

Institut für Architektur von Anwendungssystemen



Universität Stuttgart
Universitätsstraße 38
D – 70569 Stuttgart

Diplomarbeit Nr. 3251

**Framework
for Application Topology Discovery to
enable Migration of Business Processes
to the Cloud**

Jakob Krein

Studiengang:	Informatik
Prüfer:	Prof. Dr. Frank Leymann
Betreuer:	Dipl.-Inf. Tobias Binz
begonnen am:	27.10.2011
beendet am:	27.04.2012
CR-Klassifikation:	C.2.4, D.2.11, H.4.1, H.5.2

Abstract

Today, enterprises often use large and complex software systems to support their Business Processes. These software systems usually run over years and decades and have gone through various changes and modifications in order to be able to cope with changing business requirements. Successful software applications usually grow and evolve over time, and so does their architecture. The amount of modifications can reach dimensions where the resulting architecture of the software system has little in common with the originally designed one.

Knowing the overall architecture of the system is crucial for its management, especially when it comes to the migration of parts of the application or the application as a whole to another IT infrastructure, such as a *Cloud* infrastructure. Before an application can be migrated, its architecture has to be obtained in its current state, commonly known as the process of *Software Architecture Reconstruction (SAR)*. This is usually achieved by running different internal programs on the program code to gather as much information as possible on the various applications and subsequently visualizing the results.

In contrast to internal discovery, external discovery analyzes which network accessible sources provide information for a detailed picture of an application topology, e.g. a web server is not identified by looking at its source code (or code running on the virtual machine) but by issuing HTTP requests to query the server for information. This diploma thesis focuses on external application topology discovery in service-oriented applications that are defined in WS-BPEL and orchestrate multiple lower level Web services. The goal is to research and evaluate ways for external discovery of application topologies and the development of a prototypical, plugin-based framework that manages the topology information in a global model, to facilitate the migration of the application or parts of it to a cloud.

Table of Contents

Abstract	3
Table of Contents	4
Table of Figures	6
List of Tables	7
Table of Listings.....	8
List of Abbreviations.....	9
1 Introduction	10
1.1 Motivating Example	10
1.2 Problem Statement.....	11
1.3 Outline.....	12
2 Related Work and Fundamentals	13
2.1 Software Architecture Reconstruction (SAR)	13
2.2 Network-based Discovery.....	16
2.3 Searching Techniques.....	18
2.4 Service-Oriented Architecture (SOA).....	19
2.5 Cloud Computing.....	20
3 Concepts.....	22
3.1 Framework	22
3.2 Functional Requirements Analysis.....	23
3.3 Data Model.....	27
3.4 Scheduler	28
3.5 Plugins	34
4 Architecture and Design	44
4.2 Design Decisions.....	49
4.3 Resulting Architecture	54
5 Prototypical Implementation ATDFramework	57
5.1 Project Setup and Modeling.....	57
5.2 Framework Implementation	65
5.3 Plugins	79
6 Summary and Outlook	84

Appendix A Framework Manual.....	86
A-1 Framework Installation	86
A-2 Plugin Installation	87
A-3 Plugin Development	87
A-4 Initiate Discovery	91
References.....	92
Declaration.....	95

Table of Figures

Figure 1 – Motivating Example	11
Figure 2 – Software Architecture Reconstruction Life-time flow [3]	13
Figure 3 – Modeling URL as an object.....	26
Figure 4 – Modeling URL as an attribute	26
Figure 5 – Data model.....	27
Figure 6 – Data model as Diagram.....	28
Figure 7 – Priority List for Tomcat web server	31
Figure 8 – Refined scheduling algorithm	33
Figure 9 – Relationship between BPEL process and Web Service	35
Figure 10 – Tomcat Web Console	40
Figure 11 – Monitoring Apache Tomcat with JConsole	41
Figure 12 – ActiveMQ Web Console	43
Figure 13 – Use Case Overview and Roles	44
Figure 14 – Components of Eclipse RCP (Adapted from [31])	51
Figure 15 – GMF overview [36]	53
Figure 16 – Resulting architecture of the ATDFramework	54
Figure 17 – GMF Dashboard.....	57
Figure 18 – Domain model (atdframework.ecore).....	58
Figure 19 – Project structure after EMF code generation	59
Figure 20 – Graphical model definition (atdframework.gmfgraph).....	60
Figure 21 – Graphical tooling definition (atdframework.gmftool)	61
Figure 22 – Graphical mapping definition (atdframework.gmfmap).....	61
Figure 23 – Project structure of GMF editor.....	62
Figure 24 – Branding Logo of ATDFramework	64
Figure 25 – Update Manager of ATDFramework	64
Figure 26 – Installed Plugins view	71
Figure 27 – Priority Map view	73
Figure 28 – Properties view	77
Figure 29 – Type-specific properties left side	78
Figure 30 – Type-specific properties right side	78
Figure 31 – Screenshot of the final application	85
Figure 32 – Product export wizard.....	86
Figure 33 – Discovery of the Hello World Example.....	91

List of Tables

Table 1 – Relationships between objects	25
Table 2 – Use Case: Create New Project	45
Table 3 – Use Case: Load Project.....	45
Table 4 – Use Case: Save Project.....	46
Table 5 – Use Case: Edit Project in Framework	46
Table 6 – Use Case: Edit Project with Text Editor	47
Table 7 – Use Case: Start Discovery.....	48
Table 8 – Use Case: Develop new Plugin	48
Table 9 – Use Case: Install new Plugin	48
Table 10 – Use Case: Uninstall new Plugin	49
Table 11 – Adjustments in atdframework.gmfgen	62
Table 12 – IATDFPlugin interface methods	67
Table 13 – Settings for the installed plugins view	72
Table 14 – Properties of the <i>Others</i> menu item	72
Table 15 – Properties of the <i>Installed Plugins</i> menu item	73
Table 16 – Properties of the <i>Installed Plugins Show View</i> parameter	73
Table 17 – Settings for the <i>Priority Map</i> view.....	74
Table 18 – Run command	75
Table 19 – RunStep command.....	75
Table 20 – Properties of BPEL plugin.....	80
Table 21 – Properties of WSDL plugin.....	81
Table 22 – Properties of Web Service plugin.....	81
Table 23 – Properties of web server plugin	82
Table 24 – Properties of Tomcat JMX plugin	82
Table 25 – Properties of ActiveMQ JMX plugin	83

Table of Listings

Listing 1 – Scheduling algorithm in pseudo code	31
Listing 2 – Refined scheduling algorithm	34
Listing 3 – Import statement in a BPEL file	35
Listing 4 – SOAP over HTTP binding example	36
Listing 5 – SOAP over JMS binding example	38
Listing 6 – Request header.....	38
Listing 7 – Response header.....	39
Listing 8 – Response from Apache 1.3.23 [23]	39
Listing 9 – Response from IIS 5.0 [23].....	39
Listing 10 – Tomcat JMX connection string	41
Listing 11 – Calling Tomcat JMX Servlet	42
Listing 12 – Result of a JMX Servlet query	42
Listing 13 – Running a broker with <i>useJMX</i> property.....	43
Listing 14 – Connecting to URL.....	43
Listing 15 – <i>de.kreinjg.gmf.atdframework.manager.plugins.exsd</i>	66
Listing 16 – IATDFPlugin interface	66
Listing 17 – AbstractATDFPlugin.....	68
Listing 18 – AbstractATDFPlugin subclass example.....	69
Listing 19 – Example of ISafeRunnable.....	70
Listing 20 – Detecting plugins that implement the IATDFPlugin interface	71
Listing 21 – RunHandler	75
Listing 22 – Part of the scheduler	76
Listing 23 – Property tab extension	78
Listing 24 – Property section extension	79
Listing 25 – Extension provided by <i>BPELPlugin</i>	80
Listing 26 – XPath statement to retrieve locations of WSDL documents.....	80
Listing 27 – HelloWorldPlugin plugin.xml.....	87
Listing 28 – Generated HelloWorldPlugin.java class.....	88
Listing 29 – Modified HelloWorldPlugin.java class.....	89

List of Abbreviations

BPEL	Business Process Execution Language
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name System
IaaS	Infrastructure-as-a-Service
IIS	Internet Information Services
JMS	Java Message Service
JMX	Java Management Extensions
LDAP	Lightweight Directory Access Protocol
LRU	Least Recently Used
OSGi	Open Services Gateway initiative
PaaS	Platform-as-a-Service
SaaS	Software-as-a-Service
SAR	Software Architecture Reconstruction
SNMP	Simple Network Management Protocol
SOA	Service-Oriented Architecture
UDDI	Universal Description, Discovery and Integration
WSDL	Web Services Description Language
XML	Extensible Markup Language

1 Introduction

Nowadays, enterprises are facing rapidly changing market conditions which require flexible IT systems able to dynamically adjust to those changes. Staying competitive means reducing time to market, management costs, and costs for infrastructure. Inflexible IT systems, unable to keep up with those fast changes, will lead to the inability of quickly responding to competitor actions, customer demands, or economic trends, and thus a loss in sales.

All these difficulties have changed the way application development works nowadays. One of the architectural styles increasing the flexibility of application development and emerging over the last years is the service-oriented architecture (SOA) paradigm. It introduces loose coupling between software components (*services*) and fosters reuse of these services. A SOA is commonly realized through the Web service stack, a set of XML-based open standards like *WSDL* [1], *SOAP*, and *UDDI*. Web services can be orchestrated using the *Web Service Business Process Execution Language (WS-BPEL)* [2], introducing flexibility in application development by separating the business logic from the underlying services. These orchestrated services are then exposed again as web services, providing another level of granularity through the reuse of these services.

Another emerging paradigm is Cloud computing, allowing enterprises to focus on their core competences by moving parts of applications or an application as a whole from on-premise to a cloud environment. These environments can be in-house or provided by third party vendors, providing elasticity, scalability, and high availability of computational resources, and thus making the migration of a SOA application or parts of it to a cloud a possible next step on the way to a more flexible IT system.

This diploma thesis focuses on ways to support this migration step by researching and prototypically implementing discovery of application topologies of a SOA application, defined as a BPEL composition with multiple Web service calls. In particular, the focus of this work is on ways of external discovery of the participating components, such as web servers, messaging systems, databases and operating systems.

1.1 Motivating Example

The example given here illustrates a simple BPEL process that orchestrates different Web services, hosted on different machines. These Web services can be implemented using different languages, such as Java or C#, running on different Operating Systems like Linux or Windows and different Web servers, like Apache Tomcat or Microsoft Internet Information Services (IIS) and using various Database Servers like MySQL or MSSQL.

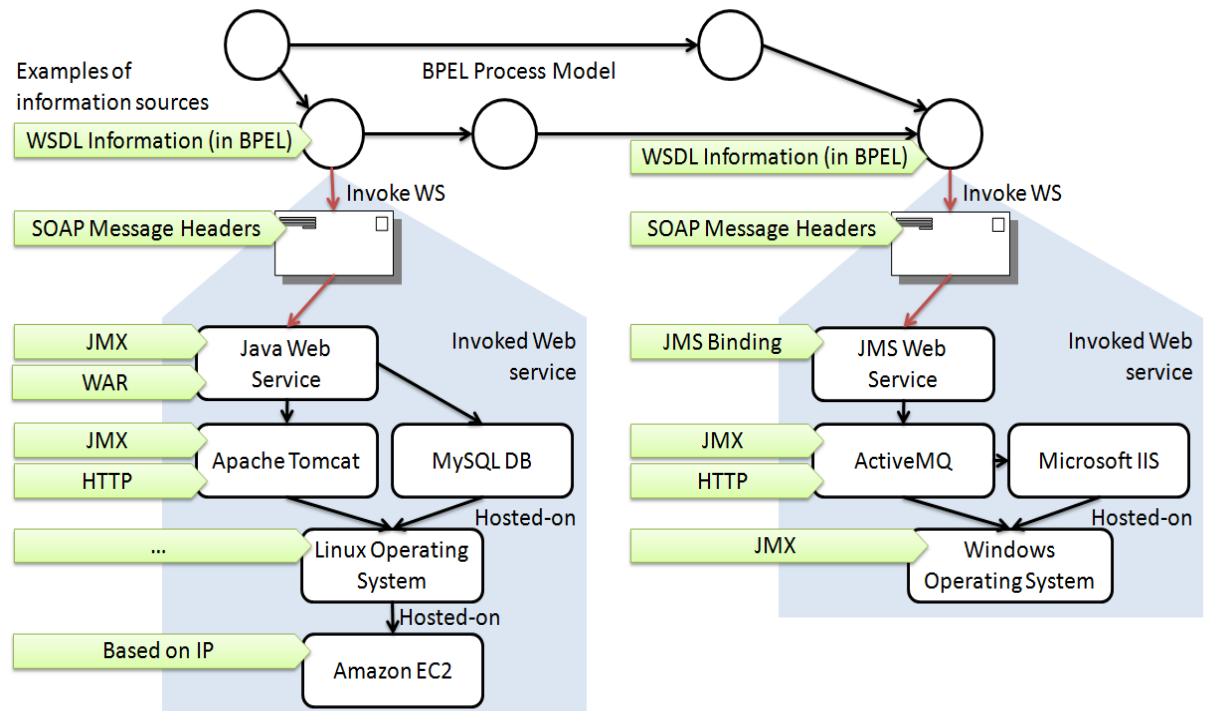


Figure 1 – Motivating Example

1.2 Problem Statement

The goal of this thesis is to research and evaluate ways for external discovery of application topologies, implement a set of prototypical discovery plugins and manage the topology information in a global model. The tasks constituting this thesis include the following:

- A state of the art regarding discovery of service information in research and products.
- Research which external, network accessible sources provide information helpful to get a detailed picture of the application topology of the Web services used in the composition.
- Implementation of a plugin-based framework which is able to manage the described discovery as well as store the gathered information in a structured way for further processing.
- A prototypical implementation of external discovery plugins including plugins for WSDL, Tomcat (Java Web service), Apache ODE (BPEL), Apache ActiveMQ (JMS), and Amazon EC2 (IaaS).

1.3 Outline

This document is divided into 6 chapters – an Introduction, a Related Work and Fundamentals, a Concepts, an Architecture and Design, and a Prototypical Implementation section. The thesis concludes with a Summary and Outlook section, summarizing the work done in this thesis and giving an outlook to future work. Furthermore the appendix contains a Framework Manual.

In detail, each chapter covers the following topics:

Chapter 1 - Introduction: The Introduction gives the reader a motivating example followed by the problem statement. It also contains the outline of this document.

Chapter 2 - Related Work and Fundamentals: Chapter 2 gives an overview of related work others contributed in this area and evaluates the research achievements over the past years. The focus is mainly directed to methods of software architecture reconstruction, network-based discovery of IT assets and discovery techniques used by search engines. It also covers the basic technologies to give the reader the necessary understanding of paradigms, architectures and technologies used in this thesis. These include cloud computing, service-oriented architecture, Web services and BPEL.

Chapter 3 - Concepts: Chapter 3 describes the concepts developed during the course of this writing. This includes concepts of the Framework and the developed plugins.

Chapter 4 - Architecture and Design: The Architecture and Design are described in chapter 4, giving a high level view on the developed framework and discovery plugins. It also describes the supported use cases of the resulting architecture.

Chapter 5 - Prototypical Implementation: Implementation aspects are covered in chapter 5. This provides a low level view on the framework and discovery plugins and describes the development process.

Chapter 6 - Summary and Outlook: The last chapter summarizes the work done in this thesis and gives an outlook to future work.

Appendix A - Framework Manual: This appendix provides instructions on how to install the framework and to start a discovery process. Additionally it describes how the framework can be extended by developing new plugins.

2 Related Work and Fundamentals

The following is a summary of related work, done by others over the last years to contribute to the task of migrating SOA applications to the cloud. This includes research in the area of software architecture recovery, which has received considerable attention recently. The focus is also on network-based discovery, like methods for asset and inventory management. Other related work also includes discovery techniques used by search engines, e.g. crawling and graph traversal. Another focus is on related technologies that some of the concepts of this diploma thesis are building on.

2.1 Software Architecture Reconstruction (SAR)

Ducassee and Pollet [3] provide a state of the art in software architecture reconstruction approaches, based on publicly available and trackable articles and documents, like PhDs and technical reports. According to [3], software architecture reconstruction is a reverse engineering approach aiming to reconstruct architectural views of a software application. They structure the field around *goals*, *processes*, *inputs*, *techniques* and *outputs* of SAR approaches (Figure 2). Excluded are works that extend traditional languages to mix architectural and other programming elements like *ArchJava* [4], because in these cases the architecture is not extracted from existing applications. The following chapters are a quick summary of what Ducasse and Pollet described in detail in their state of the art.

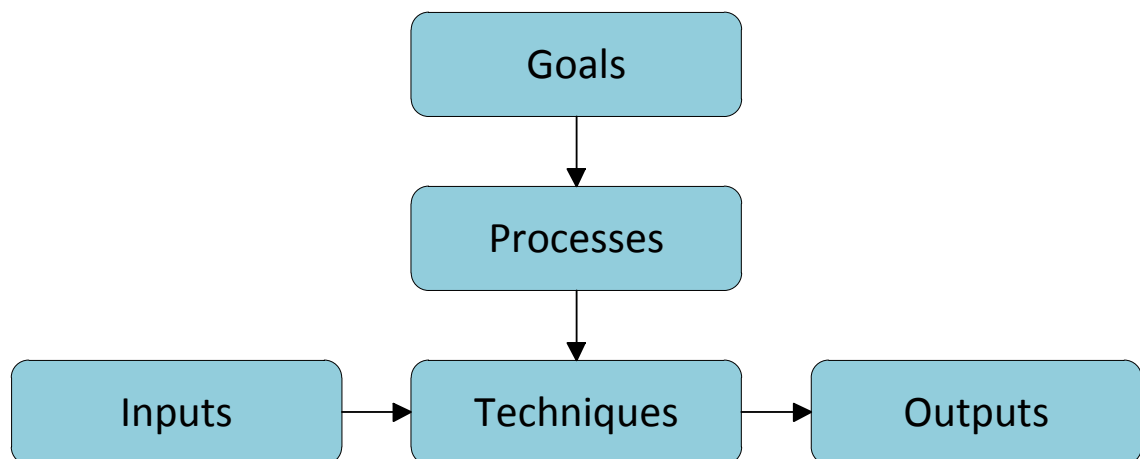


Figure 2 – Software Architecture Reconstruction Life-time flow [3]

2.1.1 Goals

Garlan [5] identifies six main goals of software architecture in software development: *Understanding*, *Reuse*, *Construction*, *Evolution*, *Analysis* and *Management*. These goals are refined by Ducasse and Pollet for architecture reconstruction.

- *Re-documentation and understanding*: Architectural views provide a view of the system on a high level of abstraction, making it easy for reverse engineers to understand the overall system.
- *Reuse investigation and product line migration*: Architectural views identify components, frameworks and patterns that can be reused. These commonalities can be shared in product lines.
- *Conformance*: As a concrete architecture often does not match the conceptual one, software architecture reconstruction can be used to check conformance of concrete and conceptual architecture.
- *Co-evolution*: SAR can be used to synchronize the deviation of architecture and implementation, as they usually evolve at different speeds.
- *Analysis*: Architectural views, provided by a SAR framework, can be used by analysis frameworks to support the decision-making process of stakeholders.
- *Evolution and maintenance*: Software architecture recovery may be used as a way to support application evolution and maintenance. This step often includes looking at an applications history, not only its current state.

2.1.2 Processes

The different SAR approaches can be categorized as *bottom-up*, *top-down* and *hybrid* processes.

Bottom-up

Bottom-up processes extract the architecture from source-code, continuously creating a higher level of abstraction until a sufficient level of understanding is reached. A process cycle consists of three steps: populating a repository with source code analyses, querying the repository for abstract system representations and presenting the results in an adequate form. The bottom-up approach is also known as software architecture *recovery*.

Top-down

Another approach is used by top-down processes, also known as software architecture *discovery*. Here, high-level information like architectural styles is used to reconstruct architecture by matching formulated conceptual hypotheses to the source code.

Hybrid

Hybrid processes use a combination of bottom-up and top-down process, creating high-level views from source code and matching these with views of high-level information. These approaches often use views from both concrete and conceptual architecture. Sartipi and Kontogiannis [6] describe a hybrid approach, presenting a graph matching model for software architecture recovery and a tool, implementing their approach, called Alborz. In a first step, Alborz parses the source code in a bottom-up phase and generates a graph as a high-level view. A reverse engineer then specifies

hypothesized views of the architecture in a top-down phase. The graph is then mapped with the specified views using graph matching.

2.1.3 Inputs

Most software architecture recovery tools use source code as their input, but dynamic information that is only available during execution, as well as historical data are also used by different tools. Ducasse and Pollet divide inputs into non-architectural and architectural inputs.

Non-architectural Inputs

The most often non-architectural input used is source code, as it is a reliable source of information. Some tools work directly on the source code, while others use metamodels to represent the code. Symbolic information, found in comments, method names or file names are sometimes also used. As static information is often insufficient, some approaches use dynamic information to analyze the behavior of a system. Another non-architectural input that is rather rarely used is historical information. While human expertise may not be as trustworthy as source code, for example, it is very helpful and usually needed to steer the iterative SAR process and to validate the results.

Architectural Inputs

Architectural inputs used by some tools are architectural styles and viewpoints. These styles include pipes and filters, layered system, data flow, blackboard and domain models. Though these abstractions are valuable and expressive, they are usually difficult to recognize.

2.1.4 Techniques

Techniques often match the data they operate on, e.g. queries on facts or queries on graphs. These are categorized by [3] into *quasi-manual*, *semi-automatic* and *quasi-automatic* techniques.

Quasi-manual

Here, the reverse engineer uses a tool only to assist in identifying architectural elements, e.g. using a visualization tool to manually reconstruct the software architecture.

Semi-automatic

With semi-automatic techniques, discovery of refinements is done automatically by the tool, while the reverse engineer instructs the tool how to do that. Techniques belonging to this group are *query based* techniques, e.g. relational queries that abstract data out of relational databases (by using SQL queries, for example), or *graph pattern matching* techniques like they are used in Alborz [6].

Quasi-automatic

There is no fully automated software architecture extraction technique. Reverse engineers always have to guide the SAR process, even at the most automated approaches. Techniques belonging to this group are *clustering algorithms* that identify groups of objects that share similarities, and *dominance analysis* that identifies related parts in an application.

2.1.5 Outputs

Outputs of a SAR process are clearly related to the goals of SAR. The usual output of a SAR tool is a visual representation of the software architecture, e.g. using boxes to present and group source code entities, or showing components and architectural elements and layers at different levels of abstraction. Some approaches also provide valuable additional information, such as conformance of architecture and implementation or conceptual and concrete architecture.

2.1.6 Relation to this work

The goals and techniques used in the described SAR approaches can be adopted to methods of external application topology discovery, but these SAR approaches usually work on a very low level, e.g. source code and they are usually executed on the same machine the application is running on. The approach described in this diploma thesis focuses more on the topology and interaction of different (*standalone*) applications, like web servers and databases, and how they can be discovered and accessed externally over the network. As the framework in this thesis is plugin-based, a possible extension to the framework could be a plugin that uses a SAR tool to complement information that cannot be provided by any other plugin.

2.2 Network-based Discovery

Tracking computing devices and assets to understand the operational infrastructure and its users has become an important aspect for enabling business-driven IT management, and plays a key role during the negotiation of outsourcing contracts and for planning mergers and acquisitions. Typical methods for asset and inventory management are periodic physical inventories, which require costly human visits or periodic self-assessment questionnaires to be filled out by individual end-users. Sometimes, servers and end-user devices are also equipped with software agents for the tracking of resources and the system, but this assumption cannot be made in every situation [7]. Therefore, a lot of recent work focuses on the discovery and tracking of networked IT assets and their relationships.

There are various sophisticated commercial products for managing enterprise networks, such as EMC Ionix¹, IBM Tivoli², Microsoft SCOM³ and HP NMC⁴, but they usually require too much manual effort, or they are restricted to a particular set of applications of the same vendor.

2.2.1 Techniques

Techniques for discovery of networked IT assets are classified into *online* methods (analyzing the actual state of the network) and *historic* log information processing (analyzing recorded network traces). Online discovery techniques are further classified into active network mapping, passive network mapping and host and service mapping.

Active Network Mapping

The network is explored exhaustively from a starting point by using a repetitive algorithm that walks the entire network up to an endpoint or until the entire IP address range has been exhausted [7]. The different techniques include SNMP walking of network topology, network-wide IP ping sweeps, DNS network domain name-space walking, DHCP lease information analysis and LDAP or Active Directory searches. Problems with this approach can occur when parts of the network are protected by firewalls.

Passive Network Mapping

Network mapping is done passively, i.e. without generating any kind of traffic that stimulates target machines to discover their presence. These techniques include network packet sniffing and subscription to network and syslogs. A packet capture application (network probe) must be placed where the traffic actually flows, i.e. packet capture is location dependent.

Host and Service Mapping

Once a host is discovered, the next step is usually to drill down on each host, e.g. identifying the operating system and the services a host provides. These techniques include TCP/IP stack analysis and OS detection as well as UDP/TCP port scans.

2.2.2 Research

Kind et al. [7] present a relationship discovery approach using passive network mapping based on *NetFlow* [8]. Their network profiling system *Aurora* detects traffic relationships by identifying flow correlations, i.e. flow pairs or flow chains based on their starting or ending times and a time-dependent correlation distribution. A flow pair could represent a database access or a directory access for authentication, for example.

¹ EMC Ionix. <http://www.emc.com/products/family/ionix-family.htm>

² IBM Tivoli. <http://www.ibm.com/software/tivoli/>

³ Microsoft SCOM. <http://www.microsoft.com/en-us/server-cloud/system-center/operations-manager.aspx>

⁴ HP NMC. <http://www8.hp.com/us/en/software/software-solution.html?compURI=tcm:245-936973>

Chen et al. [9] introduce *Orion*, a system that discovers dependencies using packet headers and timing information in network traffic based on *delay spike* analysis. It discovers service dependencies by observing the time correlation of messages between services and uses the delay distribution of service pairs to determine their dependency relationship. The number of spikes corresponds to the number of executed paths in services. The delay distribution is treated as a signal so that signal processing techniques like random noise reduction can be applied (random noise is spikes in the delay distribution due to host and network load variations). This is achieved by using Fast Fourier Transform (FFT) to decompose the signal across the frequency spectrum and applying low-pass filtering to mitigate the impact of random noise

Gantenbein et al. [10] propose an approach to complement basic network-based discovery with the combined log information from network and application servers, and then to compute an aggregate picture of assets and categorize their usage with data-mining techniques. Log files from selected network and application servers are first normalized into a denser representation (usage records). Related usage records from multiple network and application protocols are then aggregated into a server-independent perception of activities. Finally, usage and activity records are input to analytic processing and data mining.

2.2.3 Relation to this work

The work done in this diploma thesis uses a variation of the different described techniques, e.g. a lot of information can already be found in the WSDL files but a OS discovery needs more active discovery. This could be a TCP/IP stack analysis to discover the operating system, for example. Furthermore, already developed programs like port scanners could be bundled as plugins and hence contribute to the discovery process.

2.3 Searching Techniques

Different discovery techniques, often used in web search engines, also lay a background for the work in this thesis. This is more in regard to processes that are executed before a search engine can provide the search service itself to others, i.e. crawling and indexing [11]. Further important topics are ways of traversing graphs, e.g. depth-first search or breadth-first search and cycle detection in graphs, as well as the de-duplication of entities that are found multiple times.

2.3.1 Crawling

A web crawler usually creates a copy of all visited web pages for later processing by an indexer. The crawler starts with a list of URLs (*seed*). It then downloads the pages from these URLs and extracts hyperlinks found on these pages. The newly found links are added to the list of URLs so that the crawler will eventually download these pages as well. This step is repeated until there are no further pages to crawl, or a defined limit is exceeded (e.g. time or network bandwidth) [12]. *Periodic crawlers* build a brand new

collection which replaces the old one when it is necessary to refresh the collection. *Incremental crawlers* incrementally refresh the local collection by replacing less-important pages with more-important ones [13].

2.3.2 Graph Traversal

Considering web pages as nodes and hyperlinks as edges in a graph, crawling can be seen as the traversal of a graph. Traversal can be done either through *depth-first search* or through *breadth-first search*. A breadth-first search algorithm explores the graph layer by layer, the starting node being layer zero and all direct neighbors being layer one. This technique is useful when trying to find a path from one node to another with minimum edges. Opposite to this technique, a depth-first algorithm always continues to explore from the next node that it finds, and only goes back to previously explored nodes when it is running out of options. Because these algorithms remember already visited nodes, cycle detection becomes very easy, as repeatedly (on different paths) visited nodes will already be marked as visited.

2.3.3 Relation to this work

A crawler can be compared to a plugin in the later described framework for application topology discovery. For example, a plugin (crawler) searches a WSDL document (page) and discovers (finds a hyperlink to) a Java Web Service (page). Opposite to searching through web pages, the nodes (pages) in the framework are of different types, e.g. WSDL or Java Web Service, so that different crawlers (plugins) have to be developed. These crawlers can only operate on compatible nodes, e.g. a plugin for WSDL file searching and a plugin for Java Web Service searching.

2.4 Service-Oriented Architecture (SOA)

Service-Oriented Architecture [14] is an architectural style that supports the integration of applications as connected and reusable business applications or business services. A service represents a function provided at a network address that is available via different transports, formats, and quality of service, and which is always on, i.e. it does not have to be created or destroyed. The term loose coupling is often used in conjunction with SOA. Components in a loosely coupled system make little or no assumptions about other separate components.

2.4.1 Web Services, SOAP, and WSDL

An often misconception is the equality of SOA and Web Services. Web Services are a concrete implementation of a SOA while a SOA does also have other implementations. The goal of Web Services is to achieve interoperability between applications using Web standards [2]. Web Services usually take the HTTP protocol as transport because it provides the most interoperability. This is due to the fact that most firewalls are con-

figured to allow access via the HTTP protocol while other ports and protocols may be blocked.

SOAP [15] is one of the most used application protocols in Web Services that allows for the exchange of data between different systems. It is an XML format that wraps SOAP messages into an envelope. The envelope is split into a message header that contains Meta information, and a body that contains the payload. Data exchange does not have to be synchronous as it must be with remote procedure calls, for example.

Web Services are described using the *Web Services Description Language (WSDL)* [1] which is also an XML-based format. It is used for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint. The format is extensible to allow description of endpoints and messages regardless of what message formats and network protocols are used [1].

2.4.2 Business Process Execution Language (WS-BPEL)

The Business Process Execution Language (WS-BPEL) [2] is a language for specifying business process behavior based on Web Services. These business processes can be separated into two kinds. Executable business processes that model actual behavior of a participant in a business interaction and abstract business processes that are specified only partially and which are not intended for execution [2]. The latter can be seen as process views to hide internal details, for example.

BPEL describes workflows as orchestration of Web Services. It is a recursive aggregation model which means that tasks in BPEL processes are Web Services and the process itself is a Web Service again. It is based on XML and originated from the *Web Service Flow Language (WSFL)* by IBM [16] and *XML Business Process Language (XLANG)* by Microsoft [17]. In 2002 the two companies released the first version under the name *BPEL4WS* [18]. The second version was released in 2007 under the name WS-BPEL by the OASIS consortium that took over standardization. It supports primarily automated business processes but *BPEL4People* [19] is an extension that allows the integration of people.

2.5 Cloud Computing

Cloud computing [20] is one of the new models for data processing that emerged over the last years. It should cope with the fast growth in devices with internet connectivity and the growing presence of IT in business- and personal environments. Applications and data is not processed and stored on a local device anymore but on an external infrastructure. The basic principle is the outsourcing of software and hardware of the user, so that the user does not have to care, where the applications or information is actually

stored in the cloud. IT services are provided as services over the internet and accounted on basis of utilization of the service.

Depending on the sort of the service a distinction is drawn between *Infrastructure-as-a-Service* (*IaaS* – e.g. storage space over the internet), *Platform-as-a-Service* (*PaaS* – e.g. providing development tools over the internet), and *Software-as-a-Service* (*SaaS* – e.g. usage of an application over the internet). Further distinction is drawn between *private* clouds for a limited user group and *public* clouds for a variety of users. The often found solution is usually a combination of the two mixed with traditional IT environments called *hybrid* clouds.

The great advantage of cloud computing is the possibility to outsource tasks to external companies. This way, servers and applications do not have to be bought and managed by a business enterprise itself but can be rented from specialized companies. This reduces costs especially for the management of the infrastructure and provides flexibility. But concerns are the security and reliability aspects of cloud services, e.g. because of different laws in data privacy protection in different countries. Another problem is the missing interoperability between cloud service providers which prevents a user from changing the provider [21].

3 Concepts

This chapter presents the fundamental concepts developed in this thesis: The overall framework (3.1) and its plugin-based approach (3.5), data model (3.3), plugin scheduler (3.4), and the concepts on how to discover different artifacts like applications and servers with their respective properties (3.2).

3.1 Framework

The framework is an application that is responsible for the external discovery of the application topology of a composite application. Its job is to take a BPEL file as input, manage the discovery, and store the gathered information in a structured way for further processing. The name of the prototypical implementation of the framework will be *ATDFramework* – the abbreviation for *Application Topology Discovery Framework*.

As stated by the assignment of tasks, the developed framework should be a plugin-based framework. The idea of a plugin is to bundle code as an installable form so that it can be installed to the framework. It is the only way additional functionality can be added to the framework. At the same time, the framework stays independent from the developed plugins making the overall architecture of the application modular and extensible. Providing a plugin-based framework has many advantages for the framework developer as well as the plugin developer, but it also introduces difficulties for both sides.

Advantages of using plugins:

- Extensibility: The framework can be dynamically extended with additional features, allowing the application to grow with increasing user requirements.
- Independent Development: Additional features can be developed independently, allowing different development teams to implement different components, even at the same time.
- Clear development direction: The framework provides a well-defined interface for the plugin developer, giving him a direction for development of additional components.
- Simplicity: A plugin usually has one goal, allowing developers to focus on that goal.

Disadvantages of using plugins:

- **Restriction:** The design of the plugin interface is always a trade-off between defining ways for extension and on the other hand restrict the possibilities for extension. Designing extensibility needs a good requirement analysis to meet all use cases.
- **Evolution:** Managing versions and backward compatibility can be very hard, especially when the plugin interface evolves. Often plugin developers have to update their plugin with each new version.
- **Complexity:** While plugins may work when running alone, new problems may arise when plugins interact with each other, with bugs appearing only with certain combinations of plugins.
- **Testing:** Testing a plugin may only be possible by running the plugin (inside the framework), slowing down the development process.

Of course, depending on the implementation of the framework some of the disadvantages may not be relevant, e.g., some frameworks may provide a possibility to test a plugin without running the framework.

The idea for the framework developed during the course of this diploma thesis is to define a global data model and to provide the necessary management functionality to the plugins that is necessary to operate on the data. The data model will contain the topology information discovered throughout the discovery process. Individual plugins will operate on the data and add information during the discovery process. The order in which plugins are executed must be determined by the framework, e.g. through a scheduling component.

3.2 Functional Requirements Analysis

Before defining the data model, the functional requirements of the framework have to be analyzed. Most of these requirements can be obtained by looking at how a discovery proceeds. This will help finding a suitable representation for the topology.

The discovery process takes a BPEL process as its input and outputs the application topology. During the discovery process the discovered topology will go through different iterations and eventually reach a final state. A plugin should take the currently discovered topology (or part of it) as its input and try to discover new information based on the provided information (e.g. a plugin gets a URL to a web server as input and outputs the type and version of the server).

The framework has the responsibility to orchestrate the execution of the various plugins. On the basis of the currently discovered topology and the currently installed plugins, the framework should determine the plugins that are allowed to operate on the model. One question to answer is whether a plugin can modify the whole topology or just parts of it. Either way, a plugin should concentrate only on specific areas of the

topology, e.g. concentrating on the discovery of web servers or the detection of operating systems.

3.2.1 Type System

This requires a mechanism that defines how plugins recognize parts of the topology as being compatible, i.e. the framework knows which areas can be processed by which plugin and the plugins know what to do with the provided information in a specific area of the topology. A possible solution would be a type system that allows typing of plugins and topology areas. This way the framework can determine which plugin should operate on which part of the topology. In this case it would also be reasonable to provide only the relevant parts as an input to the plugin so a plugin modifies only the area that it is assigned to. Depending on how the topology itself is represented, this approach can even be refined from areas to a single point of the topology (e.g. a single node).

The type system should also provide sub types, e.g. a Tomcat web server and an IIS are both web servers. A plugin may discover the existence of a web server without knowing whether it is a Tomcat, an IIS, or any other web server. Still it must have the possibility to add the information to the model. In a type system that allows for sub typing the web server type can be modeled as the parent of a Tomcat type and an IIS type. Then a plugin that discovers the existence of the web server can add the server to the model, while other plugins later specify the exact type of web server.

The type system should be built up implicitly by the plugins. This means that the plugins provide the types they support and these types are installed to the framework when the plugins are installed. The framework builds up the type system with all type dependencies when it loads the plugins. This type system is then used to determine which plugin can operate on which part of the topology, e.g. when scheduling the execution order. The definition of a type should also be kept simple, e.g. through the representation of a type by a namespace. A plugin developer then just specifies the types it supports and the types it creates (e.g. when discovering new topologies) through a simple definition of the *QNames* of a namespace. The definition of a type should also support the specification of a parent type, e.g. when specifying a new type called *tomcat*, the plugin developer should have the possibility to specify this type as a sub type of a web server type. The responsibility to manage these dependencies in a type system lies with the framework.

3.2.2 Objects, Relationships, and Properties

A simple representation of the topology could be one that models only objects and their relations to each other. For example, a BPEL process may contain references to WSDL files, a Web Service may be hosted on a web server, and a web server is hosted on an operating system. The operating system again may run on a specific hardware. Then, BPEL files, WSDL files, web server, operating systems, and hardware

nodes may be modeled as an object while their relations may be modeled as directed connections between the objects.

Relationship	From Object	To Object
Contains	BPEL file	WSDL file
Contains	WSDL file	Port
Hosted on	Web Service	Web Server
Provided By	JMS Web Service	Messaging System
Hosted on	Web Server	Operating System
Running on	Operating System	Specific Hardware

Table 1 – Relationships between objects

But modeling everything as an object or a relationship may be insufficient in some cases. Assuming the discovery process starts with a BPEL process, a user would first have to provide some information on what BPEL process to use and where to find it. This requires that the framework provides a mechanism for the user to manually add information to the model. It is also worth noticing that in this case the user provides two sorts of information.

1. What type of object should be added to the topology, e.g., a BPEL object
2. Where to find the object, e.g., the location to the BPEL file

The question is, whether the location to the BPEL file (e.g., an URL) is modeled as an object itself or a property of an object. If the location is modeled as an object, the BPEL object would have a relationship with the location object. Then the relationship could be modeled as a connection between BPEL object and location object. While this approach seems to be adequate for different sets of objects - e.g. the relationship between BPEL file and WSDL files or web server and operating system - some information like the URL to a BPEL file, the URL to a web server, or the version number of a web server should be added to an object as a property or an attribute.

This illustrates better the affiliation of the attributes to the object. The URL for the location of a BPEL file is something that is directly dedicated to the BPEL file, while the referenced WSDL files in a BPEL file are objects themselves which can even be referenced from different BPEL files. Furthermore, this allows for the specification of specific properties that must be available in every instance of an object, e.g. a BPEL file must have a location but it does not need to contain any WSDL references.

Another benefit of properties is the possibility to identify objects according to their properties, either by generating a special identifier or using an existing property that uniquely identifies an object. For example, a URL to a BPEL file uniquely identifies the BPEL file because no two files can be at the same location. This can be used by the

framework if a plugin request the creation of a new object. The framework could then check, whether there is already an object available that has the specified unique properties matching the one that should be created. Figure 3 and Figure 4 show a URL modeled as an object and as an attribute.

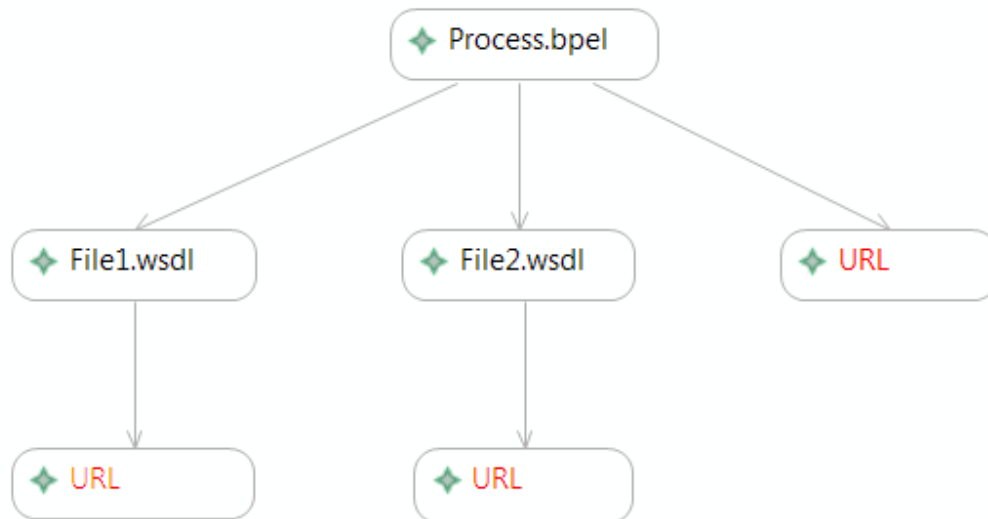


Figure 3 – Modeling URL as an object

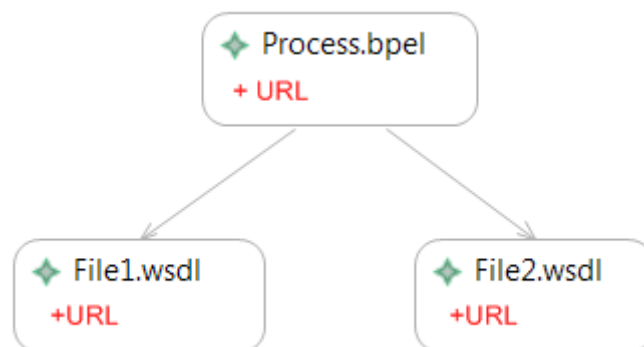


Figure 4 – Modeling URL as an attribute

If the type system supports sub types then it would also be reasonable to type properties, e.g. a web server as the parent type of Tomcat could contain the URL to the web server, while a Tomcat as the sub type can contain Tomcat specific properties, e.g. installed modules.

3.2.3 Summary

A first summary of the functions and concepts that must be provided by the framework are a simple representation of the model consisting only of objects, relationships and properties. Plugins need to know what they can do with the discovered topology which requires the adding of a type structure to the topology. The order of execution of different plugins is handled by the framework, e.g. through a scheduler. Furthermore, the user should have the possibility to add information to the model manually.

3.3 Data Model

The concepts developed in the previous chapter lead to the following data model illustrated in Figure 5 and Figure 6.

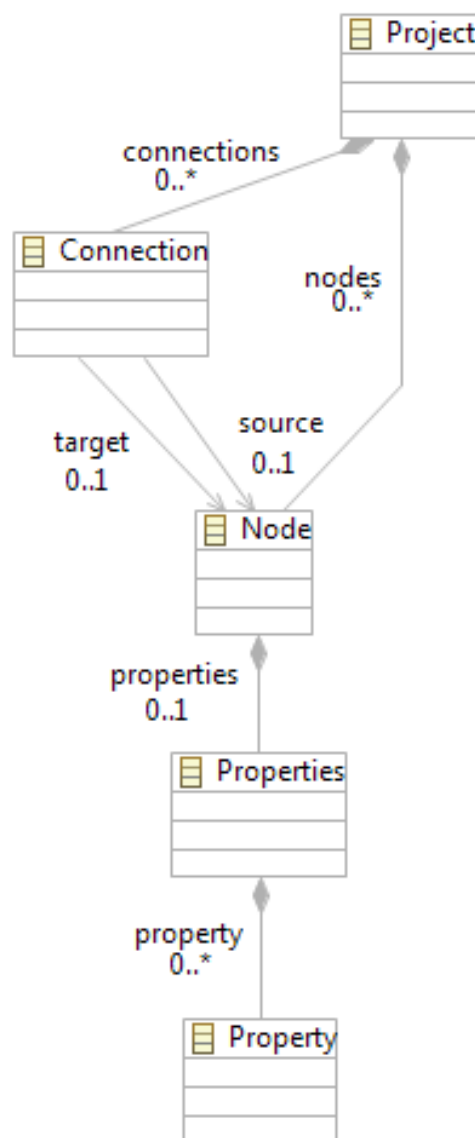


Figure 5 – Data model

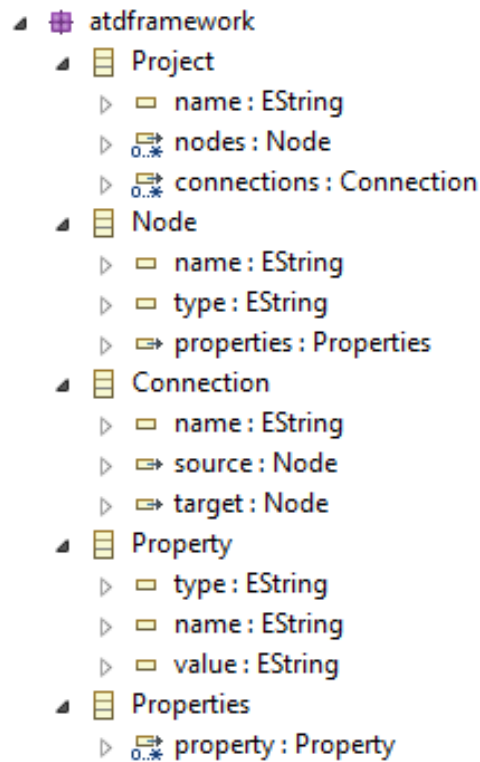


Figure 6 – Data model as Diagram

The topology information and referenced artifacts should be stored in a global data model. This data model contains all the discovered information and must be stored persistently for further processing, e.g., software that use this information to steer the cloud migration process. The top level object is a *project* that contains all the information that is found during a discovery, namely *nodes* and *connections*. A *name* can also be assigned to the project.

Each object is represented as a *node* that has a *name*, a *type*, and a set of *properties*. The type attribute is used for the typing of nodes so that plugins can check whether they support the node type (i.e. they can operate on the node). The properties element contains all properties of the node. Properties are typed themselves so they can be grouped and assigned to specific types. This allows different plugins to add properties to the node that belong to a certain group. Plugins may even read or write only properties of a certain type.

The project also contains a set of connections. Each connection has a name attribute and a reference to a *source* and a *target* node.

3.4 Scheduler

The scheduler is one of the most important and at the same time one of the most complex components of the framework, because it has to decide which plugin is allowed to execute at which time. The execution strategy can influence the order of information discovery and route the discovery process into different directions. For example, a

plugin may modify some information that another plugin is depending on. Depending on the execution order, the second plugin may not be executed at all because the information has now become invalid for the plugin, i.e. they are not compatible anymore.

3.4.1 Iterations

Based on the data model described in 3.3 the scheduler will have to conform to some guidelines. The discovery process will most likely start at a single node and produce an amount of nodes and connections. The execution process can be partitioned into iterations. Each iteration step starts with a set of nodes forming a topology and ends with a set of nodes representing a more complete topology. The first iteration will start with a single node. The last iteration should end with the complete discovered topology. This has the advantage that (1) the discovery process can be paused at a specifically defined point of the execution, allowing the user to view the current state of the discovery as well as possibly saving a snapshot or a copy of the process to the file system, and (2) a user may intervene between iteration steps, e.g. by modifying or adding information to the model, or by specifying a certain plugin that should be executed next, hence providing the possibility to route the discovery process into individual directions.

Of course this requires several functions to be provided by the framework, e.g. mechanisms for running all iterations at once or just a single iteration. In a graphical framework this could be realized through different buttons, one for running all iterations which means running till the topology does not change any more, and one for running just a single iteration step. Furthermore, the framework would have to provide the functionality to manually add or modify information, as well as the functionality for saving and loading of discovery processes to and from the file system.

The ability to save the topology at different points in time opens a new set of possibilities, e.g. a process may be stored and reloaded at another time, allowing the discovery process to start not at a single node, but at an already discovered set of nodes. Or the process may be shared with others allowing it to be executed on different frameworks that may contain a different set of plugins that are only available on specific systems, e.g. a plugin may require a license or a specific operating system to work properly.

3.4.2 Processing Order

As the topology can be seen as a directed acyclic graph, one question is which processing order should be applied and whether the processing order affects the discovered topology. For example, the processing order could be a depth-first-searching or a breadth-first-searching algorithm.

It is already clear that the order of execution of different plugins on a single node affects the discovered topology because a plugin may modify information that is needed by other plugins. Subsequent plugins may not be executed any more if the information has become invalid from a plugins perspective, i.e. the plugin does not know what to do with the information though it would have been able to be executed if the information

was not changed. For example, a plugin discovers the version number of a web server and stores the version number as a property of the web server node. A subsequent plugin may not be able to work anymore with that node because some functionality may only be available with certain version numbers of the web server, e.g. *Java Management Extensions (JMX)* [22] support may be recently added to newer version of the web server.

It is worth noticing that the information here represents a property of an object and not objects themselves, hence a plugin only becomes incompatible with a node if the properties of the node or the node itself changes and not the environment of the node, i.e. the relationships of the node with other nodes. To keep the scheduler as well as the type system – and hence the development of plugins – as simple as possible, it is reasonable to make the compatibility between node and plugin only depending on the availability of certain properties and not the relationships a node has with other nodes. This mainly affects the scheduling of plugins and the mechanism that checks for compatibility. Though using this approach may in some cases produce an overhead it is in no way a restriction of the possibilities a plugin developer has. A plugin may still depend on the environment of the node, but the responsibility to check for the availability of certain connected nodes lies in the hands of the plugin developer. The overhead occurs when plugins are scheduled for execution though they will not be able to execute because of a missing connected node. These so called *false positives* produce only little overhead when plugins check at the beginning if all required information for their execution is available and return if something is missing.

This approach even makes a specific processing order obsolete, because a processing order only affects the environment of the node. The node itself and its properties are independent from the processing order. They only depend on the execution order of multiple plugins on the same node. If the scheduler would also have to check the environment for the availability or existence of specific partner nodes, then the searching technique could influence what is actually discovered. In this case the order of the execution of plugins would furthermore depend on the environment of a node and hence produce a different execution order. Because the scheduling of plugins in the implemented framework does not consider relationships between objects, no specific processing order is favorable to the other. The scheduler can rely on the preferences of the selected system for the management of the nodes to select the order of execution. For example, if the managing system stores the nodes in an unordered list, the algorithm does not care about the order of nodes. It simply starts with the first node of the list.

3.4.3 Basic Scheduling Algorithm

Listing 1 illustrates the basic scheduling algorithm in pseudo code which is refined later in this section. In each iteration step, the scheduler would get the current set of nodes, calculate a scheduling list that contains a task for each node with the belonging plugin and finally execute each task on the scheduling list.

```

while last iteration discovered new information
    get all nodes;
    for each node of the current topology
        calculate scheduling list;
        execute scheduling list;
    end for
end while

```

Listing 1 – Scheduling algorithm in pseudo code

The difficult part is the calculation of the scheduling list which includes for each node the calculation of the plugin that is allowed to execute next on the node. This is trivial if there is only one compatible plugin for each node type. In this case, the plugin is selected for execution in the next iteration step. The difficult part is when several plugins can operate on a node. This requires the implementation of a strategy that defines priorities for plugins, either statically or dynamically, e.g. through the use of a history list and a strategy like *Least Recently Used (LRU)*.

3.4.4 Priority List and Execution History

The scheduler should contain a priority list for each node type that contains the order in which plugins are executed when there are multiple plugins for the same node type. Of course, this list would have to be initialized first by the framework, e.g. generating the list when the plugins are loaded initially. This list should also be editable by the user so that a user can define which plugin should execute first. The priorities must be managed centrally by the framework not the plugin developer. Otherwise, each plugin developer could set its priority to the highest level. Managing priorities centrally leaves the control in the hand of the user.

An example priority list for a Tomcat web server type is illustrated in Figure 7. In this example there are four plugins that can execute on a Tomcat type, a general web server plugin and three Tomcat specific plugins that use different methods to discover information (see Section 3.5.3).

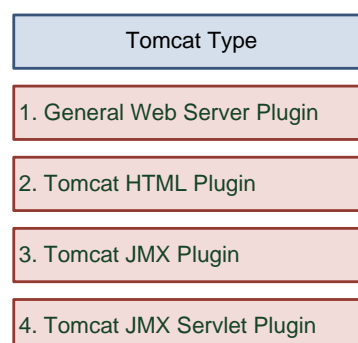


Figure 7 – Priority List for Tomcat web server

Since almost all common scheduling strategies base their decisions on what happened in the past, it seems to be reasonable to store an execution history for each node in the model (e.g. in the properties section of the node). This way, this information is portable, for example, when moving the data to another system. The scheduler uses the saved history for each node to determine the next plugin that should be executed.

3.4.5 Change of the Environment

A plugin will execute only once on a node, except another plugin changes some information after the plugin has worked on the node the last time. This is because another plugin may have added some information that the plugin can use now to discover additional information. That means that the plugin must be scheduled again if another plugin made changes, just to give it the chance to discover additional information.

Furthermore, one must consider that the environment itself has changed after a plugin has executed the last time, e.g. a plugin that wants to discover information about a web server based on a provided URL. If the server is down while the plugin is executed, the plugin will not be able to discover any information. The framework must provide a mechanism to schedule plugins after they already executed once without success. There may be cases where a plugin may be informed by an application that it is up again after being down so that a plugin may start executing again, but this will rather be the exception. A plugin could also repeatedly check for availability of the server till it is up again, but this could produce additional traffic on the network. Though it may be uncommon that an application or server is not available, still the framework must be able to handle those situations.

3.4.6 User defined Scheduling

The simplest way is to let the user decide, when to check again. For example, the user could request a complete re-check of the whole topology, e.g. by clearing the history of scheduled plugins on a node which could then force the scheduler to re-schedule all plugins again. Another possibility is to allow the user to exactly specify the plugin that should execute the next time on a node. This could be achieved by adding a property like *nextAction* to the node where the user can enter a specific plugin for execution in the next iteration step. The scheduler can check for the availability of this property and execute the specified plugin (if it actually exists). Else the scheduler continues with the usual scheduling of plugins.

The scheduler must also be notified if the user changes properties manually, e.g. the user can provide a username and a password to access an application via JMX which then enables a JMX-based plugin to execute. To inform the scheduler of the changes, the framework could listen for user actions and add an entry to the execution history list of the modified node. Based on this entry, the scheduler could re-schedule all plugins for that node that know what to do with that information. If the framework does not support listening for user actions, the user could also manually set a property that indicates

that the last action was performed manually by the user, e.g. through a property like *lastActionWasUserAction*.

3.4.7 Refined Scheduling Algorithm

The basic scheduling algorithm described in Chapter 3.4.3 is refined based on the concepts of the previous chapters to the algorithm described in this chapter and illustrated in Figure 8.

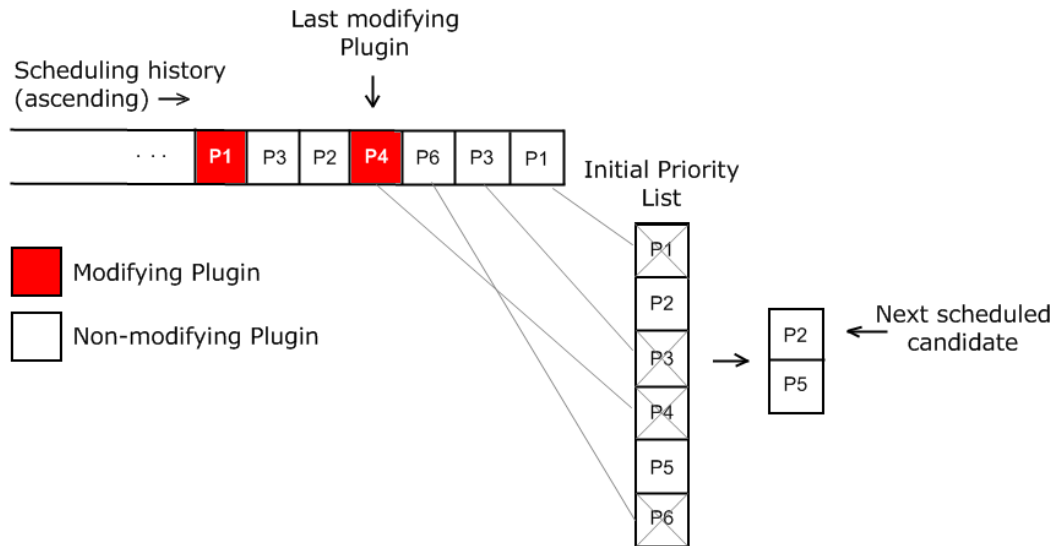


Figure 8 – Refined scheduling algorithm

When no user interaction changes the order of the scheduling of the plugins, the scheduler will work according to the following principle:

- When a plugin is executed on a node, it is added to the end of the scheduling history with a timestamp of the execution date. If the plugin modified the node, it is marked as a modifying plugin. Else, it is added as a non-modifying plugin.
- If no priority list exists for a node, an initial priority list is calculated from the available plugins.
- The scheduler takes the (initial) priority list and calculates a filtered priority list that contains the next scheduled plugin as the first element.
- The filtered priority list is calculated by taking the initial priority list and removing all plugins in descending timestamp order that are also contained in the scheduling history until a plugin is reached that modified the node.

In the example above P1, P3, P6, and then P4 are removed, because P4 was the last modifying plugin. As P4 was the last modifying plugin, it does not need to execute again, because nothing has changed since the last execution. Plugins P1, P3, and P6 also do not need to execute, because they executed and could not discover any information with the provided information. After removal of the plugins, P2 is the top element of the filtered priority list and will be scheduled for execution. The algorithm is illustrated by pseudo code in Listing 2.

```
For each node
  If node has no priority list
    Generate initial priority list;
  Fi
  Copy priority list to filtered priority list;
  Get last executing plugin from scheduling history;
  While plugin did not modify node
    Remove plugin from priority list;
    Move one step back in history list;
  End while
  If plugin modified node
    Remove plugin from priority list;
  Fi
  Use first plugin in filtered list;
End for
```

Listing 2 – Refined scheduling algorithm

The user may intervene between iteration steps by selecting a specific plugin for next execution. In this case, the plugin is selected for next execution. Afterwards the scheduler continues with normal execution. Also, all plugins may have executed already with no changes but the user wants to re-schedule all. This can either be done by clearing the history and hence leading to a re-scheduling of all plugins, or by adding a dummy plugin (e.g. a *ClearHistory* or *IgnoreHistory* plugin) to the history that simulates a modification and forces all other plugins to be re-scheduled.

3.5 Plugins

The discovery work is driven by node type-specific plugins which extract the respective information and add them to the data model. The framework should provide the mechanism of plugin installation and un-installation and contain prototypical discovery plugins for BPEL processes, WSDL files, Tomcat application servers, and Active MQ. There are different ways to discover the existence of such servers and applications, each of them described in the following chapters.

3.5.1 BPEL

As the discovery process will be built up from the BPEL process (Section 2.4.2) to the overall topology of the application, it must be determined which information of the

BPEL process can be used to discover the topology. A BPEL process is an orchestration of Web Services, so a starting point could be the determination of the participating Web Services. The information that is of interest is not the orchestration itself with its control flow and data flow, but only the participating Web Services. These Web Services can be hosted on different machines and different locations and hence already represent a part of the local distribution of the topology.

While the control flow and data flow can be of interest for a deeper analysis of the application – like it is done in software architecture reconstruction – it is not of high importance when the goal of the analysis is the migration of the application to the cloud. Because the process model will most likely be obtained in its current state, it is enough to represent only the relationship between the BPEL process and the invoked Web Services (WSDL file) as simple directed connections, without any additional control or data flow (see Figure 9).

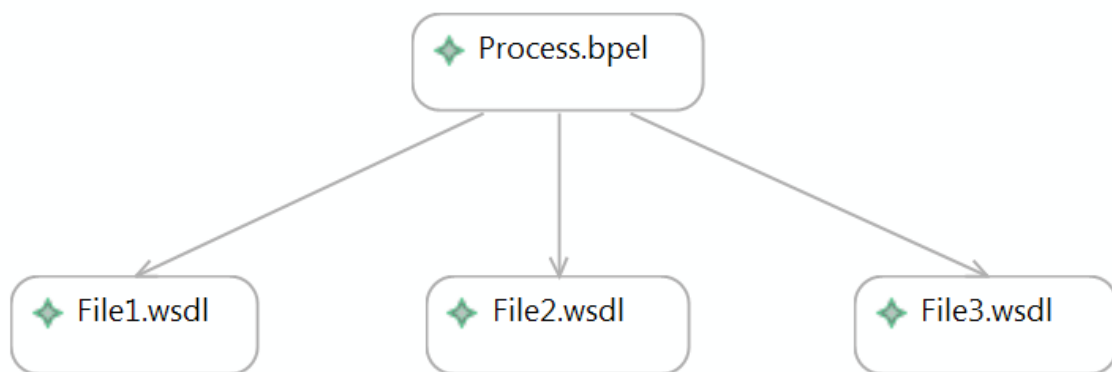


Figure 9 – Relationship between BPEL process and Web Service

A BPEL process uses *partnerLinks* or *partnerLinkTypes* to define the external services it interacts with. These partnerLinks are used by *invoke*, *receive* and *reply* activities to actually invoke a service or to receive a reply. A partnerLinkType amongst other things contains the port types of the participating services that are specified in the respective WSDL files. The BPEL process must contain an import statement for each WSDL file with the attribute *importType* having a value of <http://schemas.xmlsoap.org/wsdl/>.

```
<bpel:import location="file1.wsdl"
    namespace="http://www.example.com/somenamespace"
    importType="http://schemas.xmlsoap.org/wsdl/" />
```

Listing 3 – Import statement in a BPEL file

In a prototypical implementation it is sufficient to check only for these import statements at the beginning of a BPEL processes, whether they are actually used or not.

It is also worth noticing that the BPEL process – or rather the BPEL file – must be made available for parsing by the plugin, e.g. providing the file directly or via a URL to a download link.

3.5.2 WSDL

Some of the BPEL concepts can be applied to the concepts for WSDL (Section 2.4.1) files, namely that the WSDL file must be made available for parsing. A WSDL file describes the interface of a web service. The information that is important for a migration is where the web service is hosted on e.g., is the service hosted on a Tomcat web server.

A WSDL file consists of many constructs like *messages*, *operations*, *ports*, *portTypes*, *bindings* and *services*. While messages, operations and portTypes describe the functions that a web service provides, they do not give any information about the implementation. For example, the actual implementation can be an *Enterprise Java Bean (EJB)* developed in Java and hosted on a Tomcat or a .NET application hosted on an IIS. A first indicator of the implementation can be the WSDL binding and the port.

The most common binding is SOAP over HTTP as it provides the most interoperability by using the HTTP protocol as transport. If the binding is SOAP over HTTP, the port tag will contain a *soap:address* tag that contains the location (i.e. the URL) where the endpoint is found. Other plugins can take this URL and check which web server is behind that URL.

```
<wsdl:binding name="GlobalWSOap" type="tns:GlobalWeatherSoap">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
    style="document"/>
  <wsdl:operation name="GetWather">
    <soap:operation soapAction="http://webserviceX.NET/GetWeather"
      style="document"/>
    <wsdl:input>
      <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>

<wsdl:service name="GlobalWeather">
  <wsdl:port name="GlobalWSOap"
    binding="tns:GlobalWeatherSoap">
    <soap:address
      location="http://webservices.net/globalweather.asmx"/>
  </wsdl:port>
</wsdl:service>
```

Listing 4 – SOAP over HTTP binding example

If guaranteed delivery is more important than interoperability, another binding that can be used is SOAP over JMS that allows for specification of the location of a *Message Oriented Middleware (MOM)* via the *Java Naming and Directory Interface (JNDI)*. The

existence of the SOAP over JMS binding already gives a hint that a MOM must be involved. This information can be used by other plugins, e.g. a plugin can check whether Apache ActiveMQ is running on the provided location.

```
<wsdl:definitions name="JMSGreeterService"
  <wsdl:binding name="JMSGreeterPortBinding" type="tns:JMSGreeterPortType">
    <soap:binding style="document"
      transport="http://www.w3.org/2010/soapjms/" />
    <soapjms:jndiContextParameter name="name" value="value" />
    <soapjms:jndiConnectionFactoryName>
      ConnectionFactory
    </soapjms:jndiConnectionFactoryName>
    <soapjms:jndiInitialContextFactory>
      org.apache.activemq.jndi.ActiveMQInitialContextFactory
    </soapjms:jndiInitialContextFactory>
    <soapjms:jndiURL>tcp://localhost:61616</soapjms:jndiURL>
    <soapjms:deliveryMode>PERSISTENT</soapjms:deliveryMode>
    <wsdl:operation name="greetMe">
      <soap:operation soapAction="test" style="document" />
      <wsdl:input name="greetMeRequest">
        <soap:body use="literal" />
      </wsdl:input>
      <wsdl:output name="greetMeResponse">
        <soap:body use="literal" />
      </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="JMSGreeterService">
    <soapjms:jndiConnectionFactoryName>
      ConnectionFactory
    </soapjms:jndiConnectionFactoryName>
    <soapjms:jndiInitialContextFactory>
      org.apache.activemq.jndi.ActiveMQInitialContextFactory
    </soapjms:jndiInitialContextFactory>
    <wsdl:port binding="tns:JMSGreeterPortBinding" name="GreeterPort">
      <soap:address location="jms:jndi:dynQ/test.cxf.jmstransport.queue" />
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Listing 5 – SOAP over JMS binding example

The information in *jndiConectionFactoryName*, *jndiInitialContextFactory*, *jndiURL* and *soap:address* can be used to lookup the message broker via JNDI. Once the broker is looked up it can be queried for information, e.g. via JMX if JMX is supported.

3.5.3 Web Server / Application Server

There are different methods to identify a web server. The most obvious and easy one is to view the corresponding *server* HTTP-header that is usually sent with the HTTP response of a server, e.g. using a tool like Web-Sniffer⁵ to view request- and response headers. This header usually contains information about the server name and version number, sometimes even information about the operating system and server extensions, e.g., Apache modules. But sometimes there is no information available at all, or worse, the provided information is forged, e.g. because revealing this information might allow the server to become vulnerable to attacks. Most web servers can be configured to hide or forge this information. Listing 6 and Listing 7 show a request and a response header. The request is sent from a Firefox⁶ web browser to the host at 192.168.29.130:8080. In this example, the HTTP response that is sent from the server contains the *server* element with a value of Apache-Coyote/1.1 which indicates that there is an Apache server running on that host.

```
GET / HTTP/1.1
Host: 192.168.29.130:8080
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:11.0) Gecko/20100101
Firefox/11.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: de-de,de;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

Listing 6 – Request header

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: text/html; charset=ISO-8859-1
Transfer-Encoding: chunked
Date: Tue, 10 Apr 2012 21:00:27 GMT
```

⁵ <http://web-sniffer.net>

⁶ <http://www.mozilla.org/>

Listing 7 – Response header

Another popular but difficult way is to use fingerprinting. This technique is used, when web servers are obfuscated by changing the server banner strings. Fingerprinting uses the fact that almost all HTTP servers differ in the way they implement the HTTP protocol [23], like human fingerprints do. This is especially the case when those servers are confronted with malformed requests, see Listing 8 and Listing 9.

```
$ nc apache.example.com 80
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Date: Sun, 15 Jun 2003 17:10:49 GMT
Server: Apache/1.3.23
Last-Modified: Thu, 27 Feb 2003 03:48:19 GMT
ETag: "32417-c4-3e5d8a83"
Accept-Ranges: bytes
Content-Length: 196
Connection: close
Content-Type: text/html
```

Listing 8 – Response from Apache 1.3.23 [23]

```
$ nc iis.example.com 80
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Content-Location: http://iis.example.com/Default.htm
Date: Fri, 01 Jan 1999 20:13:52 GMT
Content-Type: text/html
Accept-Ranges: bytes
Last-Modified: Fri, 01 Jan 1999 20:13:52 GMT
ETag: W/"e0d362a4c335be1:ae1"
Content-Length: 133
```

Listing 9 – Response from IIS 5.0 [23]

This response represents a unique fingerprint or signature. The fingerprints are then usually compared to fingerprints in a database. In this example, it is easy to see that the *server* element and the *date* element are swapped. Some information is available only in one response, e.g. the *connection* element in the Apache response.

Apache Tomcat

To identify an *Apache Tomcat* Server, there are a number of things to consider when using some of these techniques. Though sometimes Tomcat is declared as being a web server, it is actually a servlet container implementing the *Java Servlet* [24] and *JavaServer Pages (JSP)* [25] specifications. Additionally, it contains the *Apache Coyote* connector that supports the HTTP/1.1 protocol, which enables Tomcat to function as a stand-alone web server [26]. When used in stand-alone mode, Tomcat will identify itself as “apache coyote/1.1” in the server header of an HTTP response.

Tomcat should not be confused with the *Apache HTTP Server*⁷, a popular web server in the internet. There are a lot of scenarios where Tomcat is not used as a stand-alone web server but behind an Apache HTTP Server and connected through the Tomcat redirector module (mod_jk), e.g. when the web server has to provide additional functions like supporting PHP or CGI scripts. Other scenarios use an Apache HTTP Server for load balancing of multiple Tomcat servers, or using a Tomcat behind a firewall that is only accessible through the HTTP Server. In these cases, the HTTP Server will identify itself as “Apache [version number]”. If the HTTP response contains mod_jk in the Apache modules, then this could be a hint that there is also a Tomcat running somewhere. Of course, the Tomcat instance (or instances) could be hosted on a complete different machine.

For Tomcat, there are additional ways to discover information about it, e.g. Tomcat provides a management console that is (remotely) accessible via a web browser at <http://localhost/manager/status> (Figure 10).

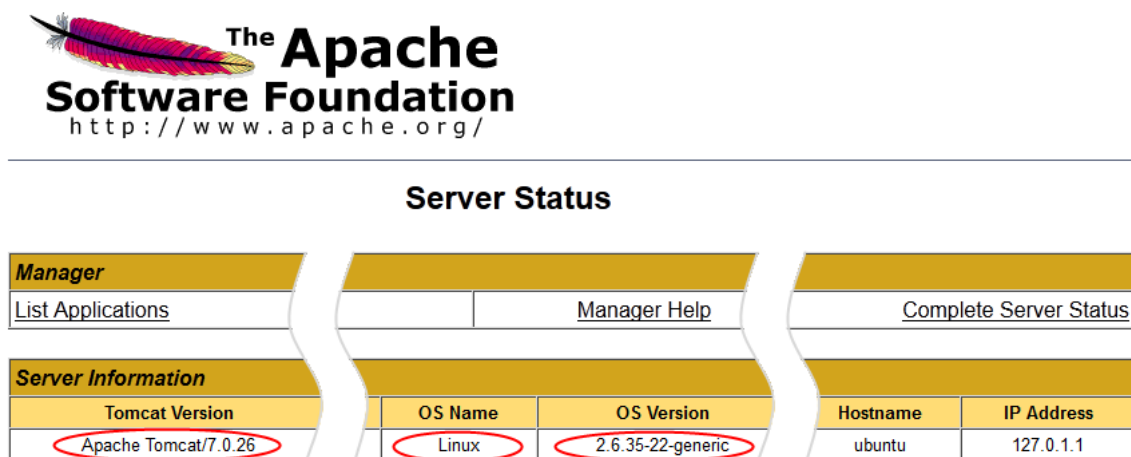


Figure 10 – Tomcat Web Console

Through parsing of this HTML document, one could detect the exact Tomcat version, e.g. in this example Apache Tomcat 7.0.26. Additionally the console provides information about the operating system (see next section). In this case, the OS name is Linux and the version is 2.6.35-22-generic.

⁷ <http://httpd.apache.org/>

Furthermore, Tomcat can be configured to use JMX for managing and monitoring of the server. Through the use of a JMX console (e.g. *JConsole*⁸) one could gain access to a running Tomcat instance. The example in Figure 11 shows a screenshot of JConsole being connected to a Tomcat instance.

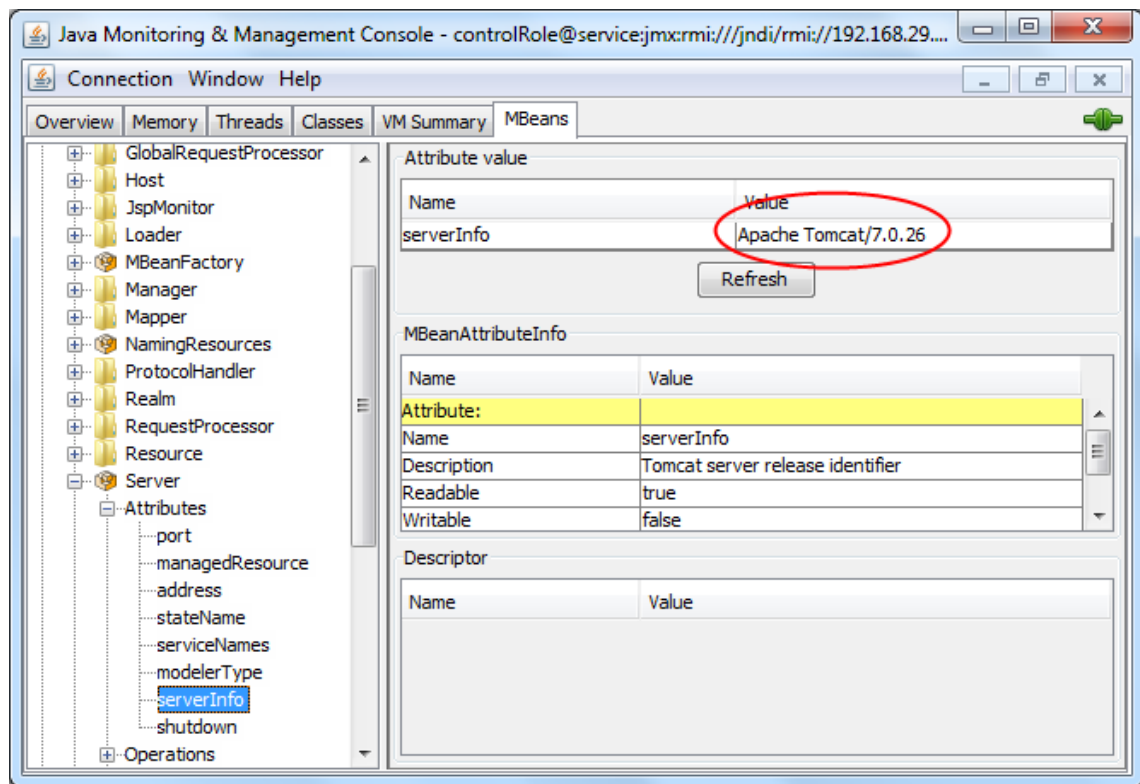


Figure 11 – Monitoring Apache Tomcat with JConsole

The connection string to connect to Tomcat via JMX is shown in Listing 10, where *host* and *port* have to be adapted to the host name and port number that Tomcat is listening on.

```
service:jmx:rmi:///jndi/rmi://host:port/jmxrmi
```

Listing 10 – Tomcat JMX connection string

As seen in Figure 11, the *serverInfo* attribute of the *server* object shows the server name and the server version. In this example, the name is Apache Tomcat and the version number is 7.0.26. Additionally, the operating systems name and version number could also be determined through JMX as it was with the management console shown above.

Another way to query Tomcat for information is to use the JMX-Proxy of the manager application, provided the application is installed. The following Listing shows a call to the servlet.

⁸ <http://docs.oracle.com/javase/1.5.0/docs/guide/management/jconsole.html>

```
http://host:port/manager/jmxproxy?qry=Catalina:type=Server
```

Listing 11 – Calling Tomcat JMX Servlet

The result of the query is shown in Listing 12.

```
OK - Number of results: 1

Name: Catalina:type=Server
modelerType: org.apache.tomcat.util.modeler.BaseModelMBean
port: 8005
managedResource: StandardServer[8005]
address: localhost
stateName: STARTED
serviceNames: Array[javax.management.ObjectName] of length 1
Catalina:type=Service
serverInfo: Apache Tomcat/7.0.26
shutdown: SHUTDOWN
```

Listing 12 – Result of a JMX Servlet query

The result shows again the *serverInfo* element which contains the server name and version number. The possibilities for JMX queries through the servlet match the ones of a real JMX connection. Hence, the operating system can also be identified through the JMX servlet.

3.5.4 Operating System

Detecting the Operating System (OS) is similar to web server detection – more specific – the use of fingerprinting. Like HTTP responses are used to fingerprint web servers, OS detection programs analyze how systems respond to TCP/IP probes. These responses represent a fingerprint. Tools like Nmap [27] usually have a huge database of heuristics to identify different systems.

Another way to detect the OS is using programs that run on the machine where the OS is running which provide this information, assuming one has access to these programs, e.g. a web server like Tomcat usually knows the OS it is running on and sometimes even provides this information with an HTTP response or via a web console (see previous section). This way, a web server plugin that detects the web server type could also detect the operating system.

3.5.5 ActiveMQ

There are various ways to monitor ActiveMQ, e.g. using the Web Console by pointing the browser at <http://localhost:8161/admin>, which is available since version 4.2 and later (Figure 1).

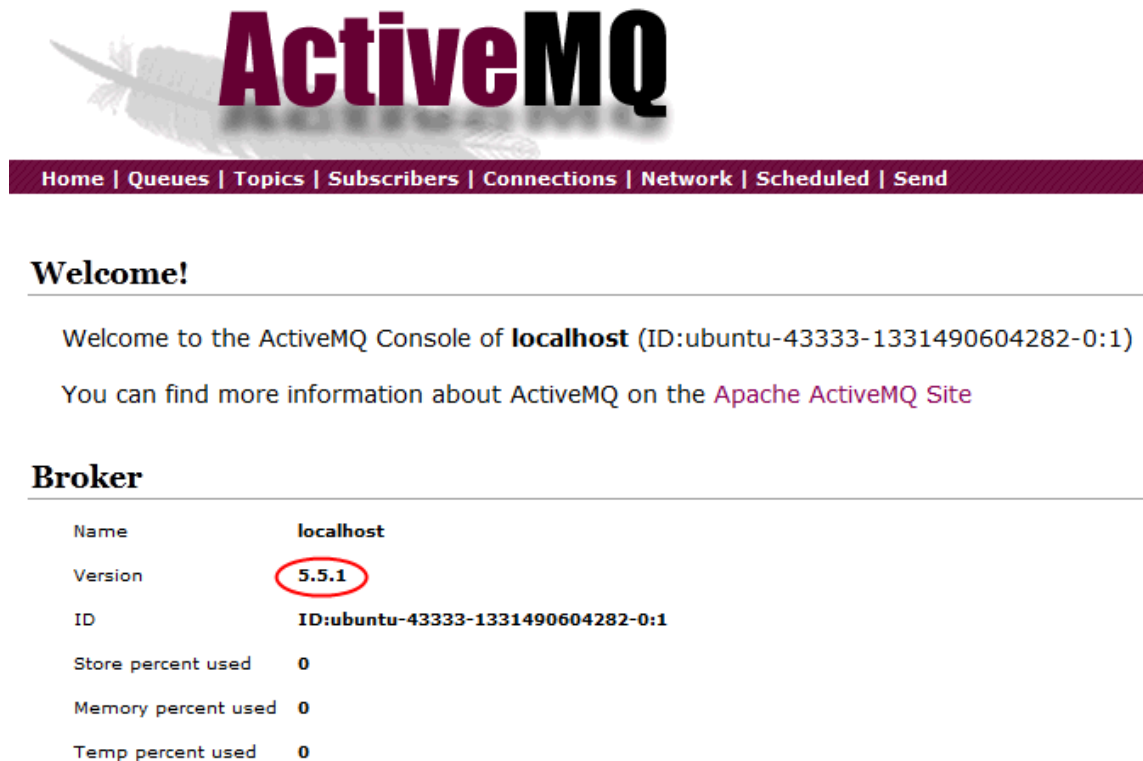


Figure 12 – ActiveMQ Web Console

For external discovery of a running ActiveMQ instance, the HTML content of this generated website could be parsed for the actual version number of ActiveMQ, much like it can be done with the Tomcat management console.

Alternatively, one can use JMX support to view the running state of ActiveMQ. JMX support can be enabled or disabled by

1. Running a broker with the broker property *useJMX* set to *true*, e.g.

```
broker:(tcp://host:port)?useJmx=true
```

Listing 13 – Running a broker with *useJMX* property

2. Running a JMX console (e.g. *JConsole*)
3. And connecting to the given URL, e.g.

```
service:jmx:rmi:///jndi/rmi://host:port/jmxrmi
```

Listing 14 – Connecting to URL

The host and port elements again have to be set to the corresponding values.

4 Architecture and Design

The previous chapters already covered the different technologies and the concepts that the framework and plugins must support. This chapter explains the resulting architecture and design decisions. It also highlights the possible advantages and disadvantages of the approach and identifies potential difficulties and problems.

4.1.1 Use Cases

The following is a description of the different use cases that the framework has to support. This includes use cases for the creation of a new (empty) project, loading and saving of projects, editing of a project (internally or externally via a text editor) and the start – respective restart – of the discovery process (Figure 1). The user should also have the possibility to develop new plugins, and to install or uninstall them.

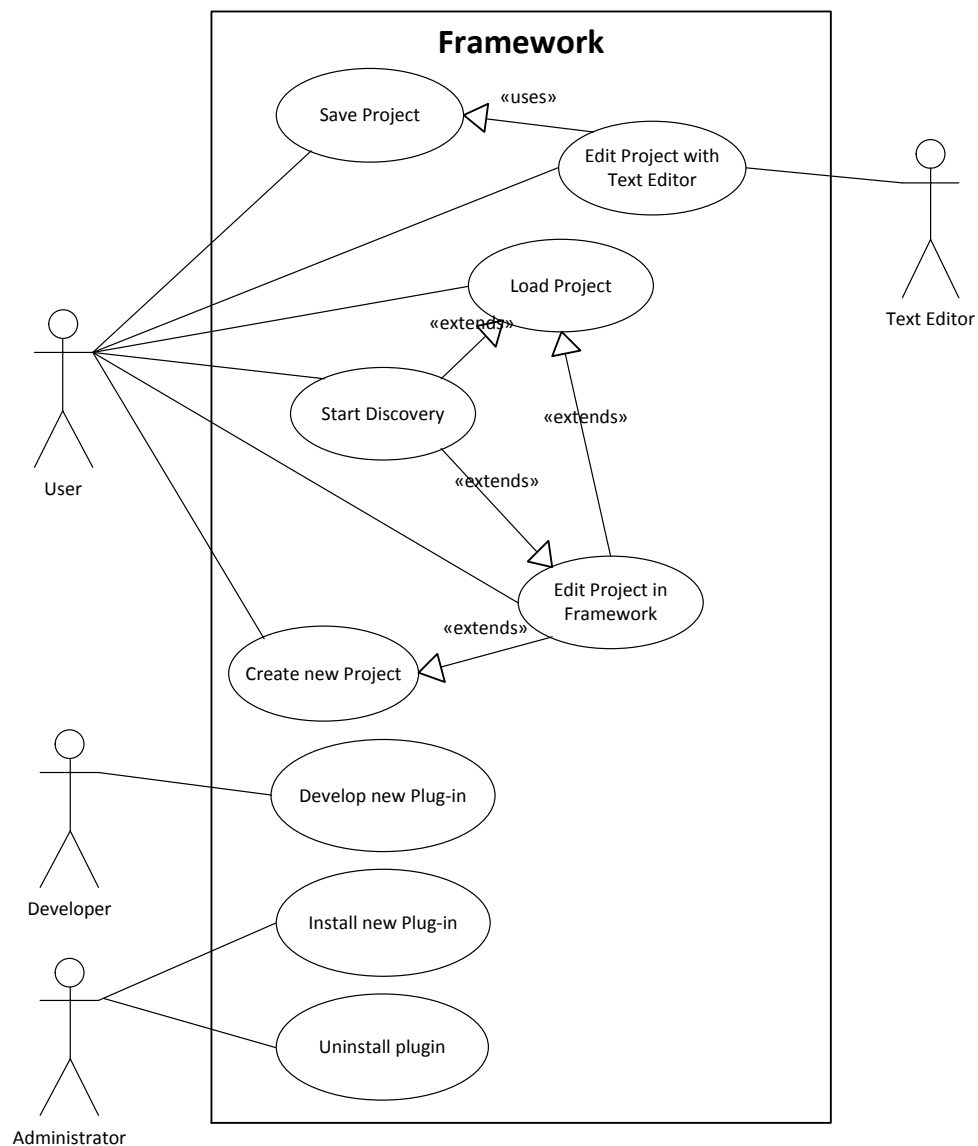


Figure 13 – Use Case Overview and Roles

Name	Create New Project
Goal	The framework should provide the user with a newly created (empty) project.
Actor	A user that wants to use the framework to start a new application topology discovery process.
Pre-Condition	The framework is running.
Post-Condition	A new, empty project is created and presented to the user.
Post-Condition in Special Case	No project was created by the framework. The user is notified that the project could not be created. The framework reverts back to the state before the user action.
Normal Case	The framework creates an empty project with the name and location provided by the user.
Special Case	The framework is unable to create a project.

Table 2 – Use Case: Create New Project

Name	Load Project
Goal	The framework should load the provided file into the framework and present it to the user.
Actor	A user that wants to use the framework to view a previously created discovery project.
Pre-Condition	The framework is running.
Post-Condition	The project is loaded and presented to the user.
Post-Condition in Special Case	The project is not loaded. The user is notified that the framework was unable to load the project. The framework continues with the state before user action.
Normal Case	The framework loads the project with the given filename from the provided location.
Special Case	The framework is unable to load the project.

Table 3 – Use Case: Load Project

Name	Save Project
Goal	The framework should save the currently active project at the user provided location.
Actor	A user that wants to save the current discovery project as a file.
Pre-Condition	The framework is running and the user has either loaded or created a new project.
Post-Condition	The currently active project is saved to the file system at the given location with the given file name.
Post-Condition in Special Case	The project is not saved to the file system. The user is notified that the framework was unable to save the project. The framework continues where it stopped before user action.
Normal Case	The Framework stores the currently active project to the file system.
Special Case	The Framework is unable to store the project.

Table 4 – Use Case: Save Project

Name	Edit Project in Framework
Goal	The user can edit the currently active project.
Actor	A user that wants to change or add information to the project.
Pre-Condition	The framework is running and the user has either loaded or created a new project.
Post-Condition	Project contains new information or certain information changed according to the users request (e.g. a new BPEL node was added to the project)
Post-Condition in Special Case	No changes were made to the active project. The framework continues in the state it was before user action. The user is informed that the requested action could not be completed.
Normal Case	The framework changes information or adds information according to the request of the user.
Special Case	The framework is unable to complete the request of the user.

Table 5 – Use Case: Edit Project in Framework

Name	Edit Project with Text Editor
Goal	The user should be able to modify a saved project with a (text) editor so that projects can be modified outside of the framework.
Actor	A user that wants to add or change information in a project without using the framework (e.g. because the framework is not available at the moment).
Pre-Condition	A project has been saved to the file system and that project is available as a file. Furthermore, an editor (e.g. a text editor) that is able to read the file format of the project must be available.
Post-Condition	The project was altered and the changed project has been saved to the file system.
Post-Condition in Special Case	No changes were made to the project. The file remains in its state without being altered.
Normal Case	The user edits the project in an editor and saves the changes.
Special Case	The user makes no changes to the project (e.g. the file is only viewed with an editor).

Table 6 – Use Case: Edit Project with Text Editor

Name	Start Discovery
Goal	The framework should start the discovery process on the currently active project. The provided information in the project is used to discover more information by using programs (plugins). The order and execution of the programs is managed by the framework.
Actor	A user that wants to start a discovery on a certain infrastructure.
Pre-Condition	The framework is running and currently there is an open and active project. Furthermore, the current project contains enough information for the start of the discovery process. To restart the discovery (if there was a previous discovery on the project), either additional information has been added to the project, or additional programs have been added to the framework.
Post-Condition	The discovery process discovered new information that is added to the project.
Post-Condition in Special Case	The discovery process did not discover any new information. The project is not altered.
Normal Case	The discovery process uses the provided information in the project to discover new information. This information is added to the

project.

Special Case	The discovery process tries to discover new information with the provided information, but no additional information is discovered.
---------------------	---

Table 7 – Use Case: Start Discovery

Name	Develop new Plugin
Goal	Develop a new plugin so that additional functionality is bundled as a plugin and later installed to the framework.
Actor	Developer that bundles additional functionality in a plugin.
Pre-Condition	A development tool must be available.
Post-Condition	A new plugin is created and bundled as an installable form for the framework.
Post-Condition in Special Case	No plugin was created.
Normal Case	The plugin is created and bundled as an installable form.
Special Case	The plugin is not created. Development is paused or canceled.

Table 8 – Use Case: Develop new Plugin

Name	Install new Plugin
Goal	Add additional programs (plugins) to the framework so that they can be used by the framework to discover additional information.
Actor	Administrator that is allowed to install new plugins to the framework.
Pre-Condition	The framework is available (i.e. on the file system)
Post-Condition	The provided plugin is installed to the framework.
Post-Condition in Special Case	No plugin was installed to the framework. The user is informed that the request could not be completed. No changes were made to the framework.
Normal Case	The plugin is installed to the framework and added to the list of known plugins.
Special Case	The plugin is not installed to the framework.

Table 9 – Use Case: Install new Plugin

Name	Uninstall Plugin
Goal	Remove installed plugins from the framework.
Actor	Administrator that is allowed to uninstall plugins from the framework.
Pre-Condition	The framework is available.
Post-Condition	The selected plugins are no longer available in the framework.
Post-Condition in Special Case	No plugin was uninstalled from the framework. The user is informed that the request could not be completed. No changes were made to the framework.
Normal Case	The framework removes the selected plugins.
Special Case	The framework is unable to remove the plugins.

Table 10 – Use Case: Uninstall new Plugin

4.2 Design Decisions

At the beginning of the decision process for a specific architecture or design, there is always the question whether there are parts of the architecture that are already developed or available and what parts must be developed newly. Some decision requirements were already provided by the assignment of tasks for this diploma thesis, e.g. that the framework should be plugin-based. The advantages and disadvantages of this plugin-based approach are already explained in detail in the conceptual chapter of this document (Section 3.1).

4.2.1 Java

Other decisions are based on the knowledge and experience that the author of this document has with different technologies, in due consideration of the possible advantages and disadvantages and the applicability of these technologies. For example, the author was familiar with the development of applications in Java. Though Java was the preferred programming language of the author, it is at the same time one of the most common programming languages and characterized by its ability to run on a wide range of systems. The interoperability of Java predestinates the language for the development of a plugin-based framework, making the application available on different systems and through its wide acceptance by developers the ideal solution for a framework that should be extendable by other developers.

On a low level basis a Java class or a Java interface could already represent a plugin. A framework – represented by a class – could lookup all classes that implement a certain interface. Adding of plugins could be done by adding additional classes to the

CLASSPATH. But adding plugins dynamically could become very difficult. This is why an advanced approach that supports installation and management of plugins is desirable, e.g. like an OSGi-based approach.

4.2.2 OSGi

The *Open Services Gateway initiative (OSGi)* [28] is a dynamic module system for Java. While Java provides the technology to run programs on different platforms, OSGi provides the technology to construct applications from reusable and collaborative components. One benefit of the service platform is the possibility to install, update, start, stop and uninstall service applications (*bundles*) both dynamically and controlled at run time. Those independent and modular bundles can run in parallel inside the same Java Virtual Machine (JVM) and they can be managed and updated throughout the whole lifecycle. Dependencies between bundles are automatically resolved and version management is available.

The origin of OSGi is in embedded systems and that is why it is often used in automobiles, mobile devices, and building automation like assisted living and facility management. In addition, a famous example of the usage of OSGi is the Eclipse IDE⁹. Eclipse uses the Equinox OSGi framework¹⁰ and since version three of Eclipse, every plugin is an OSGi bundle. As of this writing the current version of the OSGi specification is 4.3.

An OSGi framework is an open, modular and scalable service delivery platform based on Java and provides a standardized environment to applications (bundles). It is a component model with a service registry but the term *service* means nothing more than an interface and is not to be mistaken with the term service in a Service Oriented Architecture (SOA), though OSGi can be used as a fundamental component model for a SOA. The *OSGi Alliance* specifies only the execution environment, the API and the test cases for third party OSGi implementations. A reference implementation of an OSGi framework is provided by the OSGi Alliance but it is not intended for productive use [29].

The benefits of using an OSGi framework over a simple (self-developed) Java application for the framework developed in this diploma thesis are obvious. Considering a plugin as a bundle in an OSGi environment, an OSGi framework already provides everything that is needed for installation, un-installation, and managing of these plugins. The concept of OSGi has proven itself to be valuable in various projects, and many OSGi implementations are technically mature. This is why OSGi is the chosen environment for the developed application.

⁹ <http://www.eclipse.org/>

¹⁰ <http://www.eclipse.org/equinox/>

4.2.3 Eclipse RCP

The previous chapter already identified the Eclipse IDE as an OSGi framework through the use of the Equinox OSGi framework. While the Eclipse platform is designed to serve as an open tools platform, it is architected in a way that its components can be reused to build almost any client application [30]. The minimal set needed to build a rich client application is commonly known as the *Eclipse Rich Client Platform (RCP)*¹¹. That is why many of the aspects and components of the Eclipse IDE can be found in other Eclipse-based applications, for example the workbench design of the user interface, the extensible plugin system, the help components, and the update manager [31]. The strength of the platform is its set of mature components for graphical applications, a consistent elaborated concept of operations, an amount of extensions, tools and support, and the integration of development tools for plugin development into the Eclipse IDE (Eclipse for RCP/plugin developers) [31]. As the components can be used separately – i.e. independent from the IDE – Eclipse RCP is the predestinated solution for the plugin-based framework for application topology discovery, developed throughout the course of this diploma thesis. The components that make up Eclipse RCP are illustrated in Figure 14.

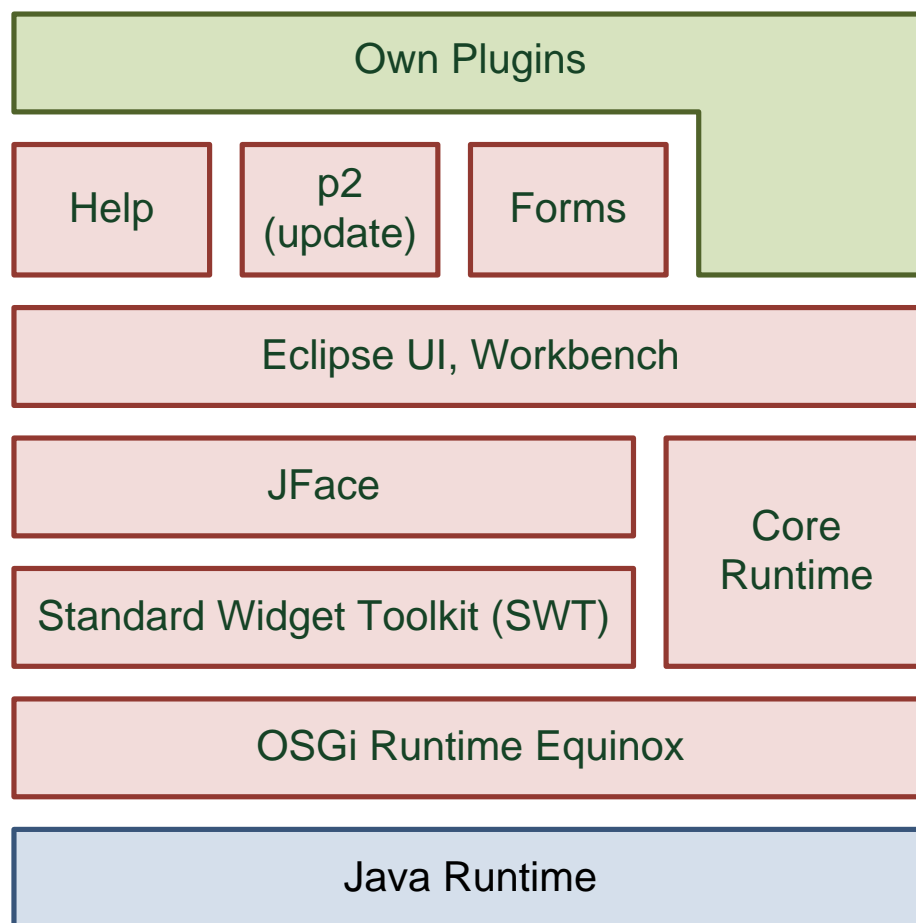


Figure 14 – Components of Eclipse RCP (Adapted from [31])

¹¹ <http://www.eclipse.org/rcp/>

- **OSGi Runtime Equinox**: OSGi specifies the runtime for the execution of modules. Eclipse Equinox implements this specification. Eclipse plugins are executed in Equinox.
- **Core Runtime**: provides general, non-UI functionality for Eclipse applications, e.g. life-cycle management and initialization of applications.
- **Standard Widget Toolkit (SWT)**: the UI toolkit of the Eclipse platform. It provides a minimal abstraction layer for the UI-widgets of the operating system.
- **JFace**: provides advanced functionality like the supplying of UI-widgets with data from Java objects.
- **Eclipse UI**: provides the workbench, an empty graphical application that supports *Views*, *Editors*, *Perspectives*, *Menu-structures*, etc. which can be extended by plugins.

The decision to use Eclipse RCP (with Equinox as the OSGi implementation) is also based on the possibility to easily add a graphical user interface to the application, which was not required by the assignment of tasks, but strongly helps to visualize the discovered information. Its wide acceptance by developers and its great community has also led to a good documentation and provides the needed information for development.

4.2.4 Eclipse Modeling Framework (EMF)

The *Eclipse Modeling Framework (EMF)* [32] is a Java/XML framework for generating tools and other applications based on simple class models. The intension of EMF is to provide easy formal modeling and code generation. Objects can be saved as XML documents and models can be created through the use of annotated Java, XML documents or modeling tools. The modeled objects can be turned into a set of Java classes that can be extended and regenerated, which means that the developer can add methods and variables that endure the regeneration of the code. These changes can also be used to update the model [33]. Furthermore, it provides a set of adapter classes for viewing and command-based editing of the model, and a basic editor.

EMF consists of three fundamental pieces [32]:

- **EMF** – The EMF framework contains a Meta model (an Ecore file) for the description of models and the runtime support for these models, persistence support, and a reflective API for the manipulation of objects.
- **EMF.Edit** – The EMF.Edit framework allows for the creation of editors for the EMF models by providing generic reusable classes. This includes content and label provider classes, property source support, and classes that support displaying of EMF models using standard desktop viewers (JFace). Additionally it provides a command framework for building of editors that support automatic undo and redo.
- **EMF.Codegen** – The EMF.Codegen framework is responsible for the generation of the code up to a complete editor for the EMF model.

Basing the developed framework on OSGi or rather Eclipse RCP with the Equinox OSGi framework already introduces a lot of valuable features that are required by the framework, especially because these tools and frameworks do not have to be newly developed. Using EMF for the definition of the data model, with the benefit of the subsequent code generation additionally eases the development process, because the generated editors already provide editing capabilities as well as persistence of the model, especially to XML files which is the preferred format for the exchange with other applications. The generated code is available as plugins that can run inside the Eclipse framework and hence are perfectly suited for integration with an Eclipse RCP application.

4.2.5 Graphical Modeling Framework (GMF)

The *Graphical Modeling Framework (GMF)* is a framework for building modeling-like graphical Eclipse-based editors [34], e.g. business process editors, flow editors, and UML editors. The framework has two components:

- *Tooling* – consisting of editors to create and edit models describing the notational, semantic and tooling aspects of a graphical editor and the code generator.
- *Runtime* – providing the runtime for the generated plugins

Before GMF was introduced, graphical frameworks often used EMF and the Graphical Editing Framework (GEF) [35]. As there were different technical challenges integrating EMF and GEF (e.g. because of different command stacks), the GMF project was developed to bridge the two technologies. In the same way EMF generates editors for EMF models, GMF generates graphical editors. The generation process is illustrated in Figure 15.

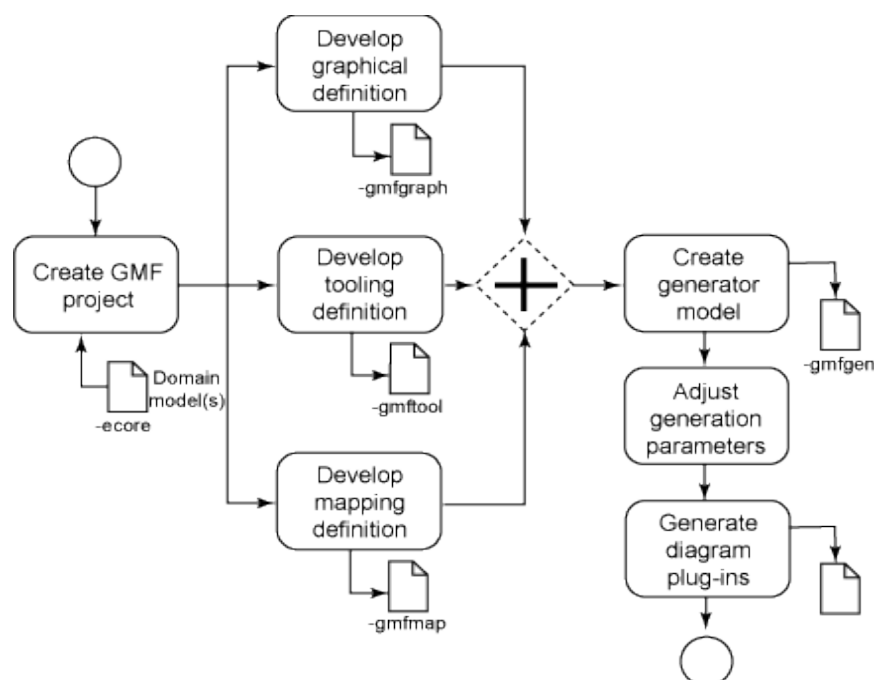


Figure 15 – GMF overview [36]

After the creation of the domain model (Ecore file), the first model to generate is the graphical definition that defines the visual aspects of the generated editor, e.g. the figures that are to be displayed on the diagram. The tooling definition contains information about the editor palettes and menus, while the mapping definition defines the mapping between the business logic (EMF model) and visual model (graphical and tooling definition) [36]. The final step of the process is the generation of the code for the editor.

The decision to use GMF to develop a graphical editor has mainly the reason to give the user a graphical representation of the discovered topology. As the core functionality is still handled by the EMF framework, e.g. persistence and programmatically editing of the model, GMF can be seen as an additional feature of the implemented framework, though GMF has far more features to show than actually used in this implementation. Eclipse provides a great utility called GMF dashboard that serves as an easy way to go through the process of generating a graphical editor. Furthermore, a tutorial [37] provides a good starting point for the development of GMF applications.

4.3 Resulting Architecture

The following architecture is the result of the use of OSGi, Eclipse RCP, and Eclipse EMF and GMF as design decision.

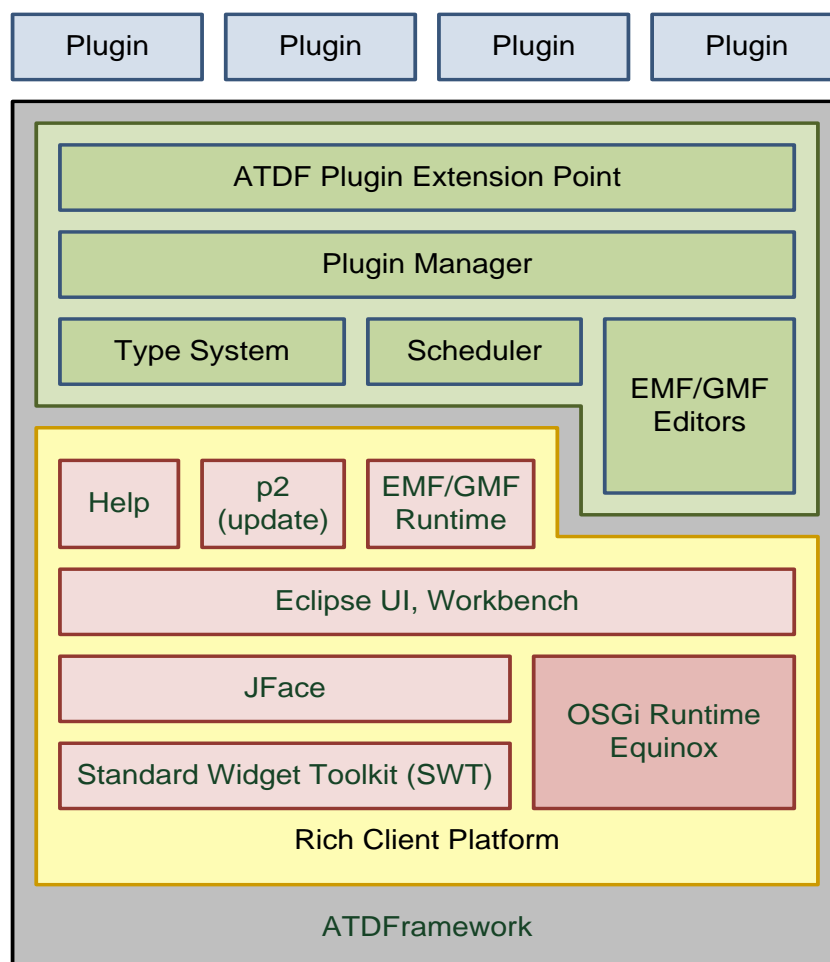


Figure 16 – Resulting architecture of the ATDFramework

The architecture mainly consists of three parts: (1) a set of plugins that form an Eclipse RCP application, (2) a set of plugins that together with the RCP plugins build the framework, and (3) a set of discovery plugins that are not bundled with the framework but developed independently to extend the framework.

4.3.1 ATDFramework

The ATDFramework contains the set of plugins that make up an Eclipse RCP application. These plugins do not need to be developed. They are generated or rather gathered by the code generator of GMF, the descriptor files (*plugin.xml*) of the different participating plugins when they specify a dependency to a specific plugin, and the product configuration file of the framework. The required components that are integral part of every user interface-enabled RCP application are the OSGi Runtime Equinox, and the Workbench that uses JFace and the Standard Widget Toolkit (SWT). Optional components, but needed by the ATDFramework, are the help system, p2 update mechanism, and the EMF/GMF runtime that is needed by the EMF/GMF editors. Figure 16 shows all these components in red and yellow.

The green parts of the diagram represent the developed parts of the framework. These are the type system that manages the installed types, the scheduler that controls the scheduling and the execution of plugins, and the plugin manager that manages the installed plugins and is mainly used as an agent between framework and plugins to make the framework more robust, e.g. to prevent the framework from crashing if a single plugin crashes. The EMF and GMF editors are generated according to the specified data model and the graphical, tooling, and mapping definition. A particular component is the ATDF Plugin Extension Point. Extension points facilitate the possibility to contribute functionality to a plugin by other plugins. A plugin can open itself up for other plugins through the definition of an extension point. This extension point then defines a contract how other plugins can contribute functionality. For example, Eclipse defines extension points for menu entries of the toolbar of Eclipse. One can add menu entries by contributing an extension to the menu extension point. The ATDFramework defines such an extension point so that other plugins can contribute functionality to the ATDFramework by contributing an extension. This extension point is the contract between the ATDFramework and the plugins and defines what functions a plugin must provide to contribute to the ATDFramework.

4.3.2 Plugins

The plugins are independent contributions to the framework by other developers. They are not an actual part of the framework, but they must conform to the contract that is defined by the ATDF plugin extension point. This contract is a simple Java interface that contributing plugins must implement.

It is worth noticing, that the overall application is still an RCP application and hence can be extended by any plugin type. This allows plugins that extend the ATDFramework by

providing actual discovery functionality also to provide additional functionality to other extension points, e.g. contributions to the menu of the ATDFramework. This gives the plugin developer a lot of options, e.g. a BPEL plugin can provide a menu entry that calls a dialog where the user provides a URL to the BPEL file. The plugin could then automatically add a new node to the project with the location property set to the specified URL.

5 Prototypical Implementation ATDFramework

The ATDFramework was developed as a prototype using the Eclipse IDE. The source code is available as different Eclipse projects. This chapter describes the necessary steps to create the GMF based project and to generate the code based on the created definition files (5.1), as well as the release to a final product. Furthermore, the extensions to the generated code that make up the framework (5.2) as well as each discovery plugin (5.3) are described in detail.

5.1 Project Setup and Modeling

The project setup follows many of the steps of the Eclipse GMF tutorial [37] with adjustments to the different definition files according to the data model of the ATDFramework. The used versions during development are Eclipse 3.7 (Indigo), GMF runtime version 1.5, and GMF tooling version 2.4.

Development of GMF-based applications becomes very simple through the use of the provided dashboard view. It accompanies the user throughout the whole definition process, e.g. it defines a diagram where the user creates or adds new definition models and displays the progress (Figure 17).

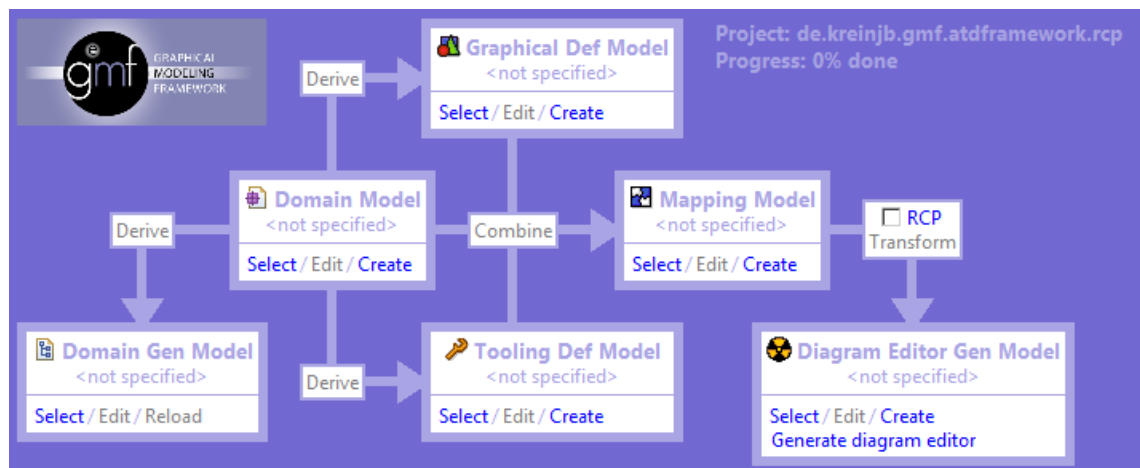


Figure 17 – GMF Dashboard

The first step is the creation of a new GMF project found under the *Graphical Modeling Framework* category in the *New* dialog (File -> New), with the dashboard option enabled. The name of the project is *de.kreinjb.gmf.atdframework*. The new created project contains a model folder where all the model definition files are stored.

5.1.1 Domain Model

The next step is the creation of the domain model that is stored in an Ecore file called *atdframework.ecore*. The complete model is shown in Figure 18. The model is created using the integrated modeler in Eclipse. Other possibilities involve the creation of the Ecore file from an XML schema or an annotated Java interface.

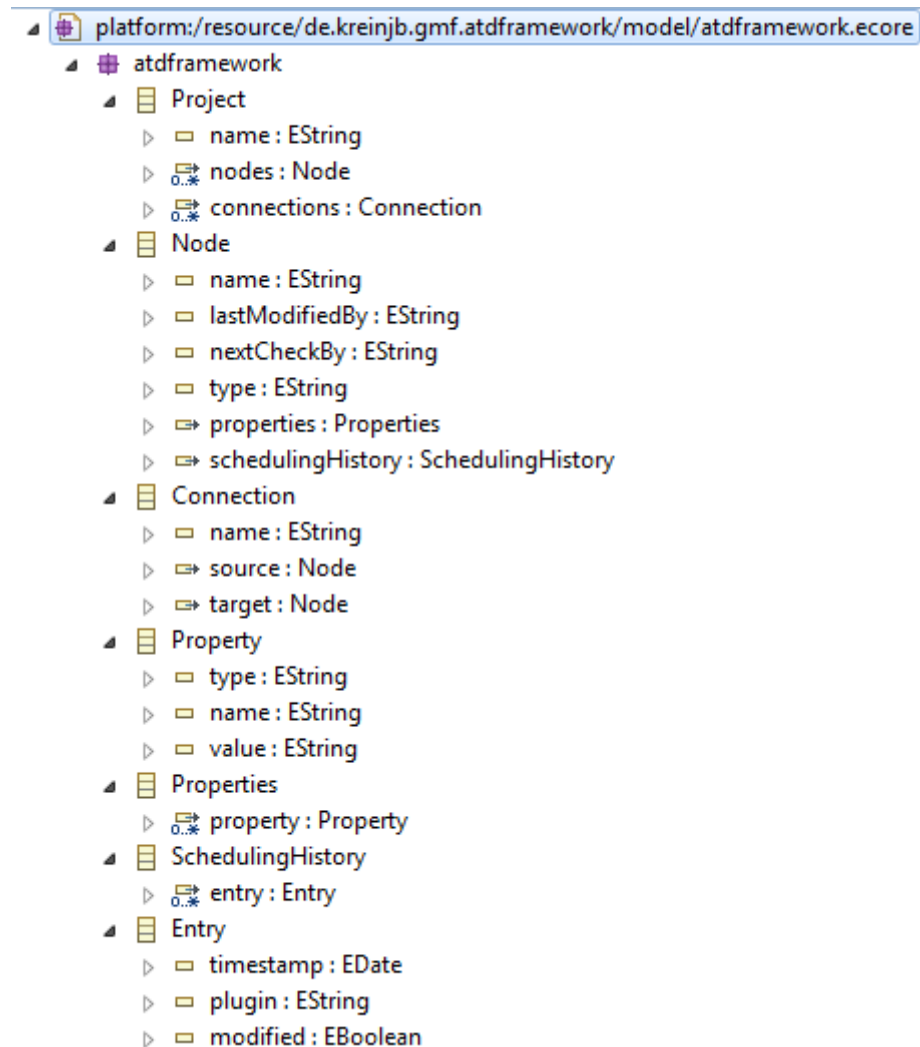


Figure 18 – Domain model (atdframework.ecore)

The Ecore file is selected as the domain model in the dashboard view.

5.1.2 Domain Gen Model

Afterwards, the *genmodel* can be derived from the Ecore file in the dashboard view. The file is named *atdframework.genmodel*. This genmodel file is almost the same as the Ecore file but it contains additional Meta information. For example, the genmodel is used to generate the code for the implemented Java classes and the editors, and hence allows for specifying of the package names of the generated classes. The *package* property in the genmodel is set to *atdframework* and the *base package* property is set to *de.kreinjb.gmf*. This way, all generated packages will start with

de.kreinjb.gmf.atdframework, e.g. the generated editor is generated in the package *de.kreinjb.gmf.atdframework.editor*. Other options that are specified are the *file extension* for the file that contains the model (*.atm*) and the *resource type* (XML). Finally, the Java classes are generated from the genmodel file. This is done by right-clicking the file. The popup menu then allows for generating the individual code parts (individual plugins) or all parts. These are four different plugins. The model code is generated inside the original created plugin that contains the model files. Furthermore, the generated plugins are an *Edit* plugin (*de.kreinjb.gmf.atdframework.edit*), an *Editor* plugin (*de.kreinjb.gmf.atdframework.editor*), and an optional test plugin (*de.kreinjb.gmf.atdframework.tests*). To show the whole amount of generated classes would go beyond the scope of this document. The four generated projects and their packages are illustrated in Figure 19.

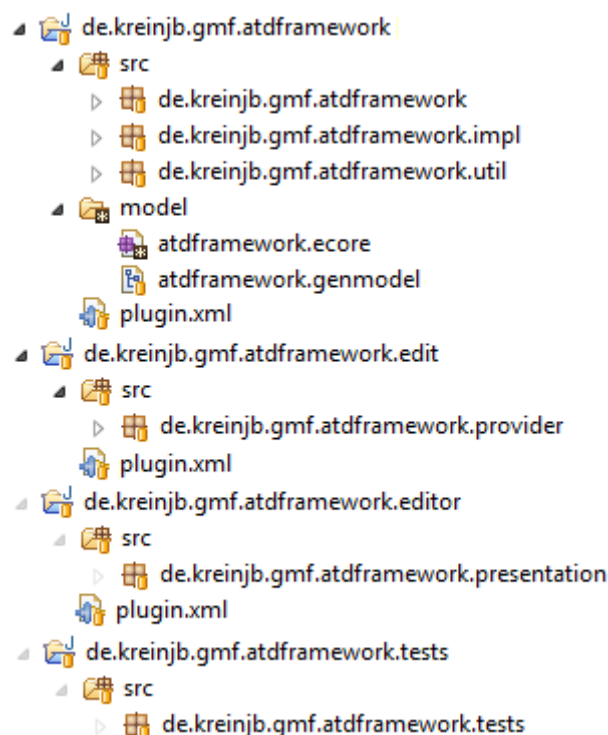


Figure 19 – Project structure after EMF code generation

The packages in the *de.kreinjb.gmf.atdframework* project contain the interfaces and the factory to create the Java classes (*de.kreinjb.gmf.atdframework*), the concrete implementation of the interfaces defined in the model (*de.kreinjb.gmf.atdframework.impl*), and the adapter factory (*de.kreinjb.gmf.atdframework.util*). Java classes for the editor are in the *de.kreinjb.gmf.atdframework.edit* and *de.kreinjb.gmf.atdframework.editor* project. Test classes can be found in the *de.kreinjb.gmf.atdframework.tests* project.

5.1.3 Graphical Def Model

The next step is the graphical definition. The graphical model definition can again be derived from the domain model using the dashboard. In the opening dialog, the *project*

element must be selected as the diagram element. The domain elements that must be processed are *Node* as nodes element, *Connection* as connections element, and the *Node-Name* and *Connection-Name* as attributes. At the end of the process a new graphical model definition (*atdframework.gmfgraph*) is created in the model folder of the original project. This file contains the graphical representations of the elements in the domain model, e.g. a rectangle shape for a node and a polyline for a connection. Adjustments made to the graphical model are mainly a rounded rectangle instead of a square rectangle for the node and some minor adjustments of the layout of the node. Figure 20 shows the contents of the file after modification.

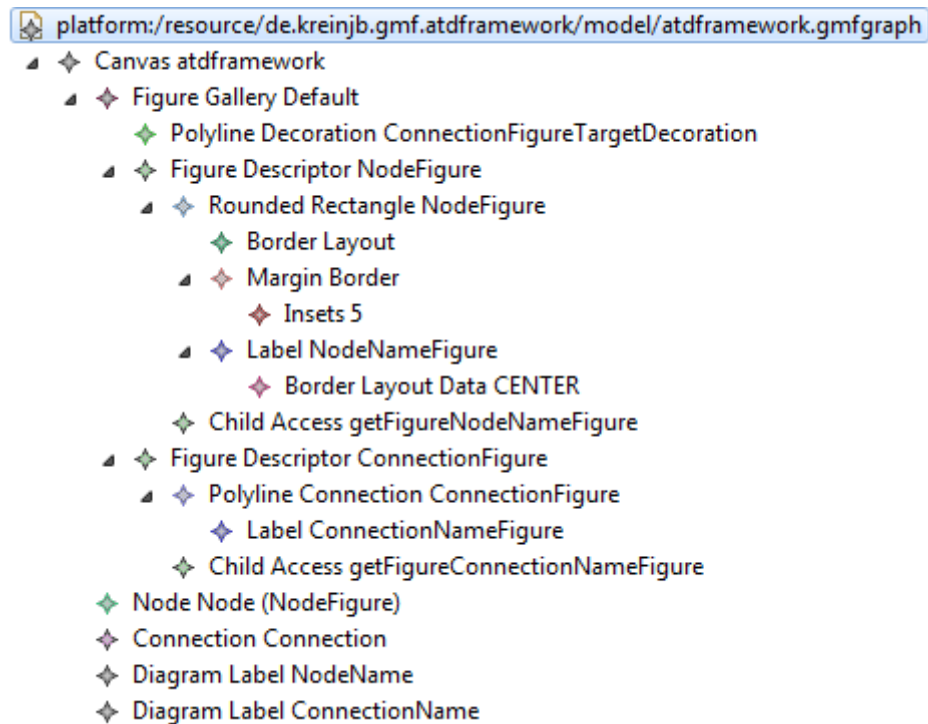


Figure 20 – Graphical model definition (*atdframework.gmfgraph*)

All elements are placed on a *Canvas* object. The *Figure Gallery* contains *Figure Descriptors* that describe the visual appearance of an object. There are *Figure Descriptors* for a *Node Figure* and a *Connection Figure* – the only two graphical figures of the editor. Further elements are *Diagram Labels* that define which attributes of an object are used to label the object, e.g. the *name* attribute of the node is used to label the node on the diagram and the *name* attribute of the connection is used to label the connection on the diagram.

5.1.4 Tooling Def Model

After the definition of the graphical model, the next step again uses the dashboard to derive the graphical tooling definition (*atdframework.gmftool*) from the domain model. The tooling definition describes the palette and the tools that are available in the graphical editor, giving the user the option of graphical modeling. The final tooling definition is illustrated in Figure 21.

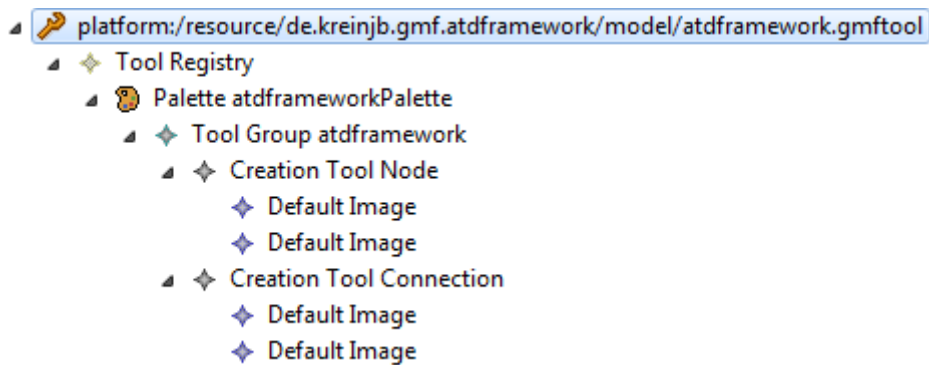


Figure 21 – Graphical tooling definition (atdframework.gmftool)

It is derived similarly to the graphical definition model by selecting *Project* as the *diagram element* and processing *Node* as a node element and *Connection* as a connection element in the opening dialog of the derive action in the dashboard view.

5.1.5 Mapping Model

Finally, the domain model definition, graphical model definition, and graphical tooling definition must be combined into a mapping definition (*atdframework.gmfmap*).

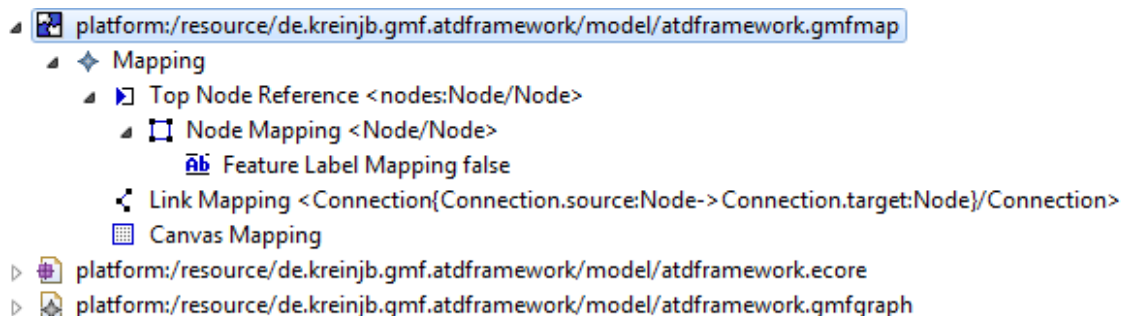


Figure 22 – Graphical mapping definition (atdframework.gmfmap)

The mapping definition defines how the other models work together, e.g. a new figure (graphical definition) is created on the canvas and a new node instance (model definition) is created in the domain model when the user drags a node from the palette (tooling definition) to the canvas. This step is again achieved through the use of the dashboard. In the opening dialog, the *Project* was selected as canvas mapping. The *Nodes* list in the domain model elements mapping contains only the *Node* element, the *Links* list contains only the *Connection* element.

Note: the GMF code generator contains a bug which sometimes creates wrong creation tools in the mapping. This must be checked manually and corrected if necessary.

5.1.6 Diagram Editor Gen Model

The last model that must be created or rather generated is the diagram editor gen model (*atdframework.gmfgen*), which is similar to the creation of the genmodel from an

Ecore file in EMF. This file contains additional information that is needed for code generation and is created by using the transform link in the dashboard. The option for RCP creation is selected because the ATDFramework should be an RCP-based application. After the file is generated it has to be modified to suit the desired final product. The list of adjustments is shown in Table 11.

Property	Value
Diagram File Extension	atd
Domain File Extension	atm
Package Name Prefix	de.kreinjb.gmf.atdframework.rcp
Editor Plugin Directory	/de.kreinjb.gmf.atdframework.rcp/src
Creation Wizard Category Id (Gen Diagram ProjectEditPart)	ATDFramework
Title (Gen Application AtdframeworkApplication)	Application Topology Discovery Framework

Table 11 – Adjustments in atdframework.gmfgen

After adjusting the file, the GMF-related code can be generated which will create an additional plugin project called `de.kreinjb.gmf.atdframework.rcp` that contains all the GMF-related Java classes. The project structure of the generated plugin is illustrated in Figure 23.

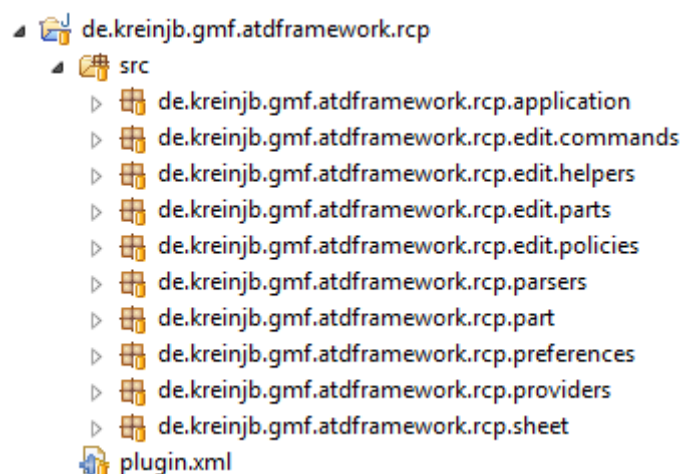


Figure 23 – Project structure of GMF editor

At this stage, the generated code is a fully functional Eclipse RCP-based GMF editor that can be extended and executed. For example, the editor allows for graphical modeling through the use of the provided palette. The model can also be stored to and loaded from a file.

5.1.7 Product Configuration

Eclipse projects can be equipped with a product configuration file. This configuration file can define application-specific branding on top of a configuration of Eclipse plugins and provides the possibility to export a product as a binary. A product must define a name, a description, and an ID for the application it is associated with. It also specifies the application window icon and the information in the *About* dialog of the application.

A new product configuration can be added via the File -> New dialog. The configuration file is called *atdframework.product* and is stored in the original plugin where the models are stored. The ATDFramework will be feature based which means that plugins first have to be bundled as a feature. This option is set in the product configuration together with settings for the application that is launched at startup and the dependencies to other features. The whole framework will be bundled as a feature (*de.kreinjb.gmf.atdframework.feature*) and all other plugins that contribute discovery functionality to the framework will be bundled as a separate feature as well. These features must then be added to the product configuration. The application that is launched at startup is the *AtdframeworkApplication* found in *de.kreinjb.gmf.atdframework.rcp* project that was generated by GMF.

To run the application a new runtime configuration is created that has the just created product configuration selected as the product to run. In the plugin section of the runtime configuration the *launch with* option is set to *selected features below*, and the dependent features are selected.

Application branding is added by adding a splash.bmp to the plugin that contains the product configuration. Furthermore, the product configuration also allows adding of icons that are used in the window of the application and in the taskbar when the application is launched. The icons are also stored in the plugin that contains the product configuration. After adding of icons and specification of these in the configuration, the configuration must be synchronized with the product's defining plugin, or else the icons will not be recognized when the application is launched. Figure 24 shows the branding logo of the ATDFramework.

Another benefit of the product configuration file is the possibility to export the product as binaries, even for different operating systems. Product export is also needed during development when the p2 update mechanism is used (see Chapter 5.1.8), because the update mechanism needs a repository that is created during export. This is achieved by using the *generate metadata repository* option in the export wizard.

Note: product export only works when the developer has administrative privileges.

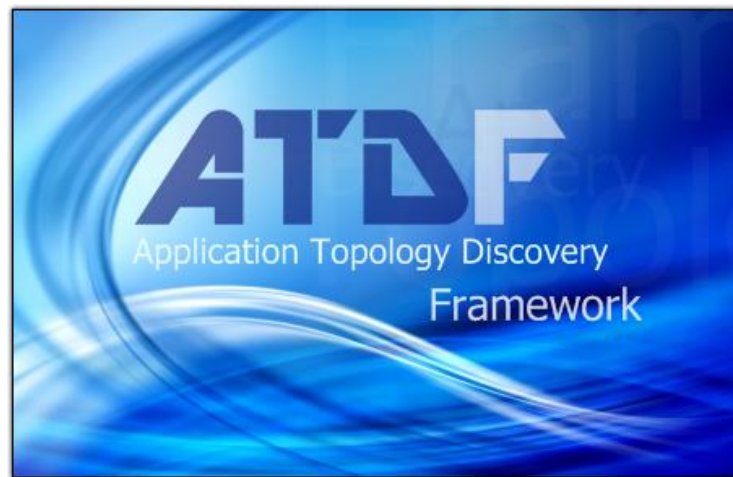


Figure 24 – Branding Logo of ATDFramework

5.1.8 Update Manager

There are different ways to install new plugins into Eclipse or an Eclipse RCP-based application. One involves simple copying of the plugin or feature into specific folders of the Eclipse application. When a plugin is exported it has a specific folder hierarchy that contains the folders *features* and *plugins*. To install the plugin or feature one simply has to copy the contents of these folders into the *features* and *plugins* folder of Eclipse or the Eclipse RCP-based application. The other way, which is recommended by Eclipse, is via the *Update Manager* (see Figure 25).

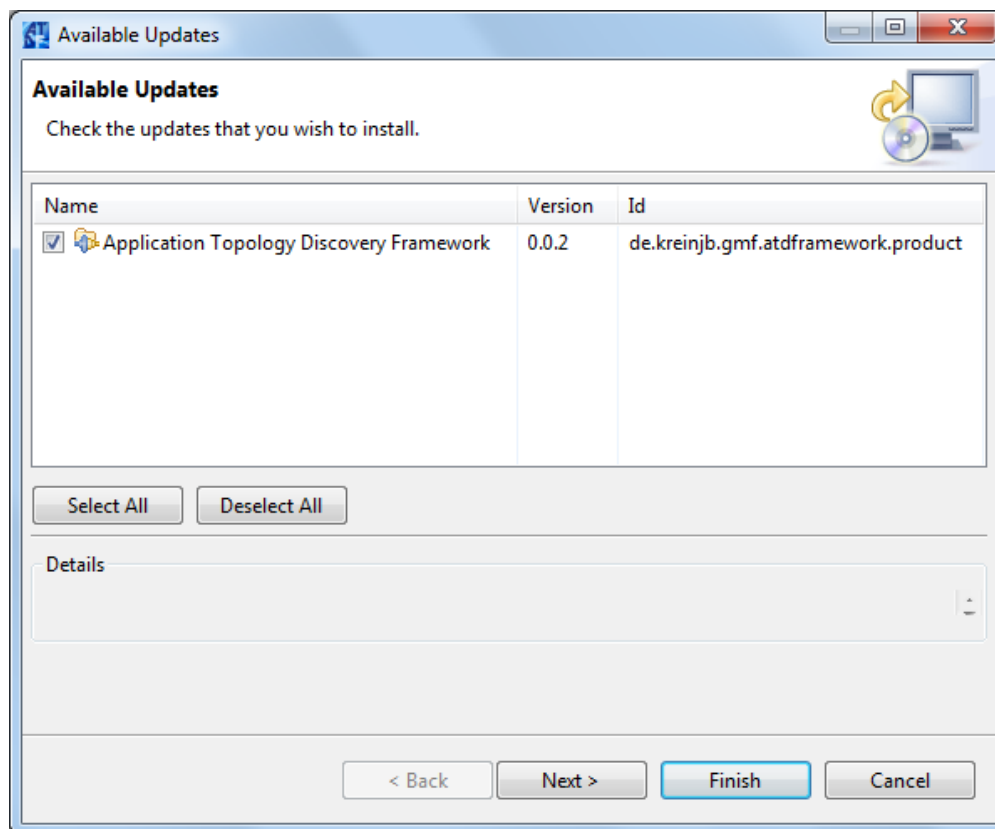


Figure 25 – Update Manager of ATDFramework

The Update Manager provides the possibility to update an application by specifying a *repository* which can be a remote update site or a repository on the local file system. It is possible to update the framework itself as well as the different plugins and to install new plugins.

To add the Update Manager to the ATDFramework different dependencies have to be added to the product configuration, namely *org.eclipse.equinox.p2.rcp.feature* and *org.eclipse.rcp*. They must also be added to the run configuration of the product in the plugins section. The required version numbers are removed from the features when specifying them as dependencies, so the framework does not depend on a specific version but can be update if new versions are released in the future.

5.2 Framework Implementation

The framework extends the GMF-generated plugins with an additional plugin called *de.kreinjb.gmf.atdframework.manager*. This plugin contains all the framework related Java classes that are not generated by EMF and GMF. The *plugin.xml*, which is the configuration file of a plugin, additionally contains the definitions of so-called extensions that are used to contribute to an RCP-based application. Most of the time, these are UI extensions, e.g. contributions to the menus or contributions of additional *views*.

5.2.1 Extension Point ATDFPlugin

An extension point gives plugins the possibility to open itself up for extension by other plugins. It defines a contract between a plugin and another plugin that wants to contribute functionality to the plugin. The ATDFramework defines such an extension point (Section 4.3.1). This extension point is used by discovery plugins to contribute functionality to the framework.

An extension point is defined in the *Extension Points* section of the plugins configuration file (*plugin.xml*) using the provided *add* wizard. For the extension point of the ATDFramework the *id* is *de.kreinjb.gmf.atdframework.manager.plugins* and the *name* is *ATDFPlugin*. In the *definition* view of the created *schema* a new element with the name *plugin* must be added. This element must have an attribute with the name *class*, a type of *java*, and *de.kreinjb.gmf.atdframework.manager.plugins.IATDFPlugin* as the implementing interface. This interface is created using the *implements* link in the definition view. Furthermore, a *choice* element must be added to the element *extension* already available. Its *Max Occurrences* must be set to *unbounded* and a *plugin* sub element must be added to the choice. Listing 15 shows the modified schema.

The interface is modified to meet the needs of the ATDFramework defined in the concepts section of this document (Chapter 3). All discovery plugins must implement this interface. Listing 16 shows the definition of the interface. The package that contains the interface must be exported to be accessible by other plugins. This is done by adding the package to the list of exported packages in the *runtime* view of the *plugin.xml*.

```

<?xml version='1.0' encoding='UTF-8'?>
<!-- Schema file written by PDE -->
<schema targetNamespace="de.kreijnb.gmf.atdframework.manager"
  xmlns="http://www.w3.org/2001/XMLSchema">
  <annotation>
    <appinfo>
      <meta.schema plugin="de.kreijnb.gmf.atdframework.manager"
        id="de.kreijnb.gmf.atdframework.manager.plugins" name="ATDFPlugin"/>
    </appinfo>
    ...
  </annotation>

  <element name="plugin">
    <complexType>
      <attribute name="class" type="string">
        <annotation>
          <appinfo>
            <meta.attribute kind="java"
              basedOn=":de.kreijnb.gmf.atdframework.manager.plugins.IATDFPlugin"/>
          </appinfo>
        </annotation>
      </attribute>
    </complexType>
  </element>
</schema>

```

Listing 15 – de.kreijnb.gmf.atdframework.manager.plugins.exsd

```

package de.kreijnb.gmf.atdframework.manager.plugins;

import java.util.List;

import de.kreijnb.gmf.atdframework.Node;
import de.kreijnb.gmf.atdframework.manager.type.Type;

public interface IATDFPlugin {

    public String getName();
    public String getID();
    public List<Type> getTypesOperatingOn();
    public List<Type> getTypesCreating();
    public List<Node> execute(Node node);
}

```

Listing 16 – IATDFPlugin interface

The interface contains five methods that must be implemented by every discovery plugin.

Method	Description
getName()	A plugin must return a name that will be used to display the plugin in different views of the framework.
getId()	The Id is very important because it is used to identify a plugin, e.g. for scheduling. A user can also use this Id to specify a specific plugin that should be executed in the next iteration of the discovery.
getTypesOperatingOn()	A plugin must specify the types it can operate on so the scheduler knows which nodes the plugin is compatible with. The plugin just returns a list of types it supports, e.g. a BPEL plugin returns a list containing a BPEL type. This list is also used to build up the type system. The types do not need to exist in the framework. They are created if they are not already available.
getTypesCreating()	Besides the types a plugin supports, a plugin can also specify the types it creates. This is mainly used to build up the type system, e.g. a BPEL plugin creates nodes of WSDL type.
execute()	When a plugin is scheduled for execution the scheduler calls the execute function of the plugin and provides the node that the plugin should operate on. The responsibility of the plugin developer is to return the list of modified or created nodes.

Table 12 – IATDFPlugin interface methods

A popular method when developing interfaces is to provide an abstract class that implements the defined interface. Instead of implementing the interface other classes simply inherit from the abstract class. This is especially useful when a lot of the methods of an interface would have similar or the same content in all implementing classes. The abstract class provides a standard implementation of the methods of the interface and classes that inherit from this abstract class use this implementation and override only the methods where they need unique content. The ATDFramework provides such an abstract class which is called *AbstractATDFPlugin* that implements the interface IATDFPlugin. It is recommended to plugin developers to subclass from this class instead of implementing the interface directly. The class is shown in Listing 17.

```
package de.kreinjb.gmf.atdframework.manager.plugins;

import java.util.ArrayList;
import java.util.List;
import de.kreinjb.gmf.atdframework.Node;
import de.kreinjb.gmf.atdframework.manager.type.Type;

public abstract class AbstractATDFPlugin implements IATDFPlugin {
    protected final String id;
    protected final String name;
    protected final ArrayList<Type> operatingTypes;
    protected final ArrayList<Type> creatingTypes;

    public AbstractATDFPlugin(String id, String name) {
        this.id = id;
        this.name = name;
        this.operatingTypes = new ArrayList<Type>();
        this.creatingTypes = new ArrayList<Type>();
    }

    @Override
    public String getName() {
        return name;
    }

    @Override
    public String getID() {
        return id;
    }

    @Override
    public List<Type> getTypesOperatingOn() {
        return operatingTypes;
    }

    @Override
    public List<Type> getTypesCreating() {
        return creatingTypes;
    }

    @Override
    public abstract List<Node> execute(Node node);
}
```

Listing 17 – AbstractATDFPlugin

The abstract class implements the methods *getName()*, *getID()*, *getTypesOperatingOn()*, and *getTypesCreating()*. These functions return values that are stored in *protected* attributes which are also available in subclasses. A plugin can edit these values and the methods will automatically return the values. This is will usually be done once in the constructor of the plugin that subclasses the abstract class. For example, the following class that subclasses *AbstractATDFPlugin* will have the id *http://hello.world/id*, the name *Hello World*, supports the type *http://hello.world/type*, and creates the type *http://good.bye.world/type*.

```
package hello.world;

public class HelloWorldPlugin extends AbstractATDFPlugin {

    private static String ID = "http://hello.world/id";
    private static String HELLO_TYPE = "http://hello.world/type";
    private static String GOOD_BYE_TYPE = "http://good.bye.world/type";

    public HelloWorldPlugin() {
        super(ID, "Hello World");

        operatingTypes.add(new Type(HELLO_TYPE));
        creatingTypes.add(new Type(GOOD_BYE_TYPE));
    }
    ...
}
```

Listing 18 – AbstractATDFPlugin subclass example

The attributes of the *HelloWorldPlugin* class are only for illustration purposes. The constructor of the class calls the constructor of the super class which will store the provided id and name. These values will automatically be returned when the plugins *getName()* and *getID()* functions are called. Furthermore, the plugin creates and adds *Types* to the *operatingTypes* and *creatingTypes* list. These lists are also automatically returned when the respective *getTypesOperatingOn()* and *getTypesCreating()* methods are called. In this example, all instantiation is done in the constructor and most of the methods will not have to be implemented because they are already implemented by the super class. The plugin will only have to implement the *execute()* method.

5.2.2 PluginManager

The plugin manager acts as a bridge between framework and plugins. Calls from the framework to the plugins are directed through this plugin manager. It wraps the calls into an *ISafeRunnable* which is typically used when a plugin needs to call some untrusted code, e.g. code that was contributed by another plugin. This way, a plugin that

crashes will not force the whole framework to crash. An example for such a wrapped method is shown in Listing 19.

```
public List<Node> executePlugin(final IATDFPlugin plugin, final Node node) {
    final ArrayList<Node> modifiedNodes = new ArrayList<Node>();

    ISafeRunnable runnable = new ISafeRunnable() {
        @Override
        public void handleException(Throwable exception) {
            System.out.println("Exception in plugin!");
            exception.printStackTrace();
        }

        @Override
        public void run() throws Exception {
            List<Node> pluginModifiedNodes = plugin.execute(node);

            if(pluginModifiedNodes != null) {
                modifiedNodes.addAll(pluginModifiedNodes);
            }
        }
    };
    SafeRunner.run(runnable);

    return modifiedNodes;
}
```

Listing 19 – Example of ISafeRunnable

Instead of calling the *execute()* method of the plugin directly the framework uses the wrapped method of the plugin manager. The code is self-explanatory. Un-trusted code is called inside the *run()* method. A possible exception can be handled in the *handleException()* method. Every method of the IATDFPlugin interface has a respective wrapped method in the plugin manager.

The plugin manager also has the responsibility to provide the list of discovery plugins to the framework. Basically, a discovery plugin differs not from any other Eclipse plugin that is installed in the Eclipse RCP-based framework. In the list of installed plugins, the plugin manager must find the plugins that implement the IATDFPlugin interface. The list of installed plugins is managed by the framework in an extension registry that can be queried for plugins of a specific type. In Listing 20 the *readPlugins()* method first gets all the configuration elements that have an *Id* of *IATDFPLUGIN_ID* which has a value of *de.kreinjb.gmf.atdframework.manager.plugins*. For every configuration element a *class* executable extension is created. If the created object is an instance of IATDFPlugin then the plugin is a discovery plugin and it is added to the list of plugins that are managed by the plugin manager.

```

Private void readPlugins() {

    IConfigurationElement[] config = Platform.getExtensionRegistry()
        .getConfigurationElementsFor(IATDFPLUGIN_ID);

    try {
        for (IConfigurationElement e : config) {

            final Object o = e.createExecutableExtension("class");

            if (o instanceof IATDFPlugin) {
                plugins.add((IATDFPlugin)o);
            }
        }
    } catch (CoreException ex) {
        System.out.println(ex.getMessage());
    }

}

```

Listing 20 – Detecting plugins that implement the IATDFPlugin interface

The UI of the framework is extended by a view that provides the list of installed plugins and their respective types, illustrated in Figure 26.

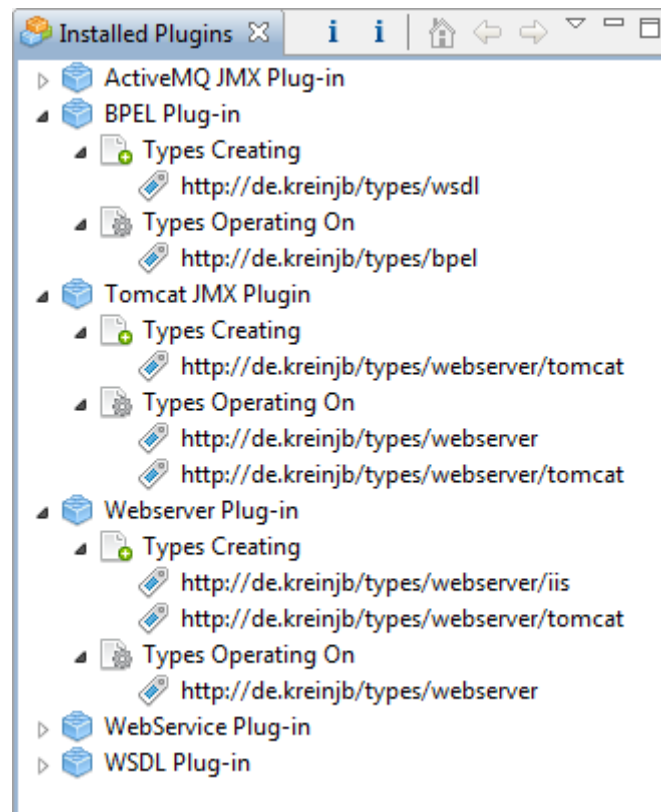


Figure 26 – Installed Plugins view

For each installed plugin it displays the list of types the plugin creates and the list of types it supports, i.e. the types it can operate on. In Figure 26 the *BPEL Plug-in* creates nodes of type *http://de.kreijnb/types/wsdl* and in can operate on nodes that have the type *http://de.kreijnb/types/bpel*.

To create this view, a new sample view is created through the extension wizard which is available in the extension section of the plugin.xml when pressing the *add* button. Table 13 shows the settings that will create the view for the installed plugins.

Property	Value
Java Package Name	de.kreijnb.gmf.atdframework.manager.views
View Class Name	InstalledPlugins
View Name	Installed Plugins
View Category Id	de.kreijnb.gmf.atdframework
View Category Name	ATDFramework
Viewer Type	Tree Viewer

Table 13 – Settings for the installed plugins view

To allow the framework to show this view a new menu item is created in the main menu of the ATDFramework. This is done by adding a *menuContribution* to the *org.eclipse.ui.menus* extension of the manager plugin. The *locationURI* of the *menuContribution* is set to *menu:window* which means that the new menu should be added to the *window* menu of the main menu. Afterwards, a menu element is added to the menuContribution with the label property set to *Show View*. This will create a sub menu under the *window* menu. This sub menu again gets two command elements. The first one with the settings in Table 14 creates a menu item that will open the *Show View* window that is provided by Eclipse. This view lists all installed views and lets the user select the view that should be opened.

Property	Value
CommandId	org.eclipse.ui.views.showView
Label	Others

Table 14 – Properties of the *Others* menu item

The InstalledPlugins view can also be opened directly without opening the Show View window first. This is achieved by adding a second command element with the settings in Table 15, which is basically the same as the one before. But this command is added a parameter element with the settings in Table 16.

Property	Value
CommandId	org.eclipse.ui.views.showView
Label	Installed Plugins

Table 15 – Properties of the *Installed Plugins* menu item

Property	Value
Name	org.eclipse.ui.views.showView.viewId
Label	de.kreijnb.gmf.atdframework.manager.views.InstalledPlugins

Table 16 – Properties of the *Installed Plugins Show View* parameter

Adding this command will lead to the direct opening of the *Installed Plugins* view because the *id* of the view is passed as a parameter to the *Show View* command.

5.2.3 Type System and Type Registry

The type system stores the types and provides a mechanism to check for equality of nodes of the same type. The type registry manages the installation of types into the type system and manages the priority list that is used to decide which plugin is allowed to execute on a node. To give the user a feedback of the current priority list and to allow the user to modify the priority list, the UI is extended with an additional view that shows for each node type the current priority list (Figure 1).

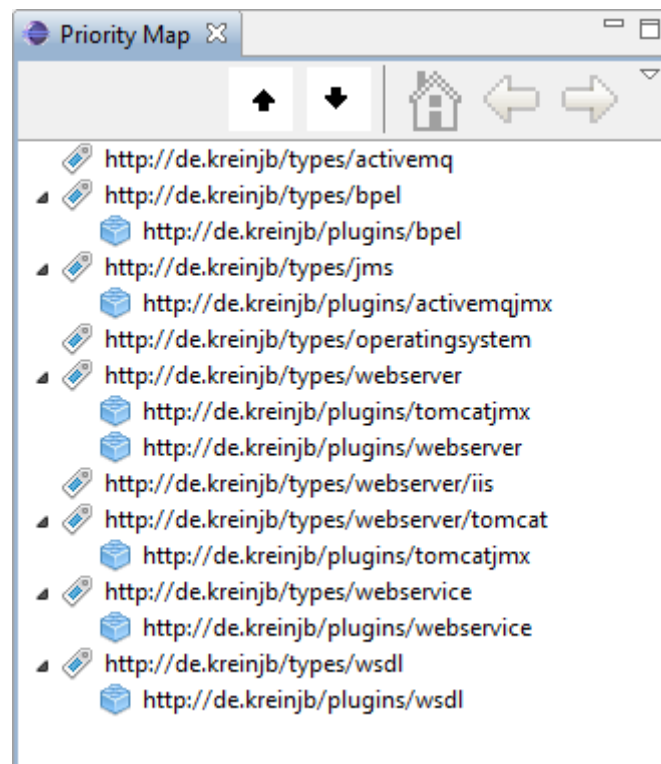


Figure 27 – Priority Map view

The steps for the creation of this view are similar to the ones of the installed plugins view, except of the *view class name* and the *view name* (Table 17).

Property	Value
Java Package Name	de.kreinjb.gmf.atdframework.manager.views
View Class Name	PriorityMap
View Name	Priority Map
View Category Id	de.kreinjb.gmf.atdframework
View Category Name	ATDFramework
Viewer Type	Tree Viewer

Table 17 – Settings for the *Priority Map* view

The type registry manages the priority lists and provides two methods to move plugins up and down in the priority list. These methods are called when the user presses the *up* and *down* buttons in the Priority Map view.

The type system implements two important concepts. One is the mechanism that checks for equality of nodes, the other manages the installation of duplicate type definitions. Equality of nodes is checked by comparing the identifying properties of two nodes. This way, the framework can decide that two nodes are the same event if they do not have the same amount of properties. They only have to match in the identifying properties. These properties are the ones that identify a type and which are stored with the type definition in the type system. Hence the type system knows which properties identify a node of a specific type and only checks whether these properties match.

Another mechanism checks for duplicate type definitions. If two plugins provide a type definition for the same type and these type definitions do not match, the type system adds these definitions to a conflicts list. An exception is the definition of a type with an empty properties list. A plugin that does not specify a list of identifying properties when defining a type expects another plugin to provide this definition.

5.2.4 Scheduler

Besides the implementation of the scheduler concepts, the workbench must be extended with different UI features, e.g. the framework provides two buttons for starting the discovery.

To add the buttons, first the `org.eclipse.ui.command` extension is added to the extension of the `plugin.xml` of the manager plugin. For each of the two buttons is added a command to the extension (Table 18 and Table 19).

Property	Value
Id	de.kreinjb.gmf.atdframework.manager.commands.run
Name	Run
DefaultHandler	de.kreinjb.gmf.atdframework.manager.commands.RunHandler

Table 18 – Run command

Property	Value
Id	de.kreinjb.gmf.atdframework.manager.commands.runstep
Name	RunStep
DefaultHandler	de.kreinjb.gmf.atdframework.manager.commands.RunStepHandler

Table 19 – RunStep command

The *defaultHandler* specifies the class that handles click events, e.g. when the user presses the run button the *RunHandler* class is called. Listing 21 shows the implementation of the *RunHandler* class.

```
package de.kreinjb.gmf.atdframework.manager.commands;

import org.eclipse.core.commands.AbstractHandler;
import org.eclipse.core.commands.ExecutionEvent;
import org.eclipse.core.commands.ExecutionException;

import de.kreinjb.gmf.atdframework.manager.Scheduler;

public class RunHandler extends AbstractHandler {

    @Override
    public Object execute(ExecutionEvent event) throws ExecutionException {
        Scheduler.getInstance().run(false);
        return null;
    }

}
```

Listing 21 – RunHandler

The *execute* method is called when the *Run* button is pressed. It then calls the *run* method of the scheduler with a parameter of *false*. This indicates that the run method should not pause after each iteration step, but run till there is nothing to discover anymore (i.e. there is no plugin that could find any additional discovery information). If the

method is called with a value of *true* as parameter, the method will pause after each iteration step, forcing the user to press the button again to run the next iteration. This is done by the *RunStepHandler*, which is similar to the *RunHandler*, except it uses *true* as parameter.

For the adding of the buttons to the framework a *menuContribution* is added to the *org.eclipse.ui.menus* extension of the manager plugin with the *locationURI* property set to *toolbar:org.eclipse.ui.main.toolbar*. This will place the buttons in the main toolbar of the ATDFramework. Beneath the *menuContribution* is added a *toolbar* element with an *id* of *de.kreinjb.gmf.atdframework.manager.toolbar*. For each button of the two run buttons are added command elements that contain the *commandId* of the previously created run handlers. This will connect the UI buttons to the handlers.

The scheduler itself implements the concepts described in 3.4. Besides the *run* method that is called to manage the discovery, it contains functions to calculate the next scheduled plugin for each node and an *execute* method that executes the scheduling list. Listing 22 shows the code that calculates the standard scheduling of plugins in the *getScheduledPlugin* method of the scheduler.

```
...
LinkedList<String> candidateList = new LinkedList<String>(priorityList);

Collections.sort(entries, new Comparator<Entry>() {
    @Override
    public int compare(Entry e1, Entry e2) {
        return e2.getTimestamp().compareTo(e1.getTimestamp());
    }
});

for(Entry entry : entries) {
    if(candidateList == null || candidateList.isEmpty()) {
        return null;
    }

    candidateList.remove(entry.getPlugin());

    if(entry.isModified()) {
        return (candidateList.isEmpty() ? null : candidateList.getFirst());
    }
}

return (candidateList.isEmpty() ? null : candidateList.getFirst());
...
```

Listing 22 – Part of the scheduler

To determine the next scheduled plugin for a node a candidate list is calculated. At the beginning, the candidate list is initialized with the current priority list of the node type. Plugins are removed from this list based on their order in the history list that is represented in the listing above by the variable *entries*. To guarantee the order of the entries the list is first sorted according to the timestamps of the entries. Starting from the last executed plugin in the history, the scheduler removes the plugin from the candidate list, because it was the last one that executed on the node. If the plugin was a modifying plugin (e.g. because it added some information that enables another plugin to execute), the algorithm stops and takes the top element of the candidate list, because this is the plugin with the highest priority. If the plugin was not a modifying plugin (e.g. the plugin executed on the node but it could not discover anything), it is removed from the candidate list and the algorithm continues with the next plugin of the history list.

5.2.5 Properties

Eclipse RCP-based applications can use the *properties* view in Eclipse to display information. An EMF or GMF editor already makes extensive use of this view, e.g. to display the current appearance of nodes. Furthermore, all attributes of an element in the model can automatically be displayed and edited in the properties view. For example, each *node* element has a *name* attribute. This attribute will be displayed in the properties view when a node is selected in the editor.

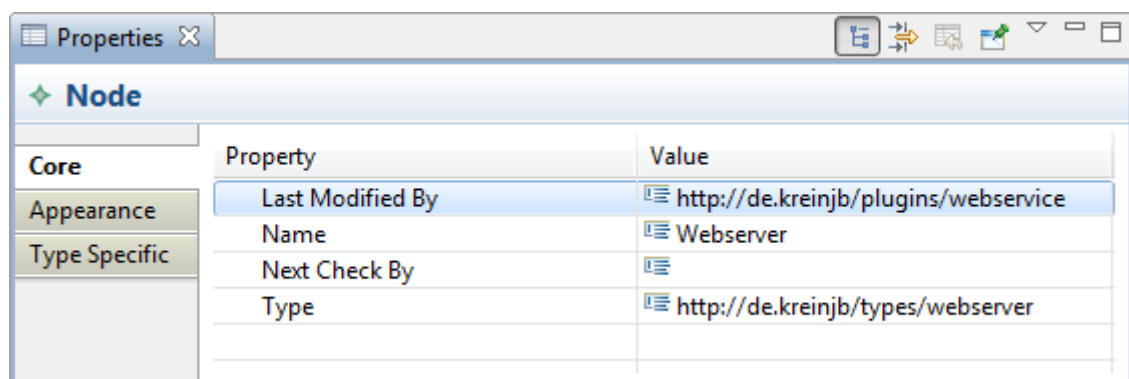
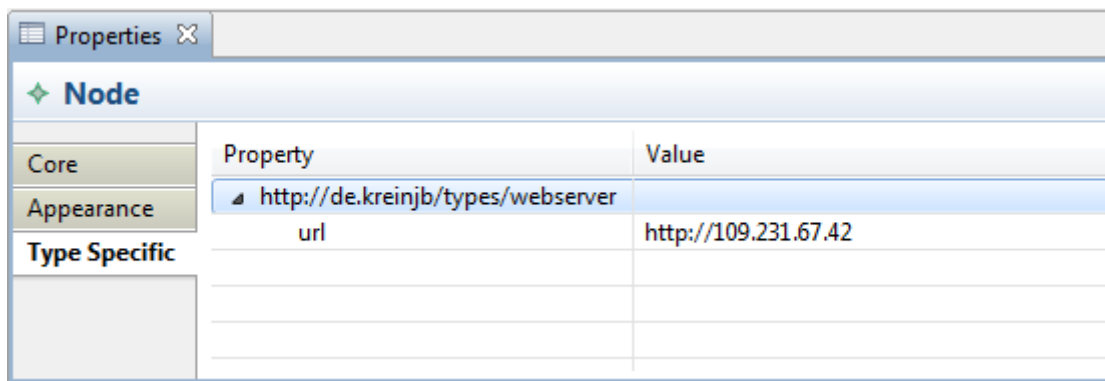


Figure 28 – Properties view

Of course, attributes can be hidden from this view and also selected as *read only* so that these attributes cannot be modified graphically by the user.

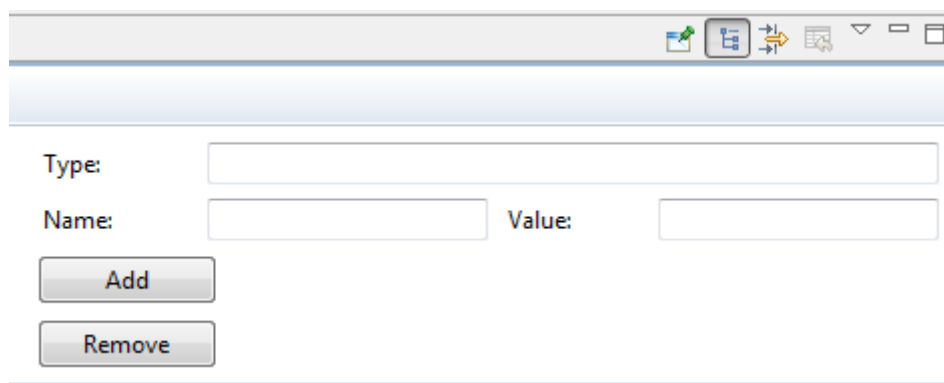
One problem during the implementation of the ATDFramework was the displaying of the type-specific properties list of a node. It varies in size depending on the current amount of defined properties and hence cannot be determined at build time. The auto-generated code for the displaying of lists that vary in size was not representative and not editable. For this special purpose the properties view was extended with an additional tab that displays the type specific properties (Figure 29 and Figure 30).



The screenshot shows a 'Properties' window with a tabbed interface. The 'Node' tab is active, showing a table of properties. The table has two columns: 'Property' and 'Value'. The first row is highlighted, showing the property 'http://de.kreijnb/types/webserver' with the value 'http://109.231.67.42'. The table is grouped under 'Type Specific'.

Property	Value
http://de.kreijnb/types/webserver	http://109.231.67.42
url	

Figure 29 – Type-specific properties left side



The screenshot shows the right side of the 'Properties' window. It contains a 'Type:' label followed by a text input field. Below this is a 'Name:' label followed by a text input field, and a 'Value:' label followed by a text input field. There are two buttons: 'Add' and 'Remove'.

Figure 30 – Type-specific properties right side

Each property of a node is displayed in this view. The type determines the grouping of properties, e.g. all properties with type `http://de.kreijnb/types/webserver` are grouped and displayed together. The right side of this view allows for adding and removing of properties. The two extensions in Listing 23 and Listing 24 were added to the `plugin.xml` of the GMF plugin, to add the tab to the properties view.

```
<extension point="org.eclipse.ui.views.properties.tabbed.propertyTabs"
  id="proptabs">
  <?gmfgen generated="true"?>
  <propertyTabs contributorId="de.kreijnb.gmf.atdframework.rcp">
    ...
    <propertyTab
      category="typeSpecific"
      afterTab="property.tab.domain"
      id="property.tab.typeSpecific"
      label="%tab.typeSpecific"/>
  </propertyTabs>
</extension>
```

Listing 23 – Property tab extension

```

<extension point="org.eclipse.ui.views.properties.tabbed.propertySections"
  id="propsections">
  <?gmfgen generated="true"?>
  <propertySections contributorId="de.kreijnb.gmf.atdframework.rcp">
    ...
    <propertySection
      id="property.section.typeSpecificSection"
      tab="property.tab.typeSpecific"
      class="de.kreijnb.gmf.atdframework.rcp.sheet.
                                   AtdframeworkTypeSpecificPropertySection">
      <input type="org.eclipse.gmf.runtime.notation.View"/>
      <input type="org.eclipse.gef.EditPart"/>
    </propertySection>
  </propertySections>
</extension>

```

Listing 24 – Property section extension

The extension to *org.eclipse.ui.views.properties.tabbed.propertyTabs* in Listing 23 added an additional tab to the properties view, with the specification of the name, id, and the position to place the tab in the properties view. Listing 24 showed the extension to *org.eclipse.ui.views.properties.tabbed.propertySections* which adds a property section to the tab. A tab can contain multiple sections. The important part is the *class* attribute in the *propertySection* element. It specifies the class that implements the property section. This class then creates the UI of the view, e.g. the text fields and the buttons.

5.3 Plugins

The plugins are bundled in a separate feature apart from the framework. They are implemented prototypically to give an idea of what could be discovered from the different types of artifacts. The feature project is named *de.kreijnb.gmf.atdframework.pluginsfeature*. It contains all the plugins that are developed. For the prototypical implementation this is only a single plugin named *de.kreijnb.gmf.atdframework.manager.plugincontribution* to keep the project structure simple. The specified extension point of the ATDFramework (see Section 5.2.1) must be extended by every plugin.

```

<extension point="de.kreijnb.gmf.atdframework.manager.plugins">
  <plugin
    class="de.kreijnb.gmf.atdframework.manager.plugincontribution.BPELPlugin">
  </plugin>
</extension>

```

Listing 25 – Extension provided by *BPELPlugin*

Listing 25 shows how the *BPELPlugin* provides an extension by defining a *plugin* element with the *class* attribute specifying the implementation class.

In the following for each developed plugin the discovery procedure and properties of the plugin are presented. The *operating types* property denotes the types of nodes the plugin can operate on and *creating types* which types of nodes may be created by the plugin. A square bracket containing properties at the end of a type means the plugin needs this property to work correctly or if it creates a node of a type it will add this property to the node. For example, *http://example.com/types/exampletype [exampleproperty]* means a plugin creates or operates on nodes of type *http://example.com/types/exampletype* that have a property of *exampleproperty*. This is only for illustration here and not the representation used in the code.

Note: The plugins only implement the concepts described earlier. See Section 3.5 for further information.

5.3.1 BPEL Plugin

Property	Value
Name	BPELPlugin
Id	http://de.kreinjb/plugins/bpel
Operating Types	http://de.kreinjb/types/bpel [location]
Creating Types	http://de.kreinjb/types/wsdl [location]

Table 20 – Properties of BPEL plugin

A BPEL node must contain the location property which is used to find and then parse the document. The plugin uses an external library called *Xalan*¹² to parse the XML document. The following listing shows how the locations of WSDL documents can be retrieved from a BPEL file using Xalan.

```
String path = "/bpel:process//bpel:import[@importType=" +
    "\"http://schemas.xmlsoap.org/wsdl/\"]/@location";
XPath.evaluate(path, inputSource, XPathConstants.NODESET);
```

Listing 26 – XPath statement to retrieve locations of WSDL documents

The plugin creates for each found WSDL document a node that has a location property which contains the actual location of the WSDL document.

¹² <http://xml.apache.org/xalan-j/>

5.3.2 WSDL Plugin

Property	Value
Name	WSDLPlugin
Id	http://de.kreinjb/plugins/wsdl
Operating Types	http://de.kreinjb/types/wsdl [location]
Creating Types	http://de.kreinjb/types/webservice [address]

Table 21 – Properties of WSDL plugin

As the WSDL document is an XML document the WSDL plugin also uses Xalan to parse the document. The plugin first looks for *port* elements in the *service* definition of the WSDL file. The port element usually contains an address element, e.g. if the port uses a SOAP binding the port element will contain a *<soap:address>* element with a *location* attribute. For each found location the plugin creates a Web Service node that contains a location property which points to the Web Service.

5.3.3 Web Service Plugin

Property	Value
Name	WebServicePlugin
Id	http://de.kreinjb/plugins/webservice
Operating Types	http://de.kreinjb/types/webservice [address]
Creating Types	http://de.kreinjb/types/webserver [url]

Table 22 – Properties of Web Service plugin

The Web Service plugin just takes the address of the Web Service and extracts the host and port parts of the address. For example, *http://example.com:8080* is extracted from *http://example.com:8080/demo/demoWS*. For each of these extracted URLs is created a web server sub node. A web server sometimes provides an information page under this URL.

5.3.4 Web Server Plugin

Property	Value
Name	WebServerPlugin
Id	http://de.kreinjb/plugins/webserver
Operating Types	http://de.kreinjb/types/webserver [url]
Creating Types	http://de.kreinjb/types/webserver/tomcat [header-server] http://de.kreinjb/types/webserver/iis [header-server] http://de.kreinjb/types/operatingsystem [name]

Table 23 – Properties of web server plugin

The web server plugin uses the URL to send an HTTP request to the server. The HTTP response usually contains a server header that sometimes contains the name and version of the server. The current version supports Apache Tomcat and Microsoft IIS. A special case is the discovery of an IIS because one can assume that it is running on Windows. The plugin will then automatically create an operating system node with a name of Windows.

5.3.5 Tomcat JMX Plugin

Property	Value
Name	TomcatJMXPlugin
Id	http://de.kreinjb/plugins/tomcatjmx
Operating Types	http://de.kreinjb/types/tomcat [url, username, password]
Creating Types	http://de.kreinjb/types/operatingsystem [name, version]

Table 24 – Properties of Tomcat JMX plugin

The Tomcat JMX plugin uses JMX to query a Tomcat server. Of course, Tomcat must be enabled on the server. Furthermore, the plugin requires the availability of a URL property in the Tomcat node and an optional username and password for authentication. The plugin then connects to a so-called *MBean Server* where JMX enabled programs are registered as MBeans. MBeans are identified by an *objectName* which consists of a domain name and a list of properties. For the Tomcat server this *objectName* is *Catalina:type=server*. This object can then be queried for attributes, e.g. the tomcat plugin reads the *serverInfo* attribute. This information is stored as a property in the node and the node type is changed from web server type to a Tomcat type. Additionally, the MBean server provides information about the operating system. The

objectName to use is *java.lang:type=OperatingSystem*. This object contains the two attributes *name* and *version* which contain the name and version of the operating system. If this information is available, the plugin will also create a sub node under the Tomcat node, indicating that the Tomcat is hosted on a specific operating system.

5.3.6 ActiveMQ JMX Plugin

Property	Value
Name	ActiveMQJMXPlugin
Id	http://de.kreinjb/plugins/activemqjmx
Operating Types	http://de.kreinjb/types/jms [url, username, password]
Creating Types	http://de.kreinjb/types/activemq [brokername, brokerversion, brokerid] http://de.kreinjb/types/operatingsystem [name, version]

Table 25 – Properties of ActiveMQ JMX plugin

The ActiveMQ JMX plugin works pretty much the same as the Tomcat JMX plugin. In contrast to the Tomcat plugin, it queries the MBean of ActiveMQ which has an objectName of *org.apache.activemq:BrokerName=*,Type=Broker*. The three attributes *brokerName*, *brokerVersion*, and *brokerId* of the object provide the name, version, and id of the broker. If this information is available the plugin creates a sub node with these properties. Like the Tomcat JMX plugin, it also tries to identify the operating system using the same mechanism and adds an operating system node if possible.

6 Summary and Outlook

Software applications usually grow and evolve over time, and so do their architectures and the topology of the actual deployment. However, knowledge of the application's components and their relations is crucial for enterprise architecture management tasks like migration and optimization. With the rise of new paradigms like Service-Oriented Architecture and technologies, like cloud computing, recent approaches tried to find ways for the external discovery of application topologies. The focus of this diploma thesis are to research ways for external application topology discovery, i.e., analyzing which external, network accessible resources provide information to get a detailed picture of the application topology. The goal was to develop a plugin-based framework and a set of prototypical discovery plugins. The developed framework is called ATDFramework and manages the discovery by scheduling the different installed plugins. Other developers can contribute their discovery code to the framework by developing additional plugins.

The discovery framework is built with OSGi, more precise, an Eclipse RCP-based application which uses Equinox as its OSGi framework. The framework is supplemented with graphical editors based on the Eclipse Modeling Framework and Graphical Modeling Framework.

A special focus of the framework was to keep the plugin development process as simple as possible. The plugin developer is supported by a simple data model consisting of nodes, connections, properties, and a simple type system that is based on namespace definitions. The types are provided by the plugins themselves and the type system is build up implicitly when installing the plugins. Furthermore, plugins have a simple code structure that only requires minimal configuration effort and allows developers to concentrate on the actual discovery code.

Along with the framework comes a set of prototypical discovery plugins. Starting point of the discovery in this work is a BPEL service composition. The set of plugins developed during the course of this thesis are a BPEL-, WSDL-, Web service-, web server-, Tomcat-, and ActiveMQ-plugin.

The discovered topology is stored in a file for further processing by other programs or to resume discovery after changes in the topology. One use case for the discovered topology is the migration of SOA-based applications to the cloud, e.g. programs can use the discovered topology to adapt and steer the migration process.

Of course, the current approach has some room for improvement as well. As the framework is only tested with the set of prototypical plugins, some bugs may remain undetected and only appear in different plugin scenarios. The UI has also some room for improvement and some ideas have not been implemented because of limited development time. For example, the framework could use a status bar to present the cur-

rent discovery status to the user and the overall type system with all dependencies between types could be visualized in an extra view. Furthermore, conflicts between plugins are resolved but not presented to the user in an extra view, e.g. the Eclipse problems view would be predestinated for this task.

The concept of namespace definitions for types makes the type system easy and hence also simplifies the development of plugins. But it would be worth to check the concept of OSGi-based types. This means that types themselves are Eclipse plugins which can be installed to and un-installed from the framework. The plugins would still provide these types but the dependency mechanism that is introduced by OSGi could be used to handle dependency resolution from plugins to types, or types to types. This would ease the process of checking for identity of nodes. For example, properties could be defined as attributes of a Java class and the class itself could provide the mechanism to identify a node or to check for equality of nodes. But this approach could require a modification of the data model and a possible remodeling of the EMF- and GMF-related models. It must also be deliberated about whether the current, simple namespace-based type system – which allows for easy creation of types and the specification of dependencies to these types – can be integrated with this approach. Because of the limited time for the implementation of the framework, this approach was not further pursued.

A screenshot of the final application is shown in Figure 31.

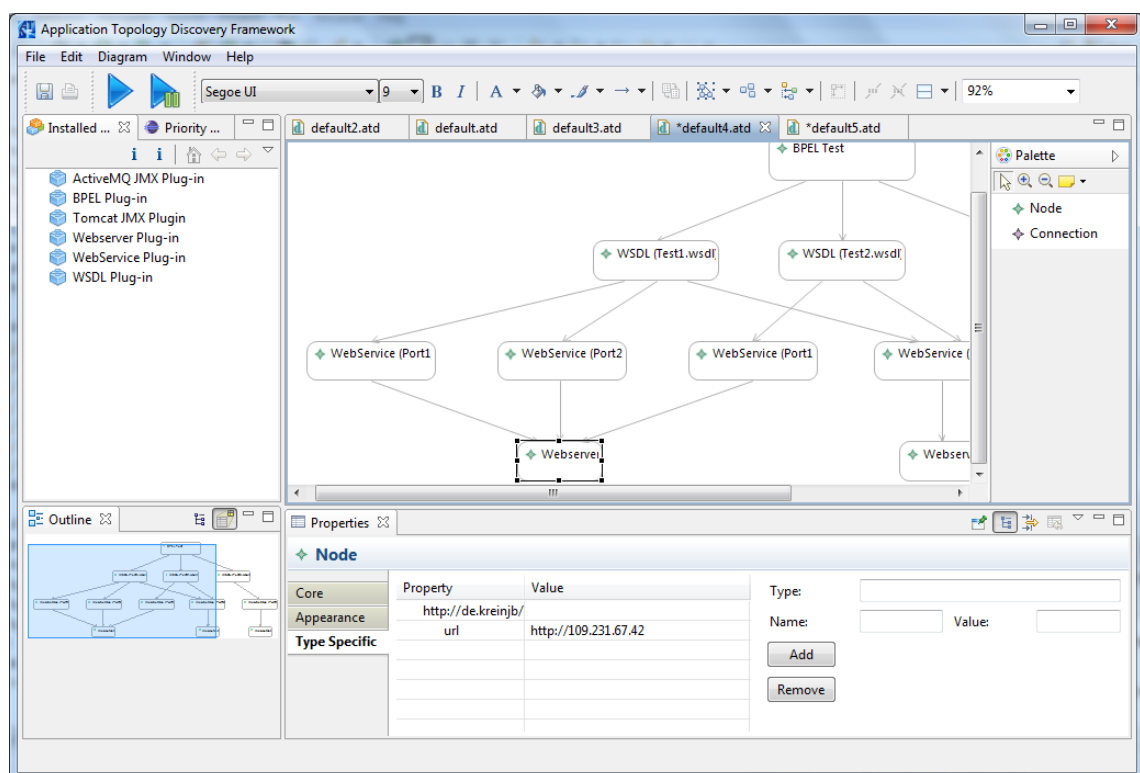


Figure 31 – Screenshot of the final application

Appendix A Framework Manual

This chapter should give the reader a short introduction on how the framework is used to discover a topology (A-4), including the steps to install the framework (A-1) and plugins (A-2) and how to create new plugins (A-3).

A-1 Framework Installation

To install the framework the user only must unzip the binary and copy it to the desired location. For Windows the zipped file will contain a folder called *atdframework* and in that folder an executable called *atdframework.exe*.

For other operating system or if the framework should be build from source code, it is advised to download the *Plugin Development Environment (PDE)*¹³ version of Eclipse and to install all GMF dependencies¹⁴. Once the projects are setup in Eclipse one can use the product export wizard (see Section 5.1.7) of the *atdframework.product* file in the *de.kreinjb.gmf.atdframework* project to create executables.

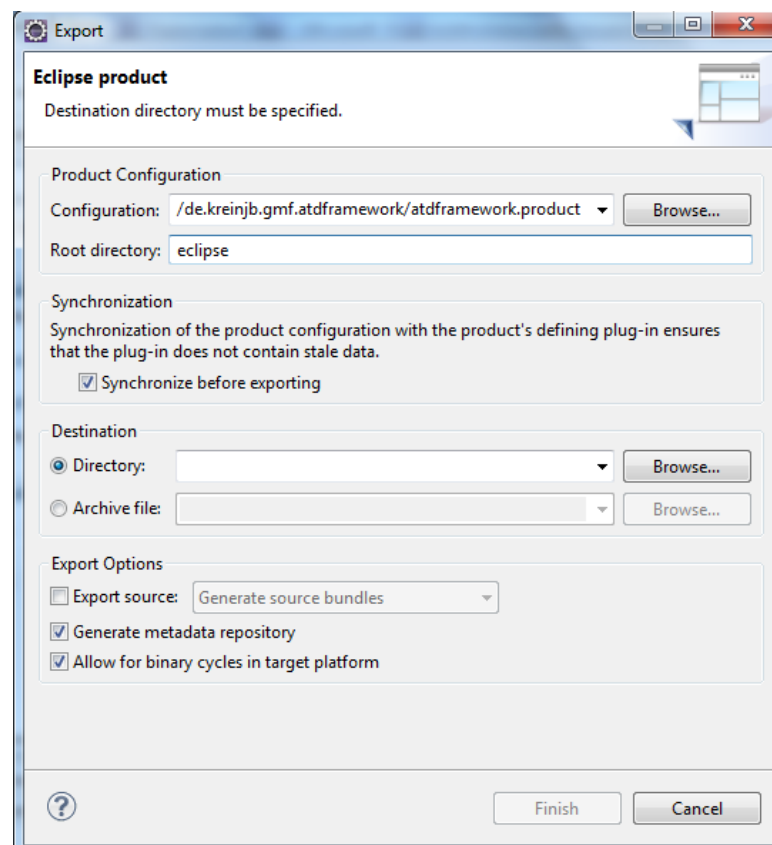


Figure 32 – Product export wizard

Note: Product export may require administrative rights.

¹³ <http://download.eclipse.org/eclipse/downloads/>

¹⁴ <http://www.eclipse.org/modeling/gmp/>

A-2 Plugin Installation

The easiest way to install new plugins is to use the *Update Manager* of the ATDFramework located in the menu under *Help -> Install New Software*. A plugin developer will most likely want to install plugins from the local file system. Anyhow, the Update Manager allows the user to install plugins from a remote repository, as well as from a local repository. The Update Manager has to be pointed to the directory that contains the plugins and features. In general, the update mechanism of the ATDFramework does not differ from the update mechanism of the Eclipse IDE.

Another way to install plugins is to copy the exported plugin to the Eclipse plugins folder, more specific, the developed feature will contain two folders named *features* and *plugins*. The contents of these folders have to be copied into the ATDFramework folders with the same name, found in the base folder of the framework that also contains the executable.

A-3 Plugin Development

The basic steps for plugin development for the ATDFramework match the development of plugins for the Eclipse IDE, e.g. with the creation of a plugin project, a feature project, and the export of the developed feature. But there are few things to consider that are different, or rather specific to the ATDFramework, namely providing an extension to the framework-specific extension point. The following leads through the development of a simple *hello world* plugin in a tutorial like fashion.

Plugin development requires specific dependencies that must be available in the Eclipse IDE. The feature that contains the required plugins is the *ATDFrameworkFeature* that is available in the repository folder of the zipped file. It must be installed to the Eclipse IDE using the Eclipse Update Manager.

Once the IDE is set up, create a new plugin project (File -> New -> Plug-in Project) with a project name of *HelloWorldPlugin*. Keep the suggested settings and click finish. In the overview section of the generated *MANIFEST.MF* select *This plug-in is a singleton*. Switch to the dependencies tab and add *de.kreijnb.gmf.atdframework* dependency and the *de.kreijnb.gmf.atdframework.manager* dependency. Open the *extension* tab and click the add button. Select *de.kreijnb.gmf.atdframework.manager.plugins* from the extension points list and click finish. Switch to the *plugin.xml* tab and modify the content to look like the content in Listing 27.

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin>
  <extension point="de.kreijnb.gmf.atdframework.manager.plugins">
    <plugin class="helloworldplugin.HelloWorldPlugin"/>
  </extension>
</plugin>
```

Listing 27 – HelloWorldPlugin plugin.xml

Note: The class attribute must contain the name of the class that will provide the implementation including the package name.

The project should already contain a *helloworldplugin* package with an *Activator* class. Add a new Java class to the package with the name specified in the plugin.xml, in this case the class *HelloWorldPlugin* in the package *helloworldplugin*. Set a super class of *AbstractATDFPlugin* from the *de.kreinjb.gmf.atdframework.manager.plugins* package in the class creation dialog. Let the wizard create constructors from the super class and click finish. The generated class should look like the one in Listing 28.

```
package helloworldplugin;

import java.util.List;

import de.kreinjb.gmf.atdframework.Node;
import de.kreinjb.gmf.atdframework.manager.plugins.AbstractATDFPlugin;

public class HelloWorldPlugin extends AbstractATDFPlugin {

    public HelloWorldPlugin(String id, String name) {
        super(id, name);
        // TODO Auto-generated constructor stub
    }

    @Override
    public List<Node> execute(Node arg0) {
        // TODO Auto-generated method stub
        return null;
    }

}
```

Listing 28 – Generated HelloWorldPlugin.java class

The generated code will have a constructor and an *execute* method. Change the content to the code in Listing 29.

Note: The generated code contains a constructor with two parameters. These are removed so that the constructor is the standard constructor without any parameters.


```
package helloworldplugin;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

import de.kreijnb.gmf.atdframework.Node;
import de.kreijnb.gmf.atdframework.manager.plugins.AbstractATDFPlugin;
import de.kreijnb.gmf.atdframework.manager.plugins.PluginUtil;
import de.kreijnb.gmf.atdframework.manager.type.ConnectionRequest;
import de.kreijnb.gmf.atdframework.manager.type.NodeRequest;
import de.kreijnb.gmf.atdframework.manager.type.Type;

public class HelloWorldPlugin extends AbstractATDFPlugin {

    public HelloWorldPlugin() {
        super("http://helloWorld.com/id", "Hello World Plugin");

        operatingTypes.add(
            new Type("http://helloWorld.com/type", new String[]{"name"}));
        creatingTypes.add(
            new Type("http://goodByeWorld.com/type", new String[]{"greeting"}));
    }

    @Override
    public List<Node> execute(Node node) {

        String name = node.getProperties().getProperty().get(0).getValue();

        Map<String, String> properties = new HashMap<String, String>();
        properties.put("greeting", "Hi " + name + "!");

        return PluginUtil.connect(
            node,
            new NodeRequest(
                "Good Bye Node", "http://goodByeWorld.com/type", properties),
            new ConnectionRequest("Says Hi"),
            getID());
    }
}
```

Listing 29 – Modified HelloWorldPlugin.java class

The code presents the basic functionalities provided by the framework. All the properties of the plugin should be specified in the constructor, e.g. the id and the name of the plugin, as well as the types the plugin supports. This is achieved by calling the constructor of the super class with the id and the name of the plugin. The abstract class also contains two lists, *operatingTypes* and *creatingTypes*. All the supported types should be added to these lists, e.g. the code declares that the plugin can operate on nodes that have a type of *http://helloWorld/type* and will create nodes that have a type of *http://goodByeWorld/type*. These declarations also define the identifying properties that the types will have, e.g. the nodes of type *http://helloWorld/type* can be uniquely identified through the *name* property while nodes of the *http://goodByeWorld/type* type will be identified through the *greeting* property. With these settings, the plugin is ready to be used by the framework, i.e. the framework can schedule the plugin for execution if the discovered topology contains a node of type *http://helloWorld/type*.

When the plugin is scheduled for execution the framework calls the *execute* method of the plugin and provides as a parameter the node the plugin should operate on. The example above assumes that the provided node will have exactly one property and does not check whether it is available or what type or name the property is of. In a real world example, a plugin would check for specific properties of a certain type and name and only proceed if these properties are available. For illustration purposes and to keep the example simple the code assumes the node has one property – more specific, a *name* property. This name is extracted and a greeting message is created that says “Hi” with the name added to the greeting.

Afterwards, a node is created that will contain a greeting property with the greeting message as the value of the property and a connection is created between the two nodes. For this task, the framework provides a utility class with a *connect* method. The connect method acts not only as a creation function for the connection but also as a creation function for the node. What the user does when he calls the function is sending a request to connect to a specific node that has certain properties that the user specified in a *NodeRequest*. For example, this *NodeRequest* contains a name, type, and list of properties that the node should contain. If the framework finds such a node in the current discovered topology, it creates a connection to this node. The check for equality of two nodes is based only on the identifying properties. If the *NodeRequest* contained more properties and the framework found a match for the requested node, then these properties will be added to the found node. Connections can also have a name which is specified in the *ConnectionRequest*, e.g. in the code above a connection with a name of “Says Hi” is created between two nodes.

If the framework does not find a node with the given properties – or more specific, a node where the identifying properties do not match – it creates a new node that contains the properties of the *NodeRequest*. This makes the creation of new nodes and connections very easy because the developer does not have to care whether to create a node or to connect to an already existing node. The framework automatically detects nodes and either creates new ones or connects to already existing ones.

To test the created plugin, create a new feature project (File -> New -> Feature Project) with a name of HelloWorldFeature. Add the previously created plugin to the referenced plugins that are contained in the feature and use the *Export Wizard* in the overview tab of the *feature.xml* to export the feature.

A-4 Initiate Discovery

Start the framework and install the feature using the Update Manager. Create a new project, drag a new node from the palette to the diagram and name it *Hello World Node*. In the properties view set the type of the node to *http://helloWorld.com/type*. Afterwards, switch to the type specific tab in the properties view and add a name property with some value, e.g. set the type to *http://helloWorld.com/type*, the name to *name* and the value to *John Doe*. Click the add button and refresh the properties view by selecting the node on the diagram. Finally, start the discovery by clicking either the *Run* button or the *RunStep* button. The result should look like Figure 33.

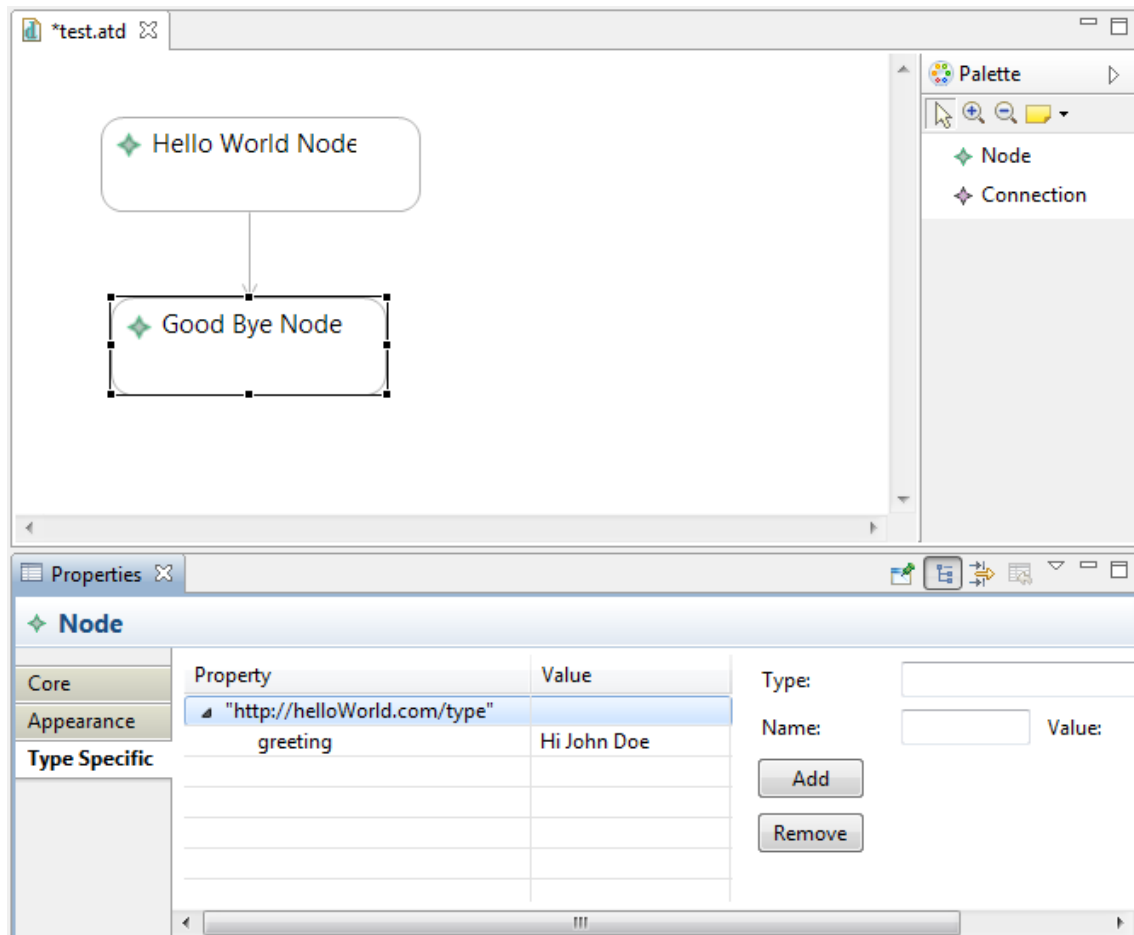


Figure 33 – Discovery of the Hello World Example

References

- [1]. **W3C**. Web Services Description Language (WSDL) 1.1. [Online] March 15, 2001. <http://www.w3.org/TR/wsdl>.
- [2]. **OASIS**. Web Services Business Process Execution Language Version 2.0. [Online] April 11, 2007. <http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html>.
- [3]. **Stephane Ducasse, Damien Pollet**. *Software Architecture Reconstruction: a Process-Oriented Taxonomy*. 2009.
- [4]. ArchJava. [Online] <http://archjava.fluid.cs.cmu.edu/>.
- [5]. **Garlan, David**. *Software architecture: a roadmap*. 2000.
- [6]. **Kamran Sartipi, Kostas Kontogiannis**. *On Modeling Software Architecture Recovery as Graph Matching*. Canada : School of Computer Science and Dept. of Electrical & Computer Engineering Waterloo, 2003.
- [7]. **Andreas Kind, Dieter Gantenbein, Hiroaki Etoh**. *Relationship Discovery with NetFlow to Enable Business-Driven IT Management*. s.l. : In Proceedings of Business-Driven IT Manageent (BDIM'06), pages 63–70, 2006.
- [8]. **Cisco**. IOS NetFlow. [Online] October 2007. http://www.cisco.com/en/US/prod/collateral/iosswrel/ps6537/ps6555/ps6601/prod_white_paper0900aecd80406232.html.
- [9]. **Xu Chen, Ming Zhang, Z.Morley Mao, Paramvir Bahl**. *Automating Network Application Dependency Discovery: Experiences, Limitations, and New Solutions*. s.l. : OSDI USENIX Association, p. 117-130, 2008.
- [10]. **Dieter Gantenbein, Luca Deri**. *Categorizing Computing Assets According to Communication Patterns*. 2002.
- [11]. **Sergey Brin, Lawrence Page**. *The Anatomy of a Large-Scale Hypertextual Web Search Engine*. s.l. : In Proceedings of the 7th World-Wide Web Conference, 1998.
- [12]. **Onn Brandman, Junghoo Cho, Hector Garcia-Molina, Narayanan Shivakumar**. *Crawler-Friendly Web Servers*. s.l. : In Proceedings of the Workshop on Performance and Architecture of Web Servers , 2000.
- [13]. **Junghoo Cho, Hector Garcia-Molina**. *The Evolution of the Web and Implications for an Incremental Crawler*. 1999.
- [14]. **ACM**. Service oriented architecture (SOA) a new paradigm to implement dynamic e-business solutions. [Online] August 2006. <http://ubiquity.acm.org/article.cfm?id=1159403>.
- [15]. **W3C**. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). [Online] April 27, 2007. <http://www.w3.org/TR/soap12-part1/>.

- [16]. **IBM**. Web Services Flow Language Version 1.0 (WSFL 1.0). [Online] 2001. <http://xml.coverpages.org/wsfl.html>.
- [17]. **Microsoft**. XML Business Process Language (XLANG). [Online] 2001. <http://xml.coverpages.org/xlang.html>.
- [18]. **IBM, Microsoft**. Business Process Execution Language for Web Services Version 1.1. [Online] 2003. <http://public.dhe.ibm.com/software/dw/specs/ws-bpel/ws-bpel.pdf>.
- [19]. **OASIS**. WS-BPEL Extension for People. [Online] 2007. <http://www.oasis-open.org/committees/bpel4people/>.
- [20]. **National Institute of Standards and Technology**. The NIST Definition of Cloud Computing. [Online] September 2011. <http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf>.
- [21]. **Deutscher Bundestag**. Aktueller Begriff: Cloud Computing. [Online] 2010. http://www.bundestag.de/dokumente/analysen/2010/cloud_computing.pdf.
- [22]. **Oracle**. Java Management Extensions (JMX) Technology. [Online] <http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html>.
- [23]. **Shah, Saumil**. An Introduction to HTTP Fingerprinting. [Online] 5 19, 2004. http://net-square.com/httpprint/httpprint_paper.html.
- [24]. **Sun Microsystems, Inc.** Java Servlet Specification, Version 3.0. [Online] December 2009. http://download.oracle.com/otn-pub/jcp/servlet-3.0-fr-eval-oth-JSpec/servlet-3_0-final-spec.pdf.
- [25]. —. JavaServer Pages Specification Version 2.1. [Online] May 8, 2006. http://download.oracle.com/otn-pub/jcp/jsp-2.1-fr-eval-spec-oth-JSpec/jsp-2_1-fr-spec.pdf.
- [26]. **Apache Software Foundation**. Coyote HTTP/1.1 Connector. [Online] 2009. <http://tomcat.apache.org/tomcat-4.1-doc/config/coyote.html>.
- [27]. **Lyon, Gordon**. Nmap. [Online] <http://nmap.org/>.
- [28]. **OSGi Alliance**. Open Services Gateway initiative (OSGi). [Online] <http://www.osgi.org>.
- [29]. **Krein, Jakob**. *Web-based Application Integration: Advanced Business Process Monitoring in WSO2 Carbon*. Stuttgart : Institut für Architektur von Anwendungssystemen, 2011.
- [30]. **Eclipse Foundation**. Rich Client Platform. [Online] 2012. http://wiki.eclipse.org/index.php/Rich_Client_Platform.
- [31]. **Ebert, Ralf**. Eclipse RCP. [Online] August 2011. http://www.ralfebert.de/eclipse_rcp/EclipseRCP.pdf.
- [32]. **Eclipse Foundation**. Eclipse Modeling EMF. [Online] 2010. <http://www.eclipse.org/modeling/emf/>.

- [33]. —. EMF/FAQ. [Online] 2011. <http://wiki.eclipse.org/EMF/FAQ>.
- [34]. **Plante, Frederic**. Introducing the GMF Runtime. [Online] January 16, 2006. <http://www.eclipse.org/articles/Article-Introducing-GMF/article.html>.
- [35]. **Eclipse Foundation**. Eclipse Editing Framework. [Online] <http://www.eclipse.org/gef/>.
- [36]. **Chris Aniszczyk**. Learn Eclipse GMF in 15 minutes. [Online] September 12, 2006. <http://www.ibm.com/developerworks/opensource/library/os-ecl-gmf/>.
- [37]. **Eclipse Foundation**. GMF Tutorial. [Online] http://wiki.eclipse.org/Graphical_Modeling_Framework/Tutorial#Get_started.
- [38]. **EMC**. Ionix. [Online] <http://www.emc.com/products/family/ionix-family.htm>.
- [39]. **IBM**. Tivoli. [Online] <http://www.ibm.com/software/tivoli/>.
- [40]. **Microsoft**. System Center Operations Manager. [Online] <http://www.microsoft.com/en-us/server-cloud/system-center/operations-manager.aspx>.
- [41]. **HP**. Network Management Center. [Online] <http://www8.hp.com/us/en/software/software-solution.html?compURI=tcm:245-936973>.

All links have been last followed on April 26, 2012.

Declaration

All the work contained within this thesis, except where otherwise acknowledged, was solely the effort of the author. At no stage was any collaboration entered into with any other party.

Stuttgart, on April 26, 2012

Jakob Krein