FAKULTÄT FÜR !NFORMATIK

Tool Support for Semantic-Model-Driven Systems Integration

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Wirtschaftsinformatik

ausgeführt von

Christian Astl
Matrikelnummer 0226577

am:
Institut für Softwaretechnik und Interaktive Systeme [E188]

Betreuung:
Betreuer: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Mag.rer.soc.oec. Stefan Biffl
Mitwirkung: Univ.Ass. Mag. Thomas Moser

Wien, 18.12.2008

_____ (Unterschrift Verfasser)

_____ (Unterschrift Betreuer)

Eidesstattliche Erklärung

Christian Astl
7540 Moschendorf 62

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 18.12.2008 _____

Danksagung

In erster Linie möchte ich mich bei meinen Eltern bedanken, die es mir überhaupt erst möglich gemacht haben ein Studium zu absolvieren und die mich über die gesamte Zeit des Studiums unterstützten – Danke.

Ganz besonders bedanke ich mich bei allen Personen, die mir bei der Erstellung meiner Diplomarbeit zur Seite standen, allen voran Ao.Univ.Prof. Dipl.-Ing. Dr. Mag. Stefan Biffl, der mir die Möglichkeit gegeben hat, diese Diplomarbeit zu schreiben.

Mein besonderer Dank geht auch an das gesamte SWIS Team, durch die es mir überhaupt ermöglicht wurde diese Diplomarbeit zu verfassen, vor allem bei Univ.Ass. Mag. Thomas Moser sowie Univ.Ass. Dipl.-Ing. Mag. Richard Mordinyi möchte ich mich bedanken, die mich über die gesamte Zeit unterstützt und zahlreiche Hilfestellungen geboten haben.

Abstract

Integration aims at creating added value from quickly combining existing business applications into a new larger system of systems. A precondition for integration is the ability of these business applications to exchange data and to coordinate the overall system. However, most business applications were designed independently and were not designed for integration with other business applications. Thus, the integration of complex systems bears a number of challenges, e.g. different system architectures, different message protocols.

A promising approach to integrate a large number of heterogeneous systems is System Wide Information Sharing (SWIS), developed in a scientific project at the Vienna University of Technology in cooperation with Frequentis AG. SWIS uses a message-based approach and helps designing a network for safety-critical data exchange between data provider and data consumer services between several organizations and with heterogeneous requirements and/or capabilities. In SWIS, ontologies are information mapping enablers and therefore contain all relevant data and information (e.g., message type, communication mode: push or request/reply, external services, needed converters, etc.) about the applications and systems to integrate. Ontologies are a main part of the semantic web technology and are used for knowledge representation of the real world; in this project for the explicit representation of requirements and for quality assurance of the transformation models in the network design process. The SWIS approach transforms all relevant requirements of the involved systems into a correct and performing solution model (i.e., a configuration for a distributed system used for the integration).

The practical part of the thesis is the creation of tool support for the SWIS approach, more precisely a user interface for the Model Transformation Algorithm (MTA) process. The visualization concept of the SWIS approach communicates emergent properties of the integrated system. The visualization should facilitate a) product improvement by the visual feedback and b) process improvement by providing better tool support and quality assurance. The main focus of the practical work lies on building an easy-to-handle user interface both for experts and non-experts. The user interface provides some process-steps to choose all specific requirements for the calculation of the SWIS solution model. A major criterion for a good user interface is the more effective and efficient enactment of process steps compared to manual enactment.

Zusammenfassung

Systemintegration versucht durch schnelles Zusammenfügen von vorhandenen Unternehmensapplikationen in ein neues großes aus Systemen bestehendes System, einen Mehrwert zu erzielen. Eine Voraussetzung für die Systemintegration ist die Fähigkeit der Unternehmensapplikationen, Daten auszutauschen und das Gesamtsystem zu koordinieren. Jedoch sind die meisten Applikationen unabhängig voneinander konstruiert und daher nicht für die gegenseitige Integration geeignet. Daher beinhaltet die Integration von komplexen Systemen zahlreiche Herausforderungen, wie z.B. unterschiedliche Systemarchitekturen oder unterschiedliche Nachrichtenprotokolle.

Ein erfolgversprechender Ansatz um eine große Anzahl heterogener Systeme zu integrieren, ist der System Wide Information Sharing (SWIS) Ansatz, der im Rahmen eines wissenschaftlichen Projekts an der Technischen Universität Wien in Kooperation mit Frequentis AG entwickelt wurde. SWIS verwendet einen auf Nachrichten basierenden Ansatz und unterstützt beim Aufbau eines Netzwerks für einen sicherheitskritischen Datenaustausch zwischen Datenanbieter und Datenverbraucher in mehreren Organisationen und mit unterschiedlichen Anforderungen und Fähigkeiten. In SWIS werden Ontologien für die Informationsabbildung verwendet; diese enthalten alle relevanten Daten und Informationen (z.B. Nachrichtentyp, Kommunikationsmodus: Push oder Request/Reply, externe Services, benötigte Konverter, usw.) über die zu integrierenden Applikationen und Systeme. Ontologien sind Hauptbestandteil des Semantic Webs und werden für die Wissensrepräsentation der realen Welt verwendet. Im Rahmen dieses Projekts werden Ontologien für die detaillierte Repräsentation der Anforderungen und für die Qualitätssicherung der Transformationsmodelle im Netzwerkaufbau-Prozess verwendet. Der SWIS Ansatz transformiert alle relevanten Anforderungen der involvierten Systeme in ein korrektes und funktionierendes Lösungsmodell (d.h. eine Konfiguration eines verteilten Systems, das für die Integration verwendet wird).

Der praktische Teil dieser Diplomarbeit ist die Realisierung eines Tools zur Unterstützung des beschriebenen SWIS Ansatzes, genauer gesagt eine Benutzerschnittstelle für den Modell Transformations Algorithmus (MTA). Das Visualisierungskonzept für den SWIS Ansatz zeigt die entstehenden Eigenschaften des Integrationssystems. Die Visualisierung sollte folgende Erleichterungen bringen: a) Produktverbesserung durch die visuelle Rückmeldung und b) Prozessverbesserung durch verbesserte Tool-Unterstützung und Qualitätssicherung. Der Hauptfokus der praktischen Arbeit liegt in der Erstellung einer sowohl für Experten als auch für Nichtexperten leicht zu handhabenden Benutzeroberfläche. Die Benutzeroberfläche stellt einige Prozessschritte zum Auswählen bestimmter Anforderungen für die Berechnung des SWIS Lösungsmodells bereit. Ein Hauptkriterium für eine gute Benutzeroberfläche, liegt in der effektiveren und effizienteren Abarbeitung der Prozessschritte im Vergleich zur manuellen Abarbeitung.

Contents

List of Figures

List of Tables

List of Code-Listings

Chapter 1

INTRODUCTION

1 Introduction

Nowadays, most business applications are built to run fully isolated from other business applications, but in many cases, more and more of the isolated applications are combined in order to exchange their data and communicate together. Organizations have the aim to build a large integrated system of their business applications in different locations, so it is possible that each isolated application can communicate with all other applications. In order to realize such integration, a specific system like a middleware is needed. A middleware offers the ability of data transformation and delivery of the messages of the different integrated business applications to the appropriate target applications.

Generally, system integration aims at creating added value from quickly combining existing business applications into a new larger system of systems. A precondition for the integration is the ability of these business applications to exchange data and to coordinate the overall system. However, most business applications were designed independently and were not designed for integration with other varying business applications. Thus, the integration of complex systems bears a number of challenges, like systems with different architectures or systems using different message protocols or systems are running on different platforms.

An Enterprise Service Bus (ESB) provides a possible software architecture construct for system integration which is used in business integration software like OpenESB[1] developed by Sun Microsystems or Sonic ESB[2] developed by Progress Software. In that solutions the appropriate applications communicate via the bus with each other. The bus translates a message from the specific protocol of the sender application to a message using the specific protocol of the receiver application. However, the Enterprise Service Bus is only one solution out of a lot of existing solutions; there exist much more different integration architectures. Some of these integration architectures are described in this thesis in Chapter 2.1.3, like Point-to-Point, Hub/Broker and Bus architecture. But this thesis focuses on an integration approach, which was developed during a scientific project, the so-called System Wide Information Sharing (SWIS) approach.

The SWIS approach was developed during a research project at the Vienna University of Technology in cooperation with Frequentis AG[3] and depicts a promising approach to integrate a large number of heterogeneous systems. This solution realizes an integration of numerous heterogeneous systems by using a message-based approach and helps designing a network for safety-critical data exchange between data provider and data consumer services belonging to different organizations and possessing varying requirements and/or capabilities. In SWIS, ontologies are used as information mapping enablers and therefore contain all relevant data and information about the applications and systems to integrate, like message type, communication mode: push or request/reply,

[1] https://open-esb.dev.java.net
[2] http://www.sonicsoftware.com/products/sonic_esb/index.ssp
[3] http://www.frequentis.com

needed external services, used converters, etc. Ontologies are a main part of the semantic web technology and are used like a knowledge representation of the real world or only part of it. Ontologies are formal models of a specific application domain, and primarily used to facilitate the exchange and partitioning of knowledge. More precisely, an ontology is a data model that represents a set of concepts within a domain and their relationships.

Building an integration solution is not an easy way, because of the diversity of each domain. Therefore a lot of integration patterns exist, which provide a reliable and approved way to realize a precise function. An integration pattern is defined by capturing the knowledge of various experts who are familiar in a field the pattern stands for. Therefore, predefined integration patterns can be used to build an individual integration solution for specific requirements and capabilities. The SWIS integration approach also uses such integration patterns.

The practical part of the thesis is the creation of tool support for the SWIS-project, more precisely a graphical user interface for the Model Transformation Algorithm (MTA) process. The MTA is a core part of the SWIS integration approach and is used to calculate a solution model out of the defined semantic input models. The solution model acts as a configuration model for the integration solution. The visualization concept of the tool support provides predictable emergent properties of the integrated system. On the one hand, the visualization should lead to product improvement by the visual feedback, and on the other hand it should lead to process improvement by providing better tool support and quality assurance. The developed tool for the SWIS approach should help the system integration engineers to find a specific integration solution for a scenario. Furthermore, the user interface should help the integration project manager because less time is needed to model, create and verify the integration solution compared to traditional solutions.

The main goal of the SWIS approach is to integrate different systems in order to enhance inter-operation. This is achieved by transforming all relevant requirements of the involved systems into a correct and performing solution model (i.e., a configuration for a distributed system used for the integration). It is a crucial task that the created solution model by the transformation is a valid solution compared to the original requirements. Another challenge is the creation of the user interface for the tool support, because building a user interface is always critical. Many existing user interfaces cannot be handled by the user because they are very complex and not clearly arranged. So the main focus of the practical work lies on building an easy to handle user interface, which can be used by experts and non-experts similarly. The user interface provides some process-steps to choose all specific requirements for the calculation of the solution model. A major criterion for a good user interface is the more effective and efficient enactment of process-steps compared to manual enactment. The visualization of the solution model as one step in the user interface offers different views (network layer, physical layer, protocol layer) and shows detailed information about the existing network infrastructure (nodes and their links). To evaluate the benefits of the tool support for the MTA process, a case study was performed. The participants had to process a very simple integration example with manual calculation and afterwards automatically by means of the tool support.

The remainder of this thesis is structured as follows: generally, the thesis is divided into related work, research questions, practical work, results, discussion, and summary and further work. In Chapter 2, the related work of this thesis is described. The related work

encompasses an overlook about following four topics: system integration, Model Driven Architecture (MDA), ontologies and integration patterns. First, in the system integration chapter (Chapter 2.1) numerous integration challenges are defined which must be handled to build an integration solution out of many heterogeneous systems. The integration challenges are described in Chapter 2.1.1 and beside some generic problems, like scale, dynamic configuration and the difficulty for finding the relevant data for an integration solution, also technical as well as organizational integration challenges are described. Also in Chapter 2 the different types of integration approaches are specified (see Chapter 2.1.2): business process integration, portal integration, entity aggregation, data integration, functional integration and presentation integration. Business Process integration is an orchestration of interactions between multiple systems by defining a business process model outside of the applications. Portal integration represents an overall user interface of multiple applications so that the user gets a comprehensive view of all the underlying applications. Entity aggregation extends the portal integration so that not only users but also applications can deal with the integration by providing a unified data view. Data integration is an approach to make the high amount of data, containing in different data sources, accessible so that all other systems can use all the data. By means of functional integration the participating systems are combined together using special interfaces. Via this interfaces the systems can access each other to use the underlying data source. With presentation integration all participating applications interacts with the host application via the user interface. Applications can access the functionality of another application through a presentation byte stream by simulating users input and get the required information back by reading the output from the display. In the next chapter (Chapter 2.1.3) the different integration architectures are highlighted, like Point-to-Point, Hub/Broker and Bus architecture. A Point-to-Point communication is the simplest technique to connect all participating systems among each other. Each system has respectively a direct connection to all other participating systems of the integration solution. In the Hub/Broker architecture all involved systems are connected via a central point, the hub. The hub controls the whole communication between the participating systems. At last the Bus communication which provides a special network component (the bus) to connect the single systems. In Chapter 2.2 an introduction to the Model Driven Architecture (short MDA) is given. The basic concepts like models and meta-models as well as the characteristics a model must conform to a certain degree are defined. After the description of the layered architecture of the MDA, the benefits of using a MDA process compared to a traditional software development process are defined. Afterwards in Chapter 2.3 ontologies, which keep the required data and information for the developed SWIS integration approach, are pictured. The different ontology languages are listed, whereas the developed SWIS approach uses the Web Ontology Language (OWL) to define the ontologies. Also designing guidelines for the construction of ontologies, according to clarity, coherence, extendibility, minimal encoding bias and minimal ontological commitment, are specified. Finally, in the ontology chapter, the tool Protégé which is a free, open-source editor for the development of ontologies, is introduced. In Chapter 2.4 some enterprise integration patterns are described.

Afterwards, Chapter 3 identifies the research questions of the thesis, containing the need for an evaluation of the developed SWIS approach by comparing it with traditional integration architectures, like individual interfaces, hub & spoke or service-oriented architecture (SOA). Therefore some evaluation criteria are used: initial planning efforts, initial development efforts, technical adaptations, non-invasive legacy/host integration,

maintainability and customizability. Also the need for an appropriate case study to obtain the benefits of the developed tool support is described within the research questions. Furthermore, also a comparison of the developed SWIS approach with traditional MDA is part of the research questions. And the fourth research question deals with the realization and the use of the defined integration patterns (Chapter 2.4) in the SWIS approach.

Chapter 4 contains the practical part of the thesis. Thereby, a closer look into the developed SWIS integration approach (Chapter 4.1) is done. The whole SWIS process can be subdivided into two main parts the design time and the run time. The design time contains the definition of the semantic models, the Model Transformation Algorithm (MTA) to transform the semantic models into a solution model, and the simulation of the MTA-generated solution model. The run time contains the deployment of the generated and simulated solution model, additional lab testing as well as regular monitoring and auditing of the deployed integration solution. Afterwards the used SWIS ontologies (abstract, domain and customer ontology) and the needed ontology components (e.g. nodes, links, protocols, services, etc.) are explained. Chapter 4.2 contains the Model Transformation Algorithm (MTA), whereas the four main steps of the MTA are described: Step 1: Preparation of semantic data comprised in the input models. Step 2: Calculation of routes between provider and consumer services. Step 3: Calculation of backup routes for each SWIS node. Step 4: Creation of the SWIS solution model. Afterwards, Chapter 4.3 deals with the tool support for the MTA process. The tool support should provide some facilitation in finding a specific integration solution, both for system integration engineers and integration project manager. The tool support provides nine major steps to guide through the MTA process and to get a solution model for the integration solution. And finally in Chapter 4.4, the instructions to the performed case study for the tool support are given.

In Chapter 5, first the evaluation results from the detailed comparison of the developed SWIS integration approach with other existing integration architectures like individual interfaces, hub & spoke and service-oriented architecture (SOA) according to the defined evaluation criteria are highlighted in Chapter 5.1. And second the results of the performed case study for the tool support are presented in Chapter 5.2.

Chapter 6, contains the comparison of the developed SWIS approach with traditional MDA processes (Chapter 6.1) and a detailed description, how the integration patterns, which are defined in the related work, are realized in the SWIS approach (Chapter 6.2).

At last, Chapter 7 concludes the thesis and gives a short outlook for further works.

Chapter 2

RELATED WORK

2 Related Work

This chapter presents an overview about the high potential term system integration. System integration becomes more and more to an important topic. The integration of almost all heterogeneous systems used in an organization can lead to an advantage in competition compared to other organizations without much effort in integration of their different systems. Furthermore this chapter provides an insight into the technique of Model Driven Architecture (short MDA). MDA is an approach for modern software development, by using a layered architecture for software system specifications and development [15]. This technique can help developing a well structured and easy to reuse system integration framework by separating the requirement specifications from the system functionality. Afterwards on introduction to ontologies is given. Ontologies are used in the semantic web technology for knowledge representation. Corresponding to the term of system integration an ontology holds the requirements and capabilities. At last some important integration patterns used to build an integration system are described. Hohpe and Woolf [1] defined a lot of possible integration patterns for building an overall application out of different heterogeneous systems. Some of these defined patterns are listed and explained.

2.1 *System integration*

Nowadays, in companies a large amount of different business applications is running. These applications are often built to work in a single isolated environment. Applications often run on a certain platform and are developed with different technologies that it is not possible to easily merge these applications to communicate together or share their data among each other. Today the computing technology changes from single systems to a coordinated set of systems in which multiple distributed resources are involved [12]. Building a big homogeneous system which covers all the business processes and systems of the entire enterprise is often not feasible as a result of high financial costs and high risks for adoption. Therefore companies are interested in finding a way to integrate all their existing systems. System integration is a naive way to combine all these legacy applications to build one big coherent system where all involved business applications can work together.

But why is system integration actually required? Why do organizations make no efforts to build one cohesive system for their business applications from the outset so that system integration is no task for them? Of course, it is not as easy as it sounds. Even the smallest company does not have a single coherent system in use. They are using numerous different software applications which handle the different needs of the individual business units. There are many reasons for this, like [1]:

- The organizations have software products developed by different vendors.

- The different systems are not built at the same time, one system is older - another system is fresh off the shelf. It is obviously that these different systems are built using different technologies.

- The development of a software system depends on the developer person. Each developer has different experiences and capabilities and this leads to different approaches on how to build a software application.

This chapter provides an introduction to system integration. First the challenges for integrating heterogeneous systems are described. These challenges show to what kind of changes or efforts system integration can lead to. Then the section describing different types of system integration gives a short overlook about the different techniques for system integration solutions. And in the last section the different integration architectures (e.g. Point-to-Point, Hub, Bus) are figured out.

2.1.1 Integration challenges

The integration of heterogeneous systems is not an easy task. There are a lot of challenges which must be handled to reach the aim of a functioning coherent integrated system (e.g. applications are running on different platforms and are located on different places). Current system integration technologies partially provide great techniques for dealing integration tasks, but implicate also numerous limitations. Gorton et al. [29] defined some challenges for system integration regarding the integration of different applications, which must be solved. These challenges can occur as a result of ever-changing technologies applications are developed with and focuses on the requirements pretended for the realization of an integration solution.

Scale

Due to the high amount of digital data sources and the increasing number of modern applications depending on rapid access to multiple data sources, scalability of integration solutions to handle numerous different data sources is a crucial task. Integration solutions should be able to rapidly merge different data from disparate data formats to provide a transparent access to this data from different applications. Therefore modern integration solutions must have a look to scalability to handle numerous data source and have to provide a flexible transformation mechanism to convert data from one format to another.

Dynamic configuration

Integration techniques often must handle different heterogeneous data sources by means of adapters. An adapter converts data from one specific format to another specific data format. But often no appropriate adapter for a data source is available and so a new adapter must be created to achieve the needed tasks. Furthermore the development of an adapter is not as easy as it sounds. It could lead to high costs for development and it is important to consider the time an adapter needs to convert the relevant data of a data source. If there are many requests for accessing the data source, the adapter must be built with main focus on performance. For system integration purposes modern integration

technologies have to minimize cost and time factors for the integration of data sources. At best an integration technology automatically creates suitable adapters for the integrated data sources to establish access to the data from participating applications.

Finding Relevant Data

Finding relevant data out of a mass of data for a specific application is becoming a real problem due to the increasing amount of integration solutions with big infrastructures and enormous existing data. Most traditional data sources do not possess with semantic search functions where data of interest can be indicated and easy located Modern integration techniques should be able to automatically find the relevant data from the data sources by extracting semantics of the data sources and linking the appropriate data to the participating applications.

Hohpe and Woolf [1] and Trowbridge et. al [2] also defined numerous challenges for the integration of different business applications. The challenges are divided into the following technical and organizational challenges.

2.1.1.1 Technical integration challenges

Technical integration challenges highlight the problems of system integration according to the hardware (e.g. networks) and regarding to the integration solution (e.g. how an integration solution must be built).

Networks are unreliable

Within the integration of heterogeneous applications and in order to exchange information among each other, data must be transported from one system to another system across the network. Unlike different processes on a single system where no network hardware is needed, the communication over networks has a large amount of possible problems. Data between two systems must be sent within a given time and the reliability of the network is fundamental for the communication. In order that the data has to pass through a lot of different network elements (e.g. phone lines, routers, switches, etc.) the amount of possible error sources is very high.

Networks are slow

A distributed integration solution must have a special focus on performance measures. Consider that the exchange of data in a network is multiple times slower than the communication on one single system. Therefore the integration solution must be designed to fulfill the needed performance measurements.

Each application is different

No application equals another application. Each application is actually unique. Applications differ between the programming languages they are written in, the operating platforms they are running on, or the used data formats, in short each application has their

own construction type and look. So it is a great challenge to build integration solutions which can deal with these different technologies.

Changes in applications are inevitable

No application can be in use forever. Technology changes over time and so also applications start to get out of date. An integration solution has to consider this fact and so the single applications must be virtually decoupled from each other. Otherwise if one application changes it has an effect on all other participating applications. This separation of the dependencies is called *loose coupling*.

2.1.1.2 Organizational integration challenges

The organizational integration challenges focus the problems that can have an implication to the organizational structure of a company or the company itself. A bad und imperfect integration can cause high loss of money and can lead to some worse effects.

Changes in business policy are sometimes required

An existing business application always covers just a certain functional field (e.g. accounting, finance, customer relationship management, sales, etc.). In some cases system integration can cause changes in actual business policy of a company because not only the communication between multiple computer systems but also the communication between business units needs to be established.

Integration can become crucial for the company

Integration has a strong implication to the company. After integration of the most essential business functions to a corporate integration solution, the correct and faultless functioning is assumed. A worse integration solution can lead to an enormous high loss of money if the business processes of the company aren't implemented optimal. Late changes in the integration solution are difficult to achieve and often not applicable. Therefore building an integration solution needs much information about the participating systems, the company's policy, the internal business functions, and so on. Integration aims for a level of sophistication to get an efficient and well-formed solution.

Limited control over the applications

Sufficient amount of control over the attended applications represents another integration challenge. In most cases a company uses many different legacy applications. There is no chance for the integration developers to alter such applications because these applications cannot be accessed for altering due to restriction of control. This means additional effort to the integrators to establish integration of such applications.

The next chapter describes different types of integration. They show how applications can be merged together and which integration techniques can be used to get an appropriate integration solution.

2.1.2 Types of integration

System integration techniques focus on different levels to combine participating heterogeneous system. There are multiple types of system integration techniques which differ at the level where the integration is done. Basically two groups of integration types exist. The first group of integration techniques focuses on the design of an integration layer. This group contains the following types:

- ***Business Process integration:*** Process integration is an orchestration of interactions between multiple systems by defining a business process model outside of the applications.

- ***Portal integration:*** Portal integration represents an overall user interface of multiple applications so that the user gets a comprehensive view of all the underlying applications.

- ***Entity aggregation:*** Entity aggregation extends the portal integration so that not only users but also applications can deal with the integration by providing a unified data view.

The second group of integration techniques focuses on the mechanism how the systems are connected together. This group consists of the following types:

- ***Data integration:*** Data integration is an approach to make the high amount of data, containing in different data sources, accessible so that all other systems can use all the data.

- ***Functional integration:*** By means of functional integration the participating systems are combined together by providing special interfaces. Via this interfaces the systems can access among each other to use the underlying data source.

- ***Presentation integration:*** With presentation integration all participating applications interacts with the host application via the user interface. Applications can access the functionality of another application through a presentation byte stream by simulating users input and get the required information back by reading the output from the display.

In the following chapters two integration types from the second group are described in detail, the data integration and the functional integration. Afterwards a comparison of all three types of the second group is done.

2.1.2.1 Data integration

The data integration mechanism integrates systems at the logical data layer. The idea is to provide an overall interface for accessing different data sources of multiple applications. In an enterprise many applications exist which keep large amounts of information in data stores like flat files or databases. Other applications which want to use

the information connect directly to these data stores. An advantage of data integration is that the applications which held the data sources must not be changed to provide an interface on where the other applications get access to the underlying data. Another advantage is that a user, who needs some data from different data sources, must not care about the location of the wanted data. The user does not need to know which application stores the specific data. The data integration approach gives users the ability to specify *what* data they want, instead of determining *how* to obtain the data [9]. But the integration solution within data integration uses a strong binding to the data structure of the underlying applications. This means that in case of changing one data model of any application, also the integration solution has to be changed to meet the modified specifications and to access the data source furthermore. In general data integration is easy to develop, because no application logic of the integrated applications is used.

Organizations have to care about the possibilities to share the data between the different heterogeneous applications. There are some techniques for such integration. Data integration can be realized by means of shared databases, maintain data copies, and file transfer. Each of these techniques gives an answer to the question, on how to integrate multiple applications that are not designed to work in correlation and are not constructed to change information among each other.

Shared databases

In a shared database all participating applications store their data to share it with all other applications. Each application can access the shared data in the database when it is needed. An advantage of a shared database is that the containing data is always up to date. Because all applications use the same database and changes of data takes place centralized and the other applications always get the actual data. But shared databases have to struggle with some disadvantages. The common use of the same database involves problems in semantic discrepancy. This problem is very difficult to solve but it cannot be left regardless because it leads to dozens of incompatible data. For the design of shared databases it is a very difficult task to build a suitable database which can handle the requirements of multiple applications.

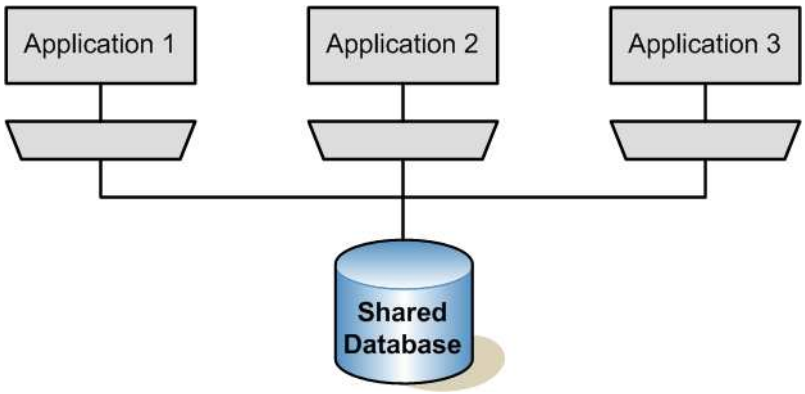


Figure 2.1: Shared Database [1]

Fowler in Hohpe and Woolf [1] explains that a resulting schema which meets the needs for the different applications is often difficult to use for application programmers. Beside the technical difficulties there are also multiple political troubles in designing the appropriate schema. Another downside for shared databases occurs by using packaged applications. Vendors use their own schema for storing the data and this schema won't work with the schema designed for the integration solution. By using just one shared database for multiple applications it can come to huge database performance problems if the applications frequently read and update the stored data. In the worst case the applications can run into deadlocks, where the applications lock each other out of the data. Figure 2.1 presents the structure of a shared database.

Data copies

The technique of maintaining data copies is an extension of the shared database approach. Instead of only one shared database for all participating applications, there exists one database for each application and all databases have exact the equal data stored in it. For synchronizing purposes the data of the individual databases are replicated to each other. This means, the changed data from one database is copied to all other databases. There exist several techniques to achieve this. Teale et al. [10] describes different patterns for maintaining data copies:

- ***Move Copy of Data:*** This pattern defines a fundamental construct which is used for all other types of asynchronous data copies. It presents the data movement building block consisting of data source, data movement set, data movement link and the data target.

- ***Data Replication:*** Data replication provides a special form of data movement. It presents an easy way to acquire and manipulate the data. But the fact that both – the data source and the data target – must be updated tends to a high complexity of this pattern. Figure 2.2 shows the structure of the data replication pattern.

- ***Master-Master Replication:*** This data movement pattern describes a bidirectional data replication between data source and data target. Including conflict detection and resolution for handling simultaneous updates of the same data to different data copies during one transmission interval.

- ***Master-Slave Replication:*** The master-slave replication is an alternative to the master-master replication. It uses a unidirectional data replication to store the data from the data source to the data target. The data of the target will be overwritten at the transmission.

- ***Master-Master Row-Level Synchronization:*** This pattern provides a special type of the master-master replication. It uses the same functionality with the only difference that the conflict detection and resolution happens at the row level.

- ***Master-Slave Snapshot Replication:*** The specific characteristic of the master-slave snapshot replication pattern is the transmission of a complete replication set. The transmitted replication set comes from the data source, may be updated and

stored in the data target. This technique is suitable to equalize the data source and data target after that a master-master replication takes place.

- ***Capture Transaction Details:*** Provides a design to manually capture transactional information which is necessary for incremental replication when no database management system transaction log is available for some reasons.

- ***Master-Slave Transactional Incremental Replication:*** This pattern provides a specific form of the master-slave replication. Only the changed data is transmitted from the data source to the data target with help of transmitting transactional information.

- ***Master-Slave Cascading Replication:*** The master-slave cascading replication pattern shows how to design a deployment for master-slave replication from one data source to multiple data targets. Thereby a concatenation of replication links with databases, which sits between and act as data source and data target, takes place. The targets can subscribe to a replication set which will be replicated.

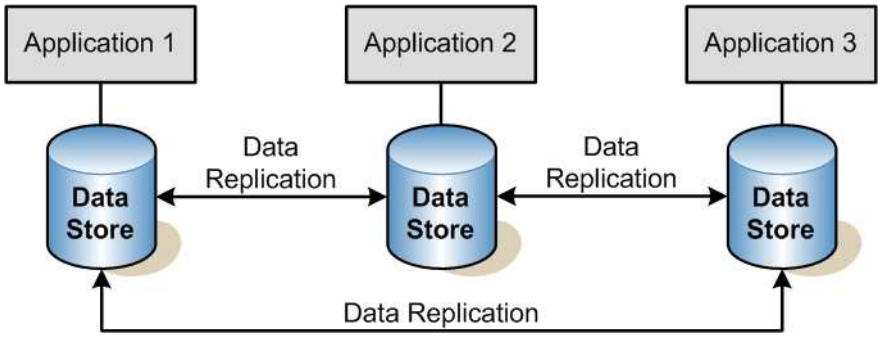


Figure 2.2: Data replication [2]

File transfer

By using file transfer to change data between multiple applications, an application stores its information in a file and another application reads the needed information from this file. A crucial task with the usage of files is the decision what format should be used. The output from one application fulfills only in the rarest cases the requirements for another application. So the output must be transformed in a neutral format so that other applications can deal with it. In the past different file formats were used, but in the last time the XML-format has established as the current to use method. An advantage of the file transfer method is that no knowledge about the internal logic of the applications is needed to build an integration solution. As you can see on Figure 2.3 a separate logic – the so-called exporter – does the transformation from the applications internal format to the neutral format to store the data in a file. On the other way a separate logic – the importer – converts the data from the neutral format to the applications internal format to read the stored data. By using such separate logics the participating applications are completely decoupled from each other. Changes in application are acceptable as long as they produce the data in the same format. The file transfer method to exchange data between

applications sounds very simple and straightforward, but there are also some issues to manage. One problem of storing data in files and read data out of the files arises when files are processed too frequently. For storing and reading files a certain amount of effort is needed. It is necessary to limit the processing of the files to just a few times (e.g. daily, weekly, etc.). But this leads to another big disadvantage. Due to low frequently processing of the data the particular applications can run out of synchronization. A short example: a change of a customer's address is made in the customer relationship management system. The data will be processed once a day at midnight. If the billing system sends a new bill to this customer on the same day the change of the address is made, it will be sent to the customer's old address because the information between the CRM and the billing system are not synchronized yet.

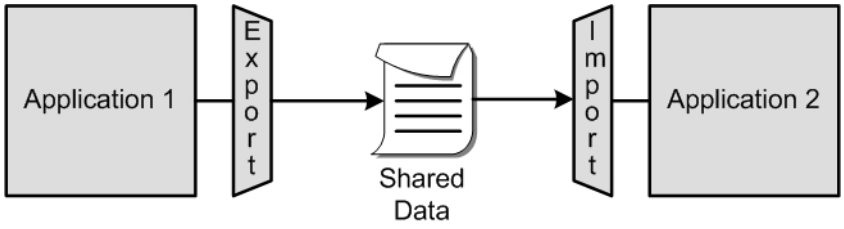


Figure 2.3: File transfer [1]

2.1.2.2 Functional integration

Functional integration is also known as application integration and integrates systems at the logical business layer. This means that the business logic of an application which keeps data in data stores is shared, so that other applications can use the data store across the application without direct access to it. The individual applications will be connected via interfaces and specifications allocated by the integrated application. But often some of the participating legacy applications don't provide any interfaces or specifications and an integration of such applications is hard or rather not possible.

To realize an integration of multiple applications by means of functional integration, two preconditions are needed. First: availability of the business function which is used for the integration in the business logic of the source application. If this condition is not given the source application must be modified to implement the needed business functionality. Second: remote access to the source applications API is needed. If an application only supports local API calls and middleware must be created which receives remote API calls and transforms them into local calls, accepted by the application.

The Implementation of a functional integration solution is realized by means of distributed objects, message-oriented middleware or service-oriented architectures.

Distributed Object Integration (Remote Procedure Invocation)

With Remote Procedure Invocation (also called *Remote Procedure Call,* RPC) an integrated application is designed as a large-scale object which has the data encapsulated inside and is used for encapsulated integration of different applications. Each application

must provide an interface so that other applications can communicate and interact with the application. An application gets the data from the participating application by asking the application directly. To modify the data of another application, the initiating application makes a call to the other application. It appears that the objects inside the remote applications communicate with each other in such a way as they would communicate via a local connection.

Figure 2.4 shows a schematically picture of a remote procedure invocation.

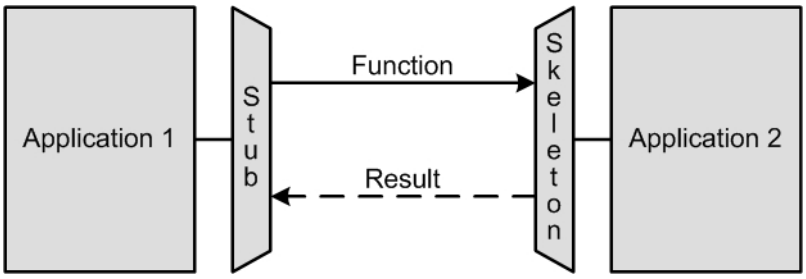


Figure 2.4: Distributed Object Integration [1]

Two specific techniques used by Distributed Object Integration are stub and skeleton. The stub has the task to create and issue a client request to send it to the receiver. And the skeleton has the task to receive an incoming client request and to deliver this request to the object implementation [19]. Examples for Remote Procedure Invocation techniques are CORBA, DCOM, .NET Remoting and Java RMI.

- ***CORBA (Common Object Request Broker Architecture):*** This technique provides a specification for distributed systems founded by the Object Management Group (OMG):

  "CORBA is a mature, standard middleware that combines the interoperability, deterministic execution, and absolute dependability required by distributed embedded systems." [20]

  With CORBA the semantic of method-calls between the participating distributed systems are standardized. The interfaces of the allocated objects and services are specified by means of the Interface Definition Language (IDL). IDL is comparable with other interface definition languages, because also an exact syntax to formulate methods and their parameters is provided. An interface consists of many methods and the objects specify which interfaces they implement.

- ***DCOM (Distributed Component Object Model):*** The Component Object Model (COM) was developed by Microsoft and builds a fundamental technique in Microsoft's operating systems beginning with Windows 95. COM is a specification for building language and location independent objects that offer particular interfaces to access each other [21]. The Distributed Component Object Model is an enlargement of the COM technology to establish a network-based communication between objects located on different systems. To invoke an object

on a target system from a calling application running on a source system the DCOM technique is responsible to coordinate the communication between source and target system. At first DCOM gets the location of the target system out of configuration information stored in the registry. Then a client proxy on the source system and a server proxy (stub) on the target system are created and finally the communication between the two proxies is conducted directly via point-to-point connection [2]. See Chapter 2.1.3.1 for more information about point-to-point integration architecture.

- ***.NET Remoting:*** The .NET Remoting technique is also developed by Microsoft and is an enhancement of the Distributed Component Object Model (DCOM) using the Microsoft .NET Framework.

  ".NET remoting enables client applications to use objects in other processes on the same computer or on any other computer available on its network." [22]

  This technique is similar to the before described DCOM technique. Remoting also uses object references for communication between client and server. The only difference to DCOM is that .NET Remoting doesn't get the reference to the target system out of configuration information stored in the registry. The client retrieves the needed reference by means of the remoting infrastructure rather gets the reference passed as parameter by the *Activator.GetObject()* method [23].

- ***Java RMI:*** Java Remote Method Invocation (short RMI) was introduced by Sun Microsystems as a mechanism where two distributed Java-based applications can communicate together. One application can invoke methods of remote Java objects from other applications running on different systems in a network. A remote method call with RMI occurs within four steps. First the server registers a so-called remote object in a specific RMI registry where the object must have a unique name. Second the client offers an interface the remote object has to implement. The client looks for the appropriate object in the RMI registry and creates a reference to this object. Third the client invokes a method of the remote object on the server. Required parameters are transferred over the network. Fourth the server sends back the return value of the invoked method or alternatively an error exception to the calling client.

Message-Oriented Middleware Integration (Messaging)

Message-Oriented Middleware (MOM) becomes more and more important due to the raising amount of developing loosely coupled applications distributed over a large scale network. MOM is defined as following:

"MOM provides an infrastructure that transmits messages and events to the widely spread components of a service, gluing them together through logical coupling." [24]

With Message-oriented middleware integration the participating systems are connected together by using asynchronous message queues. To achieve such integration standardization to a proprietary message-oriented middleware is required (see Figure 2.5). The Figure shows that the particular applications are connected by means of a Message

Bus and they communicate together through messages with a little data in it. Consider that an asynchronous and durable communication bears the risk of losing messages during transportation due to network failures or system errors.

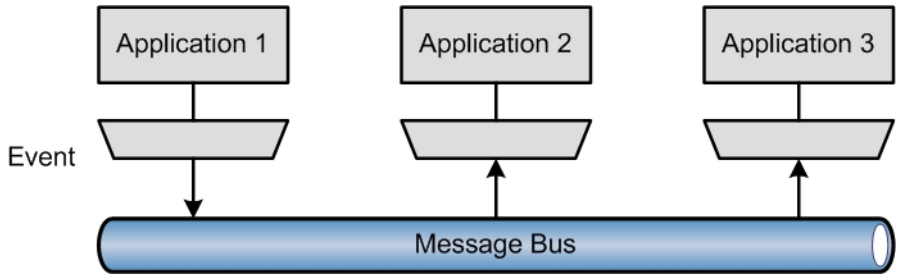


Figure 2.5: Message-Oriented Middleware Integration [1]

Service-Oriented Integration

The technique of service-oriented integration uses web services to connect different systems together. Participating systems must be built for sending and receiving XML web service messages. The W3C consortium describes a web service as following:

"A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards." [11]

The service-oriented integration approach uses a web service description language (WSDL) for the description of the application interfaces. Each application has a separate WSDL file, providing the contract how other applications can establish a communication to this application. In Figure 2.6 the basic structure of a service-oriented integration is displayed. Main parts are the service provider application and the service consumer. The offering service is implemented in the service provider application and can be used by other applications. To achieve this, the service provider has to specify a service interface containing the contract to fulfill for consuming the implemented service. The service gateway containing in the service consumer encapsulates the logic for consuming the requested service.

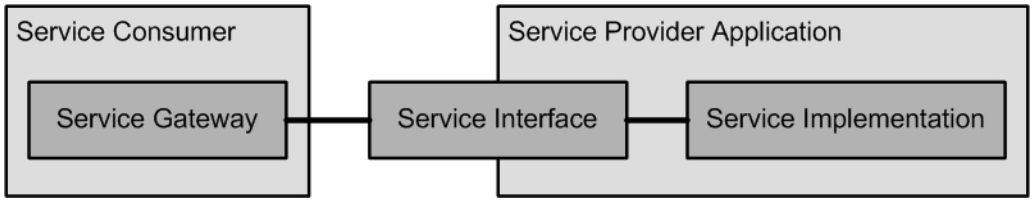


Figure 2.6: Service-Oriented Integration [2]

2.1.2.3 Comparison of the different integration types

Figure 2.7 shows a comparison of the different integration solutions depending on the layer on which the systems are connected together. The left picture represents the data integration technique. It is obviously that the integration system is connected at the data layer. The centric picture shows the functional integration where the integration solution is connected to the business logic. And the right picture displays presentation integration with its connection to the presentation layer.

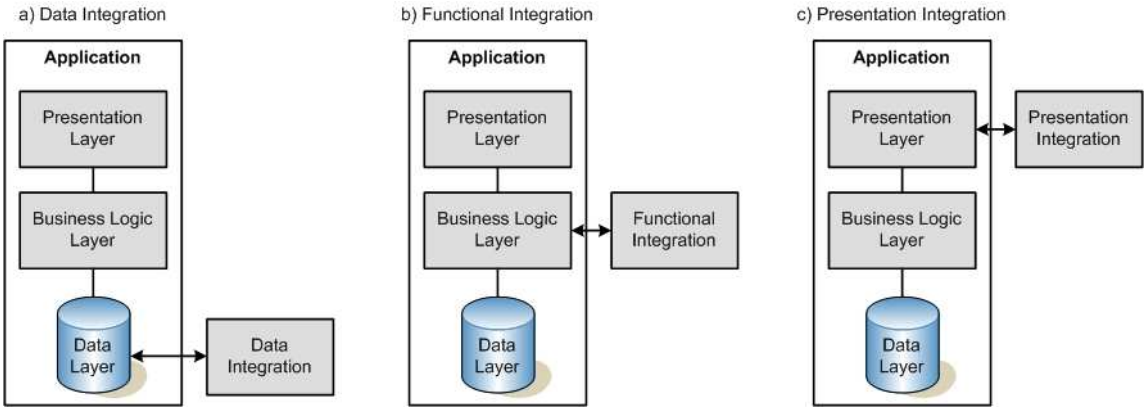


Figure 2.7: Comparison of the different system connections

In Table 2.1 a summarized overview about the different patterns for system connection integration is given. The table also describes the respective problem of each pattern and their specified solution.

Pattern or pattlet	Problem	Solution
Data Integration	How do you integrate information systems that were not designed to work together?	Integrate applications at the logical data layer. Use a Shared Database, File Transfer, or Maintain Data Copies
Shared Databases [Hohpe04]	How can multiple applications work together and exchange information?	Have multiple applications store data in a single database. Define a schema that handles the needs of all relevant applications.
Maintain Data Copies [Teale03]	How can multiple applications work together and exchange information?	Have multiple applications access multiple copies of the same data. Maintain state integrity between copies.
File Transfer [Hohpe04]	How can multiple applications work together and exchange information?	At regular intervals, have each application produce files that contain the information that the other applications must consume. After you create it, do not maintain the file.

Functional Integration	How do you integrate information systems that were not designed to work together?	Integrate applications at the logical business layer. Use Distributed Object Integration, (proprietary) Message-Oriented Middleware Integration, or Service-Oriented Integration.
Distributed Object Integration (see also *Remote Procedure Invocation* [Hohpe04])	How do you integrate applications at the logical business layer?	Develop systems that have object interfaces that can be consumed remotely by other systems.
Message-Oriented Middleware Integration (see also *Messaging* [Hohpe04])	How do you integrate applications at the logical business layer?	Use proprietary message-oriented middleware to send messages asynchronously.
Service-Oriented Integration	How do you integrate applications at the logical business layer?	Use Web services to expose interfaces that can be consumed remotely by other systems.
Presentation Integration	How do you integrate information systems that were not designed to work together?	Access the application's functionality through the user interface by simulating a user's input and reading data from the screen display.

Table 2.1: System Connection Patterns [2]

2.1.3 Integration architectures

For the integration of different systems there exist several ways how these systems could be connected together to build one big corporate system. Generally there exist three main possible integration architectures to establish the integration of systems. The difference between these architectures is the way how senders and receivers are connected together. In the following section the three various architectures are described, starting with the basic Point-to-Point connection, following by the more complex Hub connection and finally the Bus connection.

2.1.3.1 Point-to-Point architecture

The Point-to-Point communication is the simplest way of connecting participating systems among each other. Each system has respectively a direct connection to all other systems. A Point-to-Point communication infrastructure is shown in Figure 2.8. To establish the communication some precondition has to be given. The first requirement to send a message from sender to receiver is that the sending system must know where the receiving system is located because a sender could be connected to more than one system. Furthermore each involved system can only deal with specific message formats and so the sender must transform a message from one format into another format that could be handled by the receiver. That is a big disadvantage of such integration architecture. Each system needs a separate integration solution to all other involved systems. Generally each

system in a Point-to-Point integration has a direct connection to all other systems and requires a specific message transformation for any connection. If systems supported message format changes, the message transformer of all associated systems that communicate with the changed entity must be updated.

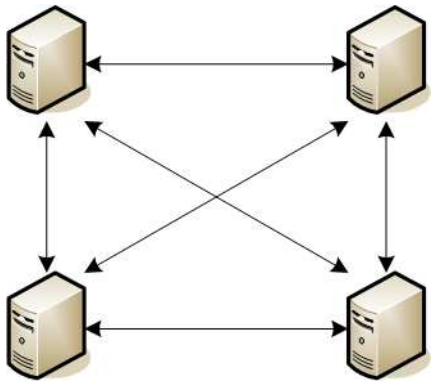


Figure 2.8: Point-to-Point integration architecture

Finally a Point-to-Point integration is easy to handle if just a few systems are connected together, but for more and more systems the effort to maintain such integration increases very fast. With *n* participating systems $n \cdot \frac{n-1}{2}$ different integration solutions exist. So it is obviously that a Point-to-Point integration is quite reasonable for small organizations with few systems.

2.1.3.2 Hub/Broker architecture

This kind of integration architecture connects all involved systems via a central point, namely the hub. Figure 2.9 shows the basic design of the hub integration architecture.

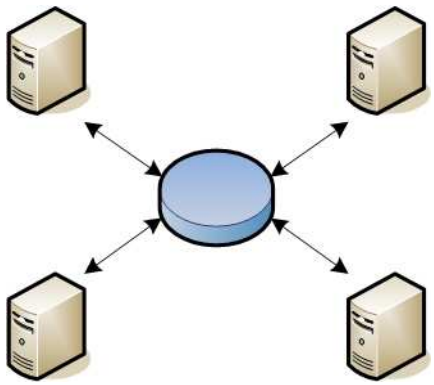


Figure 2.9: Hub integration architecture

The hub controls the whole communication between senders and receivers. All participating systems do not have to care about the location of the receiver and do not have to know the message format supported by the receiver. A sender forwards his message to the hub and the hub takes the message, transforms the message and sends the transformed message to the correct receiver. This technique is often called the “hub & spoke” integration architecture.

The hub architecture follows a broker pattern. The task of a broker is to decouple sender systems from the receiver systems by coordinating the communication between them. Systems are loosely coupled if only few common variables are used by the systems or if the common variables are less addicted to other influencing factors [13]. By using hub architectures, the single systems are separated from each other and so a loose coupling takes place. The decoupling of the participating systems is achieved by three main tasks [2]:

- ***Routing:*** Routing is the task of determining the location of the receiving system of a message and performing the routing via direct or indirect communication.

- ***Endpoint registration:*** Endpoint registration is used by the involved systems to register themselves with the broker. After registration the system is public and can be found by other systems.

- ***Transformation:*** Each participating application uses its own specific data format. To make it possible that applications can communicate with each other, the messages must be converted to the right format. The transformation is the mechanism to convert a message from one format to another format.

Generally the basic broker pattern can be classified into three other types of brokers: Direct Broker, Indirect Broker and Message Broker. Figure 2.10 pictures the Broker pattern and their three subtypes.

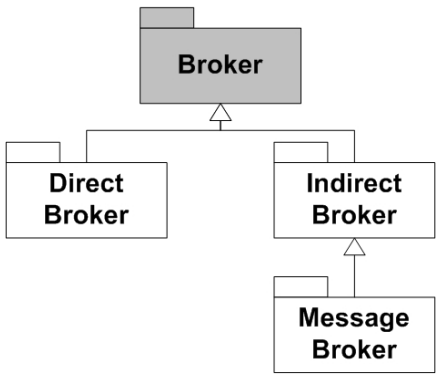


Figure 2.10: Broker pattern refinement [2]

The three subtypes of the Broker pattern are described following.

Direct Broker

The only task of a direct broker is to build up the communication between the sender and the receiver system. The sender asks the broker about the location of the target system and the direct broker only sends back the location information. After the initialization the sender communicates directly with the receiver without intervention from the broker.

Indirect Broker

An indirect broker does not only establish the communication between two endpoints but also manages the whole communication after the initialization. The broker acts like a middleman allowing central control of the traffic. A sender system transfers the message to the indirect broker and the broker forwards it to the appropriate receiver system.

Message Broker

A message broker has the same properties as an indirect broker, but provides a specialized form. This broker especially uses messages for the communications. He receives a message from the sender system, transforms the message to the correct message format of the receiving system and finally forwards the transformed message to the receiving system. A message broker is often found in a so-called “hub & spoke” architecture [1].

Examples for the Broker pattern technique are Microsoft’s Distributed Common Object Model (DCOM), Microsoft’s .NET Remoting, the Common Object Request Broker Architecture (CORBA) developed by the Object Management Group (OMG), the Universal Description Discovery and Integration (UDDI) standard and Microsoft’s BizTalk Server.

2.1.3.3 Bus architecture

In this architecture, all participating systems are connected via a special component the so-called bus (see Figure 2.11). The easiest communication mode of a bus is the broadcast communication. A system sends its message to the bus and the bus forwards the message to all other connected systems. Therefore the systems themselves must decide if a message is addressed to them or not.

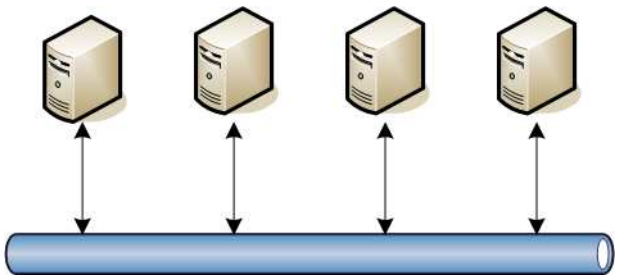


Figure 2.11: Bus integration architecture

Generally, an integration bus provides a common communication mechanism to connect heterogeneous systems. To achieve the integration, the involved systems must follow some agreements. Trowbridge et al. [2] defined three criteria for the participating systems to be able for a connection to the bus:

- ***Message schema:*** All connected systems must support the correct structure of the messages.

- ***Command message:*** Command messages are used for reliable invocation of a procedure provided by another application. A command message is a normal message with a command in it.

- ***Shared Infrastructure:*** To build a bus architecture a predefined infrastructure is needed for sending messages from sender to receiver, e.g. message router, publish/subscribe mechanism. These different types of shared infrastructures are described below.

The bus architecture provides different ways to manage the flow of the messages. One possible way is to use a message router for the administration of the messages. Another way is the publish/subscribe mechanism. Both are common ways for the shared infrastructure of a bus system.

A message router "… *consumes a message from one message channel and republishes it to a different message channel, depending on a set of conditions*". [1]

The publish/subscribe system uses a mechanism, in which subscribers (subjects who wants to get a specific messages) can express their interest in a message or a sort of messages. If the publisher (subject who publishes messages) creates a message which matches the interest registered by the subscriber, the subscriber will be notified of the message [5].

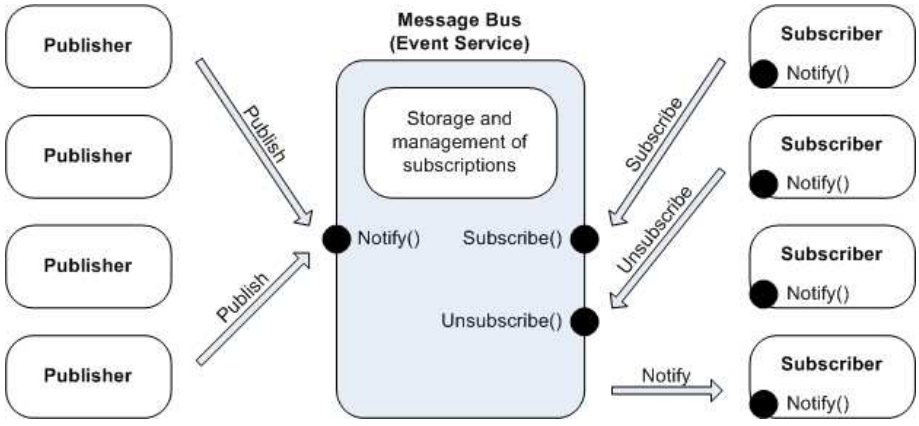


Figure 2.12: A simple object-base publish/subscribe system [5]

Figure 2.12 shows a simple publish/subscribe mechanism. In that case the sender of a message publishes the information to the message bus and the consumers have the

possibility to subscribe to certain information they want to receive. If the message bus gets a message from a publisher it checks who has subscribed for such a message type and then forwards the message to all subscribers by notifying them. The message bus is the core element and provides three operations: *Subscribe(), Unsubscribe()* and *Notify()*. All subscribers register their interests in certain messages by calling the *Subscribe()* operation of the message bus and so they don't have to know the exact publisher of such messages. The message bus stores and manages these subscriptions. A publisher don't need the subscription-information stored in the message bus, he just sends their message to the *Notify()* operation of the message bus. By calling the *Unsubscribe()* operation a subscriber quits an existing subscription.

Generally some differences in the design of the publish/subscribe mechanism lead to three main distinctions: topic-based, content-based and type-based publish/subscribe mechanisms. The different mechanisms will be described in the next sections, but first a short introduction into a message router takes place.

Message router

Messages are sent via a message channel from one system to another system. The sending system writes the message into the channel and the receiving system reads the message from the same channel. The two communicating systems don't know about each other. If a sending system is connected to more than one receiving system, it uses one message channel per receiver. The problem is that the sender doesn't know which receiver wants to get a certain message. This problem will be solved by using a message router. A router is placed between sender and receiver and is connected via separate message channels to the participating systems (similar to a hub, see Chapter 2.1.3.2). The message router takes the message from the sender's message channel and republishes it to the receiver's message channel. Thereby the router doesn't modify the message; it just handles the routing between sender and desired receiver. Using a message router, the rules defining to which receiver a certain message from a sender should be forwarded can be established in a single central location.

Topic-based publish/subscribe

This approach represents the functionality for early publish/subscribe systems and is based on grouping systems together by means of defined topics. The functionality can be described in one sentence: A subscriber joins a certain group and all assigned messages are sent to the subscribers via broadcast. The topic-based publish/subscribe mechanism can be divided into two refinements [2]: the broadcast-based and the list-based publish/subscribe.

- ***Broadcast-based publish/subscribe:*** The broadcast-based publish/subscribe mechanism uses a very simple and unmanaged way to notify the required recipients of a message. The publisher creates a message and sends it to all other connected systems via broadcast. Each system has the task to analyze the incoming message if the message comes from the publisher that it is subscribed to. If the message was meant for the particular system, the system accepts the incoming message, but if not the incoming message will be ignored from the receiving system. In fact that each message is sent to all participating systems and

each receiving system is responsible to check if the message is assigned for them, this pattern is often called the publish/subscribe channel from Hohpe [1]. A more better and sophisticated way of topic-based publish/subscribe that requires fewer network traffic is the list-based publish/subscribe pattern.

- ***List-based publish/subscribe:*** The list-based publish/subscribe mechanism is one way to control the data traffic unlike a sender broadcasts his message to all other systems. This approach uses a list which contains all subscribers for a particular subject. In other words when a specific sender transmits a message, the bus forwards the message by means of the list to those receivers who are interested on it. This mechanism is also described as observer pattern. The observer pattern is defined as following:

  "*Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.*" [4]

  With the observer pattern the relations between sender and receivers are defined. The sender is the subject and a receiver is an observer. One subject can have one or more observers. The registered observers will be notified if the associated subject has changed.

Content-based publish/subscribe

In a content-based publish/subscribe system the participating systems subscribe to subjects depending on certain conditions. The subscribers will be notified if a published message conforms to those conditions. Unlike topic-based publish/subscribe systems, a content-based publish/subscribe system can be used more flexible because the subscriptions are coupled to the message-content addicted to certain conditions and are not only coupled to an overlook of the message (the topic). In generally, the content-based publish/subscribe mechanism has its strengths in information propagation from publisher to subscriber, especially in a large-scale distributed network [6].

Type-based publish/subscribe

Eugster [14] describes the type-based publish/subscribe mechanism as a high level variant of the publish/subscribe paradigm which provides type safety and encapsulation without disruption of the routing mechanism. The type-based approach presents advantages to exchange messages in many-to-many publish/subscribe communication environments and so it is well suited to use in P2P applications.

2.2 *Model Driven Architecture (MDA)*

Model Driven Architecture (MDA) developed by the Object Management Group (OMG) is an approach for modern software development, by using a layered architecture for software system specifications and development [15]. The defined system specifications describe the software system at different abstraction levels. Each level provides a special view of the system. MDA is used for separating business and application logic from the

underlying platform technologies [25]. In other words, MDA is the separation of the specification of system functionality from the actual implementation of the specified functionalities [16]. All defined specifications are expressed as models.

This chapter gives an overview to the Model Driven Architecture technique. To understand this comprehensive topic, the first section presents some elementary explanations about models and metamodels (What is a model? Which properties does a model have? etc.). Then the layered architecture of MDA is figured and last the benefits of using the MDA technique compared to a traditional software development process are listed.

2.2.1 Models and Metamodels

This chapter describes two fundamental parts used in Model Driven Architecture, models and metamodels.

"A model is a coherent set of formal elements describing something (for example, a system, bank, phone, or train) built for some purpose that is amenable to a particular form of analysis." [31]

Another definition for a model comes from Stachowiak [26]. He specifies that a model is essentially a scale, detailedness and/or functionality shortened and accordingly abstract representation of the original system. In short, a model is a replication of the real world. It must be noted, that a model is just a representation of an original system and not a copy. For example if someone builds a true to detail object according to an original one so that the replication equals the original in every little detail, the replication is a copy and not a model. It is obviously that a model has to concentrate and represent just some particular details of the original. Models are a basic part in Model Driven Architecture.

Selic [18] has appointed five key characteristics an engineering model must conform to a certain degree:

Abstraction

Abstraction is the most important characteristic of a model. A model is always a shortened representation of a system that it specifies. Abstraction means that the model is not a one to one replication of a system, but reflects only the relevant properties of a regarding system. This means that irrelevant details are unattended in the model. Therefore abstraction is almost the only method to deal with the complexity of an always increasing sophisticated functionality of software systems.

Understandability

Understandability is also an important characteristic for a model. If a model is suppressed in a language which needs much intellectual knowledge to understand it, a model will provide no benefit. A model must present their information in a form (e.g. a notation) that it could be understood without significant intellectual effort. Therefore the

language must directly lean on our intuition. Understandability and expressiveness are in a direct relationship together. Expressiveness presents the degree of ability to illustrate a sophisticated process with less information. For that reason a model is a good model when not much intellectual effort is needed to understand the content provided by the model.

Accuracy

Useful models must be accuracy. This means that a model must provide the modeled system in such a way that it offers a concise representation of the system's features the model is interested in.

Predictiveness

With models it should be possible to exactly predict the interests the modeled system focuses on without non suggesting properties, by experimentations or formal analysis. Predictiveness relies on the accuracy characteristic of a model and the modeling form.

Inexpensiveness

The last characteristic a model should possess is inexpensiveness. The construction and analysis of a model should be essentially cheaper than the construction and analysis of the system itself. It would be very inefficient and uneconomical for building models if the modeling of a system costs more than the creation of the actually system.

Metamodels are another basic part appearing in correlation with Model Driven Architecture. They define language concepts and grammar to specify models. Rather a metamodel is a specification of a model and describes how other models are constructed. They define what is acceptable in building a model of an original system. Seidewitz [32] defines that a metamodel characterizes the possibilities about what can be expressed in the valid models of a certain modeling language.

There are some more concepts which occur in relation to the MDA approach, Platform-Independent Models (PIM) and Platform-Specific Models (PSM). A PIM represents a formal specification of systems structure and characteristic, without including technical details. The Platform-Independent Models are constructed for an implementation on different platforms. A PSM specifies how to realize the defined functions of a PIM on a specific platform. It represents enough details and information (e.g. software architecture) to generate a complete coded application [34]. But it is still defined as a model. Out of a Platform-Specific Model the code for the whole implementation of a software system can be created.

2.2.2 MDA Layered Architecture

The Model Driven Architecture approach is based on a layered architecture. Generally, the MDA architecture consists of four layers: the M3-layer which represents a meta-metamodel, the M2-layer, representing a metamodel, the M1-layer, depicting a concrete model and the M0-layer which illustrates the reality. In Figure 2.13 these four layers of the MDA architecture are displayed.

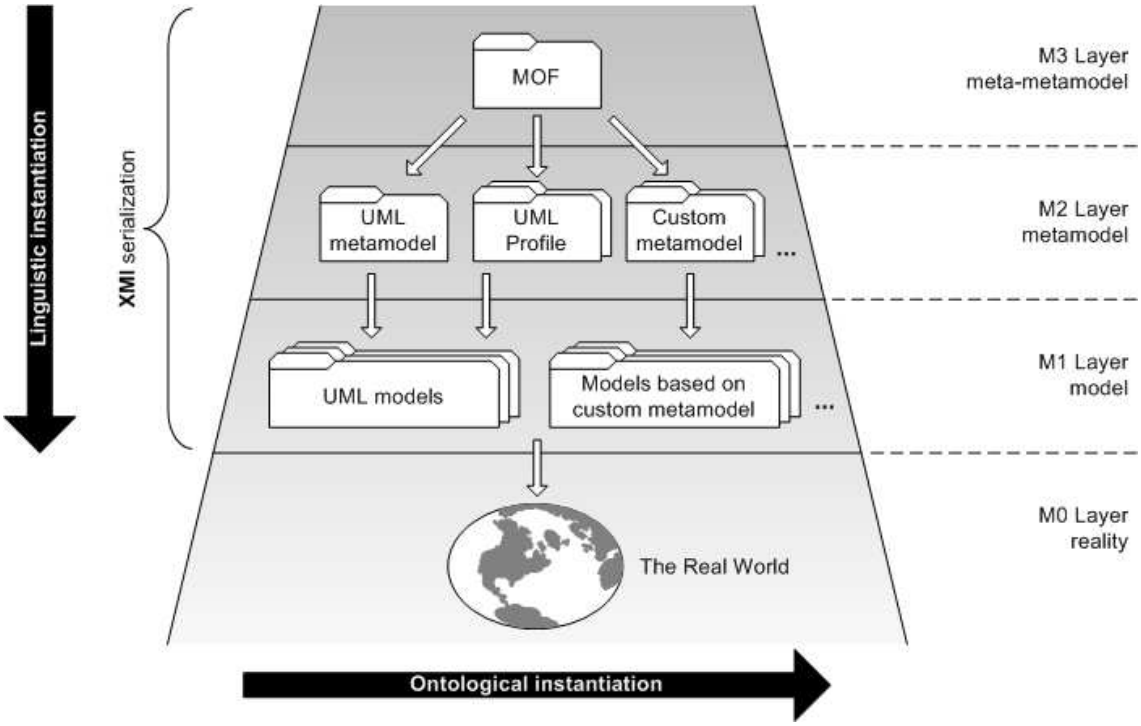


Figure 2.13: The MDA four-layer architecture [8]

The meta-metamodel layer (M3-layer) is the topmost level of the MDA architecture. This layer is represented by the Meta Object Facility (MOF). MOF builds an industry standard environment to export models from one application and import it to another application, transferred over a network and transformed into different formats [27]. MOF represents a basis to define other modeling languages, like UML (Unified Modeling Language), IDL (Interface Definition Language) used in CORBA or CWM (Common Warehouse Metamodel). Even MOF is described in MOF and can be subdivided into EMOF (Essential MOF) and CMOF (Complete MOF). EMOF is a simple language for defining metamodels and is useful for metamodelers. CMOF is an extension for EMOF with support and management of metadata. In generally the M3-layer provides a specification of modeling languages and is primarily used to express metamodels of the M2-layer [32].

The metamodel layer (M2-layer) contains the actual metamodels (model of model) defined by the MOF. This layer represents an instance of the M3-layer. UML is one of numerous metamodeling languages. The Unified Modeling Language technique is used to help system architects, software engineers and software developers by providing tools for better analysis, design and implementation of software-based systems or miscellaneous modeling challenges [33].

The model layer (M1-layer) contains representations of the real world in terms of models. Such a model is an instance of metamodels defined in the M2-layer (e.g. UML model of a software system).

The reality layer (M0-layer) represents an instance of the models defined in the M1-layer. This layer contains actual objects of the real world, like persons, buildings, etc. The following example in Figure 2.14 pictures the relations between the four MDA-layers.

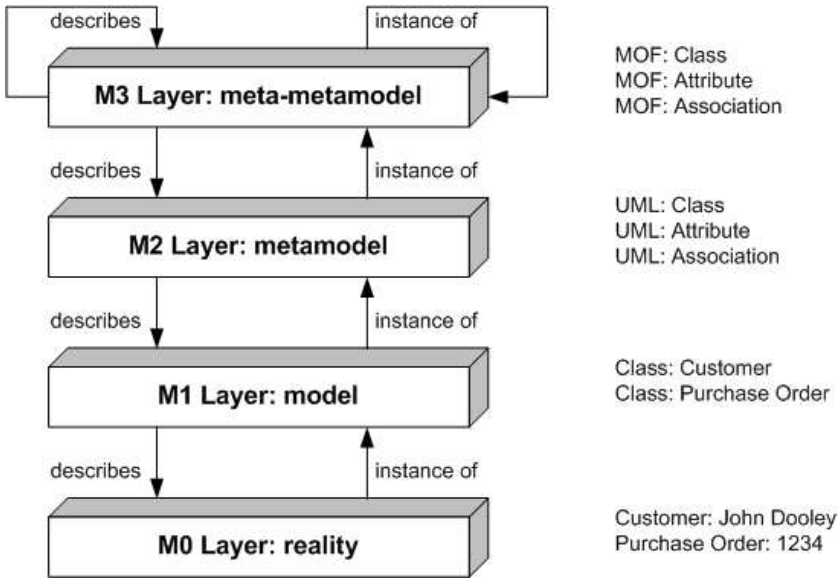


Figure 2.14: MDA-layered Architecture Example

2.2.3 MDA benefits

Developing software by means of the Model Driven Architecture approach provides some improvements of the software development process. Kleppe et al. [17] has researched the benefits of MDA and categorized them into four classifications: Productivity, Portability, Interoperability, and Maintenance and Documentation. These benefits are explained in relation to a traditional software development life cycle with their containing problems. Figure 2.15 displays a schematically representation, a so-called waterfall model of a traditional software development process.

2.2.3.1 Productivity

As you can see in Figure 2.15, specifications between the requirement, analysis, design and implementation phase are represented in terms of text and diagrams. This means that phase 1 through 3 produce many text documents and diagrams for the later software implementation. Multiple pictures and several UML (Unified Modeling Language) diagrams such as use case diagrams, class diagrams, activity diagrams, entity relationship diagrams, etc. are generated. Sometimes it is unimaginable which high amount of documents and diagrams are produced especially for one software development process. But these masses on written paper are created in the early phases and then rapidly lose their relevance after the implementation begins. During the implementation phase a code is produced and tested before deployment. The implementation mostly differs from afore

generated documents and diagrams. The written specifications are lacking maintained and so they present no exact mapping of the created implementation. This becomes a serious problem due to permanent changes at the code level. Introducing the changes to the documents and diagrams is very time-consuming and therefore hard to maintain.

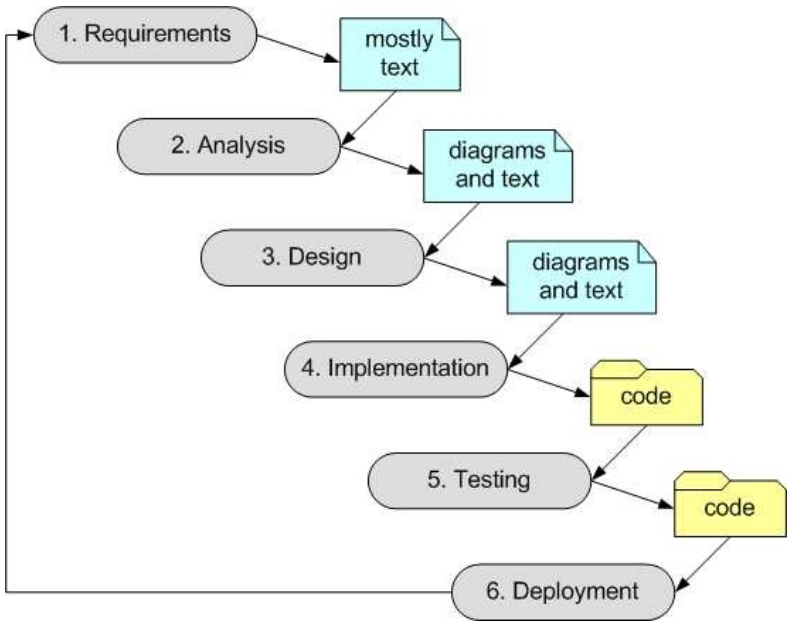


Figure 2.15: Traditional Software Development Life-Cycle [17]

Model Driven Architecture attempts to solve the problem of creating mass of documents and diagrams by using a Platform-Independent Model (PIM). By means of a PIM the determined requirements and capabilities for a software product are represented in the form of a model. This primarily created PIM will be later on transformed into a Platform-Specific Model (PSM) which comprehends specific information about the underlying platform. A PSM conforming to another platform can be easily generated out of the defined PIM. But the extensive creation of the PIM is the only disadvantage of using MDA for reaching a higher productivity. It looks very easy, but much effort is needed to produce a correct PIM for further processing. Often only a high skilled specialist can achieve the creation of the abstract PIM. But once the PIM was created, the productivity benefits to generate PSMs for different platforms are very high if tools to automatically transform the specified PIM into a PSM are used.

2.2.3.2 Portability

Portability describes the possibility to use the same program or model on different platforms without modification. In a Model Driven Architecture portability can be obtained by using Platform-Independent Models (PIMs). A PIM is defined in a platform independent manner and therefore can be used on different platforms without any modifications. If a PIM should be used on a specific platform, first of all it must be transformed into a Platform-

Specfic Model (PSM) containing platform specific details and are only able for the use on a specific platform. Out of the underlying PIM, any PSM can be generated. Therefore all information specified in a PIM is portable. The benefit for portability can be increased by using automated transformations by means of various tools. A key benefit of portability in MDA with a PIM is, that independent of new developed platform technologies the created PIM can be used furthermore. With a specific transformation tool, according to the new platform technology, the PIM is transformed into a functioning PSM without altering the original PIM.

2.2.3.3 Interoperability

Interoperability deals with the problem that a specific system should be able to interact with other existing systems developed in another technology. It is crucial that the different systems support a common working to gain a result. The multiple generated PSMs for different platforms out of one common PIM may have particular similarities and therefore some correlations, so-called bridges in MDA. The different PSMs are not able to directly communicate among each other, but by means of the bridges a communication can be established. A bridge transforms the concepts according to one platform into the concepts according to the other participating platform. Within MDA, interoperability is achieved by additionally generating the required bridges between the generated PSMs.

Due to the generation of PSMs for different platforms from one PIM, all needed information to create a bridge to establish the communication between the different PSMs is available. For each item in a generated PSM the appropriate item in the PIM is known and therefore the corresponding item in another generated PSM can be determined. Out of the gained information it is possible to deduce the relation between the items in the different PSMs. Extended with the technical details of the different platforms, which are already known for the transformation from the PIM to a PSM, a bridge can be created.

In MDA, bridges can be automatically generated by means of special tools. Therefore MDA supports interoperability for all underlying PSMs.

2.2.3.4 Maintenance and Documentation

In a traditional software development process the numerous created documents are often very hard to maintain. After creation of the source code out of the requirements the documents are neglected. If changes in the source code are made, the documents are often not updated to meet the altered requirements. Therefore in a traditional software development process changes must be updated multiple times on different places. Within MDA, changes are only updated on a single place in the PIM. Out of the PIM the different PSMs are generated and out of a PSM the source code is generated. Therefore each source code is an exact representation of the PIM and no inconsistencies between source code and specifications can occur. Generally a PIM illustrates a form of a high-level documentation used by any underlying software system.

2.3 *Ontologies*

In this chapter the term ontology will be explained. An overlook about the basic concepts of ontologies is given. Furthermore the main operational areas of ontologies are described. Primarily a definition of an ontology is presented.

"An ontology is a formal, explicit specification of a shared conceptualization. A 'conceptualization' refers to an abstract model of some phenomenon in the world by having identified the relevant concepts of that phenomenon. 'Explicit' means that the type of concepts used, and the constraints on their use are explicitly defined. 'Formal' refers to the fact that the ontology should be machine readable, which excludes natural language. 'Shared' reflects the notion that an ontology captures consensual knowledge, that is, it is not private to some individual, but accepted by a group." [30]

In general, ontologies are a main part of the semantic web technology and are used like a knowledge representation of the real world or only part of it. Ontologies are formal models of a specific application domain, and primarily used to facilitate the exchange and partitioning of knowledge. More precisely, an ontology is a data model that represents a set of concepts within a domain and their relationships. The word *ontology* has its origin from the Greek words *ontos* (=being) and *logos* (=word). From a philosophical point of view an ontology refers to the subject of existence, that is the study of being as such [8]. Gruber [35] defines an ontology as an explicit specification of a conceptualization. Where a conceptualization illustrates an abstract, simplified picture of the world used for representation and designation. Each knowledge representation follows a certain degree of conceptualization, either explicitly or implicitly. Moreover ontologies can effectively support software development processes, primarily by providing a continuous data model [36].

According to Powers [37] ontologies consist of four main components: classes, relations between classes, properties of classes, and constraints on relationships between the classes and properties of the classes. But additionally an ontology also consists of individuals which represents instances of concrete types. A class represents concepts of a domain, for example the concept "vehicle" with his specifications: car, motorcycle, bus, etc. (a set of objects with common properties). A relation represents an association between class concepts of the domain. A property represents an attribute to describe objects in the ontology. And the last component, a constraint defines statements for a relation between classes or properties that cannot be formally expressed by the other main components.

Following the main components of an ontology (individuals, classes, attributes and relations) are described in detail:

- ***Individuals (instances):*** The individuals build the basic components of an ontology and are similar to object instances in the object oriented programming. Individuals represent concrete types like house or car, and additionally more discrete types like numbers or words.

- ***Classes (concepts):*** The classes represent abstract groups, sets, or collections of objects and are similar to abstract objects in the object oriented programming. Classes can contain other classes or individuals or a combination of classes and

individuals. The single ontologies can vary among each other on the conditions they support. They distinguish whether classes can contain other classes, or whether a class can belong to itself and so on. Also restrictions can be made to prevent that an ontology can have an invalid state, like whether an individual inherits from two disjunctive classes.

- ***Attributes:*** Attributes represent properties, features and characteristics of an object in an ontology. An attribute consists of at least a name and a value, whereas the value can be a normal value type and also a complex data type.

- ***Relations:*** Relations specifies how the various objects are related together. A relation between objects in the ontology is described by means of attributes. Together, all the specified relations characterize the semantic of an ontology. Generally different types of relations exist: the subsumption relation (is-subtype-of, is superclass-of, whereas the objects are members of a common group of objects), the is-a relation (tree structure with child and parent objects, whereas each object is a child of a parent) and the meronymy relation (part-of relation).

2.3.1 Ontology languages

Ontologies can be expressed in different languages. Gómez-Pérez et al. [39] divide the logical ontology languages into traditional ontology languages and web-based ontology languages. Whereas traditional languages are developed in the early 1990s for artificial intelligence purposes, the web-based ontology languages are developed at the beginning of the web-age to use the characteristics of the internet. In this chapter we focus on the web-based languages shown in Figure 2.16.

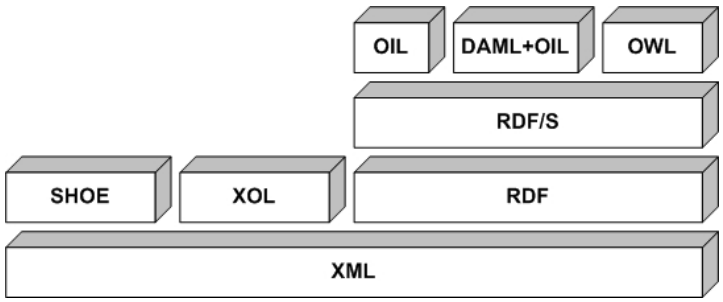


Figure 2.16: Web-based ontology languages [39]

The syntax of the web-based ontology languages is based on common web markup languages like HTML or XML. In the following enumeration the web-based ontology languages shown in the diagram are described shortly:

- ***SHOE:*** The Simple HTML Ontology Extension language uses frames and rules and was developed as an extension to the HTML markup language. With SHOE it is possible to describe a webpage in a semantic manner.

- ***XOL:*** The XML-based Ontology Language was developed to include primitives based on the OKBC (Open Knowledge Base Connectivity) protocol. OKBC is a protocol to access knowledge bases stored in different knowledge systems.

- ***RDF:*** The Resource Description Framework language is used for defining web-resources in a semantically way. RDF was developed by the World Wide Web Consortium (W3C). Listing 2.1 shows an easy example of the syntax of the RDF language.

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:dc="http://purl.org/dc/elements/1.1/">
  <rdf:Description rdf:about=" http://www.tuwien.ac.at">
    <dc:title> TU Wien Home</dc:title>
    <dc:creator> Vienna University of Technology </dc:creator>
  </rdf:Description>
</rdf:RDF>
```

Listing 2.1: Syntax of the Resource Description Framework (RDF)

- ***RDF/S:*** The RDF Schema extends the Resource Description Framework and represents an easy language to specify domain-ontologies. With RDF/S the declarations defined in RDF can be structured hierarchically into classes and instances. Furthermore it is possible to precisely specify the relations between the particular properties. RDF/S builds a basis for the next three described ontology languages: OIL, DAML+OIL and OWL.

- ***OIL:*** The Ontology Inference Layer adds a frame-based knowledge representation to the underlying RDF/S and supports formal semantics provided by Description Logics [40].

- ***DAML+OIL:*** The DARPA Agent Markup Language is a communication language for software agents [41] and builds in combination with OIL the basis for OWL. DAML+OIL uses an object oriented approach and therefore it is designed to specify the structure of a specific domain in terms of classes and properties [42].

- ***OWL:*** The Web Ontology Language is a semantic markup language used to create ontologies constructed in a formal representation language. Unlike of just providing information to humans, ontologies written in OWL can be used by applications to process the content of information [43]. OWL is best suitable for the description of relations between classes, properties and other individuals [8]. There exist three different types of OWL which differ in the capability of expression, OWL Full, OWL DL and OWL Lite. OWL Full provides all OWL language constructs and additionally offers the use of RDF constructs. OWL DL is a subtype of the OWL language constructs with some restrictions (e.g. a class must not be an instance of another class) and without support for RDF constructs. OWL Lite represents a minimal subset of the OWL language construct with several restrictions and was developed as easy to implement language. In Listing 2.2 an example to the syntax of the OWL language is given.

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns="http://www.tuwien.ac.at/ontology.owl#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:p1="http://www.owl-ontologies.com/assert.owl#"
    xmlns:dc="http://purl.org/dc/elements/1.1/"
    xml:base="http://www.tuwien.ac.at/ontology.owl">

  <owl:Ontology rdf:about=""/>

  <owl:Class rdf:ID="Car">
    <rdfs:subClassOf rdf:resource="#Vehicles" />
  </owl:Class>

</rdf:RDF>
```

Listing 2.2: Syntax of the Web Ontology Language (OWL)

2.3.2 Designing Ontologies

In this chapter some design criteria for the development of ontologies are described. Such criteria become very crucial, because if we represent something of the real world in the form of an ontology, it is essential to make the suitable design decisions. For the development of well designed ontologies a set of objective criteria is needed, which corresponds to the scope of the resulting items. Therefore Gruber [38] appointed five main criteria and principles that have to be considered for the creation of ontologies. They are significant for ontologies used for knowledge sharing and interoperation between applications in a shared manner. These criteria serve as guiding principles and help to evaluate the developed ontology design.

Clarity

In general, an ontology has to clearly represent the intended sense of the environment it is used for. The ontology must be specified in an objective manner and should not depend on social or computational impacts. Furthermore an ontology must be completely defined. Not only the essential capabilities but also additional sufficient capabilities are preferred to get a complete definition and not just a partial definition about the environment. These definitions contained in an ontology should be described with formal languages.

Coherence

It is crucial to develop an ontology in a way that it is coherent. Coherence means, that ontologies should support various implications which conform to the definitions. Therefore it is necessary that the specified conventions are logically conforming to each other. But the term coherence is not only limited to inferences which should satisfy the definitions, it should also relate to any concepts that are described in an informally way. Such informal described concepts are documents and samples specified within a formal language. If a derived concept out of the specified conventions does not conform to the definitions, the ontology is not coherent.

Extendibility

This criterion focuses on the possibility of further development and enhancement of an ontology. An ontology must be designed for arbitrary expandability, and therefore should provide a conceptual basis for later appending of anticipating tasks. It is crucial that the implementation of an ontology is designed for featuring a monotonically extension. Generally, within an existing ontology new items should be added by using the available vocabulary of the ontology without altering the previous containing definitions. If it is not possible that a new item is specified in the same scheme as the underlying ontology, the ontology must be able to deal items written in another format. But the original scheme should not be modified.

Minimal encoding bias

An ontology should be designed in a decoupled manner, in other words, the conceptualization should not depend on a specific encoding format. The design of an ontology should not match only one particular case of notation or implementation, if so, an encoding bias exist. Because of the reuse of developed ontologies the encoding bias must be as small as possible.

Minimal ontological commitment

To satisfy the purposed knowledge sharing tasks, an ontology needs to fulfill the minimal ontological commitment. On the other side for a versatile usage of the ontology it is crucial that the ontology requires as few assumptions about the underlying modeled world as possible. Therefore a basic ontology, based on a minimal ontological commitment, can be used from many different parties for many different models due to the individual configuration and instantiation of such an ontology. It is always advisable to minimize the ontological commitment of ontologies by defining only elementary conditions of the represented knowledge to allow the most models using such a minimized ontology.

2.3.3 Protégé

Protégé is a free, open-source editor for the development of ontologies. It is based on Java and is supported by a huge user community. The Protégé community provides a set of tools for the creation of domain specific models and knowledge-based applications represented as ontologies. An ontology created with Protégé can be exported into a different set of ontology language formats like RDF(S), OWL and XML Schema. Protégé consists of a complex software architecture [8], and is constructed for an easy extension by a simple plug-in mechanism. By means of the provided plug-and-play environment, Protégé builds a flexible base for the use of rapid prototyping and application development [44]. For the development of ontologies the Protégé platform provides two core editors: the Protégé-Frames editor and the Protégé-OWL editor. The first editor supports the creation of frame-based ontologies conforming to the Open Knowledge Base Connectivity Protocol (OKBC). OKBC provides a method to access the knowledge which is stored in special knowledge representation systems. The second core editor of Protégé supports the creation of

ontologies developed in the Web Ontology Language (short OWL – already described in Chapter 2.3.1) which are used for the semantic web.

Protégé can be downloaded at http://protege.stanford.edu and Figure 2.17 displays the user interface of the Protégé version 3.3.1.

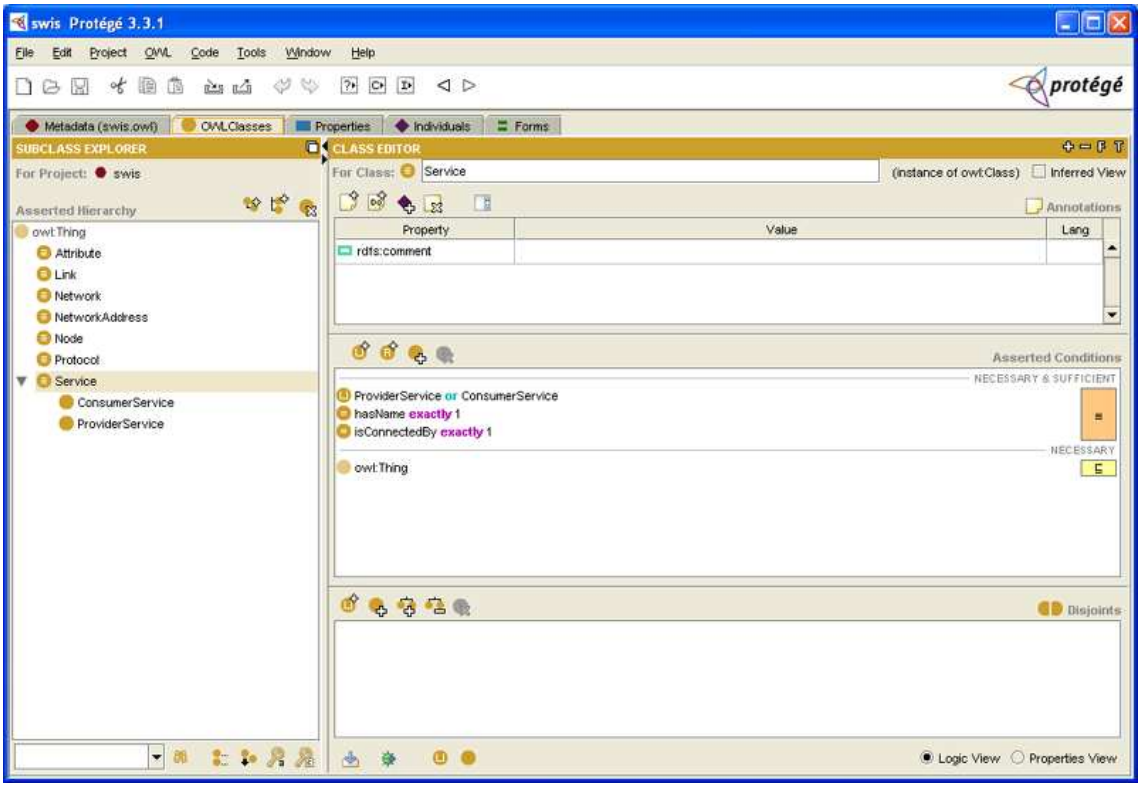
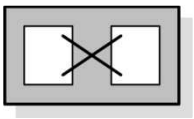


Figure 2.17: User Interface of Protégé 3.3.1

2.4 *Integration Patterns*

Integration patterns could be used as elementary parts for building a system wide integrated application to combine different heterogeneous legacy applications. Patterns represent a reliable way for capturing the knowledge of experts who are familiar in a field the patterns stand for. They are used when no “straight-forward” solution exists because each solution is unique depending on different requirements and environmental influences. So a pattern provides just a part of the overall solution which can be individually combined with other patterns to achieve the most suitable solution for a specific scenario. The advantage of patterns is that they are continuously enhanced by integration solution architects or other experts. Each pattern involves the experience of the integration engineers by frequently using the patterns in different integration solutions and considers possible failures of a pattern. Therefore a pattern will be enhanced by means of expert’s knowledge. Hohpe and Woolf [1] identified numerous enterprise integration patterns. The relevant patterns for the System Wide Information Sharing (SWIS) system are described hereafter.

Message Translator

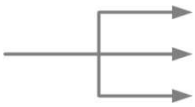


The Message Translator pattern gives an answer to the following question:

"How can systems using different data formats communicate with each other using messaging? – Use a special filter, a Message Translator, between other filters or applications to translate one data format into another." [1]

In enterprise integration solutions messages are transmitted between heterogeneous systems. Each of the participated systems only understand its own (often proprietary) data format and often is not able to handle other messages presented by the other applications and created in another data format. This proprietary data formats has to be translated from the data format of the sender to the data format of the receiver. The various applications in an integration solution often communicate via standardized data formats among each other. This means that each application has a built in Message Translator which transforms the message from the proprietary data format into a standardized data format. So it is possible to send the message to another application. The other application gets the message and transforms it with their built in Message Translator into an internally processable proprietary data format. But often it is not possible to agree on the used standardized data format and therefore other agreements must be taken. As preview, SWIS does support the translation to and from individual data formats.

Publish-Subscribe Channel



The Publish-Subscribe Channel pattern gives an answer to the following question:

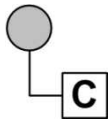
"How can the sender broadcast an event to all interested receivers? – Send the event on a Publish-Subscribe Channel, which delivers a copy of a particular event to each receiver." [1]

Many mechanisms exist to broadcast a message (or event) from a sender (originator of the message) to all receivers who are interested in this message. The observer pattern (already described in Chapter 2.1.3.3) is the most common technique for building a Publish-Subscribe Channel. The observers are completely decoupled from the originator of an event. An originator provides the mechanism where all observers for a specific event can express their interest in it and will be notified if an event is generated by the originator. The originator does not care about how many observers want to get a notification.

The Publish-Subscribe Channel pattern has a simple mode of operation. The publisher offers just one corporate input channel but splits into multiple output channels for the subscribers. One output channel for one subscriber. For the announcement of an event the publisher must send only one message into the Publish-Subscribe Channel. The channel

themselves duplicates the message and sends a copy of the primarily message to each of the output channels.

Command Message

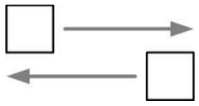


The Command Message pattern gives an answer to the following question:

"How can messaging be used to invoke a procedure in another application? – Use a Command Message to reliably invoke a procedure in another application." [1]

Basically a Command Message is comparable with a Remote Procedure Invocation. Both provide the ability to access functions embedded in other distributed applications. A Remote Procedure Invocation works in a synchronous way. This means that a call to a remote application will be processed immediately and the caller is waiting until the end of the processing. But this isn't always as good as it sounds. Often an invocation of a remote procedure cannot be done immediately. In case when the network is unavailable or the remote application is not waiting for a remote invocation a call to the distributed application is impossible. For such circumstances it is important that a call to a remote procedure is done asynchronously. The Command Message pattern is the solution to achieve an asynchronous call. A specific procedure invocation is packaged in a Command Message and will be transmitted to the remote application as a message across a message channel. The receiver gets the message and starts the packaged procedure invocation from the caller locally. After invocation an answer is send to the caller as a callback. This means that the caller specifies an operation which will be executed after the reply from the invocation receiver arrives. Therefore the calling system must not wait for the response.

Request-Reply



The Request-Reply pattern gives an answer to the following question:

"When an application sends a message, how can it get a response from the receiver? – Send a pair of Request-Reply messages, each on its own channel." [1]

In a message oriented integration solution the communication between two applications is limited to handle only a one-way communication. A sender transfers the message to the receiver and does not retrieve an answer from the receiving system. The communication just works in a single direction. However, in an integration solution with different participating systems the caller of a distributed function, which is available on a remote application, often expects an answer representing the return value of the called function. Therefore a two-way communication is needed. But how could this be achieved? A two-way communication via Messaging over a Message Channel is not possible because the

channel transfers the message only in one direction. For that reason the Request-Reply pattern uses a second Message Channel for transferring the reply message back to the sender. The first channel is used as request channel and transmits the request message from the requester to the replier. The replier receives the message and sends a reply message via the reply channel to the requestor.

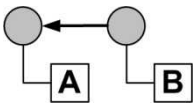
Return Address

The Return Address pattern gives an answer to the following question:

"How does a replier know where to send the reply? – The request message should contain a Return Address that indicates where to send the reply message." [1]

Because of the totally decoupling of the participating systems of an integration solution, a replier can get different messages from different requestors via the same request channel. But instead of hard-coding the reply channel of each requestor in the replier, which can make the solution very inflexible and hard to maintain, a Return Address pattern is used. Each request message will be extended with information about the used reply channel of the respective requestor. It is also possible that the requestor advertises not the address to his own reply channel but the address to the reply channel used by another system (e.g. the requestors' callback processor). So each request-reply message can be directed from the requestor, where they should be replied to. A Return Address is added to the message header and interpreted by the replier.

Correlation Identifier



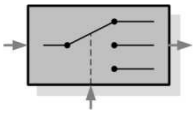
The Correlation Identifier pattern gives an answer to the following question:

"How does a requestor that has received a reply know which request this is the reply for? – Each reply message should contain a Correlation Identifier, a unique identifier that indicates which request message this reply is for." [1]

If two systems are connected together over a network and one system wants to invoke the other system via Remote Procedure Invocation (see Chapter 2.1.2.2) the call will be processed in a synchronous way. This means that the invoking system waits for the result sending by the invoked system. But in an integration solution with messages the calls are often processed in an asynchronous way. The caller can send an invocation and it can happen that the caller does not remember the call and is not able to deal with the incoming result. Or he sends numerous calls and finally does not know which result belongs to which call. This problem can be solved by using a Correlation Identifier. Each request including a request message is marked with a unique identifier by the requestor. The replier achieves the request and stores the request ID. After processing the request the replier adds the

stored identifier as correlation identifier to the completed reply. The requestor is then able to assign the reply to the appropriate request by means of this correlation identifier.

Dynamic Router

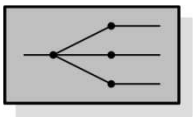


The Dynamic Router pattern gives an answer to the following question:

"How can you avoid the dependency of the router on all possible destinations while maintaining its efficiency? – Use a Dynamic Router, a router that can self-configure based on special configuration messages from participating destinations." [1]

A problem of a message-based integration solution lies in the correct routing of a specific message to their recipients. Each receiver presented in an integration system is interested only in particular messages which meet specific conditions. To achieve such correct routing, a special mechanism is needed that knows the destination and the individual interests of the particular receivers. A Message Router (see Chapter 2.1.3.3) performs the technique of transferring a message to all receivers who are interested in it. A Message Router has a built-in knowledge about the participating receivers and their specific interests. But this is not very convenient if the rules of the receivers or rather the receivers itself changes frequently. Therefore a Dynamic Router will be used. A Dynamic Router equals a Message Router but has an additional channel where the receivers can advertise their existence and their interest patterns for receiving messages. This additional channel is called as control channel. The Dynamic Router stores the provided information of the receivers in a rule base and therefore handles the correct routing of messages without much maintaining effort.

Recipient List



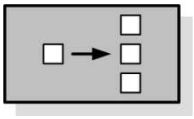
The Recipient List pattern gives an answer to the following question:

"How do we route a message to a dynamic list of recipients? – Define a channel for each recipient. Then use a Recipient List to inspect an incoming message, determine the list of desired recipients, and forward the message to all channels associated with the recipients in the list." [1]

For delivering a published message to a set of recipients presented in an integration solution a Publish-Subscribe Channel (already described above) is used. All receivers who want to get a published message subscribe to the channel. The problem hereby is that the subscriptions in a Publish-Subscribe Channel cannot be controlled by the messages itself. A receiver gets either all messages from the channel (if he is subscribed to) or no messages (if he is not subscribed to). It is eligible that a sender wants to determine the

particular recipients for each message. To achieve this, a specific mechanism is needed that can handle the use of different lists of receivers for each message type (e.g. some recipients wants to get all messages from a specific type, like flight plans). The Recipient List pattern will be used for routing messages to a list of receivers which are different depending on the sent message. A Recipient List gets a message with an embedded list of recipients, removes the list from the message for performance reasons and to prevent that other receivers can see the receiver list, and forwards the message to the respective recipients.

Splitter

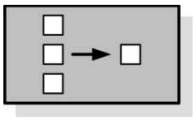


The Splitter pattern gives an answer to the following question:

"How can we process a message if it contains multiple elements, each of which may have to be processed in a different way? – Use a Splitter to break out the composite message into a series of individual messages, each containing data related to one item." [1]

In an integration solution the limitation of traffic between the participating systems is elementary. If a message contains multiple elements and each containing element should be processed on a different system, sending the whole message to all required receivers is not efficient. Therefore the entire message has to be split into many sub-messages, so that each recipient gets an individual message containing the needed elements for processing. The challenge of splitting one message into multiple messages is done by means of the Splitter pattern. Beside of just generating a message for each containing element in the origin message, the Splitter is able to place elements into several outgoing messages. For example an order message contains an order number and the different order items. After splitting the elements into particular messages, each order item is covered in a separate message. A recipient of a message containing an order item cannot do anything with it because he does not know where to assign the order item. So the recipient additionally has to know the order number. Therefore a Splitter can duplicate particular elements and places the order number into each order item message.

Aggregator

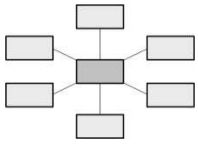


The Aggregator pattern gives an answer to the following question:

"How do we combine the results of individual but related messages so that they can be processed as a whole? – Use a stateful filter, an Aggregator, to collect and store individual messages until it receives a complete set of related messages. Then, the Aggregator publishes a single message distilled from the individual messages." [1]

A Splitter (described in the previous section) splits a message with multiple elements into several particular messages. The Aggregator pattern has reverse ambitions. In an integration solution a receiver gets a message and processes it. But often the receiving system needs more information for processing as it gets out of a single message. So it is appreciated that the receiver gets some collaborated information from one or from different senders. An Aggregator receives messages and analyzes them to find correlated messages. The Aggregator offers built in rules with defined conditions to recognize when a complete set of messages are arrived and an aggregated message out of the collected information of the particular messages can be generated. After that, the aggregated message is transferred to the recipient for further processing.

Message Broker

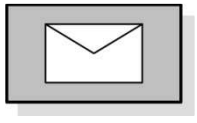


The Message Broker pattern gives an answer to the following question:

"How can you decouple the destination of a message from the sender and maintain central control over the flow of messages? – Use a central Message Broker that can receive messages from multiple destinations, determine the correct destination, and route the message to the correct channel. Implement the internals of the Message Broker using other message routers." [1]

A message-based integration solution has to deal with the routing of a message from the sender to the right receiver. Otherwise an integration solution focuses on a completely decoupling of the participating systems. So, how forwards a sender his message to the required receiver if he does not know the exact destination of the receiver? In the simplest case a sender and a receiver are connected via a simple Message Channel, where the sender knows only the Message Channel and not the destination of the receiver connected to the other end of the channel. But using one Message Channel for each connection can result in a very complex solution if the number of participating systems increases. Therefore another technique is used for connecting many systems to an overall integration solution, the Message Broker pattern (already described in Chapter 2.1.3.2). A Message Broker acts as a central mechanism to control the flow of messages between the different participating systems.

Envelope Wrapper



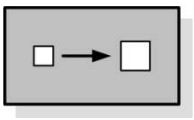
The Envelope Wrapper pattern gives an answer to the following question:

"How can existing systems participate in a messaging exchange that places specific requirements, such as message header fields or encryption, on the message format? – Us

an Envelope Wrapper to wrap application data inside an envelope that is compliant with the messaging infrastructure. Unwrap the message when it arrives at the destination." [1]

A message used in a message-based integration solution consists of two main parts, a header and a body. Whereas the header contains information about the routing of the message, the body contains the actual message content. But the information provided in the message header is often insufficient. Miscellaneous routing instances in an integration network often need specific information to route the message to the right destination, or e.g. the message should be encrypted to prevent an unauthorized access to the message data. Therefore an Envelope Wrapper is used to extend the original message with application-specific information. The additional data is added to the message like an envelope over the message. After sending the message to the desired destination, an Envelope Un-Wrapper removes the additional envelope from the message and the original message can be processed by the recipient.

Content Enricher

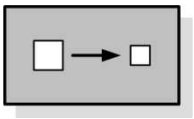


The Content Enricher pattern gives an answer to the following question:

"How do we communicate with another system if the message originator does not have all the required data items available? – Use a specialized transformer, a Content Enricher, to access an external data source in order to augment a message with missing information." [1]

In an integration solution the sending system sends a message to the receiving system and the receiving system processes the information provided by the sender. It is assumed that the sender bundles all data needed for processing by the receiver into the message. But what happens if the sender does not hold all needed data? For example, the sender creates a message with only a customer ID in it. But the receiver needs the exact name and address of the customer for further processing. So a Content Enricher pattern is used to fill up the missing data containing in another data source. The Content Enricher retrieves the entire name and address of a customer using the customer ID (e.g. by querying a database) and therefore the receiving system is able to process the request.

Content Filter

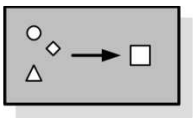


The Content Filter pattern gives an answer to the following question:

"How do you simplify dealing with a large message when you are interested only in a few data items? – Use a Content Filter to remove unimportant data items from a message, leaving only important items." [1]

The Content Filter pattern is the opposite of the described Content Enricher. Whereas the Content Enricher extends the information provided by the sender, the Content Filter reduces the provided information. The reason to remove data elements from a message is to protect the information from unauthorized usage. For security reasons a receiver should only get as much data as he needs for processing. Another reason to use a Content Filter is to minimize the message size and therefore to reduce the network traffic and the network load.

Normalizer



The Normalizer pattern gives an answer to the following question:

"How do you process messages that are semantically equivalent but arrive in a different format? – Use a Normalizer to route each message type through a custom Message Translator so that the resulting messages match a common format." [1]

A receiving system containing in an integration solution can have more than a single sending system. Each sender transfers a particular message to the recipient in their own specific message format. But the receiver does not understand all the different message formats of all senders and the processing of the messages is not possible. A Normalizer provides a mechanism to translate messages with different formats into messages provided in a common format. Now the receiver is able to process the translated messages.

Loosely Coupled Integration Solution

Out of the above described integration patterns Hohpe and Woolf [1] demonstrate how a loosely coupled integration solution can look like. Figure 2.18 shows the basic elements needed to create a message-based integration.

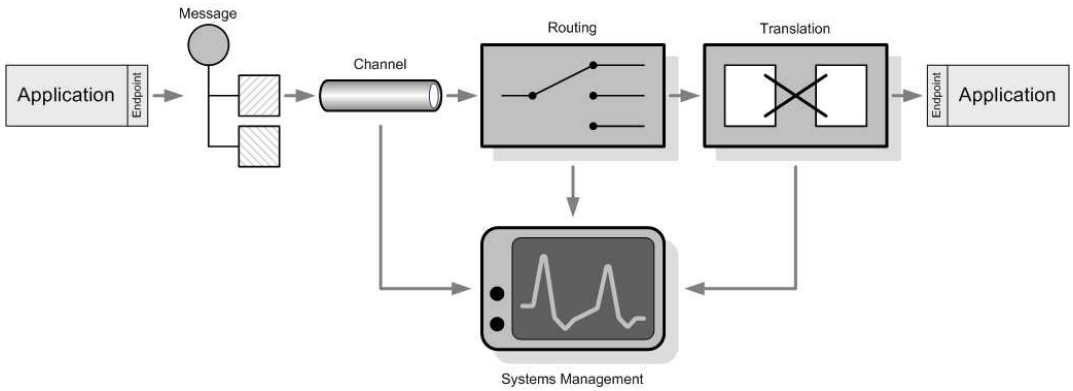


Figure 2.18: Loosely coupled integration solution

Chapter 3

RESEARCH QUESTIONS

3 Research Questions

In this chapter the research questions for this thesis, which will be discussed in the next chapters, is described. To build an integration solution out of multiple heterogeneous systems is often not easy to achieve. A developed integration framework has to handle numerous tasks and has to meet some conditions for the creation of an overall integration solution. Some of the conditions an integration framework has to achieve consist of:

- Loose coupling
- Easy installation and configuration
- Performance

In the following, the three listed conditions are described in detail in relation to their use in an integration framework.

A framework for the integration of heterogeneous systems should be able to perform the connection of the different legacy applications or systems of an organization in a loosely coupled manner. Loose coupling denotes the ability to change or remove a single system used in the integration solution without affecting the other participating systems. Therefore all other systems still continue working even though a specific system is not available anymore.

Another criterion an integration solution has to fulfill is an easy installation and configuration of the generated solution for the participating systems. For the integrated systems of the different customers it must be possible to easily and correctly prepare their systems to be used in the integration solution. Most suitable would be a self-configuring solution which can be deployed without the need of an expert. But for the developed SWIS approach a self-configuring solution without human decision is not possible, because of the safety-critical domain the SWIS system is used. Therefore the final decision to use and deploy the integration solution is done by a human.

Another important task for the development of an integration solution deals with the performance of the created solution. For a company it can be very crucial to get a high performance integration of their participating systems, in case if some of the containing applications or systems need real-time data for example in a client-server environment. Therefore the integration framework must pay special attention to this criterion.

3.1 *Evaluation of integration architectures*

Nowadays many different integration frameworks exist, which are developed in a variety of integration architectures. Generally, it cannot be said which architecture should be used for all areas by default, or which of the numerous existing integration technique presents the best of all. The finally used integration framework or architecture depends on the concrete scenario respectively the domain and is different from case to case. For a qualified comparison of the different integration architectures a set of predefined evaluation criteria is needed to get an appropriate result.

Aier, S., and Schönherr, M. [7] did a research on evaluating different integration architectures and specified numerous criteria for the comparison. These include initial planning efforts, initial development efforts, technical adaptations, non-invasive legacy/host integration, maintainability, and customizability. The single criteria are described in the next chapters. Later on in chapter 5.1, the developed SWIS approach (described in Chapter 4.1) for the integration of various systems is compared with other integration solution according to the six defined criteria. Therefore it will be determined:

According to the following defined evaluation criteria, in which of them is the developed SWIS integration approach better, worse or equal than other integration architectures, like individual interfaces, hub & spoke or service-oriented architecture (SOA)?

3.1.1 Initial planning efforts

This criterion focuses on the efforts needed for designing, implementing and processing the integration solution. Therefore the dimension on which the integration solution supports the internal staff and business policies are taken into account. Additional costs for authorizing external professionals to support the internal mandatory to realize an integration solution are attended by this criterion. External support is needed if the required arrangements cannot be handled internally for some cases. The initial planning efforts focus on the initial phase of an integration solution development.

3.1.2 Initial development efforts

While the previous described criterion focused on the planning efforts during the initial phase, this criterion focuses on the efforts for developing the integration technology, whereas also the initial development efforts during the initial phase are considered. The initial development efforts for developing an integration architecture depends on the use of already available software products. The more standardized software products are used, the lesser is the effort for the initial development. If it is decided to develop the integration architecture from the scratch, the efforts will be accordingly high. As a result to gain an individual integration, a high effort for the initial development must be accepted. By this criterion the overall costs to realize an integration project are evaluated, without differing between the amount of software coding or customizing.

3.1.3 Technical adaptations

If the requirements for an integration are altered, but the technical requirements are already realized in the developed integration architecture, the modifications of the requirements have an effect on the integration technology architecture too. An integration technology should allow the modification of the used requirements without much effort on changing the realized integration architecture to fulfill the new requirements. This criterion focuses on the development efforts needed to correct the available integration technology architecture after changing the requirements.

3.1.4 Non-invasive legacy/host integration

In companies a respectable amount of different legacy applications is running. These monolithic applications are intended to collaborate together. But the integration of legacy applications into an overall integration solution is often a real problem. Because of the limited control of legacy applications, the attended systems cannot be arbitrary customized to fulfill the needed conditions for the use in an overall integration. There is practically no chance for the integration developers to alter such enclosed applications. Furthermore, the most legacy applications provide no sufficient interface or no standardized interface description for the connection with other applications. Numerous companies using such legacy applications are not willing to replace these applications due to the risks which can come with the replacement, like complexity of the new systems, or less experience to the stability of them. This criterion focuses on the ability of an integration architecture to support the integration of unchangeable legacy applications.

3.1.5 Maintainability

A developed integration architecture should be maintained regularly to keep the generated integration solution up-to-date. Therefore the integration architecture has to provide the ability for maintenance without much effort. This criterion focuses on possibly available administration tools or monitoring facilities to maintain the integration architecture and additionally focuses on the efforts needed to arrange qualified maintenance.

3.1.6 Customizability

The customization of an integration architecture is associated with the previous described technical adaptations criterion. Whereas the technical adaptation focuses on the changes of integration technology if the requirements are modified, the customizability focuses on requirement modifications which do not affect the technical structure of the integration architecture. Such changes of the requirements should be updated by only customizing the existing integration solution, without deep restructuring of the elementary

integration technology architecture. This criterion addresses the degree of customization for the integration architecture.

3.2 *Case Study*

The Model Transformation Algorithm (MTA – see Chapter 4.2) is a core part of the developed SWIS approach and takes the task to create a solution model by using the requirement models. The solution model acts as a configuration set for the finally integration solution. So the MTA process should get special attention and the single MTA process steps should be supported by a tool to offer the possibility of human control during the creation of the solution model. Therefore a graphical user interface for the MTA process (see Chapter 4.3) was developed to guide the creation of the solution model step by step.

To determine the benefits of the developed tool support compared to manual enactment, a case study was performed. The instructions to the case study are specified in Chapter 4.4. The detailed execution and the obtained results of the performed case study are described in Chapter 5.2. Therefore by means of the case study it will be determined:

Which benefits have the automatic steps of the tool support for the MTA process, in relation to manual enactment, for gaining a consistent and correct solution model out of the input requirement models?

3.3 *Comparison of SWIS with traditional MDA*

The SWIS approach uses semantic models which contain all relevant data and needed information (e.g. message type, communication mode: push or request/reply, needed external services, converters, etc.) about the integrated applications and systems. The semantic models are expressed by means of ontologies and are defined in a layered manner. Furthermore, within the SWIS approach the semantic models are transformed into intermediate models for further processing. Thereafter, the intermediate models are transformed again into an applicable solution model. In contrast, a traditional MDA process (as described in Chapter 2.2) also uses a layered architecture, consisting of different models varying in their granularity and abstraction level. According to the layered structure of the two approaches, similarities between SWIS and MDA exist.

To find similarities between the developed SWIS approach and traditional MDA processes, a comparison of the basic design of the two approaches should be done. Therefore the following research question is asked:

Which similarities has the developed SWIS approach compared to traditional MDA processes, according to the underlying structure and the various used models?

3.4 *SWIS Integration Patterns*

An integration pattern provides a reliable and approved technique to realize a specific function by capturing the knowledge of experts who are familiar in a field the pattern stands for. Numerous integration patterns are defined and they are used if no simple “straight-forward” solution for a specific problem exists, due to the diversity of each domain.

Some of the numerous available integration patterns or rather enterprise integration patterns, defined by Hohpe and Woolf [1], are already described in Chapter 2.4. These integration patterns are also used for the development of the SWIS approach and therefore it is interesting to analyze:

How are the different integration patterns, like Message Translator, Publish-Subscribe Channel, Command Message, Request-Reply, Return Address, Correlation Identifier, Dynamic Router, Recipient List, Splitter, Aggregator, Message Broker, Envelope Wrapper, Content Enricher, Content Filter and Normalizer, realized in the developed SWIS approach?

Chapter 4

PRACTICAL WORK

4 Practical Work

This chapter describes the developed System Wide Information Sharing (SWIS) approach for the integration of numerous heterogeneous systems by using ontologies as information store. Also the SWIS architecture and the single ontologies used in SWIS are described. After the introduction to the SWIS approach, the Model Transformation Algorithm (MTA) is explained. The MTA is a core part in SWIS and accomplishes the task of processing the given information of the ontologies into a consistent and correct configuration set to use for the integration solution. For the description of the SWIS approach and their capabilities, some internal documents of the scientific project were used, like [47], [48] and [49]. Later, an introduction to the performed practical work more precisely the graphical tool support for the MTA is given. The tool supports the underlying MTA process by providing a consistent user interface to control and execute the single MTA process steps. Furthermore it will be described how the tool is operated by a human. At last an instruction to the performed case study to elicit the benefits of the tool support for the MTA process is given.

4.1 *System Wide Information Sharing (SWIS)*

In this chapter an introduction to the SWIS approach takes place. SWIS stands for System Wide Information Sharing and provides a promising approach to integrate a large number of heterogeneous systems. SWIS was developed in a scientific research project in the air traffic management domain in cooperation with the Austrian company Frequentis AG. SWIS helps designing a network for safety-critical data exchange between data provider and data consumer services between several organizations having heterogeneous requirements and/or capabilities.

In safety-critical environments like in the air traffic management domain (see Figure 4.1), it is a crucial task to get data from reliable and failure safe information systems which are processed by decision makers. SWIS was developed to replace the traditional point-to-point data integration solutions which are very reliable but also very time-consuming and cost-intensive when regular changes in the integration system occur (see Chapter 2.1.3.1 for explanations of point-to-point integration). The SWIS approach should compensate such disadvantages by providing a flexible integration framework where changes in the integration system are easy to handle and new system parts can be implemented without much effort.

In SWIS the different stakeholders act as data consumers or data providers, where some of the consumers can be seen as decision makers. For example an air traffic flow manager has the task to plan air traffic sequencing, re-routing and collision prevention. It is obvious that these decisions need correct data which must be available in real-time.

Another fundamental point is the integration of new data sources with the used legacy applications in the network. In such a safety critical domain the addition of new data sources leads to a verification of the resulting system in order to check the quality of the new integrated system.

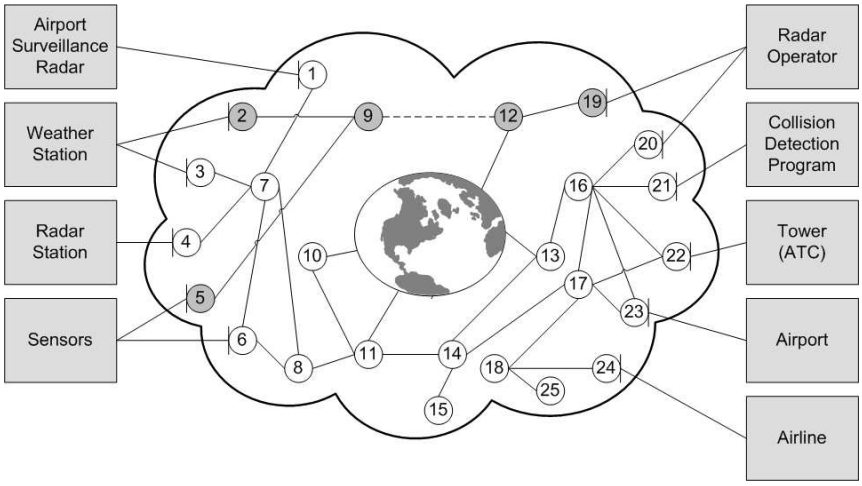


Figure 4.1: Air Traffic Management Network Structure

Figure 4.1 shows a basic scenario of an air traffic management and is used to explain the main tasks a decision maker has to do using the integration solution. The left side covers some data providers (e.g. Airport Surveillance Radar, Weather Station, Radar Station, various Sensors, etc.) and the right side contains data consumers (e.g. Radar Operator, Collision Detection Program, Air Traffic Controller (ATC) in the tower, etc.). The data providers and consumers are combined by an information sharing network which has several nodes and links. Nodes can have different properties: while the grey nodes in Figure 4.1 provide a secure connection, the white nodes do not provide any security features.

The developed integration system has to fulfill some essential requirements:

- First, due to a time- and safety-critical domain, it is very crucial that the network must work in a controllable and deterministic manner. Every decision made in the integration system (e.g. routing between two nodes) has to be evaluated and verified before deployment for error prevention. If possible errors in the system are not discovered before deployment a fault can lead to an unexpected behavior during run time and can implicate unthinkable consequences.

- Second the configuration of the network should allow that all requirements of the participating communication partners are take into account while deriving the possible integration partner candidates (e.g. specific time constraints for data which must not be exceeded by the routing of two communication partners). Furthermore the network configuration should be configurable from a wide-ranging perspective instead of a local perspective for optimizing the underlying network infrastructure and message routing.

- Third the integration system has to provide logical communication links to abstract the internal generated communication flow (e.g. middleware technologies) for a sufficient flexibility to enhancements.

- And fourth, the system has to offer a semantically routing between providers and consumers. More precisely, to determine all providers for a specific consumer they satisfy the requirements of the consumer (e.g. the right message type and context).

All these requirements expected from an integration system can be fulfilled by the SWIS approach. SWIS offers a modern platform to efficiently and correctly integrate dozens of different legacy applications. In SWIS the entire information about the underlying network and the participating systems with their specifications are described in explicit data models. Out of the defined data models a system configuration plan (solution model) is generated which covers the integration solution for satisfying the stakeholder quality requirements.

The next chapters cover some technical details about the developed SWIS approach like the used architecture and the data models in terms of ontologies.

4.1.1 SWIS architecture

Figure 4.2 displays the architecture used in the SWIS approach. The SWIS approach is divided into two main categories: the design time and the run time. The design time involves all efforts to specify the data models (ontology) as input for the integration solution, the actual Model Transformation Algorithm (MTA) for transforming the specified data models into expressive intermediate models (solution model) for later realization of the integration solution, and the simulation of the MTA-generated solution model to verify the correct behavior regarding to the original specifications. The run time involves the deployment of the generated and simulated solution model from the design time, lab testing and regular monitoring and auditing of the deployed integration solution.

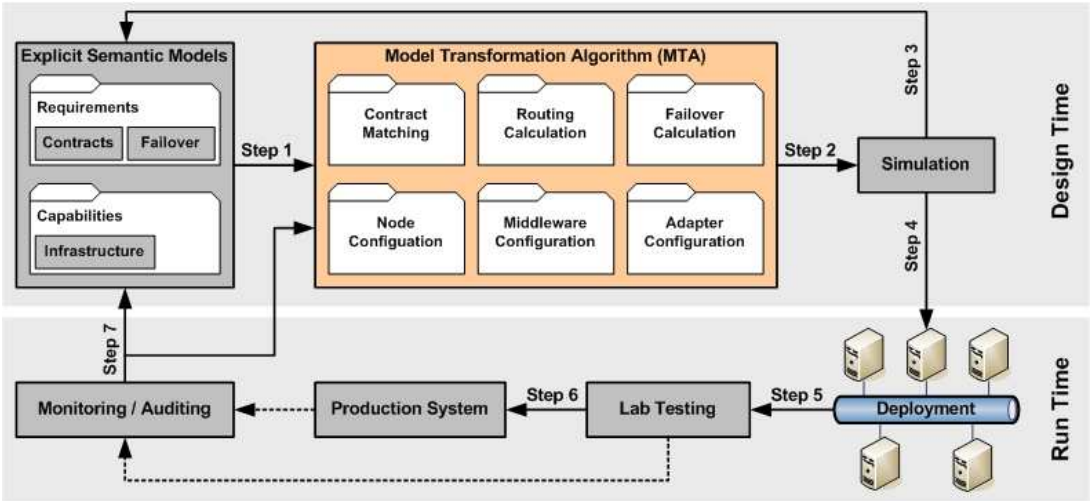


Figure 4.2: SWIS architecture

4.1.1.1 Design time

The design time contains the efforts to specify the requirements and capabilities. It primarily consists of the Model Transformation Algorithm (MTA) where the defined requirements are transformed to a valid solution model for the integration system. Additionally the design time contains a simulation process where the evolved solution model can be simulated and verified.

Explicit Semantic Models

The first task in the SWIS approach is the definition of the stakeholder requirements and capabilities. These requirements and capabilities are expressed via explicit models in a semantic manner and describe the underlying network infrastructure, the business policies, the failover contracts and all presented participating legacy applications (referred to as collaboration contracts). The infrastructure model contains the definitions of the existing nodes on the network, including their technical capabilities (e.g. supported network protocol, delay time, costs) and their connection to other nodes on the network. The contract model describes the semantic specifications of the available messages like message type and the contained message segments. It also defines the communication conditions between two collaborated services and specifies the collaboration capabilities (e.g. timeout of the collaboration, routing characteristics of a message). In general a message can be either a produced or a consumed message. With the policy model, numerous conditions for the generated integration solution are defined (e.g. restrictions to introduce by the route calculation). And at last the failover model contains all agreements for providing an adequate error handling (e.g. maximum number of calculated backup routes for the collaboration between two nodes).

With these created models the entire network used for the integration solution and their communication among each other as well as all possible collaborations between particular nodes are characterized.

Model Transformation Algorithm (MTA)

Generally the Model Transformation Algorithm builds the core part of the SWIS approach. The MTA receives the explicit semantic models as input and generates an intermediate model (the SWIS solution model) for the integration solution. Therefore the MTA has to find a route between a provider and a consumer service considering the matching of their group properties and fulfilling all collaboration specific demands of the both services. Furthermore the MTA determines backup routes for the defined collaborations to provide rerouting of messages if a part of the SWIS network used for communication drops out. The MTA offers the possibility to specify some conditions all calculated routes must fulfill like maximum cost or delay time. More detailed information about the functionalities and benefits of the Model Transformation Algorithm will be described later in Chapter 4.2.

Simulation

After the generation of the solution model by the MTA, a simulation can be started to test the generated solution model. During the simulation phase, the solution model will be

tested and it will be checked whether the calculated solution correctly describes the defined functionalities and capabilities of the requirements before the solution can be deployed to a real-world environment. If the solution model does not correctly conform to the requirements, the process goes back to the first step of the SWIS approach. The explicit semantic models are reviewed and necessarily modified. After the revision of the semantic requirement models a new solution model is calculated with the MTA and will be tested again in the simulation phase.

4.1.1.2 Run time

The run time contains all steps to adopt the SWIS solution model from the design time in the real environment. This involves deployment and testing of the integration solution and continuous monitoring and auditing of the running production system.

Deployment

The deployment of the generated solution model means, to distribute the solution to all participating SWIS nodes within the SWIS network. Therefore, the functionalities of each node will be configured for normal operation as well as for malfunction operation in case of node failures. A new SWIS solution model will be generated and again deployed to all SWIS nodes if the semantic models are significantly modified. Such modifications encompass adding new nodes, links or services into the explicit semantic models.

Lab Testing

The deployed SWIS solution model can be tested in a lab environment before the solution will be deployed to the real production system. Without testing, potential variations of the solution model towards the requirement descriptions can lead to a malfunction of the production system. Therefore the generated solution model will be systematically tested in a lab environment to find possible deviations.

Production System

The production system represents the real world environment of the SWIS network. As last step, the generated, deployed and tested SWIS solution model is used in the production system to achieve an overall integration system.

Monitoring/Auditing

The developed integration solution created with the Model Transformation Algorithm must be continuously monitored and audited after deploying the solution in a real world environment. Monitoring and Auditing delivers important information about the integration solution generated with the SWIS approach. With monitoring it is feasible to get feedback to the current system states in terms of measurement data which can be used for comparison with predefined values. The gained feedback from monitoring helps to improve the quality of the whole SWIS approach, by measuring actual technical performance and capacity data.

4.1.2 SWIS ontologies

SWIS uses ontologies as information mapping enablers and therefore the ontologies contain all relevant data and information (e.g., message type, communication mode: push or request/reply, external services, converters) about the applications and systems to integrate (ontologies are already described in Chapter 2.3).

The SWIS ontologies provide the requirements and capabilities for the calculation of the integration solution model. The ontology input models uses a subdivided architecture of three different types of ontologies, to separate the knowledge into various granularity levels. The different ontologies in the SWIS approach are the abstract ontology, the domain ontology and the customer ontology. Whereas the customer ontology extends the domain ontology and the domain ontology extends the abstract ontology (see Figure 4.3). In SWIS the ontologies are written in OWL (Web Ontology Language) because of its advantage of handling relations between classes, properties and individuals compared to other languages. The three used ontologies are described in the next chapter.

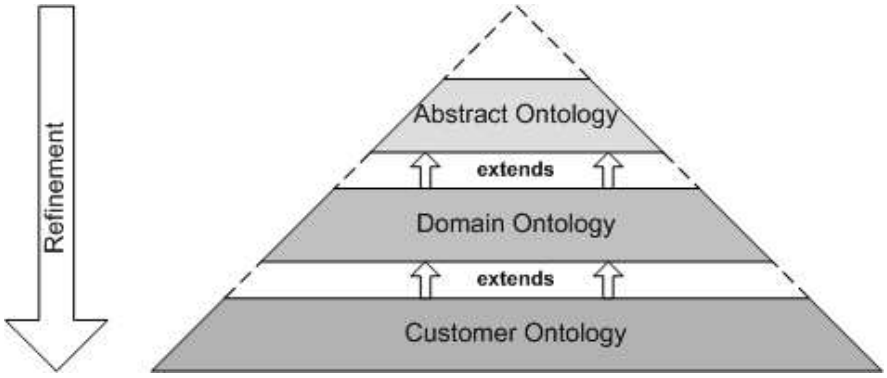


Figure 4.3: The three different types of ontologies used in SWIS

In the SWIS approach, a network consisting of numerous nodes and links presents the major component. Each node has a specified network address and is connected to another node. A link connects two nodes together and is used as transport medium to send messages between the nodes. A node supports different data transfer protocols and can – but must not – run one or more services on it. Services are classified as consumer service (only receives messages), provider service (only sends messages) or request/reply service (receives and sends messages). Afterwards the different parts used in a SWIS network (Node, Link, Protocol, etc.) are described.

Node

A node illustrates a physical component (e.g. server or host) in a network. But a SWIS node does not represent exactly one server or host. This means that for example one host can hold numerous SWIS nodes which are running on it. Furthermore each SWIS node has to support one or more network protocols (e.g. TCP/IP, SMTP, HTTP, etc.). If a node does not support any protocol, the node is not ready for a connection using the network. A SWIS

node also has the ability to work in multiple physical networks. Therefore the node needs a separate network address for each network. Additional a node can have individual attributes (e.g. delay-time, capacity, security, etc.) defined as class “Attribute” which is described later.

A unique functionality in the SWIS approach is the use of Shadow Nodes for supporting redundancy of SWIS nodes. For each node numerous Shadow Nodes can be defined. A Shadow Node has the same properties and attributes as the node they represent, but has different network addresses. If a node drops out a redundant Shadow Node is used to maintain the further communication in the SWIS network.

Link

A link illustrates a physical network connection (e.g. wired network connection or wireless network connection) between two SWIS nodes. Two connected nodes can only communicate together if both nodes understand at least one common network protocol. Two nodes are only associated with a SWIS link if they are able to communicate together even though they are connected via a physical connection. Otherwise two nodes can have more than one SWIS link if they support multiple common protocols. Out of all possible common communication protocols of two nodes, the most applicable protocol for a given scenario is used.

Each defined link is described with a start node and an end node. But the definition of start and end node does not represent a direction for the connection. A SWIS network is not constructed in a direction-oriented manner. So it has no effect to the network if the start and end node are swapped. To provide additional attributes for a link, like delay time of the link or security concerns, the Attributes concept will be used (Attributes are described later in this chapter).

Network

A SWIS network consists of numerous SWIS nodes and SWIS links. Each node holds a network address which is usually equal for all nodes in a common network. A network is defined by a unique network name. In case that a particular node is part of more than one network, this node is called a gateway node. If a link connects a node from one network with a node existing in another network, the link is called gateway link.

Protocol

Each SWIS node supports one or more protocols. A protocol is comparable with a language the node is speaking. If two different nodes support at least one common protocol they are able to communicate together. For each common protocol supported by two connected SWIS nodes one separate SWIS link exists. Therefore a protocol controls or rather enables the connection and communication between a start and end node pair. Furthermore a protocol also is responsible for the data transfer from one node to the other node. If a SWIS node does not support at least one protocol, a communication with this node is not possible. A protocol is defined with a unique name which is similar for all nodes supporting this protocol. Furthermore a protocol can have additional attributes, like security conditions or delay-time specifications.

Network Address

Numerous SWIS nodes are bundled within a SWIS network. To allocate these nodes to a common network a network address is used. Such a network address (among other things an IP address) is assigned to each node and is unique for each node in one common SWIS network. Within one network it is not possible that two nodes with the same network address exist. Usually a node has just one network address, but if a node acts as a gateway node it can have multiple network addresses. Each network address is defined by two attributes: the actual network address represented as a string (e.g. “192.168.0.1”) and the network the address is containing.

Service

Services represent communication devices available on legacy applications. A service is running either as provider or as consumer service and provides or consumes messages. In a SWIS network any service is connected to exactly one node, whereas a node can hold numerous different services. A message transferring inside the SWIS network has a clear specified format, the so-called message type (e.g. OrderMessage, InfoMessage). Each service supports exactly one message type.

Message

As already described in the previous section, a message can either be provided or consumed by a service. A message has a segmented design, and must have at least one segment. Each message contains a message header, where the conditions to successfully transport the message over the network are specified. Such conditions encompass for example the needed protocol for transportation, delay-time conditions or security terms, and are represented separately via a segment. Each segment is defined as a particular envelope of the message. This means that for each segment the original message is extended with the segment envelope.

Attribute

By means of an attribute, to each specific requirement or capability a precise value is defined. This can be a requirement of a contract (e.g. delay-time), or a capability of a node, protocol or link (e.g. security term).

Shadow Node

As already described in the Node section, a Shadow Node is a complete duplication of the Node it belongs to. A Shadow Node provides the same properties and attributes of the original Node. The only difference is that the Shadow Node uses another network address. If the primary Node drops out, the Shadow Node takes over the further communication by maintaining the dataflow instead of the original Node. The change from the primary Node to the underlying Shadow Node is fully transparent to the other network devices.

Legacy Application

In computer science, a legacy application is an isolated application or system, which acts as a single item which provides almost no public interface for the integration with other

applications. These applications communicate in a various way with other devices and do not underlie the scope of the SWIS application. But legacy applications should also be integrated into the SWIS integration solution. Therefore the legacy applications are added as independent devices with a unique name and providing at least one service.

Logical Group

Numerous legacy applications can be summarized into single logical groups to represent common physical or logical relations of the containing legacy applications. Each logical group in SWIS offers a unique name to exactly identify the group, and consists of at least one legacy application. Otherwise, without a legacy application, a logical group is not able to exist.

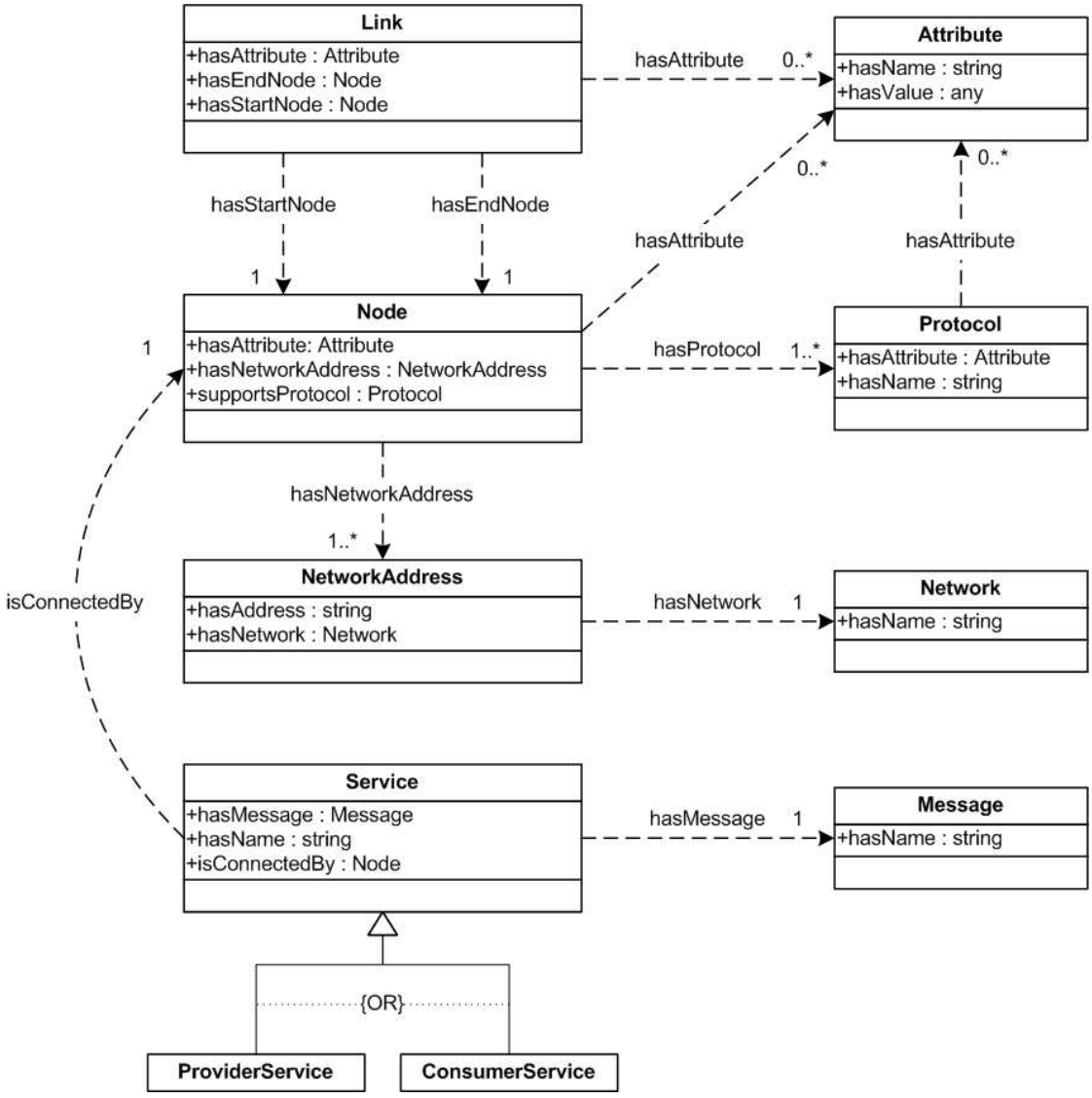


Figure 4.4: Excerpt of the Classes used in the SWIS ontology

T-Map

A transformation-map (short T-Map) has some different tasks to achieve within the SWIS approach. A T-Map is primarily used to transform the segments of a message from the source format into the needed target format. Therefore a T-Map can use specific converters or actually can call external services, e.g. to get some additional data from various data sources.

Converter

With a converter a message segment can be converted from one basic data type into another data type. Converters are used by the before described T-Maps for the transformation of a source message format into a target message format.

Figure 4.4 shows the structure of the SWIS ontology used for the arranged case study (see Chapter 4.4 for more information to the case study). The figure shows all previous described parts of a SWIS network, but represents only an excerpt of the entire developed SWIS ontology.

In the next chapters the three different ontologies (abstract, domain and customer ontology) used for the SWIS approach as information source, are described.

4.1.2.1 Abstract Ontology

The abstract ontology encompasses the basic concepts for a SWIS-based scenario. For the SWIS approach this ontology holds the concepts for the integration of different legacy applications in an Air Traffic Management Area. The abstract ontology also includes the concepts for modeling the infrastructure of the application domain. Because of its domain independent representation the abstract ontology can be used across different domains. The dissociation of domain characteristics by means of domain independent ontologies presents a powerful mechanism for a flexible and easy to adopt basis to use them in different information sharing scenarios, completely decoupled of the underlying domain. To use the abstract ontology within another domain, only the domain ontology needs to be replaced. All concepts containing in the abstract ontology are defined in a conceptual way to achieve a simple and straightforward usage of the abstract ontology in different domains. For example an instance of the node class can either be a discrete network node or an intersection of two links.

In an abstract ontology numerous concepts are defined and therefore they are grouped into functional segments. These functional segments can be classified into infrastructure concepts (node, link, network, protocol, network address), service and message concepts (service, message), policy and contract concepts (attribute), and message transformation and conversion concepts (T-Map, converter). Each concept, described by the abstract ontology, consists of a unique ID used to exactly access and query the wanted component. This unique identifier is expressed textually to make it human-readable and therefore offers better and easier human maintenance.

The abstract ontology was created with Protégé and an explanation of the generated code is shown below. The listings only show the fundamental structure of the ontology code. A complete listing of the abstract ontology is covered in the appendix. In general the ontology contains three different types: Classes, ObjectProperties, DatatypeProperties. These three types of the abstract ontology are described following in terms of code listings.

Class

A class represents a precise concept and contains all defined properties of the concept (e.g. Service, Node, Link, etc.). Listing 4.1 shows the definition of a class in SWIS written in the Web Ontology Language (OWL).

```
<owl:Class rdf:ID="ClassName">
  <owl:equivalentClass>
    <owl:Class>
      <!-- collection of the different properties in a class -->
      <owl:intersectionOf rdf:parseTyp="Collection">
        <!-- defines a ObjectProperty -->
        <owl:Restriction>
          <owl:onProperty>
            <owl:ObjectProperty rdf:ID="ObjectPropertyName"/>
          </owl:onProperty>
          <!-- defines a min. cardinality of the ObjectProperty (min 0) -->
          <owl:minCardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
          >0</owl:minCardinality>
        </owl:Restriction>
        <!-- defines a DatatypeProperty -->
        <owl:Restriction>
          <owl:onProperty>
            <owl:DatatypeProperty rdf:ID="DatatypePropertyName"/>
          </owl:onProperty>
          <!-- defines a cardinality of the DatatypeProperty (exactly 1) -->
          <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
          >1</owl:cardinality>
        </owl:Restriction>
      </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>
```

Example:

```
<owl:Class rdf:ID="Service">
  <owl:equivalentClass>
    <owl:Class>
      <owl:intersectionOf rdf:parseTyp="Collection">
        <owl:Restriction>
          <owl:onProperty>
            <owl:ObjectProperty rdf:ID="hasMessage"/>
          </owl:onProperty>
          <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
          >1</owl:cardinality>
        </owl:Restriction>
        <owl:Restriction>
          <owl:onProperty>
            <owl:DatatypeProperty rdf:ID="hasName"/>
          </owl:onProperty>
          <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
          >1</owl:cardinality>
        </owl:Restriction>
        <owl:Restriction>
          <owl:onProperty>
            <owl:ObjectProperty rdf:ID="isConnectedBy"/>
          </owl:onProperty>
          <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
          >1</owl:cardinality>
```

```
        </owl:Restriction>
      </owl:intersectionOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>
```

Listing 4.1: Abstract Ontology: Class definition

ObjectProperty

ObjectProperties have another class as type, e.g. ObjectProperty *hasAttribute* where class *Attribute* represents the type of the property, or *hasNetwork* with type *Network.* The ObjectProperty is comparable with a relation of two classes in a traditional relational database system. In Listing 4.2 an example to highlight the structure of an ObjectProperty definition is shown.

```
<owl:ObjectProperty rdf:ID="ObjectPropertyName">
  <!-- the class which contains the ObjectProperty -->
  <rdfs:domain rdf:resource="#SourceClassName"/>
  <!-- the class which represents the type of the ObjectProperty -->
  <rdfs:range rdf:resource="#TargetClassName"/>
</owl:ObjectProperty>


<owl:ObjectProperty rdf:ID="ObjectPropertyName">
  <!-- all classes which contains the ObjectProperty -->
  <rdfs:domain>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#SourceClassName1"/>
        <owl:Class rdf:about="#SourceClassName2"/>
      </owl:unionOf>
    </owl:Class>
  </rdfs:domain>
  <!-- the class which represents the type of the ObjectProperty -->
  <rdfs:range rdf:resource="#TargetClassName"/>
</owl:ObjectProperty>


Example:

<owl:ObjectProperty rdf:ID="hasAttribute">
  <rdfs:domain>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Node"/>
        <owl:Class rdf:about="#Link"/>
        <owl:Class rdf:about="#Protocol"/>
      </owl:unionOf>
    </owl:Class>
  </rdfs:domain>
  <rdfs:range rdf:resource="#Attribute"/>
</owl:ObjectProperty>
```

Listing 4.2: Abstract Ontology: ObjectProperty definition

DatatypeProperty

DatatypeProperties have a simple data type as type, e.g. DatatypeProperty *hasName* with datatype *string.* The DatatypeProperty is comparable with an attribute in a traditional relational database system. See Listing 4.3 for an example to define a DatatypeProperty.

```
<owl:DatatypeProperty rdf:ID="DatatypePropertyName">
  <!-- the class which contains the DatatypeProperty -->
  <rdfs:domain rdf:resource="#SourceClassName"/>
  <!-- the datatype of the DatatypeProperty -->
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>


<owl:DatatypeProperty rdf:ID="DatatypePropertyName">
  <!-- all classes which contains the DatatypeProperty -->
  <rdfs:domain>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#SourceClassName1"/>
        <owl:Class rdf:about="#SourceClassName2"/>
      </owl:unionOf>
    </owl:Class>
  </rdfs:domain>
  <!-- the datatype of the DatatypeProperty -->
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>


Example:

<owl:DatatypeProperty rdf:ID="hasName">
  <rdfs:domain>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Service"/>
        <owl:Class rdf:about="#Attribute"/>
        <owl:Class rdf:about="#Message"/>
        <owl:Class rdf:about="#Protocol"/>
        <owl:Class rdf:about="#Network"/>
      </owl:unionOf>
    </owl:Class>
  </rdfs:domain>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>
```

Listing 4.3: Abstract Ontology: DatatypeProperty definition

4.1.2.2 Domain Ontology

The domain ontology is an extension of the before described abstract ontology (see Figure 4.3) and precisely specifies the SWIS network. This is achieved by adding individuals to define the elements which represent the infrastructure of the underlying SWIS network. An individual is a concrete instance of the concepts defined in the abstract ontology. In addition the domain ontology also provides the concepts and therefore the classification for domain-specific knowledge.

The domain ontology contains the main information of the stakeholders for the particular domains and is used for modeling standardized domain-specific information. To achieve interoperability between different systems, the various customers relate their customer-specific information (defined in the customer ontology – described in the next chapter) to the standardized information of the domain ontology. Therefore, the single customer systems are able to interoperate with other systems. The domain ontology also contains all relevant information required to build a SWIS-based integration solution. With the information defined in the domain ontology, it is possible to find semantically equivalent information which is either provided or consumed by the different systems. Semantically

equivalent data must not obligatory have the same format, but it depends on the data content to have the same meaning. If two services in SWIS support a semantically equivalent provider/consumer message pair, the two services are combined to a common collaboration. The domain ontology can easily be used in other SWIS-based integration solutions, in case if the integration solution is in the same domain. Therefore all new applications of a single domain benefit from the available domain ontology.

The domain ontology normally is created by the domain expert, supported by the network administrators of the networks which should be integrated, whereas the network administrators are responsible for the description of the network infrastructure. In addition the domain expert has to maintain and refine the ontology. The various Subject Matter Experts (SMEs) of the participating organizations or legacy applications are responsible to identify the provided or consumed data. First the domain concepts are described and then the network infrastructure consisting of nodes, links, protocols, networks and network addresses is described.

Relation to the abstract ontology

```
<owl:Ontology rdf:about="">
  <owl:imports rdf:resource="http://www.tuwien.ac.at/abstract.owl"/>
</owl:Ontology>
```

Listing 4.4: Domain Ontology: relation to the abstract ontology

Instance of class “Node”

```
<swis:Node rdf:ID="NodeName">
  <swis:hasAttribute rdf:resource="#AttributeName"/>
  <swis:hasNetworkAddress rdf:resource="#NetworkAddressName"/>
  <swis:supportsProtocol rdf:resource="#ProtocolName"/>
</swis:Node>
```

Listing 4.5: Domain Ontology: instance of class "Node"

Instance of class “Link”

```
<swis:Link rdf:ID="LinkName">
  <swis:hasAttribute rdf:resource="#AttributeName"/>
  <swis:hasEndNode rdf:resource="#TargetNodeName"/>
  <swis:hasStartNode rdf:resource="#SourceNodeName"/>
</swis:Link>
```

Listing 4.6: Domain Ontology: instance of class "Link"

Instance of class “Protocol”

```
<swis:Protocol rdf:ID="ProtocolName">
  <swis:hasAttribute rdf:resource="#AttributeName"/>
  <swis:hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
  >TCP</swis:hasName>
```

```
</swis:Protocol>
```

Listing 4.7: Domain Ontology: instance of class "Protocol"

Instance of class "Attribute"

```
<swis:Attribute rdf:ID="AttributeName">
  <swis:hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
  >secure</swis:hasName>
  <swis:hasValue rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
  >false</swis:hasValue>
</swis:Attribute>
```

Listing 4.8: Domain Ontology: instance of class "Attribute"

Instance of class "NetworkAddress"

```
<swis:NetworkAddress rdf:ID="NetworkAddressName">
  <swis:hasAddress rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
  >10.0.0.1</swis:hasAddress>
  <swis:hasNetwork rdf:resource="#NetworkName"/>
</swis:NetworkAddress>
```

Listing 4.9: Domain Ontology: instance of class "NetworkAddress"

Instance of class "Network"

```
<swis:Network rdf:ID="NetworkName">
  <swis:hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
  >WAN</swis:hasName>
</swis:Network>
```

Listing 4.10: Domain Ontology: instance of class "Network"

4.1.2.3 Customer Ontology

Finally the customer ontology specifies customer-specific information about the underlying network used for the integration solution. The customer ontology defines all legacy applications and the used message structures for the SWIS scenario. Furthermore, the business policies, i.e. the conditions for the connection of the various participating applications, are specified. This ontology is the most refined ontology and extends the domain ontology by including the information on how the single nodes exchange their data with other nodes. In the customer ontology all necessary information of the legacy applications or the participating organizations, with their existing and respectively used services and messages, is available. Therefore, the customer ontology contains essential information about the data exchange and the cooperation of the single legacy applications. For example, it contains the specification of the used messages if they are provided or

consumed by the legacy applications. Furthermore the semantic context as well as the format of each message segment is defined.

The subject matter expert (SME) creates the customer ontology and is responsible for the specification of the legacy applications. A SME gets help from the domain expert during the creation of the ontology and after creation the domain expert checks the completed ontology for correctness and completeness. In order to create the customer ontology, the SME has to specify the following information:

- The different services regarding to the participating legacy applications.
- The requirements and/or the capabilities of the specified services, also noted as service contracts.
- The physical mapping of the specified services to a specific network node, which is described in the domain ontology.
- The frequency to send or receive messages, for each defined service.

At next, the single messages used for the communication between different legacy applications are defined precisely. Furthermore all message segments of the data formats contained in the domain ontology, have to be described in detail. Also the detailed specifications for the physical network components (e.g. nodes and links - defined in the domain ontology) more precisely the mapping between services and nodes to establish the transmission of messages is defined in the customer ontology. So, all needed information for the SWIS integration solution exists.

At last it is mentionable to say that the two ontologies (domain and customer) are closely associated together. The border between them is not fixed and depends on the particular scenario and the user roles. It is possible to transfer some parts defined in the domain ontology into the customer ontology and vice versa.

Relation to the domain ontology:

```
<owl:Ontology rdf:about="">
  <owl:imports rdf:resource="http://www.tuwien.ac.at/domain.owl"/>
</owl:Ontology>
```

Listing 4.11: Customer Ontology: relation to the domain ontology

Instance of class “Message”

```
<swis:Message rdf:ID="MessageName">
  <swis:hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
  >InfoMessage</swis:hasName>
</swis:Message>
```

Listing 4.12: Customer Ontology: instance of class "Message"

Instance of class "ProviderService"

```
<swis:ProviderService rdf:ID="ProviderServiceName">
  <swis:hasMessage rdf:resource="#MessageName"/>
  <swis:hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
  >SendInfoMessage</swis:hasName>
  <swis:isConnectedBy rdf:resource="http://www.tuwien.ac.at/domain.owl#NodeName"/>
</swis:ProviderService>
```

Listing 4.13: Customer Ontology: instance of class "ProviderService"

Instance of class "ConsumerService"

```
<swis:ConsumerService rdf:ID="ConsumerServiceName">
  <swis:hasMessage rdf:resource="#MessageName"/>
  <swis:hasName rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
  >ReceiveInfoMessage</swis:hasName>
  <swis:isConnectedBy rdf:resource="http://www.tuwien.ac.at/domain.owl#NodeName"/>
</swis:ConsumerService>
```

Listing 4.14: Customer Ontology: instance of class "ConsumerService"

4.2 *The Model Transformation Algorithm (MTA) process*

The Model Transformation Algorithm (MTA) builds a core part of the SWIS integration approach (see Figure 4.2). The MTA uses the defined semantic models to calculate possible routes and to create a SWIS Solution Model which represents a configuration set for the integration of all containing devices (e.g. nodes, links, legacy applications, services, etc.) in an SWIS network. The MTA is divided into the following fundamental steps:

- ***Step 1:*** Preparation of semantic data comprised in the input models
- ***Step 2:*** Calculation of routes between provider and consumer services
- ***Step 3:*** Calculation of backup routes for each SWIS node
- ***Step 4:*** Creation of the SWIS solution model

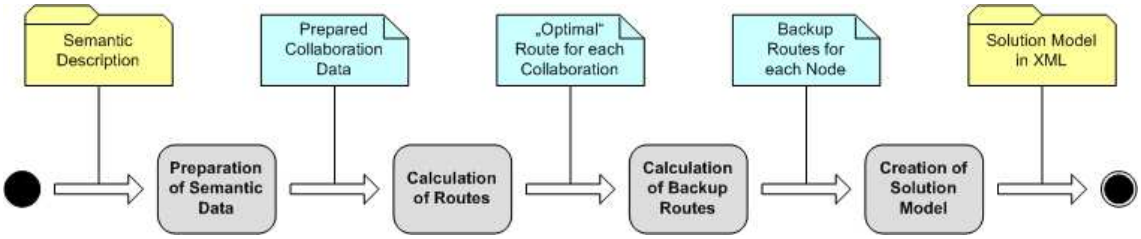


Figure 4.5: MTA process steps

The four steps of the MTA process are pictured in Figure 4.5. The figure also shows the input and output data of the single process steps. These data can be distinguished into data represented in files and data represented as internal data structures existent during the MTA process. Data represented in external files are the semantic description in terms of ontology models and the resulting solution model as XML file. Internal data structures are the prepared collaboration data which will be resolved as nodes, links, collaborations, services, and so on for the later calculation steps, the “optimal” route for each collaboration containing the routing information for a provider to a consumer service, and the backup routes for each node if the next node defined in a route is currently not available.

In the following sections the single steps of the MTA process are explained to get an overlook about the Model Transformation Algorithm technique.

4.2.1 Step 1: Preparation of Semantic Data

In this section the first step of the MTA process is described. It explains how the semantic data described in explicit semantic models will be prepared for the use in the SWIS integration approach. The semantic data represents the whole information of the underlying SWIS network containing all declarations for nodes, links, legacy applications, services, etc. (see Chapter 4.1.2 for more concepts declared within the explicit semantic models).

The main task for the preparation of the semantic data is to read all information provided in the ontology models (abstract, domain and customer ontology) and generate semantic data for processing in further steps. After this step a set of semantic collaborations out of the semantic models are created. A semantic collaboration describes the combination of at least one provider service with a consumer service. Whereas the provider service is able to exchange messages with the consumer service according to predefined conditions (e.g. maximum costs, maximum delay-time, common protocol, etc.). By means of the MTA it is possible to either automatically calculate all possible collaborations for a certain scenario out of the defined semantic models, or a user can select a subset of collaborations (as it will be used in the tool support, see Chapter 4.3.2) for further processing.

A collaboration used as major communication data in the SWIS integration approach encompasses numerous information data about

- provider and consumer services,
- message data transformations (so-called transformation maps or short T-Maps, described in Chapter 4.1.2) needed to establish the message exchange,
- used network nodes to connect the participating services,
- predefined conditions which must be satisfied to establish a connection of the involved services (so-called service contracts), and

- communication mode used for the connection (push or request/reply collaboration).

During the preparation of semantic data the semantic collaborations of the input models are collected and analyzed. After collection of the collaborations additional information is appended to the collaborations if necessary. The additional information will be derived from the semantic requirements and is needed to call external services for optional data transformations, to establish a communication inside a sender group or for multicasting of messages to a defined receiver group. This additional information is also represented in terms of collaborations. Also the correlation data of a request/reply collaboration needs to be stored to handle the flow of request message and reply message.

After the first step of the MTA process a set of enriched and refined collaborations (either all collaborations out of the semantic models or a subset of user-selected collaborations) which are extended with additional generated collaborations is available. This data serves as input for the next MTA process step whereas a route for each of the defined collaborations will be calculated.

4.2.2 Step 2: Calculation of Routes

In this section the second step of the MTA process is described. It explains how the primary routes for a SWIS-based scenario are calculated. This step uses as input a set of enriched and refined collaborations generated in the previous step of the MTA process.

For all collaborations a set of routes for further processing will be calculated. Therefore all possible routes through the network which fulfill the collaboration conditions (e.g. special network constraints, cost and delay-time conditions) are calculated. After the calculation of the routes, a mass of SWIS scenarios will be created. In a scenario the calculated routes are combined together whereas only a single route for each collaboration is used. The defined scenario represents a solution for the SWIS integration approach. For each possible constellation of combining the calculated routes of the different collaborations one separate scenario is defined. These scenarios are then filtered using a pareto optimization. Out of the mass of produced scenarios only those are stored which satisfy the pareto optimality.

Pareto optimality is *"… an economics term for describing a solution for multiple objectives. No part of a Pareto optimal solution can be improved without making some other part worse."* [46]

Figure 4.6 shows an example with different solutions regarding to their pareto optimality. As you can see in the picture out of all solutions only the blue solutions conform to be pareto efficient. These solutions are placed on the pareto curve, whereas on the one side the solution with the least costs and the highest delay is placed and on the other side the solution with the highest costs and the least delay is placed. So, no solution is the best and it must be decided which solution should be taken. If a solution with lesser delay is taken, higher costs must be accepted and if a solution with lesser costs is taken, higher delay must be accepted. All other solutions (the grey ones) illustrate a pareto inefficient state.

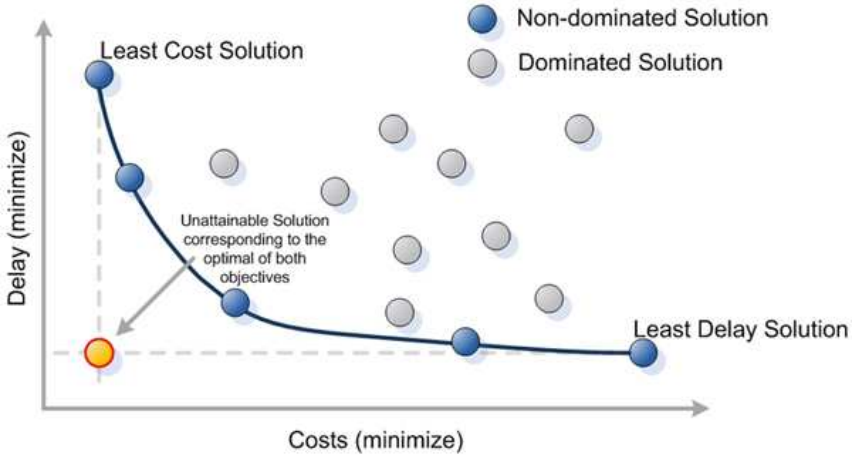


Figure 4.6: Pareto optimality of different solutions

After this process step a set of pareto optimal scenarios are generated whereas each of them containing the major routes of the defined input collaborations.

4.2.3 Step 3: Calculation of Backup Routes

In this section the third step of the MTA process is described. It explains how the backup routes for a SWIS-based scenario are calculated. As input for this step the generated data of the first two steps are used. This encompasses a set of enriched and refined collaborations and a set of pareto optimal major routes for these collaborations.

For each major route several possible backup routes are calculated. The use of backup routes represents an important technique to build a fault-tolerant and robust communication network. So it is possible to switch to an alternative route if a device (e.g. node or link) containing in the used main route fails. If a node wants to send a message to the next node which is defined by the used route but due to a failure the destination node is not available, the sending node switches to a backup route to finish the transmission of the message. Therefore the sender transfers the message not to the primary node (which is actually not available) but to another node according to the backup route. For each node in a SWIS network which is a member of the main route, several backup routes have to be calculated.

This MTA process step extends the output of the previous process steps by adding backup routes. Therefore the output after the first three MTA process steps contains pareto optimal scenarios for the defined collaborations enhanced with backup route information.

4.2.4 Step 4: Creation of Solution Model

In this section the fourth step – the last step – of the MTA process is described. It explains how the solution model for a SWIS-based scenario is created. The pareto optimal

scenarios for the defined collaborations enhanced with backup route information generated in the previous steps are used as input for this step. Out of the input data this MTA process step creates the SWIS solution model.

The finally created SWIS solution model encompasses numerous information data containing the following items:

- routing table for each SWIS node,
- naming service to establish abstract middleware-based message transmission,
- mapping instructions of the exchanged messages, and
- description of the transformation maps (short T-Maps)

The output of the last step in the MTA process is the SWIS solution model. This is a XML configuration file which will be deployed to each SWIS node containing in the SWIS network. In generally the solution model is a human readable representation of the generated integration solution used to configure the network to conform the selected scenario.

4.3 *Tool support for the MTA process*

This chapter describes the developed tool support for the MTA process. It provides a graphical user interface which helps the system integration engineer to find an optimized integration solution for a specific scenario. The tool provides an easy to handle step by step appliance from the ontology selection till the completed configuration to build the integration solution. The tool support represents a web application with JSP and Servlets and therefore the next chapter gives an overlook about the used web development technique. To set up the required environment to run the tool support, in Appendix A an installation guide is presented.

4.3.1 JSP and Servlets

Java Server Pages (JSP) and Java Servlets are a common way for building ambitious web applications. They allow designers and developers to rapidly embed dynamic content into web pages by using Java and a set of elementary markup tags.

- ***Java Server Pages (JSP):*** Java Server Pages provides a Java-based technology for developing dynamic web sites in a simplified manner [28].
- ***Servlets [3]:*** Servlets run in special parts of web applications the so-called web container. After deploying the web application into a web container, all instances of the java servlet class are loaded to the Java Virtual Machine (JVM) by the web

container. Afterwards the requests for the servlet can be handled. To build a java servlet class a java class must extend *javax.servlet.http.HttpServlet*.

4.3.2 Tool support process steps

In this section tool support for the Model Transformation Algorithm (MTA) is introduced. The GUI supports the selection process, i.e. to automatically derive if one or multiple producers match one or multiple receivers or vice versa. In addition, a java web application that renders a graphical representation of the desired Network Infrastructure Model can be started. The tool support for the MTA contains a user interface which provides predictable emergent properties of the integrated system. On the one hand, the visualization should lead to product improvement by the visual feedback, and on the other hand it should lead to process improvement by providing better tool support and quality assurance. The developed tool for the MTA has the task to help the system integration engineers to find a specific integration solution for a scenario. Furthermore the user interface should help the integration project manager because less time is needed to model, create and verify the integration solution. The tool is developed as a web application to make it possible to access the tool over the Internet, and is written in JSP (Java Server Pages).

The whole MTA process regarding the user interface is divided into nine major steps. These nine steps are described in detail in the following sections.

Step 1: Start

The first step is to create a new project or to open an existing simulation project. The user interface for step 1 provides only these two options to choose one of them to offer an easy to handle and straightforward workflow for the tool support.

Step 2: Create / Open

If it is decided to create a new project in step 1 a list with all available ontologies is shown. One ontology set has to be selected for usage in the new project. An ontology set consists of one abstract ontology, one domain ontology and one or more customer ontologies.

If chosen to open an existing project in step 1 a list with all former saved project files is shown. After the selection of a project file and one is taken directly to step 7 where all possible scenarios of the SWIS Solution Model are listed.

Step 3: Services

Step 3 gives an overview of the selected ontology. The following properties are displayed:

- path to the ontology file
- number of nodes
- number of links

- all push services (if existing) – subdivided into provider and Consumer Services
- all request reply services (if existing) – for each request reply service the ID and the name of the containing provider and Consumer Service are displayed

Step 4: Connections **(see Figure 4.7)**

In this step it is possible to select all collaborations which are used to calculate the SWIS Solution Model. A collaboration represents the selection of a consumer service and one possible provider service. If for a consumer service more than one provider services are selected, each pair of consumer and provider represents a collaboration. For example: for one consumer service three possible provider services exist. If all three providers are selected for the one consumer service, this will result in three collaborations. The screen is subdivided into push and request reply services. So it is possible to select the collaborations separately for the two main types of services.

- ***Push service features:*** The first text area shows all available Consumer Services. By clicking on one of them, the appropriate Provider Services are shown in the second text area. Now, one can choose a specific Provider Service by clicking on it. The selected collaboration is automatically saved to be used for the calculation of the SWIS Solution Model. After clicking on a Provider Service the properties of the collaboration are shown in the third text area:

  - name and ID of the selected Consumer Service
  - name and ID of the selected Provider Services
  - consumer T-Map and provider T-Map of each selected collaboration (if existing)
  - ID of the converter or external service for the selected collaboration (if existing)

  To select more than one Provider Service for a specific Consumer Service one has to press the „Ctrl“-button and click on the different Provider Services. One has to click on „no Provider“, if no Provider Service should be used for the selected Consumer Service.

  The color of a Consumer Service in the first text area is green if there is a selected Provider Service for them. So the entry is colored, if at least one collaboration for this Consumer Service is selected.

  One can search for Consumer Services by typing a search word in the search text field and click the „search“-button. Only the Consumer Services containing the search word will be displayed in the first text area. Click on the „reset“-button to clear the search and to see all Consumer Services.

  With the „select all“-button under the text area with the Consumer Services one can choose all collaborations for all Consumer Services. Unlike with the „deselect all“-button, one can clear all currently selected collaborations.

  With the „select all“-button under the text area with the Provider Services all Provider Services for the current selected Consumer Service will be selected.

Unlike, one can delete all currently selected Provider Services for the selected Consumer Service by clicking the „deselect all“-button.

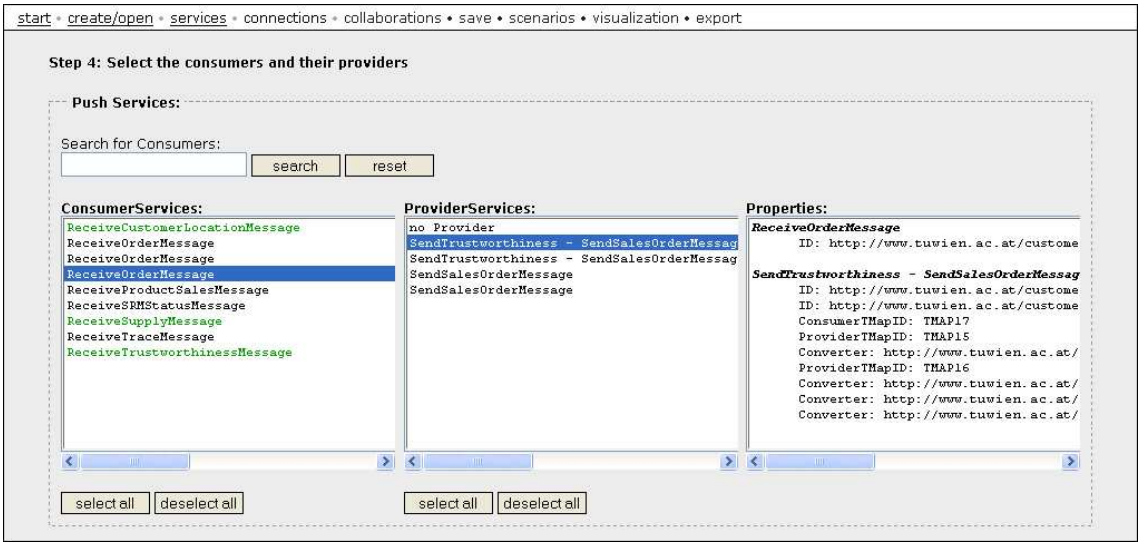


Figure 4.7: Selection of collaborations

- ***Request/Reply service features:*** The features for the request reply services are the same like the features for the push services. Only difference: the first text area shows the request receivers and the second text area shows the request senders.

Step 5: Collaborations

In this step all selected push and request/reply collaborations of the ontology from step 4 are listed.

Step 6: Save

Now it is possible to save the new created simulation project. One needs to specify a name for the simulation project without the path and without a file extension. One has to type in a name for the simulation project and click the „save“-button. Saving the project file is optional.

Step 7: Scenarios

From step 6 to step 7 the SWIS Solution Model with all selected collaborations is calculated. Now all possible scenarios from the calculation are listed in the first text area. If one clicks on one of them one can see the route description of the selected scenario in the second text area. One has to choose a scenario and then go to the next step.

Step 8: Visualization

In this step one is able to start the visualization of the selected scenario in step 7. The visualization gives a graphical view of the calculated scenario with all containing nodes and links. Figure 4.8 shows the visualization of the calculated scenario.

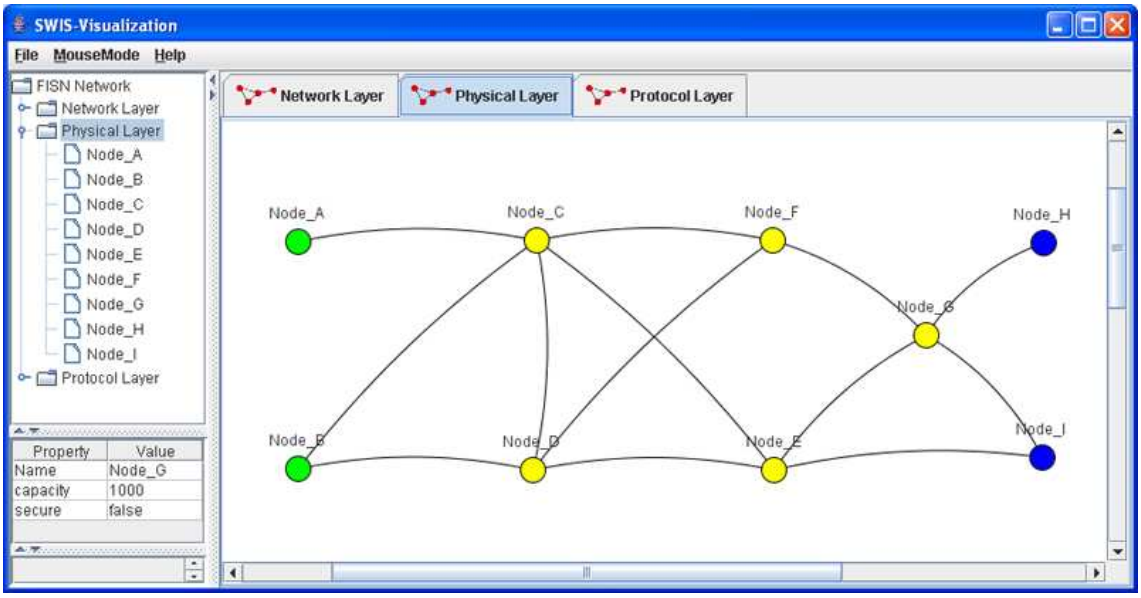


Figure 4.8: Visualization of the SWIS network with the tool support

Step 9: Export

At last it is possible to save the SWIS Solution Model into a XML-file. One needs to type in the path and the name of the file and click the „save“-button to export the SWIS Solution Model. The SWIS Solution Model is a XML configuration file, which contains all necessary settings to be made to finally build the integration solution encompasses all participating heterogeneous systems.

Correlation of tool support and MTA

Figure 4.9 shows the correlation of the tool support and the MTA. The figure also explicitly represents the human interactions during the tool support process steps. It is obvious that a human must only intervene a few times. As shown in the figure a human must come into action in step 1 (choose to create a new project or open an existing project), step 2 (choose an ontology or a project), step 4 (choose the collaborations) and in step 7 (choose one scenario out of the calculated scenarios from the MTA). During the whole process steps the exclusion of human faults are intended to finally get a correct and functioning SWIS solution model.

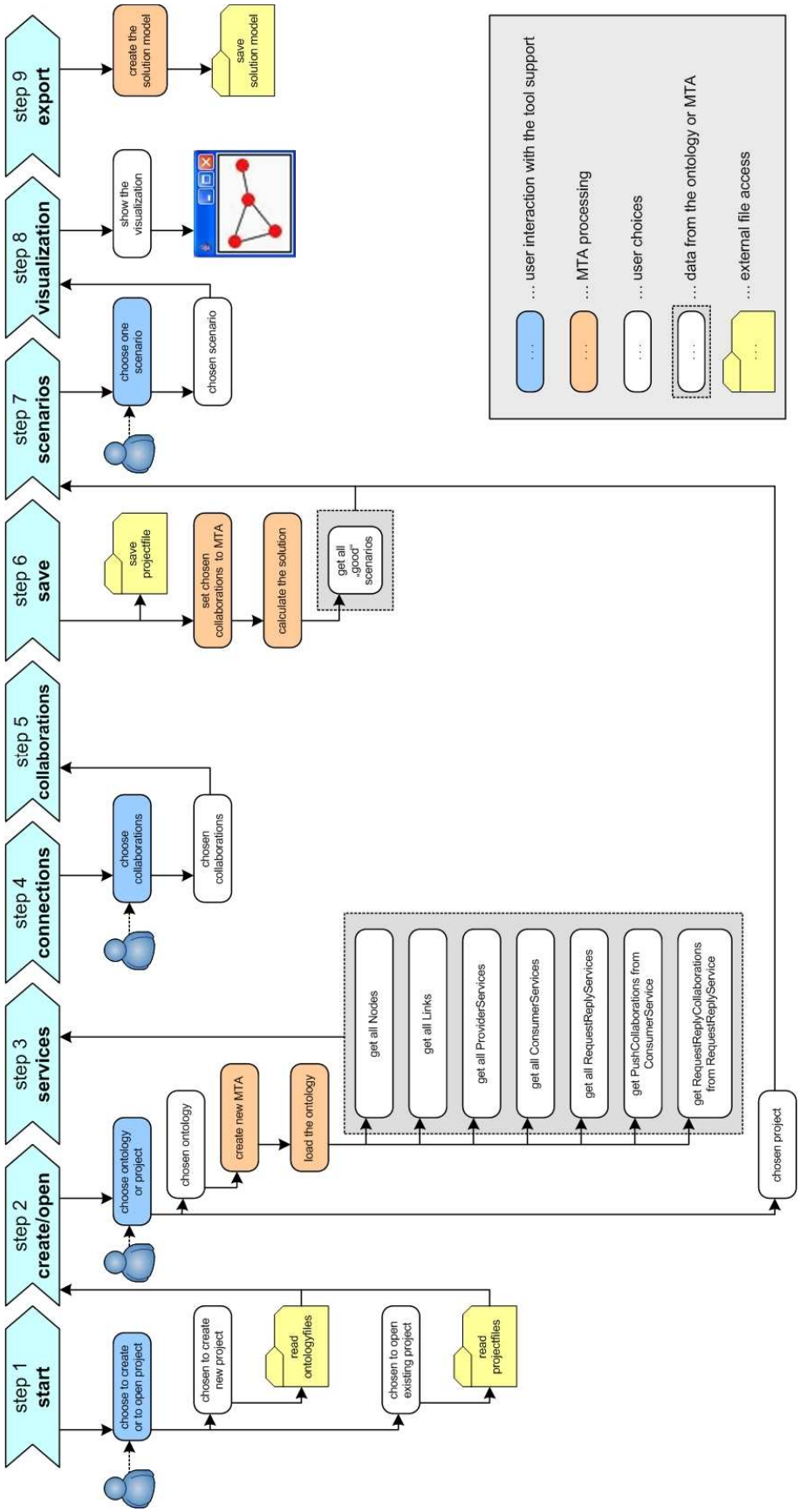


Figure 4.9: Correlation of tool support and MTA

4.4 *Case Study for tool support*

In this chapter the case study to evaluate the benefits of the tool support for the MTA process is introduced. The case study was performed in order by using a very simple example. Out of the wide range of different concepts defined in the SWIS ontologies, just a few were taken. The used ontologies contained only the essential information to create an integration solution model. No different protocols were used and all defined nodes and links supported one and the same protocol. Therefore the test persons had no additional effort to connect only those nodes and links that supports the same protocol. It could be assumed that all specified nodes and links defined in the instruction can directly talk together. Further no special splitting or aggregation of the various message segments containing in a message were used. Also no external services or type converters had to be used by the test persons. All in all the test persons had to fulfill a low-level scenario with no complex processing.

Generally, the test persons needed no special preconditions to participate in the case study. This made it easy to find an appropriate amount of suitable persons. The test persons got a list with all network devices (see Table 4.1) and they had the task to make an evaluation of the given network. The whole case study was divided into two major steps – a manual and an automated step.

Nodes	A B C D E F G H I		
Link	**Node - Node**	**Cost**	**Delay**
AC	A - C	10	50
BC	B - C	10	50
BD	B - D	1	500
CD	C - D	1	500
CF	C - F	20	500
CE	C - E	1	50
DF	D - F	20	500
DE	D - E	20	500
FG	F - G	1	50
EG	E - G	1	50
EI	E - I	1	500
GH	G - H	1	50
GI	G - I	10	50

Table 4.1: List of nodes and links for the case study example

Some explanations are necessary to understand the exercises to do in the case study. A network in SWIS consists of numerous nodes and links. Each link connects exactly two nodes, for example: link AC connects node A with node C. For all links some properties are specified, in our case the cost and delay-time. The cost property defines how expansive the use of the respective connection will be. And the delay-time specifies how long it takes to route from one node to the other node by using this link. Some nodes in the SWIS network represent endpoints and on each endpoint one or more services are running. A service is either a provider or a consumer service and can send or receive a specific message according to his message type.

Service	Node	MessageType
Provider Service	A	Order Message
Provider Service	B	Order Message
Provider Service	B	Info Message
Consumer Service	H	Order Message
Consumer Service	I	Info Message

Table 4.2: List of services for the case study example

First, in the manual step, the candidates had to draw a picture of the network architecture out of the given network devices. Figure 4.10 shows the network architecture of the case study example with all nodes and links defined in Table 4.1. The participants had to calculate possible routes from each given provider service to all possible consumer services which are able to talk together. This means that one provider and one consumer only establish a communication if both understand the same message type. After finding all possible routes of the network from provider to consumer services, the total costs and delay-time of the individual routes must be calculated. Afterwards the participants had to find an entire scenario solution with minimum costs by adding the routes with the lowest costs. Then a solution with the lowest delay-time must be specified, by adding the particular routes with the lowest delay-time.

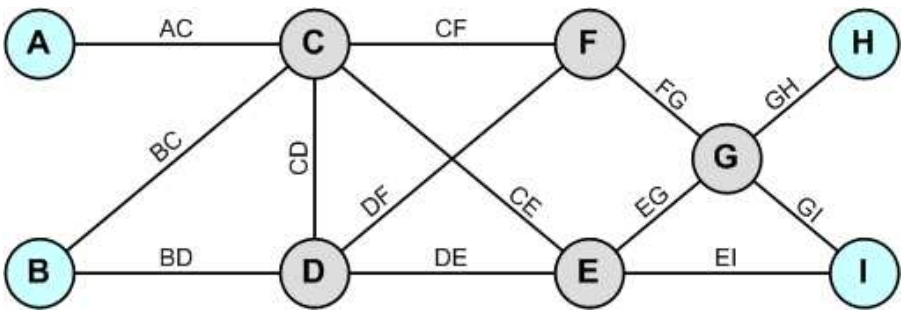


Figure 4.10: Network architecture of the case study example

Second, in the automated step, the test persons had to run the same test case with help of the tool support. Therefore the tool support was installed on a web server and the participants executed the tool support on a client machine.

Now, the single steps of the tool support which are processed by the participants are described. Look at Chapter 4.3.2 for further details on the single tool support process steps. First the test persons had to create a new project in step1. Then, in step2, the ontology for the case study must be chosen. After that, in step3, the participants got an overview about the number of nodes and number of links containing in the ontology. Also the provider and consumer services are listed. At next, in step4, the participants had to choose the wanted connections. Since in the case study was defined to find a solution were all consumer services are connected to all possible provider services, for each consumer service all available providers are selected. Either by pushing the “select all” button under the consumer service list, or by manual select of all providers for the individual consumers. Now step5 and step6 are jumped over and finally at step7 all possible optimal scenarios for routing the consumers with their providers are displayed. The participants had to check if their manual calculated scenario is containing in the list of the automatic calculated scenarios. If one’s scenario is in the list, he did his job well done.

During the case study the time was measured to identify the time difference between the time needed for the execution of the manual steps of the case study and the time consumed for the execution of the same example with the automatic steps by means of the developed tool support for the MTA. The results of this performed case study are discussed in Chapter 5.2.

Chapter 5

RESULTS

5 Results

In this chapter an answer to two of the four defined research questions is given. Chapter 5.1 describes the results of the evaluation of the developed SWIS integration approach compared to other traditional integration approaches according to the research question defined in Chapter 3.1. Afterwards, in Chapter 5.2, the results of the performed case study of the tool support are described and give an answer to the research question defined in Chapter 3.2.

5.1 *Evaluation of the SWIS approach*

This evaluation encompasses a detailed comparison of the developed SWIS integration approach with other existing integration architectures like individual interfaces, hub & spoke and service-oriented architecture (SOA). Therefore some criteria defined by Aier and Schönherr [7] are used to compare the different integration approaches. The covered integration criteria include initial planning efforts, initial development efforts, technical adaptations, non-invasive legacy/host integration, maintainability, and customizability.

5.1.1 Initial planning efforts

This criterion encompasses the efforts needed to design and implement the integration solution during the starting phase of the integration project. Thereby also the amount of support the human staff needs to be able to design and implement a correct and functioning integration solution is covered. Compared to other traditional integration architectures, the developed SWIS approach does not differ mentionable. The only difference is that the human integrators need knowledge about semantic modeling.

5.1.2 Initial development efforts

The initial development efforts encompass the efforts needed for the development of the integration technology during the initial phase. In traditional UML-based integration approaches the models created in different modeling languages are depending among each other. Due to the dependencies of the single models, an integration system can only be modeled in an incremental way and hence longer development duration is needed. Furthermore, all needed models must be created first before they are able to be verified. Therefore, possible modeling failures are detected only very late, more precisely, at the end of the development of the models. The correction of these failures implies an increase of

the development costs. Another shortcoming of UML-based integration approaches is the high complexity of the developed models due to the prevailing dependencies of the single models. Therefore, only a few designers are capable to entirely understand and enhance the different models.

In contrast, the developed SWIS approach provides concurrent modeling of the different requirements and capabilities for the participating systems. This is achieved by means of the three ontology layers (abstract, domain and customer ontology) which are used to describe the single systems. The three ontologies are already described in Chapter 4.1.2. Due to the layered ontologies, the SWIS approach offers a way that model designers can create just a comparatively small partial model for a particular system and therefore do not need to have exact knowledge about the entire system. Furthermore, the single partial models can be verified immediately without having all models of the entire system. Continuous verification of the partial models ensures proper models throughout the entire integration process.

5.1.3 Technical adaptations

The technical adaptations focuses on the development efforts needed to correct existing integration architectures to meet new requirements or to offer new capabilities after modification of them. In order to add new or to modify existing requirements or capabilities the existing models must be changed or possibly new models must be created. In traditional approaches, altering existing models can become a serious problem, because of the dependencies between the single models. Therefore, a violation of the given dependencies must be avoided by manual checking. A possibly dependency violation will be reported at the end of the development process. Otherwise, in the developed SWIS approach no manual checks are done. In SWIS, automatic consistency checks of the semantic models are performed and therefore only verified and validated models are allowed. So, possible errors will be reported immediately and appropriate measures can be taken in time.

5.1.4 Non-invasive legacy/host integration

In many organizations numerous of different legacy applications are running. Often the applications are not going to change for adaptability improvement because of the high risk and increasing complexity the change brings with it. Each of the different integration architectures has their own technique to integrate legacy systems within an overall integration solution. The non-invasive legacy/host integration criterion describes the ability of integration architectures to support the integration of unchangeable legacy applications.

Traditional integration architectures need a common data model to achieve the interoperability of different legacy applications. Therefore the messages of the single legacy systems must be transformed from the varying data formats into a uniform internal data format and vice versa. But the agreement on which common data model to be used for the different participating systems of a small integration project is already not easy obtainable.

Even in larger integration projects the decision on a common data model becomes harder and is almost impossible to achieve.

The developed SWIS integration approach offers a layered structure of the required semantic models to accomplish the need for a common data model. The layered semantic models contain the proprietary information about the participating systems and this information is mapped to more overall domain knowledge. Therefore the developed integration approach with the underlying layered semantic models allows:

- deduction of possible communication partners according to their requirements and capabilities as well as their semantic meaning of the available messages, and
- directly and automatically transformation of the exchanged messages between the participating legacy systems, from the source message presented in a proprietary message format into the target message presented in a different proprietary message format, without using a common data model.

5.1.5 Maintainability

The maintainability criterion focuses on possibly available administration tools or monitoring facilities to maintain the integration architecture and additionally focuses on the efforts needed to arrange qualified maintenance. General administration or monitoring tools to maintain an integration architecture are not to neglect in case to easily maintain the integration solution. In traditional integration approaches, the efforts needed to integrate numerous heterogeneous systems starting from the scratch are already high. But the needed efforts are still higher and more complex for later integration of additional systems into an existing integration solution, because of the permanently verification of the already created UML models after a new system was added. Furthermore, additional adaptations of the existing system in the presented integration solution may be required. Also the integration architectures developed by using the SWIS integration approach need special tools for general administration or monitoring. But the SWIS approach uses a more flexible way to describe new to add systems compared to traditional integration approaches. It is possible to immediate verify the semantic models by means of integrated checks. The semantic models of the new systems are added to the existing integration solution and no additional adaptations of the already integrated systems are needed. This is achieved by an automatic generation of the necessary transformation instructions of the proprietary message formats used by the newly added systems.

5.1.6 Customizability

The customizability criterion identifies the degree of customization for the integration architecture to meet changing requirements. In traditional UML based integration solutions, the UML models are not suitable to deal with changes or extensions of the existing requirements without adequate redesign of them. So changing or adding new requirements to the existing requirements implies a redesign of all current created models to get an

appropriate design to meet the altered requirements. Otherwise, the developed SWIS integration approach uses ontologies as underlying semantic data models. Ontologies not only have a limited view to their environment they are used in but also follow an "open-world" approach. This allows adding new concepts into existing ontologies in a more flexible manner. Therefore the SWIS approach is able to easily deal with changing existing or adding new requirements. Integration architectures created with the SWIS integration approach offer a pluggable mechanism to add new algorithms to be able to handle the new requirements provided in the semantic models.

5.1.7 Evaluation comparison

In this chapter the results of the evaluation of the developed SWIS integration approach with some traditional integration architectures are listed. Table 5.1 shows the comparison according to the six evaluation criteria described in the previous sections.

Criteria	Individual interfaces	Hub & spoke	SOA	SWIS
Initial planning efforts	+	– –	–	–
Initial development efforts	–	+	–	+
Technical adaptations	+	o	+	+
Non-invasive legacy/host integration	– –	+	o	+
Maintainability	– –	+ +	–	++
Customizability	– –	+ +	–	+

Table 5.1: Evaluation of integration architectures [7]

Compared to other traditional integration approaches the developed SWIS approach presents a good and easy to handle mechanism to integrate numerous heterogeneous systems. Foremost the flexible nature to respond to changing requirements by using layered semantic models and ontologies as data models offers a big advantage of the SWIS approach. Therefore, no extensive processing steps are needed to meet the altered requirements. Also the easy and flexible integration of different legacy applications without the need for a common data model is a big strength of the SWIS integration approach.

5.2 *Results of Case Study*

In this chapter the results of the performed case study regarding the tool support (see Chapter 4.4) are highlighted. First of all the execution steps respectively the working issues to be achieved during the case study are shown. Figure 5.1 illustrates the entire network architecture of the case study. The figure includes all nodes and links as well as the message types of the existing consumer and provider services. The consumer services are

running on node H and I, and the provider services are running on node A and B, whereas node B holds two provider services.

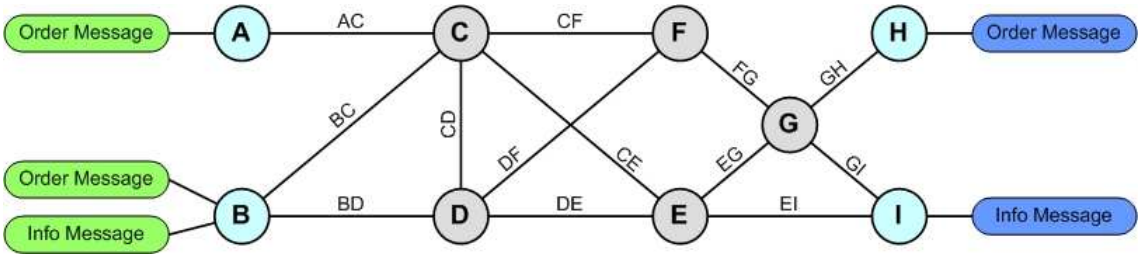


Figure 5.1: Resulting network architecture of the case study example

After the participants of the case study had drawn the network architecture, they had to find all possible routes (collaborations) from each consumer service to all appropriate provider services, which support the same message type. Figure 5.2 shows one routing example of the case study network, where the provider service running on node A is routed with the consumer service running on node H. Both connected services support the message type “Order Message”.

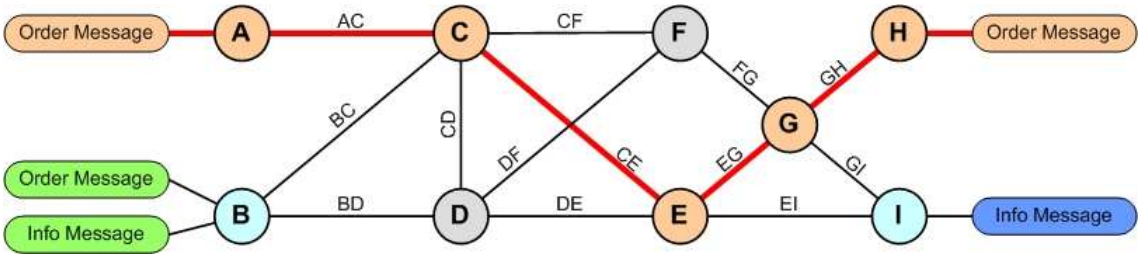


Figure 5.2: Routing example of the case study

After finding all possible collaborations, the participants had to calculate the total cost and delay-time for any collaboration. In Table 5.2 the calculation of the before described collaboration from node A to H (defined in Table 5.3 as collaboration “AH_1”) is shown. To get the result, the particular costs and delay-times of the used links are summed-up.

Link	Node - Node	Cost	Delay
AC	A - C	10	50
CF	C - F	20	500
FG	F - G	1	50
GH	G - H	1	50
	Total:	**32**	**650**

Table 5.2: Collaboration example of the case study

In this example the “Order Message” is routed from the provider service on node A to the consumer service sitting on node H, which supports the same message type. The routing goes from node A to node C, thereafter to node F and after node G the target node, node H is reached.

All possible collaborations of the case study network are listed in Table 5.3. For any collaboration the total cost and delay-time are specified. The table with the collaborations is divided into the various routes where each consumer service is connected with an appropriate provider service. Therefore all different routes to connect a consumer service with the needed provider service (both supporting the same message type) are covered. In our case node A can send the “Order Message” to node H, and also node B can send such a message to node H, node B can additionally send the “Info Message” to node I.

Collaboration		Cost	Delay	Route
A - H	AH_1	32	650	A - C - F - G - H
	AH_2	13	200	A - C - E - G - H
B - H	BH_1	32	650	B - C - F - G - H
	BH_2	13	200	B - C - E - G - H
	BH_3	5	1150	B - D - C - E - G - H
B - I	BI_1	12	600	B - C - E - I
	BI_2	22	200	B - C - E - G - I
	BI_3	4	1550	B - D - C - E - I

Table 5.3: Possible collaborations of the case study example

Out of all possible collaborations, an overall scenario which contains at least one collaboration per provider/consumer pair has to be defined. The cost and delay-time of the selected collaborations are summed-up to get the total costs and total delay-time for the specified scenario. For example, if the collaboration AH_1, BH_1 and BI_1 of Table 5.3 are used for the resulting scenario (scenario number 1 in Table 5.4), the current costs and delay-times are added together. The total costs of the resulting scenario are calculated by adding the costs for the collaborations AH_1 (32), BH_1 (32) and BI_1 (12):

$$32 + 32 + 12 = 76$$

And the total delay-time of the scenario is calculated by adding the single delay-times for the collaborations AH_1 (650), BH_1 (650) and BI_1 (600):

$$650 + 650 + 600 = 1900$$

In Table 5.4 all possible combinations of the single collaborations in Table 5.3 are listed. Out of all possible combinations, the test persons had to find one scenario with minimal costs and one scenario with minimal delay-time.

Scenario		Cost	Delay
1	AH_1 + BH_1 + BI_1	76	1900
2	AH_1 + BH_1 + BI_2	86	1500
3	AH_1 + BH_1 + BI_3	68	2850
4	AH_1 + BH_2 + BI_1	57	1450
5	AH_1 + BH_2 + BI_2	67	1050
6	AH_1 + BH_2 + BI_3	49	2400
7	AH_1 + BH_3 + BI_1	49	2400
8	AH_1 + BH_3 + BI_2	59	2000
9	AH_1 + BH_3 + BI_3	41	3350
10	AH_2 + BH_1 + BI_1	57	1450
11	AH_2 + BH_1 + BI_2	67	1050
12	AH_2 + BH_1 + BI_3	49	2400
13	AH_2 + BH_2 + BI_1	38	1000
14	AH_2 + BH_2 + BI_2	48	600
15	AH_2 + BH_2 + BI_3	30	1950
16	AH_2 + BH_3 + BI_1	30	1950
17	AH_2 + BH_3 + BI_2	40	1550
18	AH_2 + BH_3 + BI_3	22	2900

Table 5.4: Possible scenarios of the case study example

In Table 5.4 the two minimal scenarios are obviously. Scenario number 18 (blue highlighted) is the scenario with minimal costs. Only 22 units are needed to perform this scenario. And the scenario with minimal delay-time is scenario number 14 (red highlighted). This scenario takes only 600 units for the execution.

As a result not all participating test persons had at last a correct result of the scenarios. Some of the test persons had miscalculated the total costs or total delay-time of the single collaborations. In turn, other persons did not find either the minimal costs scenario or the minimal delay-time scenario. Figure 5.3 pictures the result of the case study. A total of 28 test persons performed the case study. Out of all test persons, more than a half of them, more precisely 64.3 percent, found the correct solution with the two appropriate scenarios. Furthermore 25.0 percent of the test persons had miscalculated the cost and delay-time values. And the rest of the test persons, exactly 10.7 percent, had problems to find the correct solution of the two minimal scenarios. This could have different causes, for example

not all collaborations between linked provider and consumer services were found and therefore a minimal scenario containing the missing collaboration could not be found. Altogether, 64.3 percent (almost two-thirds of the test persons) found the correct solution and a total of 35.7 percent (almost one-thirds of the test persons) did not find the correct solution.

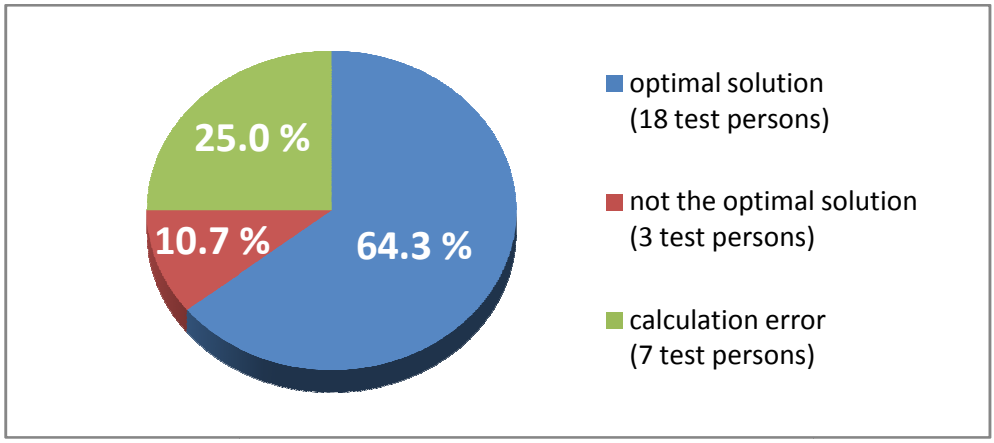


Figure 5.3: Result of the case study

Although the case study was performed with a very simple example without using different protocols, no splitting or aggregation of message segments in a message, and without using external services or special converters, a high amount of errors occurred by the manual steps.

During the case study the time a test person needed for the execution was measured. The time was separated into the time needed for the manual steps and the time needed for the automatic steps. For the manual steps an average time of 20 to 25 minutes were consumed, whereas for the automatic steps only 3 to 4 minutes were needed. Therefore the manual steps required up to six times more than the automatic steps. If we note that only a simple example was performed, the economy of time will be enormous by using a complex example, because the time needed for the automatic steps will be almost unchanged, but the time a human need for the manual steps will increase rapidly. Probably a human would not be able to manually find a correct and functioning integration solution of a complex example.

Chapter 6

DISCUSSION

6 Discussion

In this chapter, the developed SWIS approach will be discussed with reference to some of the aspects described in the related work (see Chapter 2). At first the structure of the SWIS approach is compared with the structure of a traditional MDA. Furthermore the realization of the integration patterns (see Chapter 2.4) in SWIS is described.

6.1 *Comparison of SWIS with traditional MDA*

In this chapter SWIS is compared with a traditional MDA process. In Figure 6.1 the comparison of the two approaches is displayed. On the left hand side a traditional MDA process is shown and on the right hand side the developed SWIS approach is pictured.

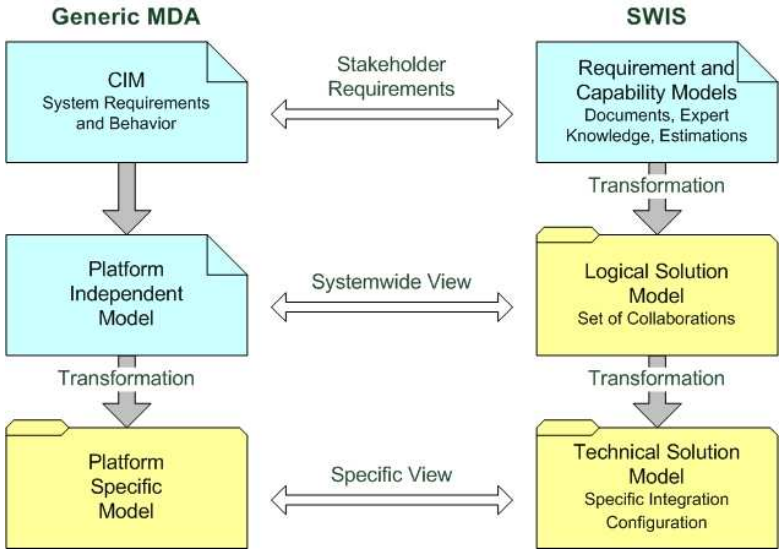


Figure 6.1: Comparison of generic MDA and SWIS

As shown in the figure, the generic MDA process and the SWIS approach have some similarities. First of all the stakeholder requirements are defined in specific models. In MDA a Computation-Independent Model (CIM) is used to specify the system requirements and behavior, whereas in SWIS the requirements and capabilities are also defined in models in terms of ontologies. In the next level, the system wide view, in the MDA a Platform-Independent Model (PIM – described in Chapter 2.2.1) is created manually, whereas in SWIS the logical solution model, which holds the abstract information about the underlying integration network containing a set of collaborations, is derived automatically based of the

information provided in the requirement and capability models. The third level consists of system specific information. On the one side, the MDA transforms the PIM into a Platform-Specific Model (PSM) which is used for a specific platform they are generated for, and on the other side, the SWIS approach transforms the logical solution model into a technical solution model which builds the configuration set for the integration solution and splits it up resulting in configurations for every single integration node.

6.2 *SWIS Integration Patterns*

This chapter focuses on the Integration Patterns defined by Hohpe and Woolf [1] in relation to their adoption in the SWIS integration approach. As first step, a schematic implementation is pictured for each Integration Pattern described in Chapter 2.4. Then the realization in the SWIS approach for each of the explained integration patterns is shown. As far as possible a precise example of the implementation is given.

6.2.1 Message Translator

This chapter explains the use of the Message Translator Integration Pattern in SWIS. First in Figure 6.2 a schematic representation of this pattern is shown to get an overview about the basic mechanism of the pattern.

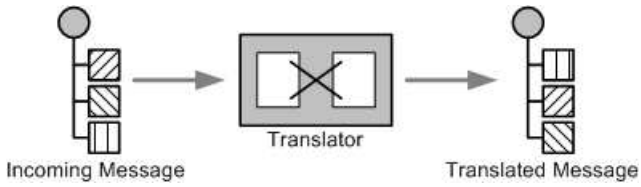


Figure 6.2: Message Translator Integration Pattern [1]

The message translator pattern is represented in the SWIS approach by means of the transformation maps (short T-Maps). A T-Map takes the task to transform the message segment of a sender (the source) into the message segment of a receiver (the target). The T-Map accomplishes the transformation either by calling an external service or by using a specific data converter. Therefore a T-Map definition contains the description how the transformation of the input message segment into the output message segment is done. The following example shows such a translation defined in a T-Map.

```
<?xml version="1.0"?>
<tmap version="1.0" name="Node_TMap">
  <inputMessage id="InputMessage" typeURI="IFPL">
    <segment domainConcept="ID" format="Character5" name="OID"/>
    <segment domainConcept="StartTime" format="Time UTC" name="STA"/>
    <segment domainConcept="Duration" format="Integer" name="DUR"/>
  </inputMessage>
```

```
  <outputMessage name="OutputMessage" typeURI="GFPL">
    <segment domainConcept="ID" format="Character5" name="OrderID"/>
    <segment domainConcept="StartTime" format="Time UTC" name="Start"/>
    <segment domainConcept="EndTime" format="Time_UTC" name="End"
    selectSegmentWithID="Duration"/>
  </outputMessage>

  <transformation>
    <converters>
      <converter id="calcEndTime" className="CalcEndTime"
        inputFormat="Integer" outputFormat="Time_UTC" losless="true"/>
    </converters>
    <externalServiceCalls/>
  </transformation>
</tmap>
```

Listing 6.1: Implementation of the Message Translator pattern

In this example the input message consisting of an ID, a start time and a given duration time is translated into the output message consisting of an ID, a start time and an end time. Whereas the wanted end time is calculated by adding the duration to the start time. In the transformation section of Listing 6.1 the needed converters or external service calls are defined. In this specific example a converter is used to calculate the end time out of the given duration.

6.2.2 Publish-Subscribe Channel

In this section the implementation of the Publish-Subscribe Channel in the SWIS approach will be described. Figure 6.3 shows a traditional realization of the Integration Pattern.

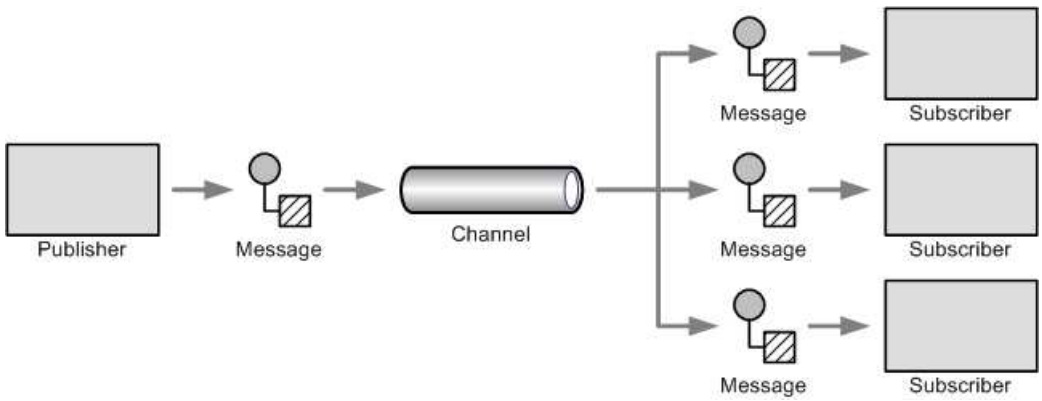


Figure 6.3: Publish-Subscribe Channel Integration Pattern [1]

The Publish-Subscribe Channel pattern is implemented by using so-called Receiver Groups in SWIS. With Receiver Groups it is possible to multicast a message from a provider service (publisher) to a defined group of numerous consumer services (subscribers). A Receiver Group is not a real physical group in a SWIS network it is only

described in a semantic manner. The Model Transformation Algorithm (see Chapter 4.2) creates a single connection for each specified subscriber with the publisher. Therefore each connection is defined by a unique DataflowID.

```
<sendmapping>
  <map>
    <input>
      <properties>
        <property name="publisher" value="publisher1" />
      </properties>
    </input>
    <output>
      <dataflow ID="publisher1-subscriber1" />
      <dataflow ID="publisher1-subscriber2" />
      <dataflow ID="publisher1-subscriber3" />
    </output>
  </map>
</sendmapping>
```

Listing 6.2: Implementation of the Publish-Subscribe Integration Pattern

In this example a message from the defined publisher “publisher1” is transmitted to three subscribers specified by three different data-flows. Thereby the message will be send to “subscriber1”, “subscriber2” and “subscriber3”. In Figure 6.4 the implementation of the Publish-Subscribe Channel pattern with the transformation of each subscriber (containing in a receiver group) to a separate collaboration with the publisher is shown.

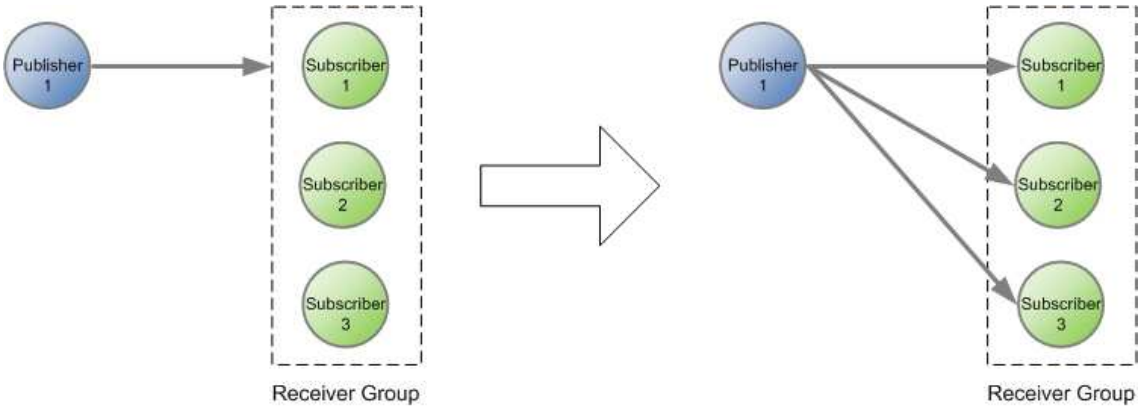


Figure 6.4: Realization of the Publish-Subscribe Channel pattern in SWIS

After the transformation a publisher sends a message to all interested subscribers whereas each collaboration uses a unique DataflowID.

6.2.3 Command Message

In this section the realization of the Command Message pattern in the SWIS approach will be described. Figure 6.5 shows the basic use of a Command Message. By means of a

Command Message a sender is able to invoke a procedure or method provided by a receiver. This is achieved by sending a Command Message whereby the receiver needs to deal with them. So a sender can inform the receiver to do something.

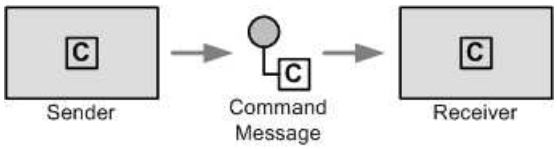


Figure 6.5: Command Message Integration Pattern [1]

In SWIS a Command Message is used to inform each participating nodes in a SWIS network that a new solution model is available. Therefore after deployment of the solution model each node receives a Command Message and reads the deployed configuration to set up their mode of operation. After reading the solution model a node is ready to work in the SWIS network. In SWIS also other Command Messages are used. One Command Message for start working of a SWIS node, one Command Message for stop working of a SWIS node and one Command Message to get special debug information of a SWIS node.

6.2.4 Request-Reply

This section describes how the Request-Reply pattern is implemented in the SWIS approach. Figure 6.6 shows a basic representation of this pattern by using a request and a reply channel to transmit the messages.

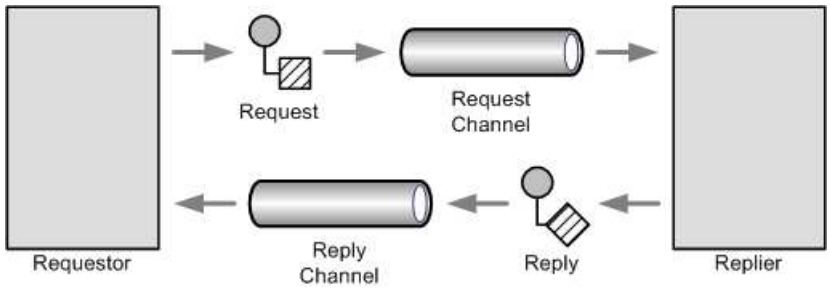


Figure 6.6: Request-Reply Integration Pattern [1]

In SWIS the request-reply communication mode is realized analogue to the pattern shown in Figure 6.6. The input semantic model defines such a communication with a single semantic collaboration containing an advice that a request-reply communication mode is used. Out of the defined collaboration two independent collaborations are created. One as request collaboration between request provider and request consumer service, and one as reply collaboration between the reply provider and the reply consumer service. To correlate the two generated collaborations together the same DataflowID of the request collaboration

will be set to the reply collaboration. So the correlation between request and reply collaboration is specified. Figure 6.7 shows the transformation from the single common semantic collaboration using a request-reply communication mode, into two separated collaborations (one for the request and one for the reply). Both single connections have the same DataflowID to define the correlation between them.

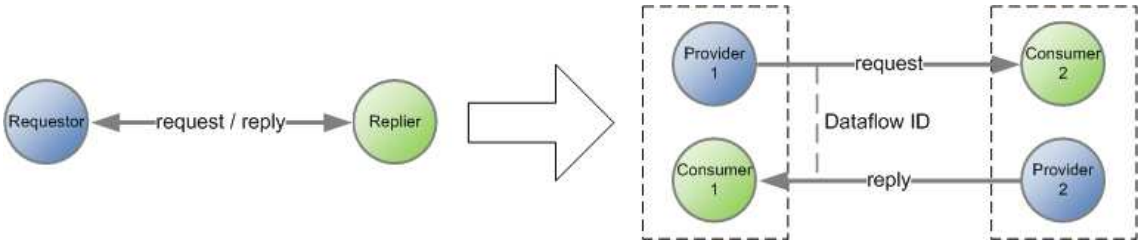


Figure 6.7: Realization of the Request-Reply pattern in SWIS

Within the SWIS approach the request consumer service and the reply provider service are containing on the same SWIS node. Otherwise it would be not possible that the reply provider can use the DataflowID from the request message.

6.2.5 Return Address

In this chapter the implementation of the Return Address pattern will be described. The integration pattern according to Hohpe and Woolf [1] is pictured in Figure 6.8.

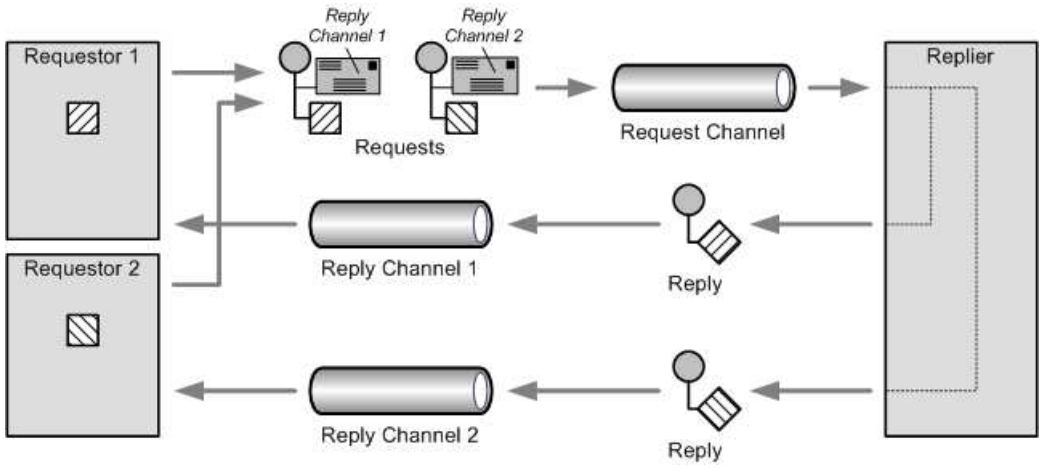


Figure 6.8: Return Address Integration Pattern [1]

This pattern is implemented just like the Request-Reply pattern. SWIS uses a unique DataflowID to define a collaboration between a provider service and a consumer service. In the Request-Reply pattern, a request-reply channel is split into two single collaborations,

one between request sender and reply receiver and one between reply sender and request receiver. Both collaborations have the same DataflowID. For the Return Address integration pattern the used mechanism is the same like the request-reply mechanism. Therefore the DataflowID demonstrates the return address.

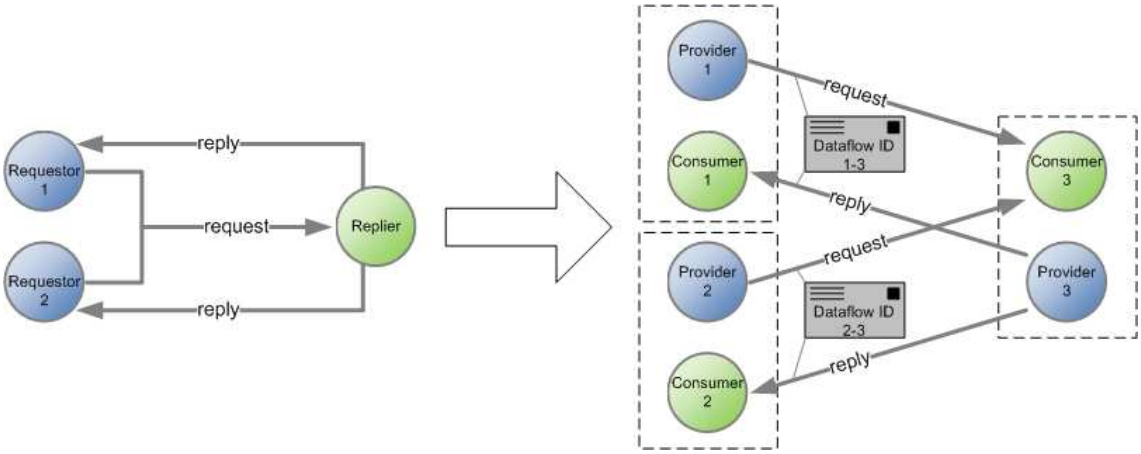


Figure 6.9: Realization of the Return Address pattern in SWIS

Figure 6.9 illustrates how the Return Address pattern is realized in SWIS. If several requestors want to send a request message to a common replier the SWIS approach transforms the connections to one separate request-reply connection for each requestor to the replier. Furthermore, each of the request-reply connections is splitted into two collaborations both containing the same DataflowID depicting the return address.

6.2.6 Correlation Identifier

This chapter shows how the Correlation Identifier integration pattern is implemented in the SWIS approach. In Figure 6.10 the traditional implementation of this integration pattern is pictured.

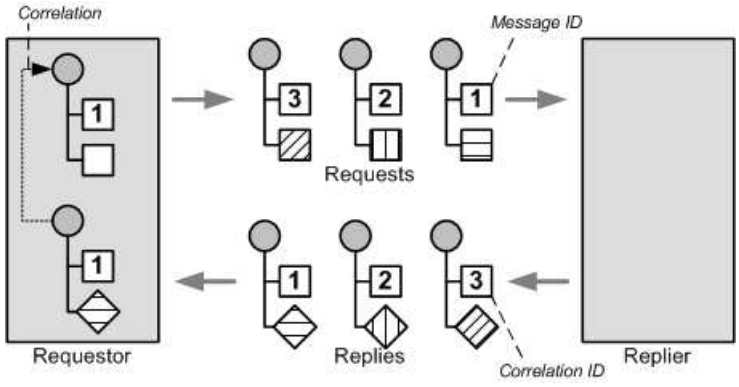


Figure 6.10: Correlation Identifier Integration Pattern [1]

In SWIS this integration pattern is implemented as shown in each requestor uses an own message counter. Each time a message is send by the requestor, the actual number of the message counter is added to the message as MessageID and then the counter is increased by one. Therefore the requestor is able to allocate the incoming reply messages to the appropriate request.

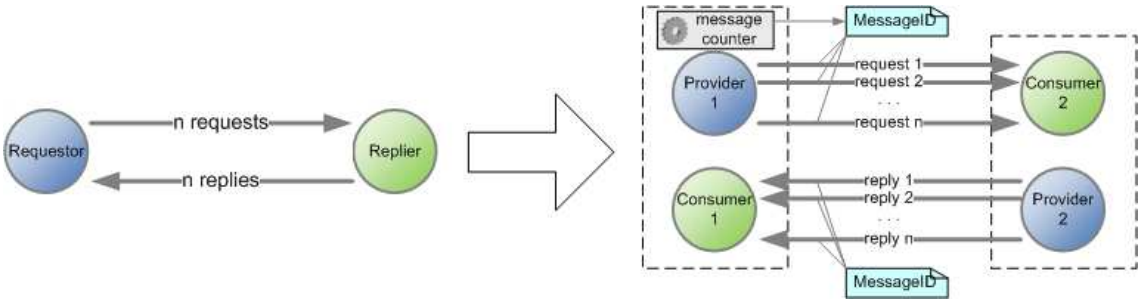


Figure 6.11: Realization of the Correlation Identifier pattern in SWIS

Figure 6.11 pictures how the SWIS approach implements the Correlation Identifier pattern by using a message counter to generate the MessageID for each sending message. The added MessageID is a unique number because the counter is incremented permanently after a message was send.

6.2.7 Dynamic Router

In this section an overlook about the Dynamic Router integration pattern according to their implementation in the SWIS integration approach is given. Figure 6.12 shows the basic design of this pattern.

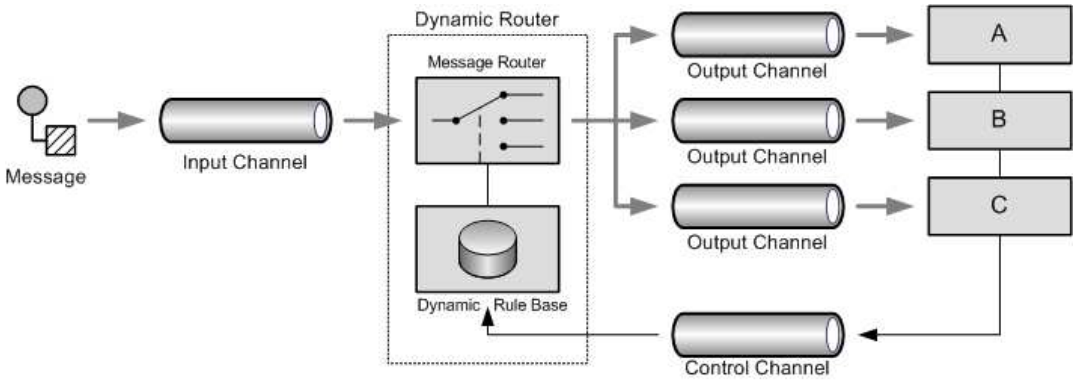


Figure 6.12: Dynamic Router Integration Pattern [1]

In SWIS the Dynamic Router is represented by the SWIS solution model. The model contains all necessary information to establish a SWIS network and therefore enables the

correct routing of a message from the source to the destination. Each node offers such a generated solution model. If a node which is currently used for the routing fails, the previous node automatically sends the message to an alternative node in order to keep up the message flow. The used alternative node is specified by a Backup Route defined in the solution model for each single SWIS node.

6.2.8 Recipient List

This chapter describes the Recipient List integration pattern (displayed in Figure 6.13) and their representation in the SWIS approach.

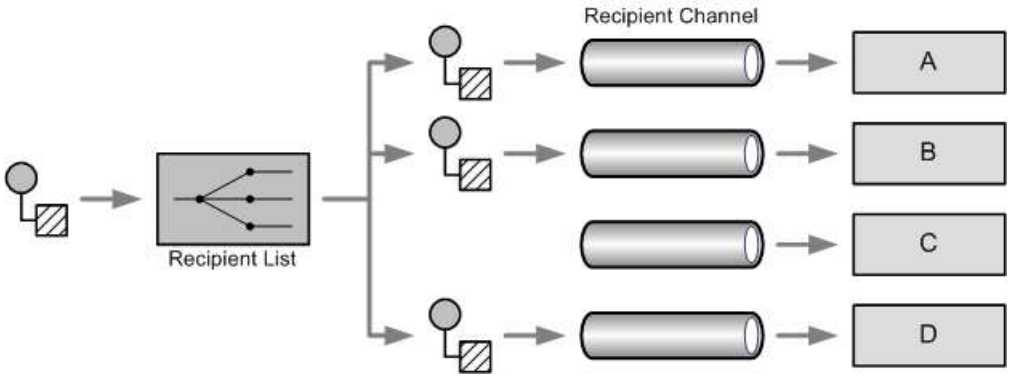


Figure 6.13: Recipient List Integration Pattern [1]

In SWIS the Recipient List is realized by using special Receiver Groups (already described in Chapter 6.2.2 for the Publish-Subscribe Channel integration pattern) or in the other case Sender Groups. By using the Receiver Group pattern it is possible that one provider service can send a message to numerous consumer services containing in a Receiver Group.

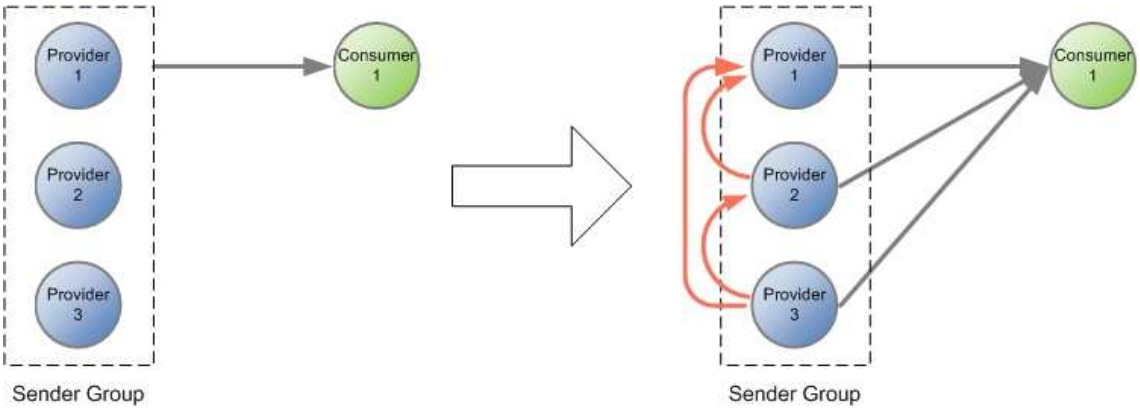


Figure 6.14: Realization of a Sender Group in SWIS

The Recipient List pattern is realized by means of a Receiver Group whereas the containing consumers are split into single collaborations with the provider. The Sender Group pattern is the opposite of the Receiver Group. In a Sender Group all containing provider services are able to send to a specific consumer service by producing the needed message type. But it is eligible that just one provider sends to the consumer. Therefore the SWIS approach supports the possibility to suspend transmitting messages for the different providers. In SWIS only one provider of a Sender Group sends a message whereas the single provider services are prioritized within the Sender Group. Figure 6.14 shows the implementation of a Sender Group in the SWIS approach.

As displayed in the figure the connection between a Sender Group and a consumer service is split into a single collaboration for each provider service (analogue to the Receiver Group implementation). Additionally the provider services combined in a Sender Group are connected to each other for inner communication (see red arrows). This is needed to determine the right provider service for sending a message to the consumer according to some conditions like prioritization and availability. If a sending system fails, the next sender containing in the Sender Group takes the task of sending the message to the required consumer. All providers in the Sender Group have exactly the same structure.

6.2.9 Splitter

This section describes the Splitter integration pattern. A Splitter is used to separate a single message consisting of several message segments into several messages containing one message segment. Figure 6.15 pictures a schematic representation of this integration pattern.

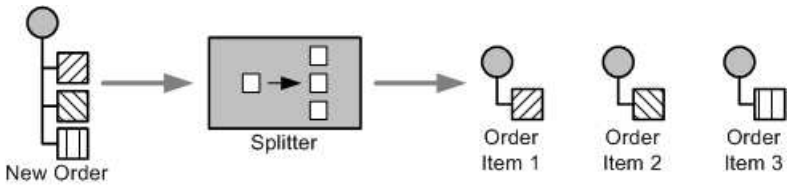


Figure 6.15: Splitter Integration Pattern [1]

The Splitter integration pattern is realized in SWIS with so-called T-Maps. T-Maps are already used in the Message Translator pattern (see Chapter 6.2.1) and are basically used to transform an input message provided in a specific format into a predefined output message provided in another format. Thereby the T-Map additionally can use external services or special data converters. In this case, a T-Map is actually a Content Filter (see Chapter 6.2.14). To achieve the implementation of the Splitter pattern several such T-Maps are needed. One T-Map is required for one generated output message. If an input message consisting of several message segments should be split into several single output messages containing only one message segment, as many different T-Maps are needed as message segments exists in the input message. Thereto each T-Map processes the same

input message. In a SWIS network, each participating node can encompass more than just one T-Map.

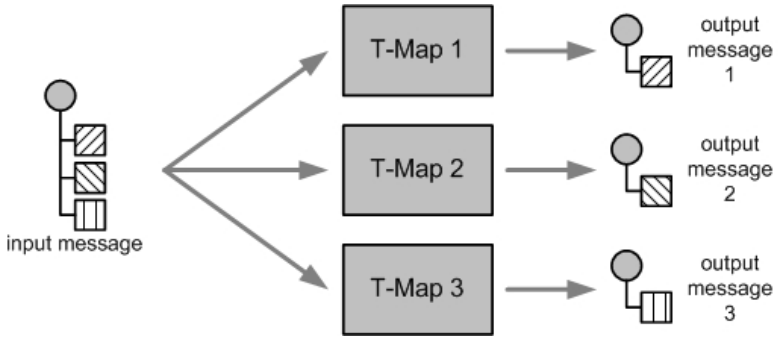


Figure 6.16: Realization of the Splitter pattern in SWIS

Therefore to split the input message completely into three single output messages, three T-Maps are needed (see Figure 6.16). In generally for the implementation of the Splitter pattern one input message and several T-Maps are working together.

6.2.10 Aggregator

In this chapter the Aggregator integration pattern is defined. An Aggregator merges several input messages into one combined output message. Figure 6.17 pictures the basic functionality of this pattern.

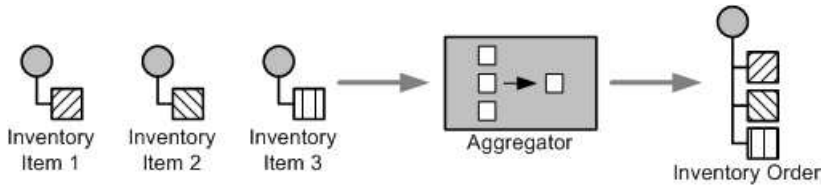


Figure 6.17: Aggregator Integration Pattern [1]

Just like the before described Translator and Splitter pattern, an Aggregator is also realized by means of T-Maps in the SWIS approach. In contrary to the Splitter realization, where one input message and several T-Maps are used, for the implementation of the Aggregator pattern several input messages and just one T-Map are working together. The T-Map which acts as Aggregator receives the different input messages which are send to the node the T-Map belongs to. After all needed input messages are arrived, the T-Map transforms the message segments from the single input messages into an output message containing all the different message segments.

```
<?xml version="1.0"?>
<tmap version="1.0" name="Node TMap">

  <inputMessages>
    <inputMessage id="InputMessage1" typeURI="IFPL1">
      <segment domainConcept="ID" format="Character5" name="OID"/>
    </inputMessage>

    <inputMessage id="InputMessage2" typeURI="IFPL2">
      <segment domainConcept="StartTime" format="Time_UTC" name="STA"/>
    </inputMessage>

    <inputMessage id="InputMessage3" typeURI="IFPL3">
      <segment domainConcept="Duration" format="Integer" name="DUR"/>
    </inputMessage>
  </inputMessages>

  <outputMessage name="OutputMessage" typeURI="GFPL">
    <segment domainConcept="ID" format="Character5" name="OrderID"/>
    <segment domainConcept="StartTime" format="Time UTC" name="Start"/>
    <segment domainConcept="Duration" format="Integer" name="Duration"/>
  </outputMessage>

</tmap>
```

Listing 6.3: Implementation of a T-Map for the Aggregator pattern

Listing 6.3 shows an example of an Aggregator T-Map to transform three input messages each with one message segment into an output message containing the three single message segments.

6.2.11 Message Broker

In this section another integration pattern from Hohpe and Woolf [1] the Message Broker pattern is pictured. Figure 6.18 shows a schematic representation of this pattern. A Message Broker enables the communication between the participating systems by controlling the flow of the messages in a decoupled manner.

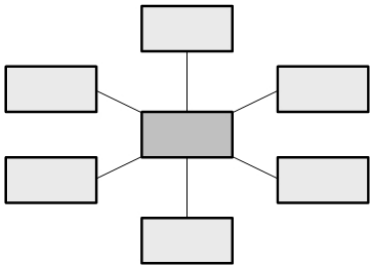


Figure 6.18: Message Broker Integration Pattern [1]

SWIS uses the underlying physical network for the basic communication and message transport. To achieve that a SWIS device works with the underlying physical network, and to establish the routing or transport of a message in the physical network some conditions

must be fulfilled. For each existing SWIS link in the SWIS network a form of a middleware is specified which connects the link into the overall integrated system. A specified middleware can be used by multiple links and otherwise multiple middleware technologies can be used in a SWIS solution. Therefore the specified middleware technologies of the SWIS network to establish the communication between SWIS devices and the transportation of messages can be defined as a Message Broker.

6.2.12 Envelope Wrapper

The Envelope Wrapper integration pattern surrounds a transmitted message with additional information to enable the communication between sender and receiver. This section describes the Envelope Wrapper integration pattern and in Figure 6.19 the operation of this pattern is represented.

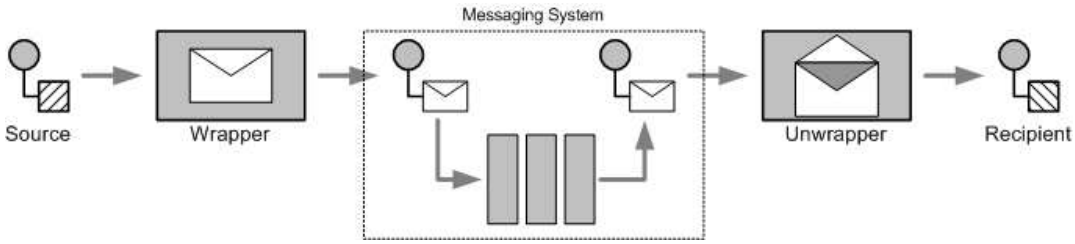


Figure 6.19: Envelope Wrapper Integration Pattern [1]

To enable the communication in a SWIS network a DataflowID is needed. A DataflowID specifies a connection exactly between two SWIS nodes or more precisely between two services running on legacy applications. Therefore a DataflowID is unique within a SWIS network and must be added to a message for a successful routing of the message. Each SWIS node offers a routing table containing the information to which node or service an incoming message with a specific DataflowID must be forwarded.

```
<dataflow ID="DataflowID1">

  <previous>
    <singlenode>
      <properties>
        <property name="linkID" value="Node1-Node2" />
        <property name="protocolID" value="TCP" />
      </properties>
    </singlenode>
  </previous>

  <forwarding>
    <priority value="1">
      <singlenode>
        <properties>
          <property name="linkID" value="Node2-Node3" />
          <property name="protocolID" value="TCP" />
        </properties>
      </singlenode>
    </priority>
```

```
    <priority value="2">
      <service ID="ServiceID" />
    </priority>
  </forwarding>

</dataflow>
```

Listing 6.4: Implementation of an Envelope Wrapper pattern

The dataflow definition contains two partitions. The first part defines the nodes or services the incoming message should be forwarded to, and the second part specifies the previous node the message should be returned to if the forwarding devices fail. In the forwarding classification more than one devices can be defined. To handle the routing with more than one forwarding device defined in the routing table, the devices are prioritized. Therefore the node tries to forward an incoming message to the node or service defined with the highest priority. If this device is not available the node forwards the message to the next device defined in the priority list and so on.

To enable the transmission of messages the DataflowID must be added to each message. In the SWIS approach the Model Transformation Algorithm (short MTA – see Chapter 4.2) calculates the DataflowIDs and surrounds the messages with the additional information.

6.2.13 Content Enricher

This chapter gives an overview about the Content Enricher integration pattern. The basic functionality of the pattern is displayed in Figure 6.20.

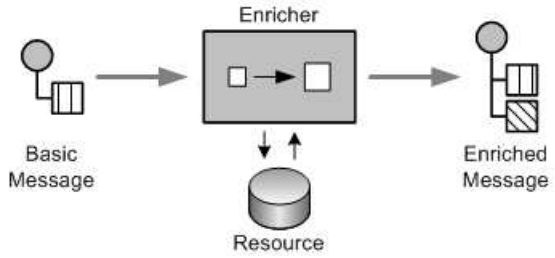


Figure 6.20: Content Enricher Integration Pattern [1]

In SWIS a Content Enricher is implemented by supporting external services calls defined for a T-Map. External services are needed if the transformation of a message segment cannot be handled from the T-Map and the included data converters. An external service call is defined in a T-Map. During the transformation of an input message into an output message, a message is sent by the T-Map to an external service by using a request/reply communication. This message contains the information which will be

converted by the external service and will then send back to the caller. Listing 6.5 shows the implementation of an external service call in a T-Map to be used as Content Enricher.

```
<?xml version="1.0"?>
<tmap version="1.0" name="Node TMap">
  <inputMessage/>
  <outputMessage/>
  <transformation>
    <converters/>

    <externalServiceCalls>
      <externalServiceCall serviceID="ExternalService">

        <inputMessage id="InputMessage" typeURI="CustomerDB">
          <segment domainConcept="CustomerID" format="Integer" name="CID"/>
        </inputMessage>

        <outputMessage name="OutputMessage" typeURI="CustomerDB">
          <segment domainConcept="ForeName" format="Character20" name="Forename"/>
          <segment domainConcept="SurName" format="Character20" name="Surname" />
          <segment domainConcept="Address" format="Character50" name="Address" />
          <segment domainConcept="ZIPCode" format="Integer" name="ZIPCode" />
        </outputMessage>

      </externalServiceCall>
    </externalServiceCalls>

  </transformation>
</tmap>
```

Listing 6.5: Implementation of a Content Enricher pattern

The Listing pictures an example where the input message for the T-Map contains a customerID and the output message should contain the forename, surname, address and ZIP code of the specified customerID. In that case the T-Map is not able to transform the message segments herself or by using special converters, because additional information is needed. So the T-Map has defined an external service call. An external service mostly use other information sources (e.g. a customer database) to get additional information. The defined external service in the example requires as input segment only the customerID and returns an output message containing the forename, surname, address and ZIP code. Now the T-Map can fill the output message segments with the returned data from the external service.

6.2.14 Content Filter

In this section another essential integration pattern, the Content Filter, is described. The following figure (Figure 6.21) pictures the basic functionality of a Content Filter for filtering several message segments containing in the input message to get an output message with fewer message segment in it.

In the SWIS approach the Content Filter pattern is realized by means of a T-Map. The T-Map specifies how an output message should be designed and is able to transform an incoming input message into an output message according the specifications. Thereto the

T-Map gets the message segments of the input message and inserts only specific message segments into the output message.

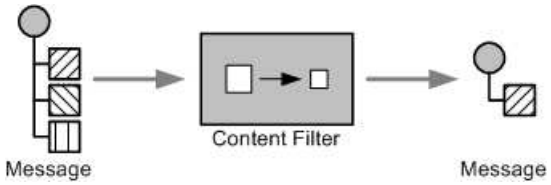


Figure 6.21: Content Filter Integration Pattern [1]

Listing 6.6 shows an example of a T-Map to transform an input message consisting of three message segments into an output message consisting of one message segment.

```
<?xml version="1.0"?>
<tmap version="1.0" name="Node_TMap">

  <inputMessage id="InputMessage" typeURI="IFPL">
    <segment domainConcept="ID" format="Character5" name="OID"/>
    <segment domainConcept="StartTime" format="Time UTC" name="STA"/>
    <segment domainConcept="Duration" format="Integer" name="DUR"/>
  </inputMessage>

  <outputMessage name="OutputMessage" typeURI="GFPL">
    <segment domainConcept="ID" format="Character5" name="OrderID"/>
  </outputMessage>

</tmap>
```

Listing 6.6: Implementation of a T-Map for the Content Filter pattern

Such a Content Filter is also used for the Splitter pattern (see Chapter 6.2.9). Multiple Content Filters depicts one Splitter (one single Content Filter for each message segment).

6.2.15 Normalizer

This chapter describes the Normalizer integration pattern for the translation of messages existing of different formats into messages existing of common formats. Figure 6.22 shows a schematic representation of the Normalizer pattern, whereas for each different message format a special translator is used to transform the message into a common message format.

In SWIS the Normalizer pattern is implemented by a combination of T-Maps and converters. The T-Map acts as a router to specify which converter has to be used for translating a specific input message format into a common output message format. The converters itself have the task to translate the single message segments from one format to another. For example, if the input message segment has the data type integer and the output segment data type should be double, the T-Map selects a converter that is able to

transform an integer number into a decimal number. In the SWIS network all needed converters for the transformation have to exist on the integration node connecting the sending service. Therefore the message segments of a message are first separately transformed and then again combined into a single message before transmission. So the receiver gets the message in the right format and can process it without further transformation.

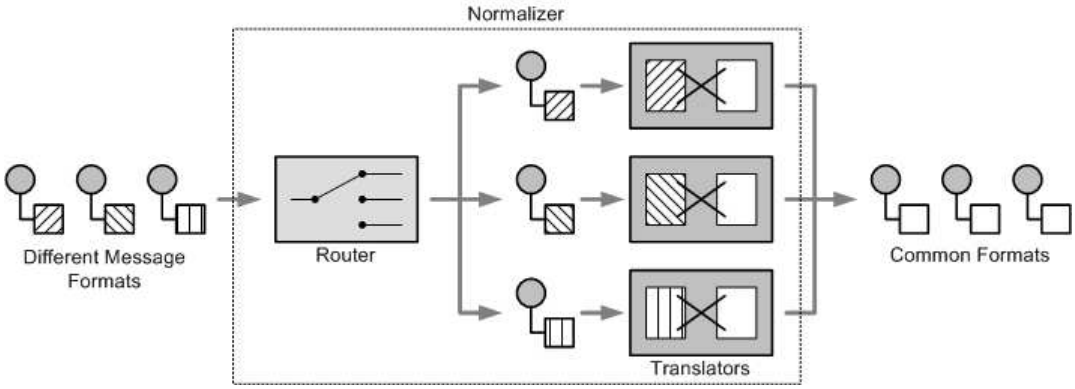


Figure 6.22: Normalizer Integration Pattern [1]

The SWIS approach also provides the possibility to define lossless converters. Such converters translate the messages without any loss of information. For example, if a decimal number is transformed into an integer number, the transformation is not lossless because the positions after the decimal point will be removed. Otherwise the transformation of an integer number into a decimal number is lossless because no parts of the origin number will be removed. If transmitted data should be converted lossless, but the collaboration between two services in a SWIS network has no appropriate converter to transform a message in a lossless way, no collaboration between the two SWIS services is possible.

Chapter 7

SUMMARY AND FURTHER WORK

7 Summary and Further Work

The integration of numerous heterogeneous systems, i.e., to build one big system out of different independent systems, gains increasing importance for organizations. Because of the probably high amount of different business applications running in a company, the integration of these systems to exchange data and information or to operate as one big system without having many single applications processed in various ways. But building an integration solution poses miscellaneous challenges. The applications are often designed to run independently from other systems and therefore cannot be easily integrated with other applications to an overall integration solution. Such legacy applications provide almost no interface for the connection with other systems and therefore the integration approach has to deal with this restriction. Furthermore, the applications differ between the programming languages they are written in, the operating platforms they are running on, or the used data formats; in short, each application has their own construction type and look. The integration approach should be able to handle such limitations and to build an overall integration solution with various legacy applications.

The System Wide Information Sharing (SWIS) approach, a promising approach to integrate a large number of heterogeneous systems, was introduced in the thesis. SWIS was developed as scientific project at the Vienna University of Technology for the air traffic management domain in cooperation with the Austrian company Frequentis AG. SWIS-style integration is based on messages and uses a layered approach of the semantic models. The semantic models are described by means of ontologies and are written in the Web Ontology Language (OWL). A total of three different layered ontologies are used: abstract, domain and customer ontology. The abstract ontology encompasses the basic concepts for a SWIS-based integration scenario, more precisely the concepts for the integration of different legacy applications in an air traffic management area. The domain ontology is an extension of the abstract ontology and precisely specifies the SWIS network by adding individuals to define the elements which represent the infrastructure of the underlying SWIS network. And finally, the customer ontology extends the domain ontology and specifies customer-specific information about the underlying SWIS network.

As described in Chapter 5.1, the developed SWIS approach offers some advantages compared to other traditional integration approaches. SWIS presents a good and easy to handle mechanism for the integration of numerous heterogeneous applications or systems. Because of the flexible nature by using layered semantic models and ontologies as data models, SWIS is able to easily respond to changing requirements. Only the affected models have to be changed and due to the automated validation and verification of the changed models no redesign of the already existing models is required. This is one of the big advantages of the SWIS approach. Therefore, no extensive processing steps are needed to meet the altered requirements. In contrary, in most traditional approaches changing an existing model can become a serious problem because of the dependencies between the single models. Traditional approaches often have to avoid such violations by

manual checking and if a possible violation is not noticed, the dependency violation is only recognized at the end of the development process. In contrary, in the SWIS approach no manual checks are needed, due to automated consistency checks of the semantic models. This allows only verified and validated models to be used and therefore possible violations are recognized immediately.

Another advantage of the layered semantic model structure is the way the models are created. A model designer is able to create just a comparatively small partial model for a particular system. So the model designers do not need to have the overall knowledge about the entire integration system. Furthermore, it is possible to permanently verify the single partial models without the need to have knowledge about all other models of the entire system, ensuring that only correct partial models are used throughout the entire integration process.

Furthermore, SWIS supports the easy and flexible integration of different legacy applications without the need for a common data model. The agreement on a common data model to be used for all containing heterogeneous system is very hard to or often even cannot be achieved. Already in small integration projects the agreement on a common data model can become an unsolvable problem, not to mention in case of larger integration projects. In SWIS, the different data models are mapped and the various message formats are transformed from the source message format to the target message format by using the Transformation-Maps (T-Maps). The T-Maps receives the message from a source (sending service) as input message and sends the message to the target (receiving service) as output message. In a T-Map, it is defined how the input and the output message have to be structured. To transform an input message into an appropriate output message external services or special converters can be used by the T-Map. Therefore, the non-use of a common data model offers a big strength of the SWIS integration approach.

The SWIS network not only supports the use of IP addresses, rather all possible kinds of address types can be used. Only address types which can be represented using a string are supported. Other address types the SWIS approach supports can be as different as radio communication or Morse code. The gateway node in a SWIS network acts as interface for two various address type and has the task to connect nodes with different address types. Therefore a node which only supports an IP address can communicate with a node only supporting Morse code.

SWIS also supports the use of backup nodes and therefore provides a more stable network with redundant nodes. This gives the SWIS network higher fault tolerance in case some of the network nodes fail. A backup node is a complete duplication of the primary node it belongs to with the same properties and attributes of the original node. Only the network address varies. In case if the primary node fails, the backup node takes over the further communication. The change from the primary node to the underlying backup node happens fully transparent to all other network devices.

Unfortunately, the SWIS approach needs human interaction for the final deployment of the generated integration solution. Because of its use in a safety-critical domain, the final decision to deploy the generated solution into the real productive environment is done by an authorized person. A human can better decide if the generated integration solution is

ready for deployment or not. If any errors are contained in the solution they should be found by the integration developer before the deployment during the simulation phase.

In summary, the SWIS approach offers following characteristics:

- Flexible handling of changing requirements or models by the use of layered semantic models with permanently validation and verification of the models.
- Model designers do not need to have knowledge about the entire integration system, but are able to create partial models representing a particular system.
- SWIS supports the integration of different legacy applications without the need for a common data model.
- Not only IP addresses are supported, rather all possible kinds of address types can be used if they are represented in a string format.
- SWIS supports the use of backup nodes and therefore increases the stability and reliability of the SWIS network.
- But, SWIS needs a final decision by a human to deploy the generated solution model into the real environment, due to its use in a safety-critical domain.

The developed tool support is a graphical user interface for the SWIS approach, more precisely the Model Transformation Algorithm (MTA). It offers facilitation for system integration engineers and integration project managers by visualizing emergent properties of the integrated system. The tool support helps the system integration engineers to find a specific integration solution for a specific scenario and the integration project managers to model, create and verify the integration solution with lower effort. The user interface provides a number of process-steps to choose all specific requirements for the calculation of the integration solution model. In addition, a graphical representation of the desired Network Infrastructure Model is presented to the user. By providing an easy to handle user interface for experts as well as for non-experts, the tool support leads to product improvement by the visual feedback, and to process improvement by providing better tool support and quality assurance.

A performed case study determined the benefits of the automated steps provided by the tool support to calculate an integration solution in comparison to manual calculation. The participants had to calculate and find an optimal integration solution for a very simple integration example. The case study was divided into two steps, a manual and an automated step. In the manual step, the candidates had to draw a picture of the network architecture out of the given information of the integration scenario. Then the participants had to manually determine all possible routes for the connection of the containing systems and to manually calculate the overall costs and delay time for the single routes, and specify an optimal solution to connect the single systems. In the automated step, the participants had to run the same test case with help of the tool support. Although a very simple integration example was used for the case study, several participants (one-third of all participants) made mistakes in the manual calculation of the integration solution, like calculation errors or not identifying the optimal solution. Only two-thirds of the candidates

calculated a correct solution for the given example. Considering that a very simple example without the need for complex processing was used for the case study, a comparative high part of the participants had problems in the manual execution. During the case study the time needed for the manual execution and the automatic execution was measured. As a result, the manual steps required up to six times more than the automatic steps. If noted that only a simple example was performed, the economy of time by using the tool support will be enormous if a complex example is used. The time needed for the automatic steps will be almost unchanged, but the time a human need for the manual steps will increase rapidly. Probably a human would not be able to manually find a correct, functioning and optimal integration solution for a complex integration example.

The developed SWIS approach has some similarities with a generic Model Driven Architecture (MDA) process. In both approaches, the stakeholder requirements are defined in specific models. A total of three main views about the requirements and models are used: stakeholder requirements, systemwide view and specific view. For the stakeholder requirements a generic MDA uses a Computation-Independent Model which specifies the system requirements and behavior. The SWIS approach uses Requirement and Capability Models to define the stakeholder requirements. For the systemwide view the MDA defines a Platform Independent Model, whereas SWIS is using a Logical Solution Model containing a set of collaborations which is transformed out of the stakeholder requirements. At last, for the specific view, a generic MDA uses a Platform Specific Model which is transformed from the Platform Independent Model in the systemwide view. In contrary, for the specific view SWIS uses a Technical Solution Model containing the specific integration configuration and is transformed from the Logical Solution Model.

The SWIS approach uses numerous predefined integration patterns. Integration patterns represent a reliable way for capturing the knowledge of experts who are familiar in a field the patterns stand for. They are used when no “straight-forward” solution exists because each solution is unique depending on different requirements and environmental influences. So a pattern provides just a part of the overall solution which can be individually combined with other patterns to achieve the most suitable solution for a specific scenario. In SWIS following integration patterns are used: Message Translator, Publish-Subscribe Channel, Command Message, Request-Reply, Return Address, Correlation Identifier, Dynamic Router, Recipient List, Splitter, Aggregator, Message Broker, Envelope Wrapper, Content Enricher, Content Filter and Normalizer.

References

References

[1] Hohpe, G., and Woolf, B., *Enterprise Integration Patterns. Designing, Building, and Deploying Messaging Solutions*, Addison-Wesley, 2007, ISBN: 0-321-20068-3

[2] Trowbridge, D., Roxburgh, U., Hohpe, G., Manolescu, D., and Nadhan, E.G., *Integration Patterns*, Microsoft, 2004, ISBN: 0-7356-1850-X

[3] Perry, B. W., *Java Servlet & JSP Cookbook*, O'Reilly, 2004, ISBN: 0-596-00572-5

[4] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1998, ISBN: 0-201-63361-2

[5] Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A. "The many faces of publish/subscribe" *ACM Comput. Surv. 35, 2*, Jun. 2003, pp 114-131

[6] Zhu, Y., and Hu, Y., "Ferry: an architecture for content-based publish/subscribe services on P2P networks", *Parallel Processing ICPP*, 2005, pp 427 – 434

[7] Aier, S., and Schönherr, M., "Evaluating Integration Architectures – A Scenario-Based Evaluation of Integration Technologies", *Trends in Enterprise Application Architecture*, Springer, 2006, pp. 2-14.

[8] Gašević, D., Djurić, D., and Devedžić, V., *Model Driven Architecture and Ontology Development*, Springer, 2006, ISBN: 3-540-32180-2

[9] Levy, A. Y. "Logic-based techniques in data integration". *Logic-Based Artificial intelligence*, *J. Minker, Ed. Kluwer International Series In Engineering And Computer Science, vol. 597*, 2000, pp 575-595.

[10] Teale, P., Etz, C., Kiel, and M., Zeitz, C., *Data Patterns*, Microsoft, 2003, ISBN: 0-7356-2200-0

[11] W3C Consortium, http://www.w3.org/TR/ws-arch/, 23.06.2008.

[12] Hao, G., Ma, S., Lv, J., and Sui, Y., "A Service-Oriented Data Integration Architecture and the Integrating Tree", *In Proceedings of the Fifth international Conference on Grid and Cooperative Computing (GCC'06),* IEEE Computer Society, 2006, pp 526-530

[13] Pinelle, D., "Designing CSCW Applications to Support Loose Coupling in Organizations and Groups", 2004. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.87.1870

[14] Eugster, P., “Type-based publish/subscribe: Concepts and experiences”, *ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 29, Article No. 6*, Jan. 2007

[15] Malveau, R., and Mowbray, T. J., *Software Architect Bootcamp*, Prentice Hall, 2003, ISBN: 0-1314-1227-2

[16] Miller, J., and Mukerji, J., “Model Driven Architecture (MDA)”, *Architecture Board ORMSC*, 2001, http://www.omg.org/docs/ormsc/01-07-01.pdf

[17] Kleppe, A. G., Warmer, J., and Bast, W., *MDA Explained: the Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., 2003, ISBN: 0-3211-9442-X

[18] Selic, B., “The Pragmatics of Model-Driven Development”, *IEEE Software, Volume 20, Issue 5*, Sep. 2003, pp 19-25.
DOI= http://dx.doi.org/10.1109/MS.2003.1231146

[19] Adam, N. R., Atluri, V., and Adiwijaya, I., “SI in digital libraries”, *Communications of the ACM, Volume 43, Issue 6,* Jun. 2000, pp 64-72.
DOI= http://doi.acm.org/10.1145/336460.336476

[20] Object Management Group OMG, http://www.omg.org/corba-e/index.htm, 16.07.2008

[21] Botton, D. “Interfacing Ada 95 to Microsoft COM and DCOM technologies”, *Ada Letters, Volume XIX, Issue 3*, Sep. 1999, pp 9-14.
DOI= http://doi.acm.org/10.1145/319295.319297

[22] Microsoft MSDN, http://msdn.microsoft.com/en-us/library/72x4h507.aspx, 21.07.2008

[23] Trowbridge, D., Mancini, D., Quick, D., Hohpe, G., Newkirk, J., and Lavigne, D., *Enterprise Solution Patterns Using Microsoft .NET*, Microsoft, 2003,
ISBN: 0-7356-1839-9

[24] Laumay, P., Bruneton, E., Palma, N. D., and Krakowiak, S., “Preserving Causality in a Scalable Message-Oriented Middleware”, *In Proceedings of the IFIP/ACM international Conference on Distributed Systems Platforms Heidelberg,* Nov. 2001, Lecture Notes In Computer Science, Vol. 2218. Springer-Verlag, London, pp 311-328.

[25] Object Management Group OMG, http://www.omg.org/mda/, 29.07.2008

[26] Stachowiak, H., *Allgemeine Modelltheorie*, Springer, 1973, ISBN: 3-2118-1106-0

[27] Object Management Group OMG, http://www.omg.org/mof/, 29.07.2008

[28] Fields, D. K., Kolb, M. A., and Bayern, S., *Web Development with JavaServer Pages, 2nd Edition,* Manning Publications Co., 2001, ISBN: 1-8847-7799-6

[29] Gorton, I., Thurman, D., and Thomson, J., “Next Generation Application Integration: Challenges and New Approaches”, *In Proceedings of the 27th Annual international Conference on Computer Software and Applications,* Nov. 2003, COMPSAC. IEEE Computer Society, Washington DC, p 576.

[30] Studer, R., Benjamins, V. R., and Fensel, D., “Knowledge Engineering: Principles and Methods”, *IEEE Transactions on Knowledge and Data Engineering,* 1998

[31] Mellor, S. J., Clark, A. N., and Futagami, T., “Guest Editors' Introduction: Model-Driven Development”, *IEEE Software, Volume 20, Issue 5,* Sep. 2003, pp 14-18. DOI= http://dx.doi.org/10.1109/MS.2003.1231145

[32] Seidewitz, E., “What Models Mean”, *IEEE Software, Volume 20, Issue 5,* Sep. 2003, pp 26-32. DOI= http://dx.doi.org/10.1109/MS.2003.1231147

[33] Object Management Group OMG, http://www.omg.org/docs/formal/07-11-04.pdf, 29.09.2008

[34] Mellor, S. J., Balcer, M. J., *Executable UML: A Foundation for Model-Driven Architecture,* Addison Wesley Professional, May 2002, ISBN: 0-2017-4804-5

[35] Gruber, T. R., “A translation approach to portable ontology specifications”, *Knowledge Acquisition, Volume 5, Issue 2,* Jun. 1993, pp 199-220. DOI= http://dx.doi.org/10.1006/knac.1993.1008

[36] Calero, C., Ruiz, F., and Piattini, M., *Ontologies for Software Engineering and Technology*, Springer, 2007, ISBN: 3-540-34517-5

[37] Powers, S., *Practical RDF*, O’Reilly, July 2003, ISBN: 0-596-00263-7

[38] Gruber, T. R., “Toward principles for the design of ontologies used for knowledge sharing”, *International Journal of Human-Computer Studies, Volume 43, Issue 5-6,* Dec. 1995, pp 907-928. DOI= http://dx.doi.org/10.1006/ijhc.1995.1081

[39] Gómez-Pérez, A., Fernandez-Lopez, M., Corcho, O., *Ontological Engineering*, Springer, 2004, ISBN: 978-1-85233-551-9

[40] Fensel, D., van Harmelen, F., Horrocks, I., McGuinness, D. L., and Patel-Schneider, P. F., “OIL: An Ontology Infrastructure for the Semantic Web”, *IEEE Intelligent Systems*, *Volume 16, Issue 2,* Mar. 2001, pp 38-45. DOI= http://dx.doi.org/10.1109/5254.920598

[41] Rebstock, M., Janina, F., Paulheim, H., *Ontologies-Based Business Integration*, Springer, 2008, ISBN: 978-3-540-75229-5

[42] Horrocks, I., “DAML+OIL: A description logic for the semantic web”, *IEEE Data Engineering Bulletin, Volume 25*, 2002, pp 4-9

[43] W3C Consortium, http://www.w3.org/TR/owl-features, 27.10.2008

[44] Protégé, http://protege.stanford.edu/, 04.11.2008

[45] Open Knowledge Base Connectivity OKBC, http://www.ai.sri.com/~okbc/, 04.11.2008

[46] Petrie, C. J., Webster, T. A., and Cutkosky, M. R., "Using Pareto optimality to coordinate distributed agents", *Arti Intelligence for Engineering Design, Analysis and Manufacturing (AI/EDAM), Volume 9,* 1995, pp 269-281

[47] FISN Architecture (internal document)

[48] FISN Model Transformation Algorithm (internal document)

[49] FISN Semantic Architecture (internal document)

Appendix

A. Installation Guide

This chapter explains how to install and configure a software environment to run Java Servlets and Java Server Pages (JSP). It is necessary to follow these steps to get a correct and functioning development environment. This environment is needed to start running of the developed tool support. The tool is developed as web application in JSP for the graphical user interface and uses Servlets for processing between the single steps.

Java Software Development Kit (J2SDK)

To build a java web application it is recommended to download and install the Java Platform on the computer. The Java Platform is offered in different packages, e.g. Java Standard Edition (Java SE) or Java Enterprise Edition (Java EE). But which Java version do we need to set up a right and correct environment? For this purposes it is sufficient to install the Standard Edition because we don't need Java EE features like Enterprise JavaBeans (EJB) or Java Messaging Service (JMS). I have downloaded the Java SE Development Kit (JDK) version 5.0 which includes the Java Runtime Environment (JRE) directly from the SUN website: http://java.sun.com/javase/downloads/index_jdk5.jsp.

After the installation the *PATH* environment variable must be set. *PATH* should refer to the directory that contains *java.exe* and *javac.exe*, which are typically in *java_install_directory*/bin. In my case the *PATH* variable must be set to the directory *C:\Programme\Java\jdk1.5.0_10*.

Apache Tomcat Web-Applicationserver

Apache Tomcat is a web application server which allows adding dynamic content generation written in Java. Tomcat is developed at the Apache Software Foundation and generates pure HTML out of the Java sources and so it is possible to view the existing Java web content with all common web browsers (e.g. Microsoft Internet Explorer, Mozilla Firefox, etc.). The Apache Tomcat Server can be downloaded from the apache tomcat website: http://tomcat.apache.org/.

But different Apache Tomcat versions must be used for different versions of Java Servlet and JSP specifications. Table A.1 shows the mapping between the different versions.

Servlet/JSP Specification	Apache Tomcat version
2.5/2.1	6.0.x
2.4/2.0	5.5.x
2.3/1.2	4.1.x
2.2/1.1	3.3.x

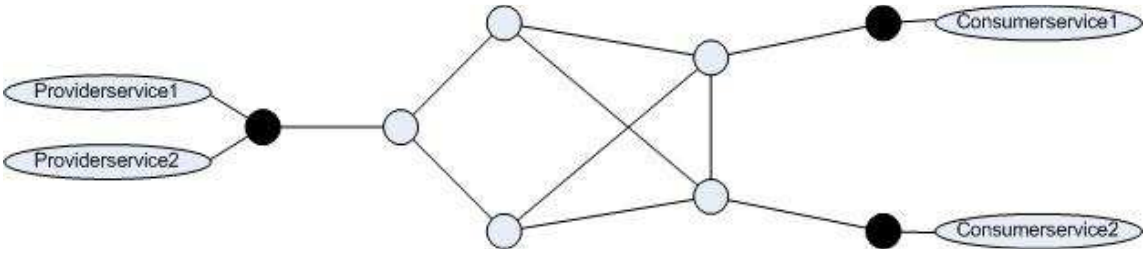
Table A.1: Servlet/JSP specification vs. Apache Tomcat version

I used the Tomcat version 6.0.14 from http://archive.apache.org/dist/tomcat/tomcat-6/v6.0.14/. After accomplishment of the installation, according to the previous described steps, the setup of the environment is finished and the tool support is ready for processing.

B. Case Study

Manual Steps

A network consists of numerous nodes and links. Each link connects two nodes. Some nodes are endpoints and on each endpoint one or more services are running. A service is either a provider or a consumer service and can send or receive a specific message. Your task is to calculate possible routes from each provider service to a consumer service. Provider and Consumer must understand the same message type. The next figure shows a possible network structure:



1. Draw the network architecture with the given tables. The network consists of nine nodes and thirteen links and has three provider services and two consumer services.

Nodes:

A	B	C	D	E	F	G	H	I

Links:

Link	Node - Node	Cost	Delay
AC	A – C	10	50
BC	B – C	10	50
BD	B – D	1	500
CD	C – D	1	500
CF	C – F	20	500
CE	C – E	1	50
DF	D – F	20	500
DE	D – E	20	500
FG	F – G	1	50
EG	E – G	1	50
EI	E – I	1	500
GH	G – H	1	50
GI	G – I	10	50

Services:

Service	Node	MessageType
Provider Service	A	Order Message
Provider Service	B	Order Message
Provider Service	B	Info Message
Consumer Service	K	Order Message
Consumer Service	L	Info Message

2. For each provider service find an optimized route to a consumer service. Note that both services (provider and consumer) must understand the same message type.

Route	Cost	Delay

3. For the three provider services choose respectively one scenario of step2 with minimal costs and one scenario with minimal delay-time. At last sum-up the total costs and delay-time of the three routes.

   Solution with minimal costs:

Route	min. Cost	Delay
Total:		

   Solution with minimal delay-time:

Route	Cost	min. Delay
Total:		

Automated Steps

Run the same test case using the tool support. Compare the automatic calculated solution with your manual calculated solution.

URL: http://127.0.0.1:8080/at.swis.mta.gui