)` to subsequent systems are special forms that consist only of parts of this procedure. Thus, we only have to explain the creation of inter mediate `artifact(s)` as it covers all other two cases.

Creating a new intermediate `artifact(s)` means that some tool or tool-chain in the Digi-Dak infrastructure requests one or several data items. For our explanation, we assume that an `artifact` (data item) A_k is requested from a tool to remove distortions due to non-planar surfaces. Therefore, the tool requests a sensor image with distortions and produces an equalized image. For simplicity, we assume that the image with distortions is stored in the same database as the resulting image. The overall procedure is illustrated in Figure 6.4 in detail. Note, we omit technical details as there are possibly multiple solutions to implement the single steps distracting from core concepts. Anyway, a detailed technical discussion is part of the case studies in the remainder of this section.

Step 1: Data request. The procedure is started by a transformation tool with requesting the input `artifact` (A_k) from our Fingerprint Verification DB. Then, the DB checks whether it trusts the tool and the tool may access the requested data. In the positive case, Fingerprint Verification DB checks whether A_k has a valid *CProveSet* with the help of a *provenance library.* If so, A_k, including respective provenance data, is send to the requesting tool. The consequence is that we assume (with a certain residential risk) that for any `artifact` A_k that is requested from a database and input to some preprocessing, the chain-of-custody holds. In case the tool is not trusted, the data request is declined and in case the *CProveSet* does not exist there is an alert, because Step 3 should ensure that this never happens. Thus, we assume that this is most likely a result of hardware errors or manipulation.

Step 2: Perform transformation. In this step, the transformation tool checks the existence and validity of the provenance data (i.e., ProveSet(A_k)). If it does not exist or is not valid, we assume the data has been modified during the transportation from Fingerprint Verification DB to the tool, as Step 1 ensures a database only sends `artifacts` that have a valid *CProveSet.* In case ProveSet(A_k) exists, it has to be removed from A_k to perform the transformation and it must be stored to create ProveSet(A_{k+1}). The transformation tool now creates the new intermediate `artifacts` A_{k+1} as $t(A_k)$ and adds ProveSet(A_{k+1}) to A_{k+1}.

Step 3: Insert new intermediate result. In the final step, the transformation sends a request to Fingerprint Verification DB to insert the new intermediate result A_{k+1}. Before accepting the request, Fingerprint Verification DB checks whether

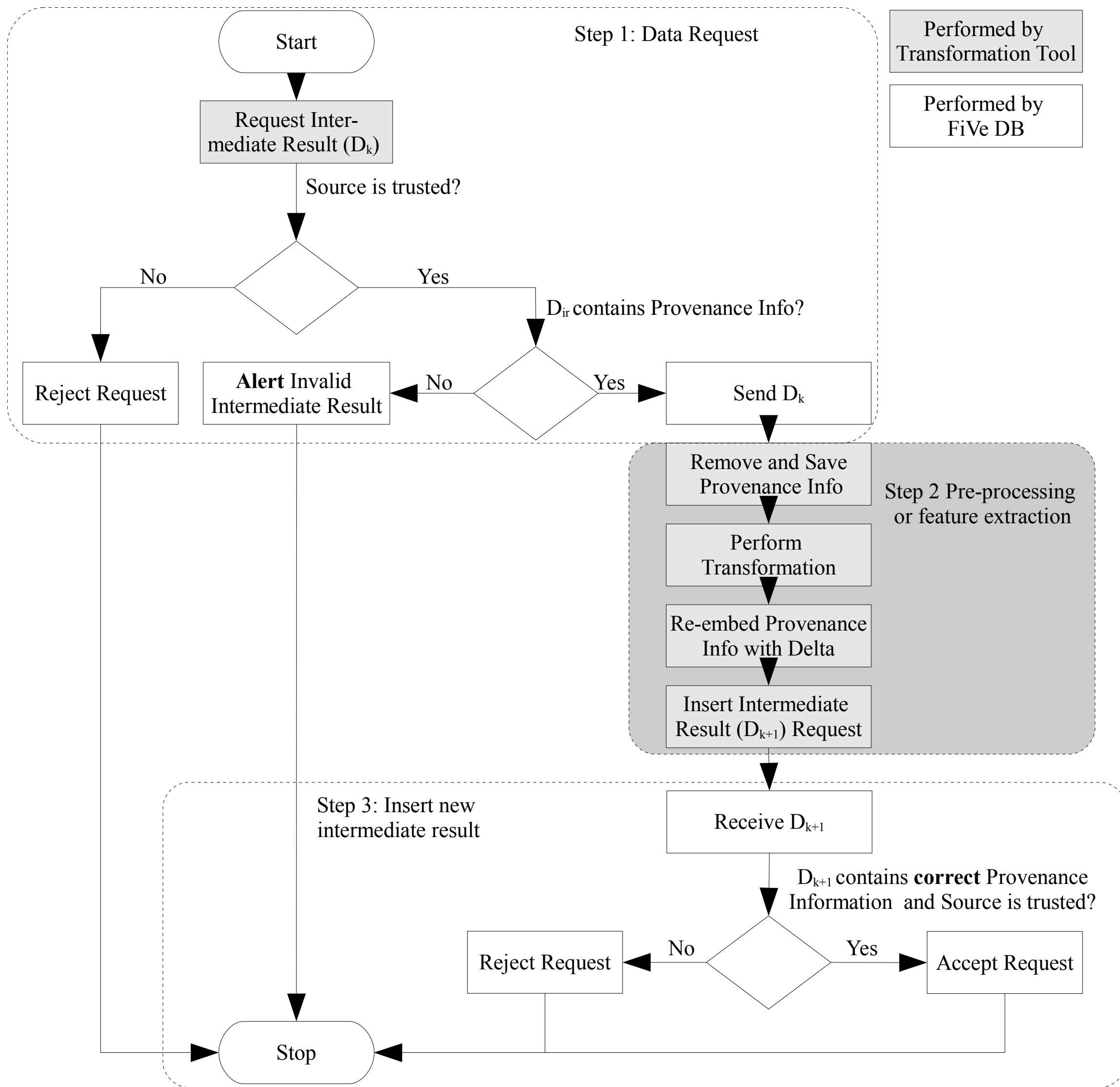


Figure 6.4: Create a new intermediate artifact

the tool is trusted, ProveSet(A_{k+1}) exists and if the CProveSet(A_k) enhanced with ProveSet(A_{k+1}) forms a valid CProveSet. As a result, we can assume that whenever an `artifact` is send to a tool or inserted into FiVe DB the corresponding CProveSet is valid and forms a chain-of-custody.

Rejecting request and alerts. For security reasons, there have to be additional mechanism such as, routines that detect incorrect *CProveSet*, prevent data requests from untrusted tools, consistency checks etc. However, as these mechanisms are not in the focus of this thesis, we do not go into the details.

Infrastructure classes

So far, we have defined the database-centric chain-of-custody as a holistic approach resulting in a thread model that allows us to define the points in the overall Digi-Dak infrastructure that needs functionality to capture and validate provenance data. These points represent the infrastructure classes from that we select our case studies. The three classes are depicted in Figure 6.5.

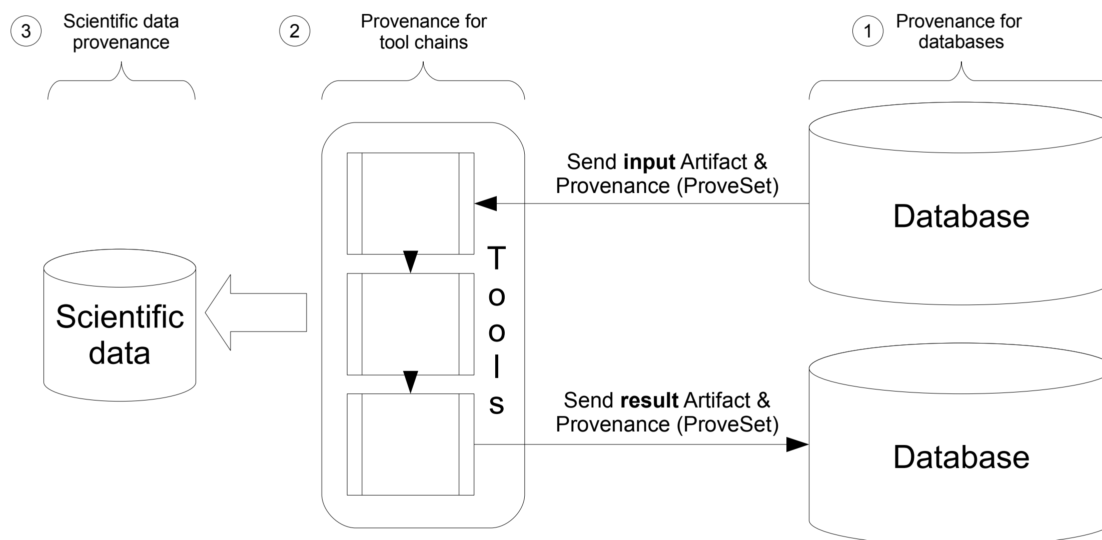


Figure 6.5: Infrastructure classes

Provenance for databases. This class ensures that every artifact send to or received from a database by any other part of the infrastructure contains a *valid* ProveSet. To this, we have to verify the existence of such a ProveSet on the database side itself. This covers also initial and final artifacts as special cases of the procedure to create a new intermediate artifact.

Provenance for tool chains. This class requires the same functionality as the first one, but is integrated on tool side.

Provenance of scientific data. The third class collects and evaluates provenance information for the scientific data that is produced within the Digi-Dak project. As most research prototypes are implemented as object-oriented programs (as the tool chains), the goal is explore whether functionality used on tool side can be used for this class as well.

To sum up, we have three different classes of systems in our infrastructure. In all of these classes, we have to integrate provenance. To this end, we select a first case study for each class to explore whether minimal-invasive provenance integration is feasible in the remainder of this thesis.

6.2 Preliminary considerations

In Section 6.1.1, we revealed that the infrastructure can be seen as software product line with a flat feature model. Adding a feature thus means deploying a variant of a tool-chain. However, before we can start to implement the first case studies, we have to take a closer look on details of the infrastructure to clearly state what we have to do in order to integrate the provenance concern for each case study.

In this section, we describe in detail the situation in the infrastructure before integrating the provenance functionality. Especially the first two case studies are closely related to each other, because the first one is about integrating coarse-grained provenance functionality on tool side (object-oriented data model) while the second one integrates provenance for relational databases. Thus, both case studies have to offer the same functionality but for a different data model. Based on the information provided in this section, we state what provenance functionality is required for the first two case studies. The information given here is used to implement and evaluate the case studies in the remainder of this section.

6.2.1 What is the initial situation?

In the following, we describe how the tool chains work without integrated provenance functionality. This is the starting point from where we start to explore how to integrate tailored provenance functionality in a minimal-invasive way. We first explain how the tools work from a technical perspective. Then, we describe the content of the features representing either a tool-chain or the base feature. Finally, we describe what ithe desired situation after integrating the provenance concern is. According to our analysis, we expect that the following assumptions hold. There is:

1. A database (DB_1) having a table, named by our conventions after the *role* of the artifact (e.g., `SensorScan` for the initial artifact), storing the input artifacts. Furthermore, there is a possibly different database (DB_2) that stores the result artifact. Both have a tailored database schema where we can store the artifact using our approach from Chapter 5. Both databases currently do not have provenance functionality.

2. A tool (or tool-chain) without provenance functionality invoking a function $f()$ using input artifact(s) from DB_1 and possibly tool-specific parameters to create the result artifact. Function $f()$ delivers the same output in case we provide the same input, which can verified by artifact-specific identity functions.

3. The tool requests data via common[2] SQL an input artifact from DB_1 and inserts the resulting artifact into DB_2. Note the databases are not necessarily from the same vendor.

[2]By common SQL we refer to Select ... From ... Where ... and Insert into statements.

4. The tool has a view on the (meta) data where a user (or the program automatically) selects the desired input artifacts. For an image that may be a unique identifier, the file name, resolution etc. without the actual pixel data. Then, the user requests the actual pixel data by using the unique identifiers.

5. For all databases, we assume the respective users are created and rights to corresponding views and tables are granted.

Note, we omitted several security related details that are important from a general security point of view, but are not required in the remainder of this chapter to integrate the provenance functionality in a minimal-invasive way. This includes, certificate infrastructures or validation of provided certificates when establishing a connection between tool-chain and a database.

Feature content - Base feature

As already mentioned there are two types of features. Now, we give details on the content of the Base feature without provenance integration. In the remainder, we use this information to state what changes in case we integrate the provenance concern.

The base feature defines the initial and final artifacts of the infrastructure by adding the tailored database schemas to the respective database. To this end, the base feature contains an abstract description of the initial and final artifact. In Figure 6.6, we depict the model of the initial artifact (the sensor scan) using the same representation as for our approach to tailor database schemas. We obtained the model with the decomposition tool as shown in Section 5.5.2 by analyzing the database schema catalog.

Note, the decomposition tool stores the artifact model internally in database tables, but is able to create and import the grammar representation, which is closely related to SQL. However, for visualization purposes, we use the grammar representation to depict artifact models. On implementation level, we automatically map the abstract implementation to an object-oriented (e.g., a Java source file) and an SQL script (of a certain SQL-dialect), where the artifact model ensures consistency between tool and database.

Feature content - Tool chain feature

In contrast to the base feature, tool-chain features contain their result artifacts and the respective tool-chain program. Their input artifacts are already introduced by a different tool-chain feature, which is indicated by a requires dependency in the feature model or directly by the base feature. We depict the feature content as well as the difference of both feature types in Figure 6.7

6.2.2 What do we want? - Required functionality

So far we described, how the infrastructure works without provenance integration and how the features are defined accordingly. Now, we define the changes that have to be applied to integrate provenance support:

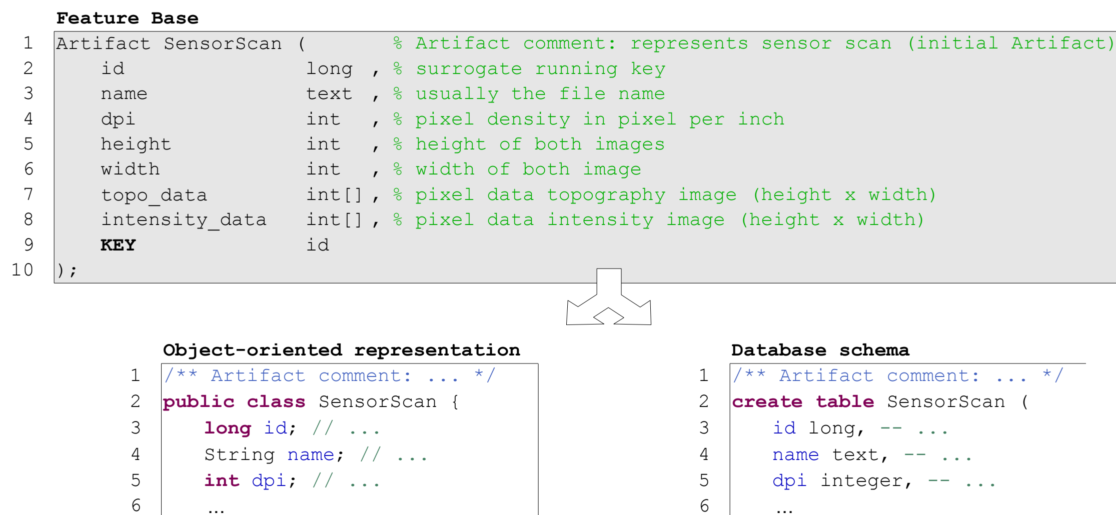


Figure 6.6: Artifact model without provenance

1. We have to provide a possibility to store the ProveSets for every artifact of forensic interest, which actually implements the chain-of-custody. Moreover, we need to link the provenance data (the ProveSets) to the actual artifacts.

2. We need functionality on database and the same on tool side to evaluate the ProveSets. This includes determining whether there is a correct CProveSet, which also means to evaluate cryptographic signatures etc.

3. Furthermore, we need portability. As one goal of minimal invasiveness is to minimize development and maintenance support the applied solution shall be as general as possible. This means that we want to be able to exchange the database system (e.g., switch from Oracle to DB2) and minimize adaptation for new tools. This results in the requirement that we have conceptually one feature *provenance* with a feature tree indicating the variability of the provenance concern itself (cf. Figure 6.8). However, we need to avoid feature interactions Section 3.3.2

4. Ideally functional and non-functional properties of the tools and database system remain the same. This especially includes the SQL statements that are send from a tool a database, as well as APIs of the tools. This is important because it ensures that provenance integration is transparent to the tool-chain developers, which contributes the goal of non-invasive provenance integration.

5. Optionally, we want to be able to recompute results to determine their correctness to trace back errors or prove that there is none.

In the remainder, we discuss how these requirements effect concept and implementation of the provenance feature tree and analyze what trade-offs have to be taken into account.

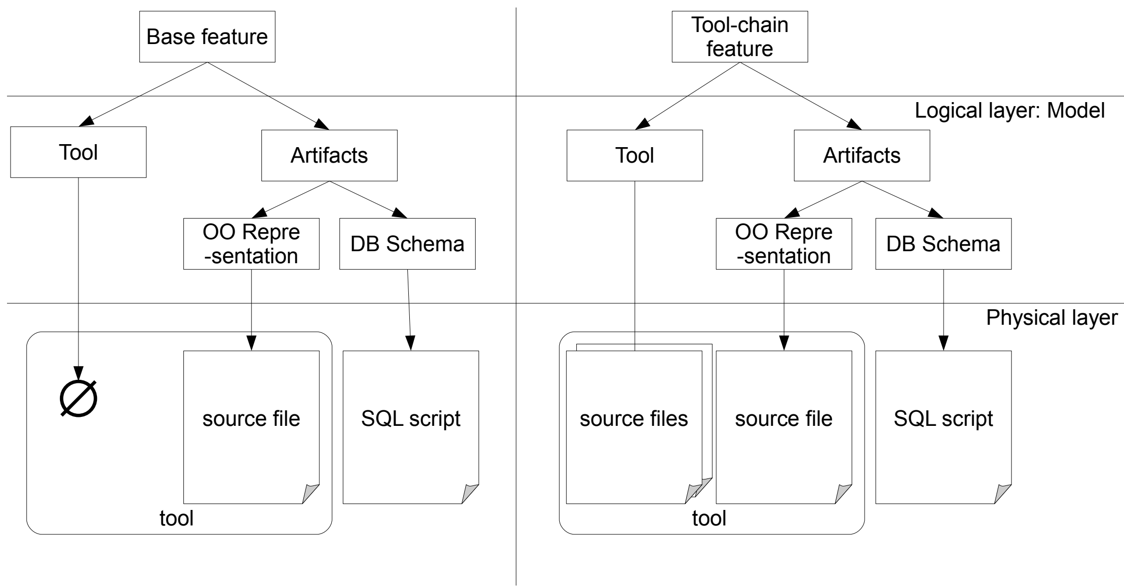


Figure 6.7: Feature content

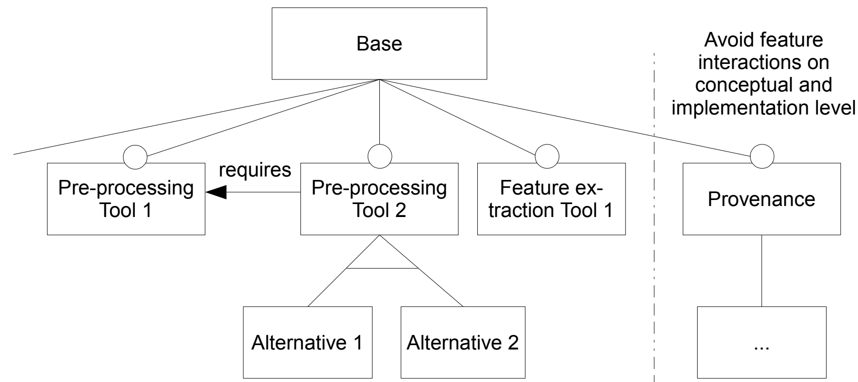


Figure 6.8: Independence of the provenance feature

6.3 The provenance feature tree

The provenance feature tree contains the provenance integration on concept and implementation level. In the following, we describe what variability is required (i.e., the single features) and how does this affect the other features conceptually and from implementation point of view. In Figure 6.9, we illustrate an excerpt of the feature tree that we use to structure our descriptions in the following.

6.3.1 The provenance feature

The provenance feature itself is the root of the provenance feature tree and offers basic provenance functionality. In fact, it allows the databases in our database-centric chain-of-custody to store the ProveSets as well as it provides a class representing a ProveSet for the tools implemented in object-oriented paradigm. In Equation 6.1, we define the

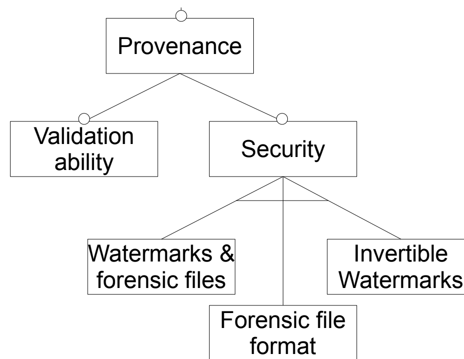


Figure 6.9: Excerpt of the provenance feature tree

ProveSet. Technically, this feature implements the parts of the ProveSet required to build the provenance graph and allows granularity refinement. Consequently, it does *not* implement security, which is an optional child feature.

Relation ProveSet

id	long	autoincrement
process	long	FK Process.id
parentID	long	nullable

Relation Children

ProveSetParent	long	FK ProveSet.id
ProveSetChild	long	FK ProveSet.id

Relation Input

ProveSetResult	long	FK ProveSet.id
ProveSetInput	long	FK ProveSet.id

Relation Process

id	long	
name	text	autoincrement
invocationID	long	

Figure 6.10: The ProveSet relations

To store data that represent a ProveSet in a database, we need to translate the ProveSet definition from Equation 6.1 to the relational data model. As the definition of the ProveSet does not change, we do not have an explicit model of the ProveSet as for other artifacts. We use pre-defined SQL scripts (for different SQL-dialects) to create the ProveSet relations. The result is depicted in Figure 6.10. However, the most important questions remain:

1. How to link the ProveSet to the actual artifacts?
2. How to capture (and evaluate) the provenance in the ProveSet in a minimal-invasive way?

We now answer the first question while the second question is case-study specific and thus, is discussed in the case-study sections in the remainder of this thesis.

6.3.2 Initial linking the ProveSet to the artifact

In the Digi-Dak infrastructure, we use object-relational as well as traditional relational databases. Moreover, on tool-side, we have classic object-oriented programs. To ensure

the chain-of-custody, we have to capture and evaluate the ProveSets for all of these data models. Consequently, we need to develop a solution, minimizing feature interactions, how to link the ProveSets to the actual data (artifacts) for every data model. Note, ensuring that this link is not tampered is the primary objective of the security feature, not of the provenance feature itself.

Object-relational and object-oriented data model. On database-side, we favor object-relational database, because here we can use inheritance. In fact, the basic idea is that each artifact inherits the ProveSet relation. The primary benefit is that we avoid feature interactions on concept level between the provenance feature and the tool features. The desired effect is that this modification is transparent. For any SQL query, there is no *visible* schema change. Consequently, we consider this solution as *minimal invasive*. In Figure 6.11, we explain the actual difference selecting the provenance feature in detail. To this end, we use the already introduced example of the SensorScan. Note, on concept level the model of the artifacts remain *unchanged*. The only difference is that the generated implementation (the SQL script) is changed. Each artifact now inherits the ProveSet relation. For the object-oriented data model, we rely on the same mechanism as shown in Figure 6.11.

Relational model. In this data model, we cannot directly use inheritance. However, we can emulate it to achieve similar effects. Basically, we have two options. First, we can add a foreign key to every artifact relation referencing the primary key of the ProveSet relation. This changes the schema of the table, which may be problematic for instance if SQL-queries use the *-Operator in the select clause. An improvement is to re-name the table and create a view that does not contain the additional foreign key. Anyway, on concept level there are no interactions between the provenance feature and the tool features. However, generating the implementation either changes the schema or requires generation of additional views. The second option is using an additional join table. This table has three attributes: (1) The primary key of the ProveSet relation tuple, (2) the (surrogate) key of the artifact relation and (3) a discriminator attribute. We require the discriminator attribute representing the role of the artifact (which also is the table name by convention), as we cannot presume that all artifacts contain globally unique identifiers. To avoid the discriminator, we experimented with using RowIDs, which are unique and moreover allow fast joining. However, we recognized that under certain (hardly predictable) circumstances several database systems re-organize tables resulting in new RowIDs making the link between ProveSet and artifact invalid.

To summarize, we favor inheritance for linking artifacts to their ProveSets. Thus, here we do not require novel implementation techniques as classic inheritance works fine. However, not in every case, we can or want to use object-relational databases but apply classic relational databases. To this end, we create Oracle syntax for relational databases

and PostgreSQL syntax for object-relational databases. This is no limitations because many database systems can parse several SQL-dialects.

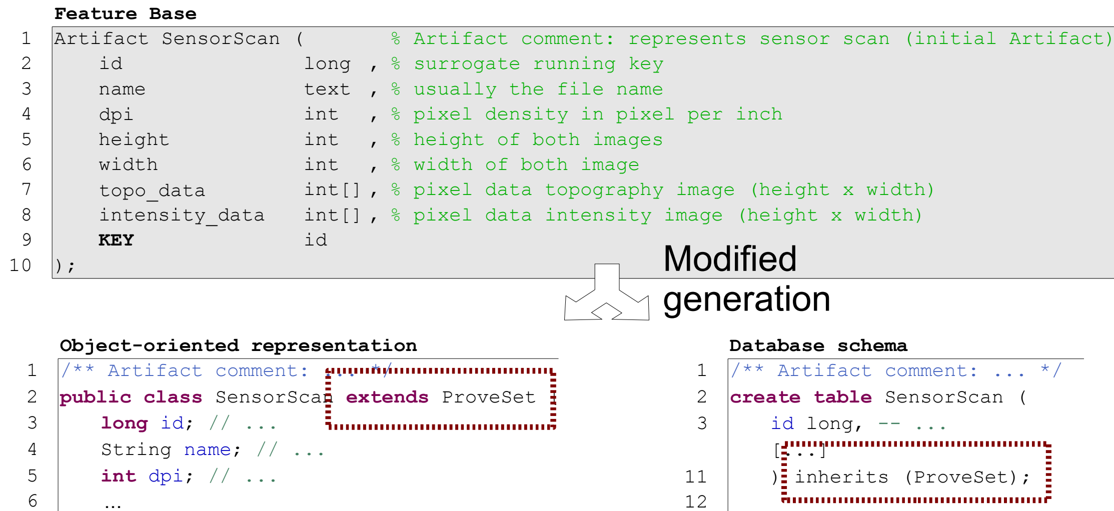


Figure 6.11: Interactions caused by provenance feature on concept and implementation level

6.3.3 The Security feature

The Security feature and associated child features (cf. Figure 6.12) address Security aspects and thus, shall improve trust into the ProveSets. Basically, we provide three major possibilities to enhance security (cf. Figure 6.12): Cryptographic hashes, digital signatures, and advanced security formats. We now explain the semantics of the first two possibilities and discuss the third one separately.

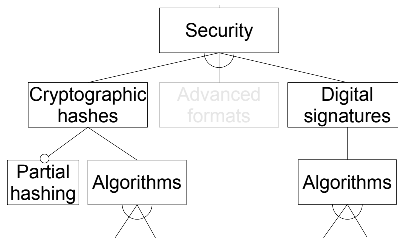


Figure 6.12: Variability for the security feature

Cryptographic hashes. Cryptographic hashes can be used to evaluate the integrity of a data item. For images and any other files, we compute hashes on the file representation (i.e., meta data and color values), while for different data types we use the string representation. Note, hashes do not ensure authenticity. However, especially during the first project years and as prove of concept, we just needed

integrity, optionally computed only on a part of an artifact (e.g., query result or image). Thus, we added an optional feature Partial hashing using either the first ten tuples of a query string representation, the first ten lines of a file, or the first 100 pixels, to speed up the hashing computation. Moreover, we support different hashing algorithms, which usually is just a parameter for cryptographic frameworks that we use to compute the hash sums.

Digital signatures. In contrast, to cryptographic hashes, digital signatures can be used to ensure integrity and authenticity of a data item with a certain residential risk. Thus, this feature is more important towards the final phase of the Digi-Dak project and for later productive usage. Similar to the hashing feature, we support different algorithms as the required security may change over time. Moreover, most cryptographic frameworks support several signature algorithms similar to the hashing functionality. Generally, we assume that all artifacts of forensic value are signed using the same algorithm (or are not signed at all).

Advanced formats. The ProveSets is currently linked to artifacts using foreign keys, which can be seen as default approach in databases. However, in the threat model, we defined that one thread are data modifications in the database itself, such as re-assigning a foreign key to a different artifact. To this end, we provide two additional features that tightly link the ProveSets to their artifact: (1) Invertible watermarks and (2) forensic file formats. To describe the semantics of these features, we have to give some additional background. Hence, we introduce them separately.

Independent of whether we apply signatures or hashes, the modification how to store artifacts and ProveSets work the same way. The algorithm features add an attribute to the ProveSet relation. Note, how the signature and hash validation is integrated into the procedure to create a new intermediate artifact is discussed in the case-study sections.

6.3.4 A short intro to (invertible) watermarks

One of the advanced techniques that allow to directly embed ProveSets into the artifacts are digital watermarks. Watermarks are a complex topic. However, we limit our brief overview on this topic to information relevant for this thesis. This is valid, because we do not want to conduct research on watermarks itself, but use them for our purposes. For more detailed overview refer the reader to [DWN01].

Generally, watermarks are a well-known technique to embed information directly into a data item by altering the data item imperceptibly [DWN01]. For instance, for images, least-significant bit encoding is an easy example. The basic idea is that altering the least significant bits of a pixel does not imply a visible change to the image. This way, every pixel in a gray-scale image carries one bit of additional information. However, using this approach alters the image irretrievably, which imposes severe problem when using signatures and hashes as they cannot be verified anymore.

There are several watermarking techniques proposed. Moreover, different watermarking techniques can be combined to create a watermarking scheme. Depending on the applied techniques, the resulting watermark has different properties, such as fragility, invisibility, or robustness [DWN01]. For our purposes it is important that the watermark is fragile (i.e., can be used to detect any modification of the artifact and ProveSet data) and invertible to be able un-embed the watermark. Thus, we focus on invertible watermarking techniques.

Invertible techniques

Our categorization of the first two techniques is based on a survey of invertible watermarking techniques [FLTC06]. The slack technique uses unused space in storage formats to embed the watermark.

Lossless compression. The basic idea of this technique is to apply a lossless compression on (parts) of the image and use the gained space to embed the watermark. To un-embed, the watermark and restore the original image the image data has to be un-compressed. The detail how the compression is applied depends on the used algorithms, such as [CSTS05]. For us, it is important that most compression-based techniques are fragile and thus can be used to ensure integrity.

Difference expansion. The basic idea of difference expansion in well-known approaches, such as [CC07, Tia03], is embedding one bit of a watermark into a pixel pair. Therefore, the distance of two pixels (i.e., subtraction of color values) is spread. For instance, in [Tia03], the authors double the distance and embed a bit of the watermark. Due to lossy integer divisions the original pixel values and the bit can be restored. The basic problem of these approaches is overflows. It is possible that spreading the distance of a pixel pair leads to color values that do not exist (e.g., are bigger than 255 for 8 bit gray values). To avoid such overflows, the watermark is extended with a *location map* that indicates which pixels do not contain an embeded bit of the watermark. Anyway, overflows lead to an unpredictable capacity (number of bits that can be embedded). Nevertheless, for our purposes, they seem to promising, because we have large images and short messages.

Slacks. In many storage formats or databases the term *slacks* names reserved but unused storage space [SML07, GSKS13]. For example, in an image format using 16 bit data types, only 10 may be used to encode the color values. Thus, there are six bytes that are allocated but not used, which can be used for embedding data, such as the ProveSets. In contrast, to the first techniques the coupling of ProveSet and artifact data is less tight, as the ProveSet can be removed by using simple bit masks. However, the embedding capacity is easily predictable.

To sum up, all techniques have benefits and drawbacks. Thus, we offer several solutions and allow the engineers to choose the adequate solution.

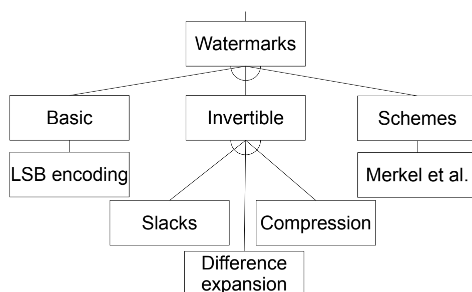


Figure 6.13: Watermarking options

6.3.5 Feature content of the Watermarking feature

In the following, we explain the selectable options for using invertible watermarking techniques as shown in Figure 6.13, which are part of the provenance feature tree from Figure 6.9. Note, although it is possible to combine different watermarking techniques nearly arbitrarily to form a watermarking schemes, we limit ourselves to four techniques and one watermarking scheme as prove-of-concept. However, additional watermarking schemes can be added quite intuitively as creating a new watermarking scheme is concatenation of applying different watermarking techniques. The selection of offered features is based on available implementations (e.g., watermarking scheme of Merkel et al. [MKDV11]) and expected implementation or modification effort.

Least significant bit encoding. We provide a feature with modified least significant bit encoding. The modification is that we first modify the image using a bit mask setting the least significant bit to zero. In this way, we create an additional slack, which can be used to embed one bit. Any hash sum or signature is computed preprocessed image (with least significant bit set to zero). As a result, the watermark can be removed and the image be restored by setting all least significant bits to zero. Therefore, we permanently modify each image (taking sensor acquisition noise into account) perceptually and a potential attacker with knowledge of the algorithm may remove the ProveSet. Nevertheless, it is easy to use approach, especially in research phase of the project that can be seen as ground truth for more advanced approaches and is thus, offered. Note, a similar approach that uses sensor noise to embed watermarks is presented in [CSV10], however their modifications of the data is less perceptual as they focus on embedding capacity and cannot remove the watermark.

Difference expansion. We use the embedding scheme of Coltuc and Chassery [CC07], because their evaluation has shown that their technique allows fast embedding. This supports our goal of minimal invasiveness regarding non-functional properties, such as response time, as well as it provides good embedding capacity. Finally, this technique is used to create the privacy-preserving watermarking scheme of Merkel et al. [MKDV11].

Compression. Our compression-based technique is a zip-function known from compressing files. The resulting watermark creates a visible watermark that appears as random noise in areas that are compressed. We selected this technique as it used to form, together with the difference expansion approach of [CSTS05], the offered privacy-preserving watermarking scheme.

Privacy-preserving watermarking scheme. We included one watermarking scheme, which was explicitly designed for the Digi-Dak project, as example to show how to combine watermarking techniques to a watermarking scheme.[3] The basic idea of this scheme is to protect privacy of (potential) fingerprints. Therefore, first bounding boxes for all regions potentially containing a fingerprint are computed. Second, image data in the bounding boxes are compressed and *encrypted*. The gained space is used to embed a watermark message, such as a signature, or filled with arbitrary noise. As a result, the image contains regions appearing as arbitrary noise where potentially a finger print is located. Finally, the location and expansion of the bounding boxes are embedded using a non-visible embedding technique [CC07]. To un-embed the watermark, first the bounding boxes have to be determined, the data in the bounding boxes needs to be decrypted, un-compressed and re-inserted into the image.

From a data modeling perspective, there is no overhead for *storing* the watermarks. For instance, for the initial artifact (cf. Figure 6.6), we can embed the watermarks into `topo_data` attribute. Thus, the storage location is already there. However, for the case studies the libraries on client and database side have to know where the watermark is located in. Again for the initial artifact, there are two attributes that may contain the watermark. To avoid confusions, we have to provide additional information, which attribute contains the watermark. To this end, the Watermark feature has a table that contains the mapping (cf. Figure 6.14). Unfortunately, the mapping is artifact specific. Thus, if we want to avoid sticking to naming convention, which would be an invasive change, we have to have a derivative for every artifact that has a watermark, to define which attribute contains the mapping. To avoid explicit derivatives, we added optional information the `-WM` flag, which can be added to the artifact comment and is understood by the parsers generating the implementation of the SQL script containing the schema implementation. Thus, in case the artifact has no watermark there is no `-WM` flag.

6.3.6 A short intro to forensic file formats

The basic idea of forensic file formats is having an all-in-one solution. The result is a *container* in that all data (including signatures) and transformation steps are stored, for instance using an XML structure. To this end, they correspond to our CProveSets, but contain also the artifact data of *every* (priorly computed) artifact, usually in an encrypted way. As a result, the forensic file format grows with every transformation

[3] A watermarking technique itself also is a watermarking scheme. However, combining different techniques to a complex scheme offers interesting opportunities.

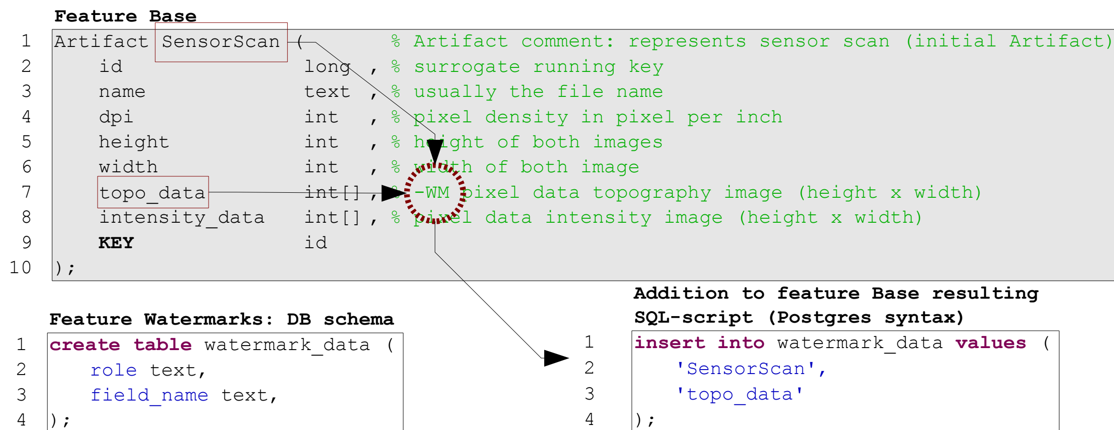


Figure 6.14: Effects adding the Watermark feature to database schema generation

step creating a new intermediate artifact. Especially for our high-resolution fingerprint images, this easily creates large container files. It is problematic, because per definition the whole container is shipped to any requesting tool, possibly requiring a large amount of time. Nevertheless, they can be seen as ultimate solution to ensure integrity and authenticity as well as providing all provenance data. However, during research, for rapid prototyping and evaluation of novel preprocessing always using such a container format is simply too much overhead. Consequently, we offer all the previously introduced alternatives and do not only rely on forensic file formats. For demonstration purposes, we limit ourselves to two formats: the first one specifically, designed for Digi-Dak, the Brandenburg Container Format [KVL11] and the second one, the more widely-known, Advanced Forensic Format Version 4 (AFF4) [CGS09]. Both formats come with complex

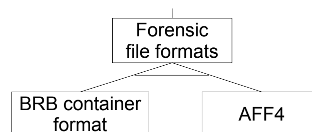


Figure 6.15: Forensic file formats feature

APIs that allow to validate a container, read parts of the containers, and extend it (if proper keys for respective signatures are provided). Thus, we do not describe their architecture in detail, as we rely on existing functionality. From the data modeling point of view, we can add an attribute containing the binary data[4] of the container file, as we do not require artifact-specific treatment. This attribute contains the *current* version of container file. However, we have to keep in mind the container file shall be send to the tool-chains instead of the pure pixel data.

[4]Usually this data tape is named BLOB such as in PostgreSQL: http://www.postgresql.org/docs/9.1/static/datatype-binary.html

6.3.7 Re-computation feature

So far, we concentrated on how to store the provenance data. To this end, we did not consider what is required on concept and implementation level to re-compute artifacts. Therefore, we have to be able to determine whether identity of the old and the newly computed artifact holds. Re-compute an intermediate or final artifact means to re-invoke the same process with the same input artifacts. All required information is stated in the ProveSet and the identity of the input and output artifacts are determined using (semantic) identity functions. In the remainder, we discuss how validation works. Based on that explanation, we state what is required on database and tool side.

Artifact validation procedure

The validation procedure is in fact a modification of the procedure to create an intermediate artifact (cf. Figure 6.4). To this end, we introduce a totally new functionality for that the original tool-chain was not designed for. However, the applied modifications to perform Step 2 are a regulative change of the procedure to create a new intermediate artifact. This is the least invasive possible modification. For comprehensiveness, we depict the procedure in the appendix Figure A.1. The actual differences are:

1. The tool starts a validation request for a previously produced *result* (A_{k+1}) of this tool-chain using the unique ID of A_{k+1}.
2. The database storing A_{k+1} determines the input artifact[5] A_k and sends A_k to the tool-chain in the same way as when creating a new intermediate artifact.
3. Additionally, the ProveSet of A_{k+1} is send in case there are tool-chain specific parameters, which have to be provided to ensure functional dependency.
4. In case there are tool-chain specific parameters, the transformation function has to be invoked using these parameters. Note, the actual computation remains the same.
5. The tool-chain sends the resulting artifact A^n_{k+1} to the database, which invokes `identic(`A^n_{k+1}`,`A_{k+1}`)` and send the result back to the tool-chain approving or declining the validation request.

Local artifact validation procedure

In later productive use, we expect that there is a database administrator responsible for the database and a forensic expert administrating the tool-chain. In case the validation is declined (i.e., an error is detected), it is not clear where the error occurred. Therefore,

[5]Note, for simplicity of explanations, we assume that there is only one input artifact of forensic relevance (e.g., and image), as we encountered it frequently in Digi-Dak. However, the explanations are easily adapted to multiple input artifacts as the resulting query remains the same and subsequent steps have to be repeated for all input artifacts.

we offer an optional feature *Local validation* that shall allow to verify results locally, in the responsibility of one person. The idea is to give the tool a local provenance store that allows to determine identity of the input and result artifact on client side. Always having these provenance stores when developing light weighted tools may reduce development efficiency. Thus, we offer this functionality as an optional feature. The primary differences to the normal validation are:

1. The tool-chain needs a local provenance store containing either IDs of the input artifacts if only the provenance feature is selected. In case the Hashing or Signature feature is selected also the hash (or signature) values values are stored.
2. The tool-chain needs (potentially artifact-specific) identity functions.
3. A_{k+1} itself is send to the tool-chain instead of only sending the respective ProveSet. This is required to invoke the identity functions locally.
4. The tool can decline the validation request locally in case one of the identities does not hold.

Effect on modeling level

So far, we were able to avoid interactions of the provenance feature tree and the tool-chain features, which have to be resolved using derivative features. However, as artifact-specific identity functions as well as tool-specific parameters, cannot be defined globally without highly restrictive conventions, making prototyping practically impossible, we see no better option than using a derivative for each tool-chain feature. However, not every artifact requires a special identity function as the default semantic identity function (comparing all fields of a database table) works well in many cases. Nevertheless, practically every tool-chain has tool-specific parameters. Anyway, the derivatives the following content on concept level as depicted in Figure 6.16:

Tool-specific parameters. Considering our provenance framework from Section 2.3.2 collecting tool-specific parameters means we switch from the Workflow layer, showing causal dependencies, to the Existence layer, allowing recomputation of the resulting artifact. To this end, we do not add tool-specific parameters to the ProveSet relation, but create an extra relation referencing the artifact relation. This way, we can easily query the data for the two provenance layers. By convention, we name the resulting table after the artifact role and add `_parameters`. To this end, the resulting table for the `EqualizedImage` artifact is `EqualizedImage_parameters`. In case we use watermarks, we do not have to create another relation, because we store the parameters in the watermarks. Similarly, this works for forensic file formats. However, embedding the additional provenance data is required and considered in the case-study sections.

Artifact-specific identity functions. In contrast to any previous feature, we have to consider *user defined functions* on database side, which require an extension of our approach to tailor database schema. To this end, every artifact may contain specific identity functions. Upon variant creation the default identity functions are superimposed with the artifact model (cf. Figure 6.16). Thus, the artifact-specific definitions prevail. Note, it is conceptually and technically possible to have multiple definitions of the identity functions. Then, the last one prevails. However, as we did not encounter a useful application for multiple definitions, we forbid this case. An error occurs if a user tries to define an identity function multiple times.

Derivative Feature Equalization Tool/Provenance

```
Parameter EqualizedImage (
    blockSize    int   ,
);

Identity function semantic EqualizedImage scan_1 scan_2 (
    scan_1.dpi = scan_2.dpi
    and scan_1.height = scan_2.height
    and scan_1.width = scan_2.dpi.width
    % int[] is complex artifact calls identity function for int[]
    % identity(int[] i_1, int[] i_2) compares length and all values in array
    and scan_1.topo_data = scan_2.topo_data
    and scan_1.intensity_data =  = scan_2.intensity
);
```

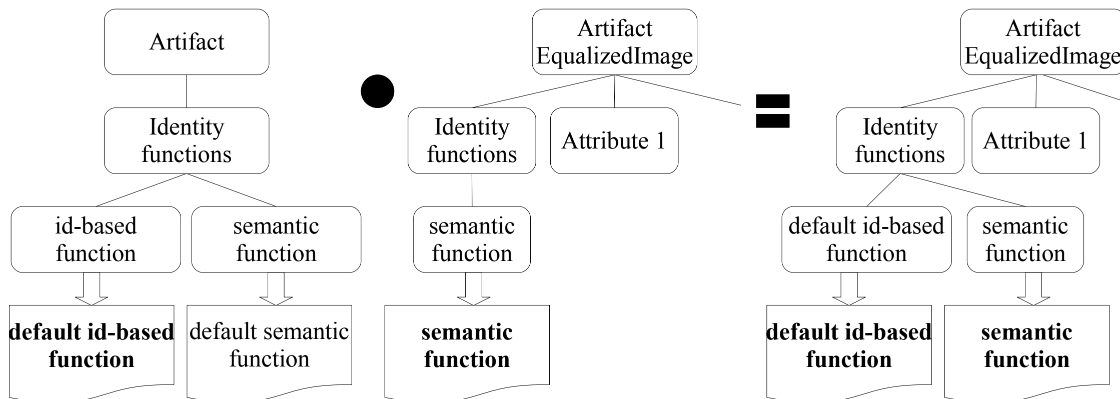


Figure 6.16: Provenance derivative

6.4 Summary

In this chapter, we contribute the concept of the database-centric chain-of-custody [SSK11] as preparation in order to conduct the first case studies in the next chapter. Particularly, the chain-of-custody, therefore, is a concept that defines the level of granularity regarding the artifacts for that we have to capture reliable provenance information. Moreover, we

amplify this concept to develop a threat model that results in the definition of classes of systems within our infrastructure requiring similar provenance functionality. For each of these classes, we select one case study in the remainder.

The second contribution is the *design* of a versatilely applicable provenance SPL in order to extract the desired provenance data. Based on this design, in the following chapter, we first implement the provenance SPL (the Soma), secondly integrate the desired provenance functionality (the Dendrites) into the case studies itself, and finally report on our observations.

7. First exploratory case studies

In our research agenda in Section 4.3.4, we defined the goal of this chapter as: *Explore feasibility and beneficial as well as problematic properties of programs when integrating provenance by conducting first exploratory case studies.* In particular, we investigate the following questions and their (possible) mutual interaction:

1. What is the impact of applying different implementation techniques regarding the goal of minimal-invasive integration of the provenance concern as defined in Section 4.2.1?
2. What characteristics, especially in the decomposition of existing solutions, support or complicate the integration of the provenance concern in a minimal-invasive way?
3. Are there mutual interactions between applied implementation technique to integrate the provenance concern and characteristics of the existing solution?

In this chapter, we explore whether it is feasible to integrate tailored provenance functionality into existing systems. Mainly, we are interested in the major challenges that arise when integrating the provenance capturing capability. These insights shall help to implement and evaluate more advanced case studies in the next chapter. And consequently, in the end, allow a more reliable answer to the question whether provenance integration into existing systems is practically feasible. To this end, we rely on the infrastructure classes from the previous chapter. For each of the three groups, we select a representative case study to integrate the provenance capturing capability and finally summarize the insights gained.

7.1 Coarse-grained provenance integration on tool side

In the prior chapter, we designed the provenance SPL that we want to integrate into our case studies. Moreover, we succeeded in minimizing feature interactions. In the following, we explore the remaining question how to incorporate provenance capturing and evaluation of provenance data in the single parts of the infrastructures. To this end, we start with small case studies allowing for more advanced and complex ones in the remainder. The question we work on is: *How to capture (and evaluate) the provenance in the ProveSet in a minimal-invasive way?*

7.1.1 Objectives

Besides implementing the features from Section 6.3 with techniques introduced in Section 3.2, there are several objectives that have to be considered for this case study. In the following, we point out these objectives and explain why they are of special relevance for this case study to ensure minimal invasiveness:

Minimizing implementation and maintenance effort. This contains the view of the developer trying to integrate the provenance functionality.

Minimizing functional impact. This objective contains the view from tool side according to the classification given in Section 4.2.1. It considers two points: (1) Intended invasiveness of the change itself and (2) additional invasiveness implied by limitations of the implementation technique.

Minimizing non-functional impact. In this objective, we try to minimize impact on non-functional properties. The evaluation is based on the unmodified system as optimum and the intuitive implementation technique as ground truth. This way, we are first interested into getting an impression of the influences without integrated provenance. Moreover, we are able to explore potential differences using different implementation techniques.

In addition to the aforementioned objectives, we are also interested into properties of the programs that either support or complicate the integration of the provenance functionality. In the considered case studies, we have effectively no support from the developers that would simplify integration of the new concern. However, allowing easy integration with no support of the developer may turn out to be naive. Thus, we are interested into properties that can easily be realized for instance by known refactoring, which should be applied anyway. Finally, we explore where to integrate the provenance concern considering different opportunities and discussing benefits and drawbacks.

7.1.2 Implementation concept

In the following, we explain the implementation concept as required to comprehend details of the actual implementation and the subsequent evaluation. To this end, we first explain why a first general approach fails. Then, we use the insights gained to develop the solution concept.

The neuron analogy revisited

In Section 4.3.2, we defined the overall architecture using the neuron analogy. Now, we refine this analogy for this case study in Figure 7.1 and recapitulate the architecture parts used in this case study. In addition, we give additional information on the semantics of the single parts.

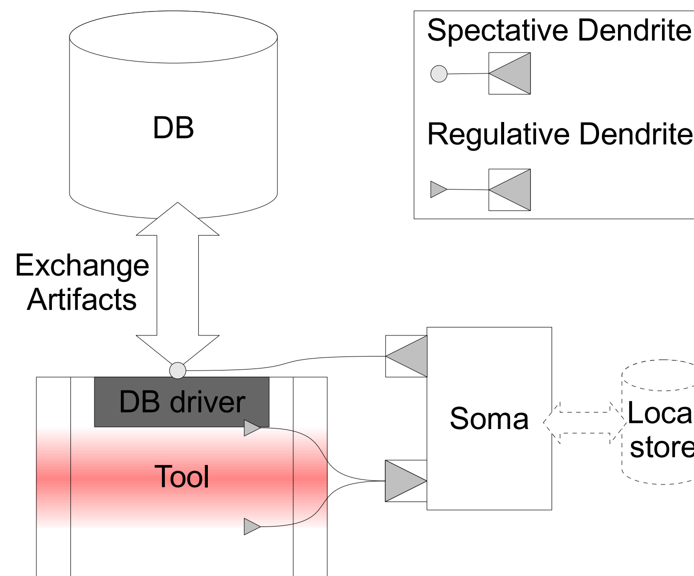


Figure 7.1: Refinement of the neuron analogy for the first case study

Soma implementation. The Soma implementation encodes the actual provenance functionality. However, as described in Section 4.3.2, the Soma itself does not contain any connection to the original tool. Recapitulate that these connections are named Dendrites.

Dendrite implementation. In contrast to the Soma, the Dendrites are the invasive parts that either intercept the artifacts or propagate regulative changes from the Soma to the tool-chain (e.g., abort in case signature validation fails).

Optional provenance store. In case the optional Validation feature with Local validation is selected, the tool-chain has an additional local provenance store containing signatures, keys, and hashes of priorly consumed and produced artifacts.

On the way to the actual solution - The road not taken

Based on the Figure 7.1 and similar to the ideas of communication-centric architectures for services [SPG08, GMM05] or buses [CBSA11], our first idea was to link the provenance functionality to database drivers. These drivers are essential for any communication with a database. The benefits are that drivers, such as JDBC[1], are implemented for all major database vendors and are the gateway from and to the database. Thus, we do not have go into the details of the tools themselves. We linked our provenance capturing and evaluation to the methods of the `JDBC Statement` interface that are responsible for query execution and the `ResultSet` class representing an abstract query result. However, the tool chains query the database not only to receive or insert artifacts, but also for getting meta data etc. To this end, we cannot treat every SQL query the same way, but have to identify the queries and corresponding results representing either an intermediate result request or an insert of a new intermediate artifact Figure 6.4 at run time, by *parsing* the SQL strings. This is possible, but not practical and does not support non-invasiveness regarding non-functional properties. To sum up, it would have been desirable to modify only the database drivers, which is very well possible for instance to log, which queries have been sent to a database. However, for our purposes the level of abstraction at the database driver is too much. Consequently, we have the need for a different solution that we introduce in the remainder.

Defining the extension points via source-code generation

Our initial solution attempt showed that we need more semantics than simple SQL strings. To this end, the next logical step is trying to link provenance capturing and evaluation to the *construction* of the object representing the artifacts. This is a *novel concept* to exploit the structure of object-oriented programs for provenance functionality that we first published in [SSS12b]. So far, provenance capturing focused on abstract descriptions, such data models (e.g., [CCT09]), but ignored how these descriptions are implemented in object-oriented languages. Similar approaches, published simultaneously (at the same venue) [TAG12, BCK12], also acknowledge this gap and try to exploit the program structure. However, in [TAG12] the authors use modified compilers resulting in an all or nothing property. This does not allow for tailored provenance capturing and they cannot handle complex objects (artifacts). In a similar way, the approach in [BCK12] contains a formal model, which does not consider the tailoring requirement. A later approach is also defined on object-oriented program structures [CSRH13], but it is not designed for minimizing the integration effort.

We already have a model of the artifacts that can be easily mapped to source code. Thus, the basic idea is to create the source of a class file that represents the artifact. However, we discovered that the classes representing the artifacts in the tool chains contain additional methods or fields. Consequently, we cannot generate these source files. However, we can generate a class that can be easily mapped to the desired one. To this end, we generate a helper class named by convention after the role of the corresponding

[1] Java Database Connectivity specification: http://java.cnam.fr/iagl/biblio/spec/jdbc-3_0-fr-spec.pdf

artifact attached with `Artifact` (e.g., `SensorScanArtifact` for the `SensorScan`). This can be seen as a light weighted object-relational mapping without using respective tools. We do not use such tools especially, because we consider integrating these tools as not minimal invasive, especially regarding the footprint of the tool chains.

Refactoring effort - A practical solution

To create the actual classes representing the artifact (e.g., `SensorScan`) from a helper class (e.g., `SensorScanArtifact`), the actual classes need two methods that fetch the data from the database and send new results back to the database. Per class, this requires three lines of code, where we can generate the code fragments (complete methods). However, as they have to be present also in case we have no provenance integration, these methods have to be integrated manually or result in refactoring effort. Our analysis of the existing tool chains reveal that often the functionality to load the data is tangled somewhere in the source code and not explicitly capsuled in an extra method. Consequently, a refactoring is suggested anyway. Moreover, we experienced that this enhances the acceptance of the original programmers of the tool chains for the integration of our new functionality, as it gives them an impression where and how the integration takes place. Finally, in case there are small changes to the objects, such as renaming of fields, the helper classes provide some limited schema independence. To sum up, we consider our approach of the generating the source code of the helper classes as a practical solution that allows us to integrate the provenance functionality.

7.1.3 Provenance integration

We implemented the Soma functionality in form of a provenance API in an extra package `de.ovgu.provenance` that is deployed as jar-file. Note, the Soma currently contains no connection to the program, because this is handled by the Dendrites. Moreover, the variability within the API is contained in independent sub packages that are either deployed or not. In the following, we briefly sketch some implementation details that are relevant in the remainder.

Internal watermark scheme generator. As we proposed in prior work [SSM+11], we designed the provenance API in such a way that we can nearly arbitrarily combine different watermarking techniques to form a watermarking scheme. To this end, a watermarking scheme is a concatenation of techniques embedding a bit string (the message). How to interpret (and generate) the bit string, is defined in a pattern, which in fact is a schema for the bit string. Moreover, we automatically map the provenance data of the artifact (e.g., artifact id, signatures etc.) to such a pattern, which then can be embedded or read.

Applied cryptographic framework. For computing the signatures or hashes, we use `java.security` package. When deploying our provenance API, the API comes with default keys and certificates to test functionality. However, for intended productive application the keys and certificates have to be exchanged requiring a certificate infrastructure. This is not part of the provenance SPL.

Forensic file formats. The forensic file formats are deployed with a fully functional API to query and extend these files. The challenge is that these APIs do not exist for all programming languages. For instance, the Brandenburg Container Format [KVL11] API is only available in C Sharp, while a lot of our prototypes are programmed in Java. To integrate these container files, we have to call these libraries via command line from within the tool-chain and store the desired result in a temporary file. We are aware that this imposes new attack vectors and imposes performance drawbacks. However, it is the only way to integrate the container formats instead of providing an own API.

Dendrite implementation using different implementation techniques

Implementing the Dendrites means implementing the previously defined functionality with all introduced implementation techniques. This results into five different implementations of the same functionality. The implementations for the two variants of the intuitive techniques are nearly identical and considerably similar to the CIDE implementation. However, the implementation feature and aspect-oriented implementation differ significantly and are in fact new implementations.

Implementation procedure. To observe differences, we started the implementation using the intuitive techniques and recorded core challenges to compare them to the more advanced implementation techniques. In the remainder, we also use the implementation to evaluate the effect on functional and non-functional properties.

Evaluation coverage. Subsequently, we focus on a sub set of the available features selecting a representative from different groups of functionalities. For instance, we consider one hashing and signature algorithm. In addition, we limit ourselves to one forensic file format (BRB), two watermarking techniques least significant bit & the technique from Coltuc et al. [CC07] and one hashing (SHA256) as well as one signature algorithm (AES256 with SHA256) in accordance with [BSI14].

7.1.4 Observations

In the following, we discuss observations that we made during the implementation and when testing and evaluating the implementations in the context of minimal-invasive integration as defined in Section 4.2.1.

Integration and maintenance effort

From point of view of the developer, the first and most important observation is that the feature model changes from the quite complex one for the Soma functionality (cf. Section 6.3) to the one displayed in Figure 7.2. This is because the additional functionality is hidden in the Soma implementation. This eases integration and is one argument why our approach is practically feasible. All case studies are implemented

in Java 1.7 using Eclipse Kepler with FeatureIDE extension[2] except for CIDE having an own Eclipse plug-in offering roughly the same functionality. This way, we ensure comparability and soundness of our results and we can automatically generate tailored tool-chain variants.

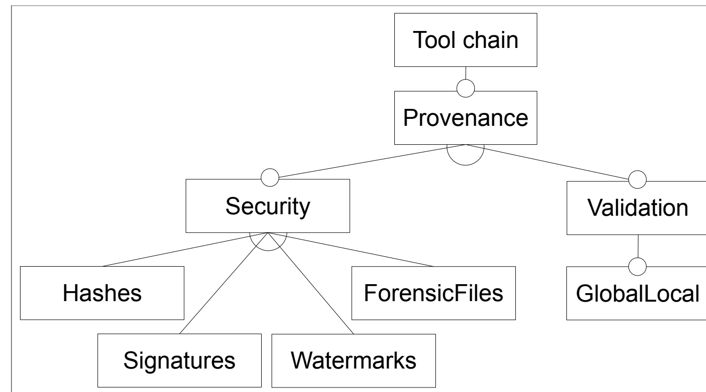


Figure 7.2: Change of the provenance feature tree

In the following, we summarize the most important properties regarding (1) integration effort, (2) impact on maintenance of particular variant (mainly measured by feature cohesion), and (3) obfuscation of the resulting source code for every implementation technique. As it is hard to measure these criteria objectively, for instance by giving total numbers, we compare them subjectively from the point of view of a developer and justify our explanations.

Intuitive implementation techniques. Recapitulate that the two intuitive implementation techniques use conditional statements to encode the intended variability. The difference between both techniques is that the dynamic approach utilizes variables, while the static one utilizes constants. The main intention for two techniques is the assumption that the compiler removes not required code in the second technique denoted by conditional expression that are evaluated as `false` and contain not required code.

Integration effort. Integrating the provenance capturing and evaluation functionality into the tools requires to extend the two methods `get()` and `send()` of each generated helper class (e.g., `SensorScanArtifact`) communicating with the database. An extension means, for instance, to copy thirty lines of additional source code into the `get()` method of each artifact used in the tool as depicted in Figure 7.3. This source code consists of nested `if` conditions to avoid code duplication. However, as we do generate these classes, we can easily generate these parts as well, but we have to encode artifact-specific behavior using reflection [GJSB05][3] to allow for such generation. In summary, we do *not* have to integrate the Dendrites manually.

[2]FeatureIDE 2.6.7 can be obtained on http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/ or via the Eclipse marketplace.

[3]http://docs.oracle.com/javase/7/docs/api/java/lang/reflect/package-summary.html

```
1  long aid = result.getLong("aid");
2  Artifact temp = getInstance(aid);
3
4  if (!(Provenance.Security.XOR == Provenance.Security.FORENSIC_FILES)){
5    database.DB.map(temp,result);//Using reflection
6
7    if(Provenance.Security.XOR == Provenance.Security.WATERMARKS){
8      WaterMarkPattern pattern = new WaterMarkPattern((Object)temp);
9      pattern.verify();
10   }else{//Provenance.Security.SIGNATRUES || Provenance.Security.HASHES
11     String verificationString = getVerificationString(temp);
12     byte[] toTest = CryptoFramework.getVerificationString(temp)
13       .getBytes(java.nio.charset.Charset.forName("UTF-8"));///Using reflection
14     CryptoFramework crypto = CryptoFramework.instance();
15
16     if(Provenance.Security.XOR == Provenance.Security.HASHES){
17       byte[] hash = result.getBytes("hash");
18       crypto.verifyHash(toTest, hash, temp);
19     }else{// Provenance.Security.SIGNATURES)
20       byte[] signature  = result.getBytes("signature");
21       crypto.verifySignature(toTest, signature, temp);
22     }
23   }
24 }else{
25   String fileName = "temp_container.zip";
26   byte[] container = result.getBytes("container");
27   Container.writeTempFile(fileName,container);
28   Container.validate(fileName);
29   temp = readContainer(fileName, aid);
30 }
```

Marked as dead code

Marked as identical expression

Figure 7.3: Code to integrate using static if-approach with activated hashing feature

Note that using reflection may introduce a performance penalty, but does not introduce problems regarding type safety as all artifacts are generated using *one* artifact model. The previously described functionality contains only the Security feature tree. In addition, we have to integrate the validation feature. In contrast, to the Security functionality the Validation has no fixed point in the program structure where to integrate the required Dendrites. Often, we can directly change the `main()` function, but the functionality encoding the program logic of the tool may be hidden in the source code. As a result, we have to look for the right point in the source of the tool and integrate the Validation functionality manually. To sum up, the main effort is writing the code generator and locating the program logic where to integrate the Validation.

Maintenance. Regarding maintenance the provenance security functionality is scattered over all artifacts, which themselves may be distributed over the whole source of the source code, and the Validation functionality is hidden somewhere in the source code of the tool. To this end, there is little feature cohesion. However, using Eclipse, the development environment automatically provides an overview of all source code locations that contain feature specific code. From our point of view, this nearly *totally* compensates the missing feature cohesion regarding maintenance.

Source obfuscation. The major drawback of both intuitive implementation techniques is the missing variant generation step. In fact, every variant contains on source code level the program code for all possible variants. This obfuscates the variant source code (cf. Figure 7.3) not only the general code base used for variant generation. Furthermore, in the static if-approach with features encoded as constants, the applied development environment additionally puzzles the developer with its syntax highlighting as shown in Figure 7.3. The reason is that it *correctly* displays all code fragments of inactive features as *dead code* and the conditional statements of activated features as *unnecessary* (in Eclipse comparing identical expressions). For our explanations, we used different (but similar) colors to mark dead code and identical expressions. However, per default the development environment does not. Consequently, the developer can hardly determine, which features are activated based on the current configuration. Finally, another drawback of this integration of the Validation feature is that we cannot remove the Dendrites from the source automatically, as we can for the helper classes. In case we want, for instance, benchmark a variant of the tool totally without provenance, there are remainders of the Validation feature in the source code of the tool. Consequently, there has to be a part of the Soma functionality as well. This is not in the sense of our goal of minimal-invasive provenance integration and hence, hardly acceptable.

Advanced preprocessor technique. As defined in Section 3.2.2, we use CIDE (in Version 2.2.0) as preprocessor technique and report our observations subsequently.

Integration effort. The integration effort is very similar to the intuitive techniques. However, one difficulty arises due to implementation details of CIDE. The preprocessor annotations are stored in a different file and not directly in the source code. Thus, we have to generate this file as well. Moreover, we had to re-engineer how the content of the mark-up file is interpreted. A different difficulty arises due the idea of using colors in general. In Figure 7.3 Line 4, it is defined that a certain part of the source code only is *not* executed in case the Forensic Files feature is selected. This information is hardly depictable with colors. For code that is only executed in case multiple features are selected the colors are mixed, for *inclusive ors* however, we have to replicate code. Finally, using this technique, we also have the issue that we need to locate where to integrate the Validation features. However, in contrast to the intuitive technique the variant code does not contain un-required code.

Maintenance. The feature cohesion is the same as for the intuitive approach. The provenance code is scattered over all artifacts and the Validation code is hidden somewhere in the program itself. In contrast to the intuitive technique, there is no global overview that can be used to jump directly to the feature code for the whole project.

Source obfuscation. From our subjective point of view, which is supported by empirical studies [FKA+12] using colors, the source-code obfuscation is less than using if-conditions, especially as we have little overlapping features.

Feature-oriented programming. To implement our case studies, FeatureIDE offers to use FeatureHouse [AKL13] or AHEAD [Bat04]. We decided to use FeatureHouse instead of AHEAD, because with FeatureHouse, we can directly use the source of the original program and copy it into the base feature implementation module. Using AHEAD, we would have to convert every source file from `.java` into a `.jak` file and the `.jak` files do not contain information regarding the implementation package which we would have to remove from the `.java` file.

Integration effort. Technically, for each artifact and feature there has to be an implementation (a refinement). However, as the refinement encodes the same functionality, we can generate these code fragments for each artifact as well, which are then used to compose a tailored variant automatically. As a result, we do not have to implement each refinement manually. In addition, we do not have to replicate code, but we had to integrate two empty *hook* methods [MLWR01] per artifact. Nevertheless, for the validation feature, the effort to locate the source code where to integrate the Validation feature remains the same.

Maintenance. From point of cohesion, the Dendrites of a feature are located in separate files belonging to that feature (including the Validation feature). We consider this as very beneficial, because it is clear where which functionality is added in case a certain feature is selected.

Source obfuscation. In fact, there is no obfuscation of the original source code contrary to any prior implementation technique. Moreover, as the Dendrite feature model mainly contains alternatives and there are little method refinements, we argue that the source code is easy to understand.

Aspect-oriented programming. For the aspect-oriented integration, we use the FeatureIDE extension for AspectJ (v. 2.6.7) in concert with AspectJ Development Tools for Eclipse (v. 2.1.3) [Kic96], because it is already part of FeatureIDE and offers full development support. This includes a feature modeling tool and allows to create particular variants containing only selected aspects, which is usually not the main focus of aspect-oriented programming.

Integration effort. Interestingly, we only need two pointcuts for the Security feature sub tree. Thus, we also have to implement only two advices per feature not per combination of artifact and feature. However, as we generate parts of the source code, this only results into less effort to implement the code generator, which is compensated by the learning effort to deal with complex syntax of AspectJ.

Nevertheless, this is an interesting observation for more advanced case studies in the next section. In contrast to feature-oriented programming, we do not have to implement hook methods nor do we have to re-factor the source code of the tools. Finally, integrating the Validation functionality is as laborious as using the other programming techniques.

Maintenance. Regarding feature cohesion, here aspect-oriented programming is very similar to feature-oriented programming. The Dendrites, encoded as pointcuts, are located in a separate package and the advice codes are located in separate files of the single features modules too. Subjectively, it is uncommon that even in a particular variant the executed source code in the advices remains physically separated and we require tool support to see where what additional code is executed, which is not as intuitive as using feature-oriented programming.

Source obfuscation. As the Dendrites are implemented in separate files there practically is no obfuscation. The already mentioned physical separation of concerns in variant code may be less intuitive to programmers used to object-oriented paradigm, but in summary we cannot observe a difference compared to feature-oriented programming.

Comparison of the implementation techniques. Generally, we observed little difference regarding the implementation techniques and report that each technique is valid and practical choice under certain circumstances. Considering our observations, we only recommend to use the intuitive technique, in case conventions or other restrictions do not allow to use FeatureIDE, mainly due to the missing variant generation step resulting in massive code obfuscation. CIDE would always be a valid choice with better tool support for this kind of case studies. Finally, here we cannot make a distinction between aspect-oriented programming and feature-oriented programming. Although, we have to use hook methods in feature-oriented programming and need to re-factor small parts of the code, we argue that the aspect-oriented language extension is far too powerful and thus, too complex for this first case study. Anyway, the observation that we can integrate the provenance functionality using aspect orientation with very little source code is highly interesting.

Effect on functional level

In the following, we report on the invasiveness of the single features that we integrated into the source of the application. In fact, we are interested whether any of the implementation techniques forced us to perform more invasive changes. We depict our observations in Table 7.1. This table contains the categorization (cf. Table 4.1) for each integrated Dendrite (block of code) and short explanation on the semantics of the respective Dendrite.

The general result is that most *additional* changes are minor ones. The only real problem that occurred are un-removable code fragments for the Validation feature

Table 7.1: Effects on functional level

Feature		Intended invasivenes		Additional invasiveness			
		Category	Description	Intuitive	CIDE	FOP	AOP
Provenance	`get()`	1 Spect.	Monitor which artifact is requested				
	`store()`	1 Spect.	Monitor which artifact is send to DB				
Security	`get()`	1 Spect.	Adds new method, but actual			hook method	reflective call
	`store()`		control-flow changes in child features			hook method	reflective call
Hashes	`get()`	1 Regul.	check & read container data		code cloning		
	`store()`	1 Spect.	computes hash				
Signatures	`get()`	1 Regul.	adds signature does not match alert		code cloning		
	`store()`	1 Spect.	computes signature				
Watermarks	`get()`	2 Regul.	unembed watermarks, adds signature does not match alert				
	`store()`	1 Regul.	embed watermark				
ForensicFiles	`get()`	2 Regul.	check container, read container data				
	`store()`	2 Regul.	append & send container to DB				
Validation	`main()`	2 Invasive	Validation request contacting DB	un-removable			
LocalValid.	`main()`	1 Regula.	Local validation error	un-removable			

using the intuitive implementation techniques. Whether, the additional reflective calls of the aspect-oriented technique (allowing to use two advices per feature) introduce a performance penalty is examined shortly. Summarily, we observed one additional potential drawback for the aspect-oriented technique.

Effect on non-functional properties

Subsequently, we examine the effects of the provenance features on the non-functional properties footprint, performance, and main-memory consumption as defined in Section 4.2.2.

Footprint. We have already determined that, except for the intuitive techniques, in a particular variant there is only required code. Thus, the footprint is of minor importance. However, a different footprint of the compiled helper classes (containing the Dendrites) delivers a first and easy to examine hint whether the compiler is able to remove unrequired (dead) code from the binaries. Indeed, if compare, for instance, the resulting `SensorScanArtifact.class` files produced with the help of the dynamic and static intuitive approach (with selected Feature Hashing), we observe a difference in the file sizes although the source code files are of equal size. Consequently, we hypothesize that the compiler removes un-required code fragments.

To verify our hypothesis, we analyzed the resulting byte code gained by invoking the `javap` command, from the Java Development Kit 1.7 on the `SensorScanArtifact` class file. Our in depth analysis reveals that using the static implementation technique indeed all dead code fragments and the conditional statements of the activated feature code are removed. In contrast, using the dynamic approach the code fragments are part of the compiled code. This firstly, justifies that it is necessary to differentiate between the two techniques and secondly, we have to answer the question whether this results into a measurable performance difference.

Performance. From the point of view of minimal-invasive integration, we have to minimize the introduced performance penalty. Thus, now we have to determine two things: First, the general performance penalty introduced by the different provenance functionalities. And second, we have to know whether we can measure a performance difference for the same variant due to the application of a different implementation technique. Of special interest is whether we can determine a difference between the two intuitive techniques and whether there is a general performance penalty using AspectJ due to language overhead. Generally, we assume that we cannot measure a significant performance difference caused by different implementation technique. Consequently, we expect that the measured performance difference per implementation technique is less than the variance of the measured response times.

We use the `get()` function of an artifact to determine the response times. Response time is defined from invoking the function until it resumes. We select the `get()` function as the `store()` functions performs very similar tasks. We measure 100 invocations of the function via requesting the same image and compute the mean values to ensure statistical soundness. All images have the same size as the one we that was used in [KS13a] and thus represents realistic workloads. All measurements are performed on an Intel Core 2 processor with 2GB main memory and the tendencies have been verified using an Intel i7 with 8GB main memory. In addition, we used Oracle Java Development Kit in version 1.7.0 and the artifacts are stored using PostgreSQL in version 8.4.

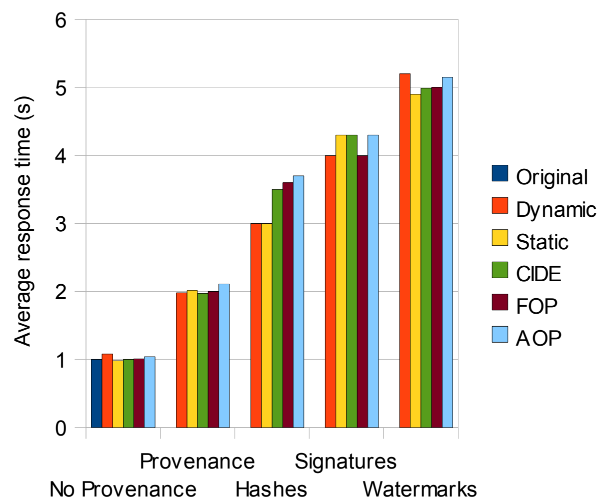


Figure 7.4: Average response time per feature and implementation technique

As expected, in Figure 7.4 we cannot observe a significant difference using different implementation techniques. However, we do observe large differences between the single features. As a result, for coarse-grained provenance, we conclude that regarding performance there is no difference between the implementation techniques. However, for future case studies, we hypothesize that we face additional performance penalties which have to be taken into consideration.

Main-memory consumption. What we are interested in, is the total amount of main memory consumed, as it may reach the capacities of a given system. Hence, we run a tool in an infinite loop, requesting always new results, and record the total amount allocated memory from the task manager. In Figure 7.5, we depict our results for each feature and technique using the equalization tool from [KS13a].

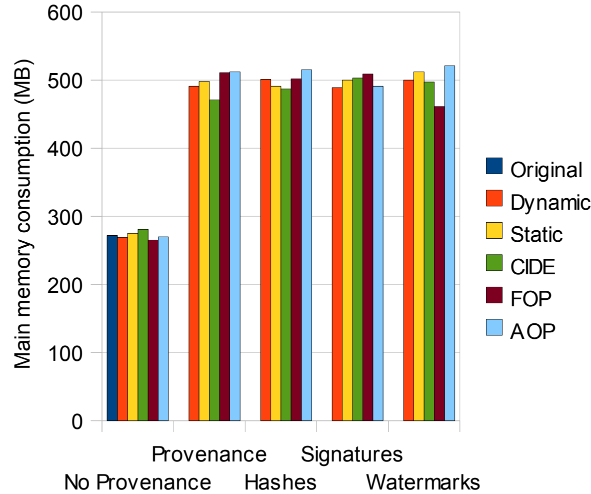


Figure 7.5: Average main-memory consumption after five minutes per feature and implementation technique

We observe that there are in fact two different amounts of main memory required. In depth analysis using profilers reveal that this results from the applied cryptographic frameworks to validate hashes and signatures. Consequently, we do not see a measurable difference between the single implementation techniques not even with the additional runtime libraries required for AspectJ.

7.1.5 Lessons learned

Conducting the first exploratory case studies reveals mainly three findings. First, the simplification of the feature model on Dendrite side eases the integration of the provenance functionality, which is important for practical feasibility of our approach. Secondly, by application of our helper classes (representing the artifact), we ensure that the object-oriented decomposition is congruent to the integrated provenance functionality and the underlying data model of the provenance framework (i.e., is encoded in one `class` or `method`). This causes that the functionality to integrate remains crosscutting but *not heterogeneous*, which would have made integration a lot more difficult. It allows us to integrate the same functionality at one location only. This, in turn, suggests that in more complex systems where the decomposition is not congruent, integration of the provenance concern may be practically infeasible. To this end, we need to examine more advanced case studies to reveal whether there are case studies where provenance is a heterogeneous cross-cutting concern. Finally, considering the different implementation

techniques, we found that the intuitive techniques impose severe drawbacks, as we cannot remove all feature-specific code fragments. Moreover, we hypothesize that aspect-oriented programming may be a solution to easily integrate more fain-grained forms of provenance by introducing a performance penalty.

7.2 Coarse-grained provenance for databases

As already imposed by the name, in our database-centric chain-of-custody, databases play pivotal role. Consequently, we need to integrate the provenance functionality not only on tool side but partly on database side as well. Especially, the databases in our infrastructure have to be able to check hashes, signatures, and validate watermarks. To this end, we first need a concept how to integrate this functionality, because, so far, we considered databases mainly only for storing artifacts. To this end, this section addresses the dynamic behavior of the database (e.g., procedures and functions) and not the schema (i.e., tables and views). Note, large parts of this content have been priorly published in [SSM+11].

7.2.1 A concept for coarse-grained provenance integration on database side

As a first step to integrate the provenance concern, we need a concept how and where to integrate the provenance functionality. Basically, we have two options. Firstly, we can integrate our provenance library on code level. This option has the advantage that we can re-use our Soma implementation of the first case study. Secondly, we can use procedural extensions of SQL. This way, we do not have to locate respective source-code fragments. In addition, we do not interfere with performance critical implementation parts, because we integrate our functionality on top of the database system and not within. However, then we have to re-implement the Soma functionality.

We decided for the second option to use procedural language extension, because currently we want to integrate coarse-grained provenance and thus, have no need to change the query processing itself. Moreover, based on our experiences with locating the source-code location to integrate the Validation feature, we argue that locating the source code location within a mature database system requires more time than re-implementing the Soma functionality. To the best of our knowledge, there in no experience to incooperate provenance in such a way so far. Consequently, this represents a novel concept.

PostgreSQL as example

Priorly, we decided to integrate the provenance functionality on top of a database system. As proof of concept, we integrate our provenance functionality into one mature database system. Although most database systems provide a procedural language extension, we choose PostgreSQL and to this end, PL/pgSQL[4] mainly because it is available open source.

[4]http://www.postgresql.org/docs/8.4/static/plpgsql.html

7.2.2 Provenance integration

In the following, we briefly describe how the Soma and Dendrite functionality are integrated into a PostgreSQL database system.

General Soma functionality

First, we have to enable PostgreSQL to compute hashes, signatures, and to be able to deal with watermarks. Fortunately, PostgreSQL already contains a package named `pgcrypto` that allows to compute these values. We illustrate the usage in Figure 7.6. The function `compute_sha256` determines the SHA256 hash value for an artifact identified by its `artifact_id`. For simplicity, we now assume that there is a table `raw_data` that contains a linearized representation of the artifact (e.g., as file), which is the database-side equivalent for the `getVerificationString` function from Figure 7.3 Line 12-13. The `compute_sha256` function computes the SHA256 hash value using the `digest` function by providing the desired hash algorithm as second parameter. Note that the `compute_sha256` function is implemented in own *schema* `prov_crpyto`. Schemas are equivalent to packages. Using different schemas allows us to capsule the child feature implementations of the Security feature tree and finally allows us to roll out only the required functionality.

```
1  CREATE OR REPLACE FUNCTION prov_crypto.compute_sha256(artifact_id integer)
2    RETURNS character AS
3  $BODY$
4  DECLARE
5    blob_data bytea;
6    hash character(256);
7  BEGIN
8    SELECT INTO blob_data data FROM signatures.raw_data WHERE id = artifact_id;
9    hash = digest(blob_data, 'sha256');
10   return hash;
11 END;$BODY$
```

Figure 7.6: Exemplary usage of pgcrypto library

From a practical point of view, there are scripts creating the single schemas. Due to development reasons, we offer scripts, based on the reduced feature model in Figure 7.2. Therefore, in case the Watermark feature is selected the respective Watermarking script creates the Watermarking schema and all Watermarking algorithm schemas are deployed as well.

Artifact-specific Dendrites

So far, we explained how we integrate the Soma functionality. However, the real challenges are artifact-specific functions and activating a certain feature. We have three types of artifact-specific functions that need to be integrated.

1. Identity functions. For each artifact there is a specialized identity function named by convention `compare<ArtifactRole>(aid1 int64, aid2 int64)`. The function first determines whether both `aids` refer to artifacts of the same role. If so,

there are two possibilities. In case there is a specialized semantic identity function defined (cf. Figure 6.16), we generate the respective pl/pgSQL code. Alternatively, we generate code that executes the default semantic identity function (cf. Section 2.8.2). As a result, we can automatically generate these functions, which is important regarding integration effort as a factor of minimal invasiveness.

2. Linearized representation to compute hash sums or signatures. Similar as in the object-oriented integration, we need a representation of each artifact to compute a hash sum or signature. To this end, we generate a string representation by concatenation of all attributes and compute the underlying bytes based on a predefined character set (in our case UTF-8). Therefore, this approach is very similar as generating the default semantic identity functions and can thus, automatically generate these functions.

3. Function to determine the field containing the watermark. In contrast to the first two artifact-specific functions, the solution here is rather trivial. In the artifact model (cf. Figure 6.14), the developer has to mark where the watermark is embedded. This is stored in an additional table. Consequently, there is one function `getWatermarkedAttribute(aid int64)` that first determines the artifact role, queries the additional table, and finally returns the watermarked field of the artifact identified by the provided `aid`. In contrast to the first two functions, we do not need to generate artifact-specific source code, but this functions works for every artifact.

In summary, there is practically no manual implementation effort, as we can either write functions that work in any case or generators that are able to generate artifact-specific pl/pgSQL code. However, initially developing these generators results in high implementation effort.

Activating a feature

In addition to artifact-specific Dendrites, we have to activate the right features when a tool requests an artifact or tries to store a new intermediate result. To this end, we tested two possibilities:

1. Introduction of `getArtifact()` functions and revoke the user the right to directly execute `select` queries on the actual tables as we proposed in [SSM+11],
2. Application of constraints (and triggers) to ensure that signatures and hashes hold.

The second alternative means that we do not have to change the way how artifacts are requested from the database. However, as there is no trigger that works just on `select` queries, we have to trust the check constraint upon inserts is sufficient. However, there is no explicit check, as defined in Figure A.1. The first alternative requires that we change the `select` queries to request artifacts. As we generate the source code anyway, there is no additional overhead and thus, we selected the first alternative.

7.2.3 Observations

Similarly, as for the provenance integration on client side, we now report on the observations we made when integrating provenance on top of the database system.

Integration and maintenance effort

In contrast to the client side, we cannot use the implementation techniques here, but have to generate scripts that create the required functions in the database. Recapitulate, also on client side, we used some source code generation. The idea now is to examine what we miss regarding our implementation techniques, or if code generation is a practical alternative.

Integration of the provenance features into a database system is rather simple, because it means executing scripts. However, the development of the script generator is not. Developing such a script-generating solution requires high effort and is error prone. Moreover, we need to adapt our script generators, in case we use a different database vendor. However, from a conceptual point of view it is possible. Finally, adapting such a solution (i.e., extend with new features) or maintaining it (or a variant) and keeping all other variants functionally and consistent is highly difficult. This becomes visible, for instance, in delivering Soma functionality that is not required. Moreover, the Dendrites are scattered in constraints and trigger function all over the database system. Even with naming conventions and extra schemas, subjectively the source code is hardly understandable.

As priorly stated, one of the goals of this case studies is to examine whether code generation could be an alternative. Summarily, we conclude that from point of view of the developer as well as for maintenance tasks, this approach is conceptually possible, and currently the only alternative, but far less elegant than the solutions on client side.

Effects on functional level

For the solution on database side, we exactly achieve the desired invasiveness levels as defined in Table 7.1. This is because the generated code is tailored and the complexity is in the script generator.

Effect on non-functional level

Subsequently, we examine the effects of the provenance features on the non-functional properties performance and main-memory consumption in a similar sense as on client side. Note that the footprint is not a criterion here. We could measure the lines of codes in the scripts or the additional size of the table spaces storing the functions. However, based on our integration concept, we do not modify source code files of the database system itself, and thus the footprint is not applicable.

Performance. In contrast to the client side, we do not measure the execution time for the combinations of feature and implementation, but solely the original system and the additional effort for the provenance features. This helps to answer, whether

the integration of the provenance feature is generally feasible. Our measurements are performed on a locally installed PostgreSQL 8.4 (to avoid network latencies) using a machine with an Intel Core 2 processor and 2GB main memory. Moreover, we applied pgAdmin III as tool to send the queries and record the execution times for the same image as on client side.

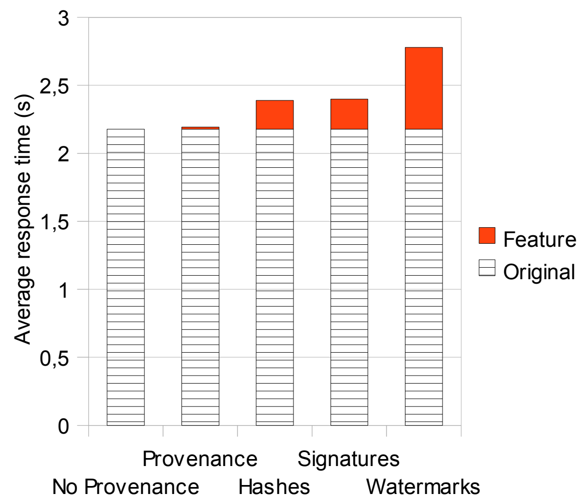


Figure 7.7: Average response time per feature on database side

In Figure 7.7, we depict the response time per feature on database side for requesting an artifact. As we are interested in the performance penalty introduced by the provenance features, which is the same for requesting an artifact or inserting a new one, we concentrate on the request. We show the additional required time in Figure 7.7. We acquire this value executing the whole procedure of requesting an artifact and the subtraction of the original system performance response time. Therefore, we assume that the response time for the original functionality remains the same.

Generally, we observe that in all cases the major cost factor is copying the artifact (the image) from hard disk to an output buffer requiring on average 2 seconds. There is no measurable difference, when the provenance feature is enabled (i.e., the specific artifact table inherits the general artifact table). Moreover, we cannot observe a difference between computing (and comparing) of hashes or signatures. Both require on average 0.22 seconds more time. By contrast the un-embedding of watermarks requires observable more time. On a more general level, there is less performance penalty than on client side. To sum up, we conclude because there is little overhead from point of performance, we are able to minimally-invasively integrate coarse-grained provenance on database side.

Main-memory consumption. Now we are interested, whether we can measure increased main-memory consumption. To this end, we monitor the maximum main-memory consumption of the `postgreSQL` process during our performance measurements as this may cause the database to crash due to insufficient heap space. We repeat this at least four times and use the median to receive reliable maximal numbers (cf. Figure 7.8).

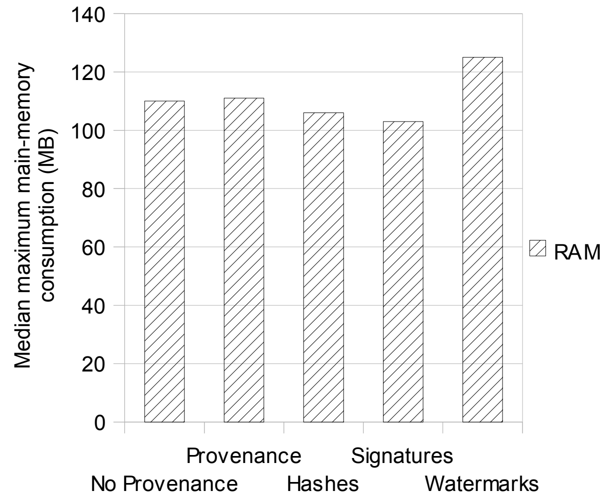


Figure 7.8: Median maximum main-memory consumption

In summary, we observed little differences in main-memory consumption. In particular, except for the watermarking, there is no measurable difference as the median is approximately around 110 MB. Moreover, we observed that the allocated main memory does not increase linearly, but continuously alternates within a range of 60 MB up 110 MB. Thus, PostgreSQL efficiently frees no longer required memory. Consequently, regarding main-memory consumption, we achieved minimal-invasive provenance integration.

7.2.4 Lessons learned

In summary, although we achieved good results regarding functional and non-functional minimal invasiveness, limitations of the script generation approach clearly state the benefits of the introduced programing techniques that we encountered in the first case study on client side. Nevertheless, the concept of integrating provenance with the help of procedural extensions of SQL is successful. This is demonstrated with the proof-of-concept implementations in PostgreSQL, which can be ported to arbitrary database systems with procedural language extension. Nevertheless, based on our experiences here, for more fine-grained provenance approaches, the solution concept seems practically infeasible. Consequently, for the case studies in the next section, we have to integrate the provenance functionality on code level, extending the actual source code of the database system itself using our programming techniques.

7.3 Scientific data management for QuEval

In the following, we present our QuEval framework[5], which is designed to allow for reliable and reproducible evaluations of high-dimensional index structures [SGS+13]. In particular, we apply the framework and the imposed challenges regarding scientific

[5]www.queval.de

data management to show the generality and limitations of our provenance framework from Chapter 2. In addition, we show how and to what extend QuEval contributes to the goal of minimal-invasive provenance integration. Finally, we report on novel interesting application scenario for minimal-invasive integration of features with special concerns regarding non-functional properties. In this case, the system is itself variable (a multi product line [RS10, SST13]) and not a monolithic one as the systems in the focus of this thesis. Note, parts of the content of this section as well as evaluation results and a detailed description on QuEval itself have been published priorly in [GBS+12], [SGS+13], or [KSS14].

7.3.1 Contribution of QuEval for the goal of minimal-invasive provenance integration

To determine the identity of two artifacts (cf. Section 2.8.2), we compare the values inside the artifacts. For instance, for an image, the pixel values, the resolution, and the height and length. Mathematically, every artifact has a multi-dimensional key with the exception of primitive artifacts that are a value, such as an `integer`. Hence, we compare two multi-dimensional keys to determine the identity of two artifacts. However, in Digi-Dak, there are several application scenarios where we have to compare one artifact (*the query*) to thousands, millions, or even more artifacts. For instance, in the development phase, we have to assemble different benchmarking data sets. In these data sets, each artifact has to be present only ones when inserting a new intermediate result. In addition, such a data set may consist of a training and an evaluation set, where there must be no artifact present in both sets. Finally, in later productive use it is a means of fake prevention to determine whether an identic (not similar) finger print has been found, as it is practically impossible lay the fingerprint twice. This is due to different pressure, distortions etc. In addition, scanning the same assay with a sensor does not result in exactly the same image (having the same color values), due to analog digital conversion and sensor noise.

As a result, we cannot *only* rely on exact comparison of values (exact match), but have to use different query types, namely epsilon distance and k nearest neighbor queries. For us, these three query types are the most important to determine artifact identity and are defined as follows:

Exact match. An exact-match query $exact(a, A)$ determines whether there exists, for a given artifact a, at least one exact copy in a set of artifacts A having exactly the same key.

Epsilon-distance query. An epsilon-distance query $epsilon(a, A, \epsilon, m)$, determines potential duplicate artifacts for a given artifact A in a set of artifacts a for further investigation. All query answers have a multi-dimensional key that has a maximum distance of ϵ from a, according to an additionally given metric m.

k **nearest neighbor.** Finally, we apply k nearest-neighbor queries $knn(k, a, A, m)$ to determine potentially identic artifacts having the same set of neighbors, which basically is a classification. To this end, we use a query artifact a, a set of artifacts A, a metric m to compute the distances, and a value k (usually between one and ten) to state the number of neighbors.

Speeding up identity queries

To summarize, the procedures and Digi-Dak in general, requires to solution to answer identity queries fast (i.e., minimize the response times). There are several possibilities to speed up such queries. We decided to use multi-dimensional index structures. There are plenty of such index structures [GG97, BBK01, Sam05] and we have to consider a lot of influence factors of the data [GBS+12, BGRS99], have to optimize parameters of the index structures [SGS+13], and even provide tailored implementations [KSS14]. Consequently, applying these indexes results itself in a multi-dimensional challenge of selecting an appropriate index structure. Unfortunately, there currently is no practical solution to compare index structures fairly. We are interested in minimizing response times and in comparing them empirically and not mathematically (e.g., O notation). To this end, we developed *QuEval*. QuEval is a Query Evaluation framework that allows for a sound and reliable empirical comparison of multi-dimensional index structure (implementation) for user-defined workloads.

7.3.2 Scientific-data management

Besides, speeding up identity queries, QuEval itself imposes provenance challenges regarding the evaluation data upon them we decide for a suitable index structure. In case, we have to work out a suitable index structure for a new use case, we can narrow down the list of candidate index structures to a small number. The selection is based on our experiences and experimental results from the literature. However, even with this small selection the number of runs, especially to optimize the index parameters, is easily larger than 100. For instance, for one use case in the empirical evaluation of [SGS+13], we tested over 200 different index configurations to find the best parameters, which have to be executed several times to ensure statistical soundness.

QuEval and provenance

We consider each execution of a test case as a process returning query answers, according to the data model of our provenance framework. However, we are not primarily interested in the result itself, but the amount of time that is required to compute the result (i.e., the response time). The result itself is just required to ensure that the index works correctly. In fact, we invoke an identity function on the result set and take the sequential scan, which is the approach to beat, as reference. In Figure 7.9, we show an example process that executes several exact-math queries (one for each artifact in Q). Then, QuEval collects the results of each query, which are in our case *tuple identifiers*, allowing easy comparison of query results. The basic challenges regarding validity of the measured response times, and to this end, concerning provenance are:

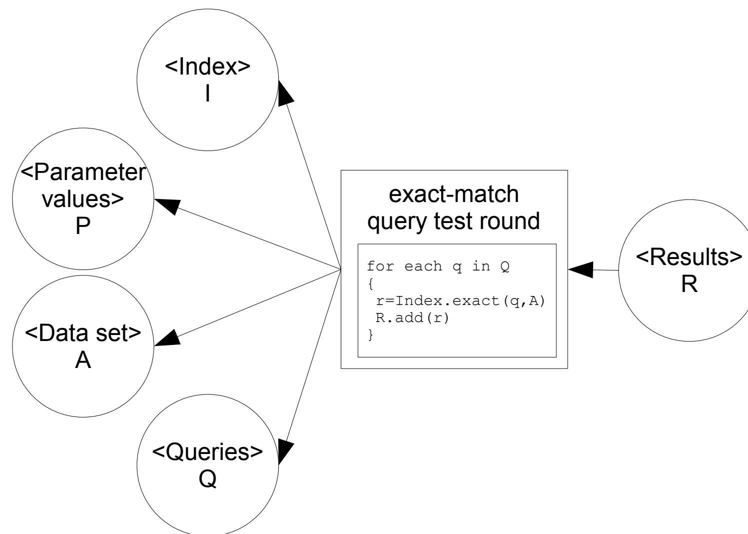


Figure 7.9: Query execution in QuEval

1. We need to be sure that a measured response time belongs to the designated input data set.

2. As far as possible, we have to record influence factors, such as hardware and software specification of the applied machine.

3. We would like to have a possibility to verify that a given index-structure implementation works as proposed by the inventors - Or in other words: Is the index what it claims to be?

Input output relationship

In the research process, we evaluated for instance, the influence of different stochastic data distribution on the performance of index structures. To this end, the properties of the data sets are the same regarding number of points, dimensionality etc. Consequently, it is not possible to determine what data set is used by looking at the data set itself. This led to questions whether the right data sets were used. Once it concluded in the consequence to run the whole evaluation again; one and a half week before submission deadline (which required four days). As solution, we integrated parts of our provenance functionality by introducing a naming convention that included a hash value computed over the first 100 points in the data sets for the data sets themselves as well as for the result sets. So, here we could use our provenance SPL as well.

Influence factors

Generally, we cannot reach functional dependency between two executions of test runs in QuEval using the same input parameter and resulting in the same measured response times. Anyway, a direct functional dependency is not required. The assertion is that

the same workload using the same hardware configuration results into measured value having the same stochastic distribution. To this end, computing robust mean values or medians is sufficient to compare index structures. In this sense, we treat a hardware configuration in the same way as tool-specific parameters and record them as well.

Verification of index structures

For an index implementation, we can determine whether the result is correct. However, we cannot verify whether an implementation of an index-structure works as designed by the original inventors. Nevertheless, we can use reference implementations and compare newly implemented ones, step by step whether they work in the same way. In addition, and for us of more interest, there are generalizations of index structures for the R-tree family. R-trees are balanced trees to index multi-dimensional data. The basic idea is to use minimum bounding rectangles as nodes to group regions that contain points. When searching for a point the concept is to exclude large parts of the data space as early as possible resulting in logarithmic complexity of an exact-match query for example [Gut84]. However, it turned out that when building an R-tree overlapping nodes are created, which unpredictably slow down queries as multiple paths within the tree have to be considered. Sometimes even the whole tree has to be searched. As a result, several extensions and variations of the original R-Tree have are proposed to address the challenge of overlapping nodes [GLL98]. For instance, different ways how to split nodes upon insert [AT97, BKSS90, BKK96], clipping nodes to avoid overlaps [SRF87] by creating more nodes, using spheres [WJ96] instead of rectangles, or combining spheres and rectangles [KS97].

Anyway, most algorithms remain the same. The difference is in how and when to split the nodes leading to a generalization of R-trees in the *Generalized Search Tree* framework [HNP95]. This generalization allows us to use the general properties to check the correctness of a tree implementation. In fact, we integrated an index named R-variant that can be extended in the sense of GIST and we can verify that at certain points in the algorithms the correctness of the tree. However, these correctness tests are cost intensive and thus, cannot be present in all variants of the index structure. Consequently, they have to be an *optional* feature. As a result, we found another scenario that requires for integration of new features into existing implementations that we discuss in more detail subsequently.

7.3.3 New features for tailored index structure implementations

Priorly, we identified a new application that requires to integrate optional features into highly performance relevant implementations. An in depth analysis reveals that there are several other features, which have to integrated on source-code level [KSS14]. Examples are allowing online update, multi-threading [Her13], index visualization [BSG13], or privacy awareness [GSKS13]. Moreover experiences gained in the QuEval project running over several years (cf. [SSG$^+$13]), reveal that even small changes in the source code can

have a significant influence on the overall performance. Under certain circumstances, we are able to measure the influence of evaluating a single `if` condition, which can be excluded in certain variants. Finally, we encountered that development of indexes takes place in an iterative process. That means there is first a working implementation, which is then optimized or extended with additional features.

A future research direction

The consequence is that there cannot be a one-size-fits-it-all index structures implementation and that we have to be able to integrate new features by avoiding performance penalties [KSS14]. Therefore, the task is similar as the objective of this thesis: To explore whether modern software-engineering techniques are suitable to integrate new variability dimension into monolithic systems showing the significance and importance of the topic. The difference is the application domain (indexes instead of provenance). However, there is also a fundamental difference. In this thesis, the scope is on integrating a new variability dimension into monolithic systems. However, the QuEval framework is not monolithic, but the infrastructure consists of variable parts itself [Tob13]. Moreover, each index is itself a software product line containing dozens of currently used variants.

To this end, we hypothesize that integrating new features consistently, is even more challenging and thus, beyond the scope of this thesis. Nevertheless, this outlines a new research direction in the field of multi product lines.

7.3.4 Results and lessons learned

The objective of integrating tailored multi-dimensional index structures is to minimize performance penalties within the procedure to insert new intermediate artifacts. This way, we want to minimize the effects regarding performance when integrating provenance compared to the original solution contributing to the goal of minimal-invasive provenance integration. Generally, we achieved significant response-time reductions for all considered query types and published parts of the results in well-known database conferences, such as VLDB [SGS+13] and RCIS [KSS14]. In the sequel, we present evaluation results. For all results, we use the same three data sets. The size and dimensionality refers to forensic data sets or have the same characteristics as forensic data [SGS+13]. In particular, we use three different sizes and dimensionalities: (1) 16 dimensions and 10,992 artifacts representing hand-writing features [AA97], (2) 43 dimensions and 411,961 artifacts containing spectral features used to adjust our sensors [KFV11], and (3) 50 dimensions and 131,000 artifacts covering scientific data from physics [RYZ+05]. To abstract from stochastic distribution, we use, per data set, two additional artificially created data sets. The first has a uniform and the second a multivariate Gaussian distribution (MVG) [ADV96]. For the evaluation, we run 120 measurements and compute robust mean values using a γ-trimming approach with $\gamma = 16.67$ to ensure statistical soundness.

We select promising index structures either from literature or based on our own experiences also covering a broad range of different index classes addressing completeness. We do not go into details of how the indexes work, as they are complex structures and

the explanation would distract from core points. Moreover, in this thesis, we are not interested in explaining why there is a difference in performance, but we only want to select the best index. To this end, we refer the reader to the original literature proposing the index structures. In particular, we used a sequential scan (SEQ) as worst case reference, an R-Tree variant based upon the idea of the Generalized Search Tree [HNP95], the k-d Tree [Ben75], the Pyramid Technique (Pyr) [LK03], the Vector Approximation File (VA) [WB97], the Prototype-based Approach (Prot) [GFN08], and p-stable Locality-sensitive Hashing (p-stable) [DIIM04].

Results for exact matches

Normal identity queries are executed without distance computation as exact-match queries. For this type of queries, we recognize performance gains up to five magnitudes compared to a sequential scan. In Figure 7.10, we depict exemplary results showing that each applied index clearly outperforms the sequential scan. Moreover, we observe that there are robust indexes regarding changes of the stochastic distribution, while others are not. Anyway, for all tested use cases the Pyramid Technique delivers the best results. To this end, we use this index for high-dimensional data spaces having at least 16 dimensions as they occur in the Digi-Dak project. However, additional evaluations [KSS14] reveal that the observations regarding the Pyramid Technique cannot be transferred to densely-populated low-dimensional spaces (up to 10 dimensions). For these data, we observe a significant increase of the response time of several magnitudes. Therefore, we have to use different index structures, such as our variant of the Dwarf [SDRK02] that we applied in [KSS14]. It outperforms the Pyramid Technique by at least one magnitude. In case the data space is small enough to fit into the main memory, we can also linearize the data space into a Cube, resulting in the best possible response time [GCB+97]. Summarily, independent of the characteristics of the data, we can identify and optimize a suitable index structure supporting this kind of identity queries using QuEval.

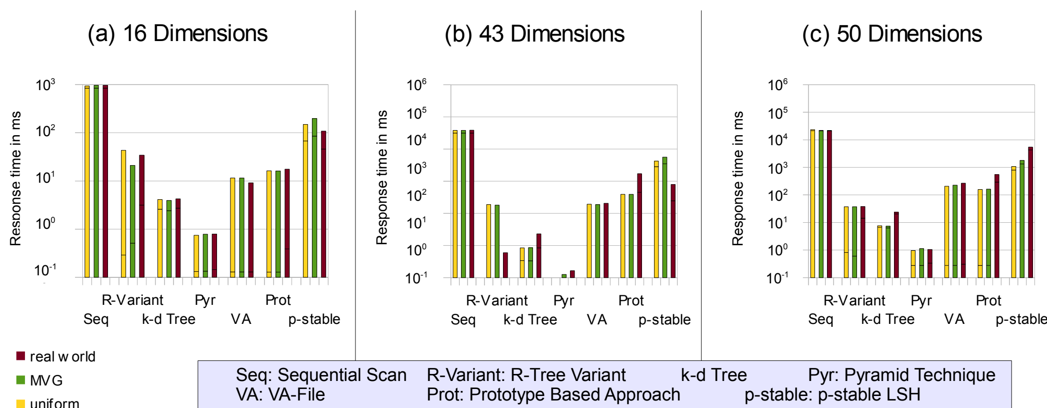


Figure 7.10: Exact-match query speed up for low-populated spaces - Adapted from [SGS+13]

Results for epsilon-distance queries

Identity functions that have to be robust, for instance against sensor noise, are defined as epsilon-distance query. An evaluation of our index structure implementations considering epsilon-distance queries is conducted in [Weh13]. The results are similar to the results of exact-match queries. The basic result is that most indexes outperform the sequential scan serving as reference approach. However, as depicted in Figure 7.11, we first observed that, as expected, the response time per query, are clearly higher. To this end, we prefer using exact matches if possible and not to compute exact matches as epsilon-distance query with $\epsilon = 0$. Secondly, the difference between all index structures is smaller than for exact-match queries including the sequential scan. Finally, for all measured use cases, the R-Variant delivers best results. In summary, although the performance gain is smaller than for exact-math queries, we can significantly speed-up this kind of identity query as well.

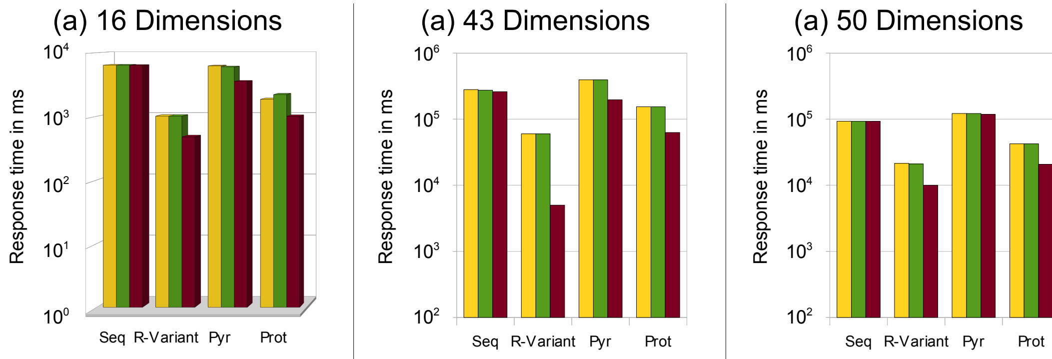


Figure 7.11: Epsilon-distance query speed up for low-populated spaces

Results for k nearest-neighbor queries

Another important query type is the k nearest-neighbor query, used to define identity functions (similar to a classification) and also in different application scenarios in the Digi-Dak project itself. For this query type, we have to be aware that the Prototype-based approach and p-stable locality-sensitive hashing are approximative index structures. This means that they not necessarily deliver the same result as the reference approach. This may include neighbors that are not the exact nearest neighbors, but have a similar (i.e., larger) distance. The fraction the correctly found nearest neighbors is known as accuracy. To ensure a sufficient result quality, we define the minimal accepted accuracy as 0.9, meaning that 90 percent of the neighbors have to be correct. We obtain such values by optimizing the parameters of the index structures accordingly [SGS+13].

The results in Figure 7.12 are very similar to the ones of epsilon-distance queries. However, we observe large differences between the different stochastic distributions. For example, the tree-based index structures highly excel for real-world data. However, they require for a different parameter configuration.

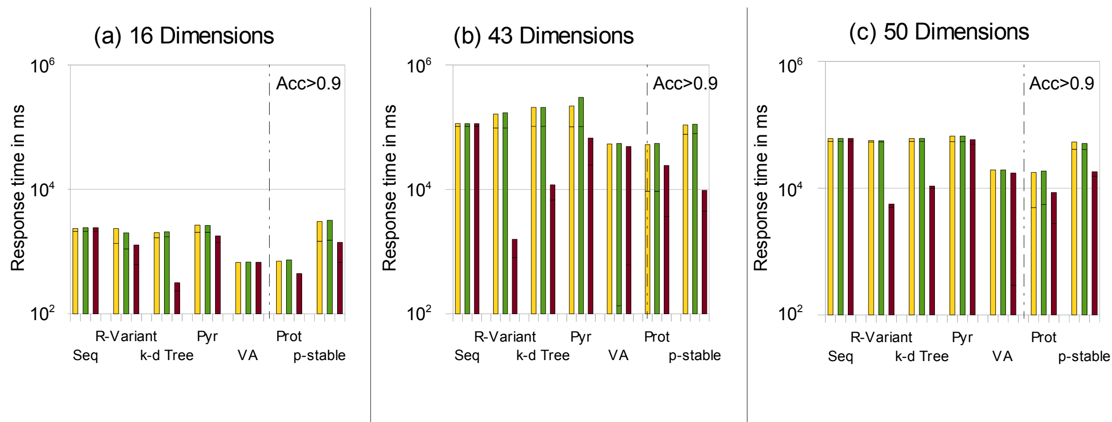


Figure 7.12: k nearest-neighbor query speed up for low-populated spaces - Adapted from [SGS+13]

Case study summary

In summary, our results reveal that using tailored index structures, identified with the help of QuEval, results in performance benefits of at least one magnitude. In fact, using the indexes makes the provenance integration with enabled identity detection feasible in practice. It furthermore reveals, without optimizing the indexes and comparison of different alternative index structures much of the possible performance gain is probably lost (e.g., if parameter values are not guessed correctly). In some cases, the performance is even worse than with using an optimized implementation of a linear scanning approach. This is the default approach that works in any case and the observation is a well-known phenomenon [WSB98]. The performance optimization often results in increased main-memory consumption, which is another factor that we want limit as part of minimizing the effects on non-functional properties. To this end, the main-memory consumption is considered in QuEval as well, allowing us to select a suitable index structure.

Summarily, the results of the QuEval project are an important part supporting the goal of minimal-invasive provenance integration. Moreover, we learned that we could use our provenance framework and parts of the implementation of our provenance SPL to address challenges regarding scientific data management, showing the versatility of our solutions.

7.4 Contributions and conclusions

In this chapter, we perform another step of the research agenda developed in Section 4.3.4. Based on the approach to tailor the database schema of the provenance data store from Chapter 5, the objective is to identify benefits and drawbacks of our approach in general and of the implementation techniques in particular. To this end, we integrate the provenance functionality into the first exploratory case studies to report on our observations. In summary, this chapter makes the following contributions in order to achieve the objective of the chapter.

1. Integration of the provenance SPL into different parts of the infrastructure using different implementation techniques in order to identify benefits and drawbacks of the approach in general and the implementation techniques in particular. The integration is conducted on client *and* databases side.

2. The application of multi-dimensional index structures to minimize the effect of the provenance integration on performance. This results in a general procedure to tailor multi-dimensional indexes bundled in our QuEval framework [SGS+13] offering large performance benefits. Furthermore, it issues a new research direction as it shows the necessity for integrating novel variability dimensions into itself variable infrastructures with the objective to gain an optimized performance [KSS14].

On a more abstract level, our results clearly show the drawbacks of the intuitive implementation techniques. Although still applicable, they impose severe drawbacks, which are in opposition to our goal of minimal-invasive provenance integration. Moreover, we observe that most provenance features are crosscutting, but homogeneous, which is beneficial for the integration. Moreover, on coarse-grained provenance level the object-oriented decomposition, except for the validation feature, is congruent to the provenance data model and integrated provenance functionality. This allows us to easily encode features in refinements or aspects that are bound to object-oriented structures (i.e., methods or constructors) resulting in low integration effort. However, our observations result into the hypothesis that this characteristics does not hold for more fine-grained provenance approaches. To verify or decline this hypothesis, we have to use more complex case studies in the next chapter.

8. Fine-grained provenance integration into complex systems

In the prior chapter, we integrated coarse-grained provenance functionality into our case studies. In this chapter, we select more complex systems and enhance them with provenance functionality. The goal is to reveal additional insights regarding our goal of minimal-invasive provenance integration. In particular, we are interested whether our findings from the prior chapter are still valid for these kinds of case studies. Especially, we are interested whether our observations regarding the homogeneous nature of the provenance concern remains valid. To this end, we first integrate fine-grained provenance on tool side and second on database side.

8.1 Granularity refinement on tool side

In Section 7.1, we integrated coarse-grained provenance into the tools of our infrastructure. In the following, we implement provenance granularity refinements into one of these tools. Therefore, we use the *equalization tool*, which has been published in [KCDV12]. We select this case study, as it is a classic example for a tool in our infrastructure that is designed as proof of concept, as well as for evaluating novel preprocessing approaches. Moreover, for this tool the authors present a detailed technical and mathematical description, which is used to validate the provenance integration. Note, the initial analysis regarding the original tool is published in [KS13a]. Moreover, the basic concept to exploit the object-oriented decomposition in order to allow for tailored provenance capturing is contained in [SSS12b]. Finally, the provenance integration and parts of the evaluation are published in [KS13b].

Motivation

Now, we shortly motivate why we need more fine-grained provenance than integrated in the first exploratory case study and why this functionality is not always integrated. To

motivate the requirement for more fine-grained provenance within a tool, recapitulate the semantics of the Validation feature. The basic purpose of that feature is to verify the identity of an artifact. We re-invoke the process (tool) that created this artifact by providing the same input and execute the identity function. Imagine the identity does not hold. Then, we first have to be sure that the provided input was used as desired, to exclude that the error occurred on tool side. Therefore, we apply more fine-grained provenance capturing functionality. Similarly, we verify that an implementation of the tool works as it is designed. The observations of our prior case studies reveal that the provenance functionality significantly slows down the tools. To this end, we do not integrate this functionality in every tool in case the Variability feature is selected, but use this as an additional optional feature.

8.1.1 Expected insights

So far, we revealed that integrating the provenance concern is manageable, if we can exploit object-oriented decomposition. To this end, our approach integrates generated source code (i.e., the classes representing the artifact). However, we also observe that for the integration of the Validation feature, we have to manually review the source code. The basic question that arises is: What effort has to be performed in order to integrate the desired provenance capturing functionality. To this end, we now explore whether we can exploit the object-oriented decomposition to ease integration effort. Moreover, we explore what additional effects regarding the goal of minimal-invasive provenance integration can be observed using different implementation techniques. Finally, we are interested into what knowledge on architecture details of the tool is beneficial for integrating the concern.

Correctness of extracted provenance data

One challenge that arises in order to answer the previously issued question is whether the extracted provenance data is correct. Fortunately, in [KCDV12], the authors state the input and result artifacts as well as the executed functions (i.e., processes with functional dependency between input and output). We depict their specification in Figure 8.1.[1] Moreover, Figure 8.1 contains the view on the provenance data that we have at different layers of our provenance framework. For the Workflow layer, we only know there is a process *equalization* that uses a `SensorScan` I as input to produce an `EqualizedImage` I'. However, in case we invoke the process with the same input, the output is not necessarily the same, as the tool-specific parameter `blockSize` b is missing. For functional dependency in the Existence layer, we need this parameter as well. Therefore, it is added in the tool-specific Derivative of the Validation and the Equalization tool feature (cf. Figure 6.16). We denote the change in layers by using $f_{<process>}$ to indicate that functional dependency between input and output holds. To this end, the executed function of the tool is: $I' = f_{equalization}(I, b)$.

In [KCDV12], the authors state that the equalization tool consists of three functions and that these functions use only parts of the input artifact. The first functions results

[1] A more detailed and visual explanation is given in the Appendix Figure A.3.

in a filter mask that is used in the second function, together with the `blockSize` and the topography data of I etc. The expected gathered provenance data therefore is: $I' = f_{scale}(I.inten, f_{blockgradients}(I.topo, f_{lowpass}(I.topo), b))$. However, we are particularly interested, whether we can detect and locate additional computation steps that modify the data.

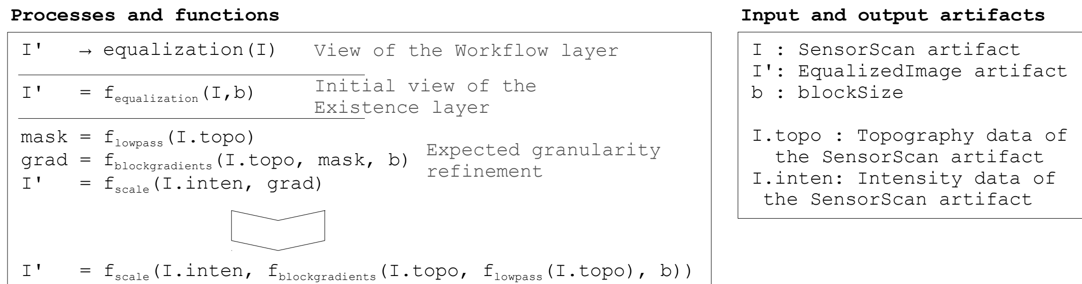


Figure 8.1: Expected extracted provenance data - Based on [KCDV12]

8.1.2 Provenance integration

As we now know what the correct extracted provenance data is, we integrate the required provenance functionality into the equalization tool. To this end, we have to locate the method(s) implementing the priorly stated functions, record their execution, and the provided input data. As first step to integrate the provenance functionality, we shortly analyze the architecture of the implementation.

Analysis of the implementation

An analysis of the tool reveals that the tool is more complex as suggested by the function specification (cf. Figure 8.2). Moreover, we find that the tool consists of two parts. One that contains the logic and another that performs the image transformations. For the second part, the authors use an existing image processing library ImageJ[2].

Integration concept

The tool basically invokes for every image transformation the ImageJ library. Therefore, the basic idea is to enhance every method in this API, record the supplied arguments, and compare whether there is a complete sequence from input to output. Moreover, there shall be no modification of the artifacts between the single computation steps (i.e., method executions) and we determine in how far the extracted provenance data corresponds to the announced specification. By concept, this means that we have to integrate Dendrites into 144 methods of the class `ij.IJ`, which contains static processing methods and into 162 methods of the class `ij.ImagePlus` and several more methods for the classes `CurveFilter, ColorProcessor, and ImageProcessor` to provide a

[2]ImageJ project website http://imagej.nih.gov/ij/

general solution for this API. A seamless integration of Dendrites, therefore is only possible using aspect-oriented programming, as we can use one Dendrite to cover multiple methods and classes. However, we can limit the number of methods by automatically analyzing the source code of the tool, which methods of this API have been used. This reduces the number of methods where to integrate provenance to 15.

We conduct our initial provenance integration using aspect-oriented programming trying to use as less Dendrites as possible. To this end, we annotate all used methods of the ImageJ API and the `execute()` method of the tool, where the Validation feature functionality is already integrated. As this may result in a significant performance decrease, we repeat the integration for different implementation techniques and try to optimize the initial integration.

Two-phase integration

In order to map functions of the specification to methods in the implementation, we need to record their execution, the order and hierarchy of the execution, as well as respective input and output artifacts. To this end, we use a two-phase integration [SSS12b]. This approach integrates two Dendrites into each method of interest and performs the following functionality:

Method begin. When a method is called, a Dendrite determines the input artifacts and the method name. Moreover, it puts the method on the current stack view to allow refinement and track additional input. In addition, we need to record the current status of the class members by invoking the associated identity function. This way, we reveal hidden input and output. This mainly corresponds to member accesses or modifications without using a `get()` or `set()` method. Note, this is an optional Soma feature and only activated in case we assume that there is such hidden input.

Method end. At the end of a method, a Dendrite initiates the output artifact determination, performed in the Soma. This includes the return values as well as modified class members (e.g., modified by `set()`). Hidden output artifacts are determined by invoking identity function on all class members as well as arguments passed by reference and compare them to their prior state at the begin of the method. In case the identity does no hold they have been modified.

Regarding our implementation techniques that means that we have to manually introduce *two* Dendrites per provenance augmented method for the intuitive approaches and the advanced preprocessor approach. Using feature-oriented programming, we have to write one refinement per method, which is then automatically composed. For aspect-oriented programming, we hypothesize that we can use one pointcuts to bind an advice to several (or even all) provenance augmented methods taking advantage from the homogeneous nature of the provenance concern.

File name	#classes	#methods	#members	#lines of code
`main.java`	1	1	1	72
`RasteredImage.java`	1	52	29	1039
`RasteredImageTools.java`	1	7	0	342

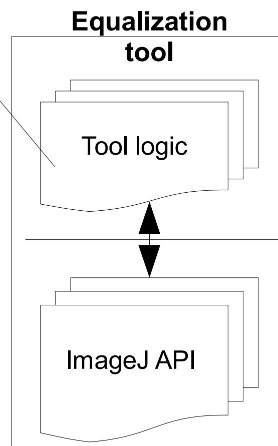


Figure 8.2: Architecture of the equalization tool

8.1.3 Initial integration

We conduct our initial provenance integration using aspect-oriented programming by using the Dendrites from the exploratory case study capturing the data requested from and send back to the database. In addition, we add one Dendrite to the `execute` method of the tool to detect whether the tool-specific parameter is applied. In the following, we explain what additional functionality in the Soma and what additional Dendrites are required to record the final provenance data shown in Figure 8.3. Therefore, we report on challenges we faced and introduce our applied solutions as published in [SSS12b]. Moreover, we summarize what properties should be avoided when implementing such tools and what alternative solution is preferred in order to reach the goals of minimal-invasive provenance integration.

Invisible input and output

In our two-phase integration, we are already aware that there may be hidden input or output and considered members of the classes which are explicitly included (via `get()` and `set()` methods) as well as implicitly used members with the help of the identity functions. However, in the case study, we encounter additional invisible input due to calls of static variables of different classes and implementation details of the ImageJ library. This library contains internal storage capacity (a stack) that is used to transform images. For instance, the tool computes the difference of two images. Both images are assembled by preprocessing steps and therefore present on the stack of the ImageJ library. Then, at some point, the source code calls the ImageJ library to execute

some function without specifying the input, which is then taken from the stack. We discovered similar challenges, for instance, in the batch processing component of JDBC driver that executes a series of priorly added SQL statements.

As solution, we need to have access to the internal storage to add input artifacts to the method execution. As this functionality is not case-study specific (i.e., works for all tools using the ImageJ library) and is hidden in the Soma this does not affect the integration effort criteria of minima -invasiveness. However, it would be beneficial, considering the integration, as well as minimizing the performance overhead to avoid such hidden input. In particular, we favor that input is either directly passed as argument in the method call or retrieved via a `get()` method. In the same way, output should be either only a return value or set explicitly via `set()` methods.

Determining provenance fragments

The basic goal of our provenance capturing is to assemble a graph that shows, for instance, which input artifacts are used to create an output artifact. The data we collect, however, are fragments of the desired graph at different levels of granularity. Therefore, we have to assemble the fragment to form that graph. It turned out that the basic assumption that method names correspond to the functions of the specification is not generally applicable. For instance, the tool calls the generic `ij.run()` method and passes the desired filter function and input as argument: `"image1=Raw operation=Subtract image2=Filtered"`. As a result, we need to integrate special parsing and assembling logic for the `ij.run()` method in the Soma and link the execution of that method to the Soma implementation using a separate Dendrite.

Technically, there are two pointcuts matching the `ij.run()` method. Therefore, the more specific one (matching only this method) is executed. Consequently, we do not have to change the Dendrite augmenting all methods of that ImageJ Library resulting in two required Dendrites. To conclude, we suggest that developers avoid such generic functions and use explicit method calls not requiring parsing of passed arguments.

Issues regarding the object-oriented decomposition

To map methods to the functions of the specifications, we used a third Dendrite augmenting all methods with provenance capturing capability that are directly called form the `execute()` method of the tool. To this end, we can map executions of any method within the ImageJ library to one method that is supposed to correspond to one function in the specification. Our observations reveal that this works well for most functions, but not for everyone. For instance, there is a method that contains several computation steps. Our solution then was to use the extract method refactoring, which creates separate methods that then correspond to functions of the specification. This modification of the tool may be seen as an invasive change to the original tool. However, we argue that it is not - for the following reason. We do not change the intended function of the tool, due to the nature of the refactoring not to alter the way how the program works. Moreover, this refactoring helps to separate concerns within the implementation

improving understandability of the source code generally and therefore this should be applied anyway.

Generally, we favor that the decomposition of the tool reflects the intended functionality. That means that there is a method for every function. In addition, in complex programs, locating the respective methods may be difficult. Therefore, it would ease provenance integration, if there is an additional tool or language support helping us to localize the respective source-code locations. For instance, the functionality can be implemented in a separate feature, the methods could stick to a naming convention, or use annotations in the method comment that helps us to define pointcuts integrating the Dendrite into the desired methods.

Provenance granularity level

Augmenting all used methods of the ImageJ library result in vast number of provenance graph fragments of different granularity levels. To assemble the provenance graph associated to the specification, we need less than 30 graph fragments. Our initial integration attempt results in more than 32,000,000 graph fragments using the evaluation image having 5439×2943 pixel. These fragments mainly show Where provenance, when results of a computation are copied from a result buffer to the corresponding object, pixel by pixel. Moreover, the execution time of the tool increased from 12 minutes to more than an hour. As a result, we decided to use specific pointcuts referencing only the desired methods. Nevertheless, using one pointcut is a good initial idea to identify the important methods. Moreover, this result indicates we can even track Where provenance for parts of the implementation and it is another argument why we need tailored provenance capturing solutions.

Results

The results of our initial provenance integration, using aspect-oriented programming, are several beneficial as well as problematic properties for the programming guide. Moreover, and in contrast to our initial assumption, it is not possible to augment the whole API in order to collect the desired granularity level of provenance data. Nevertheless, there still is a benefit using aspect-oriented programming regarding the integration effort criterion. The benefit compared to feature-oriented programming is that we can use one pointcut for several methods and do not have to write one refinement per method or compare or even integrate them manually into the source code for the remaining techniques. On a more general level integration still is manageable and, if performed once, is composed automatically. However, for larger case studies integration effort may be a severe challenge.

8.1.4 Extracted provenance data

In Figure 8.3, we depict the extracted provenance data. For the Workflow layer (`I`) and coarse-grained Existence layer data (`II`), we also depict the expected provenance data from Figure 8.1. For the Existence layer, the captured provenance data reveal that

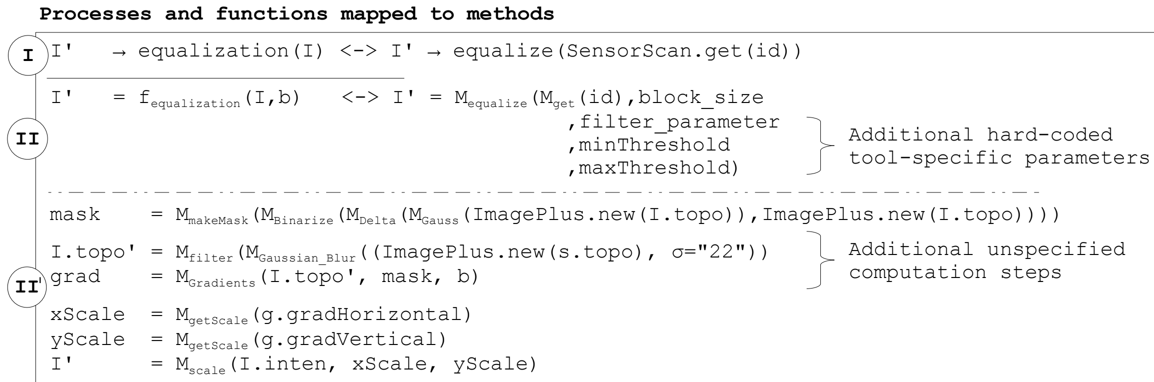


Figure 8.3: Extracted provenance data

there are *additional* tool-specific parameters. However, they are passed as hard-coded literals and therefore not in contradiction to the specification of the tool. Nevertheless, they should be recorded as well, which is performed by our provenance integration. To indicate that this is an actual provenance graph and not a specification, we do not use the symbol f naming a function but use m for method. Note, for visualization purposes, we skipped the complete qualified names of the extracted methods. In addition, we depict calls of static members as `ClassName.methodName()` and constructor calls as `ClassName.new()`. Finally, we note that for primitive values and strings, the values (or hash values) are recorded but not depicted, again due to visualization reasons.

The granularity refinement of the Existence layer (`II'`) is the first complete refinement for that we can assemble a complete provenance graph. That means that there are no intermediate computations steps that modify the named artifacts in the specification. The extracted provenance data shows more details than the specification, but can be easily mapped to that specification. Consequently, we consider our provenance integration as correct. A manual review of the source code confirms this assumption. In particular, we first observe that the `mask` is computed as the difference of the original topography image and the filtered topography image. Moreover, we see that the `grad` artifacts containing the gradients is a complex artifact and that the gradients are computed of the filtered topography image `I.topo'` using a Gaussian blur. Finally, we see that the parameter `b` is passed to that method. This allows us, for instance, to verify that the correct block size parameter was used upon validation of a computation result.

To summarize, the extracted data is correct, contains interesting additional insights, and allows us to verify whether the tool-specific parameters are used, which was the motivation for the need of the provenance granularity refinements. To this end, we consider our provenance integration as successful and re-implement this functionality with different implementation techniques for the evaluation in the remainder of this section.

8.1.5 Performance considerations

In the prior section, we give information regarding the implementation effort. Moreover, we state that the granularity refinement, independent of the applied implementation technique, is purely spectative. Therefore, we now concentrate on the effects regarding non-functional properties as final criterion of minimal-invasive provenance integration.

Initial situation and performance tuning

Recapitulate that for every tool in our infrastructure, performance is an important factor, which is one of the motivations for the goal of minimal-invasive provenance integration. To this end, we analyze, as initial step, the performance of the equalization tool in [KS13a]. We use the same image as for all prior evaluations and perform our measurements on an Intel Core2 duo in a statistical sound manner. The result consists of two main observations:

1. The primary focus of the initial implementation is to serve as proof-of-concept. To this end, performance is of minor importance. However, an overall response time of more than 10 minutes is barely acceptable mainly caused by reading the input artifacts (cf. Figure 8.4(a) first column). More suitable would be a response time of several seconds up to one minute.
2. By contrast to the first observation, the amount of required main-memory does not impose a problem based on current main memory sizes. Therefore, we can use more main memory in order to speed up computation.

The consequence is that we first need to tune the performance of the tool itself. The basic idea is to use techniques known from databases to improve the performance. Then, we compare the performance penalty introduced by the provenance integrations with the original and the tuned implementation. This comparison allows us to draw even more sound conclusions regarding the practical applicability of our approach.

The results of our tuning as well the applied optimization are published in [KS13a]. The basic result is that we can reduce the average response time from 10 to 2 minutes on an Intel Core2 duo (98 seconds on an Intel Core i7). Moreover, using multi-threading there is a linear scale up in throughput imposing also a linear increase in required main memory. Therefore, the above stated performance requirements are matched. The question that remains is: How much of the gained performance is lost due to the provenance integration and are there implementation-technique specific differences?

Comparison after provenance integration

In the following, we present the results of the evaluation to determine implementation technique-specific impacts. In addition, we compare the effects of the provenance integration in comparison to the performance gain of the tuned version of the equalization tool.

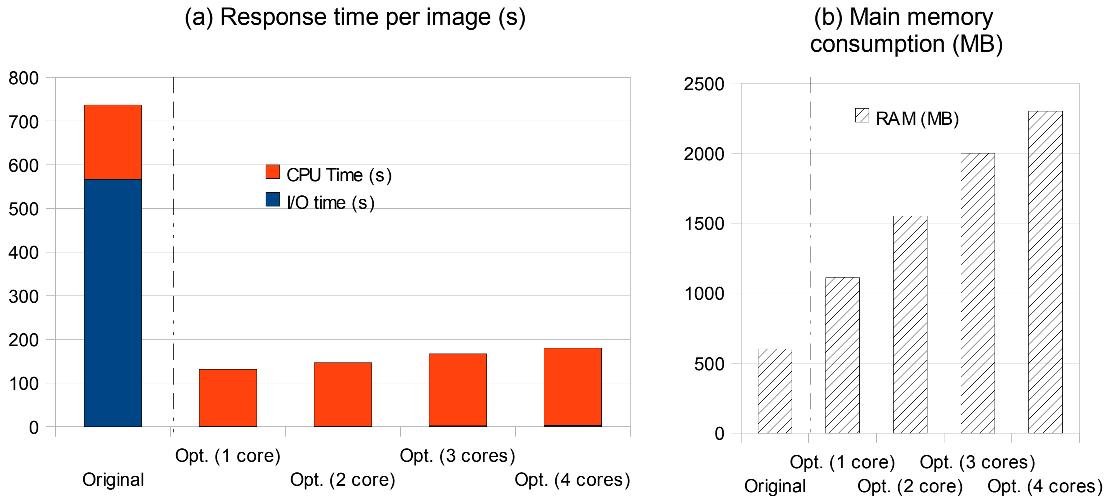


Figure 8.4: Initial and optimized non-functional properties

Implementation-technique specific impacts. Regarding the response time of the tool, we cannot measure an implementation-technique specific impact. In Figure 8.5(a), we depict the average response time. The difference between the single implementation techniques is within three seconds, which is, compared to the overall response time, negligible. By contrast, we do measure a small implementation-specific influence regarding main-memory consumption. Our results indicate that there is a small overhead when using aspect-oriented programming. In addition, we note that, in case we collect the Where provenance data for some parts of the tool resulting in more 32,000,000 graph fragments, we do see an observable performance difference. It is introduced by the way the Dendrites are called, especially regarding aspect-oriented programming. That means, in case there is an application scenario having these call frequencies, we expect differences with respect to response time are the major selection criteria. However, here and in all prior case studies this is not the case.

Comparison. Compared to the performance gain of the tuning, there is little overhead for the provenance integration in general. For instance, we observe an increase of the response time of about 30% after integrating the provenance concern. This results in an average increase of over 40 seconds. However, the response is still 23 percent of the initial response time. Moreover, our approach allows for additional, more specific optimizations at the cost of additional, tool-specific integration and development effort. The major cost factor in provenance capturing is the execution of identity functions to identify hidden input and output. In case, we are sure there is no hidden input, we can skip their execution. Then, we can barely observe the performance overhead.

To summarize, our results indicate that minimal-invasive integration with respect to non-functional properties is feasible using our approach. This conclusion is based on

our observation that, compared to the inherent performance potential, the provenance integration is negligible. In addition, we do observe some implementation-technique specific behavior, but not in magnitudes that allow us to state that one of the techniques is inappropriate for these kinds of case studies. The major reason therefore is that the major cost factor lies in the Soma functionality not within the number of calls of the provenance functionality by Dendrites, which may change in the next case study.

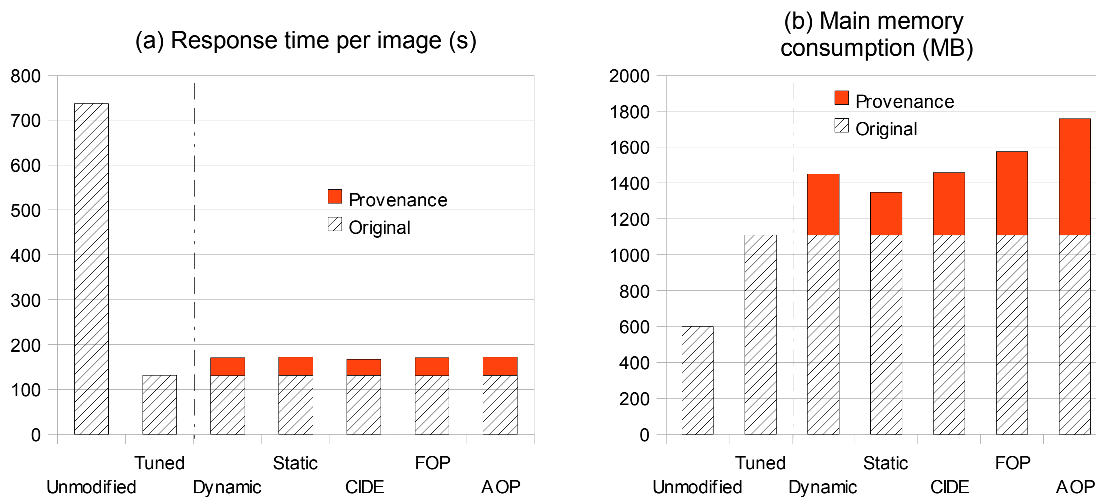


Figure 8.5: Initial and optimized performance

8.1.6 Lessons learned

The general conclusion of this case study is that we can again exploit the object-oriented decomposition to augment a program with tailored provenance integration. Moreover, the integration effort remains practically feasible. In particular, we use homogeneity in the infrastructure (e.g., an image processing library or communication infrastructures such as the JDBC driver etc.) to integrate Dendrites into the implementing methods decreasing the integration effort. However, there are several properties that either ease or harden the provenance integration. This includes for instance, hidden input and output, or additional provenance fragments resulting in additional computation effort or improper object-oriented decomposition. Some of these properties result in the necessity to integrate additional functionality into the Soma in order to assemble a complete provenance graph from the collected provenance data fragments. In summary, the best implementation technique to integrate the provenance concern in a minimal-invasive way for these kinds of case studies is aspect-oriented programming. This is mainly based on the minimization of integration effort and based on the homogeneous and cross-cutting nature of the provenance concern. In addition, there is only a small and to this end negligible additional main-memory consumption using aspect-oriented. Nevertheless, similar to the first case study, we encounter that the complexity and syntax of AspectJ slows down integration. What we require is a technique that also allows to apply the same refinement to several methods, which is exactly the motivation for aspectual feature modules [ALS08]. This would be completely sufficient for our purposes.

Relationship to recent results

Recently, the CAPS Framework[3] is proposed [BFH14] in the domain of provenance integration for scientific workflows. The basic idea is to use aspect-oriented programming to integrate provenance for scientific workflows. To this end, their motivation, assumptions, and objectives are very similar to ours. They even use the term minimal-invasive integration in the same sense [BH12]. They also do acknowledge that provenance is a cross-cutting concern, but do not distinguish between homogeneous and heterogeneous nature of such concerns. We hypothesize that they do not encounter heterogeneous cross-cutting concerns due to their application scenario, which focuses on scientific workflows. We assume that the processes and artifacts here are the dominant decomposition criterion. As a result, they are reflected in classes and methods allowing to easily bind pointcuts to these program structure. Consequently, aspect-oriented programming is the best implementation technique, similar as in our first case studies. However, in [BFH14] the authors do not focus on offering tailored provenance capturing and storing capability, again due to their application scenario.

In summary, the CAPS framework strengthens the motivation, objectives, and results as well as it highlights the importance of the overall topic. From, our point of view it is not a competitive, but highly related approach due to its slightly different focus.

8.2 Fine-grained provenance for databases

In this section, we discuss what additional insights are expected upon integration of the provenance concern into a (mature) database system. Moreover, we justify why such integration is required in context of the Digi-Dak project with the help of a motivating scenario. Finally, we design an extension of our provenance API and discuss in how far the single features interact with the query processing including the intended level of invasiveness.

8.2.1 Expected insights

In the prior section, an analysis of the source code of the case study reveals that the major criterion to decompose the tool remains the intended functionality. Therefore, it is congruent to the provenance features to integrate. Hence, we can easily exploit the object-oriented decomposition to integrate the Dendrites. This results in low manual implementation effort using either feature-oriented or aspect-oriented programming. To this end, we select as final case study in this thesis, a complex system for that we assume that the primary decomposition is not reflected or at least hidden in the object-oriented decomposition.

Database systems have the advantage that the offered functionality is clearly stated in the relation algebra and that they use a limited, but powerful operator set. That means, most queries use a well-defined set of operators (e.g., projection and selection)

[3] CAPS - Capturing and Archiving Provenance in Scientific workflows.

with known semantics. Moreover, for the relational data model, there are currently the most fine-grained provenance approaches defined. Finally, as performance in databases is of special interest, the objective of minimal invasiveness regarding non-functional properties is clearly motivated. To this end, we integrate provenance into databases in the remainder of this chapter. The intended insights in particular thereby are:

1. Explore whether integration effort remains feasible in practice.
2. Determine the effects on non-functional properties regarding the single features. Moreover, we are interested in differences between the implementation techniques, as prior case studies suggest that there are measurable differences. This reflects one criterion of minimal invasiveness.
3. Investigate whether we can re-use (parts) of the Soma implementation and whether we can automatically integrate Dendrite into different database systems. This is intended to show the comprehensiveness of our approach.

Before we start integrating the provenance functionality into an existing and mature database system, we first have to motivate the practical relevance of provenance integration in the context of the Digi-Dak project. This motivation, secondly, allows us to extend the feature model of our provenance API defining what to integrate in the sequel.

8.2.2 Motivating scenario

To explain the necessity to integrate provenance into databases in our infrastructure, we again use a scenario from the Digi-Dak context. This forms the motivation for the final case study of this thesis. An important step in the overall infrastructure is to automatically locate areas on an evidence that potentially contain a fingerprint (called regions of interest). This location is based on a coarse-grained scan, which can be produced fast and does not contain details of the fingerprint pattern. The purpose of a region of interest is to define areas that shall be scanned in more detail to extract details of the fingerprint pattern. These detailed scans are time consuming (up to several hours for one scan), thus we only want to scan regions of interest that contain finger prints with a certain probability. Furthermore, we must not exclude forensic evidences and therefore have to determine the error rates of our algorithms with a manual review.

Now we may use different sensors or one sensor may produce different results that have to be *combined*. For instance, our default scanner creates an intensity image (based on amount of reflected light) and a topography map (contains height information). In Figure 8.6, we depict an evidence that contains four anomalies. Our region of interest identification algorithm looks for something that is significantly different from the rest of the surface having the right size. Dependent on the input data, this algorithm has benefits and drawbacks regarding to certain anomalies. For instance, considering Figure 8.6 the intensity image, the scratches are detected to be a possible fingerprint. By contrast, in

the topography image the algorithm detects that the scratches are below the surface and thus cannot be a fingerprint. Meanwhile, it is possible that dirt is classified to be a potential finger print based on topography information, but not in the intensity image since it is a very homogeneous area. Finally, the low-quality print has not been located at all, which is an error and important for determining the error rates.

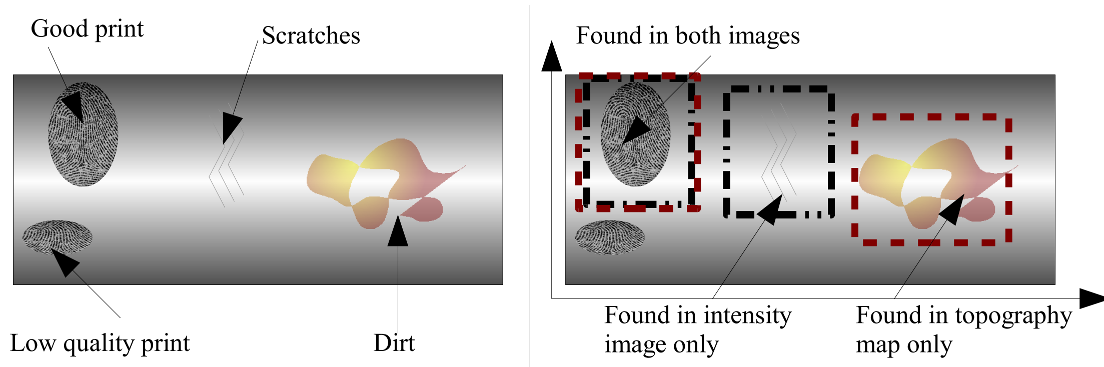


Figure 8.6: Evidence containing four potential regions of interest

The regions of interest example

For our final case study, we apply the *regions of interest* example. Specifically, we want to modify existing database systems in a way that we can, based on the variant of our provenance API, gather either (1) lineage or why provenance, changes of aggregated values as well as view deletions.

For the regions of interest example, we have several (independent) databases in our distributed infrastructure, in some databases we only need to know which tuples were used to create a result tuple, thus we need *lineage*. For, evaluation purposes, we may temporarily want more fine-grained forms of provenance, e.g. error propagation, or probability annotations. Furthermore, local copies may have a different provenance requirement. Summarily, we need variants of our provenance API able to collect different forms of fine-grained provenance.

8.2.3 Feature model and resulting changes

As Cheney et al. state [CCT09], detailed forms of provenance are instances of each other. Lineage is contained in Why provenance, and Why provenance is itself contained in How provenance. In the same sense, recent extensions for aggregate queries or the pig latin language are based on How provenance as well. Finally, in Chapter 2 we have shown that many of the forms of provenance (including coarse-grained forms) known, have similarities used to unify them in a conceptual framework. As a result, we can extend *our provenance API* with additional functionality that is easily integrated in case we need a different form of provenance.

In Figure 8.7.(a), we depict the feature model of an extension for our provenance API based on a domain analysis performed in Chapter 2. The basic proposition is that we

modeled the more detailed forms as extensions of each other. The reason is that we want to able to build a variant of our provenance API in a modular way by adding/replacing only the necessary parts overtaken from the first exploratory case study on client side. Nevertheless, in Figure 8.7(b)-(e) we reveal that, based on the variant of provenance API, we change tuple processing of our database systems. Therefore, we mark the applied change in the query processing with the color of the feature. Tuple processing, however, is a performance critical part. Thus, we do not want to include unnecessary modifications into a database system to capture the provenance or change the behavior of query processing. On the other hand, we want to provide a general solution that allows for easy integration into arbitrary database systems. Consequently, the major research question for this case study is: *What software engineering techniques allow that we integrate the desired provenance form into existing database systems efficiently?* Here efficiently means that we have to consider the trade-off between integration effort, computation overhead as well as maintenance criteria.

Deletion propagation for materialized views

So far, all provenance functionality works retrospective. That means that we only collect provenance, but do not actively use collected provenance to compute new results. To consider these kinds of features as well, we add the materialized view feature. In particular, we want to integrate deletion propagation for materialized views.

We decided for this feature, because deletion propagation for materialized views is s well-known problem in databases that can be addressed using provenance [BKT02, CW01]. Moreover, the intended regulative change in the query processing is complex and thus, well suited. Finally, for instance, in case we detect that a sensor delivered erroneous results, we need to know which intermediate artifacts are affected.

In the sequel, we use this feature model to first implement the Soma functionality and then, integrate the Dendrites to a mature database system.

8.3 Provenance integration for databases

In this section, we integrate the priorly designed extension of our provenance API into a database system and evaluate the effects regarding our objective of minimal-invasive provenance integration. Then, we explore in how far we can port the extension to different database systems. To this end, we firstly select a suitable database system, secondly integrate and evaluate the intended functionality, and finally try to generalize our implementation for different database systems.

8.3.1 Case study selection

Now we select the database system into that we integrate our provenance functionality. Therefore, we have two selection criteria. We need:

1. A mature database system containing real-world challenges. Consequently, we need a system that is used productively in large scale.

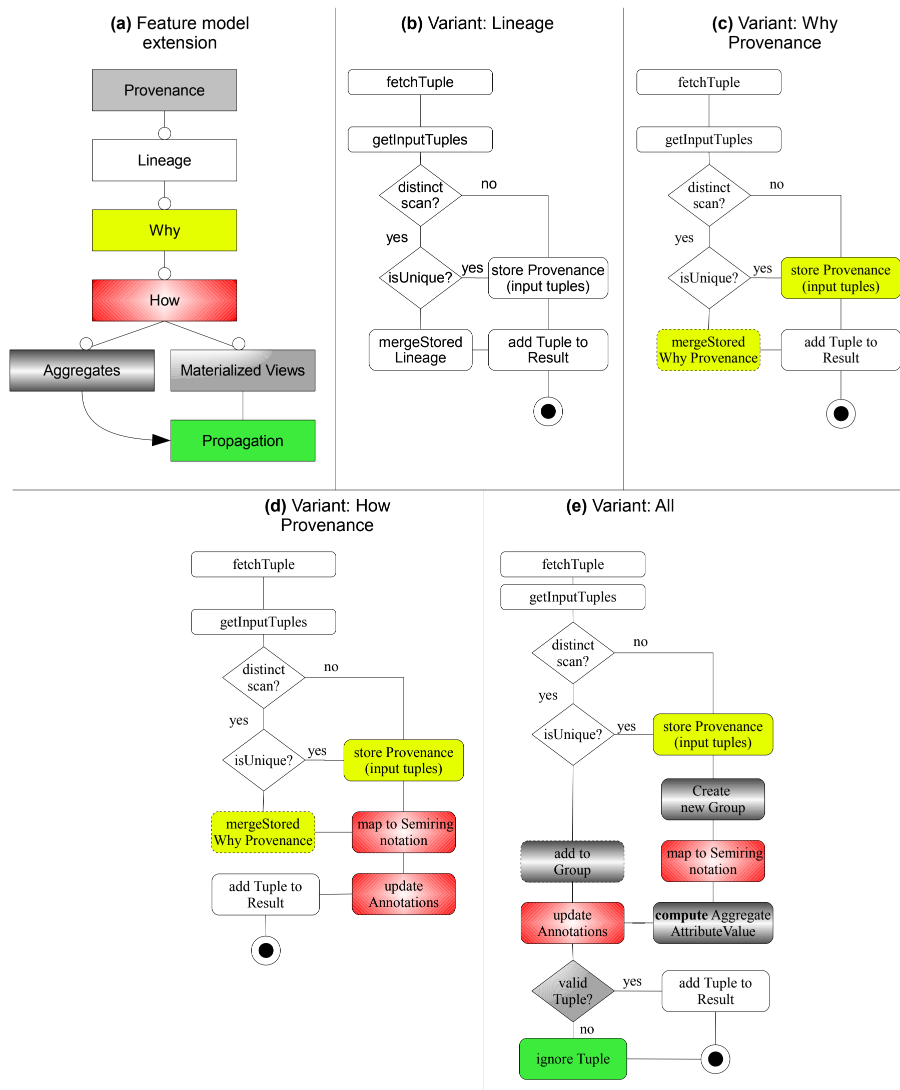


Figure 8.7: Extensions for the provenance API

2. As our implementation techniques are not available for all programming languages, which includes proper tool support, we have to limit ourselves to languages that have respective tool support. Currently, we found the best support for Java. As Java is one of the most frequently used programming languages this does not impose limitation with respect to the intended results.

Priorly in this thesis, we applied PostgreSQL. However, as this database system is written in C, we cannot use our implementation techniques. We decide to use HyperSQL[4] for the following reasons. First, it is an open-source database system entirely implemented in Java, which is, amongst others, used in Open Office. Consequently, we consider this

[4]Project website: http://hsqldb.org/

database system as mature. In addition, this system has been used by other authors to show that newly developed concepts are feasible for real-world application, such as in [PKC+13]. We conduct our case study for version 2.2.5, which was the current version when we started implementation. Finally, as the source code for prior and later versions is available, we can test whether our implementation can be integrated into these versions as well. This is a *first* means to investigate the generality of our solution.

8.3.2 Exploratory implementation

As a first step to evaluate the effects of different implementation techniques, we integrate the provenance functionality using our intuitive techniques. When implementing this case study, we looked for the desired functionality and documented changes and pitfalls.

As we assume that a major challenge is to locate the source-code locations where we need to integrate the provenance Dendrites, we looked for the locations in a structured way. The basic idea is to locate the implementation of the single relational operators, handling of intermediate result, and additional functionality like duplicate elimination and sorting. To this end, we started sending simple queries containing selection and projection only to the HyperSQL database system. We added a breakpoint to the execute methods of the JDBC driver used to send the query. Then, we used the debug functionality of the Eclipse IDE navigating via single step execution until we reached the source code implementing the desired functionality. Then, we extended the complexity of the query and repeated the procedure until we found all source code locations we were looking for. The advantage of this procedure is that it first can be used for every database system and delivers a valid starting point. Altogether, we needed three days to integrate the desired functionality not including developing the Soma parts of the provenance API. We argue that this amount of time is no counterargument regarding the practicability of our overall approach.

Code scattering and heterogeneity

Generally, there is series of very small modifications (cf. Table 8.1) that are scattered over the whole source code of HyperSQL. For instance, most modifications require three lines of code. Only in the `buildResult` method there are plenty of modifications. Nevertheless, 6 out of 13 changes require structural modification of the HyperSQL source code that are not totally removed when building the binary. For us, especially the code scattering and code obfuscation (due to if-conditions) when debugging non-trivial behavior was a major challenge. In contrast, to prior case studies provenance is no longer a homogeneous, but a *heterogeneous* concern as we have to integrate different code for the single relational operators.

Additional system modifications

One major problem in HyperSQL are non-persistent and non-accessible `RowIDs`. There is a `RowID` that is encoded as long value having 64bit. The first 32 bit integer contains the `TableID` and the second one the row number. However, the `TableID` is not persistent,

Feature	Class	Method	Line count	#Changes	Line inc.	Premise/Purpose	Structure change
Study 1 (JDBC)	JDBCStatement	fetchResult	39	1	5	Study 1	no
	JDBCResultSet	getFetchSize	7	1	5	Bug	yes
	Table	getID	5	1	2	persistent ID's	yes
Lineage	Query-Specification	buildResult	231	3	23	Tuple creation	1 of 3
		getSingleResult	24	1	3	Commit	no
	JDBCStatement	execute	9	1	5	Vizualization	no
	SubQuery	materialize	31	1	3	Sub query	no
	ArraySort	moveAndInsertRow	10	1	3	ID's & sort	no
		quickSort	55	3	9	ID's & sort	no
	RowSet-Navigator	removeDuplicates	33	1	4	Duplicate	yes
		union	27	1	3	Dupli. & sub query	no
		unionAll	16	1	3	Sub query	no
	ResultMetaData	addProvenance	20	-	17	Sub query	yes
Why	ResultMetaData	addProvenance	20	1	3	Sub query	no
Where	QuerySpecification	buildResult	s.o.	3	3	Copy origin	no
Mat views	StatementSchema	getResult	1067	1	3	Save provenance	no
Σ	9	14	1594	21	94		6

Table 8.1: Necessary modifications in HyperSQL per feature using

because the `TableID` is assigned when a running instance of a HyperSQL server loads a table space and is used to organize the table of running instance. As a result, we had to implement additional functionality to provide persistent identifiers. Furthermore, the row number can only be used for identification as long as the table is not re-organized (i.e., sorted). Unfortunately, sorting is used to determine duplicates and can be issued by a user. Consequently, we need to implement functionality to map tuples to new `RowIDs` after sorting, which was not intended and is based on implementation details of HyperSQL.

A challenging task is to integrate materialized views with deletion propagation. HyperSQL does not support materialized views in version 2.2.5. Thus, we first have to integrate this functionality and also integrate the deletion propagation. The idea is to use normal tables and annotate these tables as materialized views. The basic advantage is that these tables act like normal tables (e.g., regarding transactions). However, in case they are queried, we extend the evaluation of the selection predicate s where s is a propositional formula for each considered tuple. In fact, we add a term p, where p results from the evaluation of the provenance data and indicates whether this tuple has been deleted. In detail, p is `true` in case there remains at least one path in Why provenance creating the current tuple otherwise it is `false`. A tuple only is result of a query if the evaluation of the selection predicate *and* the provenance predicate is true. Therefore, the new selection predicate s' for any tuple is computed as $s' = (s \cap p)$ ensuring this property. Note, in case there is no selection predicate, it is set to `true` resulting in $s' = (1 \cap p) = p$. In addition, we have to extend the commit functionality of HyperSQL to ensure that the *fine-grained* provenance data are only committed in case the query is committed. Otherwise the provenance data is only send to the tool along with the query result. As a result, the provenance data are only saved persistently

in case the query is committed and in turn the database system ensures in case the modification is committed the provenance data is saved persistently as well.

Correctness of the integration

Although most changes are by concept spectative (with exception of the view deletion propagation), we need to validate the integrated functionality. This includes:

1. That we do not inflict any undesired modification to the tuple processing of the database system.
2. That the provenance data are gathered and stored correctly.

Generally, we validate the integration via *testing* using specific test cases. For both above mentioned properties, we use a test database with the schema of the TPC-H benchmark [TPC-H10][5]. The TPC-H benchmark is a well-known benchmark to evaluate the database system performance for complex queries, which suits best to our workloads. We validated that the processing functionality delivers correct results by testing first the single operators including distinct scans and sorting and compared the results to the expected ones of the original database system. In addition, we compared the results of the TPC-H queries showing that these results are correct. To this end, we conclude that our integration does not impose an unintended change to the query processing with certain, but small, residential risk of undetected errors.

We validate capturing and storing of provenance data in a very similar way. First, we test the single operators and compared them to manually calculate results and then compare the resulting provenance data of the TPC-H queries. Subsequently, we query the stored the provenance data again to test whether they are stored correctly. Finally, we create materialized views and delete exemplary data in the original tables to test whether deletion propagation works correctly. This includes also that transitive deletion works. That is that a materialized view is created from another one which then is based on normal tables.

In summary, the tests indicate that our provenance integration *is correct*. Therefore, there is no (unintended) functional modification except the desired ones and thus, the integration is conform to our goal of minimal-invasive provenance integration regarding functional properties.

8.3.3 Evaluation of non-functional properties

In the following, we examine the impact on performance and main-memory consumption for different provenance features and queries. We compare them to the measurements of the original system. This allows us to draw a first conclusion on the impact of the integration in general. Moreover, the gathered data allows us to compare the effects of

[5]Project website: http://www.tpc.org/tpch/

different implementation techniques in the remainder. The query selection is similar to the tests for functional correctness. We first measure the effects for simple queries and then for more complex ones. All subsequently presented measurements are robust mean values.

Evaluation results

In the following, we evaluate the provenance integration with respect to non-functional properties based on the priorly introduced evaluation concept. To this end, we define several queries, to examine the effects of the operations separately (Q1 to Q6). In Table 8.2, we depict the queries and note our intention for the design of the query. For instance, Query 2 uses a distinct projection, while Query 4 is designed for evaluation of `union all` operator, which does not remove duplicates after the union of two relations.

Table 8.2: Evaluation queries

Query	SQL Command	Operation
Q1	`Select * from lineitem;`	Simple query
Q2	`Select distinct L_PARTKEY from lineitem;`	Distinct scan
Q3	`Select L_ORDERKEY from lineitem where L_SHIPDATE > $value;`	Selection
Q4	`Select L_ORDERKEY from lineitem where L_SHIPDATE > $value union all Select L_ORDERKEY from lineitem where L_SHIPDATE <= $value;`	Union without duplicate elimination
Q5	`Select sum(L_QUANTITY) from lineitem group by L_PARTKEY;`	Aggregation
Q6	`Select * from lineitem, orders where O_ORDERKEY = L_ORDERKEY;`	Join

In Table 8.3, we depict our measurements regarding the average response time and main-memory consumption for all queries. Generally, the table contains on the left side the values for the original system. This means, for instance that, in the original system, Query 1 required on average 23 ms to execute. Then, we depict the values with Lineage (Lin), Why provenance (Why), and Where provenance (Where). As we could not observe a significant difference between both intuitive techniques, we only depict one result.

Table 8.3: Non-functional properties per feature and query

	Response time in ms				RAM in MB			
	Original	Lin	Why	Where	Original	Lin	Why	Where
Server start	516	703	719	724	70	119	119	110
Q1	23	31	32	34	110	208	195	209
Q2	60	2390	3188	3221	73	161	164	178
Q3	18	31	24	26	72	157	168	163
Q4	36	46	60	56	73	149	159	165
Q5	59	204	154	143	73	155	169	170
Q6	60	196	157	133	70	158	154	174

Response time

Regarding response time, we observe that performance penalty due to the provenance functionality depends more on the included operators than on the provenance feature

(i.e., granularity level). For operations that do not require duplicate detection, our data reveals a performance overhead of less than factor two. We argue that this is a very good result. However, for queries Q5 and Q6 the observed increase is between factor two and three, which still seems manageable. For Q2, however, we encounter an increase of factor 30. An in depth analysis reveals that this effect is caused by an interaction with an implementation detail of the database system. To execute a distinct scan, the result table is sorted and then duplicates are *removed* from the result. Therefore, after sorting our initial provenance data shows that the result table contains all tuples including duplicates. In case, a duplicate is removed from the result, we have to remove this line in the provenance data *and* copy the provenance data to the identic tuple that is not removed. This causes a lot of additional work. However, the major problem is that our data structures in the Soma are designed for fast insert not for removing, resulting in large reorganization effort. We tested different implementations that result in an additional performance penalty for other queries. To this end, we have to accept the problem with distinct scans and suggest a refactoring of the original system. Similar to the computation of groups for aggregations, duplicates should not be removed, but after sorting new tuples should be copied into a different result table.

Main-memory consumption

Considering the required main memory in Table 8.3, we observe an overhead of factor two to three. Similar to our response time observations, there hardly is a difference between the different provenance features showing the efficiency. This is mainly due to the fact that we do not store different data but just change the way how to interpret the stored data.

8.3.4 Alternative implementation techniques

We repeat the initial provenance integration using feature-oriented programming with FeatureHouse, aspect-oriented programming with AspectJ and CIDE. Transferring our initial implementation using conditional statements to CIDE is fast and simple, as we only have to delete the introduced `if` statements and mark the remaining source code. Moreover, we are able to remove all permanent additional modifications that we had to introduce using the intuitive techniques. To this end, we conclude that this technique is less invasive.

Comparing the integration effort of all techniques there is little difference among them. Basically, the main effort is finding the source-code location where to integrate the Dendrites. Using feature-oriented programming, we have to integrate additional hook methods, while in AOP we could avoid most of these hook methods. Usually, there is a method that can be instrumented in order to capture provenance, but this method is called from multiple locations and not in every case we want to capture provenance. To specify the source locations, we can use the `within` and `withincode` keywords in the pointcut definition, which really helped us to avoid unnecessary hooks. However, in AOP we have to integrate hooks as well. In addition, we have to change the visibility of class

members (or add additional `get()` methods). Finally, AOP resulted in hard to explain errors especially with respect to aspect execution in or around synchronized methods requiring redefinition and moving of respective pointcuts. Therefore, we consider AOP and FOP techniques more invasive than the CIDE approach, but less invasive than the intuitive techniques as we have a smaller number of permanent code modifications. For all integration, we successfully ran the test queries. Thus, we conclude that the integration is successful.

Effects on non-functional properties

In the following, we explore the effects on non-functional properties. Based on our prior observations, we expect that we cannot observe a significant difference between the techniques except for AOP. In particular, we are interested whether we can measure an additional implementation-technique specific performance overhead for AOP as it is suggested by prior case studies. For main-memory consumption, we also expect an overhead using AOP.

Initial experiments with unmodified variants of the database systems (i.e., a variant where no provenance feature is selected) for each technique, resulted in the same average values as for the original system. Therefore, we conclude that there is no provenance functionality related overhead, which is one goal of minimal-invasive provenance integration and also indicates that our subsequent measurements are correct.

Response time. As expected, we cannot measure a significant performance overhead for FOP and CIDE compared to the intuitive techniques. However, we expected a significant penalty for AOP due to the large number of reflective Dendrite calls, but we could not measure such penalty. In fact, most of the additional time is consumed in storing and merging the provenance data in the Soma or when either persistently storing the data or integrating it into the query result. In addition, we assume that Java's run-time optimization may remove some performance bottlenecks.

Main-memory consumption. Our measurements reveal that our hypothesis that there is no significant difference between the required main-memory is correct. For any provenance augmented variant, we measure a higher initial main-memory consumption upon server start. After starting the database the original database and the unmodified variant using AOP the server required 120MB (cf. Table 8.3), while all other implementation techniques require about 70 MB. This is due to an implementation detail of FeatureIDE and AspectJ resulting in the effect that the provenance API is even loaded in case it is not part of the variant. However, we cannot observe additional memory consumption for provenance augmented variants.

In summary, our observations result in two conclusions. First, we state that aspect-oriented programming and the intuitive techniques are unsuitable for provenance integration in databases - mainly due to permanent un-removable source source-code

modifications and runtime overhead. Second, we cannot give a clear recommendation whether feature-oriented programming or preprocessors should be used as they are very similar regarding the criteria of minimal-invasive provenance integration. To this end, we determine in the remainder, whether we can re-use Dendrites of FOP for seamless integration or at least to speed-up integration provenance integration into (newer) versions of HyperSQL and other java-based database systems.

8.4 Adaption for different database systems

In the following, we determine whether our Soma and Dendrites are database-system specific or can be re-used for different database systems. In particular, we want to know whether our solution represents a, to some extend, general solution or we have to develop completely new provenance capturing solutions for every database system. To this end, we first review our implementation to find commonalities that we expect to find in all or most database systems. Then, we integrate the exiting functionality into newer versions of HyperSQL and a different java-based database system.

8.4.1 Premises for generalization

To generalize our results and allow easy integration into different database systems, we need requirements that we premise for provenance integration and thus, need to be offered by the database system. Consequently, the basic idea is that the database system already provides all the functionality we need for its own query processing. Furthermore, the assumption is there are only a few but powerful relational operators (e.g., projections etc.) and respective well-known algorithms. Thus, there also is a limited set of functionalities the provenance API has to offer. Note that the amount of system-dependent additional required functionality especially regarding the Dendrites is one issue addressed in the remainder.

Identifiers. Persistent and accessible unique tuple and table identifiers (`RowID` and `TableID`): We store only tuple identifiers and not the tuples itself (for privacy and performance reasons). Hence, we need a possibility to identify tuples in a database by a unique identifier that should also include the unique `TableId`. In HyperSQL, we did not have such persistent identifiers and thus needed to integrate them.

Tuple creation. When creating a new tuple, we need the `RowId` r of the new tuple (r_{output}) and a list L of tuple(s) used to create this tuple ($r_{output}, L(r_{input})$). Additionally, we use a simplification introduced by a difference between the implementation of relational database systems and the relational algebra. Generally, the relational algebra uses *set semantic*s. In contrast in SQL, we have to specify distinct scans explicitly or use set operation (e.g., UNION). Hence, there is no possibility to have multiple paths creating one result tuple (the duplicates are also in the result). As a result, this premise covers *all basic non-distinct relational operators*: Selection, Projections, Crossproducts, and Joins. Finally, this allows us to benefit from optimization, such as performing projections and selections at the same time.

Duplicate elimination. In case a user specifies a distinct scan, our implementation requires a function provided by the database systems to identify the duplicates by their `RowID`. Then, our provenance API merges the provenance information as follows. For two tuples (r_1, L_1) and (r_2, L_2) identified as duplicates provenance API creates $(r_1, Merge(L_1, L_2))$, where $Merge$ depends on the form of provenance.

Groupings & aggregates. For groupings we need a similar functionality as for duplicate elimination. A tuple either adds a new group to a result or is added to an existing group. Additionally, we need a possibility to access the attribute value and information about the aggregate function to document the computation formula, necessary to propagate updates or deletes for materialized views.

Sub-queries. In case sub-queries are materialized (e.g., for inner queries or set operations such as Union), we need a functionality indicating that independent results are merged. Note, dependent on the implementation, we can re-use the duplicate elimination functionality.

These requirements are in fact Soma functionalities offered by our provenance API. To this end, there are methods, which can be called in the Soma that compute and store respective provenance. Consequently, we hypothesize that our Soma offers general provenance capturing functionality. However, we can currently not hypothesize in how far we can re-use (parts of) the Dendrites for HyperSQL. Thus, we explore their re-usability in the sequel.

8.4.2 Versions of HyperSQL

In the following, we explore whether we can re-use our Dendrites (and Soma) implementation for newer versions of HyperSQL. In the previous section, we could not give a clear recommendation whether preprocessors, such as CIDE, or feature-oriented programming shall be used. The following exploration intends to answer this question. In case, we can re-use our Dendrite implementation, we clearly recommend feature-oriented programming due to less integration effort. In case we cannot re-use the Dendrites and have to adapt the Dendrites, we suggest CIDE. To this end, our integration procedure is as follows.

1. Create a new FeatureIDE project using the FeatureHouse composer.
2. Copy feature model and all feature implementations except for the Base feature.
3. Copy the implementation of the newer HyperSQL version (e.g., 2.2.6) into the feature Base implementation module (folder).
4. Check for build errors and correct them if present.
5. Execute the correctness tests from Section 8.3.2.

Note, we use two versions for our Dendrite integration. The first one minimizes manual integration effort by overwriting whole methods in case hook methods would be required. The second one requires for manual hook integration before we are able to generate variants.

Table 8.4: Dendrite and Soma reusabilty for HyperSQL versions

		Dendrites reusable		Soma reusable
HyperSQL Version	Released	Method override	Manual hook integration	
Version 2.2.5	06.07.11			
Version 2.2.6	20.11.11	yes	yes	yes
Version 2.2.7	15.01.12	yes	yes	yes
Version 2.2.8	22.01.12	1 build error	1 build error	yes
Version 2.2.9	06.08.12	2 build errors	yes*	yes
Version 2.3.0	08.07.13	no	yes*	yes
Version 2.3.1	08.10.13	no	yes*	yes
Version 2.3.2	14.02.14	no	yes*	yes

yes* with changes for 2.2.8

In Table 8.4, we depict our results regarding the re-usability of the Soma and Dendrite implementation. Recapitulate that we perform the initial integration for version 2.2.5. Thus, the table only contains the release date. We emphasize that the version history covers a time span of two and a half year and a major release (switch from 2.2 to 2.3). Therefore, we argue that our results contain interesting and valid results regarding the applicability of our approach considering software evolution.

Generally, the integration of the Dendrites is easily manageable. Especially for the releases before version 2.2.8, we can simply integrate the Dendrites by overwriting the methods, if refinements are possible (and hooks are required). In these cases, we just copy the implementation into the feature implementation modules without any manual modification. For version 2.2.8, we have to change the method signature of the `QuerySpecification.buildResult()` method, so that the correct method is overridden. Since HyperSQL version 2.2.9, there are bigger changes affecting the provenance integration. For instance, not anymore existing members in the `buildResult()` method or deletion of the `SubQuery` class results in additional modification effort of our Dendrite implementations. However, it does not affect the Soma, as intermediate sub query handling (as intermediate results) works the same way. Nevertheless, adapting the implementation is a nasty task and manually integrating the hook methods was faster and less error prone for our purposes. An interesting observation for our purposes is that the source code close to the Dendrites was not changed at all with exception of the deleted `SubQuery` class.

The purpose of these experiments is to evaluate whether there is a clear benefit using feature-oriented programming. The hypothesis is that due to re-usability of the Dendrites we have less integration effort. Our results clearly indicate that there is a significant benefit until (including) version 2.2.8. This covers a time span of half a year. For any later version, we have to manually adapt the source code. As final comparison, we

manually integrated the provenance functionality into the latest HyperSQL version with preprocessor annotation. The idea is to compare the manual development effort to the adaption of the automatic ones. This integration (including test) required less than an hour. For the feature-oriented programming the time is very similar, mostly to tool related waiting times. For instance, in case we integrate a hook, FeatureIDE issues a full build of the project to check all resulting variants for correctness. This requires about a minute.

Based on the presented information, we make the following conclusions. In the short term, we suggest FOP. However, in the long run and due to performance optimization potentials, we prefer preprocessor based solutions.

8.4.3 H2 Database

To examine the generality of our solution, we integrated the provenance API into an additional java-based database system H2[6]. H2 is a database comparable in complexity, offered functionality, and performance to HyperSQL[7]. To this end, we consider this database system as an ideal case study to examine, which parts of our prior implementations for HyperSQL can be re-used. Based on our previous explanations, we assume that the Soma implementation can be re-used completely and already contains all required functionality. In addition, we are interested into integration effort, especially the location the respective source-code locations and whether the Dendrites are similar to those of HyperSQL. On a more abstract level, we want to ensure that our observations are not specific to or based on limitations of HyperSQL.

Integration effort. Altogether, locating the respective source-code location where we need to integrate the Dendrites took less than a day. This includes setting up the database system and running it from source code. In particular, the procedure to locate the source-code locations is the same as for HyperSQL. We use a client application and send a simple select query to the database server and then start investigation from the `execute` methods of the JDBC driver. For H2, the integration is even easier than for HyperSQL. The reason is that the object-oriented decomposition of H2 corresponds more to the actual functions performed by the database system (i.e., the relational operators). We assume that this difference is either due to performance reasons or an effect of the evolution of HyperSQL. For instance, there are several classes in the package `org.h2.command.dml` that first by naming convention correspond to relational operators and second extend from and abstract `Query` class. To this end, we can easily detect all classes *and* the methods where we need to integrate the Dendrites. In summary, we argue that integration effort is feasible in practice and therefore not in opposition to our goal of minimal-invasive provenance integration.

[6]H2 project website: http://www.h2database.com/html/main.html
[7]Comparison of H2, HyperSQL, and PostgreSQL: http://www.h2database.com/html/features.html

Soma re-usability. When integrating the provenance concern, we observed that, as we expected, the Soma contained the required functionality with one exception. Therefore, we had to perform only little additional implementation overhead for duplicate elimination for distinct scans. In HyperSQL, duplicate elimination means sorting a relation and subsequent removing of duplicates. By contrast, H2 uses hash buckets containing duplicates (without prior sorting). To this end, we have to extend our Soma with one additional method, using the small hashing methods to group duplicates in the same hash bucket and add the desired provenance data. In fact, H2 functionality corresponds even more to the data structures in our Soma. This, results in a smaller relative performance penalty (factor 3) than for HyperSQL. As we argue there are only few algorithms for each relational operator that are different from point of view of provenance capturing, we consider our Soma as general provenance API for database systems. It allows for tailored provenance integration, which can be easily extended if necessary.

Dendrite similarity. Similar to HyperSQL, the Dendrites are scattered over several files and we cannot simply wrap around methods to extract the desired provenance data. Mostly, the provenance data is within `for loops` looping over intermediate results or test selection predicates. To this end, we have to either directly insert code using preprocessors for tailoring purposes or inject hook methods. In addition, we observe similar to HyperSQL that provenance on this granularity level is no longer a homogeneous concern, but heterogeneous mostly due to the different semantics of the relation operators and specifics of their implementation.

Summary

In summary, we conclude that our results from HyperSQL and H2 are very similar and to this end not artificial, but generalizable. The major result of these cases studies is that we can integrate the provenance concern in a minimal-invasive way, although its nature changes from homogeneous to a heterogeneous cross-cutting concern, which is hardly congruent to the object oriented decomposition. Consequently, our suggestion of using preprocessor remains best suiting our goal of minimal invasiveness remains. As a result, we conclude that current object-oriented decomposition is hardly sufficient to separate the concerns clearly.

8.5 Contributions and conclusions

In the preceding two chapters, we explore how to integrate provenance in a minimal-invasive manner for different case studies. In general, we explore whether using software product lines in order to integrate a totally new variability dimension into an existing monolithic data-intensive system is possible and in how far we can observe differences between different well-known implementation techniques. In addition, we are interested into properties of our case studies that either ease or harden provenance integration to formulate rules and guidelines for engineers designing new solutions. Therefore, the

case studies have different complexity and different provenance requirements to address the challenge of provenance integration in a comprehensive manner. In this chapter, we integrate fine-grained provenance into complex systems resulting in the following contributions:

1. We evaluate in how far our concept to exploit the object-oriented decomposition of programs [SSS12b] can be applied for tool chains in the Digi-Dak infrastructure. The basic purpose of our concept is to speed-up provenance integration using AOP or FOP. Using this concept delivers good results if certain properties are avoided and a small performance overhead [KS13b].

2. We design, implement, integrate, and evaluate a variable provenance solution for java-based database systems. In addition, we show that the solution can be easily integrated into different database systems exploiting the limited number of (relational) operators and respective algorithms. Thus, it can be easily integrated in additional database systems.

3. The conduction of case studies reveals several benefits and pitfalls in the architecture of programs, which can serve as recommendation for developers when designing new systems. We assemble the complete list of recommendations in the appendix of this thesis as they are scattered over the case study conduction in the preceding chapters.

The primary goal of this thesis is to explore whether we can integrate a provenance concern into existing monolithic systems efficiently. We call this integration minimal invasive. As provenance is itself a highly versatile concern and different systems have different requirements regarding provenance, the key idea is to treat the provenance concern and its integration as a software product line. This way we are able to automatically generate tailored provenance capturing solutions. These solutions, however, need to be integrated efficiently. Therefore, another idea is to use well-known implementation techniques from SPL development and compare them to two intuitive integration techniques. Generally, we observe that minimal-invasive provenance integration is possible exploiting homogeneity within these systems, such as global communication infrastructure or object-oriented decomposition. According to our results it is practically feasible considering integration effort and effects on non-functional properties to integrate provenance into existing systems efficiently. Nevertheless, we reveal that integration effort and performance penalties depend more on properties of the systems itself than on the applied implementation techniques. To this end, we document these properties for each case study.

In summary, based on the results of our empiric research method and proper selection of the applied cases studies and their evaluation, we consider the primary goal of this thesis as successfully addressed.

9. Conclusion and future work

Even integration of a monolithic additional concern into an existing monolithic system often results in severe issues regarding functional correctness and non-interference with non-functional properties. To this end, integrating a whole variability dimension in a minimal-invasive manner is even more challenging. However, our results indicate that due to the versatile nature of provenance and different user requirements (that often change over time) there cannot be an ones-size-fits it all solution, but we need to generate tailored provenance capturing solutions, based on a selection of required features. Thus, it is required to enhance existing systems and infrastructures with provenance support. However, so far, most of the research in the domain of provenance resulted in solutions and approaches that cannot be flexibly adapted to changing requirements. To this end, this thesis contributes knowledge on software product lines. In particular, in how far SPLs can be used to integrate tailored provenance as a cross-cutting concern into an existing system. We explore how to efficiently integrate this variable concern based upon the term minimal-invasive integration. We borrow this term from surgical procedures and define it in the context of computer science.

The research in this thesis is based upon a series of case studies that explore how to integrate tailored provenance capturing and storing capability. Therefore, we mainly rely on case studies or application scenarios from a real-world large-scale joined research project with partners from industry, academia, and law enforcement agencies. In addition, we consider object-oriented programs as well as (object-) relational databases. To this end, we ensure that our findings are sound and generalizable. In addition, this research project allows us to claim that this thesis contains valuable contributions to real-world challenges and academic questions, which is for instance shown by respective publications.

The primary result of this thesis is that minimal-invasive integration of provenance is practically possible as long as we can exploit certain homogeneity in the system(s), such as communication infrastructures etc. This includes modeling the provenance concern

in terms of features, minimizing undesired feature interactions due the nature of the provenance concerns, as well as implementing the feature content. In addition, software product line implementation techniques outperform intuitive implementation techniques and are, to this end, very well suited to integrate the additional features. In particular, we made several contributions to reach the goal of this thesis. The most prominent are:

1. Defining the term *minimal invasiveness* for provenance and adapting the neuron analogy for a universe of provenance.
2. Introducing the *database-centric chain-of-custody* as concept that can be ported to arbitrary communication-centric infrastructures.
3. Design and show the practical relevance of our tailoring approach for relational database schemas.
4. Propose and show the applicability and limitations of an approach to exploit object-oriented decomposition in order to enhance programs with tailored provenance support.
5. Define a general provenance API based on our provenance framework, which can be easily integrated into relational database systems.
6. Introduce *QuEval* to support minimal-invasive provenance integration.

On a more general level, we discover that provenance in general is a cross-cutting concern. In addition, the source code locations where to integrate provenance are scattered over the source of most of our case studies due non-congruency with the dominant decomposition in classes and methods. On the other hand, we reveal that fine-grained forms of provenance require heterogeneous source code while for coarse-grained we can exploit the homogeneous nature of the provenance concern to ease provenance integration. Both named properties offer interesting directions for future work.

Future work

In this thesis, we identified several points that offer potential for future work.

A request for research on multi product lines

One of the propositions of this thesis is that we integrate provenance into monolithic systems. However, for future work, we suppose integrating provenance into systems that are itself variable results into interesting findings. We hypothesize that, due to the decomposition into features, finding the source-code locations is less time consuming than for monolithic systems. However, as variability in the original program contains the challenge of how to ensure type safety, we are interested in the resulting trade-off of between less integration effort due to easier location of the source code where to integrate the desired provenance functionality and the penalty due to requirements for

safe composition. In addition, we expect that for tailored data-management solutions verification that our provenance functionality is integrated correctly is easier possible due to less supported functionality. We also reveal that multi-dimensional index structures are a good starting point as they can be tailored in order to exploit characteristics of a use-case achieving optimized performance. The future vision is to have interacting software product lines in a multi product line that interact with each other first ensuring safe composition (or product generation in general) and second minimizing negative effects regarding non-functional properties. This also includes a standardized procedure and respective tool support for such kinds of product lines including modeling, implementation, maintenance etc. First results, such as [SST13], seem promising in this area. Consequently, this is a request for researchers to address this topic

Interpretation of aspect-oriented paradigm

During our experimental case-studies, we encountered that today aspect orientation does not work as we would have required it. In particular, the problem is that aspects, especially in ApsectJ, are designed as additional decomposition dimension besides the object-oriented decomposition. This, in turn, results into the property that it is sometimes hard to differentiate whether limitations in our aspect-oriented case studies are due to the limitation of the aspect-oriented concept in general or due to its current interpretation and tool support. We would prefer that aspects, in the context of feature-oriented development, only exist on modeling level but not in the source of a particular variant, similar to refinements in feature-oriented programming. In this sense, we argue that aspectual feature modules as defined upon the formal bases of [ALMK08] are one of the most underestimated concepts. However, whether these findings are artificial for integration of additional features in existing case studies or have even more application scenarios is open to future investigations.

Tool support

Only with proper tool support, we can really benefit from new approaches. To this end, we really appreciate projects such as FeatureIDE. Therefore, this is a call to offer proper tool support. For the approaches presented in this thesis, we often use self-implemented tools. Therefore, integration into existing well-known tools, such as FeatuerIDE, is desirable especially for other scientist to benefit fully from our results.

Provenance research

It is commonly known in the provenance community that provenance is a cross-cutting concern. However, so far researchers did not consider whether their approaches or concept result into heterogeneous or homogeneous features. For our case studies, integrating coarse-grained and to this end homogeneous forms of provenance results in less integration effort, as we can reuse existing implementations or bind one aspect to several method executions. For fine-grained forms, the change from a homogeneous to a heterogeneous concern is problematic. We argue that this nature of the provenance concern is not only relevant for integrating provenance but for incooperating provenance in general.

A. Appendix

Programming guide - Do's and Don'ts

A seamless integration of provenance in particular, or functionality in general is hardly possible without adequate implemented and designed software systems. To this end, we summarize our insights in a programming guide. This guide is not a full-scale modeling procedure but shall be seen as some kind cook book where our recommendations are the flavors.

Persistent and accessible global keys

Having persistent and accessible keys belonging to artifacts for that we capture provenance is one of the most important properties. In case we do not have these keys, we have to introduce along with our provenance integration usually resulting in changes that increase the invasiveness of our integration. Without these keys, key-based identity instead of application of semantic identity functions is not possible. This finally results in highly increased response times, which are in opposition to our goal of minimal invasiveness.

Homogeneity labeling and separation of concerns

We integrate our provenance functionality based on homogeneity. However, sometimes the implementation is or cannot be congruent to the intended functionality (requiring provenance). Thus, labeling the respective source-code parts is of high importance in order to minimize the integration effort. This includes the object-oriented decomposition, naming conventions, or annotations.

Stick to the decomposition and integration of a functional layer

Our general assumption is that methods in object-oriented programs encode functions. Thus, their names correspond to the name of the indented function or process, the

arguments are input artifacts and the return value is the output artifact. This is an ideal that can hardly be achieved in any case. However, this highly eases our integration using AOP or FOP. Additional input and output, should be specified by using `set()` and `get()` methods. For us, it is especially difficult to identify additional input or output that is hidden in internal storage structures such as the `ImageStack` for our first case study. We also recommend, not to use generic functions where the intended functionality is supplied as argument etc.

Ideally the functions of system are encoded in a functional layer, meaning that the methods are used as functions. This does not mean we should change the programming paradigm form object-orientated to functional languages, but the current languages should be used in a functional way. Particularly, we encountered that using static processing library usually results in avoiding hidden input and output.

Artifact validation procedure

In Figure A.1, we depict the validation procedure, which is in fact a modification of the procedure to create an intermediate Artifact (cf. Figure 6.4). Although, we introduce a totally new functionality for that the original tool chain was not designed for. The applied modifications to perform Step 2 are, from perspective of the tool chain, a regulative change of the procedure to create a new intermediate Artifact. This is the least invasive possible modification.

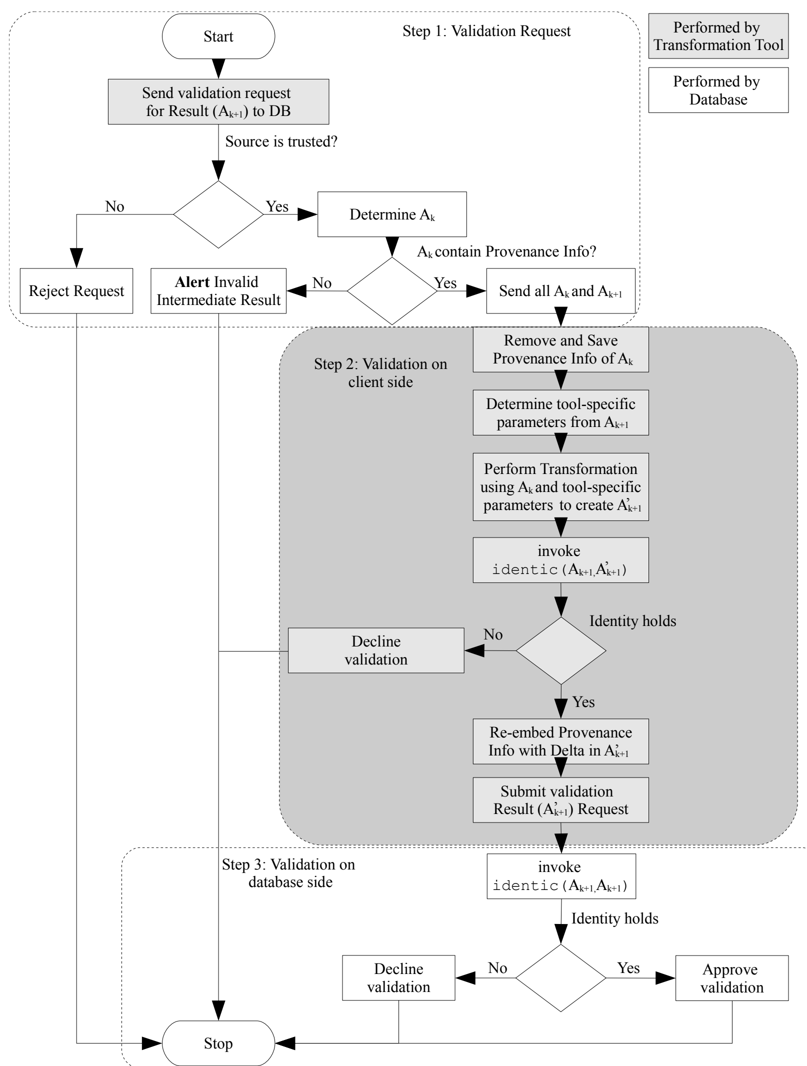


Figure A.1: Artifact validation procedure

Complexity of pg/plSQL procedures

```
1  CREATE OR REPLACE FUNCTION coltuc_embedding."extract"(marked_image integer[])
2    RETURNS bit varying AS
3  $BODY$
4  DECLARE
5    i integer;
6    bit_string bit varying(65536);--longest valid watermark 8KB
7    wm_length integer;--current length of the watermark
8    x_t integer;
9    y_t integer;
10   overheadSize integer;
11   payloadSize integer;
12 BEGIN
13   i := 1;--Arrays start with 1
14   bit_string := '';
15   wm_length := 0;
16   RAISE NOTICE 'Starting loop.';
17   LOOP
18     x_t := marked_image[2*i-1];
19     y_t := marked_image[2*i];
20
21     IF (coltuc_embedding.even(x_t)=false) THEN --Case(1) odd pixel pair
22       bit_string := bit_string || watermark_infrastructure.get_lsb(y_t)::bit(1);
23       wm_length := wm_length + 1;
24     ELSE
25       IF (coltuc_embedding.isPixelPairInD(x_t, y_t)) THEN--Case(2) pixel pair
26         bit_string := bit_string || watermark_infrastructure.get_lsb(y_t)::bit(1);
27         wm_length := wm_length + 1;
28       ELSE--Case (3) not embedded pixel pair
29         --do nothing
30       END IF;
31     END IF;
32     --first 32bit contain the overhead_size
33     IF (wm_length = 32) THEN
34       RAISE NOTICE 'WM: %',bit_string;
35       overheadSize := watermark_infrastructure.int32(bit_string);
36       RAISE NOTICE 'Overhaed size: %',overheadSize;
37     END IF;
38     --Read WM the first 32bit of the payload contain the length of the payload
39     IF (wm_length = 32 + overheadSize) THEN
40       payloadSize := watermark_infrastructure.int32(substring(bit_string,overheadSize+1,32));
41       RAISE NOTICE 'Payload size: %',payloadSize;
42     END IF;
43     --found end of watermark
44     EXIT WHEN (wm_length >= overheadSize + payloadSize);
45     i := i+1;
46   END LOOP;
47   RETURN bit_string;
48 END;
```

Figure A.2: Example implementation pg/plSQL - Coltuc watermark

To give the reader an example how complex the function are, we depict the pg/plSQL code extracting a Coltuc watermarking [CC07] in Figure A.2. The function receives as argument the field (attribute) that contains the watermark and returns the embedded message as bit string. This function assumes that the watermark is embedded sequentially and that the first 32 bit contain the length of the watermark. It loops over each pixel pair of the image and extracts a bit of the watermark message until the whole message is extracted. There are three cases of pixel pairs separated via `if` conditions. Generally the actual method bodies are very similar to object-oriented counterparts.

Details of expected extracted provenance data

In Figure A.3, we depict the details of the expected captured provenance data for provenance refinement on tool side. This includes data for the Workflow and the Existence layer of our Provenance Framework.

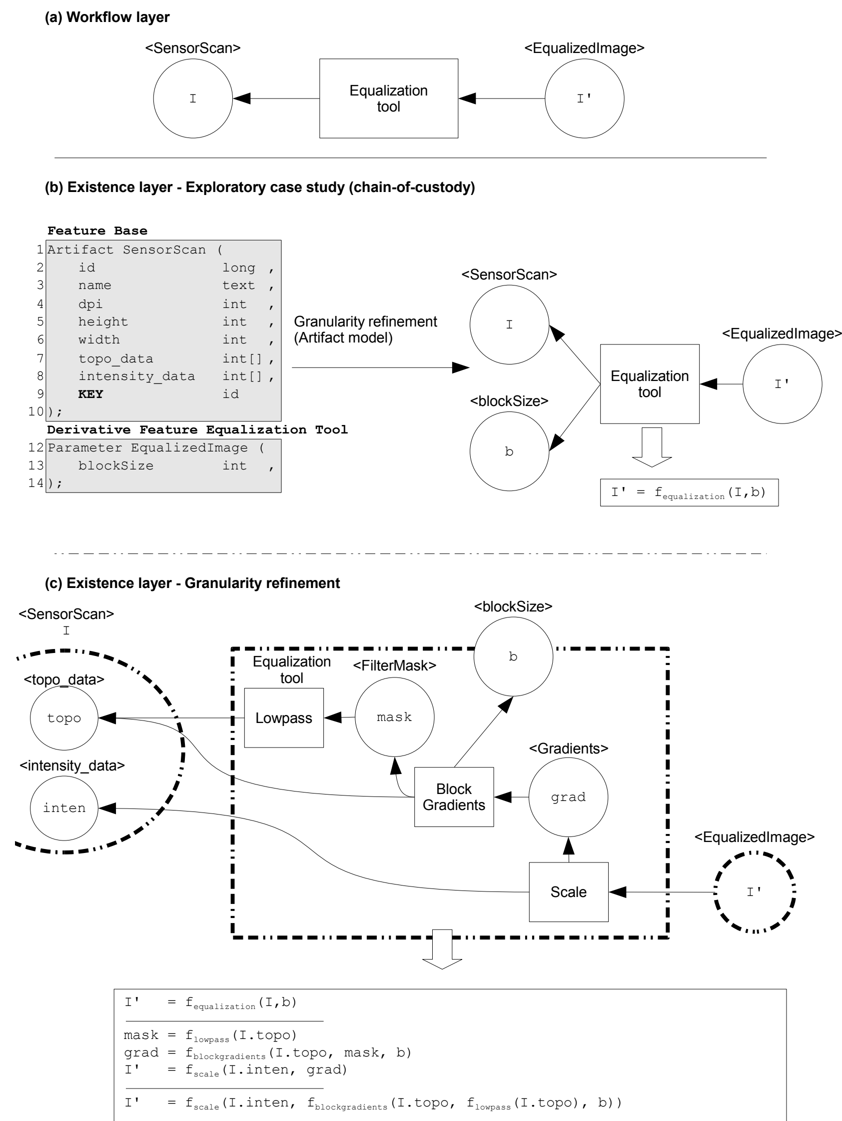


Figure A.3: Expected extracted provenance data

Bibliography

[AA97] Fevzi Alimoglu and Ethem Alpaydin. Combining multiple representations and classifiers for pen-based handwritten digit recognition. In *Proc. Int'l Conf. on Document Analysis and Recognition (ICDAR)*, volume 2, pages 637–640. IEEE, 1997. (cited on Page 131)

[ABC+10] Umut Acar, Peter Buneman, James Cheney, Jan Van Den Bussche, Natalia Kwasnikowska, and Stijn Vansummeren. A graph model of data and workflow provenance. In *Proc. Workshop on Theory and Practice of Provenance (TaPP)*, pages 8/1–8/10. USENIX, 2010. (cited on Page 8 and 15)

[ABKS13] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-oriented software product lines: Concepts and implementation*. Springer, 2013. (cited on Page 39)

[ACBS10] David Allen, Adriane Chapman, Barbara Blaustein, and Lenn Seligman. Provenance capture in the wild. In *Int'l Provenance and Annotation of Data and Processes Workshop (IPAW)*, number 6378 in LNCS, pages 98–101. Springer, 2010. (cited on Page 51 and 52)

[ADD+11] Yael Amsterdamer, Susan Davidson, Daniel Deutch, Tova Milo, Julia Stoyanovich, and Val Tannen. Putting lipstick on pig: Enabling database-style workflow provenance. *Proc. VLDB Endowment (PVLDB)*, 5(4):346–357, 2011. (cited on Page 51 and 52)

[ADT11] Yael Amsterdamer, Daniel Deutch, and Val Tannen. Provenance for aggregate queries. In *Proc. Symposium on Principles of Database Systems (PODS)*, pages 153–164. ACM, 2011. (cited on Page 11, 27, and 51)

[ADV96] Adelchi Azzalini and Alessandra Dalla-Valle. The multivariate skew-normal distribution. *Biometrika*, 83(4):715–726, 1996. (cited on Page 131)

[AKL09] Sven Apel, Christian Kästner, and Christian Lengauer. FeatureHouse: Language-independent, automated software composition. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, pages 221–231. IEEE, 2009. (cited on Page 70)

[AKL13] Sven Apel, Christian Kästner, and Christian Lengauer. Language-independent and automated software composition: The FeatureHouse experience. *IEEE Transactions on Software Engineering*, 39(1):63–79, 2013. (cited on Page 116)

[AL08] Sven Apel and Christian Lengauer. Superimposition: A language-independent approach to software composition. In *Proc. Int'l Conf. on Software Composition (SC)*, pages 20–35. Springer, 2008. (cited on Page 43 and 70)

[ALMK08] Sven Apel, Christian Lengauer, Bernhard Möller, and Christian Kästner. An algebra for features and feature composition. In *Proc. Int'l Conf. on Algebraic Methodology and Software Technology (AMAST)*, pages 36–50. Springer, 2008. (cited on Page 42, 44, and 167)

[ALS08] Sven Apel, Thomas Leich, and Gunter Saake. Aspectual feature modules. *IEEE Transactions on Software Engineering*, 34(2):162–180, 2008. (cited on Page 43, 45, and 147)

[AT97] Chuan-Heng Ang and Tuck-Choy Tan. New linear node splitting algorithm for R-trees. In *Proc Int'l Symposium on Advances in Spatial Databases (SSD)*, pages 339–349. Springer, 1997. (cited on Page 130)

[Bat04] Don Batory. Feature-oriented programming and the AHEAD tool suite. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, pages 702–703. IEEE, 2004. (cited on Page 116)

[BBK01] Christian Böhm, Stefan Berchtold, and Daniel Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):322–373, 2001. (cited on Page 128)

[BCD07] Olivier Biton, Sarah Cohen-Boulakia, and Susan Davidson. Querying and managing provenance through user views in scientific workflows. Technical Report MS-CIS-07-13, University of Pennsylvania, 2007. (cited on Page 20)

[BCDH08] Olivier Biton, Sarah Cohen-Boulakia, Susan Davidson, and Carmem Hara. Querying and managing provenance through user views in scientific workflows. In *Proc. Int'l Conf. on Data Engineering (ICDE)*, pages 1072–1081. IEEE, 2008. (cited on Page 20 and 31)

[BCH+10] Quentin Boucher, Andreas Classen, Patrick Heymans, Arnaud Bourdoux, and Laurent Demonceau. Tag and prune: A pragmatic approach to software product line implementation. In *Proc. Int'l Conf. on Automated Software Engineering (ASE)*, pages 333–336. ACM, 2010. (cited on Page 41)

[BCK12] Peter Buneman, James Cheney, and Egor Kostylev. Hierarchical models of provenance. In *Proc. Workshop on Theory and Practice of Provenance (TaPP)*, pages 10/1–10/4. USENIX, 2012. (cited on Page 27, 28, and 110)

[Ben75] Jon Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975. (cited on Page 132)

[BFH14] Peer Brauer, Florian Fittkau, and Wilhelm Hasselbring. The aspect-oriented architecture of the CAPS framework for capturing, analyzing and archiving provenance data. In *Poster presentation at Proc. Int'l Provenance and Annotation Workshop (IPAW)*, LNCS. Springer, 2014. (cited on Page 148)

[BGRS99] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is nearest neighbor meaningful? In *Proc. Int'l Conf. on Database Theory (ICDT)*, volume 1540 of *LNCS*, pages 217–235. Springer, 1999. (cited on Page 128)

[BH12] Peer Brauer and Wilhelm Hasselbring. Capturing provenance information with a workflow monitoring extension for the Kieker framework. In *Proc Int'l Workshop on Semantic Web in Provenance Management*, volume 856 of *CEUR Workshop Proceedings*. CEUR-WS, 2012. (cited on Page 148)

[Bis02] Matthew Bishop. *The art and science of computer security*. Addison-Wesley, 2002. (cited on Page 48)

[BKK96] Stefan Berchtold, Daniel Keim, and Hans-Peter Kriegel. The X-tree: An index structure for high-dimensional data. In *Proc. Int'l Conf. on Very Large Data Bases (VLDB)*. Morgan Kaufmann, 1996. (cited on Page 130)

[BKS13] Hendrik Brummermann, Markus Keunecke, and Klaus Schmid. Managing the evolution and customization of database schemas in information system ecosystems. In *Proc. Int'l Conf. on Advanced Information Systems Engineering (CAiSE)*, volume 7908 of *LNCS*, pages 417–432. Springer, 2013. (cited on Page 70)

[BKSS90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. Int'l Conf. on Management of Data (SIGMOD)*, pages 322–331. ACM, 1990. (cited on Page 130)

[BKT01] Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. Why and where: A characterization of data provenance. In *Proc. Int'l Conf. on Database Theory (ICDT)*, volume 1973 of *LNCS*, pages 316–330. Springer, 2001. (cited on Page 7, 8, 9, 14, 21, 23, 26, and 51)

[BKT02] Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. On propagation of deletions and annotations through views. In *Proc. Symposium on Principles of Database Systems (PODS)*, pages 150–158. ACM, 2002. (cited on Page 151)

[BLN86] Carlo Batini, Maurizio Lenzerini, and Shamkant Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323–364, 1986. (cited on Page 70 and 77)

[BQR07] Cristiana Bolchini, Elisa Quintarelli, and Rosalba Rossato. Relational data tailoring through view composition. In *Proc. Int'l Conf. on Conceptual Modeling (ER)*, pages 149–164. Springer, 2007. (cited on Page 67)

[BSG13] David Broneske, Martin Schäler, and Alexander Grebhahn. Extending an index-benchmarking framework with non-invasive visualization capability. In *Workshop Proc. of the German Nat'l Conf. on Database Systems in Business, Technology, and Web (BTW)*, volume 216 of *LNI*, pages 151–160. Köllen-Verlag, 2013. (cited on Page 130)

[BSI14] BSI Technische Richtlinie - Kryptographische Verfahren: Empfehlungen und Schlüssellängen. Technical Report TR-02102-1, Bundesamt für Sicherheit in der Informationstechnik (BSI), 2014. (cited on Page 112)

[BSR03] Don Batory, Jacob Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, pages 187–197. IEEE, 2003. (cited on Page 42)

[BSS08] Uri Braun, Avraham Shinnar, and Margo Seltzer. Securing provenance. In *Proc. Workshop on Hot Topics in Security*, pages 4:1–4:5. USENIX, 2008. (cited on Page 1, 8, 12, 14, 16, and 58)

[CBSA11] Adriane Chapman, Barbara Blaustein, Len Seligman, and David Allen. Plus: A provenance manager for integrated information. In *Proc. Int'l Conf. on Information Reuse and Integration (IRI)*, pages 269–275. IEEE, 2011. (cited on Page 1, 51, 52, and 110)

[CC07] Dinu Coltuc and Jean-Marc Chassery. Very fast watermarking by reversible contrast mapping. *IEEE Signal Processing Letters*, 14(4):255–258, 2007. (cited on Page 99, 100, 101, 112, and 172)

[CCF+09] James Cheney, Stephen Chong, Nate Foster, Margo Seltzer, and Stijn Vansummeren. Provenance: A future history. In *Proc. Int'l Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 957–964. ACM, 2009. (cited on Page 7, 8, 15, and 57)

[CCT09] James Cheney, Laura Chiticariu, and Wang Chiew Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009. (cited on Page 8, 9, 11, 27, 28, 30, 37, 51, 110, and 150)

[CE00] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications.* ACM Press/Addison-Wesley, 2000. (cited on Page 35, 36, and 38)

[Cen93] US Supreme Court Center. Daubert v. Merrell Dow Pharmaceuticals, Inc. - 509 U.S. 579. online - https://supreme.justia.com/cases/federal/us/509/579/case.html, 1993. (cited on Page 48)

[CFLV12] James Cheney, Anthony Finkelstein, Bertram Ludäscher, and Stijn Vansummeren. Reports of the dagstuhl seminar on principles of provenance, 2012. (cited on Page 37)

[CGS09] Michael Cohen, Simson Garfinkel, and Bradley Schatz. Extending the advanced forensic format to accommodate multiple data sources, logical evidence, arbitrary information and forensic workflow. *Digital Investigation*, 6:57 – 68, 2009. (cited on Page 102)

[Cha08] Adriane Chapman. *Incorporating provenance in database systems.* PhD thesis, University of Michigan, 2008. (cited on Page 51 and 61)

[CRB04] Adrian Colyer, Awais Rashid, and Gordon Blair. On the separation of concerns in program families. Technical Report OMP-001-2004, Computing Department, Lancaster University, 2004. (cited on Page 44)

[CSRH13] Lucian Carata, Ripduman Sohan, Andrew Rice, and Andy Hopper. IPAPI: Designing an Improved Provenance API. In *Proc. Workshop on Theory and Practice of Provenance (TaPP)*, pages 10/1–10/4. USENIX, 2013. (cited on Page 110)

[CSTS05] Mehmet Celik, Gaurav Sharma, Ahmet Tekalp, and Eli Saber. Lossless generalized-lsb data embedding. *IEEE Transactions on Image Processing*, 14(2):253–266, 2005. (cited on Page 99 and 101)

[CSV10] Stephen Chong, Christian Skalka, and Jeffrey Vaughan. Self-identifying sensor data. In *Proc. Int'l Conf. on Information Processing in Sensor Networks (ISPN)*, pages 82–93. ACM, 2010. (cited on Page 30 and 100)

[CT06] Laura Chiticariu and Wang-Chiew Tan. Debugging schema mappings with routes. In *Proc. Int'l Conf. on Very Large Data Bases (VLDB)*, pages 79–90. VLDB Endowment, 2006. (cited on Page 10)

[CW00] Yingwei Cui and Jennifer Widom. Lineage tracing in a data warehousing system. In *Proc. Int'l Conf. on Data Engineering (ICDE)*, pages 683–684. IEEE, 2000. (cited on Page 7, 9, 14, and 51)

[CW01] Yingwei Cui and Jennifer Widom. Run-time translation of view tuple deletions using data lineage. Technical Report 2001-24, Stanford InfoLab, 2001. (cited on Page 151)

[DEMD00] Saumya Debray, William Evans, Robert Muth, and Bjorn De Sutter. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems*, 22(2):378–415, 2000. (cited on Page 39)

[DG01] Lloyd Dixon and Brian Gill. *Changes in the standards for admitting expert evidence in federal civil cases since the Daubert decision.* RAND Corporation, 2001. (cited on Page 48)

[DGGH11] Julius Davies, Daniel German, Michael Godfrey, and Abram Hindle. Software bertillonage: Finding the provenance of an entity. In *Proc. Conf. on Mining Software Repositories (MSR)*, pages 183–192. ACM, 2011. (cited on Page 7, 20, and 32)

[DIIM04] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proc. Symposium on Computational Geometry (SCG)*, pages 253–262. ACM, 2004. (cited on Page 132)

[DSC+07] Curtis Dyreson, Richard Snodgrass, Faiz Currim, Sabah Currim, and Shailesh Joshi. Weaving temporal and reliability aspects into a schema tapestry. *Data Knowledge Engineering*, 63:752–773, 2007. (cited on Page 66)

[DWN01] Jana Dittmann, Petra Wohlmacher, and Klara Nahrstedt. Using cryptographic and watermarking algorithms. *IEEE MultiMedia*, 8(4):54–65, 2001. (cited on Page 98 and 99)

[FKA+12] Janet Feigenspan, Christian Kästner, Sven Apel, Jörg Liebig, Michael Schulze, Raimund Dachselt, Maria Papendieck, Thomas Leich, and Gunter Saake. Do background colors improve program comprehension in the #ifdef hell? *Empirical Software Engineering*, pages 1–47, 2012. (cited on Page 41 and 116)

[FLTC06] Jen-Bang Feng, Iuon-Chang Lin, Chwei-Shyong Tsai, and Yen-Ping Chu. Reversible watermarking: Current status and key issues. *International Journal of Network Security*, 2(3):161–171, 2006. (cited on Page 99)

[FMS08] James Frew, Dominic Metzger, and Peter Slaughter. Automatic capture and reconstruction of computational provenance. *Concurrency and Computation: Practice and Experience*, 20(4):485–496, 2008. (cited on Page 52)

[GBS+12] Alexander Grebhahn, David Broneske, Martin Schäler, Reimar Schröter, Veit Köppen, and Gunter Saake. Challenges in finding an appropriate multi-dimensional index structure with respect to specific use cases. In *Proc. German Nat'l Workshop on Foundations of Databases (GvD)*, pages 77–82. CEUR-WS, 2012. (cited on Page 127 and 128)

[GCB+97] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997. (cited on Page 132)

[GFN08] Edgar Gonzalez, Karina Figueroa, and Gonzalo Navarro. Effective proximity retrieval by ordering permutations. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(9):1647–1658, 2008. (cited on Page 132)

[GFRD09] Simson Garfinkel, Paul Farrell, Vassil Roussev, and George Dinolt. Bringing science to digital forensics with standardized forensic corpora. *Digital Investigation*, 6, Supplement(0):S2 – S11, 2009. (cited on Page 49)

[GG97] Volker Gaede and Oliver Günther. Multidimensional access methods. *ACM Computing Surveys*, 30:170–231, 1997. (cited on Page 128)

[GIT09] Todd Green, Zachary Ives, and Val Tannen. Reconcilable differences. In *Proc. Int'l Conf. on Database Theory (ICDT)*, pages 212–224. ACM, 2009. (cited on Page 11 and 27)

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java^{TM} Language Specification - Third Edition*. Addison-Wesley, 2005. (cited on Page 113)

[GKT+07a] Todd Green, Gregory Karvounarakis, Nicholas Taylor, Olivier Biton, Zachary Ives, and Val Tannen. Orchestra: Facilitating collaborative data sharing. In *Demonstration at Proc. Int'l Conf. on Management of Data (SIGMOD)*, pages 1131–1133. ACM, 2007. (cited on Page 31 and 51)

[GKT07b] Todd Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *Proc. Symposium on Principles of Database Systems (PODS)*, pages 31–40. ACM, 2007. (cited on Page 7, 8, 9, 11, 14, 21, 27, 31, and 51)

[GLL98] Yván García, Mario Lopez, and Scott Leutenegger. On optimal node splitting for R-trees. In *Proc. Int'l Conf. on Very Large Data Bases (VLDB)*, pages 334–344. Morgan Kaufmann, 1998. (cited on Page 130)

[GMM05] Paul Groth, Simon Miles, and Luc Moreau. Provenance recording for services. In *Proc. UK eScience All Hands Meeting*. EPSRC, 2005. (cited on Page 52 and 110)

[GSKS13] Alexander Grebhahn, Martin Schäler, Veit Köppen, and Gunter Saake. Privacy-aware multidimensional indexing. In *Proc. German Nat'l Conf. on Database Systems in Business, Technology, and Web (BTW)*, volume 214 of *LNI*, pages 133–147. GI, 2013. (cited on Page 99 and 130)

[Gut84] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. Int'l Conf. on Management of Data (SIGMOD)*, pages 47–57. ACM, 1984. (cited on Page 130)

[Her13] Tim Hering. Parallel execution of knn-queries on in-memory k-d trees. In *Workshop Proc. of the German Nat'l Conf. on Database Systems in Business, Technology, and Web (BTW)*, volume 216 of *LNI*, pages 257–266. Köllen-Verlag, 2013. (cited on Page 130)

[HKGV11] Mario Hildebrandt, Stefan Kiltz, Ina Grossmann, and Claus Vielhauer. Convergence of digital and traditional forensic disciplines: A first exemplary study for digital dactyloscopy. In *Proc. Int'l Workshop on Multimedia and Security (MMSec)*, pages 1–8. ACM, 2011. (cited on Page 48 and 82)

[HNP95] Joseph Hellerstein, Jeffrey Naughton, and Avi Pfeffer. Generalized search trees for database systems. In *Proc. Int'l Conf. on Very Large Data Bases (VLDB)*, pages 562–573. Morgan Kaufmann, 1995. (cited on Page 130 and 132)

[HP05] Joseph Halpern and Judea Pearl. Causes and explanations: A structural-model approach. part i: Causes. *British Journal for the Philosophy of Science*, 56:843–887, 2005. (cited on Page 22)

[KAK08] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, pages 311–320. ACM, 2008. (cited on Page 41, 67, and 68)

[Kat06] Shmuel Katz. Aspect categories and classes of temporal properties. In *Transactions on Aspect-Oriented Software Development I*, volume 3880 of *LNCS*, pages 106–134. Springer, 2006. (cited on Page 53 and 54)

[KCDV12] Stefan Kirst, Erik Clausing, Jana Dittmann, and Claus Vielhauer. A first approach to the detection and equalization of distorted latent fingerprints and microtraces on non-planar surfaces with confocal laser microscopy. In *Proc. Int'l Conf. on Optics and Photonics for Counterterrorism, Crime Fighting, and Defence (SPIE 8546)*, pages 0A/1–0A/12. SPIE Digital Library, 2012. (cited on Page 83, 137, 138, and 139)

[KCH+90] Kyo Kang, Sholom Cohen, James Hess, William Novak, and Spencer Peterson. Feature-oriented domain analysis (FODA) - Feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990. (cited on Page 36 and 37)

[KFV11] Tobias Kiertscher, Robert Fischer, and Claus Vielhauer. Latent fingerprint detection using a spectral texture feature. In *Proc. Int'l Workshop on Multimedia and Security (MMSec)*, pages 27–32. ACM, 2011. (cited on Page 131)

[KHD12] Rainer Kärgel, Mario Hildebrandt, and Jana Dittmann. An evaluation of biometric fingerprint matchers in a forensic context using latent impressions. In *Proc. Int'l Workshop on Multimedia and Security (MMSec)*, pages 133–138. ACM, 2012. (cited on Page 48)

[KHDV12] Stefan Kiltz, Mario Hildebrandt, Jana Dittmann, and Claus Vielhauer. Challenges in contact-less latent fingerprint processing in crime scenes: Review of sensors and image processing investigations. In *Proc. Europ. Signal Processing Conf. (EUSIPCO)*, pages 1504–1508. IEEE, 2012. (cited on Page 48)

[Kic96] Gregor Kiczales. Aspect-oriented programming. *ACM Computing Surveys*, 28(4), 1996. (cited on Page 42 and 116)

[KKHL10] Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. TypeChef: Toward type checking #ifdef variability in C. In *Proc. Int'l Workshop on Feature-Oriented Software Development (FOSD)*, pages 25–32. ACM, 2010. (cited on Page 53)

[Kom05] Peter Komarinski. *Automated fingerprint identification systems (AFIS)*. Academic Press, 2005. (cited on Page 83)

[KS97] Norio Katayama and Shin'ichi Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In *Proc. Int'l Conf. on Management of Data (SIGMOD)*, pages 369–380. ACM, 1997. (cited on Page 130)

[Käs10] Christian Kästner. *Virtual separation of concerns: Toward preprocessors 2.0*. PhD thesis, University of Magdeburg, Germany, 2010. (cited on Page 40)

[KS13a] Stefan Kirst and Martin Schäler. Database and data management requirements for equalization of contactless acquired traces for forensic purposes. In *Proc. Workshop on Databases in Biometrics, Forensics and Security Applications (DBforBFS), Workshops of the German Nat'l Conf. on Database Systems in Business, Technology, and Web (BTW)*, volume 216 of *LNI*, pages 89–98. GI, 2013. (cited on Page 119, 120, 137, and 145)

[KS13b] Stefan Kirst and Martin Schäler. Database and data management requirements for equalization of contactless acquired traces for forensic purposes - provenance and performance. *Datenbank-Spektrum*, 13(3):201–211, 2013. Selected for the special DASP issue Best Workshop Papers of BTW 2013. (cited on Page 137 and 164)

[KSS14] Veit Köppen, Martin Schäler, and Reimar Schröter. Toward variability management to tailor high dimensional index implementations. In *Proc. Int'l Conf. on Research Challenges in Information Science (RCIS)*, pages 452–457. IEEE, 2014. (cited on Page 127, 128, 130, 131, 132, and 135)

[KVL11] Tobias Kiertscher, Claus Vielhauer, and Marcus Leich. Automated forensic fingerprint analysis: A novel generic process model and container format. In *Europ. Workshop on Biometrics and ID Management (BioID)*, volume 6583 of *LNCS*, pages 262–273. Springer, 2011. (cited on Page 102 and 112)

[LAB+06] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006. (cited on Page 51)

[LBL06] Jia Liu, Don Batory, and Christian Lengauer. Feature-oriented refactoring of legacy applications. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, pages 112–121. ACM, 2006. (cited on Page 76)

[LK03] Dong-Ho Lee and Hyoung-Joo Kim. An efficient technique for nearest-neighbor query processing on the SPY-TEC. *IEEE Transactions on Knowledge and Data Engineering*, 15(6):1472–1486, 2003. (cited on Page 132)

[LLLS10] Rongxing Lu, Xiaodong Lin, Xiaohui Liang, and Xuemin Shen. Secure provenance: The essential of bread and butter of data forensics in cloud computing. In *Proc. Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 282–292. ACM, 2010. (cited on Page 8, 29, and 30)

[LM10] John Lyle and Andrew Martin. Trusted computing and provenance: Better together. In *Proc. Workshop on Theory and Practice of Provenance (TaPP)*, pages 1/1–1/10. USENIX, 2010. (cited on Page 8, 16, 29, and 30)

[LSR07] Frank van der Linden, Klaus Schmid, and Eelco Rommes. *Software product lines in action*. Springer, 2007. (cited on Page 35)

[LST+06] Daniel Lohmann, Fabian Scheler, Reinhard Tartler, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. A quantitative analysis of aspects in the eCos kernel. *ACM SIGOPS Operating Systems Review - Proc. of the 2006 EuroSys Conf.*, 40(4):191–204, 2006. (cited on Page 40)

[Luc97] Luc Moreau and Juliana Freire and Joe Futrelle and Robert McGrath and Jim Myers and Patrick Paulson. *Open Provenance Model*, 1997. (cited on Page 9)

[Mah02] Wolfgang Mahnke. Towards a modular, object-relational schema design. In *Doctoral consortium at Proc. Int'l Conf. on Advanced Information Systems Engineering (CAiSE)*, pages 61–71. Springer, 2002. (cited on Page 66)

[MBK+09] Pierre Mouallem, Roselyne Barreto, Scott Klasky, Norbert Podhorszki, and Mladen Vouk. Tracking files in the kepler provenance framework. In *Scientific and Statistical Database Management*, volume 5566 of *LNCS*, pages 273–282. Springer, 2009. (cited on Page 7 and 15)

[MBM+10] Patrick McDaniel, Kevin Butler, Steve McLaughlin, Radu Sion, Erez Zadok, and Marianne Winslett. Towards a secure and efficient system for end-to-end provenance. In *Proc. Workshop on Theory and Practice of Provenance (TaPP)*, pages 2/1–2/5. USENIX, 2010. (cited on Page 8, 29, and 30)

[MCF+11] Luc Moreau, Ben Clifford, Juliana Freire, Joe Futrelle, Yolanda Gil, Paul Groth, Natalia Kwasnikowska, Simon Miles, Paolo Missier, Jim Myers, Beth Plale, Yogesh Simmhan, Eric Stephan, and Jan Van den Bussche. The open provenance model core specification (v1.1). *Future Gener. Comput. Syst.*, 27(6), 2011. (cited on Page 7, 9, 14, and 15)

[MGD+12] Ronny Merkel, Stefan Gruhn, Jana Dittmann, Claus Vielhauer, and Anja Bräutigam. On non-invasive 2D and 3D chromatic white light image sensors for age determination of latent fingerprints. *Forensic Science International*, 222(1-3):52–70, 2012. (cited on Page 48)

[MGH+10] Alexandra Meliou, Wolfgang Gatterbauer, Joseph Halpern, Christoph Koch, Katherine Moore, and Dan Suciu. Causality in databases. *IEEE Data Engineering Bulletin*, 33(3):59–67, 2010. (cited on Page 8, 14, 21, 22, and 23)

[MGMS10] Alexandra Meliou, Wolfgang Gatterbauer, Katherine Moore, and Dan Suciu. The complexity of causality and responsibility for query answers and non-answers. *Proc. VLDB Endowment (PVLDB)*, 4(1):34–45, 2010. (cited on Page 21 and 22)

[MKDV11] Ronny Merkel, Christian Kraetzer, Jana Dittmann, and Claus Vielhauer. Reversible watermarking with digital signature chaining for privacy protection of optical contactless captured biometric fingerprints - a capacity study for forensic approaches. In *Int'l. Conf. on Digital Signal Processing (DSP)*, pages 1–6. IEEE, 2011. (cited on Page 100)

[MLWR01] Gail Murphy, Albert Lai, Robert Walker, and Martin Robillard. Separating features in source code: An exploratory study. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, pages 275–284. IEEE, 2001. (cited on Page 116)

[MMJP05] Davide Maltoni, Dario Maio, Anil Jain, and Salil Prabhakar. *Handbook of fingerprint recognition*. Springer, 2005. (cited on Page 48)

[MHBS06] Kiran Muniswamy-Reddy, David Holland, Uri Braun, and Margo Seltzer. Provenance-aware storage systems. In *Proc. Int'l USENIX Annual Technical Conference (ATC)*, pages 43–56. USENIX, 2006. (cited on Page 52)

[MS12] Peter Macko and Margo Seltzer. A general-purpose provenance library. In *Proc. Workshop on Theory and Practice of Provenance (TaPP)*, pages 6/1–6/6. USENIX, 2012. (cited on Page 52)

[Muc97] Steven Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers, 1997. (cited on Page 39)

[Paz11] Abdolreza Pazouki. Minimally invasive surgical sciences: A new scientific opportunity for all scientists. *Journal of Minimally Invasive Surgical Sciences*, 1(1):9–10, 2011. (cited on Page 53)

[PBL05] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software product line engineering: Foundations, principles, and techniques*. Springer, 2005. (cited on Page 35)

[Pea95] David Pearson. *Provenance research in book history: A handbook*. The British Library Publishing Division, 1995. (cited on Page 33)

[PKC+13] Mario Pukall, Christian Kästner, Walter Cazzola, Sebastian Götz, Alexander Grebhahn, Reimar Schröter, and Gunter Saake. JavAdaptor - Flexible runtime updates of Java applications. *Software: Practice and Experience*, 43:153–185, 2013. (cited on Page 39 and 153)

[RS10] Marko Rosenmüller and Norbert Siegmund. Automating the configuration of multi software product lines. In *Proc. Int'l Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, volume 37 of *ICB-Research Report*, pages 123–130. University Duisburg-Essen, 2010. (cited on Page 127)

[RSB04] Martin Rinard, Alexandru Salcianu, and Suhabe Bugrara. A classification system and analysis for aspect-oriented programs. In *Proc. Int'l Symposium on Foundations of Software Engineering (FSE)*, pages 147–158. ACM, 2004. (cited on Page 53)

[RYZ+05] Byron Roe, Hai-Jun Yang, Ji Zhu, Yong Liu, Ion Stancu, and Gordon McGregor. Boosted decision trees as an alternative to artificial neural networks for particle identification. *Nuclear Instruments and Methods in Physics Research*, 543(2-3):577–584, 2005. (cited on Page 131)

[Sam05] Hanan Samet. *Foundations of multidimensional and metric data structures*. Computer Graphics and Geometric Modeling. Morgan Kaufmann, 2005. (cited on Page 128)

[SC92] Henry Spencer and Geoff Collyer. #ifdef considered harmful, or portability experience with C news. In *Proc. Int'l USENIX Annual Technical Conference (ATC)*, pages 185–197. USENIX, 1992. (cited on Page 40)

[Sch10] Martin Schäler. Produktlinientechnologien für den Entwurf variabler DB-Schemata unter Berücksichtigung evolutionärer Änderungen. Master's thesis, University of Magdeburg, Germany, 2010. (cited on Page 73 and 74)

[Sch12] Christopher Schulz. Ansätze zur Erzeugung variabler Datenbankschemata in Softwareproduktlinien. Master's thesis, University of Magdeburg, Germany, 2012. (cited on Page 73 and 74)

[SdM03] Damien Sereni and Oege de Moor. Static analysis of aspects. In *Proc. Int'l Conf. on Aspect-oriented software development (AOSD)*, pages 30–39. ACM, 2003. (cited on Page 53)

[SDRK02] Yannis Sismanis, Antonios Deligiannakis, Nick Roussopoulos, and Yannis Kotidis. Dwarf: Shrinking the petacube. In *Proc. Int'l Conf. on Management of Data (SIGMOD)*, pages 464–475. ACM, 2002. (cited on Page 132)

[SGS+13] Martin Schäler, Alexander Grebhahn, Reimar Schröter, Sandro Schulze, Veit Köppen, and Gunter Saake. QuEval: Beyond high-dimensional indexing à la carte. *Proc. of the VLDB Endowment*, 6(14):1654–1665, 2013. (cited on Page 126, 127, 128, 131, 132, 133, 134, and 135)

[SKR+09] Norbert Siegmund, Christian Kästner, Marko Rosenmüller, Florian Heidenreich, Sven Apel, and Gunter Saake. Bridging the gap between variability in client application and database schema. In *Proc. Nat'l German Conf. on Database Systems in Business, Technology, and Web (BTW)*, volume 144 of *LNI*, pages 297–306. GI, 2009. (cited on Page 65, 66, 67, and 68)

[SLRS12] Martin Schäler, Thomas Leich, Marko Rosenmüller, and Gunter Saake. Building information system variants with tailored database schemas using features. In *Proc. Int'l Conf. on Advanced Information Systems Engineering (CAiSE)*, volume 7328 of *LNCS*, pages 597–612. Springer, 2012. (cited on Page 38, 65, and 71)

[SLS+11] Martin Schäler, Thomas Leich, Norbert Siegmund, Christian Kästner, and Gunter Saake. Generierung maßgeschneiderter Relationenschemata in Softwareproduktlinien mittels Superimposition. In *Proc. Nat'l German Conf. on Database Systems in Business, Technology, and Web (BTW)*, volume 180 of *LNI*, pages 514–533. GI, 2011. (cited on Page 65 and 72)

[SML07] Patrick Stahlberg, Gerome Miklau, and Brian Levine. Threats to privacy in the forensic analysis of database systems. In *Proc. Int'l Conf. on Management of Data (SIGMOD)*, pages 91–102. ACM, 2007. (cited on Page 99)

[SPG08] Yogesh Simmhan, Beth Plale, and Dennis Gannon. Karma2: Provenance management for data-driven workflows. *Int'l Journal of Web Services Research*, 5(2):1–22, 2008. (cited on Page 7, 15, 52, and 110)

[SRF87] Timos Sellis, Nick Roussopoulos, and Christos Faloutsos. The R$^+$-Tree: A dynamic index for multi-dimensional objects. In *Proc. Int'l Conf. on Very Large Data Bases (VLDB)*, pages 507–518. Morgan Kaufmann, 1987. (cited on Page 130)

[SRK+12] Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. SPL Conqueror: Toward optimization of non-functional properties in software product lines. *Software Quality Journal*, 20(3-4):487–517, 2012. (cited on Page 57)

[SSG+13] Martin Schäler, Sandro Schulze, Alexander Grebhahn, Veit Köppen, Andreas Lübcke, and Gunter Saake, editors. *Techniken zur forensischen Datenhaltung - Ausgewählte studentische Beiträge*, volume 1. University of Magdeburg, 2013. (cited on Page 130)

[SSK11] Martin Schäler, Sandro Schulze, and Stefan Kiltz. Database-centric chain-of-custody in biometric forensic systems. In *Europ. Workshop on Biometrics and ID Management (BioID)*, volume 6583 of *LNCS*, pages 250–261. Springer, 2011. (cited on Page 82, 85, 87, and 105)

[SSM+11] Martin Schäler, Sandro Schulze, Ronny Merkel, Gunter Saake, and Jana Dittmann. Reliable provenance information for multimedia data using invertible fragile watermarks. In *Proc. British Nat'l Conf. on Databases (BNCOD)*, volume 7051 of *LNCS*, pages 3–17. Springer, 2011. (cited on Page 30, 111, 121, and 123)

[SSS12a] Martin Schäler, Sandro Schulze, and Gunter Saake. A hierarchical framework for provenance based on fragmentation and uncertainty. Technical Report FIN-01-2012, University of Magdeburg, 2012. (cited on Page 7, 14, and 51)

[SSS12b] Martin Schäler, Sandro Schulze, and Gunter Saake. Toward provenance capturing as cross-cutting concern. In *Proc. Workshop on Theory and Practice of Provenance (TaPP)*, pages 12/1–12/5. USENIX, 2012. (cited on Page 28, 53, 110, 137, 140, 141, and 164)

[SST13] Reimar Schröter, Norbert Siegmund, and Thomas Thüm. Towards modular analysis of multi product lines. In *Workshop Proc. of the Int'l Conf. Software Product Line Conf. (SPLC)*, pages 96–99. ACM, 2013. (cited on Page 127 and 167)

[SW10] Richard Stallman and Zachary Weinberg. *The C Preprocessor*. Free Software Foundation, 4th edition, 2010. (cited on Page 40)

[TAG12] Dawood Tariq, Maisem Ali, and Ashish Gehani. Towards automated collection of application-level data provenance. In *Proc. Workshop on Theory and Practice of Provenance (TaPP)*, pages 16/1–16/5. USENIX, 2012. (cited on Page 28 and 110)

[Tan10] Val Tannen. Provenance for database transformations. In *Keynote at Proc. Int'l Conf. on Extending Database Technology (EDBT)*, pages 1–1. ACM, 2010. (cited on Page 11)

[TGM+06] Victor Tan, Paul Groth, Simon Miles, Sheng Jiang, Steve Munroe, Sofia Tsasakou, and Luc Moreau. Security issues in a SOA-based provenance system. In *Proc. Int'l Provenance and Annotation of Data Workshop (IPAW)*, volume 4145 of *LNCS*, pages 203–211. Springer, 2006. (cited on Page 30)

[Tia03] Jun Tian. Reversible data embedding using a difference expansion. *IEEE Transactions on Circuits and Systems for Video Technology*, 13(8):890–896, 2003. (cited on Page 99)

[TKB+14] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 79(0):70–85, 2014. (cited on Page 4 and 43)

[Tob13] Martin Tobies. Konzeption, Modellierung und prototypische Umsetzung eines Benchmarks für mehrdimensionale Indexstrukturen. Master's thesis, University of Magdeburg, 2013. (cited on Page 131)

[TOHS99] Peri Tarr, Harold Ossher, William Harrison, and Stanley Sutton (Jr.). N degrees of separation: Multi-dimensional separation of concerns. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, pages 107–119. ACM, 1999. (cited on Page 41)

[TPC-H10] Transaction Processing Performance Council. TPC BENCHMARK^{TM} H - Decision Support Standard Specification. Revision 2.11.0. White Paper, 2010. (cited on Page 155)

[WB97] Roger Weber and Stephen Blott. An approximation-based data structure for similarity search. Technical Report ESPRIT project, no. 9141, ETH Zürich, 1997. (cited on Page 132)

[Weh13] Adrian Wehrmann. Exemplarische Erweiterung mehrdimensionaler Indexstrukturen um Epsilon-Range-Queries. Bachelor thesis, University of Magdeburg, 2013. (cited on Page 133)

[Wic87] John Wickham. The new surgerys. *British Medical Journal (Clin. Res. Ed.)*, 295(6613):1581–1582, 1987. (cited on Page 53)

[WJ96] David White and Ramesh Jain. Similarity indexing with the ss-tree. In *Proc. Int'l Conf. on Data Engineering (ICDE)*, pages 516–523. IEEE, 1996. (cited on Page 130)

[WSB98] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proc. Int'l Conf. on Very Large Data Bases (VLDB)*, pages 194–205. Morgan Kaufmann, 1998. (cited on Page 134)

[YPXJ09] Pengfei Ye, Xin Peng, Yinxing Xue, and Stan Jarzabek. A case study of variation mechanism in an industrial product line. In *Proc. Int'l Conf. on Software Reuse (ICSR)*, pages 126–136. Springer, 2009. (cited on Page 66 and 67)

[ZCL09] Jing Zhang, Adriane Chapman, and Kristen LeFevre. Do you know where your data's been? - Tamper-evident database provenance. Technical Report CSE-TR-548-08, University of Michigan, 2009. (cited on Page 1)

E h r e n e r k l ä r u n g

Ich versichere hiermit, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; verwendete fremde und eigene Quellen sind als solche kenntlich gemacht. Insbesondere habe ich nicht die Hilfe eines kommerziellen Promotionsberaters in Anspruch genommen. Dritte haben von mir weder unmittelbar noch mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen. Ich habe insbesondere nicht wissentlich:

- Ergebnisse erfunden oder widersprüchliche Ergebnisse verschwiegen,
- statistische Verfahren absichtlich missbraucht, um Daten in ungerechtfertigter Weise zu interpretieren,
- fremde Ergebnisse oder Veröffentlichungen plagiiert,
- fremde Forschungsergebnisse verzerrt wiedergegeben.

Mir ist bekannt, dass Verstöße gegen das Urheberrecht Unterlassungs- und Schadensersatzansprüche des Urhebers sowie eine strafrechtliche Ahndung durch die Strafverfolgungsbehörden begründen kann. Die Arbeit wurde bisher weder im Inland noch im Ausland in gleicher oder ähnlicher Form als Dissertation eingereicht und ist als Ganzes auch noch nicht veröffentlicht.

Magdeburg, den