ARTIST

FP7-317859

Advanced software-based seRvice provisioning and migraTIon of legacy Software

Deliverable D10.1

Repository Requirements

Editor(s):	Oliver Strauß
Responsible Partner:	Fraunhofer
Status-Version:	Final – v1.0
Date:	30/01/2013
Distribution level (CO, PU):	Public

CC BY-SA

Project Number:	FP7-317859
Project Title:	ARTIST

Title of Deliverable:	Repository Requirements
Due Date of Delivery to the EC:	31/01/2013

Work package responsible for the Deliverable:	WP10 – Common migration artefacts provisioning and management
Editor(s):	Fraunhofer (Oliver Strauß)
Contributor(s):	Fraunhofer (Oliver Strauß, Jürgen Falkner, Tatiana Senkova), ATOS (Yosu Gorroñogoitia), INRIA (Javier Canovas, Hugo Brunelière), ENG (Stefania D'Agostini), SPIKES (Bram Pellens)
Reviewer(s):	Yosu Gorroñogoitia, ATOS
Approved by:	All Partners

Abstract:	This report documents the requirements for an artefact repository and marketplace in the ARTIST context as well as an overview of the state of the art in fields related to model management.
Keyword List:	artefact repository, artefact marketplace, versioning, traceability, evolution
Licensing information:	This work is licensed under Creative Commons Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0) http://creativecommons.org/licenses/by-sa/3.0/

(cc) BY-SA

Document Description

Document Revision History

Version	*Date*	*Modifications Introduced*	
		Modification Reason	*Modified by*
v0.1	13/11/2012	First draft version of the ToC (Oliver Strauß)	Fraunhofer
v0.2	21/11/2012	Second draft version of the ToC (Oliver Strauß)	Fraunhofer
v0.3	10/12/2012	Integration of first contributions from ENG (Stefania D'Agostini), SPIKES (Bram Pellens) and Fraunhofer (Oliver Strauß)	ENG, SPIKES, Fraunhofer
v0.4	18/12/2012	Integration of contributions from INRIA (Javier Canovas, Hugo Brunelière) and ATOS (Yosu Gorroñogoitia)	INRIA, ATOS, Fraunhofer
v0.5	08/01/20123	Integration of contributions from SPIKES (Bram Pellens)	SPIKES, Fraunhofer
v0.6	16/01/2013	Version for internal review (Oliver Strauß, Jürgen Falkner)	Fraunhofer
v0.7	25/01/2013	Updated content in section 3.6.3 and section 3.6.4	ENG
v1.0	30/01/2013	Final version. Integration of all feedback received from the review	Fraunhofer

CC BY-SA

Table of Contents

Table of Contents ... 4
Table of Figures ... 5
Table of Tables ... 5
Executive Summary ... 6
Abbreviations ... 7
1 Introduction ... 8
1.1 Motivation ... 8
1.2 Purpose of this document ... 8
1.3 Structure of this document ... 8
2 Requirements for the artefact repository and marketplace ... 9
2.1 Methodology ... 9
2.2 Common repository and marketplace requirements ... 10
2.2.1 CF-1: Support for common MDE artefact types ... 10
2.2.2 CF-2: Support the organization of a big number of artefacts ... 10
2.2.3 CF-3: Efficient retrieval of artefacts ... 10
2.2.4 CF-4: Record user feedback ... 11
2.2.5 CF-5: Access control ... 11
2.2.6 CF-6: Version control for artefacts ... 12
2.2.7 CF-7: Eclipse integration ... 12
2.2.8 CF-8: Integration with external tools ... 13
2.3 Artefact repository requirements ... 13
2.3.1 RF-1: Traceability between artefacts ... 13
2.3.2 RF-2: Visualization of relationships between artefacts ... 13
2.3.3 RF-3: Support for artefact evolution ... 14
2.4 Artefact marketplace requirements ... 14
2.4.1 MF-1: Publication of artefacts ... 14
2.4.2 MF-2: Web based user interface ... 14
2.4.3 MF-3: Support for commercial artefacts ... 15
2.5 Non-functional requirements ... 15
2.5.1 NF-1: Ease of use ... 15
2.5.2 NF-2: Extensible architecture ... 16
2.5.3 NF-3: Sufficiently large initial artefact population ... 16
3 State of the art in various model management aspects ... 17
3.1 Model management ... 17
3.2 Artefact traceability ... 18

BY-SA

3.3 Model versioning 19
3.3.1 Concepts 19
3.3.2 Conflict detection and resolution 20
3.3.3 Overview of existing solutions 20
3.4 MDE model repositories 22
3.4.1 CDO Model Repository 22
3.4.2 Teneo 22
3.4.3 EMFStore 23
3.4.4 ModelBus 23
3.4.5 EMFTrace 23
3.4.6 Morsa 24
3.5 Evolution 24
3.5.1 Schema Evolution 24
3.5.2 Ontology Evolution 26
3.6 Relevant research projects 28
3.6.1 Momocs – XSM Knowledge Base Repository 28
3.6.2 MODELPLEX 29
3.6.3 SeCSE – Service Centric System Engineering 30
3.6.4 SLA@SOI 31
3.6.5 Morse 32
4 Conclusions 33
4.1.1 Evaluation of the state of the art 34
References 36

Table of Figures

Figure 1: MOMOCS XSM KB Repository Model Historic View 29

Table of Tables

Table 1: Comparison of current Model Versioning Systems (based on [23]) 21
Table 2: Overview of requirements and first estimate on priorities and technical feasibility 34

BY-SA

Executive Summary

In model driven engineering (MDE) and especially in MDE based migration projects such as those considered in ARTIST, a lot of knowledge and information is contained in the form of models, meta-models and in transformations between models. To keep these kinds of projects manageable and to reduce development effort and time, the mentioned artefacts have to be effectively managed and the potential reuse of high level artefacts has to be leveraged.

ARTIST therefore supports MDE migration projects with an artefact repository that stores, manages, and classifies its contents as well as leverages user feedback and relationships between artefacts to facilitate efficient access to the MDE artefacts. The repository is complemented with a marketplace where potentially reusable artefacts can be offered or consumed.

This deliverable D10.1 “Repository Requirements” presents the results of the user requirements elicitation and provides an overview over the state of the art in field related to model management as well as related research projects.

(cc) BY-SA

Abbreviations

CASE	Computer-Aided Software Engineering
CDO	Connected Data Objects
CF	Common Requirement
CVS	Concurrent Versioning System
DDL	Data Definition Language
DTD	Document Type Definition
EMF	Eclipse Modelling Framework
M2M	Model-to-Model-Transformation
M2T	Model-to-Text-Transformation
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MF	Marketplace Requirement
NF	Non-functional Requirement
NoSQL	Not only SQL
OCL	Object Constraint Language
OWL	Web Ontology Language
P2P	Peer-to-Peer
QoS	Quality of Service
RF	Repository Requirement
SCM	Software Configuration Management or Source Control Management
SLA	Service Level Agreement
SQL	Structured Query Language
SVN	SubVersioN
UDDI	Universal Description, Discovery and Integration
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XSD	XML Schema Definition

(cc) BY-SA

1 Introduction

1.1 Motivation

During an ARTIST migration a great amount of information is captured from the reverse engineering process and produced in the forward engineering process. Most of this information is captured and contained in a big number of different and diverse artefacts that are interrelated in various ways. It is essential for the success of the ARTIST approach to manage and organize these artefacts in a repository and to make the knowledge (e.g. meta-data) contained in them accessible (see Section 3.1).

Aside from the application specific artefacts many higher level artefacts exist. For example they describe technologies (by means of a meta-model) or enable a mapping between different technologies (via transformations). These artefacts are expensive to create and require a lot of technical and domain knowledge. However since they are application independent they can be reused across many migration projects. A marketplace[1] for MDE artefacts can bring producers and consumers of these artefacts together, foster reuse and create added value for both of these groups:

- Once such an artefact has been produced, the creator can benefit from his efforts by publishing the artefact on the ARTIST marketplace. Then he or she can get feedback from the community to improve the artefact, improve his or her reputation by demonstrating competence or good will for the community and can potentially generate additional revenues by selling the artefact over the marketplace.
- Developers can save time and effort by reusing artefacts from the marketplace. This will work best, if the fitting artefacts can be identified and evaluated quickly and easily.

Repository and marketplace are complementary in that information gathered in the repository during development can be used to improve the findability of artefacts in the marketplace. Feedback on artefacts gathered via the marketplace can in turn be used to improve the artefacts in the development space.

1.2 Purpose of this document

This deliverable D10.1 "Repository Requirements" documents the results of the user requirements elicitation process conducted in Task T10.1 "Developing a repository for migration artefacts". As a basis for further in depth investigations the requirements have been complemented with an overview of the state of the art in relevant model management related fields and related research projects.

1.3 Structure of this document

This document presents the user requirements for the ARTIST artefact repository and marketplace in Section 2. This is followed in Section 0 by a survey of the state of the art in various fields that are relevant in this context. A set of conclusions are presented in Section 4.

---

[1] In the context of ARTIST the term "marketplace" is by no means meant to imply a purely commercial approach. Instead it should be interpreted synonymous to "directory" or "platform of exchange" although the possibility of offering commercial artefacts should not be ruled out.

BY-SA

2 Requirements for the artefact repository and marketplace

2.1 Methodology

The ARTIST consortium agreed to use an agile development approach. This is an appropriate choice for research projects that have an explorative element and live in a changing research landscape. The iterative approach allows to flexibly adapting to changes. As a consequence not all requirements can be specified upfront. Instead they are adapted and refined if the course of development makes it necessary.

Therefore the exact functionality of the ARTIST repository and marketplace cannot be specified completely at this point in time. Instead an initial set of requirements is described in the current document with focus on of high level requirements to give an idea of the intended scope of the system. The requirements will be adjusted and refined in an iterative process based on interaction between the different work packages.

Elicitation

The ARTIST artefact repository is a component that is in contact with tools from other work packages and has implications for the ARTIST process and for dissemination via the public repository interface. Therefore the needs and requirements of the other work packages were taken into account. The requirements elicitation process had the sources of input:

- A questionnaire has been designed and circulated in order to collect the requirements and expectations of the other work packages towards the repository. It covers various functional and non-functional aspects of the repository and its public interface.
- Further input was gathered from the use case descriptions taken from the preliminary version of deliverable “D12.1 Use cases definition and migration architecture” and from the results of the “Identification of Components” questionnaire produced in WP6.

Analysis and specification

The gathered requirements are documented in the following sections according to the following template:

- ID: A unique identifier used for references
- Description: A short description of the requirement
- Explanation: The reason and background for the requirement
- Priority (users views): high| medium | low
- References and dependencies: Relationship with other requirements

The ID of a requirement has the following format “<Type>-<running number>” where type can be one of the following:

- <C|R|M>F = (common | repository |marketplace) functional requirements
- NF = non-functional requirements

The ID is followed by a short name for the requirement.

(cc) BY-SA

Initial verification

In a first iteration the requirements have been sent to the partners for feedback and validation. Especially the priority and feasibility attributes were checked in this step.

2.2 Common repository and marketplace requirements

This section contains requirements that are valid for both the ARTIST artefact repository and marketplace.

2.2.1 CF-1: Support for common MDE artefact types

Description: The repository and marketplace must at least be able to manage the following types of artefacts:

- Meta-models
- Models
- Model-to-model-transformations (M2M)
- Model-to-text-transformations (M2T)

Explanation: The listed artefact types are essential for MDE processes. In addition to this list other types like e.g. model discoverers can optionally be supported.

Priority: high

References and dependencies: -

2.2.2 CF-2: Support the organization of a big number of artefacts

Description: Users describe potentially reusable artefacts with structured text based descriptions and keywords. Artefacts can further be classified in a (facetted) classification schema and by user defined tags.

Explanation: Reengineering and migration projects have to deal with a big number of diverse artefacts. In order to support efficient search and retrieval, artefacts are attributed with meta-data. Part of it has to be supplied by the users (e.g. descriptions based on structured text, keywords or classification). Another part can be deduced automatically (like the relationship between models and meta-models).

Priority: high

References and dependencies:

- Extends CF-1

2.2.3 CF-3: Efficient retrieval of artefacts

Description: The repository provides users with efficient searching and browsing capabilities.

Explanation: The key functionality for reuse oriented repositories is to enable users to easily and reliably find the assets that he or she is looking for. This can be achieved by exploiting information extracted from the artefacts (e.g. artefact type or links between models), user

(cc) BY-SA

supplied meta-data (e.g. keywords, descriptions, classification) or data inferred by some logic (e.g. by rule or heuristics). The repository should support

- Browsing of artefacts by facetted classification, artefact type and artefact relationships
- Content filtering
- Context based search that exploits the context of the user, e.g. search for transformations that take the type of model as input that the user currently edits in the model editor.

The repository can optionally support searching by providing a custom in a query language.

Priority: high

References and dependencies:

- Extends CF-2

2.2.4 CF-4: Record user feedback

Description: Users of artefacts should be able to provide feedback for artefacts to indicate improvements and problems or to rate their quality. This will be achieved via user comments and ratings.

Explanation: User feedback helps developers to improve their artefacts and supports users to evaluate the quality of an artefact and the competence of its creator.

Priority: medium

References and dependencies:

- Supports RF-3

2.2.5 CF-5: Access control

Description: The repository should support access control for all managed artefacts based on access rights that can be assigned to users, users groups or user roles.

Explanation: Reverse and forward engineering artefacts contain knowledge about the underlying concepts, processes and procedures of a software system. This potentially valuable property has to be protected by proper access control mechanisms that ensure that only authorized users have access to restricted artefacts.

The repository should support

- user groups,
- user roles,
- the notion of projects with distinct sets of artefacts
- the artefact visibilities "private", "company", "project" and "public" (possibly by means of access rights)

The repository should support to share artefacts between projects.

(cc) BY-SA

Having user based access control implies that the repository and marketplace must have a user management component.

Priority: high

References and dependencies:

2.2.6 CF-6: Version control for artefacts

Description: The repository must be able to store and make accessible previous versions of an artefact. It should also support more advanced features known from source code management systems like branching and tagging, as well as conflict detection and resolution.

Explanation: Versioning is a key element of artefact management. During development it provides a safety net for developers. It also helps to capture the changes of an artefact over time and thus supports evolution and maintenance. Since conflict detection and resolution for structured hierarchical data such as models can become very complex, the focus of the ARTIST repository is on providing versioning and change notifications. Conflict detection and resolutions could be deferred to existing tools.

Priority: medium

References and dependencies:

- Extends CF-2
- Supports RF-3

2.2.7 CF-7: Eclipse integration

Description: The repository and marketplace must integrate well with the Eclipse[2] workspace in order to optimally support the developer in its usual working environment.

Explanation: The baseline technology for the ARTIST tooling is the Eclipse platform. Gathering data for management and reuse always causes additional effort for developers. Therefore the artefact repository and marketplace should

- avoid extra work if possible by using the extension mechanisms of Eclipse to capture as much of the needed information automatically and
- be easy to use and learn by adhering to the common GUI idioms and Eclipse interfaces.

Priority: high

References and dependencies:

- Complements CF-8

[2] Eclipse development environment, see http://www.eclipse.org for details.

(cc) BY-SA

2.2.8 CF-8: Integration with external tools

Description: Non-Eclipse tools should be able to use the most important functionality of the artefact repository and marketplace via web service interfaces.

Explanation: Other ARTIST tools can use the repository and marketplace via the Eclipse integration. A web service based interface is necessary for integration with other external tools.

Priority: medium

References and dependencies:

- Complements CF-7

2.3 Artefact repository requirements

2.3.1 RF-1: Traceability between artefacts

Description: The repository should track relationships between the managed artefacts to provide traceability.

Explanation: Knowing dependencies and other relationships between artefacts is essential to provide good management support, since it makes services like change impact analysis or automatic validation possible and offers new possibilities to search and navigate the managed artefacts. In a first step, the relationships will be tracked on the artefact level.

Possible relationships of interest are:

- A model *conforms to* a meta-model
- A model or text file acts as *input* to a transformation
- A model or text file is *produced from/by* a transformation
- A transformation expects instances of some meta-model as input
- A transformation produces instances of some meta-model as output

The first relationship can be used for change notifications or verification. The next two can be the basis for tracing chains of transformations, or for browsing and navigation. The last two can be exploited for context sensitive search.

Priority: high

References and dependencies:

- Extends CF-2
- Supports CF-3
- Supports RF-3

2.3.2 RF-2: Visualization of relationships between artefacts

Description: The repository should provide functionality to visualize the relationships between artefacts in order to make it easy for users to get information about the artefacts.

(cc) BY-SA

Explanation: Visualizations are able to provide complex information in a way that is easy to understand. The repository should therefore be able to visualize the relationship between artefacts and should support different views on the artefact content according to artefacts type.

Priority: medium

References and dependencies:

- Extends RF-1
- Supports RF-3

2.3.3 RF-3: Support for artefact evolution

Description: The repository should track changes to artefacts and provide support for their evolution.

Explanation: By recording changes to artefacts and exploiting the relationships between artefacts, it will be possible to perform change impact analysis. In the case meta-model changes, even change propagation could be performed. By also recording the rationale for a change, a history of decisions can be provided.

Priority: high

References and dependencies: -

2.4 Artefact marketplace requirements

2.4.1 MF-1: Publication of artefacts

Description: Artefact producers can explicitly provide artefacts that they consider reusable to the public via the artefact marketplace.

Explanation: Not all artefacts are suitable for reuse because they are specific to certain applications. In the ARTIST context, possible reuse candidates are models, meta-models and transformations. When an artefact provider wants to release an artefact, he/she has to perform a publication process where additional meta-data (like a description, keywords, classification, or licence information) have to be provided.

A provider should be able to update or remove his/her published artefacts.

Priority: high

References and dependencies:

- Extends CF-2

2.4.2 MF-2: Web based user interface

Description: The marketplace should use a publicly available web site interface to address as many users as possible.

Explanation: Integration of the marketplace in Eclipse is mainly valuable for developers. A much bigger audience can be reached by providing a marketplace web site. Besides offering reusable MDE artefacts an ARTIST marketplace web site could provide the ARTIST tools and the marketplace Eclipse integration plugin as download.

Priority: high

References and dependencies:

- Complements CF-7

2.4.3 MF-3: Support for commercial artefacts

Description: The marketplace should support free artefacts but also provide the possibility to offer commercial artefacts that have to be paid.

Explanation: One of the most obvious incentives to invest in the provision of reusable artefacts is the possibility to generate revenues from them. Therefore it should be possible to sell artefacts over the marketplace. This has the following implication:

- Price and licensing information have to be included in the meta-data
- There has to be a "shopping cart" and payment facility
- Access rights management must make sure that purchased artefacts are only accessible by user who purchased them
- Access right management should support hiding information in the public artefact description that would reveal too much about the artefact.

Priority: low

References and dependencies:

- Extends MF-2

2.5 Non-functional requirements

2.5.1 NF-1: Ease of use

Description: The repository and marketplace must be easy to use and learn.

Explanation: Since providing extra meta-data or searching for external artefacts is additional effort for users, it should be minimised. The tools should be unobtrusive and should avoid unnecessary inputs where possible. It should be conformant to user expectations and look and behave like other similar systems.

Priority: high

References and dependencies: -

(cc) BY-SA

2.5.2 NF-2: Extensible architecture

Description: The tools should have an extensible architecture in order to be able to add new functionality or exchange some of the underlying technology.

Explanation: In a research project, requirements can change and it is important to be able to change some parts of a system without having to adapt all the other parts. It makes the system more valuable to other users and later research projects if it can be adapted and extended in an easy way.

Priority: medium

References and dependencies: -

2.5.3 NF-3: Sufficiently large initial artefact population

Description: The usefulness of the marketplace is more than proportional to the number of available artefacts. A sufficiently large number of initially available artefacts is very important for the acceptance of the marketplace.

Explanation: One of the main difficulties with getting an ARTIST solution off the ground will be building critical mass of SW architects using it. This, in turn, is based on the cost/ease of using the solution, and in turn, on the number of ready to use generic artefacts. If there is not a critical mass of reusable artefacts the costs of using it will be too high, we will have no customers and no commercial or developer interest.

The project should focus on getting the most generic artefacts available, with a focus on coverage to attract enough users (critical mass) for more specific artefacts to be worth creating, whilst leaving the opportunity for better generic artefacts to be produced in time.

In addition to creating artefacts in the project, existing freely available content (e.g. like the AtlanMod model zoo[3]) could be imported into the marketplace to achieve this goal.

Priority: high

References and dependencies:

- Depends on NF-1

[3] The AtlanMod model zoo is a collection of meta-models in various formats. It is available from http://www.emn.fr/z-info/atlanmod/index.php/Zoos

(cc) BY-SA

3 State of the art in various model management aspects

3.1 Model management

The building of complex software solutions such as model-driven modernization processes involves more and more models in the broad sense. These numerous models are not only UML models but can be of very various natures (e.g., models can be expressed in XML documents that conform to specific XSD schemas or DTD, it can also be EMF models that conform to different meta-models expressed in Ecore format, etc.). Moreover, these models are often linked to each other and are also involved in complex chains of operations which, many times, consist of sequences of transformations like in model-driven modernization processes. As a consequence, facilities are strongly needed in order to manage all these models and specify all the relationships and dependencies that may exist between them.

Thus, the application of model-driven modernization processes means that we need to handle a huge number of various and varied modelling artefacts. These artefacts are:

a) **Numerous**. One modernization case may use several hundreds of such artefacts that were derived from a big number of existing legacy components. Sometimes the number can get as high as tens of thousands of units and even more.
b) **Distributed**. Some artefacts may be available on Web sites or in other modernization projects.
c) **Interrelated**. The artefacts are related by strong semantic links. For example a model-to-model transformation should refer to its source and target meta-models. These meta-models may be themselves versions or extensions of other meta-models. A model M_b obtained from a model M_a by a model-to-model transformation M_t may record its origin model and its transformation model M_t. Moreover these models M_b and M_a may be related by a traceability relation. These are only a few of the semantic relations that may be found in a set of MDE-based artefacts.
d) **Heterogeneous**. The artefacts are of different natures. They should be categorized in a very systematic organization, i.e., a typing system. The simplest idea that comes to mind is to consider that all artefacts are models. Then we have a simple solution which consists in stating that each one is typed by its meta-model.
e) **Complex**. The artefacts may contain a lot of internal elements. Here again, if we follow the simplifying conjecture stated above, the possible nature of elements contained in one artefact is defined by the meta-model.

In order to provide access to these MDE artefacts without increasing the accidental complexity introduced by their use, we need to identify appropriate ways to create, store, view, access and modify their metadata. This metadata mainly represents detailed information on these modelling artefacts, also including the various kinds of possible links between them. Similarly to what already exists for traditional programming artefacts, the provision of a large repository to store a set of models, meta-models, model transformations, etc. is an example of such model management facilities that should be provided. This is also the main idea of “Modelling in the Large” which advocates the use of MDE principles and techniques for the better handling of models [1]. In the last past years, there have been several approaches proposed to provide an actual support for managing models in this sense.

First, the notion of megamodel [2][3] has been proposed and introduced to support this idea of “Modelling in the Large”. A *megamodel* is a model representing artefacts (e.g. models, meta-models, transformations, etc.) and their relationships by specifying associated

(cc) BY-SA

metadata. A megamodel should make it possible to reason about a complex software engineering process without entering into the implementation details of the involved technological spaces [4]. Obviously, the results obtained when reasoning on a megamodel must be consistent with those that would be obtained directly looking to the actual artefacts.

Then, based on this core concept of *megamodel*, the notion of *Global Model Management (GMM)* has been initiated within the context of the IST MODELPLEX European project (cf. Section 3.6.2). GMM aims to provide support for managing global modelling resources, usually heterogeneous and distributed, in the field of MDE-based software developments. Thus, to access them without increasing the accidental complexity of MDE, the initiative proposes using megamodels to manage those that are involved when developing a practical solution. The goal of this initiative is to provide a simple access to the MDE developer so that he/she may deal with complex problems with simple interfaces. Concretely, this has been applied on practical scenarios such as Performance Engineering [5] as well for experimenting on more advanced features such as typing inference [6].

A related initiative relies on the concept of *macromodel* [7] a model that describes the structure of *multimodels* (i.e. sets of interconnected models) according to particular development processes. Complementary to what is already represented in megamodels, the focus of this work is put more on the detailed semantic of the inter-model relationships and the representation of the modellers' intentions.

3.2 Artefact traceability

When developing a model repository, the need to deal with traceability, not only within single model-to-model transformation but also within chains of such transformations or other components such as model discoverers and code generators, becomes clearly a fundamental prerequisite. Indeed, it is required to be able to trace the different models as well as their elements during the whole model-driven modernization process (i.e. to navigate the various traceability links between them both forwards and backwards).

Although the area of research about traceability is currently emerging, as commented in [8], several types of trace links can be identified according to the involved artefacts. In the following, we identify and describe the main trace links which are going to be considered within the ARTIST project:

a) Traceability in model transformations (i.e. model-to-model). Model transformation traceability has already been considered by some works, in the MODELPLEX project for instance (cf. Section 3.6.2), where existing model-to-model transformations are enriched in order to generate trace models along with the regular target models. Such trace models include traceability links between elements from the source and target models. At the time being, model-to-model transformation traceability is still a hot research topic.
b) Traceability in model discoverers (i.e. notably text-to-model). The definition of trace links between legacy artefacts and the discovered models is still a new area of research in MDE. Proprietary tools such as Agility [9] or Yakindu [10] offer some built-in traceability support in the particular context of their respective frameworks. However, there is currently no generic support for such a feature. The Eclipse based Open Source tool MoDisco [11] that will be used and extended in ARTIST aims to fill this gap.
c) Traceability in code generators (i.e. model-to-text). Code generation languages and tooling usually do not provide a full traceability mechanism. Instead, some solutions

(cc) BY-SA

(e.g., Acceleo [12] or Xpand [13]) provide the definition of *protected regions* which allow defining the areas which can be edited manually afterwards in the generated code, without the risk of being rewritten during a regeneration phase.

This capacity of efficiently preserving and using these different types of traceability links is often strongly related to the provided model management support (cf. Section 3.1). Actually, some experimental works (e.g. [14]) have already shown the interesting results that can be obtained by treating simultaneously Model Management and Model Traceability.

3.3 Model versioning

Model versioning is a novel research stream that has attracted a lot of attention in recent years as it is highly relevant to current software engineering problems [15]. As today's software systems are characterized by a high level of complexity, high level representations of software systems are needed to keep the complexity manageable. Building the models of software systems allows for dealing with the complexity on a higher level of abstraction [16]. However, such systems are usually the result of a collaborative work performed concurrently. Thus conflicting changes to models can occur and have to be dealt with. In optimistic versioning approaches (where distributed parallel team work is allowed) the need to resolve conflicts arises that occur when concurrently evolved versions of one model are merged [16]. The following sections summarize existing approaches to cope with this problem.

3.3.1 Concepts

The following overview is mainly based on the work of Brosch et al. [15] who give a current and comprehensive overview of the state of the art in model versioning. They define the twofold goal of software versioning systems as follows: *"First, such systems are concerned with maintaining a historical archive of a set of artefacts as they undergo a series of operations and form the fundamental building block for the entire field of Source Configuration Management (SCM), which deals with controlling change in large and complex software systems. Second, versioning systems aim at managing the evolution of software artefacts."* [15]

Versioning systems could be either state-based or operation-based with regard to how conflicting versions are recognised and merged in order to create a consolidated version.

In the operation-based approach a buffer containing the operations between the two states of a document is used to keep the information about the evolution of the document. All the operations are recorded, stored in a log and analysed. Brosch et al. [15] note: *"The downside of operation recording is the strong dependency on the applied editor, since it has to record each performed operation and it has to provide this operation sequence in a format which the merge approach is able to process. The directly recorded operation sequence might include obsolete operations, such as updates to an element which will be removed later on. Therefore, many operation-based approaches apply a cleansing algorithm to the recorded operation sequence for more efficient merging. The operations within the operation sequence might be interdependent because some of the operations cannot be applied until other operations have been applied."* This approach is efficient for large documents, because the number of operations that transform the previous version into a new one is statistically smaller than the number of objects in the model graph. With this approach the conflict resolution might be better achieved by partially preserving the developer's intensions of the operations in conflict. The strong disadvantage of this approach is editor

(cc) BY-SA

dependence. Besides recording atomic operations, the record of composite operations could be done.

The state-based approach is based solely on the information contained in the different states of a document and does not take the knowledge about the operations into account that transformed one state into another. The states of to documents to be merged are compared in order to generate the set of differences between them. A state-based comparison requires a match function which determines whether the elements of the compared artefact correspond to each other. State-based operation detection could be done based on the content of the model elements or on unique identifiers. This approach is more efficient in merging small model graphs.

3.3.2 Conflict detection and resolution

Although conflict detection and resolution as well as model merging are not in the focus of ARTIST we summarize these topics for completeness.

Altmanninger et al. [17] give a generalized model versioning process consisting of the three phases: *"In the change detection phase the performed modifications on two working copies of one model are identified. The detected changes build the basis for the two subsequent phases, the conflict detection and inconsistency detection. In the second phase conflicts are detected by analysing concurrent changes solely, whereas in the third phase consistency problems are revealed which would occur in the merged model incorporating all changes."* After conflicting changes have been identified, they have to be resolved to produce a consolidated new version that captures all changes made in the different models. Although in simple cases, conflict resolution can be performed automatically e.g. by applying a rule based approach in general conflict resolution will have to be done manually. This process can be supported by showing the different versions side by side so that the user can choose the version to keep on a conflict to conflict basis. Non-conflicting changes can be merged automatically.

3.3.3 Overview of existing solutions

Brosch et al [15] give an overview of existing model versioning solutions that we still consider current and comprehensive. They review the current systems and analyse their capabilities with regard to aspects of conflict detection and resolution mentioned in the previous section. We have added information about the granularity of versioning and the support for creating branches of versions and tagging of versions (see Table 1).

(cc) BY-SA

Table 1: Comparison of current Model Versioning Systems (based on [23]).

	Operation Detection						Conflict Detection				Reso-lution		Versioning			Reuse in ARTIST	
	Operation recording	Model differencing		Composite Operations	Adaptability		Operation-based Conflicts		Inconsistencies	Adaptability	Graphical visualization	Adaptable Resolution Strategies	Branching support	Tagging support	Granularity of versioning (m=model, f=fragment, e=element)	Available for Artist	Implementation exists
		UUID	Content		Match	Operations	Atomic	Composite									
ADAMS [18]	×	✓	✓	×	×	×	✓	×	×	~	×	×	×	×	e	×	✓
CoObRA [19]	✓	×	×	✓	×	×	✓	×	~	×	✓	×	×	×	e	~	✓
EMF Compare [20]	×	✓	✓	~	×	~	✓	×	×	~	×	×	×	×	n/a	✓	✓
EMFStore [21]	✓	×	×	✓	×	~	✓	~	✓	×	×	×	✓	×	f	✓	✓
IBM Rational Software Architect [4]	×	✓	✓	×	×	×	✓	×	✓	×	✓	×	✓	×	?	✓	✓
SMOVER [22]	×	✓	×	×	×	×	~	×	×	~	×	×	?	?	?	×	✓
AMOR [15]	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	?	?	n/a	✓	✓

Legend	
✓	Feature applies
×	Feature does not apply
~	Feature partially applies
?	Unknown

[4] http://www.ibm.com/developerworks/rational/library/05/712/comp/index.html

(cc) BY-SA

3.4 MDE model repositories

MDE meta-models and models are usually stored in XML/XMI serialized form on the local disc in the developer's workspace. Complex projects like the migration / re-engineering projects conducted in ARTIST need facilities to support collaborative work with many diverse models and meta-models of considerable size. Conventional source code management systems like SVN or Git are not well suited to deal with models because they store models in textually serialized form. This format however does not capture the inherently hierarchical nature of models. This leads to problems when conflicting changes have to be merged (see Section 3.3).

This is where model repositories come into play. The usually provide storage and retrieval of models and meta-models, transaction support and versioning. A considerable number of existing MDE model repositories is already available. Since models and meta-models in ARTIST will be based on EMF/Ecore [23] the focus in this section is on Eclipse/EMF based repositories. The most relevant are described in the following sections.

3.4.1 CDO Model Repository

The Eclipse project CDO (Connected Data Objects) Model Repository[5] is a Java based distributed shared model repository for EMF models and meta-models. CDO is both a development-time model repository and a run-time persistence framework. It supports different deployment scenarios such as embedded repositories, offline clones or replicated clusters. *Multi user access* to the models is supported through the notion of repository sessions that support *secure authentication* of users.

CDO is special in that it allows different users to work on models *simultaneously* where changes are instantly synchronized in the different instances when a model is saved. An application can receive notifications about remote changes to the object graph and the collaboration features can be extended by plugging custom collaboration handlers into the asynchronous CDO protocol.

Scalability is achieved transparently by loading single objects on demand and caching them in the application. Multiple transactions can be opened from within a single session and they all share the same object data until one of them modifies the graph. This ensures that multiple threads of the application can access and modify the object graph without worrying about the synchronization details. The arising commit conflicts are handled as if they are between different sessions.

Offline work with your models is supported either by creating a clone of a complete remote repository, including all history of all branches, or by checking out a single version of the object graph from a particular branch point of the repository into a local CDO workspace.

3.4.2 Teneo

The Eclipse project Teneo[6] is a runtime database persistency solution for EMF and integrates EMF with Hibernate or EclipseLink. It supports automatic creation of EMF to Relational Mappings. EMF Objects can be stored and retrieved using advanced queries (HQL or EJB-QL).

[5] http://www.eclipse.org/cdo
[6] http://wiki.eclipse.org/Teneo

(cc) BY-SA

Teneo supports persistence of virtually every Ecore model without additional manual mapping work.

3.4.3 EMFStore

The Eclipse project EMFStore[7] provides a model repository (server) for sharing instances of EMF models. It includes a *version control* system like SVN or CVS designed especially for models and supports *merging* and *conflict detection* more effectively than text based version control systems, as pointed out in section 3.3.3. EMFStore follows the checkout/update/commit interaction paradigm of SVN and CVS, but, in contrast, if a change is performed on a model instance, this change should be immediately persistent without the need to manually save the model instance to a resource. If two clients alter the same data in a model instance, EMFStore supplies interactive merge user interfaces to support users in model merging.

There is also an *offline mode* support, where clients do not need to connect to the repository other than to update or commit changes. EMFStore enables supportive tooling, including an EMFStore Browser, History Browser and Model Element Editor. The EMFStore server can be applied for EMF Model instances and can be integrated in any kind of existing tool.

3.4.4 ModelBus

ModelBus[8] [24] is a model-driven tool integration framework which allows to build integrated tool chains based on well-established MDE technologies by increasing the consistency between the artefacts produced in a development process and by facilitating the communication between the tools involved. It allows connected modelling tools to access models using a unified mechanism for model identification. This enables tool integration cross application handling of modelling artefacts.

ModelBus has a model repository component that permits concurrent work on different parts of a model. This repository supports *model versioning*, check-out of partial models and coordinated *merging* of model versions and model fragments. All the changes applied to a model by other developers are tracked by ModelBuses notification system.

ModelBus has been developed in the European Project MODELPLEX (see Section 3.6.2).

3.4.5 EMFTrace

EMFTrace[9] is built upon the EMF Store repository, which is used as storage for models and traceability links. EMF Trace extends the underlying repository by integrating several import interfaces for different CASE tools and automated *traceability detection* techniques. The explicit modelling of dependencies and traceability links of different types is supporting evolutionary changes by impact analysis, early evaluation of quality flaws, and better comprehension.

EMFTrace uses a rule based approach to traceability detection [25]. Rules can build different kinds of traceability links that allow performing different types of analyses, including impact analysis and traceability recovery. A rule consists of a query to select appropriate model elements, an action that creates traceability information and a condition that defines if the rule should perform its task.

[7] http://eclipse.org/emfstore
[8] http://www.modelbus.org/modelbus
[9] http://sourceforge.net/p/emftrace/wiki/Home

(cc) BY-SA

3.4.6 Morsa

Morsa [26] is a research prototype of a model storage engine. It uses a document based NoSQL database (MongoDB) as storage backend that is well suited to take account of the highly interlinked nature of models. The design goals of Morsa were to provide model storage that

- scales for large models
- integrates seamlessly into Eclipse by implementing the Resource interface
- does not need to pre-process models or generate code from meta-models

Morsa focusses on model storage and does not provide versioning or conflict detection and resolution.

3.5 Evolution

A repository like ours involves a large number of complex application artefacts such as models, meta-models or transformations. The dynamic nature of their knowledge results in the fact that the artefacts continuously change over time. Therefore, a method for efficiently coping with changes in an artefact and consequently, i.e. artefact evolution, is essential for a repository or knowledge base such as the one we are creating. This is a challenging task since changes may have to be propagated, correctly and efficiently, to dependent artefacts or applications. Ideally, dealing with evolution should require as little manual work as possible.

The problem of evolution has been a long term research topic in various domains and still continues to be an important research topic today. It can be seen as one of the key activities in the overall repository maintenance task. Evolution has been investigated in a variety of diverge research domains (e.g., software evolution in the field of software engineering [27]). In ARTIST we plan to investigate the application of results from the fields of schema and ontology evolution to the evolution of MDE artefacts and especially meta-models.

3.5.1 Schema Evolution

Database schema evolution and versioning has been an active research area for a long time. It has been investigated primarily for object-oriented databases and to a lesser extent for relational databases. According to [28], the difference between (database) schema evolution & versioning is defined as follows:

- Schema evolution is the ability to change a database schema without the loss of existing data.
- Schema versioning is the ability to query all the data through user-definable version interfaces.

In this subsection, we will first consider state of the art in commercial database systems and their support for schema evolution. We then consider some recent research efforts on this topic.

The approach usually taken by the commercial DBMS vendors such as Oracle [29] and Microsoft SQL Server [30] is to rely on a Data Definition Language (DDL) to perform schema evolution. The schema evolution primitives in this DDL are atomic in nature so only simple changes can be described directly. For example, individual tables, columns and/or constraints may be added or dropped. Additionally, properties of a single object may be changed. However, one cannot specify more complex changes. These are accomplished by

(cc) BY-SA

bundling a sequence of atomic changes inside a transaction that can then be undone via a rollback. Changes in Oracle and SQL Server are specified incrementally. However, Oracle has a mechanism called redefinition that allows someone to specify multiple schema or semantic modifications on a table. The evolution mapping is represented as set of incremental changes using the SQL DDL. Oracle has a comparable functionality for two schema versions. The diff is also expressed using the DDL. SQL Server does not currently support such feature. In both systems, there is little support for change propagation. Oracle takes the non-transactional approach where instances migrate if no other objects depend on the changed table. SQL Server takes the transactional approach and has automatic instance translation for objects with no dependencies. There is typically no support for multiple schema and database versions. Once a change script has been executed, the previous version is gone. However, Oracle has a so-called editions feature with support for multiple versions of non-persistent objects. SQL Server utilizes an internal versioning mechanism only during instance migration. Both vendors have some tooling that operated above the database. The Oracle Change Management Pack compares schemas, bundles changes and predicts errors. The SSMS from Microsoft created change scripts from designer changes.

One important research effort is the tool called PRISM [31]. The two primary goals are to specify schema evolution and provide support for multiple schema versions. It includes an evolution mapping language using Schema Modification Operators (SMOs) which closely resembles the DDL. However, the unit of change is at a much broader scope than the DDL enabling both simple and complex changes. Furthermore, every statement is expressed by means of a forward and backward translation respectively doing and undoing the mapping. Changes in PRISM are specified incrementally. The evolution mapping is represented by means of formal logic and SQL mappings. In terms of update propagation, the focus is on automatic query rewriting when possible. Notifications will be given if queries become invalid. PRISM does have support for versioning by generating views supporting both backward and forward compatibility.

The main focus of work of the approach called HECATAEUS [32] is on the propagation of evolution primitives between dependent database objects. While the commercial systems have tight constraints in this regards, this approach allows for a fine grained control over when to propagate schema changes to an object, to the queries, statements and views that depend on it. Changes are typically represented by means of SQL DDL statements and in an incremental manner. The evolution mapping is just represented as set of incremental changes using the SQL DDL. The propagation of updates is done through a construct called evolution policies. They determine how and when to automatically update dependent objects such as views and queries. Each policy has three enforcement options: propagate, block and prompt. Underlying, they use a graph model, termed Evolution Graph, in order to regulate schema evolution.

A specific type of schema evolution but very much different from the database schema evolution is that of XML schema evolution. Since notions such as DTD and XML Schema have emerged, also in research people began investigating evolution building on these notions. The commercial database vendors provide support the storage of XML data based on an XML schema. Limited support is added to cope with changes to those schemas. Oracle offers a very flexible copy-based mechanism to handle evolution and a limited in-place evolution mechanism. The evolution mapping is represented as a set of changes via the diffXML language. The update propagation can be specified using XSL transformations. An important tool to mention here is the Altova Diff Dog [33], a commercial tool for differencing XML schemas. Here, mainly simple changes (e.g. rename, reordering…) supported. By specifying both the old and new version of the schema, the tool can retrieve the difference semi-

(cc) BY-SA

automatically. The result is the evolution mapping represented as a set of element-element correspondences. The propagation of updates is done via the execution of XSL transformations generated by the tool.

3.5.2 Ontology Evolution

For ontologies, however, we cannot distinguish between evolution and versioning as we did for schemas. Multiple versions of the same ontology are bound to exist and must be supported. Ideally, developers should maintain not only the different versions of an ontology but also some information on how the versions differ and whether or not they are compatible with one another. Hence, typically ontology evolution and versioning are combined into a single concept defined as the timely adaptation of an ontology to the arisen changes and the consistent propagation of these changes to dependent artefacts [34].

At this point we would like to clarify the terms evolution, versioning and traceability used in this document:

- *Traceability* means the recording of relationships between artefacts. It can have different dimensions depending on the kind of relationship to be tracked. Vertical traceability tracks the relationship of artefacts across different phases of the development cycle (e.g. from requirements to tests) while horizontal traceability focusses on relationships between different artefacts in the same development phase (e.g. dependencies) [35].
- *Versioning* is the tracking of different states of one artefact over time.
- *Evolution* builds on versioning and traceability but focusses less on the state of an artefact at a given moment in time and more on the changes that were performed to go from one version of an artefact to another. Evolution also takes the effects of changes on other related artefacts into account.

The remainder of this subsection will cover some of the most important research efforts in the field of ontology evolution.

In [36] and [37] a component-based approach supporting ontology evolution for the OWL ontology language is described. This framework consists of the following four main components: a meta-ontology of change operations, complex change operations, transformation sets, and the specification of relations between different ontology versions. The meta-ontology of changes defines a set of basic change operations that the different users of this framework need to agree upon. A standard set of basic changes is proposed for the OWL ontology language. Besides basic change operations, the framework also allows complex change operations. Complex change operations provide a mechanism for grouping a number of basic change operations that together constitute a logical entity. Another important element of the framework is the notion of a transformation set. A transformation set is a set of change operations that specifies how an old version of an ontology can be transformed into a new version. A transformation set may contain both basic and complex change operations. A minimal transformation set contains only change operations that are necessary and sufficient to specify a transformation. The fourth component of the framework is the specification of how two ontology versions are related. This specification can be seen as a mapping ontology.

The focus of attention of the framework is mainly on the problem of identifying changes and providing semantic specifications for the changes between different versions. The framework provides methods for finding changes when only two versions of an ontology are available, and methods for deriving additional change information. The framework itself

(cc) BY-SA

doesn't take into account the problem of consistency maintenance after changes are applied.

Another ontology evolution approach is proposed by [38][39]. The authors propose a six-phase ontology evolution framework for the KAON ontology language consisting of the following phases: change capturing phase, change representation phase, semantics of change phase, change propagation phase, change implementation phase, and change validation phase. The first phase, the change capturing phase, captures changes either from explicit requirements or from the result of change discovery methods. They distinguish structure-driven change discovery (e.g., a concept without Properties is a candidate for deletion), data-driven change discovery (e.g., a concept without instances may be deleted), and usage-driven change discovery (e.g., concepts that are never queried may be deleted). The second phase, the change representation phase, distinguishes between elementary and composite changes, similar to the previous approach. The purpose of the semantics of change phase is to resolve possible inconsistencies introduced after a change. The task of the change propagation phase of the ontology evolution process is to bring automatically all dependent artefacts into a consistent state after an ontology update has been performed. The role of the next phase of the ontology evolution framework, the change implementation phase is (1) to inform an ontology engineer about all consequences of a change request, (2) to apply all the changes and (3) to keep track about performed changes. Finally, the change validation phase enables justification of performed changes and undoing them at user's request. It has as purpose to increase the usability of the evolution process. Compared to the previously discussed ontology evolution framework, this framework concentrates more on ontology consistency checking and inconsistency resolving, although the focus is on structural consistency. The framework, however, doesn't provide mechanisms for detecting complex changes.

The authors of [40][34] have taken the previously discussed framework as a basis and extended it to support evolution for the OWL ontology language, instead of solely supporting the KAON ontology language. Their work has mainly focused on handling inconsistencies introduced by a change to an ontology. Checking consistency and resolving inconsistencies for OWL is quite different then for the KAON ontology language mainly due to the difference between respectively the open-world vs. closed-word assumption taken by both languages. They focus on different forms of consistency, including structural consistency, logical consistency and user-defined consistency. In the occurrence that an inconsistency cannot be resolved, they propose an approach to reason with an inconsistent ontology. Another extension concerns the support for collaborative and usage-driven evolution of ontologies. This extension targets personal ontologies i.e., ontologies that are shared among different users, but that are customized and personalized to a specific user. So, every user may have a slightly different version of the original ontology. Their approach allows users to annotate concepts and axioms of their ontology with a rating (both positive and negative) indicating the importance that a user attached to the concept or axiom. Based on the ratings given by various users, recommendations of ontology changes are suggested to other users.

In [41] and [42], an ontology evolution approach is proposed based on detecting change using a so-called version log. The authors propose a framework consisting of two parts corresponding to two different tasks. The first task handles the evolution process of an ontology as consequence of a change request by an ontology engineer. We use the term evolution on request to refer to this task. This task consists of the following phases: change request, consistency maintenance, change detection, change recovery, change implementation, cost of evolution, and version consistency. The second task handles the

BY-SA

evolution process of a depending artefact as consequence of changes to an ontology it depends on. We refer to this second task with the term evolution in response. It consists of three phases: change detection, cost of evolution and version consistency. A combination of a version log and evolution log is used in this approach. The version log stores for each concept ever defined in the ontology, the different versions it passes through during its life cycle: from its creation, over its modifications, until its retirement. The purpose of an evolution log on the other hand is to give an overview of the evolution of an ontology by listing all changes that have occurred. It serves maintainers of depending artefacts in understanding the changes occurred, and is therefore a great benefit in the decision whether to update or not. The difference between the version log and the evolution log: the former lists the different versions of the ontology concepts, the latter lists the interpretations of these versions in terms of conceptual change definitions. The typical use of evolution log is therefore extended with, besides requested and deduced changes, occurrences of conceptual change definitions detected during the change detection phase. In addition, a Change Definition Language, based on temporal logic, is introduced to allow the specification of change definitions i.e., to state the conditions a modification to an ontology should satisfy to be considered as an occurrence of a particular change definition. The Change Definition Language makes it possible to define changes in a declarative fashion. In ARTIST we will try to apply this approach to the evolution of modelling artefacts.

3.6 Relevant research projects

3.6.1 Momocs – XSM Knowledge Base Repository

Momocs - Methodology and related tools for fast reengineering of complex systems [43] was a project co-funded by the European Commission under the Sixth Framework Programme. Momocs provided a software modernisation methodology called XIRUP (eXtreme end-User dRiven Process) and supporting tooling, XIRUP tool suite, designed to drive the modernisation of legacy complex systems. XIRUP methodology defines a reusable modernisation methodology, that is, the methodology itself is reusable for a wider range of modernisation scenarios. Moreover, some of the artefacts obtained and/or consumed during the XIRUP modernisation phases, when the methodology is applied to some concrete modernisation projects, are reusable in other similar projects, either because the modernisation challenges are similar or because they accomplish similar modernisation tasks. In order to foster this reusability, reusable artefacts have to be accessible throughout the XIRUP modernisation tools. For this purpose, Momocs shipped the XSM[10] Knowledge Base Repository [44] as part of the Momocs XIRUP tools suite, a repository of XIRUP artefacts seamlessly integrated (through the Eclipse workbench) with other XIRUP suite tools: the XSM Model Editor and the XSM Transformation Tool.

XSM Knowledge Base Repository is a container for XIRUP artefacts produced during the different phases of a modernisation process, that converts the legacy system into a modernised one, to promote their reuse whenever is possible, in the same or a different modernisation project. Examples of reusable artefacts stored into this repository are: XSM models, such as models describing the legacy system or the modernised one, XSM M2M transformations or XSM transformation historic data, including set of transformation mappings. As a result of applying a M2M transformation to a XSM, it is transformed into another XSM, constituting a XSM model transformation mapping. Momocs repository traces transformation mappings, displayed as a transformation historic, allowing the user to navigate through the model evolution process.

[10] XIRUP System Model

(cc) BY-SA

Other operations supported by the XSM KB Repository are: managing the repository (e.g. defining the repository structure), storing/retrieving/deleting artefacts, browsing and searching the repository, browsing the model transformation history, attach additional files to artefacts or add notes to artefacts.

XSM KB Repository can be organised by the user according to a hierarchical classification (e.g. folder structure) determined by the user and annotated by attaching semantic metadata selected from domain specific ontologies. XIRUP artefacts stored in the repository can be semantically annotated in a similar way than the repository structure is.

Semantic annotations are later exploited by the XSM KB repository to support semantic-driven searching of artefacts, which relies on semantic reasoning. Best matching artefacts are ranked and shown to the user, sorted by matching score.

Apart from semantic searching, the repository supports keyword matching, including advance searching based on complex logical keyword relations. Additionally, the repository can be browsed through its structure, permitting a manual discovery and retrieval of artefacts. Browsing supports content sorting, filtering, DND[11] re-arrangement and comparison (restricted to model comparison).

Different views are included in the XSM KB repository, assisting users to manage, browse and query the repository content, such as the KB Repository Explorer, the Ontology Browser, the Semantic Annotation Browser, the KB Historic Explorer, the KB Query results, the KB Attachments or the KB Notes list.

Technically, XSM KB repository is provided as an Eclipse RCP client application seamlessly integrated within the XIRUP tool suite. The repository itself was implemented on top of an eXist XML database [45].

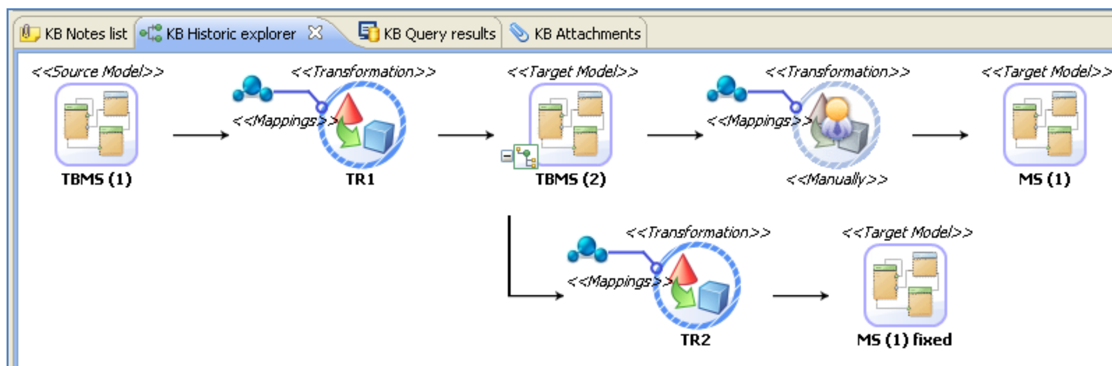

Figure 1: MOMOCS XSM KB Repository Model Historic View

3.6.2 MODELPLEX

The MODELPLEX IST-FP6 European project was a three years and a half Integrated Project that aimed at providing an open modelling solution dedicated to the quality and productivity improvement of complex software systems development. As part of this project, interesting works have been conducted on two topics also relevant in the context of the ARTIST Repository: Model Management and Model Traceability. In both cases, they could be reused and/or extended within ARTIST, or simply just considered as a potential good source of inspiration.

[11] Drag and Drop

(cc) BY-SA

As introduced in Section 3.1, a Global Model Management (GMM) approach has been developed and proposed as a possible solution for dealing with model management issues. This has resulted in the expression of some overall principles and methodology [46], but also in the implementation of a related concrete tooling support [47].

According to what has been stated in Section 3.2, and based on the GMM approach mentioned just before, some support for inter-model traceability has also been developed as part of MODELPLEX. The main underlying concepts and an initial tooling support have been proposed and published accordingly [48].

Another result of MODELPLEX is the MDE support infrastructure *ModelBus* that is described in Section 3.4.4.

3.6.3 SeCSE – Service Centric System Engineering

SeCSE [49] – *Service Centric System Engineering* is an Integrated Project co-funded by the European Commission under the Sixth Framework Programme. This project has provided a set of methods, tools and techniques for system integrators and service providers to develop and deploy service-centric applications. In particular, SeCSE has realized an integrated development and execution environment covering the modelling of the service requirements, the specification of quality of service (QoS) aspects and providing support for using these specifications within service discovery and binding mechanisms.

One of the aspects addressed by the SeCSE project concerns the specification of the services to support service discovery. In this context, registries play a key role since they provide searchable directories where service descriptions can be published. UDDI [50] and ebXML [51] are the most widely adopted specifications for service registries, however, during the SeCSE investigations, these kinds of registries have been considered insufficient to deliver the full range of run-time service discovery capabilities expected by the project. Indeed, it was noticed that UDDI and ebXML registries permit to store only a subset of the information about different service properties that may be required (e.g. information about the semantics of service operations or their exception behaviour is not supported). Moreover, it was noticed that their retrieval functionalities were not enough to support all the SeCSE tools. Consequently, a *SeCSE Registry* has been implemented within the project. The registry is a repository of service descriptions and facets complementing an UDDI registry that provides a set of operations for storing, modifying and retrieving the content of the repository.

In the realization of the SeCSE Registry the concept of faceted specification, which allows a service to be modelled according to a set of orthogonal partial description, plays a central role. Since one of the functionality that the ARTIST Repository should support is the browsing of artefacts by facetted classification the work done within the SeCSE project can be a source of inspiration. In the SeCSE approach, a *facet* is the model for the specification of the services. It represents a property of a service and it is used to express both functional and non-functional information (e.g. service interface, service behavior) and other additional information (e.g. user ratings measured QoS, usage history). In particular, a facet is a couple consisting on a *facet type* and a *facet instance*. A facet type is based on a XML Schema describing its structure, while the facet instance is the XML document describing the facet according to the XML Schema (i.e. the facet instance is considered as a reference implementation of the facet). The set of facet types is not fixed, but new ones can be defined. Therefore, in the SeCSE project the services have been modelled using a Service Specification Language based on facets providing a more wide and flexible way of describing services with respect to the one offered by UDDI. Moreover, the SeCSE project has improved

(cc) BY-SA

the existing registries approaches by stressing the idea of *federation* of information repositories. The registry can be organized in federations (i.e. collections of autonomous, possibly heterogeneous, and cooperating registries) resulting from an agreement made by organizations running service registries to achieve a joint aim (e.g., being focused on a similar topic, having some trust relationship, gain scalability, etc.). Federation interconnects different registries by exploiting a peer-to-peer (P2P) approach. This approach suits well the idea of federation since it supports decentralization, distribution, dynamic configuration, and proper communication protocols among peers. Federations are used to propagate information (e.g., service requests or service advertisements) among different registries. The registry classification scheme is shared and made available within each federation. In addition, information can be propagated according to a pull (i.e., on demand) or a push (i.e., event based publish-subscribe) approach.

3.6.4 SLA@SOI

SLA@SOI – *Empowering the service industry with SLA-aware infrastructures*[12] is an Integrated Project co-funded by the European Commission within the Seventh Framework Programme. Its aim has been to deliver and showcase an innovative open Service Level Agreement (SLA) Management Framework providing holistic support for service level objectives and enabling a dynamic SLA-aware market for European service providers. Within the SLA@SOI project the *SLA Registry* (SLAR) and the *SLA Template Registry* (SLATR) components, providing storing mechanisms, have been developed. The SLAR has been implemented in order to save established SLAs. Indeed, this component provides functionalities to save and restore agreement documents and hierarchies of such documents. The SLATR, instead, enable to store and browse SLATemplates (SLATs). Within the SLA@SOI project the SLAT provides a model for service level agreement encapsulating the functional properties of services (e.g. interfaces and operations), standard aspects (e.g. availability, response time), actual SLA aspects (e.g. on effective dates, involved parties) and business terms (e.g. pricing, billing).

One of the key innovations in SLA@SOI has been the development of the fully queriable SLATR, including the specification of a suitable SLAT query language and the development of query resolution algorithms. The negotiation of a SLA typically begins with a SLAT describing the service and the kinds of guarantees are offered (i.e. a SLAT is an enriched kind of service description). SLATs serve as both the means by which a provided describes the services they offer, and the means by which customers can describes their requirements. Within the SLA@SOI architecture the SLAT are developed by the service providers and advertised through the SLATR. Therefore, in the SLA@SOI approach the SLATR has been conceived as an extended kind of service description registry enabling SLA-aware service discovery, without foregoing the traditional views of discovery, based on the purely functional properties of the service. In particular, the aspects characterising the SLATR that have been implemented are:

i. Provide a persistent and accessible store for SLAT
ii. Enable the QoS-based discovery of SLAT
iii. Enable the metadata-based retrieval of SLAT
iv. Provide the validation of SLAT content and SLA conformance using domain-specific terms formally classified according to a meta-model specifying the role played by each term in an SLA
v. Provide support for asynchronous notifications (e.g. notifications in case of (un)successful registration events)

[12] www.sla-at-soi.eu

(cc) BY-SA

From the ARTIST Repository point of view, the metadata-based retrieval functionality provided by the SLATR is one of the aspects that can be interested to explore to understand how it has been achieved. Every registered SLAT has associated some metadata which are essentially a dictionary of key/value property pairs. Among other things, these metadata include information, identifying the provider of a SLAT, its creator and the time at which it was registered. A query over metadata can be readily expressed as a constraint on property values. The result of a metadata query is a list of (UUID) identifiers of those SLATs whose metadata meets the expressed constraints.

The SLATR exposes its functionalities as a single interface which is accessible by the SLA@SOI internal components and, at the same time, the various query mechanisms are also encapsulated as a web service in order to be used by external components.

3.6.5 Morse

The Model-Aware Repository and Service Environment (Morse) [52] is a service-based environment for the storage and retrieval of models and model-instances at both design- and runtime that was supported by the FP7 project COMPAS[13]. Its main goal is to support the development of model aware services by enabling the services to reflectively access the models that they were generated from.

The Morse repository identifies models by Universally Unique Identifiers (UUID) and provides versioning capabilities so that models can be manipulated at runtime and new and old versions of the models can be maintained in parallel. This makes it possible for generated services to access different versions of models (e.g. the current model or the model they were generated from).

[13] http://www.compas-ict.eu

(cc) BY-SA

4 Conclusions

The requirements documented in this deliverable constitute the basis and starting point for design, architecture, implementation and subsequent evaluation of the ARTIST artefacts repository and marketplace.

The requirements are summarized in Table 2. The table also contains a first estimate on the priorities of the requirements and an estimate on the technical complexity and feasibility of implementing these requirements. The priorities have been determined from a users' perspective based on the input and feedback of the consortium partners. The feasibility was estimated based of the following criteria:

- Availability of techniques and tools that can be reused to realize the requirement
- Availability of theoretical solutions/techniques from the literature
- The estimated gap between the required features and the tools/techniques that could potentially be reused
- Estimated development complexity

The presented requirements will be iteratively detailed, refined and reviewed during the project following the agile development process of ARTIST. The WP10 development partners will try to implement the requirements as far as possible taking the following criteria into account:

1. Requirements with higher priority will be considered first.
2. Requirements considered affordable in terms of their technological feasibility (that is, whether or not the current technology allows their reasonable implementation), with affordable development efforts and available resources within the project lifetime, will be also faced first.

(cc) BY-SA

Table 2: Overview of requirements and first estimate on priorities and technical feasibility.

ID	Requirement	User priority	Technical feasibility
Common functional requirements			
CF-1	Support for common MDE artefact types	High	High
CF-2	Support the organization of a big number of artefacts	High	Medium
CF-3	Efficient retrieval of artefacts	High	Medium
CF-4	Record user feedback	Medium	High
CF-5	Access control	High	Medium
CF-6	Version control for artefacts	Medium	Medium
CF-7	Eclipse integration	High	High
CF-8	Integration with external tools	Medium	High
Functional repository requirements			
RF-1	Traceability between artefacts	High	Medium
RF-2	Visualization of relationships between artefacts	Medium	Medium
RF-3	Support for artefact evolution	High	Medium
Functional Marketplace requirements			
MF-1	Publication of artefacts	High	High
MF-2	Web based user interface	High	High
MF-3	Support for commercial artefacts	Low	Medium
Non-functional requirements			
NF-1	Ease of use	High	Medium
NF-2	Extensible architecture	Medium	Medium
NF-3	Sufficiently large initial artefact population	High	n/a

4.1.1 Evaluation of the state of the art

Based on the state of the art overview possible approaches and software components have been identified that have to be investigated on a more technical and detailed level in the coming months.

Model management

The Global Model Management approach seems to be a good basis for the information model of the repository.

Traceability

The traceability support of the Global Model Management approach seems to be a good starting point but other approaches such as the rule based approach of EMFTrace should be considered as well.

(cc) BY-SA

Model versioning

If versioning is not covered by other tools the most adaptable tools for conflict detection and resolution seems to be EMFCompare and AMOR. Given that the ARTIST repository focusses on more stable and reusable artefacts, it will be investigated, if fully fledged version control is needed or if a simpler solution is sufficient.

Model repositories

To be useful for ARTIST a model repository should fulfil the following criteria that were inspired by [26]:

- Support for versioning
- Scalability (memory usage and processing time) for large models
- Transparent tool integration (use of standard interfaces to access models)
- Uniform support for models and meta-models (no code generation should be necessary to add new types of models, no pre-processing should be needed)

CDO is very mature and seems to scale well but its tool integration is not transparent since code has to be generated for new meta-models. Teneo focusses on storage and has problems with large models due to the mapping on a relational data model. EMFStore and EMFTrace provide versioning and seem to be adaptable and easy to integrate, but their scalability behaviour is unclear and the tool integration seems not to be completely transparent. The ModelBus repository is based on SVN and it therefore supports versioning. It offers a web service interface for repository access but information on scalability was not available. Morsa seems to provide the most transparent tool integration, scalability and uniform model and meta-model support but it does not provide versioning.

The mentioned repositories will be further investigated.

Evolution

Concepts from the evolution of ontologies and schemata can be adapted for use with meta-models and models.

(cc) BY-SA

References

[1] J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez, “Modeling in the Large and Modeling in the Small,” in *Model Driven Architecture*, U. Aßmann, M. Aksit, and A. Rensink, Eds. Springer Berlin Heidelberg, 2005, pp. 33–46.

[2] J. Bézivin, F. Jouault, and P. Valduriez, “On the need for megamodels,” in *Best Practices for Model-Driven Software Development workshop (Proceedings of the OOPSLA/GPCE 2004)*, Vancouver, BC, Canada, 2004.

[3] J. Favre and T. NGuyen, “Towards a Megamodel to Model Software Evolution through Transformations,” in *SETRA Workshop, Elsevier ENCTS*, 2004, pp. 59–74.

[4] M. Barbero, F. Jouault, and J. Bézivin, “Model Driven Management of Complex Systems: Implementing the Macroscope’s Vision,” in *Proceedings of the 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, Washington, DC, USA, 2008, pp. 277–286.

[5] M. Fritzsche, H. Bruneliere, B. Vanhooff, Y. Berbers, F. Jouault, and W. Gilani, “Applying Megamodelling to Model Driven Performance Engineering,” in *Engineering of Computer Based Systems, 2009. ECBS 2009. 16th Annual IEEE International Conference and Workshop on the*, 2009, pp. 244 –253.

[6] A. Vignaga, F. Jouault, M. C. Bastarrica, and H. Brunelière, “Typing artifacts in megamodeling,” *Software & Systems Modeling*, Feb. 2011.

[7] R. Salay, J. Mylopoulos, and S. Easterbrook, “Using Macromodels to Manage Collections of Related Models,” in *Proceedings of the 21st International Conference on Advanced Information Systems Engineering*, Berlin, Heidelberg, 2009, pp. 141–155.

[8] I. Santiago, Á. Jiménez, J. M. Vara, V. De Castro, V. A. Bollati, and E. Marcos, “Model-Driven Engineering as a new landscape for traceability management: A systematic literature review,” *Information and Software Technology*, vol. 54, no. 12, pp. 1340–1356, Dec. 2012.

[9] “Agility - OBEO.” [Online]. Available: http://www.obeo.fr/pages/agility/en. [Accessed: 11-Jan-2013].

[10] “Products - itemis AG.” [Online]. Available: http://www.itemis.com/itemis-ag/language=en/43237/products. [Accessed: 11-Jan-2013].

[11] “Eclipse MoDisco.” [Online]. Available: http://www.eclipse.org/MoDisco/. [Accessed: 30-Jan-2013].

[12] “Acceleo.” [Online]. Available: http://www.eclipse.org/acceleo/. [Accessed: 11-Jan-2013].

[13] “Eclipse Modeling - M2T - Home.” [Online]. Available: http://www.eclipse.org/modeling/m2t/?project=xpand. [Accessed: 11-Jan-2013].

[14] F. Jouault, B. Vanhooff, H. Bruneliere, G. Doux, Y. Berbers, and J. Bezivin, “Inter-DSL coordination support by combining megamodeling and model weaving,” in *Proceedings of the 2010 ACM Symposium on Applied Computing*, New York, NY, USA, 2010, pp. 2011–2018.

[15] P. Brosch, G. Kappel, P. Langer, M. Seidl, K. Wieland, and M. Wimmer, “An Introduction to Model Versioning,” in *Formal Methods for Model-Driven Engineering*, M. Bernardo, V. Cortellessa, and A. Pierantonio, Eds. Springer Berlin Heidelberg, 2012, pp. 336–398.

[16] A. De Lucia, F. Fasano, G. Scanniello, and G. Tortora, “Concurrent Fine-Grained Versioning of UML Models,” in *13th European Conference on Software Maintenance and Reengineering, 2009. CSMR ’09*, 2009, pp. 89 –98.

[17] K. Altmanninger, P. Brosch, G. Kappel, P. Langer, M. Seidl, K. Wieland, and M. Wimmer, “Why model versioning research is needed⁈ an experience report,” in *Proceedings of the MoDSE-MCCM 2009 Workshop@ MoDELS*, 2009, vol. 9.

BY-SA

[18] A. De Lucia, F. Fasano, G. Scanniello, and G. Tortora, "Concurrent Fine-Grained Versioning of UML Models," in *13th European Conference on Software Maintenance and Reengineering, 2009. CSMR '09*, 2009, pp. 89 –98.

[19] C. Schneider, A. Zündorf, and J. Niere, "CoObRA-a small step for development tools to collaborative environments," in *Proc. of the Workshop on Directions in Software Engineering Environments (WoDiSEE), Edinburgh, Scotland, UK*, 2004.

[20] C. Brun and A. Pierantonio, "Model differences in the Eclipse Modeling Framework," *Upgrade, Special Issue on Model-Driven Software Development IX*, p. 2008.

[21] M. Koegel, M. Herrmannsdoerfer, O. von Wesendonk, and J. Helming, "Operation-based conflict detection," in *Proceedings of the 1st International Workshop on Model Comparison in Practice*, New York, NY, USA, 2010, pp. 21–30.

[22] T. Reiter, K. Altmanninger, A. Bergmayr, W. Schwinger, and G. Kotsis, "Models in conflict - Detection of semantic conflicts in model-based development," in *proceedings of 3rd International Workshop on Model-Driven Enterprise Information Systems*, 2007.

[23] *EMF: Eclipse Modeling Framework*, 2nd ed., Rev. and updated. Upper Saddle River, NJ: Addison-Wesley, 2009.

[24] X. Blanc, M. P. Gervais, and P. Sriplakich, "Model bus: Towards the interoperability of modelling tools," *Model Driven Architecture*, pp. 900–900, 2005.

[25] M. Riebisch, S. Bode, Q.-U.-A. Farooq, and S. Lehnert, "Towards Comprehensive Modelling by Inter-model Links Using an Integrating Repository," 2011, pp. 284–291.

[26] J. Espinazo Pagán, J. Sánchez Cuadrado, and J. García Molina, "Morsa: A Scalable Approach for Persisting and Accessing Large Models," in *Model Driven Engineering Languages and Systems*, vol. 6981, J. Whittle, T. Clark, and T. Kühne, Eds. Springer Berlin / Heidelberg, 2011, pp. 77–92.

[27] M. W. Godfrey and D. M. German, "The past, present, and future of software evolution," in *Frontiers of Software Maintenance, 2008. FoSM 2008.*, 2008, pp. 129 – 138.

[28] C. Dyreson, F. Grandi, W. Käfer, N. Kline, N. Lorentzos, Y. Mitsopoulos, A. Montanari, D. Nonen, E. Peressi, B. Pernici, J. F. Roddick, N. L. Sarda, M. R. Scalas, A. Segev, R. T. Snodgrass, M. D. Soo, A. Tansel, P. Tiberio, and G. Wiederhold, "A consensus glossary of temporal database concepts," *SIGMOD Rec.*, vol. 23, no. 1, pp. 52–64, Mar. 1994.

[29] "Oracle XML Schema Evolution | Schema Evolution." [Online]. Available: http://se-pubs.dbs.uni-leipzig.de/node/972. [Accessed: 11-Jan-2013].

[30] R. Rankins, P. Bertucci, C. Gallelli, and A. T. Silverstein, *Microsoft SQL Server 2008 R2 Unleashed*. Pearson Education, 2010.

[31] C. A. Curino, H. J. Moon, and C. Zaniolo, "Graceful database schema evolution: the PRISM workbench," *Proc. VLDB Endow.*, vol. 1, no. 1, pp. 761–772, Aug. 2008.

[32] G. Papastefanatos, P. Vassiliadis, A. Simitsis, and Y. Vassiliou, "HECATAEUS: Regulating schema evolution," in *2010 IEEE 26th International Conference on Data Engineering (ICDE)*, 2010, pp. 1181 –1184.

[33] "Diff Merge Tool." [Online]. Available: http://www.altova.com/diffdog.html. [Accessed: 11-Jan-2013].

[34] P. Haase, and P. Haase, and L. Stojanovic, "Consistent Evolution of OWL Ontologies," 2005, pp. 182–197.

[35] G. Spanoudakis and A. Zisman, "Software Traceability: A Roadmap," *Handbook of Software Engineering and Knowledge Engineering*, pp. 395–428, 2004.

[36] M. Klein, D. Fensel, A. Kiryakov, and D. Ognyanov, "Ontology versioning and change detection on the Web," in *In 13th International Conference on Knowledge Engineering and Knowledge Management (EKAW02*, 2002, pp. 197–212.

CC BY-SA

[37] N. F. Noy, A. Chugh, W. Liu, and M. A. Musen, "A framework for ontology evolution in collaborative environments," in *Proceedings of the 5th international conference on The Semantic Web*, Berlin, Heidelberg, 2006, pp. 544–558.

[38] A. Maedche, B. Motik, L. Stojanovic, R. Studer, and R. Volz, "An Infrastructure for Searching, Reusing and Evolving Distributed Ontologies," in *In: Proceedings of WWW 2003*, 2003, pp. 439–448.

[39] L. Stojanovic, A. Maedche, B. Motik, and N. Stojanovic, "User-Driven Ontology Evolution Management," in *Proceedings of the 13th International Conference on Knowledge Engineering and Knowledge Management. Ontologies and the Semantic Web*, London, UK, UK, 2002, pp. 285–300.

[40] P. Haase, A. Hotho, L. Schmidt-thieme, and Y. Sure, "Collaborative and Usage-driven Evolution of Personal Ontologies," in *In Proc. Second European Semantic Web Conference (ESWC)*, 2005, pp. 486–499.

[41] P. Plessers and O. D. Troyer, "Resolving inconsistencies in evolving ontologies," in *In Proceedings of the Third European Semantic Web Conference (ESWC-2006)*, 2006.

[42] P. Plessers, O. De Troyer, and S. Casteleyn, "Understanding ontology evolution: A change detection approach," *Web Semant.*, vol. 5, no. 1, pp. 39–49, Mar. 2007.

[43] "www.momocs.org." [Online]. Available: http://www.momocs.org/. [Accessed: 11-Jan-2013].

[44] MOMOCS, "MOMOCS - Data Modernization Tool Implementation (XSM Knowledge Base Repository)," MOMOCS Project, Deliverable 5.1b, Sep. 2008.

[45] "eXist-db Open Source Native XML Database." [Online]. Available: http://exist-db.org/exist/index.xml. [Accessed: 11-Jan-2013].

[46] H. Bruneliere, J. Bezivin, M. Barbero, and N. Dowler, "Global Model Management Principles," MODELPLEX Project, Deliverable D2.1.a, Mar. 2008.

[47] H. Bruneliere and G. Doux, "Global Model Management Supporting Tool," MODELPLEX Project, Deliverable D2.11.b, Oct. 2009.

[48] H. Bruneliere, "Global Model Management Traceability Extension," MODELPLEX Project, Deliverable D3.2.d, Dec. 2008.

[49] "SeCSE » Home." [Online]. Available: http://www.secse-project.eu/. [Accessed: 11-Jan-2013].

[50] "UDDI | Online community for the Universal Description, Discovery, and Integration." [Online]. Available: http://uddi.xml.org/. [Accessed: 11-Jan-2013].

[51] "ebXML - Enabling A Global Electronic Market." [Online]. Available: http://www.ebxml.org/. [Accessed: 11-Jan-2013].

[52] T. Holmes, U. Zdun, and S. Dustdar, "MORSE: A Model-Aware Service Environment," in *Services Computing Conference, 2009. APSCC 2009. IEEE Asia-Pacific*, 2009, pp. 470–477.

CC BY-SA