Definition of a traceability framework (including the metamodel and the modelling of processes and artefacts to allow traceability in the presence of uncertainty) for SPLs.

ABSTRACT
This document describes the definition of a traceability framework for aspect-oriented, model-driven software product line development. This framework consists of the traceability metamodel, traceability tool support and complementary areas of interest. We describe the first steps to realizing the platform, the definition of the metamodel and the organisation of the framework. The metamodel integrates the software process description, taxonomy of dependencies, and the necessary elements to represent design decisions, their rationale and the influence of uncertainty. Also, the responsibilities of the framework with respect to configuration and software evolution management are addressed. Finally, we define the initial organisation of the framework, its configuration, the relations with other work packages and the traceability tool support.
<table>
<thead>
<tr>
<th>Document ID:</th>
<th>AMPLE D4.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deliverable/Milestone No:</td>
<td>D4.1</td>
</tr>
<tr>
<td>Workpackage No:</td>
<td>WP4</td>
</tr>
<tr>
<td>Type:</td>
<td>Deliverable</td>
</tr>
<tr>
<td>Dissemination:</td>
<td>PU</td>
</tr>
<tr>
<td>Status:</td>
<td>COMPLETE</td>
</tr>
<tr>
<td>Version:</td>
<td>V1.0</td>
</tr>
<tr>
<td>Date:</td>
<td>March 31, 2008</td>
</tr>
<tr>
<td>Author(s):</td>
<td>Ismênia Galvão (UT), Safoora Shakil Khan (ulanc), Joost Noppen (EMN), Jean-Claude Royer (EMN), Andreas Rummler (SAP), Pablo Sánchez (UMA), Ralf Mitschke (TUD), Nicolas Anquetil (EMN), Birgit Grammel (SAP), Hugo Morganho (HOLOS), Christoph Pohl (SAP), Christa Schwanninger (Siemens), Lidia Fuentes (UMA), Awais Rashid (ULANC), Alessandro Garcia (ULANC)</td>
</tr>
</tbody>
</table>

Project Start Date: 01 October 2006
Duration: 3 years
## History of Changes

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>2007-11-13</td>
<td>Initial Version</td>
</tr>
<tr>
<td>0.2</td>
<td>2007-01-07</td>
<td>Initial inputs about the process</td>
</tr>
<tr>
<td>0.3</td>
<td>2007-01-22</td>
<td>Add traceability metamodel</td>
</tr>
<tr>
<td>0.4</td>
<td>2008-02-22</td>
<td>Add overview, change structure</td>
</tr>
<tr>
<td>0.5</td>
<td>2008-02-25</td>
<td>Change the taxonomy and fix some typos</td>
</tr>
<tr>
<td>0.6</td>
<td>2008-03-07</td>
<td>Metamodel completion</td>
</tr>
<tr>
<td>0.7</td>
<td>2008-03-14</td>
<td>Polishing, restructuring, fixing bugs</td>
</tr>
<tr>
<td>0.8</td>
<td>2008-03-26</td>
<td>Internal review and final version</td>
</tr>
<tr>
<td>1.0</td>
<td>2008-03-28</td>
<td>Submitted version</td>
</tr>
</tbody>
</table>
## Contents

1 Traceability Metamodel 11
   1.1 Introduction .................................................. 11
   1.2 Traceability in Software Product Line Engineering .......... 14
   1.3 The traceability reference metamodel ....................... 16
       1.3.1 Main decisions ........................................... 17
       1.3.2 Design .................................................. 18
       1.3.3 Implementation ........................................... 23
   1.4 Modelling of SPL Process and Artefacts .................... 27
       1.4.1 Summary of SPL Process .................................. 28
       1.4.2 The AMPLEx Dedicated Solution ......................... 30
   1.5 Related Work .................................................. 31
       1.5.1 General Traceability ...................................... 31
       1.5.2 Traceability and SPL ...................................... 34
   1.6 Conclusions .................................................. 35

2 Taxonomy of Dependencies and Traceability of Variations 37
   2.1 Introduction .................................................. 37
   2.2 Objectives Guiding the Classification of Traceability Dependencies 37
   2.3 Four Dimensions of Traceability ................................ 38
   2.4 Examples of Detailed Traceability Relationship ............ 40
   2.5 Templates for Documenting Trace Information ............... 41
   2.6 Trace Information Examples .................................... 43
   2.7 Conclusion and Issues ........................................ 45

3 Introducing Uncertainty Modelling in SPL Process 46
   3.1 Introduction .................................................. 46
   3.2 Uncertainty during Software Product Line Development .... 46
       3.2.1 Locations of Uncertainty during Software Product Line Development .................................................. 47
       3.2.2 Models for Describing Uncertainty in Design Decisions .... 48
   3.3 Rationale Management .......................................... 50
       3.3.1 Assumptions .............................................. 51
   3.4 Rationale Metamodel ............................................ 52
   3.5 Tracing Rationale ............................................... 55
       3.5.1 Traceability and Rationale Management in SPL .......... 57
       3.5.2 Rationale in the Traceability Metamodel ............... 58
       3.5.3 Related Work .............................................. 59
   3.6 Conclusions .................................................. 61
# Configuration and Evolution Management

## 4.1 Conceptual Foundations

- **4.1.1 Software Evolution**
- **4.1.2 Software Evolution in Software Product Lines**
- **4.1.3 Configuration Management**
- **4.1.4 Versioning Models in CM Systems**
- **4.1.5 Configuration Management in Software Product Lines**

## 4.2 Industrial Practices and Approaches

- **4.2.1 Holos**
- **4.2.2 SAP**
- **4.2.3 Siemens**

## 4.3 Configuration Management in AMPLE

- **4.3.1 Requirements (WP1)**
- **4.3.2 Architecture (WP2)**
- **4.3.3 Implementation (WP3)**

## 4.4 Connections between Traceability and CM

- **4.4.1 Connections to Traceability**

## 4.5 Framework for Traceability in CM

- **4.5.1 Representation in the Reference Metamodel**
- **4.5.2 Collaboration with the Traceability Repository**
- **4.5.3 CM in a Model Driven context**
- **4.5.4 Feature Model based CM**

## 4.6 Conclusions

## 5 Conclusion and Future Work
List of Figures

i  Initial framework for traceability. 10
1.1 High-level overview of an SPL process 15
1.2 Tracing of variations in a Software Product Line 16
1.3 The traceability reference metamodel 19
1.4 Core Tracing Datamodel 24
1.5 Repository Metamodel 26
1.6 Repository Infrastructure 27
1.7 Software Product Line Engineering. 29
1.8 Overview of the Process. 30
2.1 Representation of the two traditional dimensions of traceability in Software Product Line Engineering. 39
3.1 A triangular fuzzy number 50
3.2 Rationale in problem-solving 50
3.3 Decision modeling with assumptions (adapted from [Pos97]) 52
3.4 Rationale metamodel with uncertainty support 53
3.5 Traceability and rationale management in SPL 57
3.6 Traceability between decisions and other artefacts 58
3.7 Rationale model for requirements traceability [RJ01] 60
4.1 Nine sub-problems of variation management [Kru02] 69
4.2 Holos CVS projects structure 71
4.3 Holos branches lifetime cycles 72
4.4 Holos documentation package references 73
4.5 SAP product portfolio 73
4.6 Scenario for evolution through revision 84
4.7 Scenario for evolution through variants 85
4.8 Example instance of traceability metamodel 88
4.9 Collaboration of the CM- and TraceRepository 90
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Templates for documenting trace information in model driven, aspect oriented, product line systems</td>
<td>42</td>
</tr>
<tr>
<td>2.2</td>
<td>Initial documentation of a traceability link according to template in section 2.5</td>
<td>43</td>
</tr>
<tr>
<td>2.3</td>
<td>Initial documentation of a traceability link according to template in section 2.5</td>
<td>44</td>
</tr>
</tbody>
</table>
Overview

It is the goal of AMPLE to combine AOSD and MDD techniques to address variability at every stage in the Software Product Line (SPL) engineering life cycle. Since this variability stretches from changes in architecture and implementation to managing variations in associated artefacts such as requirements documents, a good understanding of the relationship between these elements becomes vital for achieving this goal. Within AMPLE, work package 4 addresses this task. The overall objective of WP4 is the definition and implementation of a software platform that supports general traceability for aspect-oriented and model-driven software product line development.

Work package 4 is a pivotal element of the AMPLE project, since traceability overlaps the entire SPL process and it intersects and shares dependencies with other work-packages. In particular, it is related to the three first WPs, which focus on specific development steps during SPL development. It is the responsibility of WP4 to provide the traceability platform that will be utilized by the other work packages to store tracing information and build up the trace model.

In this deliverable, the first phase of realizing the traceability platform is described. This first phase focused on the definition of a traceability framework, which consists of a traceability metamodel flexible enough to facilitate the requirements from the other work packages. With this metamodel, it is possible to describe and follow the life of requirements, classes or any other artefacts, in both forward and backward directions and through any development step. This is important since it gives essential assistance in understanding the relationships that exist within and across software requirements, design and implementation.

The metamodel supports the classical traceability dimensions, such as tracing inside one step to denote interdependencies, the so-called horizontal traceability, and between development steps, the so-called vertical traceability. However, as software product line engineering defines a new context for tracing, two new dimensions have been considered. The first dimension results from the SPL development process, which makes a distinction between domain engineering and application engineering. The SPL process has an explicit representation based on three levels: the engineering level (domain, application), the development phase (requirement, architecture, implementation and transitions between these phases) and specific development steps of interest (e.g. Java compilation or requirement analysis). The second new dimension is the result of the product line specific interests of configuration and variation. Where vertical, horizontal and variation traceability can be called development space traceability, configuration and evolution can be called time traceability.

To facilitate refined tracing and provide specific support for software product line development, the traceability metamodel is complemented by three areas: a taxonomy of traces and trace links, support for tracing in the presence of uncertainty in design rationale and support for tracing of configuration management. The taxonomy defines a categorization of trace link types and the dimensions in which these types belong. With this taxonomy, the tracing of SPL development
can be largely standardized in a uniform manner. To validate the relevancy of the taxonomy, it has been applied to industrial case studies, as can be found in chapter 2.

The second area that complements the usage of the metamodel is the support for tracing and reasoning with uncertainty in the rationale of design decisions. The design of software product lines is a very complex activity, which requires a considerable amount of information to be done effectively. However, as a product line is developed to exist for a prolonged period of time, the usefulness of the information can be limited as a result of ambiguities, incompleteness and vagueness among others. With the introduction of a design rationale model with explicit support for modeling uncertain information, the influence and impact of such information on design decision can be analyzed using the available trace information. This point is discussed in detail in chapter 3.

The third complementary area is tracing of configuration management. As configuration management typically has its own repository and rules to store artefacts and their versioning, the difficulty was to facilitate a consistent trace and configuration repository without efficiency-loss, information-loss and difficult querying. In this deliverable, the first towards complementing the metamodel with support for configuration management is taken, which can be found in chapter 4.

As part of the framework, work package 4 also has to provide tool support for feeding, querying and analysing trace information. In this deliverable, the overall structure of this tool support is defined. As traceability may be used for various purposes, several dimensions and also at a more or less fine-grained level, providing tool support is a very complex task. As a result of this complexity, it was decided to omit a completely automatic traceability system.

The result of this decision is that the traceability framework has to be configured for each tracing activity by the architect. The architect has to choose the types of artefacts he wants to trace, the dependency links he wants to observe and he also has to perform the correct trace actions during development. Nonetheless, the framework will provide basic support for storing the chosen artefacts and dependencies as well as interactive help with inserting trace actions in the source code. In this deliverable, the first effort on tool support is made by defining the location and role of basic and advanced query support of the trace model within the framework. In subsequent steps, complementary tool support will be design, such as the visualisation of trace information or measures and specific analysis support.

Work package 4 gives the traceability metamodel a central place inside the global SPL development process. As such, it integrates information related to artefacts, trace links, software processes as well as data to allow design rationale analysis. Tool support for the platform can utilize the querying interfaces to build up and analyze trace models during development. The overall picture of the framework is depicted in Figure i.

The contents of this document is organised as follows. The first chapter discusses the traceability metamodel and its resulting structure. This chapter also addresses questions related to the product line development process and its
Figure i: Initial framework for traceability.

representation within the traceability metamodel. Chapter 2 describes the taxonomy of dependencies in the context of SPL development and chapter 3 introduces the concept of uncertainty modeling and tracing as part of design decisions. A general overview of configuration management, software evolution and their impact on traceability and the traceability metamodel is given in chapter 4. In chapter 5, the deliverable is concluded and open issues and future work are identified.
1 Traceability Metamodel

In this chapter, the metamodel is presented, which is an integral part of the traceability platform. This metamodel will be used in the AMPLE project for tracing in the scope of aspect-oriented, model-driven software product line development. It has been constructed: (1) taken previously defined traceability metamodels as basis, specifically the MODELWARE traceability metamodel [Pro06]; (2) using the outcome of an internal survey on traceability for software product lines with special focus on aspect-oriented traceability performed inside the AMPLE project; and (3) driven for the traceability necessities and requirements of the different AMPLE partners, elicited after collecting a set of traceability scenarios. The development of the metamodel consisted of two phases. The first phase was the development of a reference metamodel, aimed at supporting the definition of specific traceability metamodels constructed for various software projects. The second phase was the development of a specialised traceability metamodel, dedicated to the specific needs of the AMPLE project. This metamodel is called the core metamodel and facilitates the definition of trace links with multiple sources and targets while considering type of links, types of artefacts and additional information as the development context of the trace links. The core metamodel is used as the datamodel for tooling support.

1.1 Introduction

Software development processes are comprised of several steps (e.g. requirements engineering, architectural design, implementation and/or testing), performed iteratively, where different artefacts are created (e.g. requirements descriptions, component specifications or deployment files). Artefacts of one step (e.g. components of an architectural design) are created based on the information contained in the artefacts of the previous steps (e.g. requirements descriptions) plus some decisions adopted when creating these elements (e.g. the application of a specific architectural style). Hence, a software development process can be considered, from a high-level point of view, as a chain of interconnected artefacts, created at different stages of the software development life cycle.

Keeping track of these relationships is key for the software development process, overall for the maintenance and evolution of the software products developed, since the modification of some artefacts will require reviewing the artefacts related with the modified artefacts in order to check they are still valid. For instance, a change in a certain artefact (e.g. a requirement) will require updating all the artefacts that depend on the modified one (e.g. components that
realise a requirement) as well as all the artefacts that have been influenced by this modified artefact (e.g. decisions adopted when mapping requirements into architecture). The management, i.e. identification, recording and utilisation, of these relationships between artefacts is what is known in software engineering as traceability [GF94, ARNRSG06].

This chapter presents the infrastructure required for the adequate storage of the relationships between artefacts in the scope of Aspect-Oriented Model-Driven Software Product Lines. Each one of these technologies or development paradigms creates new challenges regarding traceability.

Aspect-Oriented Software Development (AOSD) [FECA04] aims at providing improved modularisation and composition techniques to handle crosscutting concerns. These concerns are encapsulated in separate modules, known as aspects, and composition mechanisms are used to specify how and when these crosscutting concerns must be applied on the modules they crosscut [BCA+06]. Aspect-Oriented Software Development creates new challenges regarding traceability, since aspects created at one stage may not have a direct correspondence in the next development stage. This is the case since certain aspects can disappear and appear during the software development process [SFJC07]. For instance, an aspectual requirement could be simply mapped into an architectural constraint or influence an architectural decision, not manifesting in the architectural design by means of an architectural aspect [MRA05]. In addition to this, aspects introduce new kinds of dependencies and interactions [DFS02, KG06, MM06, BVMR07] of which traceability must be aware for adequate maintenance and evolution of aspects.

Model-Driven Development [BBG05, SV06] is a new technology for Software Development where models becomes first class citizens of the software development process. Using Model-Driven Development, a software system is obtained through the definition of models at different abstraction layers. Models of a certain abstraction layer are derived from models of an adjacent higher abstraction layer by means of automatic model transformations. In a Model-Driven Development process, the trace information is supposed to be generated with the highest degree of automation possible by the model transformations that are used to derive models of a certain development stage from the models of the previous stage [ARNRSG06]. This trace information is intended to make model transformation more efficient, avoiding the full regeneration of a target model when a source model is modified [Pro06, ARNRSG06]; and also allowing the specification of more expressive model transformations that make use of this trace information [VBJB07].

Software Product Lines (SPL) [PBvdL05] aims to create the infrastructure for the rapid production of software systems for a specific market segment, which share a subset of commonalities but they also have several differences or variabilities between them. In a software product line context, traceability must also deal with: (1) the relationships between the models of a specific products and the infrastructure (i.e. domain engineering or family models) used for the rapid construction of these models; (2) the relationship between the specification of variations (e.g. some feature is optional) and the design of such a variation
(e.g. a component plug-in system that support the addition of optional features); and (3) the introduction of new dependencies and interactions between variations [PBvdL05, SDNB06].

The goal of the metamodel in the traceability platform is to provide a uniform and reusable structure for defining trace models during software product line development. There exists a large body of knowledge on related to metamodels for traceability, such as [PB88, RD92, BW01, RJ01, RTK02, RJ01, CHCS’02, CHCC03, Egy03, Rie04, VB05, Jou05, JZ05, AvEI06, FHN06, Pro06, SFJC07]. A variety of approaches is proposed, in the form of a database schema [PB88, RD92, BW01], XML Schema [Rie04, JZ05, SFJC07] or metamodels [VB05, FHN06, Pro06], for storing traceability information. However, none of them address traceability when aspect-orientation, model-driven development and software product line engineering are used in conjunction.

In this chapter, the definition of the traceability metamodel to be used in the platform is described. This definition is performed in two phases. First, a reference metamodel is defined by building on the strong points of earlier efforts in literature. This metamodel can be used to define specific metamodels for storing traceability information coming from concrete aspect-oriented model-driven software product lines. The generality of this metamodel enables it for being used in aspect-oriented, model-driven software product lines processes. More specifically, it allows the creation of specific metamodels for recording the traceability information generated during the development process. The completeness and expressiveness of this reference metamodel has been validated using a set of traceability scenarios provided by each partner of the consortium. The metamodel has designed in MOF 2.0 and implemented in Ecore [BSM’03].

The second phase was the development of a specialized traceability metamodel, aimed at providing traceability support specific for the tooling of the AMPLE project. This metamodel is called the core metamodel and facilitates the definition of trace links with multiple sources and targets while considering type of links, types of artefacts and additional information as the development context of the trace links. The core metamodel is used as the datamodel for tooling support.

After this introduction, this chapter is structured as follows: Section 1.2 explains the different parts where traceability could be useful inside the software product line context. Section 1.3 describes the traceability reference metamodel proposed in this document, detailing the main decisions that have driven its design and implementation. Section 1.4 presents the elements of the software process we need to capture in the trace model. Section 1.5 comments on related work. Section 1.6 outlines some conclusions.
1.2 Traceability in Software Product Line Engineering

As commented in the introduction of this chapter, software product lines (SPL), see [PBvdL05], aim to create the infrastructure for the rapid production of software systems for a specific market segment. Typically, these systems share a subset of commonalities but they also have several variations between them.

Software product line development often implies dealing with two different but related software development processes, known as domain engineering and application engineering. The former deals with the construction of the infrastructure that represents a whole family of product, including the commonalities and all the variations contained in that family of products. The latter focuses on the development of specific products using the domain engineering infrastructure for their rapid and automatic construction.

In addition, software product lines also introduce a new kind of artefact or model that is used to specify or document: (1) commonalities; (2) variations; (3) the nature of the variations (e.g. alternative or optional); (4) relationships between commonalities and variations; (5) relationships between variations; and (6) interactions (e.g. mutual exclusion) and dependencies between variations. The most well-known and used notation for specifying variabilities in a software product lines is arguably the feature model [CHE05].

Splitting of the software development process into domain engineering and application engineering, and the use of models for specifying variations introduced new traceability demands. These demands are explained below using Figure 1.1. This figure shows a high-level view of a generic software product line process, such as the one used in the AMPLE project. This process is divided into the two main phases: domain engineering and application engineering. Each one of these phases is divided in typical phases of a software development life cycle, such as: requirements engineering, architectural design and implementation.

Traceability between domain engineering levels or between application engineering levels. Artefacts of one development phase at the domain engineering level are transformed/mapped into artefacts of the next phase at the domain engineering level (Figure 1.1, labels 1 and 2). This can be considered vertical traceability at the domain engineering level. Similarly, artefacts of one development phase at the application engineering level are transformed/mapped into artefacts of the next phase at the application engineering level (Figure 1.1, labels 6 and 7). This can be considered vertical traceability at the application engineering level. The traceability problem in this case is similar to the traceability problem in single-system engineering, but in the SPL case, traceability must also be able to deal with traceability between the models used to specify variations, such as feature models.

For instance, a feature model at the requirements engineering level could specify two alternatives, where each alternative represents systems with a dif-
different cost. Assume these alternative are simply named “cheap” and “expensive”. During the architectural design, several architectural solutions are valid for realising the same set of requirements. For instance, for providing availability, different cache and replication solutions are possible. Software architects decide to design a subset of these alternatives, allowing the selection of one of them when a specific product of the family is derived. Each one of the designed alternatives is incorporated in to the feature model as an alternative feature. Moreover, each architectural solution has associated to it a certain cost. Hence, the selection of one of the “cheap” or “expensive” features at requirements level will imply discarding some features at the architectural level. For instance, the selection of the “cheap” option will automatically discard costly replication strategies for the architectural design. In the context of SPL, traceability must also be able to deal with these kind of relationships.

**Traceability between the domain and the application engineering levels.**

In a software product line, artefacts of one development phase at the application engineering level are (automatically) obtained from the artefacts defined at the domain engineering level. This is achieved after adopting some decisions which specify the subset of the possible variations that must be incorporated into a specific product of the family of products (Figure 1.1, labels 3, 4 and 5). This implies the binding of certain variants in specific variation points, and the discarding of other variants, when there exists a relationship between the
elements contained in a application engineering model and its corresponding domain engineering model. Therefore, traceability in a SPL context must also include the management of these kind of relationships.

**Traceability between variability specification and variability realisation.** As already commented, many SPL approaches uses specific models, such as feature models, for specifying the variations of a family of products. However, mechanisms that support these variations must be incorporated into the domain models. Each variation requires the inclusion of a *variation point* in the domain models, plus the modelling of each possible variant that can be bound to this variation point and represent different alternatives when configuring a specific product.

For instance, assume in an SPL for business management software, there are several available solutions for visualising data: (1) using plain text; (2) using a pie chart; or (3) using a bar chart. The end-user must select one of these options for displaying the application data. These variations are specified by means of a feature as depicted in Figure 1.2 (left). These variations are realised by means of a plugin mechanism: a general Graphical User Interface (GUI) provides a `RegisterPlugin` interface that enables to add different plugins for data visualisation to this interface (see Figure 1.2 (`RegisterPlugin`). This interface plays the role of a variation point. Each of the possible alternatives are designed as a plugin component that will connected to the general GUI in case it is selected. Each one of this component plays the role of a variant.

Hence, there are relationships and dependencies between the specification of the existence of a variation point and how this variation point is designed and also between a set of variants and how these one are realised. Therefore, traceability in the context of a software product line must also be able to manage these relationships.

### 1.3 The traceability reference metamodel

This section describes the process followed for elaborating the traceability reference metamodel, the main decisions adopted and its design. A specific implementation of this metamodel for the AMPLE project and using Ecore is also...
described.

1.3.1 Main decisions

The definition of a traceability metamodel (or any other structure for storing tracing information), requires the definition of which artefacts \( \text{(e.g. use cases, features, components or interfaces)} \) we want to trace as well as the relationships between these artefacts \( \text{(e.g. dependencies or influences)} \) we want to be aware of. Nevertheless, these artefacts and relationships can vary between different projects, as the artefacts of interest can vary depending on the nature of the project.

Therefore, our aim is to provide an infrastructure that facilitates the specification of the artefacts and relationships between artefacts software engineers and architects are interested on tracing per project.

The specific traceability metamodels that can be defined per project are not completely disjoint; rather, they share an important subset of commonalities. This allows the definition of a reference metamodel that can be further extended in order to create the specific metamodels.

As a substantial body of knowledge exists on the generic representation of traceability models \[ \text{Jou05, FHN06, Pro06} \], we have taken this work as the starting point for the traceability metamodel. Further, the metamodel must be easily implementable in Ecore/EMF \[ \text{BSM}^+03 \] for facilitating the integration of the traceability tools with the other tools being developed in the AMPLE project. Therefore, we opted for adapting existing traceability metamodels, mainly the work done in the MODELWARE project. The result of this project is particularly useful as it focuses on traceability in model-driven development and has been implemented in ECORE, both of which are important aspects of AMPLE.

The adaptation of the MODELWARE traceability metamodel is based on the following assumptions:

1. Inside a traceability framework, a traceability metamodel plays a passive role in the sense that it records and stores the traceability information that is required for the other components of the traceability framework.

2. The traceability metamodel can be considered as a traceability data repository, which can be defined as a database or XML schema.

3. The traceability metamodel will be kept as simple as possible, shifting the intelligence or complex logic for achieving complex traceability requirements to other components of the traceability framework \( \text{(e.g. a dependency analyser for change impact analysis)} \).

4. Traceability information can be adequately represented as a directed \( \text{(hyper)} \) graph, where the nodes of the \( \text{(hyper)} \) graph are the traceable artefacts and the \( \text{(hyper)} \) arcs of the directed \( \text{(hyper)} \) graph represents trace links or relationships between traceable artefacts.
5. Since aspects can be managed as traceable artefacts, aspect-orientation does not require special extension from the traceability metamodel. Aspects can be traced in the same manner as traditional artefacts using the metamodel.

6. Model-driven development does not require any special extension of the traceability metamodel. Model-Driven Development mainly affects the way in which the traceability information is generated, since the mapping between different stages of the software development process will be performed by automatic model transformation whenever possible. Therefore, these automatic model transformations are now also responsible for feeding the traceability repository properly with adequate trace information. These transformation can also use the information stored in the traceability metamodel for specifying more powerful model transformations [VBJB07].

7. Software product lines do not require any special support from the traceability metamodel. New dependencies and relationships introduced by the use of software product lines can be stored in the traceability metamodel using the same structure as for traditional traceability links. It is the responsibility of the traceability components that use this information to interpret it properly, being aware of their new meaning.

1.3.2 Design

This section describes the design of the reference traceability metamodel. This metamodel is designed in MOF 2.0 [OMG06], as it facilitates the integration of tools and languages developed in AMPLE by providing easy mapping/transformation to Ecore.

The metaclasses that comprise the traceability reference metamodel are depicted in Figure 1.3. This traceability reference metamodel basically defines the notion of TraceModel, which is the root element of a traceability model. A trace model stores all the traceability information regarding the mapping of a set of source artefacts into a set of target artefacts, i.e. two sets of traceable artefacts. A traceable artefact in the repository can play the role of source artefact and target artefact at the same time. For instance, an architectural artefact could be a target traceable artefact when considering a requirements to architecture mapping, and a source traceable artefact when considering an architecture to implementation mapping.

A TraceModel stores both TraceableArtefacts and TraceHyperLinks between these artefacts. These TraceableArtefacts will normally be references to actual artefacts that live in some kind of source or target model. These references will be instances of the TraceableArtefactRef metaclass, which are named elements that contain in addition a URI to the location where the actual source or target artefacts are stored (e.g. a text document, a UML model), plus an id that allows to identify the traceable element inside this location (e.g. a reference to a template number inside a text document, the name or id of a model
Figure 1.3: The traceability reference metamodel

TraceHyperLinks represent explicitly trace relationships between a set of source artefacts and set of target artefacts. This enables the management of traceability model as a set of annotated, directed, bi-partite hypergraph $G = (V_1 + V_2, E)$, where $V_1$ is the set of source artefacts, $V_2$ is the set of target artefacts and $E$ is a set of annotated arcs from $P(V_1)$ to $P(V_2)$, where the annotations serve to distinguish different kinds of relationships between source and target elements. In order to allow a more fine-grained control of the traceability relationships, hyperlinks can also be decomposed into a set of traceability links, which represents a relationship (e.g. a dependency) between one source artefact and one target artefact.

Specific traceability metamodels will be defined by specifying new metaclasses for specific Traceable Artefacts and Trace Links that inherit from the corresponding abstract classes. The different metaclasses of the traceability reference metamodel are described in detail in the next subsections.

**TraceModel**

A Trace Model represents a traceability model between sets of traceable artefacts. A trace model plays the role of a container for all the traceability information related to the relationships between these sets of traceable artefacts. For example, an instance of this metaclass can be a trace model that contains the traceability information related to the mapping of a set of requirements into a software architecture. Tools processing, extracting or visualising trace information will use the instances of this metaclass as the entry point for retrieving traceability information. A typical software project will be comprise of several
trace models, which contains trace relationships between different sets of traceable artefacts, e.g. a trace model for the mapping of requirements into architecture or another trace model for the mapping of architecture into implementation, another trace model for the mapping of requirements into test cases. The traceability tools that retrieve information from the traceability repository may use several trace models in order to make more powerful queries. For instance, they could concatenate the information contained in a trace model from requirements to architecture and another trace model from architecture into implementation in order to show how a requirement manifests itself in code. On the other hand, all traceable artefacts and trace links could also be stored in a single trace model, which would represent the whole software development process. In this case, it would responsibility of the tools that retrieve information from the traceability repository to distinguish to which development stage each link and traceable artefact contained in this global TraceModel belongs to.

Properties

**traceHyperLinks**: A Trace Model contains zero or more trace hyperlinks (see TraceHyperLink).

**traceableArtefacts**: A trace model contains zero or more traceable artefact (See TraceableArtefact). Part of these traceable artefacts will belong to one of the set of bipartite hypergraphs, i.e. the source model, and the other one will belong to the other set of the bipartite graph, i.e. the target model.

TraceableArtefact

A **Traceable Artefact** can represent any artefact in the development lifecycle of which traceability management should be aware. This can be an explicit artefact contained in some kind of model. For example, this can be requirements, components, interfaces, classes or test cases. It is also possible to represent artefacts generated during the development process that are not explicitly represented in any kind of model, e.g. architectural decisions, but that they are of relevance for traceability management. In this latter case, in order to avoid the losing of these artefacts, they will be stored in the traceability model as instances of a TraceableArtefact metaclass.

Associations

**isSourceHyperLink**: A Traceable Artefact can be part of the set of source artefacts of zero or more traceability directed hyperlinks. For instance, a requirement can be part of the source of several (including zero) hyperlinks, which associate this requirement (and maybe other requirements) with a set of com-
isTargetHyperLink: Traceable Artefact can be part of the set of target artefacts of zero or more traceability directed hyperlinks. For instance, a component can be part of the target of several (including zero) hyperlinks, which associate this component (and maybe other components) with a set of requirements.

isSourceLink: Traceable Artefact can be the target of zero or more traceability single directed links. For instance, a requirement can be the source of several (including zero) links, which associate this requirement with different components (each link would associate this requirement with exactly one component).

isTargetLink: Traceable Artefact can be the target of zero or more traceability single directed links. For instance, a component can be the target of several (including zero) links, which associate this component with different requirements (each link would associate this component with exactly one requirement).

Properties

traceContext: justification for the existence of this traceable artefact.

TraceableArtefactRef

A subtype of TraceableArtefact are those “physical” artefacts that live or are placed in some kind of model of the software development life cycle. These artefacts are placed outside the traceability repository, and the traceability repository, in order to avoid redundancies, limits itself to reference these TraceableArtefacts. An explicitly existing TraceableArtefact is unambiguously identified by an Universal Resource Identifier (URI), which describes where the model that contains this artefact is located (e.g. in a file), plus an id that serves to find this element in such a location (e.g. the name of a class in a file, the complete name of a UML element or its id).

Properties

URI: Represents the Universal Resource Identifier or location (e.g. physical file, model, repository) of the container for this Traceable Artefact.

id: Represents a unique identifier for the Traceable Artefact inside this location. For instance, this value could be the name of the element or an unique identifier inside this location, such as the id of a UML model element.
TraceHyperLink

A TraceHyperLink abstracts the transformation of one set of source artefacts into another set of target artefacts. An instance of this metaclass represents a directed hyperedge connecting two set of artefacts in the trace graph.

Properties

sources: Represents the traceable artefacts that play the role of source nodes of a directed hyperlink.

target: Represents the traceable artefacts that play the role of target nodes of a directed hyperlink.

traceContext: A justification for the existence of this directed hyperlink.

contains: The directed links in which this directed hyperlink can be discomposed.

TraceLink

A TraceLink abstracts the transformation of one set of source artefacts into another set of target artefacts. An instance of this metaclass represents a directed hyperedge connecting two set of artefacts in the trace graph.

Properties

source: Represent the traceable artefact that plays the role of source node of this directed link.

target: Represent the traceable artefact that plays the role of target node of this directed link.

TraceContext

A TraceContext contains the justification or motivation behind a certain Trace Hyperlink or a Traceable Artefact. Not all artefacts and transitions would require a trace context. For instance, trace links created for automatic model transformation rules are practically self-explanatory: their existence is due to they are coded in an automatic model transformation rule, this rule being the justification behind that link.
1.3.3 Implementation

This section presents an implementation for the metamodel depicted in previous section. Some modifications have been done in order to ease implementation. Since one of the goals is to make the traceability repository configurable per project, new elements are introduced in the implementation metamodel (see Figure 1.6). More specifically, Traceable Artefact instances and their type definitions, i.e. Traceable ArtefactTypes are stored at the same level, which enables runtime configuration of the traceability metamodel. This is equivalent to a mixing of the model (M1) and metamodel (M2) levels in only one level. The same technique has been applied to TraceLinks and TraceLinkTypes. Other solutions could be adopted, such as the use of Dynamic EMF [BSM+03], in order to achieve runtime configuration. We have opted for this mixing of the M1 and M2 levels for efficiency reasons. In addition, TraceHyperLink and TraceLink have been compressed in TraceLinks for efficiency reasons, as well as Traceable ArtefactRef and Traceable Artefact have also been joint in Traceable ArtefactType. The declaration of Traceable Artefact is possible at the implementation level since the attributes for referencing existing artefacts are optional (0..1 multiplicity). All the mentioned elements are explained in this section in greater detail.

As already stated, a dedicated trace model based on the reference model including instances of the appropriate metaclasses has to be created by a user. This is an option for tool builders who decide to create applications for particular stages in the development process. However, the prototype tools of the AMPLE project do not address particular stages, instead the tooling is of general purpose nature. For this reason some aspects will be implemented in the tooling in a modified fashion. The concepts described in this section and implemented in the tooling are similar to the concepts described earlier, however they are tailored to runtime configuration/specification rather than design time specification. The following paragraphs about Traceable ArtefactType and TraceLinkType are an accommodation to that premise.

The metamodel is depicted in figures 1.4 to 1.6. It is centred around the assumption that all trace information can be represented by a directed graph. The elements of the metamodel are explained in greater detail below. The metamodel consists of two parts: one containing artefacts, links and the relations between them and one containing the storage of these items in a central repository. Note that the figures follow Ecore conventions which may be different from traditional UML. One such difference is that association are implicitly bidi-

---

1See Eclipse Modeling Framework project: http://www.eclipse.org/modeling/emf/
rectional in UML whereas they are directed (or unidirectional) in Ecore. This explains why many relations are duplicated in the figures to allow forward and backward traceability.

Figure 1.4: Core Tracing Datamodel

**TraceableArtefact:** A *TraceableArtefact* represents a (physical) artefact that plays a role in the development cycle. The complexity of such an artefact is arbitrary, it may represent a requirement, an UML diagram, an element inside a diagram, a class or a method inside a class. An artefact is an abstraction for any element that is created during the development of a software system. An artefact is unambiguously identified by a locator (resourceId), which describes where this artefact is located (such as in a file or a directory) and how it may be accessed. Using this mechanism, a model element inside an UML diagram or a method inside a class can be identified. An artefact carries a (not necessarily unique) name for the purpose of displaying in tools, a unique id, which is generated automatically and is used for internally identifying an artefact and an optional set of properties. These properties are held in a map and may be set by the creator of the artefact. An instance of *TraceableArtefact* corresponds to a node in the trace graph.

**TraceLink:** A *TraceLink* is the abstraction for the transition from one artefact to another. An instance corresponds to an hyperedge linking two artefacts in the
trace graph. A transition is always directed, therefore a from-to-relation between artefacts is created by a trace link (between source and target artefacts). While transitioning between artefacts, several source artefacts may be combined to a single target artefact. Analogous one source artefact may lead to the creation of several target artefacts. For this reason the associations between TraceableArtefact and TraceLink sources and targets have a cardinality of $1..*\$ assigned — at least one source and target artefact is mandatory when we have a TraceLink. On the opposite, the associations incomingLinks and outgoingLinks have a cardinality of $0..*\$ assigned — artefacts may exist without incoming or outgoing links. This is the case if artefacts are situated at the beginning or the end of a trace chain. They may even exist without incoming and outgoing links, this way unused artefacts may be identified by a reasoning framework.

**TraceableArtefactType:** During the process of tracing information about the design of a software system, different artefacts of different types must be taken into account. For this reason each TraceableArtefact has an instance of TraceableArtefactType assigned. This type separates artefacts from each other. In the most simplest form, an artefact type is indicated via a name and an unique identifier. A user may create his own artefact types by instantiating a new object of TraceableArtefactType. Examples for types would be requirements document, requirement, class or UML model. The justification behind the decision to design types of artefacts in this manner is that is it very likely that users may add types that cannot be provided by the tooling in the first place, because users may want to trace very specific artefacts. As a convenient solution, a user may create a such a type at runtime, rather than having to create an own implementation and compile it into the system. In addition, applying the concept of subclassing only makes sense in cases where a subclass contains significant changes in structure and/or behaviour, which is not the case for types here (the main difference between is the naming and the position in the type hierarchy). The AMPLE tool(s) will already define the most common artefact types and expose them to the user.

Artefacts types may be grouped in a hierarchical manner. This is enabled by the self-associations baseType and subTypes. For instance, a basetype Document may have subtypes LaTeX Document and Word Document.

**TraceLinkType:** Analogous to the type of an artefact each link does have a type because the relation between two artefacts may differ. Examples for such types would be contains, depends of or is generated from. For this reason each instance of TraceLink is assigned an instance of TraceLinkType. Being able to define a new link type at runtime follows the same design considerations as stated for TraceableArtefacts. Similarly to TraceableArtefactType the AMPLE tool(s) will define the most common relation types and leave room for user-specific extensions. The relation type is (at the moment) defined with a name and unique identifier. In addition, trace link types can also be grouped in a hierarchical manner.
**TraceContext:** The existence of an artefact or the relation from one artefact to another may be justified in some way, in the case of a “depends on” transition for example or the creation of a factory class. Not all artefacts and transitions would require such a justification, for example a “contain” transition is rather self-explanatory. This can be modelled by attaching a TraceContext to relations and/or artefacts. By using artefacts and relations in conjunction with the context, situations like artefact A depends on artefact B because of some reason can be modelled.

![Figure 1.5: Repository Metamodel](image)

**TraceRepository:** The TraceRepository represents a container (or better some kind of database) for collecting trace information. All artefacts and all relations that are created during the assembly of trace information are stored in this repository. The logic behind this repository has to ensure, that each relation and each artefact is stored exactly once. The internal data structure for storing these elements is open and is hidden from a user. Tools processing, extracting or visualising trace information rely on this repository as a central entry point. It may also be seen as a self-contained layer between tools providing trace information (i.e. generators that create code from model elements) and tools extracting facts out of trace information.

In addition to collecting trace information artefact and relation types are registered at this repository. This enables tools to identify existing artefact and relations types. User-specific types must be registered at the repository too.

**RepositoryManager:** As shown in figure 1.6 there are a couple of auxiliary components that enable access to the trace repository and manipulation of its content. The central point of entry is the RepositoryManager which is responsible for initialisation, connection handling and handling of other manager classes.

---

2It is assumed that all situations that may occur in the context of tracing can be modelled this way.
Figure 1.6: Repository Infrastructure

**TypeManager:** The *TypeManager* is an auxiliary component that exposes services like adding, deleting and managing artefact and link types to the user.

**ItemManager:** The *ItemManager* is an auxiliary component that exposes services like adding, deleting and managing artefacts and links and creating and removing relations among them.

**QueryManager:** The *QueryManager* is the point of entry for querying the repository content. It offers access to several common, preconfigured queries, enables the creation and execution of queries and provides the results of these queries.

**ExtractionManager:** The *ExtractionManager* manages extractors – modules that extract actual trace information. Extractors may be registered by users, configured in a proper way and executed to collect trace information. Usually each extractor is specialised on a special aspect of all the development stages when designing a product line, i.e. finding relations between features and requirements.

### 1.4 Modelling of SPL Process and Artefacts

Traceability may be used for various purposes, several dimensions and also at a more or less fine-grained level. Considering this complexity, it was not possible, at least in a first time, to provide an automatic traceability system. Providing a simple automatic system which collects every traces is not so useful
and can be really costly. Automatic and intelligent configuration is a complex challenge: first, we have to known the need for tracing, then to infer the information (artefacts and links) to trace and finally to create and collect the traces. The current difficulties are: i) the needs for tracing are various and complex and not yet well-known, ii) tracing impacts many artefacts (qualitatively and quantitatively), dependencies and related informations, iii) we have a complex and realistic software development with four traceability dimensions, and iv) trace creation actions have to be inserted in some specific locations of the process which is sometimes manual. In a first attempt, we choose to enable the configuration of the traceability framework for each usage by the product line architect. The architect has to choose the kinds of artefacts he wants to trace, the dependency links he needs to observe and also to insert the correct trace actions in the right place. However, on one hand, the framework will provide a basic support for storing the chosen artefacts and dependencies using interface managers. On the other hand it is not difficult to provide some interactive help to insert trace actions in the source code. Automatic insertion is also possible in case of MDE steps or using AOP, but the difficulty is the information knowledge and management rather than the trace creation. One question remains about the automatic insertion of trace actions, this raises some more or less difficult questions which will be studied and summarised in the next deliverable.

In order to propose tool support for traceability and to get a consistent support for configuration and evolution management, the question of the development process has to be raised. Two questions are related to this. The first concerns the traceability dimensions, and as we discussed earlier four have been defined. The software PL process is strictly linked with vertical, horizontal and variation traceability. The configuration traceability is orthogonal to the software process. The second question are our hypothesis about the manual configuration of tool support for traceability.

1.4.1 Summary of SPL Process

The software process of a product line is usually split in two dimensions: domain engineering and application engineering. Each of these dimensions can support a more or less classic software cycle but with emphasis on artefact production and reuse. A specificity is the management of artefact and product variations. Figure 1.7 represents the general view of a product line engineering. The upper half is domain engineering: That means analysing the domain of interest and creating the core assets. The lower half is application engineering: It contains the activities related to the creation of an actual product from core assets. Left and right parts distinguish between problem space and solution space. Problem space consists in activities carried out by domain expert independent from software implementation and generally use a domain-specific vocabulary and knowledge. The solution space are the activities related to the product creation by software developers and requires a software engineering vocabulary and programming techniques. A zoom on the process is depicted in
The main steps are the following:

- From **domain requirements** a **formal domain metamodel** is derived using some automated and manual steps. Requirements are natural language inputs but natural language processors are able to analyse them and compute a semi formal requirements model expressing structure and dependencies.

- Transformation steps are defined to convert the domain metamodel into a **formal solution space metamodel**. This process is mainly manual even if the use of tools is possible to assist in this task. Two alternatives for code generation are considered, either model transformations or powerful code generation. One parallel activity concerns the development and management of the core assets.

- Collect product requirements and instantiate the **formal metamodel** into a formal domain model of the **product**.

- Apply the transformations defined in domain engineering space to generate a **formal solution space model**. The product is then created by combining the elements of the core assets needed for the product.

This process will be more detailed and tuned until the end of the project. For traceability we need to know the precise steps which generate the artefacts and the trace links. We need a model of the development process that is: the steps and their temporal ordering which may be expressed by a directed graph. One additional characteristic is to know if this step is manual or automatic and then a reference to the tool or computation details has to be stored. This is obviously useful in many scenarios to localise change impact. This is also related to the software configuration management question, see Chapter 4.
for more details. Generally software configuration management systems have knowledge of the software life cycle. For instance, this information is represented as annotation metadata on the versioned items. Thus, in order to get a consistent view and to propose a sensible tool chain, we have to propose a way to store the process information. This information will be further exploited by traceability tool support, configuration and evolution management as well as needed for other activities, for instance, the design rationale analyser coping with uncertainty.

1.4.2 The AMPLE Dedicated Solution

The most general solution is to model our specific process using some general existing metamodel like SPEM [OMG05]. However, it is only required if we want to have a process which may vary and rather independent from the artefacts. A related solution is to define an explicit and specific metamodel for our purposes. Since we are defining a precise software process, these solutions are not really required in our context.

In our case a light weight solution is sufficient enough since we have mainly to know artefacts, their kinds, the software steps involved and the ordering of these steps. The model elements related to this question are the following:

- **TraceableArtefactType**: requirements, class, file and so on.
- **TraceableArtefact**: properties.
- **TraceContext**: properties.
- **TraceLinkType**: some constraints with the taxonomy of dependencies.
A first remark is that an artefact can be used in several parts of the SPL process, for example, components are used at the application and domain engineering levels. Thus, the process value cannot be attached to the artefact type but to the artefact instance. A second point is that we have not only process steps in requirements or architecture but also to go from requirement to architecture. Even if the frontier is generally a bit fuzzy we have to explicitly precise it. Thus our process steps may belong to four kinds: Requirement, RequirementsToArchitecture, Architecture, ArchitectureToImplementation, Implementation. Another point to consider is that we have generally an iterative process, an iteration number can be used to precisely identified a process step. The key/values pairs to include in the trace context properties could be:

**Engineering Phase:** \{Domain, Application\}

**Development Step:** \{Requirement, RequirementToArchitecture, Architecture, ArchitectureToImplementation, Implementation\}

**Process Step:** \{Manual Step, EAMiner Extraction, XPand Weaving, Java Compilation, ...\}.

**Iteration Number:** an Integer.

Associated to the process description, information is the ordering of these steps which may be defined as a separated directed graph. Another more flexible alternative for the process steps could be to define an object-oriented class for this information. A last remark is that our precise process is not yet completely defined, thus full details cannot be written. One also have in mind that the real process may vary from one project to another one thus a user configuration is required.

### 1.5 Related Work

As already commented in the introduction, there is a considerable body of work that defines the data structure, under different forms, for storing traceability information [PB88, RD92, BW01, RJ01, RTK02, CHCS02, CHCC03, Egy03, Rie04, VB05, Jou05, JZ05, AvEI06, FHN06, Pro06, SFJC07], as identified as part of an internal survey carried out in the AMPLE project during the M4.1 milestone. Their main results related to the construction of a traceability metamodel are summarised below.

#### 1.5.1 General Traceability

Potts and Bruns [PB88] are among the first to mention the recording of early design decisions and deliberations as key factor for the success of software development process. A design history is regarded as a network consisting of artefacts and deliberation nodes. Artefacts represent specifications or design
documents. Deliberation nodes represent issues, alternatives or justifications. Existing artifacts give rise to issues about the evolving design, an alternative is one of several positions that respond to the issue, and a justification is a statement giving the reasons in favor of and against the related alternative. The metamodel is implemented both in hypertext and in Prolog database. This work established the basis for the development of new traceability metamodels, but due to the time since its publication, it needs to be updated in the light of the emergence of new development paradigms, such as aspect-orientation, and technologies, such as model-driven development.

Ramesh and Darh [RD92] continue the work of Potts and Bruns [PB88]. They propose REMAP (Representation and Maintenance of of Process Knowledge) as a mechanism to record in some structured manner the history of a software development process. The most important component of this process is the knowledge about reasons behind design decisions or design rationales that shape the design. Although the ideas and concepts behind REMAP are similar to our goals, the technologies and standards behind this work are out of date (e.g., it is based on the use of DFD’s (Data Flow Diagrams)).

Ramesh and Jarke [RJ01] point out that different projects and teams have also different traceability needs. Therefore, traceability schemas, such as proposed by [PB88, RD92], require being adapted per project and organisation in order to fit in with the actual traceability requirements of each development process. They propose the creation of reference traceability metamodels that facilitate the creation of specific traceability metamodels for concrete projects, similarly to our approach. However, Ramesh and Jarke focus on requirements traceability, whereas we focus on traceability of any artefact existing in the software development life cycle, e.g., architectural components, test cases or implementation classes. They define a set of trace link types, based on empirical studies. Nevertheless, the metamodel these authors propose is not a metamodel in the strict model-driven sense, it can be considered as a conceptual model for traceability. This conceptual model is implemented using a traceability tool called SLATE (System Level Automation Tool for Engineers). We have based our approach on the experiences and the experiments carried out by Ramesh and Jarke in order to define a reference traceability metamodel, in the Model-Driven Development sense, that can be easily implemented in Ecore and integrated with the AMPLE tools. This work is revisited in Limón and Garbajosa [ELG05], who propose a categorisation of traceability link types that can help to the creation of specific traceability metamodels per project.

Almeida et al [AvEI06] present a methodological framework for facilitating the management of traceability relationships in Model-Driven Development. They represent traceability information by means of tables where the headers of rows and columns represents traceable artefacts and the cells represent trace links. These cells can contain different information depending on the kind of traceability link it represents. Eyged [Egy03] presents an automated approach to generating and validating trace dependencies, where traceability information is represented by means of a graph, where the nodes of the graph are scenarios used for testing and the edges of the graph represent traceability links between
these scenarios. The traceability links will be inferred from the execution of the scenarios. The representations for traceability information used both in Almeida et al. [AvEI06] and Eyged [Egy03] are equivalent. Traceability information represented in tables can be represented as an annotated graph, and graphs can be represented by means of tables. We have based on this work for representing trace information as bipartite hypergraph.

Cleland-Huang et al [CHCS+02, CHCC03] use a representation similar to the tables and graphs of Almeida et al [AvEI06] and Eyged [Egy03], but they add mechanisms to the trace links that helps their maintenance. Cleland-Huang et al [CHCS+02, CHCC03] propose the attachment of event-notification information, or triggers, to traceability links for helping their maintenance. They also propose the use of clusters of traceable artefacts for reducing the number of traceability links. They integrate their approach into DOORS. Nevertheless, the representation of the traceability information is not modified by the addition of these triggers.

Vanhoff and Berbers [VB05] propose a UML Profile for representing traceability information directly in UML models. Hence, this trace information can be used for specifying more expressive model-transformations over these UML models [VBJB07]. The main drawback of this work is the tangling of traceability information and model information, which could hamper the understanding of both model and traceability information, reducing scalability. A interesting solution could be to maintain the traceability information in a separate model, which can be a table or a graph, such as in Almeida et al [AvEI06] or Eyged [Egy03], and to add the traceability information to the models when required, by means of a model transformation that merges the system model and the traceability model, such as proposed by Kolovos [KPP06].

Jouault [Jou05] considers the role of traceability in automatic model transformations and presents an approach for facilitating the addition of code for generating traceability links to model transformations. Nevertheless, nothing it is stated about the structure of the traceability information beyond it can be managed as another model, which can be both input of model transformations, similarly to Vanhoff et al [VBJB07], and output of a model transformation. Falleri [FHN06] refines this work proposing a generic traceability metamodel for the Kermeta model transformation language\(^3\). Such a metamodel represents traceability information as a bipartite graph whose nodes are traceable artefacts contained in models. A trace is managed as a chain of traceability links between a set of source and target models.

The MODELWARE project proposes a traceability framework [Pro06] based on two modelling levels: (1) a traceability reference metamodel level, which defines the metaclasses for traceable artefact and traceability link types; and (2) a traceability model level, which is comprised of instances of traceable artefacts and traceability link types. Traceability tools use this metamodel and these models for storing and retrieving traceability information. This two levels organisation is hidden to the traceability tool that use it by means of an API that provides a

\(^3\)http://www.kermeta.org/
set of Traceability Services. Traceability information is managed in the MOD-ELWARE approach as a bipartite graph whose nodes are traceable artefacts contained in either source or target models.

Sánchez et al [SFJC07] present a generic traceability schema, expressed in XML, for tracing in the scope of aspect-orientation. It allows the grouping of artefacts in alternatives. These alternatives are potential solutions for the mapping of some artefacts of the previous level. Rationales for each alternative are recorded, as well as which the selected is and the justification for that selection.

1.5.2 Traceability and SPL

There also are several work that address traceability in the context of Software Product Lines [BW01, RTK02, Rie04, PBvdL05, JZ05]. They focus mainly on establishing traceability links between models that describe variability, such as, for instance, feature models, and the other models that specify the system, such as, for instance, use case models or architectural models.

Bayer and Widen [BW01] propose the establishment of traceability links between the elements of the metamodels used in already existing methodologies for Product-Line Engineering, such as PuLSE [BFK99]. Once the kind of traceability links that can be created are defined, traceability links are created between models elements as instances of their corresponding metaclasses. Nevertheless, nothing is stated about how these traceability link types should be defined at the metamodel level and/or how they should be stored (authors mention they are carrying out experiments using a database for the storage of traceability information).

Ramesh et al [RTK02] present traceability based knowledge management system to support the design, customisation and delivery of information product and e-service families. They basically apply their previous work, called REMAP [RJ01], initially designed for single-system engineering to software product line. An important point here is that modifications in REMAP must be introduced due to the Software Product Lines development processes are different from single-system engineering development process, but the traceable artefacts are similar in both cases.

Riebisch [Rie04] proposes the use of feature models for facilitating traceability between requirements and design artefacts, as well as the enrichment of traceability links with the description of design decisions. Traceability information is stored using third-party repositories and XML documents and tagging system models with traceability information.

Jirapanthong and Zisman [JZ05] aims to provide a solution for the management of multiple and heterogeneous documents in Software Product Line Engineering. They propose a reference traceability metamodel, containing a set of predefined traceable artefacts and traceability relation types. They propose a rule-based approach, where rules assist in the creation and management of traceability information. These rules are represented in XQuery, and traceability information stored in XML documents. As a major drawback of this approach,
traceable artefacts (e.g. use cases documents) must be preprocessed and converted into a XML form before being incorporated into the traceability framework.

Among all these approaches, the Orthogonal Variability Metamodel (OVM) presented by Pohl et al [PBvdL05] is probably the most well known. It proposes the modelling of variability separately from the other artefacts of the system development and the establishment of traceability links between the OVM and the artefacts of the system where variability is designed or realised. However, nothing is stated at the current moment about what kind of links could be established or how they should be stored.

These work address traceability in single-system engineering, and, in some cases, they also cover Aspect-Orientation, Model-Driven Development or Software-Product Lines, but none of them address the Aspect-Orientation, Model-Driven Development and Software-Product Lines at the same time, which is the goal of the AMPLE project. Our metamodel follows a two level organisation similar to the MODELWARE project, but it represent traceability information as a bipartite hypergraph, which allows to reduce the amount of traceability links, using a clustering technique similar to Cleland-Huang et al [CHCS+02, CHCC03].

1.6 Conclusions

This chapter presented the traceability metamodel that is designed and implemented as part of the traceability platform in the AMPLE project. This model has been based on existing work, in particular the MODELWARE traceability metamodel was selected as the source for driving the design and implementation of the metamodel. Additionally, the traceability needs of the different AMPLE partners were collected as set of templates, in order to ensure all the partners’ requirements were fulfilled during the design and implementation of the AMPLE traceability metamodel.

The development of the metamodel consisted of two phases. The first phase was the development of a reference metamodel aimed at supporting the definition of specific traceability metamodels constructed for various software projects. The second phase was the development of a specialised traceability metamodel, dedicated to the specific needs of the AMPLE project. This metamodel was called the core metamodel and facilitates the definition of trace links with multiple sources and targets while considering type of links, types of artefacts and additional information as the development context of the trace links. The core metamodel is used as the datamodel for tooling support. As a novel contribution, traceability information is managed as a bipartite hypergraph, instead of a simple graph as usual in the literature. The traceability metamodel is considered as passive part of all the traceability framework, used by other traceability components and tools for storing and retrieving traceability information. Therefore, the main logic and intelligence of the traceability framework is the responsibility of the tools that access the traceability metamodel. The AMPLE traceability metamodel has been implemented in Ecore, and is accessed by
means of an API.

Future work on the metamodel includes performing several experiments for analysing the expressiveness and scalability of our approach. In addition, the integration of the SPL process description and the taxonomy have to be done. Finally, the API for the metamodel repository will be extended to support tooling: configuration management, and rationale design analysis.
2 Taxonomy of Dependencies and Traceability of Variations

2.1 Introduction

The aim of this chapter is to explore the semantic dependencies existing across the artefacts as defined in the metamodel (Chapter 1). A classification of possible TraceLinkType is proposed that will allow developing tools to automatically manipulate the traceability links. This is ongoing work and the proposal included here only sets a framework for more detailed classification that may depend on specific needs.

These traceability dependencies fit into the traceability metamodel (Section 1.3.3) as TraceLinkTypes. For now, two levels of TraceLinkType are envisioned: at the higher, more abstract level, several traceability dimensions are defined; at the second level, more detailed categories of traceability dependency may be defined. This is consistent with the metamodel proposed (subTypes and baseType). The present document focuses mainly on the first level. Work has already been done, in the team, on categories for the second level and will be alluded to in this chapter.

2.2 Objectives Guiding the Classification of Traceability Dependencies

Several objectives and decisions guided the definition of the classification scheme for traceability links. They are listed here to help understanding the result.

First, two main objectives have been considered for the traceability links. Traditionally, traceability is used to help software maintenance, for example, when doing change impact analysis or to help understand the constraints that played a role in the design of a particular artefact. The second objective is Software Configuration Management, which implies traceability between two versions of a same artefact, usually called “Time variability” [PBvdL05, p.65–67]. Note, however, that the example proposed in [PBvdL05, p.65–67] is a particular type of variation in time as it does appear to refer to the creation of new variants over time for a given variation point. Software Configuration Management would also need to deal with traceability over various versions of a given software artefact with apparently little differences. A typical example would be the case of a small bug correction that does not change the general behaviour of the component.
2.3 Four Dimensions of Traceability

Traditional software engineering (e.g. [Gra06, p.59], [Pfl05, p.526]) defines two forms of traceability: vertical and horizontal. Vertical traceability refers to relationships along the development process: from requirements to models to implementation. Horizontal traceability refers to relationships between artefacts at the same level of abstraction: between related requirements (a change in one could imply having to change the other one), between models, between software components, etc. Unfortunately, all authors do not agree on the di-
Figure 2.1: Representation of the two traditional dimensions of traceability in Software Product Line Engineering.

rection of the axes. Some (e.g. [Pfl05]) consider that horizontal traceability is between higher and lower abstraction levels and vertical traceability within a given abstraction level.

Note that considering the conventional representation of SPL engineering (see, for example, figure 1.7 in Section 1.4), the definition of vertical traceability would actually correspond to the horizontal dimension in the figure (see also figure 2.1). It would occur either within Domain Engineering or within Application Engineering. In the figure, horizontal traceability would occur within the intersection of any two boxes (e.g. between two requirements of the Problem space in Application engineering).

In this initial framework, Software Product Line Engineering introduces a third dimension, orthogonal to the two other ones, for the following reasons: the management of variability, in itself, does not differ fundamentally from more traditional software development in the sense that it corresponds to specifying the system at an abstract level using appropriate elements (requirements, variation points, etc.) and then progressively refining it to less abstract elements (usually called components in SPL engineering). In this respect, vertical and horizontal traceability apply as usual with the introduction of new elements (related to variability). However, SPL differs from traditional software development in the introduction of the domain, versus application engineering (see figure 2.1). Relationships between artefacts from application engineering, to their counterparts, in domain engineering, do not fit in the two traditional traceability dimensions:

- Horizontal traceability relates artefacts of the same level of abstraction. However, artefact in application engineering would be more concrete than their counterparts in domain engineering.

- Vertical traceability relates low abstraction level artefacts that realize high abstraction level artefacts. However, artefact in application engineering essentially represent the same things as their counterparts in domain engineering, and only make choice that are left open in the latter.

The proposition is to call these three dimensions X, Y, Z in reference to the horizontal and vertical traceability and a third, orthogonal, dimension. X would
be the horizontal traceability axis, among artefacts at the same abstraction level and inside one of Domain or Application engineering (any intersection of two boxes in figure 2.1); Y would be the vertical traceability axis, from higher abstraction levels to lower abstraction levels (inside the white boxes of the figure); and, Z would be the software product line traceability axis, for the relationships between domain and application (within the gray boxes in the figure).

Since dealing with configuration management is also a goal of the taxonomy (Section 2.2), it will include a fourth dimension (called T for time) for relationship between the various versions and revisions of a given artefact.

## 2.4 Examples of Detailed Traceability Relationship

To help understanding the four dimensions of traceability, some examples are discussed here. Each of the four axes will be considered in turn.

The X axis is one of the two traditional traceability axes. Traceability links on this axis relate artefacts at a given level of abstraction. For example, two closely related requirements. A class model and a sequential model (or a state model) realizing a particular requirement (in the form of a use case) would also be related by this kind of link.

If there is a need to work at a finer level of details, for example, at the level of individual classes, one will consider that all relations that happen in a model or a document should be considered on the X axis. For example, if we want to consider independently the different possible scenarios of a use case, they would be related by an horizontal (X axis) traceability link. The dependence between a use case and its scenarios would also be considered an X traceability link.

In the specific case of Software Product Lines, the traceability between a variation point and its possible variants would be on the X axis. One may also propose that the initial variability model, created from requirement analysis, is at the same abstraction level than the requirement and would be linked to it by an X traceability link.

The Y axis is the other traditional traceability axis. Traceability links on this axis relate artefacts at different level of abstraction, typically artefacts that are produced at different stages of the software development process. Examples may include a link between a use case formalizing a requirement, a design model realizing a use case, or a software artefact implementing a model. Other examples of Y traceability links may be found in [KGGR08].

For Software Product Lines, one may cite the dependence between internal and external variability. For example, Pohlet et al. [PBvdL05, p.69] state that “internal variability often emerges when refining or realising external variability.” This is an Y traceability relationship.

The Z traceability axis was added to link artefact at the application engineering level to their counterparts at the domain engineering level. It is essentially a matter of “variability reduction”: a given application makes choices between the various variants offered at each variability point. One must understand that
this kind of link is bidirectional: an application makes some choices among the
variant proposed in the domain model, but it may also introduce new variants or
modify existing variants.

Finally, the T traceability axis links between themselves different versions of
a given artefacts. This is the well known Configuration Management issue. It is
treated in length in Chapter 4 of this document.

There are also interconnections between the different axes. For example, a
vertical or horizontal traceability links may change or disappear between two
versions of an artefact. This suggests that one may talk about versions of X
and Y traceability links. That is to say, X and Y traceability links are subject to
evolution over time and therefore may be themselves related by T traceability
links. X and Y traceability links may also be subject to Z traceability. That
is to say that if two artefacts have a vertical or horizontal traceability link in
the domain model, and if both appear in the corresponding application model,
then they should exhibit the same vertical or horizontal traceability link in the
corresponding application model.

Note that the opposite does not seem to be possible, for example, it is hard
to imagine an horizontal (X) traceability, between two T (Configuration Manage-
ment) traceability links. In this sense, one may consider that the T and Z dimen-
sions dominate the X and Y dimensions, and that the T dimension dominates
the Z dimension.

### 2.5 Templates for Documenting Trace Information

In this section, we will define a template for documenting trace information. This
template will facilitate the documentation and recording of the trace connections
for model driven, aspect oriented, product line systems. We plan to detail fur-
ther the categories of traceability links (e.g. as proposed in [KGGR08]). The
template will also help verifying whether these fine grained dependencies hold,
whether they sufficiently represent trace connection in real life, and whether
there is a need to further extend and elaborate them. Finally, the template can
be used to populate the traceability meta-model defined in Chapter 1.

The template records the observations that are gathered during the analy-
sis phase of the case studies to elicit trace information. They allow identifying
dependencies among requirement specifications, architectural elements, and
implementation, to facilitate the forward and backward tracing of variation. They
also allow analyzing whether the requirements and the architecture of a product
line are being adapted with or without modifications for all other products. This
template will be further extended for task 4.2.4 to analyse configuration man-
agement and evolution in Software Product Line. It should be reduced, later, to
relevant and useful information for the framework.

Table 2.1 describes the template for documenting the information.
Table 2.1: Templates for documenting trace information in model driven, aspect oriented, product line systems

<table>
<thead>
<tr>
<th>Field</th>
<th>Possible Values</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sources</td>
<td>TraceableArtefacts</td>
<td>Artefact(s) originating the link. For example, a link “realizes” will originate from low abstraction level artefacts such as classes. Keep in mind that traceability links are bidirectional.</td>
</tr>
<tr>
<td>Targets</td>
<td>TraceableArtefacts</td>
<td>Artefact(s) receiving the link. For example, a link “realizes” will target a high abstraction level artefact such as a model or a requirement. Keep in mind that traceability links are bidirectional.</td>
</tr>
<tr>
<td>Dimension</td>
<td>X, Y, Z or T</td>
<td>See description in section 2.3.</td>
</tr>
<tr>
<td>Nature</td>
<td>More detailed cate-</td>
<td>Research as [KGGR08] proposed a fine grained classification of traceability links. Such categories allow to better specify inside a traceability dimension what is the nature of link.</td>
</tr>
<tr>
<td></td>
<td>gories inside the di-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mensions of trace-</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ability</td>
<td></td>
</tr>
<tr>
<td>Trace context</td>
<td>A text</td>
<td>Description of reason for the traceability (see meta-model in Chapter 1).</td>
</tr>
<tr>
<td>Creation</td>
<td>“automatic” or “man-</td>
<td>How was the link created? The idea is that automatically created link should be maintained automatically. One may also expect less details (e.g. missing optional rationale) in automatically created links.</td>
</tr>
<tr>
<td></td>
<td>ual”</td>
<td></td>
</tr>
<tr>
<td>Author</td>
<td>A person or tool</td>
<td>Name of the person or tool who/which created the link</td>
</tr>
<tr>
<td>Version</td>
<td>A number</td>
<td></td>
</tr>
<tr>
<td>Date/Time</td>
<td>Date and time</td>
<td>Date and time of last modification</td>
</tr>
</tbody>
</table>
Table 2.2: Initial documentation of a traceability link according to template in section 2.5

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sources</td>
<td>Quotation Management component</td>
</tr>
<tr>
<td>Targets</td>
<td>Product Management Components and Account Management Components</td>
</tr>
<tr>
<td>Dimension</td>
<td>X horizontal traceability</td>
</tr>
<tr>
<td>Nature</td>
<td>-</td>
</tr>
<tr>
<td>Rationale</td>
<td>The dependency of the quotation management component to account and product management components: the Quotation Management wizard collects data by interacting with the account management component to retrieve all available business partners on the customer data, and secondly queries the product management for all registered products. If these two components were not available due to a certain product configuration, the quotation management wizard would have to be altered.</td>
</tr>
<tr>
<td>Creation</td>
<td>Manually</td>
</tr>
<tr>
<td>Author</td>
<td>Birgit Grammel</td>
</tr>
<tr>
<td>Version</td>
<td>0.1</td>
</tr>
<tr>
<td>Date/Time</td>
<td>28/02/2008</td>
</tr>
</tbody>
</table>

2.6 Trace Information Examples

Two examples of links documented with the template are presented. They come from the Sales Scenario case study revolving around Quotation Management, and were provided by SAP. Some background on Quotation Management will be provided before the documented links are presented in Tables 2.2 and 2.3.

After subsequent evaluation and go/no-go decision handling by sales management, an employee of the sales office creates a quotation (i.e. offer) by using quotation management functionalities. Accordingly, a quotation template is configured with details of the given sales opportunity, including prospect (customer) details and the products to be offered. Based on categorisation of the prospect in a customer group, estimated sales volume, and sales probability, resulting in an overall customer rating, an individual pricing strategy is used to calculate a discount for the customer. Once the quotation is complete, it is sent to the potential customer by e-mail or letter. In case of positive response on the quotation, an order is created.

Quotation Management has to be bundled with appropriate architectural parts, providing necessary data structures for persistence and in memory storage, user interface parts allowing the user to manipulate quotations, as well as process descriptions, e.g. to define stepwise workflows for creating and editing quotations. While the different parts themselves depend on each other (a concrete quotation attribute may have a certain representation in the user interface), they may also interact with architectural parts of other components. For instance, the dependency of the quotation management component to account and product management components: the Quotation Management wizard col-
<table>
<thead>
<tr>
<th>Field</th>
<th>Possible Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sources</td>
<td>(respectively)</td>
</tr>
<tr>
<td></td>
<td>1. Data model</td>
</tr>
<tr>
<td></td>
<td>2. Data interaction model</td>
</tr>
<tr>
<td></td>
<td>3. Dialog model</td>
</tr>
<tr>
<td></td>
<td>4. Presentation model</td>
</tr>
<tr>
<td></td>
<td>5. Data context model</td>
</tr>
<tr>
<td>Targets</td>
<td>(respectively)</td>
</tr>
<tr>
<td></td>
<td>1. EJB Entity Bean Class</td>
</tr>
<tr>
<td></td>
<td>2. Persistence Manager calls in EJB session beans</td>
</tr>
<tr>
<td></td>
<td>3. JPDL Page flow Definition</td>
</tr>
<tr>
<td></td>
<td>4. JSF pages (UI)</td>
</tr>
<tr>
<td></td>
<td>5. JBoss Seam conversation annotations and navigation rules</td>
</tr>
<tr>
<td>Dimension</td>
<td>Y vertical traceability in Domain Engineering, as well as Application Engineering</td>
</tr>
<tr>
<td>Nature</td>
<td>Linktype: “is generated”</td>
</tr>
<tr>
<td>Rationale</td>
<td>-</td>
</tr>
<tr>
<td>Creation</td>
<td>Automatically</td>
</tr>
<tr>
<td>Author</td>
<td>Birgit Grammel</td>
</tr>
<tr>
<td>Version</td>
<td>0.1</td>
</tr>
<tr>
<td>Date/Time</td>
<td>28/02/2008</td>
</tr>
</tbody>
</table>
lects data by interacting with the account management component to retrieve all available business partners on the customer data, and secondly queries the product management for all registered products. If these two components were not available due to a certain product configuration, the quotation management wizard would have to be altered.

2.7 Conclusion and Issues

We have defined categories of dependency for model driven, aspect oriented, product line systems to trace variability in forward and backwards direction across software life cycle. These dependencies will be verified in cooperation with the industrial partners. This will lead to analyze if the taxonomy well represents the trace connection across the software cycle, if there is a need to refine these dependency, or to introduce more dependencies.

At the moment we do not have a complete architecture for both smart home system and sale scenario case study. There is still a need of more detailed architecture for both case studies.

Some issues remain open:

- The categories are at a very abstract and general level. To be of interest, more specific categories need to be identified. Some work was done at University of Lancaster in this direction and has been partially published [KGGR08]. It needs to be integrate with the present proposal.

- Automatic tools could to be defined to help populate the repository with traceability links

- Automatic tools could be defined to process the traceability links and pro-actively assist developers in their tasks.

- Real traceability scenarios exercising the template (and the metamodel) need to be defined.
3 Introducing Uncertainty Modelling in SPL Process

3.1 Introduction

The design of software systems is a very complex activity, which requires a considerable amount of information to be done effectively. Ideally, the required information must be of perfect quality, i.e. clear and accurate, since it will be used in decisions that determine the design of the system. However, in practice it has proved to be very difficult to define or attain accurate information when it is required. As a result, the design activities generally are performed with descriptions that only partially provide the information with the desired quality. The usefulness of information can therefore be limited as a result of ambiguities, incompleteness and vagueness among others. We refer to such information as *imperfect information*.

In the previous chapters, an effort has been made to define a metamodel that captures the typical elements needed for tracing the design activities of software product line development. In order to address the consequences of uncertain information and accommodate it accordingly in tracing information that is made based on the traceability metamodel, we present a metamodel that conceptualises software engineering rationale with uncertainty modeling. This model can be adopted in rationale management aligned with traceability in SPLE in the presence of uncertainty.

3.2 Uncertainty during Software Product Line Development

An imperfect description of what is expected from a software system is fairly common and software development has a number of solutions for this problem. In the case that the variability of the context in which the system will be used is known, the design can be fitted with mechanisms to handle the variability. Software product lines are explicitly aimed at handling a large variety of products effectively in order to supply volatile market demands. By isolating the common elements of a family of products and supporting variability with variability points and mechanisms, the entire product family can be enabled effectively.
3.2.1 Locations of Uncertainty during Software Product Line Development

In the previous section, we have established that like traditional software design processes, the development of software product lines can suffer from information that contains uncertainty. As a result of the uncertainty in the information that is used, the design of the product line can become less effective in addressing market demands for the intended set of products. As a further complication, uncertainty can occur at many different locations during the development of software product lines. In each of these locations, the nature and impact of the uncertainty will differ, which means that it is vital to accurately understand the causes and consequences of such information. In the following, we briefly examine two of the most prominent locations of software product line development where the occurrence of uncertain information can have a considerable influence on the resulting product line architecture.

First we examine the influence of uncertain information on the variability analysis. The success of a product line architecture largely depends on the extent to which the potential variability is known or can be predicted accurately. The scoping and variability support that is considered in the product line directly reflects on how well this variability can be managed. If at a particular point in time, the requirements stretch beyond the capabilities of the implemented mechanisms, the initial information on the expected variability was not accurate enough. As a result, the product line might need considerable modifications to provide the required product configurations.

A second location where uncertainty can manifest itself during product line development is during the assessment of design decisions. The development of product lines is comprised of a large number of design decisions. Typically, for each design issue several candidate solutions or design alternatives are considered. These design alternatives are then evaluated based on the functional attributes and quality expectations, in order to establish an ordering among them. The design alternative that offers the best characteristics is then selected to fulfill the design issue under consideration. However, the evaluation of such design alternatives is typically performed in the early stages of the design process, a time at which the software engineer only has a partial, abstract view of what the resulting system will be. This is even amplified for software product lines, since the effectiveness of the decisions can only be assessed after the completed system is used for the configuration of actual products. As a result, it is very unlikely that the evaluations of the design alternatives are based on accurate information. Expectations about properties of the completed system are therefore naturally subject to uncertainty, which can influence the accuracy of the selection of design alternatives. Nonetheless, this uncertainty is not recognized, captured and considered during the evaluation activity, therefore it is likely that the established ordering of the candidate solutions does not reflect an ordering based on the actual resulting system characteristics.

Within AMPLE, the focus of handling uncertain information during software product line development is on design decisions, and in particular on the design
rationale that is used to resolve them. By identifying the uncertainty that exists in design rationale, modelling and considering it accordingly in the decision process, its negative influence can be minimized and its impact on the development of the software product line understood.

3.2.2 Models for Describing Uncertainty in Design Decisions

A design rationale is naturally subject to uncertainty, as it reasons about properties of the system under design for which the underlying structure is not yet completely known. In the literature on design rationale, the existence of uncertainty is acknowledged and analyzed, for example in [Leh05]. As a result, approaches that aim to describe design rationale using various paradigms have been proposed, ranging from natural language to quantified reasoning approaches.

Within AMPLE, we aim to support reasoning with rationale information in particular when this information is subject to uncertainty. As the models we have applied to model and reason with uncertainty are based on probability theory and fuzzy set theory, we have chosen to extend on approaches that quantify rationale information. However, as the concepts of uncertainty modelling and reasoning can be applied to a multitude of quantified rationale description approaches, we will first describe the models that will be used for uncertainty.

During software product line development, a large number of design decisions must be resolved. Typically, for each design issue several candidate solutions or design alternatives are identified and considered. The rationale for selecting one particular solution over all the other alternatives is assumed to be a single number, based on which an ordering amongst the alternatives can be established. This single can represent, for example, a result from a computation based on metric values, such as response times, cohesion and complexity. The single number then becomes a representation of the overall quality. However, the evaluation of design alternatives is typically performed in the early stages of the design process. At this point, the software engineer only has a partial, abstract view of what the resulting system will be. As a result, the value for the rationale can not be calculated using accurate measurements, but only estimated or approximated at best. Rationale based on expectations of properties of the completed system are therefore naturally subject to uncertainty. Nonetheless, in most quantified models, the rationale (and its components) is expressed using traditional numerical expressions. Further, these numerical expressions are then used to establish the ordering among the design alternatives, which makes the ordering vulnerable to the unaddressed uncertainty in the quality estimations. For example, the uncertainty about the response times can greatly differ, which means that one alternative carries a larger risk than the other. Typically, this kind of information would lead a different ranking (i.e. the rationale for this alternative should include a consideration with respect to this uncertainty). In order to establish a ranking among design alternatives that accurately represents the understanding of the situation, the nature and severity of the uncertainty should therefore be considered explicitly.
We focus on mathematical representations of imperfect information by means of probability theory and fuzzy set theory. These theories offer a solid mathematical foundation for the specification and manipulation of uncertain information and therefore are natural candidates to extend on quantified rationale models. Since the kind of uncertainty that is described by the models is different, the character of fuzzy sets and probability is orthogonal and complementary. Probability theory is used to describe situations where the results of experiments, when performed under apparently identical circumstances yield different results, such as flipping a coin. For this purpose, probability distributions are used to describe the value for random variables in future points in time. This reflects the situation where the value of the variable is not known at the current point in time, but will be known when some point in the future is reached. With respect to uncertainty in design rationale, the distributions can be used in particular when behavior needs to be described that is itself subject to a continuous random variable, such as response time that depends on a waiting queue.

The second model that is used for modelling uncertainty is fuzzy set theory [KY95], which is an extension of classical set theory. In classical set theory, membership of elements in a set is defined in a binary manner; an element is either a member of the set or it is not. Fuzzy set theory allows elements to be a partial member of a set. With this partial membership, it is possible to describe vague (or in this case uncertain) boundaries of, for example, quantified rationale models. In a fuzzy set, the membership of an element \( x \) in a fuzzy set is given by the membership value \( \mu(x) \), where \( \mu \) is a function that maps the universe of discourse to the interval \([0, 1]\). This value is the degree to which \( x \) is an element of the fuzzy set, where 1 means "completely a member" and 0 means "completely not a member". By considering the degree of membership during manipulations of uncertain information modeled by fuzzy sets, the resulting conclusions explicitly include the uncertainty that was present in the design rationale.

As we have mentioned before, within AMPLE we assume that rationale for design decisions is expressed and modelled using a quantified approach. As a result, the natural candidate for modelling uncertainty in these quantifications using fuzzy sets are fuzzy numbers. Fuzzy numbers are a fuzzy set representation of imperfect (uncertain) descriptions of numbers. In such a fuzzy set description, exactly one element has a membership degree equal to one. Besides this restriction, the fuzzy set can take many different shapes, such as a triangular shape, commonly referred to as triangular fuzzy numbers. As an example, a fuzzy number is depicted in 3.1.

In the picture, a triangular fuzzy set is used to describe a response time of approximately 10 milliseconds. The uncertainty about this value is modelled by including values from 8 to 11 milliseconds, each with their own specific degree of membership. This degree is an indication of how much this value is considered to be the actual response time of the completed system. When this number is used in computations, all the values in the fuzzy set will be considered in accordance with their membership value. Naturally, the operators for manipulating fuzzy numbers should be defined as the traditional operators no longer suffice.
In this section we have introduced two techniques that can be used to extend quantified models for design rationale with support for reasoning with uncertain information. To achieve this, we utilize the extensions for modelling uncertainty in design decisions given in [Nop07]. These extensions cover a considerable aspect of the uncertainty support required, as they define fuzzy set and probabilistic models as well as a number of the required operators. By applying these results in the context of rationale modelling and reasoning, support for uncertainty can be realized.

3.3 Rationale Management

Software development methodologies are concerned with the creation of solutions to meet stakeholders’ needs. The processes and paradigms that are adopted by such methodologies involve a core set of activities, abstractly being the following: the study of user needs; the identification of problems; the specification of a solution domain; and the development of solutions that resolve the problems. As software evolves, this cycle is repeated and previous solutions may be adapted to new requests.

Specifying solutions requires not only taking into account good design practices, but also utilizing techniques for the assessment of candidate solutions, with respect to desired qualities, at significant design steps. Figure 3.2 illustrates some elements on which the rationale for a solution is based, such as invariants, assumptions, design alternatives and associated decisions.
A rationale is defined in [DMMP06] as being the justification behind a decision. Capturing a rationale comprises the elicitation and formalization of tacit knowledge about a decision. The term SER (Software Engineering Rationale) denotes the various kinds of rationale that can be captured and used for software engineering activities [DMMP06]. Specifically, a design rationale is the representation of the reasoning influencing decisions about the design of an artefact. Analogously, a maintenance rationale is used for keeping track of maintenance decisions and their justification. Therefore, depending on the development time at which the decision is taken and on the kind of decision, the term rationale receives a tag that makes it more particular. In this section we adopt rationale as the generic term instead of SER.

The rationale behind decisions in software engineering problem-solving is usually constructed by using some imperfect information as source, which reflects deeply on next steps of the development cycle. For that reason, in this research we propose the management and traceability of a rationale in the presence of imperfect information. Mostly, we are interested in the uncertainty that can exist on the assumptions of the rationale behind relevant software product line engineering decisions.

Potential uses of rationale information include: supporting reuse and change; increasing consistency of decisions; supporting traceability and maintenance; and enabling knowledge transfer. Several approaches exist for rationale management, such as QOC [MYBM91], IBIS [KR70, Rit72] and DRL [Lee91]. A survey of concepts and techniques of rationale management in software engineering is given in [DMMP06].

### 3.3.1 Assumptions

Developers are continuously making the specification of software systems based on assumptions about the application and its real-world [LR02]. Assumptions are made continuously when resolving issues all through various development activities, such as the conception, definition, design and implementation of software systems. However, the process of making the assumptions to justify a design explicit and usable is not trivial. First, because stakeholders and developers tend to provide unstructured, informal, incomplete, imprecise or inconsistent information [Pos97] during the rationale capture process. Second, as a consequence of the previous statement, it becomes difficult to perform proper reasoning about assumptions made a priori. Therefore, it is important to keep track of the assumptions in a structured and formal way, in which they can be recorded, reviewed and controlled.

Assumptions can also express the conditions in which a certain decision is valid. An example classification of design assumptions is given in terms of qualitative or quantitative constraints, as suggested in Figure 3.3. Qualitative assumptions are specified in terms of non-numerical descriptions of constraints, such as "propertyX is small enough", while quantitative assumptions are represented by a numerical form, such as "0 <= propertyX <= 10".
Figure 3.3: Decision modeling with assumptions (adapted from [Pos97])

As a consequence, assumptions are precise or imprecise [AK06]. The meaning of preciseness relates to elements that could be: "strictly described; accurately stated; definite; distinctly defined with no variation; or strictly conform to usage and rules". The imprecision can be interpreted as: "not clearly, precisely, or definitely expressed or stated; indefinite in shape, form, or character; hazily or indistinctly seen or sensed; not sharp, certain, or precise in thought, feeling, or expression; imprecisely determined or know; or uncertain in nature".

In [Leh05], Lehman discusses the role of uncertainty in systems that need to evolve continuously to meet the specified or implicit goals of the real world. This uncertainty can be related to known and unknown changes on the software system (the software application, the environment where it functions, and the domain it involves). Such changes can impact the validity of the assumptions made during the development processes, which are part of the design rationale and many times are not recorded properly.

Detected and controlled assumptions about the alternative solutions are important elements of a rationale. For this reason, we aim at documenting and using during the development process and during software maintenance, the relevant assumptions that are associated to each design decision, as well as their likelihood of change. Moreover, when uncertainty is recognizable in the assumption and can be quantified, we aim at dealing with it as well.

3.4 Rationale Metamodel

In this section we propose a metamodel that conceptualises software engineering rationale with uncertainty modeling. It comprises elements from argumentation-based rationale methods, problem-solution approaches and quality evaluation methods, considering that uncertainty can be present on the assumptions made in the rationale behind decisions. This model can be adopted in rationale management aligned with traceability in SPLE in the presence of uncertainty. In the next section we give an overview of the processes involved and explain how the rationale and uncertainty can be traced according to the traceability framework.

Figure 3.4 presents the rationale metamodel considering uncertainty. The metamodel is based on a problem-solving perspective. An artefact can represent the source of a problem (source artefact) or the realization of a solution
(target artefact). Source and target artefacts can be other decisions, as they are intermediate artefacts.

For each decision, the rationale regards problems and their solution space. The rationale modeling space contains problems, solution alternatives and solutions. A rationale is considered for each decision about a problem. The rationale for selecting a solution for a certain problem is composed of a set of alternatives and a set of arguments or assumptions for quality features, which can be used for evaluating the alternatives and make the decision between them. Each rationale can be made by using arguments (in favour or against) and/or quality features (quality attributes and/or quality indicator expressions).

The values assigned to the quality features are assumptions that are subject to uncertainty. These assumptions can represent, for instance, estimations related to how each alternative is evaluated for each quality feature. Since we are interested in quantifiable assumptions, we are proposing the representation of it in terms of crisp, fuzzy or probabilistic assumptions. However, the model can be extended to support qualitative assumptions as well, by means for instance of the use of linguistic values.

The core elements of the design rationale model and those that arise from problem-solving approaches [TA06] are defined as follows:

- **Decision**: a design decision is a step towards the resolution of a design problem, on which a rationale is used to choose a solution from possible
alternatives.

- **Problem**: a problem is a design issue that needs to be solved.

- **Solution**: a solution represents an alternative that has been chosen according to the design decision rationale.

- **Implication**: an implication is a summary of actions that need to be taken into consideration if the alternative is chosen as the solution for the problem. These actions could be, e.g., application of modeling rules to the source artefact.

- **Rationale**: a rationale is the justification behind a decision [DMMP06]. Rationale is created when a relevant decision about how to solve a design problem is made.

- **Alternative**: a design alternative is the description of a feasible candidate solution for solving the a problem.

- **Context**: a context consists of the surrounding conditions in which the problem-solving decision rationale is to be interpreted.

For making the decision, every rationale has an **overall judgment** that is based on arguments or on assumptions about quality features and quality values of problems and alternative solutions. Under the perspective of quality-oriented software engineering [Aks04], we assume that the following elements are necessary for the rationale model to cope with the representation of the rationale made according to qualitative evaluations:

- **Assumption**: an assumption is a statement that is assumed to be true about properties of an alternative, such as an alternative quality feature.

- **QualityFeature**: a quality feature is a property that should hold or condition that should be satisfied giving desired and measured quality values. Problem quality feature and solution quality feature should be comparable with each other [Aks04].

- **QualityAttribute**: a quality attribute is a quality feature which can be expressed in terms of non-functional requirements, such as usability and modularity.

- **QualityIndicatorExpression**: a quality indicator expression indicates a more specific measurement of the quality attribute. For instance, expressions involving metrics for determining how modular a design is.

- **QualityValue**: a quality value is a number or symbol assigned [Aks04].

From argument-based approaches [KR70, MYBM91], the design decision rationale model accommodates the representation of statements about alternatives:
- **Argument**: an argument about an alternative is a relevant statement for its judgment in comparison with other alternatives inside the rationale context.

- **Pro**: a *pro* argument is a virtue of an alternative or a positive statement about it.

- **Con**: a *con* argument represents a disadvantage of an alternative or a negative statement about it.

The proposed model conceptualizes the software engineering rationale we are interested in. We consider the capture, interpretation and use of the rationale important across the four main product line engineering processes. The domain engineering rationale can be used for deciding about trade-offs of which properties of the domain are in or out the product line. The application engineering rationale, on the other hand, can be used in various design steps related to both core and custom assets. Decisions about market demands, costs and scheduling regard the product management rationale. At last, the variability management rationale is about decisions about scoping the variability and changes on the variation mechanisms.

At any of these processes, some interesting queries can aim at: finding the problems a certain alternative refers to; identifying the context in which the rationale can be reused; relating design problems to design solutions in SPLE and therefore discover design patterns; identifying the assumptions (with or without uncertainty) made *a priori* in terms of estimations about a solution for the sake of comparing such values with measurements made *a posteriori*; and so on.

### 3.5 Tracing Rationale

Traceability practices should help stakeholders, developers and maintainers with the understanding, capturing, tracking and verification of software artefacts and their relationships and dependencies with other artefacts during the software life cycle. Under a general perspective on software development, traceability of decisions is an important and relevant issue. Capturing and storing the rationale are means already of tracing decisions. However, traceability should serve for linking designs to their rationale. It may help on linking designs to justifications, important decisions and assumptions behind them [RJ01].

We would like to recall that both rationale and trace information are only helpful when they fit to be used for a certain purpose. As an example, rationale management combined with traceability management can facilitate change impact analysis and understanding the causes of a certain impact. Moreover, assessments are usually subjective and assumptions eventually may become obsolete. Tracing the rationale of decisions improves the understanding of the important contextual factors that have major impact on the quality of the software.
The following are important parts to consider when making use of traceability approaches on the evaluation of design alternatives:

- Capturing decision dependencies by means of traces between traceable artefacts and design decisions (based on the assessment of a stakeholder or based on rules). These decisions should be structured in a model and explicitly related to artefacts built during software development steps.

- Capturing context dependencies in traces relating the explicit representation of context to other artefacts (development artefacts, decisions, variation artefacts,...). A context model should capture the surrounding conditions of an event (e.g. a decision, a trace, a realization) which can affect its interpretation [Eks99]. Tracing contexts is necessary to support design processes, traceability and evaluation approaches, since correct decisions depend on the contexts they are embedded into [McG05].

In addition, it is essential to identify in which context an alternative has been chosen and which kinds of relevant decisions are natural to occur and necessary to be understood in the development process. The software development approach under research in the AMPLE project in its essence is driven by the following three areas: (i) model evolution and transformations, (ii) variability specification and (iii) separation of concerns.

- **Model evolution and transformations**. Decisions in model-driven engineering usually involve the evolution of metamodels or models and model transformations. In this approach, traces can be generated by semi-automatic transformations based on metamodels. The rationale of decisions is usually embedded into model transformations.

- **Variability specification**. The specification of high-variable software influences the design decisions involving managing complexity of alternative mapping of features.

- **Separation of concerns**. Good separation of concerns should facilitate modularization, reuse and comprehension of the software system or its parts. Design decisions may address the decomposition of the system parts using practices of separation of concerns, that are rich in constraints to what regards proper decomposition.

The variety of traceability links between software artefacts with respect to these three approaches for software development have in common the need of performing rationale management integrated with traceability management. In the next subsections we present an overview of the processes involved and our approach for accommodating rationales in the traceability metamodel.
3.5.1 Traceability and Rationale Management in SPL

A general view of the processes involved in tracing uncertainty during software product line development is shown in Figure 3.5. We consider these processes distributed across three layers.

In the **Software Product Line Engineering (SPLE)** layer, we can observe four important processes: **Domain Engineering**, **Application Engineering**, **Variability Management** and **Product Management**. Activities of each of these processes depend on decision making regarding how to engineer the software product line and achieve the goals of attending the time-to-market and optimizing the software production costs. SPLs are dependent on market demands and on the context in which decisions concerning the software products development are made during the production phases, as well as on a proper software specification and variability management. In this way, the SPLE processes provide inputs for **Rationale Management** and **Traceability Management**. A design state-space is maintained and structured based on design modeling rules and a design data model.

![Figure 3.5: Traceability and rationale management in SPL](image)

The **Traceability Management** layer involves all processes of a traceability analysis framework. These processes, such as **Trace Capture**, **Trace Interpretation** and **Trace Analysis**, are core for providing good traceability of SPL artefacts. Traces can be collected from and provided to many activities of the SPLE processes. A trace state-space is maintained and structured based on a trace model and trace rules.

The other layer concerns **Rationale Management** issues. **Rationale Capture**, **Rationale Interpretation** and **Rationale Use** are the processes involved in a classic approach for rationale management. In addition to this, we consider the capture and incorporation of the uncertainty that is manifest in parts of the rationale. The **Uncertainty Capture** process regards these points and enriches the captured design rationale. Such information about the rationale can be useful for many purposes, for instance for optimization of design decisions and risk analysis. The rationale is recorded into a rationale state-space and a rationale...
model is necessary for the conceptualization of the design decisions taken during SPLE.

### 3.5.2 Rationale in the Traceability Metamodel

For achieving the goal of tracing in the presence of uncertainty we need to trace the rationale about design decisions made across the SPLE processes. We propose considering a decision as a traceable artefact. Therefore, the structure of the current traceability metamodel does not need to change for accommodating rationale capture. In this way it is possible to trace decisions to other traceable artefacts, which includes other decisions. As a consequence, there is a connection between the rationale state-space and the design state-space also by means of the trace state-space.

The rationale behind each relevant decision is represented by an instance of the rationale metamodel introduced in section 3.4. It carries the representation of uncertainty in terms of non-crisp assumptions made for taking the decision between alternatives to find a solution for a design problem. Traces between this decision and other artefacts will indicate that changes on the assumptions may have some impact. On a fine grained level, traces between elements of decisions will indicate dependencies that concern rationale management issues, such as the decomposition of problems or the evolution of the rationale.

![Figure 3.6: Traceability between decisions and other artefacts](image)

By using the structure of the traceability metamodel, each trace link connects a source artefact to a target artefact. In the case of the rationale behind a design decision, when tracing artefacts to a decision we consider that these source artefacts to be the ones the problem the decision addresses is identified from. Tracing a decision to artefacts implies that these targets are artefacts that refer to the realization or the refinement of the solution. Figure 3.6 illustrates examples of traceability relations among decisions and artefacts. We can distinguish a number of cases when tracing the rationale:

---

1 This model is inspired on the schematic model of design deliberation and artefact synthesis proposed by Potts and Bruns [PB88] and on the model dependency descriptor proposed in [CLV92]
Decisions can be source or target artefacts of a trace link. Multiple dependencies can exist between a decision and other traceable artefacts. The source artefacts must be traced to the decision and the decision to the target artefacts of the design step. For instance, decision $D_a$ traces backward to both artefacts $A_1$ and $A_3$ and forward to $A_5$ and $A_8$.

If it is relevant and necessary to record decisions that concern a single candidate solution, the same applies to the creation of trace links. In this case, however, the rationale model for the decision can be a simplified version of the rationale metamodel, adjusted for the needs of the kind of information that needs to be recorded about the decision and the problem it involves.

Decisions related to other decisions also need to be traced explicitly. This may allow us to observe also the impact of changes on other decisions and, as a consequence, on other artefacts. Moreover, tracing the connections between decisions as well as their history improves the understanding of the software system development. The traces between $D_b$ and $D_g$ or $D_c$ and $D_j$ are examples of this situation.

When there are no multiple candidate solutions to be considered and no decisions to be recorded, a source artefact can be traced directly to a target artefact. This could be done, for instance, simply by means of a derivation relation, as in [CLV92]. The traceability metamodel supports various trace link types. Therefore, the semantics of each kind of trace link can provide the basic insight about the existence of the relationship between source and target. In Figure 3.6, trace links can be found between $A_1$ and $A_2$ and between $A_{11}$ and $A_{12}$.

### 3.5.3 Related Work

#### Design Patterns

The description of knowledge about solutions design, the context in which they can be applied and the rationale behind them is an issue treated widely in the area of design patterns [GHJV95]. However, design patterns are concerned with the description of the design rationale per solution, without representing the decision rationale. Our approach aims at making it possible to combine both design rationale and decision rationale, by being flexible on considering design patterns as possible alternatives that can be evaluated during the design decision process.

#### Rationale Management in SPL

A rationale-based approach for variability management of SPL at requirements engineering level is proposed in [Thu07]. Thurimella presents an approach for combining the QOC [MYBM91] model with respect to elements of an orthogonal
variability model. The authors apply the QOC method, including justifications to decisions taken while managing variability at this level. Two applications of QOC are shown. The first is its use for rationale-based variability constraint, in which knowledge regarding constraints among variation elements is elicited, evaluated and justified using some criteria. The second is its use for rationale-based product instantiation, which aims at facilitating product instantiation by evaluating different alternative configurations using justification matrices and considering, for instance, the particular criteria of a customer. In this approach, the assessment of the options is made by using positive or negative contributions to each criterion. This kind of value system is very abstract and does not take uncertainty in consideration in an explicit way. Argumentation is also made in a textual form. The application of this approach is limited to early stages of SPL development and the approach does not cope with evolution of the design rationale.

**Tracing Design Rationale**

A reference model for traceability of design rationales during requirements management is presented by Ramesh and Jarke [RJ01]. It gives support to the detailment and explicit recording of the rationale behind traceable objects. The authors associate the rationale to each traceable object and consider the decisions that can affect it. Alternatives considered, arguments opposing or supporting these alternatives and assumptions for making the decision are elements of this model. This model shows a way in which the design rationale of a traceable object can be traced. The model supports recording the alternatives that were not chosen but considered for the decision making. Basically, most part of the elements of the rationale model depend on assumptions. Figure 3.7 shows the proposed model.

![Figure 3.7: Rationale model for requirements traceability [RJ01]](image)

In [TJH07], a rationale-based architecture model and traceability techniques
for performing root cause and change impact analysis over architectural rationales and architectural elements are presented. The model comprises the representation of design rationales by considering the artefacts and their relationships. Traces are captured between architecture elements and architecture rationales and different kinds of information can be stored about the architectural design decisions, such as quantitative or qualitative reasoning. Differently from the previous related work, the authors show a practical application of their approach to a real case study. However, the approach does not consider any kind of uncertainty in the assumptions made during architectural design and the authors do not make clear how quantitative reasoning is represented.

### 3.6 Conclusions

We have presented in this section a proposal of a metamodel for conceptualizing software engineering rationales. It comprises elements from argument-based rationale methods, problem-solution approaches and quality evaluation methods, considering that uncertainty may be present on the assumptions made in the rationale. The rationale metamodel accommodates uncertain information in terms of fuzzy and probabilistic assumptions. It captures the main elements necessary to represent the rationale behind decisions, which can be considered as traceable artefacts. This model can be adopted in rationale management aligned with traceability in SPLE in the presence of uncertainty.

Future work consists of developing an approach to ensure that the rationale management processes are performed properly and are aligned with the traceability analysis framework. We also aim at applying it in domain and application requirements engineering, and observe the impact of invalidation of assumptions in the next development phases of product line engineering. Moreover, architecture quality evaluation methods for product lines will profit from this, while we will investigate the influences of the rationale and uncertainty in architectural metrics and trade-off analysis. Tooling support is expected to be provided in the next deliverable of work package 4.
4 Configuration and Evolution Management

This section elaborates on configuration management and evolution of software product lines, as part of the investigations performed in task 4.2.4. We have investigated the conceptual foundations of configuration management and evolution of software product lines. In addition, we elaborate on the specific requirements of the different work packages, in the AMPLE project, regarding configuration management. Moreover, we report in this section the practices and approaches for CM and evolution of SPLs adopted by the industrial partners of AMPLE. Finally, we present a framework for traceability that addresses the traceability in configuration management in general as well as configuration management in a model driven and software product lines context.

4.1 Conceptual Foundations

In this section we describe the conceptual foundations of software evolution and configuration management and discuss the application of these concepts in the context of software product line engineering.

4.1.1 Software Evolution

The concept of software evolution was first identified by Lehman in the late 60s [Leh69]. Ramil and Lehman [RL00] define software evolution as "all programming activity that is intended to generate a new software version from an earlier operational version". According to [LR01], "software evolution processes, including those of software change, encompass all activities required to maintain stakeholder satisfaction over the operational life of a software system. The objectives of such activities include fixing, adaptation, and enhancement of the software".

One of the main activities of software evolution is the maintenance of software, which may be categorized into the following categories:

- **Corrective maintenance**: Modifications of a software product performed after the delivery to correct discovered faults [LS80].

- **Perfective maintenance**: Implementation of new functional or non-functional requirements to improve performance and maintainability [LS80].
- **Adaptive maintenance**: Changing software to reflect a known change in the software environment [LS80].

- **Preventive maintenance**: Acting to prevent known problems in the future, e.g. refactoring [LS80].

- **Speculative maintenance**: This form of maintenance arises in web-based applications and can be seen as the effort of checking for broken links, which may trigger one of the other forms of maintenance [WBM99].

Regularities in software evolution have been identified by Lehman [Leh96] and are referred to as the Lehman’s laws of software evolution (based on the evolution of IBM 360 mainframe OS over a period of 30 years):

- Continuing change: Due to evolution of the system environment, software has to be continually adapted.

- Increasing complexity: During program evolution its complexity increases, unless work is done to maintain or reduce it.

- Self regulation: Program evolution is a self-regulating process. System attributes such as size, time between releases and the number of reported errors is approximately invariant for each system release.

- Conservation of organizational stability: The average effective global activity rate on an evolving system is invariant over the product lifetime.

- Conservation of familiarity: Over the lifetime of a system, the incremental change in each release is approximately constant.

- Continuing growth: Functional content of a program must be continually increased to maintain user satisfaction over its lifetime.

- Declining quality: Software will be perceived as of declining quality unless rigorously maintained and adapted to a changing operational environment.

- Feedback system: Programming processes constitute multi-loop, multi-level feedback systems and must be treated as such to be successfully modified or improved.

**4.1.2 Software Evolution in Software Product Lines**

In the context of software product lines the evolution mechanisms presented in the previous section, i.e. maintenance activities and the Lehman’s laws, still apply in general. However, software product lines pose additional challenges regarding their evolution. The benefits of software product lines come from the reuse of core assets, which express the commonality of the product line. In addition, the approach features concrete products, which are derived from a combination of core assets with custom assets that apply to a single application
product or a small group of products. The following key challenges have to be met, in order to preserve the benefits of the software product line approach:

- **Declining quality.** As with single application software systems, the product line is subject to declining quality unless measures are taken to preserve the quality of the product line. Therefore changes to software artefacts need to be kept track of, to avoid the deviation of core assets from the general architecture, or even disconnection between assets and products. Both cases may be referred to as "decaying" of software product lines [YR06].

- **General maintenance:** All other forms of maintenance that were discussed in the previous section hold the same challenge as the application of corrective maintenance, in the sense that every maintenance activity may impact a large number of products. In contrast to corrective maintenance the other maintenance activities may entail an evolution of the products and the product line as a whole. In general, all maintenance activities that are not corrective must not be reflected in currently deployed products and the resulting product line and products represent successor versions of currently deployed products. In particular, perfective maintenance may result in changes that add new variability, resulting in the inception of new products. These changes require careful planning in order to reduce the risk of product line "decay".

- **Corrective maintenance:** In contrast to single application systems, corrective maintenance may impact several application products. We can discern between maintenance to core assets and custom assets, where the maintenance on core assets potentially impacts a higher number of products, while maintenance to custom assets only affects a smaller number. In any case, corrective changes need to be evaluated carefully in order to ascertain the applicability of the performed changes to all involved products.

### 4.1.3 Configuration Management

Configuration management (CM) is a management discipline defining organizational processes and mechanisms governing the entire product lifecycle, covering development, building, deployment, maintenance and the evolution of a system. In particular, CM deals with changes and variations occurring during the system lifetime, striving for maintaining and reconstructing product versions and changes. CM practices usually focus on building and documenting versions and variants of systems (products), collaboration during their development, the management of changes and their evolution.

CM is frequently defined as the control of the evolution of systems [BGD96, Dar91, EDA97, Est00]. Thus, a software configuration consists of all documents, plans, and program sources necessary to reproduce a software product. Consequently, the process of software configuration management tracks
all changes made to the information managed and allows for reproducing the software product as it was at any point in time during its creation, reporting on the status, completeness and correctness of the configuration items.

While the intent of software configuration management as a controlling and supporting discipline for software evolution is quite clear, existing CM systems provide a broad spectrum of functionality for this task. In the following sections we discuss the key areas of functionality in software configuration management, that were identified based on several categorizations of CM systems [Dal03, Dar91, Kru02]. In addition we provide a taxonomy of concepts that are used in the respective functionality areas, which is also partly based on [CW98].

**Version Management**

Refers to the automated act of tracking the changes of a particular file or software artefact over time. The scope of version management is centered around the evolution of a single artefact, with the purpose of later reconstruction of previous versions or the auditing of changes. The following concepts are related to version management:

- **Versioned Item**: A *versioned item* is an item that is put under version control. The term covers anything that may be put under version control, including, for example, files and directories as well as models, entities, relationships, or requirements and documentation. Versioning requires a sameness criterion; that is, there must be some way to decide whether two versions belong to the same item. This decision can be performed with the help of a unique identifier.

- **Version**: A *version* represents a state of an evolving versioned item. Within a versioned item, each version must be uniquely identifiable through a version identifier (VID). Many SCM systems automatically generate unique version numbers or offer additional symbolic (user-defined) names that may serve as a VID.

- **Revision**: Versioning is performed with different intents. A version that is intended to supersede its predecessor is called a *revision*. Revisions evolve along the time dimension and may be created for various reasons, such as fixing bugs, enhancing or extending functionality or adapting to changes in base libraries.

- **Version Space**: The management of versioned items as part of the evolution of a software system introduces an additional dimension to the complexity of the software system. This dimension can in general be referred to as the version space. The version space represents the evolution of the versioned items over time. The additional complexity arises due to relationships between artefacts in a software development process. With the introduction of the version space these relationships depend on the software artefacts and may be valid only for a restricted range of versions of the artefacts.
Branch Management

To allow for parallel concurrent development, independent branches of file evolution must be supported.

- **Variant:** Versions intended to coexist are called *variants*. For example, variants of data structures may differ with respect to storage consumption, run-time efficiency, and access operations. Furthermore, a software product may support multiple operating systems or window systems, which is reflected in the underlying implementation through variants of versioned code items.

- **Branch:** A branch is an agreed split in the evolution of an item into two paths. Each path can maintain independent changes to the versioned item(s). Branches are not reduced to single versioned items but usually include a whole product or system with all the encompassing versioned items.

- **Merge:** Merging is the process of combining two independently evolved variants of versioned items. Most CM systems provide merge functionality on a semiautomated textual basis. Such systems require user intervention if both variants were modified in overlapping textual areas.

- **Trunk:** The unique line of development that is not a branch. The trunk is sometimes also called Main-Trunk, Main-Line or Head.

Build and Release Management

- **Configuration:** A configuration is a fixed set of versioned items. Configurations provide the basis for build and release management. They specify the items that together make up a product build, release, patch or any otherwise significant grouping of elements.

- **Baseline:** The software baseline refers to all items such as source code, run time files, documentation, configuration files, and installation scripts that comprise a software release, both input and output of the build management process. The baseline must include any data or database related scripts or files that are used in the creation of a specific build or release. A baseline may include additional information about compilers, operating systems, and dependent systems (internal and third party) that, together with the items in the baseline, will enable the organization to recover from a disaster, comply with an audit, or understand the full release and its context.

- **Product Configuration:** Defines the labeled list of component versions that is used to define the baseline of a product. A product configuration can be either static or dynamic. A static configuration refers to an immutable (frozen) collection of component versions - one for each of the
components contained in the product. A dynamic product configuration refers to the collection of working (development) components.

- **Build**: The term build refers either to the process of converting source code files into standalone software artefact(s) that can be run on a computer, or the result of doing so. One of the most important steps of a software build is the compilation process where source code files are converted into executable code. While for simple programs the process consists of a single file being compiled, for complex software the source code may consist of many files and may be combined in different ways to produce many different versions.

- **Release**: A named instance of a product or product configuration.

- **Tag**: A tag refers to a type of metadata involving the association of descriptors with versioned items for the purpose of configuration building. All versioned items that are associated with a specific tag belong to the configuration that is denoted by the tag.

### 4.1.4 Versioning Models in CM Systems

Software configuration management systems can be based on different models, which may result in differences regarding the traceability supported by the CM system. The following key categories of versioning were identified by [CW98]:

#### State-Based Versioning

In our definition of versioning, a version has been defined as a state of an evolving item. Version models that focus on the states of versioned items are called *state-based*. In state-based versioning, versions are described in terms of revisions and variants.

#### Change-Based Versioning

Changes provide an alternative way of characterizing versions. In *change-based* version models, a version is described in terms of changes applied to some baseline. To this end, changes are assigned change identifiers (CID) and potentially further attributes to characterize the reasons and the nature of a change. Change-based versioning provides a link to change requests: a change request is implemented by a (possibly composite) change. Thus a version may be described in terms of the change requests it implements.

#### Extensional Versioning

A versioned item is a container, often called a version group, for the set \( V \) of all versions. The functionality of version control is heavily influenced by the way \( V \)
is defined. In the case of extensional versioning, V is defined by enumerating its members: \( V = \{ v_1, \ldots, v_n \} \).

Extensional versioning only supports retrieval of previously constructed versions. New and combined versions must be created by explicit merging. Each version \( v_i \) is identified by a unique number or VID, e.g., V1 or V2.3.4. The user retrieves some version \( v_i \), performs changes to the retrieved version, and submits the changed version as a new immutable version \( v_{i+1} \). Any version \( v_i \) can be retrieved at any time.

### Intensional Versioning

Intensional versioning supports flexible, automatic construction of consistent versions in a large version space. In intensional versioning, the version set is defined by a composite predicate or query:

\[
V = \{ v | vd(v) \land c(v) \}
\]

Both \( vd \) (a user-defined version description) and \( c \) (additional internal constraints) are boolean expressions over the space of selection-attributes \( A_i \). Examples of such attributes are an OS-attribute to determine the operating system, or a boolean Fix-attribute to indicate whether a certain bug fix should be included. \( vd \) is typically a query over a tuple of user-defined VIDs. An example is \( vd = (\text{Unix}, X11) \) that describes a Unix version supporting the X11 window system.

### 4.1.5 Configuration Management in Software Product Lines

Configuration management in the context of software product line engineering entails new challenges, adding to the complexity of CM tasks, in comparison to single application development. In order to preserve the benefit from the use of software product lines, at first an effective variability management is needed. In [Kru02] this problem is addressed by the principle of "divide and conquer". As shown in Figure 4.1, a lightweight solution is given, which divides the management activities into nine sub-problems of variation management. Four of the problem areas were already discussed as basic configuration management, these are sequential and parallel time variation for files and components. The sub-problem Krueger refers to as baseline management is concerned with building configurations in order to represent a specific component as the aggregation of several versioned items. The branched baseline management sub-problem denotes a combination of branching techniques together with Krueger's baseline management approach.

In addition to the four sub-problems covered by basic configuration management, Krueger has identified five additional sub-problems, which are associated with managing variability for products and in the domain space. These sub-problems pose additional challenges for the task of configuration management in the context of product line engineering and are each discussed in detail below.
Composition Management

Composition management is defined as the management of sequential versioning of valid component compositions into a product version, while the component versions evolve over time. According to Krueger, this sub-problem is concerned with the management of components that evolve over time independently from different component baselines, as opposed to components evolving together from one product baseline. The latter case does not necessitate composition management support since component versions evolve as part of a product module. The first case draws on two general approaches, referred to as "the shelf" and "the context" approach.

The "shelf" approach deals with composition management in maintaining all versions of all components on a "shelf", with reusable components. Instantiating a product precedes selecting one version of each component. Thus, guidance for the selection process is needed, since not all combinations will be compatible.

The "context" approach maintains all evolving component versions within a certain context only, which behaves as an evolving module. The context naturally evolves incrementally, as new component versions are added. The suggested solution criticizes the "shelf" approach in view of the significantly fast-growing number of "shelf" combinations - the majority being invalid and using the effort of maintaining their composition rules.

Generally the "shelf" solution proposes to maintain the composition engine and rules in CM. However, it is advised to use the "context" method, as it is simpler. The underlying idea is to implement a context as a list in a single file that tracks the "latest and greatest" compositions of components for a product. Due to future development, this list evolves accordingly, each time a new component baseline is released.

Branched composition management

Branched composition management refers to independent branches of component compositions. The solution relates to CM, by analogy. In the case of the
"shelf" approach the composition engine and rules use the CM branching mechanism. Equally the "context" approach uses CM for managing branched context file versioning.

**Variation point management**

Variation point management refers to the management of variation points in the software artefacts, including their different ways of instantiation. As a solution, variation points are implemented to include a collection of file variants and logic to be able to identify such a file variant at an given point in the domain space.

**Customization management**

Customization management aims at consistent compositions of common and variant files. A customized component is created by assembling the component’s common and variant artefacts, followed by the instantiation of any variation points. This problem is solved by using logical names, corresponding to a point in the domain space and assigned to a component customization. Such a logical name then serves as an identifier for the associated point in the domain to be able to instantiate each variation point in the component that is in the customized component.

**Customization composition management**

Customization composition management refers to the composition of customized components, where customized products are composed of customized and common components. The proposed solution entails associating a logic name to a customized product with a list of customized components.

**4.2 Industrial Practices and Approaches**

In this section we report the practices and approaches for configuration management and evolution of software artefacts adopted by the industrial partners of AMPLE. In particular, some practices for configuration management of software product lines are described by SAP and Siemens.

**4.2.1 Holos**

The agile development process applied at Holos requires a mechanism to support the continuous development process. The need of this control system arises not only on the produced code elements but also on the documentation produced during the prototypes cycles.

---

1See [AMPLE D6.3] and http://www.agilemodeling.com
Source Code

To manage the produced source code it is used the CVS system. The CVS provides a stable and flexible tool that supports the existence of parallel code production. The structure of a CVS project at Holos is presented in Figure 4.2.

The development of the products is done on the main branch (HEAD). For every release of the product a new branch is created to support the correction of the possible existing bugs. To support the existence of concurrent development, it’s offered the possibility to create a new branch on the HEAD that will have a limited lifetime, and should be merged to the HEAD on the end of the development (or at any partial moment that the developed item is considered stable to be incorporated on the main branch).

The definition of the structure was the result of the analysis of the development cycle at Holos. The development process at Holos is mainly focused on the release of new versions upon correction of bugs, and that leads to the decision of having all development process on the HEAD of the project instead of on a separate branch. The usage of CVS merging (from branch to HEAD) is a hard and painful job and with this structure we avoid the necessity to do that several times during process development.

Branches. A big part of the branches created for projects are dedicated to the correction of bugs of the several application releases. The software cycle usually deals with these branches within a limited lifetime. In fact, after several bug fixes releases (there is no limit to the number of patches in each branch) the branch is closed, and all the fixes are merged to the project HEAD and a new release is produced.

This mechanism acts as a safeguard to simple merging processes, since small branches usually are associated with small merging processes. The branch lifetime cycle is presented in Figure 4.3.
Versions. The creation (and managing) of a release version is supported on the CVS with the usage of the Tag functionality. The release of a new version is always associated with the creation of a version tag. The release of the main version (on the HEAD branch) is also associated with the creation of a new branch dedicated to the bug fixes of that release. The bug fixes could produce several patch releases versions to support the corrections. This mechanism allows the team to control each version that is produced and the status of the project at the end of each cycle of the development process.

Documentation

The control version of the documentation is done with a different process as there isn’t the need to constant change of the documents and the occurrences of multiple concurrent revision is not usual and it’s easily managed by the users.

All the documents produced on each phase of the development cycle (Management Documents, Progress reports, Technical documents, specific notes, etc.) follows a guideline on the number that allows the quick identification of the associated phase of the development.

Each deliverable document is referenced with a unique identification number. The nomenclature is defined as `PROJ_XXX_Zzz_ed-rev`, where:

- **PROJ**: The name of project documentation.
- **XXX**: Represents a reference to the documentation package (see Figure 4.4).
- **Zzz**: Represents an ID number for the documents where the first 'Z' corresponds to the WP number and the following 'zz' are incremented per document.
• **Ed-rev**: Composed number for the edition and the revision of the document.

<table>
<thead>
<tr>
<th>Reference</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO</td>
<td>Proposal</td>
</tr>
<tr>
<td>TNS</td>
<td>Technical Note</td>
</tr>
<tr>
<td>RBL</td>
<td>Requirements base line</td>
</tr>
<tr>
<td>TSP</td>
<td>Technical Specification</td>
</tr>
<tr>
<td>DDF</td>
<td>Design Definition File</td>
</tr>
<tr>
<td>DJF</td>
<td>Design Justification File</td>
</tr>
<tr>
<td>MTA</td>
<td>Meeting Agenda</td>
</tr>
<tr>
<td>MTR</td>
<td>Meeting Report</td>
</tr>
<tr>
<td>PRG</td>
<td>Progress Report</td>
</tr>
</tbody>
</table>

**Figure 4.4**: Holos documentation package references

The management of all the produced document versions is done by the documentation manager, who controls the release of all final documents and is responsible for the consistency of all the documentation items.

### 4.2.2 SAP

The growing diversity of SAP’s product portfolio leads to an ever increasing complexity of software configuration management and software product line evolution. Figure 4.5 hints at this trend:

**Figure 4.5**: SAP product portfolio

This has impacts on the following dimensions:

- Stack Management (commonalities, variations, and dependencies);
- Software Lifecycle Management (traceability throughout consecutive development stages);
• Version Management (compatibility and integration of component versions in platform stacks).

Key drivers for efficient software configuration management and evolution of such configurations include:

• Flexibility for composing new products out of existing components;
• Development efficiency (creating and maintaining multiple parallel products as configurations of common components with minimal extensions), and ultimately;
• Reduction of total cost of ownership (TCO) for customers.

In the following we will review a number of mechanisms in place for tackling these challenges.

Software configuration management

The internal product development lifecycle at SAP, referred to as Product Innovation Lifecycle (PIL) as described in D6.3, incorporates software configuration management actions, however many of these were created out of best practice and without standardisation. Furthermore development documents like specification and design documents or test plans are handled outside of formal configuration management.

The main tasks of SCM are due to the fact that standard software solutions are modified to fulfill customer business needs. The main variability mechanisms inside SAP are supported by the Enhancement Framework, Business Add-Ins and Switch Framework, as described in D6.3. Besides these options manual code adjustments cause variability. The current SCM picture may be categorized into three cases:

1. For ABAP programs the SAP Transport Management System (TMS) is used. Software configurations are stored as dedicated SAP systems.

2. Most non-ABAP configurations are managed with Perforce [Per07]. Software configurations are typically accessible on so-called branches using a time-stamp or related information to denote the time-dependent state of the configuration.

3. Some Java projects use the SAP Design Time Repository (DTR) which is integrated into the central SAP development landscape supporting the entire software lifecycle. The DTR is a system for version control, is used to store java code centrally and allow distributed java developments. All source files that are created or changed during the development are stored in the DTR. Changing files necessitates developers to check out relevant files, while they have to be checked in again after the change process.
SAP is currently evaluating new products, methods, and processes for software configuration management in order to speed up development and production and allow for extended use of agile development methodologies like Scrum [SB01] and Extreme Programming [BBvB’01, BA04].

Software evolution

In the next section an overview is given, focusing on data migration and system upgrading. Within the context of software evolution, the issue of data migration is ruled by the Legacy System Migration Workbench (LSMW). LSMW is a tool that supports data migration from legacy systems (non-SAP systems) to SAP systems. Utilizing custom programming for the transformation of data and updating SAP systems is extremely costly, time-consuming and error-prone process.

The LSMW tool makes it possible to transfer data from a variety of sources without any programming, thus counteracting the before mentioned disadvantages. A changeover is basically governed by a set of rules, which has to be defined in advance. The LSMW then uses this definition to generate a program, giving considerable support during the migration. When data is imported, the system performs the same checks as it does during online entry. In more detail, the LSMW covers the following steps:

- Read the legacy data from one or several files (e.g. spreadsheet tables, sequential files).
- Convert the data from source format to target format.
- Import the data using standard interfaces (e.g. Batch Input, Direct Input)

The most important benefits of the LSMW include its independency from SAP releases, platforms, and the kind of data to be migrated; tracing of the migration history at any time and maximum quality and consistency through import checks.

During an upgrade or the import of a support package, existing objects of the SAP standard are overwritten with the objects redelivered. To help the customer retain the objects modified in a previous release, SAP provides all modified objects which are redelivered (in an upgrade or Support Package) in the upgrade adjustment. The basic transaction process is supported by specific tools, as follows.

Those object, that have been modified by the customer and those redelivered by SAP during the upgrade or with the support package are presented for adjustment. During the redelivery import, the system automatically recognizes, whether a modification adjustment is required and visualizes its locus. At each modification request, it is necessary to decide, whether to retain ones original adjustments, or to reset them to the original. Otherwise, the new original stays active in the system.
The Modification Assistant supports this process of adopting customer modifications. In general, objects altered using the Modification Assistant can now be automatically accepted into the upgraded system, if the modifications undertaken in the original version do not directly overlap those made in the customer version. If collisions occur between the two versions at upgrade (naming collisions, or if SAP has deleted an object modified by a customer), the system offers support during the upgrade, which means that a semi-automatic adjustment is made.

In some cases, however, it is still essential to manually adjust objects. Objects modified according to the old system used prior to the advent of the Modification Assistant must be manually maintained after an upgrade has been run.

CM and Software Evolution at SAP

Concluding and consolidating the current picture of CM and Software Evolution at SAP, the core mechanism of CM described as basic configuration management in [Kru02] is covered by TMS, DTR and Perforce to a good extent. These tools are mainly used to manage source code and other relevant artefacts during requirements engineering, design and implementation. In the context of CM in software product lines, the following kinds of variation management ‘composition management, branched composition management, variation point management, customization management and customization composition management’ [Kru02] provide CM approaches to still be improved at SAP.

The main important management tasks followed at SAP are:

- Heuristics for building configuration variants in product lines e.g. core asset compositions over architecture models with component dependencies, configurations models and properties etc.

- Customer enhancements of existing product variants as in ‘customization composition management’. In view of future upgrades, the focus is to minimize incompatibilities with existing customer enhancements.

- Configuration data evaluated at customer runtime as in ‘customization management’ has to be migrated effectively and correctly when moving through different releases.

- With respect to ‘customization management’, the LSMW is an example of how SAP handles this problem currently and not unique to evolution of software product lines in general.

Altogether, the core development strategy of SAP lies in ‘not reinventing the wheel’ with every software product, but to develop existing products in line with a software evolution process that provides a seamless migration path for business and customization/configuration data as well as custom extensions. Currently, this is done at the basis of separate code lines (i.e. branches), one per product variant. In future, this mechanism will not hold, due to an increasing number
of variants and customer enhancements. Therefore, SAP has interest in more flexible enhancement and configuration mechanisms for software product line evolution.

### 4.2.3 Siemens

Siemens consists of a number of business units that concentrate on domains like medical engineering or transportation systems. These business units are again divided into divisions that build product groups, *e.g.* hospital information systems or imaging systems. These divisions are again subdivided into divisions producing system families, *e.g.* imaging systems falls into 23 different modalities, among the x-ray systems, magnetic resonance systems or mammographs. Looking at this heterogeneity, it becomes obvious, that there is not the one way configuration management that is handled within Siemens. There is a need for light weight approaches as well as reliable, integrated configuration management process for big, distributed software development. We will give some examples of current tools and processes for configuration management used at Siemens.

#### Configuration Management Tools

Commercial as well as proprietary tools are used in Siemens for configuration management. The most common commercial or open source configuration systems are: ClearCase, CVS, Subversion, PVCS-Dimensions, Visual SourceSafe, Caliber and Telelogic Synergy.

Typically, products like CVS, Subversion and Visual SourceSafe are used in smaller, collocated development teams. ClearCase and Caliber are the most popular tools for big, distributed systems currently. For big projects these tools are integrated in the development process explicitly and there are metamodels for the requirements structure in these tools. These tools can be heavily customized, and they are, concerning the requirements metamodel and the reports and documents that are generated from the tools.

Currently there is a trend towards task based working in configuration management as opposed to file based configuration management. The problem with the latter is that there is no connection between changes made in different artefacts that were changed together to fulfill one purpose, *e.g.* one development or error correction task.

When developers work on tasks, impacted objects are automatically linked to the task and delivered into the process with it. Configurations and releases are created by selecting the tasks to include. This improves the traceability for crosscutting changes and the integration of development or error correction tasks into the project’s workflow. Tools that support task based configuration management are Telelogic Synergy and ClearCase in the UCM variant.
Variant Management Support in Configuration Management

The following mechanisms are mainly used in Siemens for variant management with respect to configuration management:

- **Branching.** There is one branch per variant. Error corrections in one branch have to be merged to the other variant branches, which can be cumbersome. Every release gets its own branch. A release branch is not included in error corrections anymore. The content of a release branch is further used as a basis to build patches for the customers that got the release. A patch typically bundle several error corrections.

- **Variants in code - versioning of configuration information.** To keep configuration management easy, variants can be managed in code using conditional compilation or configuration files. For every valid variant the adequate configuration file is then also stored in configuration management. However, for managing releases branching used in such projects, still.

- **Duplication for variants.** Another way of avoiding the complexity of branching is to create copies of variants of artefacts. In combination with useful naming schemes variants are easily visible in the code base. The disadvantage is that error corrections have to be done manually in every variant. This procedure is typically done in smaller projects using less powerful, cheap or open source configuration management tools.

These are ways how to represent variants (instantiations of variations points) in basic configuration management systems. There is no real mechanism for managing variants of base assets in parallel in these systems.

Examples for Configuration Management Practices

The following are a few examples on configuration management practices adopted at Siemens.

- **Proprietary Configuration Management.** A group developing software for the banking domain develops its own configuration management system. The domain is heavily data base driven and performance is the main quality attribute. The applications therefore consist mainly of stored procedures. The configuration management system is implemented in stored procedures, too. The artefacts to be managed are simply stored in database tables together with versioning information. The group added a task based change management system on top of the configuration system. Requirements lists are included in the task information in the change management system. The advantage of this approach is that the developers stay within their database world also when using their change management and configuration management. Those tools have the same look and feel as the applications they write.
High-End Configuration Management. In bit, highly distributed software projects configuration management is supported by powerful tools and typically integrated with requirement management, change request management and build tools. An example is a care engine product line development, where the ClearCase UMC is used to combine ClearCase change management, a requirements management tool and build support (Build Forge). The center of the tool chain is the change management tool, whereby changes are new features as well as error corrections. The change management tool keeps the links to all involved artefacts in the configuration management and to the requirements in the requirements management tool that are the trigger for the change. Complete tool integration makes traceability from requirements to code files and back possible.

Low-End Configuration Management For a small, localized development team of automation software, Visual SourceSafe is used as configuration management system. Variants are handled by having several copies of code files. The naming of the files indicates that they are variants of the same artefact. No branching/merging mechanism is in place. For error corrections that might affect all the variants of an artefact, a simple file compare tool is used to detect if error corrections need to be replicated for other variants. The configuration management tool is integrated with the IDE, but not with requirements management or change management. This approach can nevertheless be successful for small, agile development teams.

4.3 Configuration Management in AMPLE

This section aims at identifying the needs of each work package (WP) in AMPLE, regarding configuration management (CM) and software product line (SPL) evolution. The discussion focuses on the work packages 1-3. These work packages may provide various artefacts, that are subject to evolution and require configuration management during requirements engineering, architectural design and implementation. The following key issues are discussed for these work packages:

- The connections of each work package regarding configuration management and software product line evolution.
- The list and description of artefacts of each WP that should be managed as configuration items.

4.3.1 Requirements (WP1)

In general, there are strong links to configuration management from a requirements perspective. Frequently, only a subset of the requirements can be met at
the inception of a product, as a result of limited development resources, market expectations or legislative restrictions. Therefore, software product lines, as well as single software systems, evolve by allocating requirements to specific system releases. A configuration management can provide additional support when planning a release strategy that provides maximal value for the planned release stages, by reflecting the planned releases as configurations in the CM system. In addition, changes to the product line arise due to changes in requirements, especially if a product line context changes over a period of time. Therefore, the management of requirements configurations is quite important.

Software product line evolution deals not only with changes (addition/removal/ modification) of requirements, but also with the need to address each of these requirements as a specific SPL variability. Changes in the requirements imply changes on the features and they are propagated to all the SPL artefacts in other development stages, such as design, architecture or implementation. Thus, all the requirements and variability models produced during requirement engineering need to be managed as configuration items.

The following list describes the artefacts that should be managed as items in configuration management for WP1:

- **Requirement models**: This can include Use Case Models, Activity Diagrams as well as Requirement Definition Language (RDL) documents
- **Feature Models**: Feature models form the basis for managing variability in the AMPLE approach
- **Composition Rules between Requirements Models**: In the Volatile Concern approach, different composition rules are defined between different activity diagrams

### 4.3.2 Architecture (WP2)

The WP2 mainly requires management and evolution support for the architectural models, that are defined as part of the AMPLE approach. The architectural realization of both the core and custom assets tend to evolve over time. Therefore, the configuration management is needed to keep track of these changes and to perform evaluation and analysis tasks on the how/which architectural elements evolved in different product versions.

The following list describes the artefacts that should be managed as items in configuration management for WP1:

- **Component and connector models**: These models show the decomposition of a software system into a set of interconnected component types. The main elements of the component view are components, ports, interfaces and the operations within those interfaces.
- **Interaction models**: Sequence diagrams are used in order to show the dynamic behavior of a set of interconnected components. The main el-
elements of a interaction view are lifelines, event occurrences (e.g. the sending or the reception of a message), messages and signals.

- **Composite structure models**: These models specify how the instances of the different component types designed in the component view are more concretely connected and related. Composition relationships, i.e. component hierarchies, multiplicities and roles are specified in this view.

- **Deployment models**: The deployment view helps to visualize how a system will be physically configured; the hardware involved in the system; how the components are assigned to different hardware units; and how this hardware placed in different nodes are connected.

- **VML descriptions**: the Variability Modeling Language (VML) is a language with the means to identify and affect (i.e. connect, merge, remove) variation points in architectural elements. VML descriptions provide the link between the feature model and the architectural elements, avoiding the inclusion of either information relative to architectural realization of variability points back in the variability models or information relative to variability points in the software architecture description.

### 4.3.3 Implementation (WP3)

The artefacts of a product line (core) and the product specific (extension) artefacts are often evolved independently from each other. The extension artefacts can be evolved by a customer, while the core artefacts are evolved by the vendor of the product line. Even when extensions are evolved by the vendor of the product line, their porting to new versions of the product line can be delayed because of limited resources or financial issues. In a more complicated scenario a customer may integrate multiple third-party extensions of the product.

The support for independent development of core and extension artefacts poses specific requirements on configuration management. It should support multiple versions of artefacts, and dependencies between artefacts should be annotated by version information. Since separate artefacts may be too small units for independent releases, it may be necessary to group artefacts into release packages and manage version dependencies at the level of such packages. It can be useful to distinguish between the changes that are compatible with the old versions of extensions and the changes that require appropriate adaptations for the extensions. Even when a change is compatible with old versions of extensions, this compatibility must be tested. Thus it may be useful to track what versions of dependent packages were tested. When upgrading a package it may be necessary to know which other packages must be upgraded.

WP3 is also concerned about tracking relationships between models and the code. In case of model to code transformation it is necessary to track information between inputs and the output, in particular between the output code, input models and generators. Since both input models and generators can be
evolved, it is also necessary to track version information. This information is useful both for automatic update of the output artefacts, as well as for analysis of compatibility of old extensions or estimation of necessary porting effort in face of product line evolution.

It can also be useful to track dependencies between inputs and outputs at a finer level of granularity: between the elements of the input model and the elements of the generated code. Such dependencies can help to understand the effect of the input to the behavior of the produced software, what is especially useful for finding and fixing errors. At the level of tool support, this relationship can be exposed as a possibility to navigate between the elements of model and the output. In case of template based generation or other forms of transformations that involve pieces of code as input, it is useful to track the relations between input and output lines. This information can be used to relate the compiler errors with the input lines that caused them and to enable debugging on the input lines. In order to make such tracing information useful, its generation must be automated as much as possible.

### 4.4 Connections between Traceability and CM

This section presents the preliminary research results regarding the connection points between currently existing software configuration management concepts and traceability. Traceability can be seen as an integral part of Software Configuration Management (SCM). As the artefacts stored in the SCM system evolve, the system itself should be able to trace the evolution of single artefacts as well as the evolution of the interdependencies between these artefacts. In addition, SCM systems are charged with the task of building configurations that can be used for example as the basis for product deployments, and in general hold the possibility to group versioned elements into any meaningful unit. Here again, the information on the built configurations provides traceability into the different parts of the development process where these configurations are used.

The field of SCM is not limited to the versioning and configuration building of the versioned elements. In particular most SCM systems that are deployed in practice also focus on the team collaboration aspect of software development. These systems provide the means for team members to work in parallel on the same elements and provide access control to the various stored artefacts. Other responsibilities of SCM systems include lifecycle management of stored artefacts and can also include statistical analysis of the evolution. In the course of this section we mainly focus on the versioning and configuration management aspects of SCM systems, but we will also discuss the connection between lifecycle management and traceability.
4.4.1 Connections to Traceability

This section discusses different connections between traceability, software configuration management and software evolution. In general, these connections can be seen from two viewpoints. First of all, there exists a connection between traceability relationships between artefacts and the consistency of these relationships in the context of software configuration management and evolution. In addition a software configuration management system introduces a new dimension to the software development process. This dimension is the version space [CW98], which is driven by the development process outside of the SCM system, but has no representation outside of the system. The version space offers new possibilities of information that may be traced during the software development process.

Consistency of Traceability Information during Evolution

The here discussed scenarios in software evolution, deal with traceability relationships that are already established and possibly modified during the development process but must be kept consistent inside the SCM system. The scenarios are kept in a general manner, where traceability relations are seen as undirected, since the direction is irrelevant to the basic connections that the scenarios make between traceability and software evolution. In addition, the relations in these scenarios are seen as relations between arbitrary artefacts, which could be, for example, requirements, components or classes and other source code level artefacts. While the type of the artefact is relevant for the engineered software system, the consistency of traceability information must apply to all artefacts regardless of their use during the software development process.

Evolution through Revision. In this scenario we explore the possibility that several related artefacts undergo changes that result in new revisions. In order to provide a simple example, we can see the traceability relation as a parent child relationship. In the beginning, the parent and children are stored under their respective versions in the SCM system. Due to the evolution such as a change in a requirement, that can be seen as the parent, the children (e.g. components or code) are also revised resulting in new versions and/or additional elements.

A respective scenario is depicted in figure 4.6, where the initial situation consists of a parent in version V1 which has a relation to two children that are also stored in version V1. Due to the evolution of the requirement (Parent) several changes happen in the system that also affect the traced relationship between the parent and its children. The artefact of Child1 undergoes some changes resulting in a new version V2. The artefact Child2 is not responsible anymore for the fulfillment of the requirement and thus the relationship is removed. An additional element NewChild is introduced in version V1 and has a relationship to the parent, but only in version V2.
From this scenario we can deduce a key property that is required of an SCM system that deals with traceability relationships. The key property is that traceability relationships, which exist between two versions of items, must but not be present in successor or predecessor versions. Therefore, to support the evolution of a software system the traceability information needs to be stored on the basis of versioned artefacts. Only this property allows the traceability information to remain consistent in the SCM system.

The depicted scenario does not cover all distinct use-cases regarding addition and deletion of traceability relationships during software evolution. Such use-cases can depict all atomic operations that are required of the SCM system in order to provide consistency of the traceability information during a revisional evolution. However, the scenario holds true in general and such use-cases may be easily obtained.

**Evolution through Variants.** Similar to the above described scenario for evolution through revisions, we explore the evolution of artefacts through variants in this scenario. In general the introduction of variants poses the problem, that the version space becomes vastly more complex as variations increase the number of possible valid configurations. In addition variants in SCM hold the possibility of being merged again to conclude the parallel evolution and consolidate the changes into a single version.

In order to illustrate a scenario for variant evolution, Figure 4.7 depicts an example, where the initial situation is that an item \textit{Parent1} holds a traceability relation to an item \textit{Element}. In the course of the evolution, the item \textit{Element} is evolved in parallel through versions \textit{V1a} and \textit{V1b}, and an additional traceability relationship is introduced from an item \textit{Parent2} to the version \textit{V1b}. In further step the versions \textit{V1a} and \textit{V1b} are merged into consolidated version \textit{V2}, which now has traceability links to both \textit{Parent1} and \textit{Parent2}.

In the above described scenario, we can see that the introduction of variants is similar to the introduction of new revision in regards to traceability. The traceability relationships must be stored under the assumption that they are only valid...
The merge process has to ensure, that those parts of a variant item, which are responsible for the existence of the traceability relationship, are also included in the consolidated version. Otherwise, the traceability relationship is not valid for the consolidated version and has to be removed.

Traceability in Versioned Items

The primary aspect of traceability that is enabled by an SCM system is the traceability of the evolution of versioned items. Items inside the SCM system are now subject to evolution through revisions (linear evolution) or through variants (parallel evolution). To represent the evolution of versioned items, the version set is often organized in a version graph, whose nodes and edges correspond to (groups of) versions and their relationships, respectively. Each versioned item has its local version graph, which may vary between the items, as there is no uniform global version graph. By employing an SCM system the development process is enriched with traceability information, which is represented by the version graph of each item.

Traceability in Baselining and Configuration Building

Baselining and configuration building serve as a simplification of a complicated build and/or release process. Configuration building is the process of selecting for a given version of an artefact, which may be a revision or a variant. However, the merge process holds possible pitfalls for the propagation of traceability relationships to the consolidated version.

Figure 4.7: Scenario for evolution through variants
a group of elements in the SCM system in the specific versions, that are required for the configuration. A configuration could be, for example, the group of elements that constitute a major release of the product or all elements that are bug fixes that happened after such a release. Configurations may be created by facilitating intensional or extensional versioning techniques.

In the case of intensional versioning the configuration is built via queries or rules over the set of versioned elements in the SCM system. In contrast extensional versioning facilitates a list of elements and their versions, which are the included in the configuration. A combination of both techniques is possible and also seen in practice, as for example in ClearCase [IBM07], which allows to build a configuration that includes the latest revisions of each file with the exception of manually defined older versions. For the sake of simplicity, we can see baselining as a special kind of configuration building, that is, a configuration that includes all elements that are under the version control in an SCM system.

In regards to traceability, the configurations that are built inside an SCM system provide an additional layer of traced information, that does not exists anywhere outside the system. Outside of the SCM system, in other words during the development process, all artefacts are generally treated as belonging to a single configuration. This configuration may not have an equivalent in the SCM system as it may contain the latest development changes, but must be seen as unique configuration nevertheless. The traceability information that stems from the SCM configurations such as a major release configuration are especially valuable in the context of product maintenance. In this context, software product vendors have to deal, for example, with the situation that a customer has a bug in a specific release. The additional traceability information, that relates the artefacts in the versions as they were used for the release to the configuration of the release, may be used to recover the state of the software that was delivered to the customer.

Traceability in State-Based and Change-Based Systems

In contrast to state-based version models, change-based version models offer an additional level of traceability information. In a state-based SCM system all elements are versioned independently, thus no additional information may be extracted from the system regarding the interrelationship between changes to different artefacts. In contrast, change-based systems provide the means to group a set of changes into a composite change. This composite change can be used to provide further traceability information, as for example that a group of changes belongs to a bug fix or was the result of a change request in a requirement.

Traceability in Lifecycle Management

One aspect of SCM systems is the management of a lifecycle for the software development process. For example, in the lifecycle a build and deploy process can help to insure that items are not moved to production environments without
going through some accountability process. Lifecycle information can for example be present as annotational metadata on the versioned items, which conveys the current state in the development lifecycle as a property of the versioned item.

Lifecycle management is not the focus of all SCM systems as some primarily focus on the version control aspect. In general lifecycle management is backed by defined configuration management processes. Such a process represents the sequence of tasks needed to carry out configuration management. Essentially, the process is a plan that defines what needs to be done, who does it and how it is be carried out. Supporting the process is a management function. The process model takes into account policies and procedures of the organization and its software development lifecycle model [Dar91].

The lifecycle information, if present, offers an additional layer of traceability information. The information can provide answers to questions such as "What is the state of components that satisfy a particular requirement?"; e.g. are they currently undergoing a testing process or are they already deployed somewhere and so on.

### 4.5 Framework for Traceability in CM

This section discusses a framework for traceability in the context of software configuration management and evolution. First of all, we describe how the evolution of software systems will be represented in the reference traceability metamodel as well as the collaboration between the CM system and the TraceRepository presented section 1. In addition we discuss the implications of the use model driven development techniques, that are incorporated into the AMPLE development process, as well as the application of feature models to software configuration management.

#### 4.5.1 Representation in the Reference Metamodel

The reference metamodel described in section 1 is suitable to represent all kinds of traceability information and will also be used to capture the evolution of software artefacts that arise as part of the AMPLE development process. In the following we outline how each of the aspects of traceability identified in 4.4 may be expressed in the metamodel.

**Versioned Items**

In order to illustrate how evolution in versioned items is represented, when using the metamodel, we provide an example in figure 4.8 that can be seen as a model instance of the metamodel. In the example two artefacts named "ArtefactA" and "ArtefactB" evolve overtime from versions "V1" to version "V2". The "ArtefactB"
in version "V1" has a dependency on both versions of "ArtefactA", while the version "V2" does not depend on "ArtefactA" anymore.

The example is represented in the traceability metamodel, by providing explicit TraceLinkTypes for "depends" and "evolves". The "evolves" links each connect two instances of TraceableArtefact, which represent the different versions of "ArtefactA" and "ArtefactB" respectively. This approach of representing different versions of artefacts as different model entities has the benefit, that the dependencies between such artefacts may be stored on a per-version basis. As discussed in 4.4.1 this is an important requirement for traceability in software evolution. In the example, we can represent the "depends" link through a TraceLink instance, that reflects the connection from "ArtefactB, V1" to both versions of "ArtefactA" but omits the instance of "ArtefactB, V2".

The version information of "V1" and "V2" is stored in the properties of the TraceableArtefact instances. In contrast to having an explicit version attribute in the TraceableArtefact, this approach has the benefit, that not all traced artefacts need to provide versioning information. Therefore, the traceability framework has the ability to express links between all elements in the development process, including those not under configuration management.

**Baselining and Configuration Building**

Baselines and configurations in general may be represented in the AMPLE Traceability metamodel through distinct TraceableArtefacts, that denote the configuration and provide a "contains" relationship with the versioned items that constitute the configuration. In conjunction with the approach for versioned items presented in the last section this approach also supports the versioning of the configurations themselves.
Change-Based Systems

As described in section 4.4.1, change-based CM system provide the ability to group a set of changes into a composite change and attribute the change with additional information regarding the reason for the change, such as a "bug fix". This information can be captured using the reference metamodel by providing a **Rationale** element to an "evolve" **TraceLink**.

Lifecycle Management

Information regarding the current state of development and other artefacts in the context of the product lifecycle may also be captured using the reference metamodel. Such information can be reflected by using the properties of the **TraceableArtefact**.

4.5.2 Collaboration with the Traceability Repository

In this section, we present the basic collaboration mechanisms, that serve to provide consistency between the configuration management system and the **TraceRepository**. The TraceRepository, presented in section 1, is designed to be a container for traceability information in the form of directed graph connections. In general, the version information represented in a CM system introduces an additional dimension into the traceability graph, i.e., the dimension of time. The resulting graph includes links that represent the evolution of artefacts over time. The nodes of these graphs are versions of artefacts, and have a corresponding representation inside the CM system. Therefore, the information provided by both systems must be kept consistent.

Figure 4.9 depicts the basic collaboration between the two separate repositories. The central elements that both repositories are concerned with, are artefacts. On the CM side artefacts can be stored inside the CM system to create versions. This mechanism optionally incorporates the existing version information in order to update version numbers for previously stored artefacts. Whenever new versions are created inside the CM system, the system automatically creates evolution links for the involved artefacts inside the TraceRepository. These links reflect the version history of the CM system and may not be modified by other means than through the CM system.

Since, the CM system introduces different versions for artefacts into the traceability graph, new TraceLinks between artefacts must be specified by using the respective artefact version information. Without the version information the system can not decide for which versions the link is applicable and must rely on a default policy that decides this applicability. Such a policy can be for example that the link is provided only for the latest version or as another example for all versions of an artefact. These non-evolutionary TraceLinks can be stored inside the TraceRepository without further considerations to the CM system. When the CM system creates new artefact nodes for successor versions of some arte-
facts, the non-evolutionary TraceLinks are examined and extended to the new version if applicable.

### 4.5.3 CM in a Model Driven context

In a model driven development (MDD) context an additional question arises regarding the use of an CM system. The MDD process results in a possibly large number of generated artefacts. In regards to CM the question is whether generated artefacts should become part of the version space. One of the aspects of CM is the regeneration of artefacts, i.e., a retrieval of older versions in order to provide a prior existing state of the developed software system. However, by utilizing an MDD approach the generated artefacts may be regenerated at any point using model transformation techniques, thus regeneration of these can be performed without the help of a CM system.

As part of the AMPLE development process, there may arise dependencies between generated artefacts and manually created artefacts. For example on the implementation level, a generated class can need sub-classing to further refine the behavior. In such a case it is very difficult to express the traceability relations of the manually defined class if the parent class exists only virtually through the generation process and has no representation inside the CM system. In order to provide an end-to-end traceability, these dependencies have
to be captured inside the traceability repository. The traceability model for the generated artefacts must be attributed with configuration information, which expresses the relationship from the artefacts in the CM system to the generated artefacts. In addition the dependencies of generated artefacts to artefacts that contain manual refinements must also be stored in the trace repository. Thus, we can provide traceability for all artefacts that are under the control of the CM system, by forming transitive relationships from the input artefacts of a generative process, over the generated artefacts, to the manual refinement artefacts.

4.5.4 Feature Model based CM

As discussed in section 4.1.5 an important prerequisite for the management of software product lines is an effective system for variability management. As part of the AMPLE project, Feature Models [KCH+90] are used to manage variability in the product line engineering process. Feature models also form an ideal basis for variability management in the context of configuration management. In order to become useful in the process of configuration management, feature models must be treated as first-class entities inside the CM system, as opposed to treating feature models as another artefact that is stored and versioned inside the system. In addition, feature models must also be subject to evolution. This entails versioning of single features as well as configuration building and versioning of such feature configurations that express valid selections of features for deriving concrete products. By elevating feature models and feature configurations into the status of first-class entities, the CM system provides the possibility to attribute artefacts to different features or feature configurations. Thus, the system has the possibility to provide end-to-end traceability in the context of software product lines that are expressed through feature models.

The combination of feature models with basic CM mechanisms address several problems that were presented in section 4.1.5. Feature models can be used for managing compositions and customizations of components through their underlying artefacts, as well as managing customized compositions through feature configurations. Branched composition management can also be addressed, since the feature models and feature configurations are themselves subject to evolution. In the following paragraphs we discuss each of these sub-problems and their relation to feature models.

Composition Management

Feature models can be seen as the versioned component compositions identified by Krueger. The feature models are a form of the identified "context" approach for the component compositions. The context of feature models addresses how components and artefacts may be composed, where the context includes variable parts, as well as the constraints for compositions, expressed for example as cardinalities in the feature model.
Branched composition management

The proposed solution addresses branched composition management by providing feature configurations as well as the evolution of these configuration. Through these mechanisms independent configurations provide the means to express independent compositions. Through the evolution of these configurations the compositions hold the possibility of being independently evolved.

Customization management

Feature models are used to express commonalities as well as variabilities. Therefore, the feature models are well suited for customization management. Different variants that are expressed in the feature model may be composed and the artefacts attributed to these variants form the basis for the customization.

Customization composition management

This last sub-problem refers to the building of concrete product instances, which are composed out of common and customized components and artefacts. In the feature model this refers to a configuration of features that includes all mandatory features that must be selected from the root node and selects valid variants of features at all appropriate variation points. The underlying components and artefacts are attributed to such a feature configuration which may then serve as a concrete product instance.

4.6 Conclusions

We have defined an initial framework for traceability in the context of AMPLE and configuration management. The traceability metamodel provides the necessary expressiveness to adequately represent evolution of software artefacts for subsequent analysis steps. The proposed framework is capable of expressing software product line configurations and their evolution through feature model evolution. In addition, the framework accommodates for the generative programming approach that is incorporated into the AMPLE development process.

We have not yet addressed the issue of software support for the framework. Current planning assumes that existing configuration management systems such as CVS or SVN [Fou06, Col07] provide adequate support for the basic configuration management tasks identified in section 4.1.5. Therefore, we propose to utilize one of these systems and provide additional functionality as extensions or through thin-layered adapters to these systems.
5 Conclusion and Future Work

In this deliverable we have described the first phase of the definition of the traceability framework that will be used within AMPLE to support aspect-oriented, model-driven software product line development. The platform consists of a traceability metamodel, traceability tool support and complementary areas of interest. The first phase consisted of the definition of the metamodel, the organization of the tool support structure and the description of the complementary areas: a taxonomy of dependencies, tracing of design rationale in the presence of uncertainty and tracing in the field of configuration management.

The first step that has been taken is the definition of an abstract and generic traceability metamodel, which allows the representation of trace links with multiple sources and targets. We have elaborated a reference traceability metamodel as a separated bipartite graph instead of a simple graph as is usual in many approaches. Types of artefacts and links have been introduced to enable more advanced queries and a better organisation of the information. As the main logic and intelligence will be placed in the tools that access to the trace repository, no special features are required to address traceability of aspects or models within the metamodel. The various dimensions of the product line development have been also incorporated in a generic manner. The implementation of this metamodel has been completed and will be the core of the traceability tool chain.

To complement the metamodel, a general dependency taxonomy has been defined, which explicitly identifies the various dimensions of tracing in the context of software product lines. With this taxonomy, tracing information can be refined and contain additional information in a compact manner. At this point, the taxonomy provides a basic dependency classification and a more refined classification soon to be integrated.

A second complementary step to the metamodel is an extension that facilitates the conceptualisation of the rationale for design decisions in software product line development. This model comprises elements from argument-based rationale methods, problem-solution approaches and quality evaluation methods. A unique feature lies in the ability of capturing and reasoning with uncertainty that may be present on the assumptions made in the rationale.

Finally, an initial framework for traceability in the context of AMPLE and configuration management has been proposed. This traceability metamodel provides the necessary expressiveness to adequately represent evolution of software artefacts for subsequent analysis steps. The proposed framework is capable of expressing software product line configurations and their evolution through feature model evolution. In addition, the framework accommodates for the generative programming approach, which is incorporated into the AMPLE development
process.

The traceability platform will come towards completion over the next period while working towards deliverable D4.2. Future work will resolve open and remaining issues that have not been addressed in this deliverable. One of the main activities is the realisation of a tool support prototype in the next six months. For this, the link between the traceability metamodel and the rationale and uncertainty model needs to be elaborated. The first captures basic knowledge of trace links, while the second aims at providing a support to reason during the development process and to evaluate design alternatives. The place of this evaluation process for reconsidering design choices in the tool structure needs to be decided.

The dependency taxonomy will also be extended, based on classification activities that have been performed. This will provide a more detailed taxonomy for requirements to architecture and architecture to implementation. A second important future activity is the validation of the taxonomy within an industrial context using case studies and data available from the AMPLE partners.

For configuration management, the main focus for future work is on tooling support. The current view is that existing configuration management systems such as CVS or SVN provide adequate support for the basic configuration management tasks identified in section 4.1.4. Therefore, it is proposed to utilise one of these systems and provide additional functionality as extensions or through thin-layered adaptors to these systems.

A prototype of the tool support for the traceability metamodel has been developed and is already used by partners. However, as the prototype only provides base functionality, the tool chain needs to be enriched with complex queries and functionality like measures and views of traces. Furthermore the analysis of uncertainty in rationale of design decisions needs to be integrated in order to provide sophisticated tool support. Finally, while the last chapter describes an initial organisation and some initial rules to manage configurations and traces, several issues have to be solved regarding the management of fine-grained variations and the tool support ensuring a consistent management of both repositories.
Bibliography


[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.


