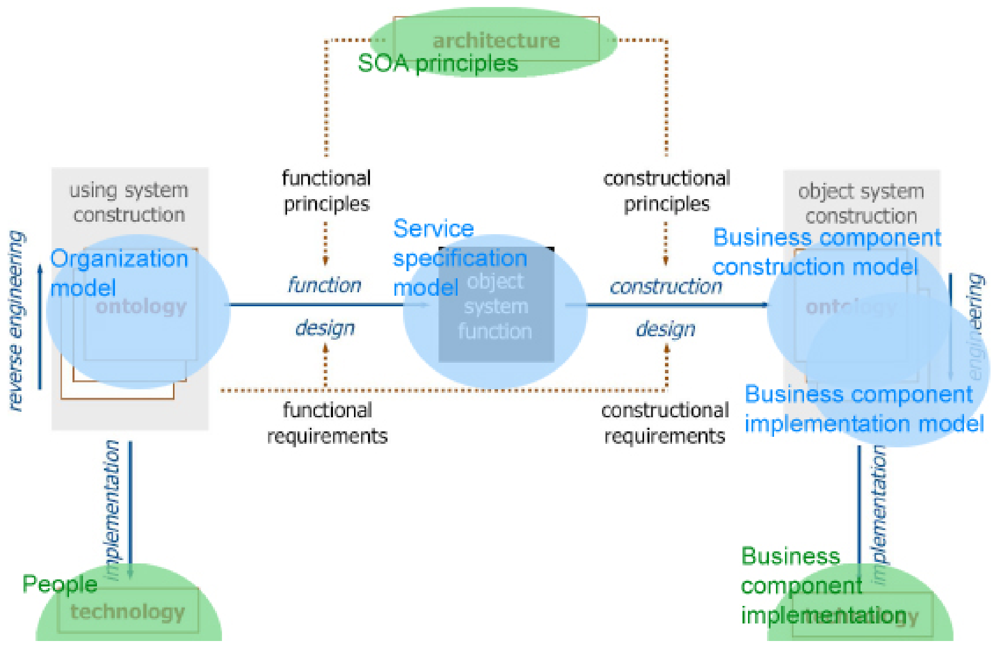
An Enterprise Ontology based approach to Model-Driven Engineering

MDEE



architecture
SOA principles
functional principles
constructional principles
using system construction
Organization model
ontology
reverse engineering
function
design
Service specification model
object system function
construction
design
object system construction
Business component construction model
ontology
engineering
Business component implementation model
functional requirements
constructional requirements
implementation
People
technology
implementation
Business component implementation

Johan den Haan

An Enterprise Ontology based approach to Model-Driven Engineering

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Johan den Haan
born in Nieuwdorp, the Netherlands

Information Systems Design
Department of Information Systems Algorithms
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

An Enterprise Ontology based approach to Model-Driven Engineering

Author:	Johan den Haan
Student id:	1174509
Email:	jdenhaan@gmail.com

Abstract

Because of the lack of a significant increase of productivity in the last 20 years we are still in a huge need for increasing the return a company derives from its software development effort. Model-Driven Engineering (MDE) aims to raise the level of abstraction in application modeling and increase automation in application development, thereby increasing the productivity in software development.

The Model-Driven Architecture tries to define an MDE approach, but is mainly focused on technical variability and lacks formalism. Other approaches, using Domain-Specific Languages, do not define a process or framework at all. Research in this area is focused on language engineering and multi-modeling. Literature on formalizing MDE is focused on defining the concepts and assets needed to construct an MDE framework. We only know one attempt on formulating an end-to-end (from business model to IT implementation) MDE approach, but the resulting framework does not have a theoretical foundation.

Concluding we can state that no end-to-end MDE approach exists describing abstraction layers, models, modeling languages, and transformations, based on a formal, theoretical foundation. This thesis presents an MDE approach based on a sound theoretical foundation, providing end-to-end guidance to refine and transform an organization model into an IT system supporting that organization.

Preface

Model-Driven Development isn't about making the computer do your work for you, but stopping you from doing the computer's work via proper separation of concerns.

- Rafael Chaves (@abstratt)

This thesis has been submitted in partial fulfillment of the requirement for the degree of Master of Science in Computer Science, with Information Architecture as specialization track. The Information Architecture track is an interfaculty master that combines courses from the faculty of Electrical Engineering, Mathematics and Computer Science and the faculty of Technology, Policy and Management, both of the Delft University of Technology.

My work on this thesis was a valuable add-on to my experience in practice with my work at Mendix. I think the combination of Model-Driven Engineering with a methodology like DEMO is very powerful. In the spirit of the subject of this thesis I did not specify this thesis in low-level TEX code, but I generated the TEX code from a higher level definition in LYX[1].

I would like to thank Arjen Hoekstra for reviewing a draft of this thesis and for providing feedback and suggestions. I also like to thank my other colleagues at Mendix for creating an innovative and stimulating working environment.

I especially like to thank my supervisors Antonia Albani and Linda Terlouw for finding time in their busy schedules to give me valuable feedback. Their insights have made this thesis much more well-founded.

Last, but definitely not least, I thank Marlinda, my real friend for life, for supporting me with everything, even in spending less time together.

Johan den Haan
Zwijndrecht, the Netherlands
September 28, 2009

[1]www.lyx.org

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Since the beginning of programming people are striving to increase the productivity of programming languages. From Assembler to BASIC productivity has increased with about 400% [86]. However, after that first leap productivity has not increased that much. The average productivity in Java is only 20% better than in BASIC. Why was the first leap so big, and how can we achieve it again today?

In this chapter we will answer these two questions and explain that a Model-Driven Engineering (MDE) approach can support the next leap in productivity (1.1). We shortly describe the existing MDE approaches used in practice (1.2) and how scientific literature tries to formalize these approaches (1.3). We identify a clear gap in this research, leading to the formulation of our research goal (1.4) and research approach (1.5). We conclude this chapter with an overview of the structure of this report (1.6) and a reading guide (1.7).

1.1 Increasing productivity

A main success factor of BASIC over Assembler was the step up to a next level of abstraction. The second success factor was that each line in BASIC could be automatically translated into Assembler. Because of the lack of a significant increase of productivity in the last 20 years (see [86]) we are still in a huge need for increasing the return a company derives from its software development effort. That is exactly the reason why MDE is getting more and more attention.

MDE aims to raise the level of *abstraction* in program specification and increase *automation* in program development [14]. The idea promoted by MDE is to use models at different levels of abstraction for developing systems [43], thereby raising the level of abstraction in program specification. An increase of automation in program development is reached by using executable model transformations. Higher-level models are transformed into lower level models until the model can be made executable using either code generation or model interpretation.

The underlying motivation for MDE is to improve productivity. This benefit is delivered in two basic ways [12]:

1. By improving the short-term productivity of developers. That is, by increasing the value of primary software artifacts in terms of how much functionality they deliver. MDE delivers this benefit by specifying a software artifact at a higher

abstraction level and by providing automatic transformations from this specification to an executable artifact.

2. By improving the long-term productivity of developers. That is, by reducing the rate at which primary software artifacts become obsolete. To do so, software artifacts should become less sensitive for changes in personnel (by specifying models using domain-specific abstractions), requirements (by providing automated transformations from a model of these requirements to a software artifact), development platforms (by storing models in an interchangeable format), and deployment platforms (by specifying software artifacts in a platform independent way).

1.2 MDE approaches

Different approaches exist trying to fulfill the goals of MDE. The best known approach is the Model-Driven Architecture (MDA) [71]. The MDA defines models at different abstraction levels and states that transformations are needed between these models. The main focus of the MDA is on technical variability, i.e. modeling systems in a platform independent way and providing automatic transformations to platform dependent models. Although the MDA also defines a computation independent abstraction level, computation independent models are not used a lot in practice.

The main problem of the MDA is the lack of formalism. In books and literature on MDA no formal model of the concepts exists [41]. The layers of abstraction presented in the MDA are not defined in a formal way too. No theory exists to determine whether a model is platform independent or dependent with respect to a certain platform definition. While every model in the MDA is specified using UML, no difference in modeling language abstraction exists between the models at the different MDA abstraction layers.

The strong coupling between the MDA and other OMG standards like UML and MOF also leads to a lack of formalism in the modeling languages, leading to vagueness, confusion, and debate. Atkinson and Kühne point at a couple of problems with the UML meta-model structure [11]. Besides that, a formal language definition is more than just a meta-model [60].

Because of the strong focus of MDA on UML, a lot of the model-driven approaches in current practice are not following the MDA. These approaches are focusing on domain-specific abstractions to reach a higher level of abstraction in program specification. In most cases these approaches are referred to as Domain-Specific Modeling (DSM), i.e. using multiple small models defined in different Domain-Specific Languages (DSLs). This leads to the use of multiple small languages, tailored to a specific domain.

Although DSM and DSLs are successfully used in industry nowadays [63], no formal framework exists describing what kind of models, languages, and model transformations are needed to successfully implement an MDE approach. Research in this area is mainly focused on language engineering (see for example [68, 98, 48]) and multi-modeling (see for example [49, 50, 28, 19, 18, 100]). The main results in the field of DSM at the moment are:

- DSLs providing abstractions on programming languages like Java and C#, thereby increasing developer productivity, and,
- DSLs for very specific domains (e.g. insurance or pension applications) which are directly transformed into an executable programming language, thereby enabling domain experts to be involved in the development process because the design freedom is restricted and the notation is familiar.

1.3 Formalizing MDE

In literature several attempts to formalize MDE have been made. A couple of researchers (e.g. Favre [41], Kühne [61], Thirioux et al. [94]) focus on formalizing the concepts of MDE, i.e. they define the concepts used in MDE: model, meta-model and transformation. This work is useful to understand what MDE is, but it does not tell how to use MDE to build software systems. A framework or process is needed, exactly describing the sequence of models to be developed, and how to derive a model from another one at the abstraction level immediately above it [43]. Fondement and Silaghi [43] define the assets needed to construct such a framework.

An approach, providing end-to-end guidance and assistance to refine and transform business models created by business experts into IT models and then to IT implementation, is still missing. The only attempt, we know about, into this direction is the recent work of Anaby-Tavor et al. [8]. In their work they define an MDE approach consisting of four different *type* of models: requirement models, enterprise models, execution models, and IT governance models. They also define the *kind* of transformations involved. They illustrate their work with an example build around an insurance information system scenario, with choices for concrete modeling languages and the mappings between the models.

Although the work of Anaby-Tavor et al. is a good example of an end-to-end MDE approach aimed at fulfilling the goals described in section 1.1, it still lacks a formal foundation. They just use a model for each phase of a service engineering process, there is no link to a formal theory describing what models to use and why.

1.4 Research goal

As we have seen in section 1.1 MDE aims to raise the level of abstraction in application modeling and increase automation in application development. As discussed in section 1.2, the MDA tries to define an MDE approach, but is mainly focused on technical variability and lacks formalism. We have also seen that the most used approach in industry, DSM, does not define a process or framework at all. Research in this area is focused on language engineering and multi-modeling. Literature on formalizing MDE is focused on defining the concepts and assets needed to construct an MDE framework. We only know one attempt on formulating an end-to-end (from business model to IT implementation) MDE approach, but the resulting framework does not have a theoretical foundation.

Concluding we can state that no end-to-end MDE approach exists describing abstraction layers, models, modeling languages, and transformations, based on a formal, theoretical foundation. Hence we formulate our research goal as follows:

> *Design an MDEE approach based on a sound theoretical foundation, providing end-to-end guidance to refine and transform an organization model into an IT system supporting that organization.*

We use the term *Model Driven Enterprise Engineering (MDEE)* to refer to an outline of the assets (e.g. models, languages) needed to define a model of an organization and the assets (e.g. transformations) needed to refine and transform that model until an IT system supporting that organization is constructed. In other words: the assets needed to describe, refine, and transform business models created by business experts into IT models and then to IT implementation.

The MDEE approach has to be based on a sound theoretical foundation underpinning the needed process steps, models, languages, and transformations. We base our MDEE approach on a theoretical foundation obtained from *enterprise ontology* [34].

In order to achieve the research goal, a number of research questions can be formulated:

1. What are the requirements for an MDEE approach?
   a) What is MDE?
   b) What is MDEE?
   c) What stakeholders are involved with MDEE?
   d) What are the goals of these stakeholders?
   e) What requirements can be extracted from these goals?
2. How can enterprise ontology provide a sound theoretical foundation for an MDEE approach?
   a) What is enterprise ontology?
   b) How is enterprise ontology related to MDEE?
3. How can an MDEE approach be designed with use of enterprise ontology?
   a) What abstraction layers are needed?
   b) What models are needed?
   c) What refinements and transformations are needed?

Because of the broad subject the following constraints apply to this research:

- The resulting MDEE approach will be focused on a top-down approach. Although it is necessary in practice to combine a top-down approach with a bottom-up approach to incorporate existing IT systems in the resulting implementation, we leave this as future work.

- To specify the needed models we will use existing modeling languages as much as possible. If no modeling language exists fulfilling our needs we need to define a custom language (e.g. by specializing an existing language). In such cases we define the basic elements of the language, we do not provide a full formal language definition.

- The end result of the MDEE approach needs to be an executable implementation model. We do define this model and explain how to execute it. However, we do not specify a full example including all needed details to execute it. We also do not implement the needed technology to execute the implementation model.

1.5 Research approach

Our research is focused around the construction of an artifact, an MDEE approach. In our research approach we follow the guidelines for design-science research described by Hevner et al. [51]:

1. *Design as an artifact*. The product of our research is a viable artifact in the form of the description of an approach, i.e. a description of the process steps and the artifacts to produce.

2. *Problem relevance*. Because of the lack of a significant increase of productivity in software development in the last 20 years we are still in a huge need for increasing the return a company derives from its software development effort in both building new software and changing existing software. As we have seen MDE can increase productivity, but no formal approach exists describing how to reach that goal. Our research will provide a formal MDE approach providing end-to-end guidance to refine and transform an organization model into an IT system supporting that organization.

3. *Design evaluation*. Within the scope of our research we can not execute a rigorous evaluation. We will show the applicability of the resulting MDEE approach with a running example (i.e. the example grows along with the research and explanation of our MDEE approach).

4. *Research contributions*. We base our research on the theoretical foundation of enterprise ontology. The combination of MDE and enterprise ontology has not been researched before. As we have shown no formal MDE approach exists providing end-to-end guidance to refine and transform an organization model into an IT system supporting that organization. We refer to section 9.1 for a complete overview of all contributions.

5. *Research rigor*. The construction of the MDEE approach will be done using the formal foundation of the Ψ-theory of enterprise ontology [34].

6. *Design as a search process*. This is beyond the scope of this research.

7. *Communication of research*. We follow this guideline by producing a concise research report, readable for both technology-oriented and management-oriented audiences.

Our research approach consists of several phases, each phase resulting in concrete results. We distinguish the following phases:

1. *Formulation of research goal.* The result of this phase is this chapter.
2. *Research the requirements of an MDEE approach.* The result of this phase is an explanation of the assets of an MDEE approach and their relations.
3. *Research enterprise ontology.* The result of this phase is a description of enterprise ontology and a description how an MDEE approach can be based on enterprise ontology.
4. *Design MDEE approach.* The result of this phase is the actual MDEE approach based on the research performed in the previous phases.
5. *Evaluate MDEE approach.* Evaluate the constructed approach with a real-life example. The result of this phase is a set of example artifacts created following the MDEE approach.
6. *Write thesis report.* This phase results in a research report.

An overview of the research phases, the deliverables, and the applicable guidelines is shown in Table 1.1.

Research phase	Deliverable	Guideline
Formulation of research goal	Thesis proposal	Problem relevance, Research contributions
Research MDEE requirements	Explanation of MDEE approach concepts and their relations	Research rigor
Research enterprise ontology	Description of enterprise ontology and a description how the MDEE approach can be based on enterprise ontology	Research rigor
Design MDEE approach	MDEE approach	Design as an artifact
Evaluate MDEE approach	Application of the approach on an example case	Design evaluation
Write thesis report	Research report	Communication of research

Table 1.1: Research phases

1.6 Report structure

This report consists of nine chapters. We will give a short description of each chapter.

Chapter 1 describes the background, motivation, objective, scope, and approach of this research.

Chapter 2 gives an overview of the related work which also focuses on describing approaches guiding the transformation and refinement of organization models into a working IT system.

Chapter 3 explains the theory of Enterprise Ontology in detail.

Chapter 4 gives a formal definition of MDE and MDEE and describes the requirements for an MDEE approach.

Chapter 5 describes the organization model and the reverse engineering step producing the organization model.

Chapter 6 describes the service specification model and the function design step producing this model based on the organization model.

Chapter 7 gives a description of the business component construction model and the construction design step producing this model. The previously produced models in the MDEE approach are the input for the construction design step.

Chapter 8 defines the business component implementation model and the engineering step producing this model. The previously produced models in the MDEE approach are the input for the engineering step. This chapter also describes the implementation step by describing how to implement the business component implementation model on technology.

Chapter 9 concludes this report with an overview of the contributions, an evaluation of the MDEE approach, the conclusions of this research, and a description of what we see as future work.

1.7 Reading guide

Table 1.2 presents the reading guide for this thesis.

If you are interested in...	you should read:
... a quick overview of the resulting MDEE approach	Section 9.2
... an overview of the contributions of this research	Section 9.1
... the background, objective, scope and approach of this research	Chapter 1
... the used scientific foundation for the MDEE approach	Chapter 3
... a formal definition of MDE and MDEE	Section 4.1
... an overview of the requirements for MDEE approaches	Section 4.6
... how to model organizations in MDEE	Chapter 5
... how to identify and specify services in MDEE	Chapter 6
... how to identify and specify components in MDEE	Chapter 7
... how to implement IT systems without programming	Chapter 8
... an evaluation of the MDEE approach	Section 9.3
... what work still has to be done	Section 9.4

Table 1.2: Reading guide

Chapter 2

Related work

The goal of this research is to design an approach to MDEE based on a sound theoretical foundation, providing end-to-end guidance to refine and transform an organization model into an IT system supporting that organization. In this chapter we present related work focusing on frameworks that describe how to transform an organization model into an IT implementation.

Most existing methodologies are centered around the idea of service orientation. Two methodologies cover the whole development process, namely the methodology of Papazoglou and Van den Heuvel [82], and SOMA [10]. Terlouw [93] concludes that both methodologies do not describe all phases very thoroughly. Both methodologies do not specify formal models and automated transformations between these models.

As our research is focused on Model-Driven Engineering (MDE) we only see methodologies using formal models and automated transformations as related work. We have found three works describing a model-driven approach covering the development process from organization model to IT system. We first describe the Model Driven Architecture (MDA) (2.1), followed by the Business to IT Transformation Framework (2.2) and the Model Driven Service Engineering Process (2.3).

2.1 Model Driven Architecture

The term MDA was first mentioned in 2000 in an OMG whitepaper [87]. Based on this whitepaper OMG members decided to form an architecture team to produce a more formal statement of MDA. This formal but still incomplete definition of the MDA was presented in 2001 in the document "Model Driven Architecture - A Technical Perspective" [78]. OMG members voted to establish the MDA as the base architecture for their organization's standards in late 2001. In 2003 a more detailed definition of MDA, presented in the document “MDA Guide Version 1.0.1” [71], was adopted by the members of the OMG.

The MDA provides a framework for software development that uses models to describe the IT system to be built. The system descriptions that these models provide can be expressed at various levels of abstraction, with each level emphasizing certain aspects or viewpoints of the system [66].

The MDA defines three levels of abstraction called Computational-Independent Model (CIM), Platform-Independent Model (PIM) and Platform-Specific Model

(PSM). Each of these levels focuses on different concerns of the software system being developed. The CIM models the IT system in an independent way to that of any computational system, that is, it makes up either a domain model, or a requirements model, etc. The PIM models the IT system from a computational viewpoint but independent of any underlying platform, and the PSM models the IT system for a specific platform.

Amaya et al. [7] and Kent [58] extend the MDA with additional dimensions, like system aspects and subject areas. Each model (CIM, PIM, and PSM) is split into different aspect models to improve the use of MDA for complex systems. However, they do not specify the needed elements for each dimension. They, for example, do not state what aspects are needed to successfully model a complex IT system supporting an organization.

2.2 Business to IT Transformation Framework

Stein, Kühne, and Ivanov [88] have conducted an extensive literature research in the field of model-driven business to IT transformations. They conclude that the reviewed works only cover a subset of the different criteria and present a new framework providing the building blocks of business to IT transformations from a practical point of view.

Mens and Van Gorp [67] distinguish between horizontal and vertical transformations. A horizontal transformation is a transformation where the source and target models reside at the same abstraction level. A vertical transformation is a transformation where the source and target models reside at different abstraction levels. According to Stein, Kühne, and Ivanov [88] most approaches for business to IT transformations follow a horizontal transformation strategy. A horizontal transformation starting with an abstract business process model results in an abstract orchestration model requiring significant refinement efforts to make it executable. An alternative horizontal transformation strategy is to start from a more detailed business process model (i.e. a process model augmented with technical details). This will result in a more detailed orchestration model which is directly executable. However, this forces the business analyst to think in terms of executable business processes.

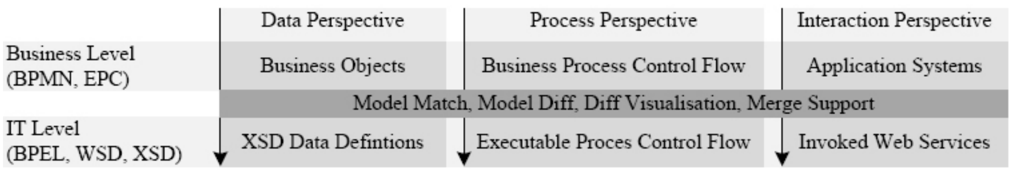
Due to the previous arguments, Stein, Kühne, and Ivanov [88] formulate the following axiom, which forms the basic assumption for their business to IT transformation framework:

> Business process models must be platform independent and a platform specific IT implementation must be derived through a vertical transformation strategy.

Business process models can for example be defined in the Business Process Modeling Notation (BPMN) [72] or Event-driven Process Chains (EPC) [57]. A platform specific IT implementation can for example be specified in the Business Process Execution Language (BPEL) [70].

Consecutively they come to the following consequences and requirements for their framework:

- The business process model should not contain any platform specific details, e.g. references to webservice interfaces defined in the Web Service Description Language (WSDL) [23].
- Source models should be restricted to a subset, which can be unambiguously transformed.
- Full code generation is desirable but not achievable.
- Target models should be comprehensible for human users (i.e. users which have experience with the language of the target model).
- The framework should use iterative development processes through change detection, change visualization, and merge functionalities (i.e. if the same model is changed by different users it should be possible to merge these changes into one version).



	Data Perspective	Process Perspective	Interaction Perspective
Business Level (BPMN, EPC)	Business Objects	Business Process Control Flow	Application Systems
	Model Match, Model Diff, Diff Visualisation, Merge Support		
IT Level (BPEL, WSD, XSD)	XSD Data Defintions	Executable Proces Control Flow	Invoked Web Services

Figure 2.1: Business to IT transformation framework [88]

Figure 2.1 exhibits the general business to IT transformation framework [88]. The framework shows three different model perspectives: process, data, and interaction. These perspectives reduce cognitive complexity at design time [88]. The framework also presents two abstraction levels, the business level and the IT level, which are related with vertical transformations.

2.3 Model Driven Service Engineering Process

Anaby-Tavor et al. [8] define a model driven service engineering process aimed at providing a methodology giving end-to-end guidance and assistance to refine and transform business models created by business experts into IT models and then to IT implementation. A service engineering process is a sequence of activities: analysis, strategy design, development, and deployment, done in this order to produce a services system [8].

Figure 2.2 presents the methodology in three layers. The upper layer shows the service engineering process, the middle layer describes the models supporting each phase of the service engineering process, and the bottom layer describes the aspects of the services system that the models affect. Anaby-Tavor et al. follow the definition of a services system given by Zang et al. [102] which states that a services system is a 6-tuple: <Inputs, Outputs, Goals, Transformation, Components, Sensors>.

In the analysis phase the system requirements are analyzed and modeled. In the next phase a strategy design is conducted to describe the static and dynamic aspects of the services system. Static enterprise models define the components' black box view.

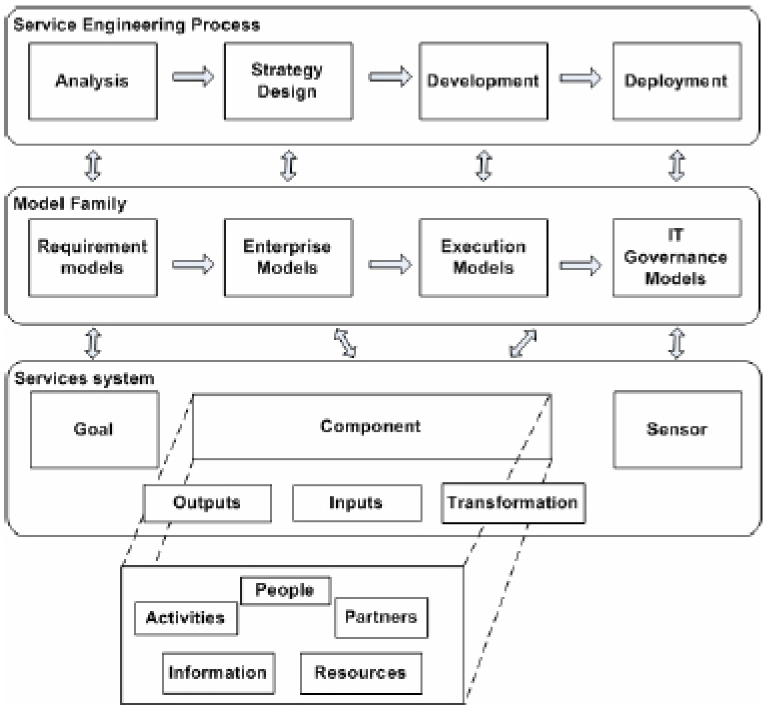


Figure 2.2: The Model Driven Service Engineering Process [8]

The components collaborate using services, this forms the first step of the dynamic modeling of the enterprise. During the development phase detailed executable data is added leading to execution models describing the control flow of the activities of the services. Finally, sensors use IT management models to oversee the deployment of the services system.

Chapter 3

Enterprise Ontology

As we have shown in the previous chapter, a formal, theoretical foundation for MDE is needed. In this chapter we explain why enterprise ontology is ideal for providing such a foundation (3.1). We also explain the basic notions of enterprise ontology (3.2) and its four axioms (3.3), resulting in its central Organization Theorem (3.4).

3.1 Why Enterprise Ontology?

Enterprise ontology is focused on the essence of the operation of an organization, meaning that it is fully independent of the (current) realization and implementation of the organization. The theory that underlies the notion of enterprise ontology as presented by Dietz [34] is called the Ψ-theory [30]. Dietz uses this theory to construct a methodology providing an ontological model of an organization, i.e. a model that is coherent, comprehensive, consistent, and concise, and that only shows the essence of the operation of an organization model. This methodology is called Design and Engineering Methodology for Organizations (DEMO).

DEMO has been widely accepted in both scientific research and practical appliance. It has been used as a base for formalizing enterprise architecture and governance [54, 53, 32] and for formalizing the splitting and allying of enterprises [77]. Further research has extended DEMO by constructing an ontological model of an information system supporting the enterprise model (the BCI-3D method) [4]. More recently enterprise ontology has been used to construct a formal framework for service specification [47]. An extensive ten year study executed with 28 projects concluded that DEMO is a good method for the fast (re)design of organizations [69].

Since the Ψ-theory provides a very sound theoretical foundation for a layered view on organizations and how to distinguish these layers, this theory provides us with a formal foundation for an MDEE approach providing end-to-end guidance to refine and transform an organization model into an ICT system supporting that organization.

3.2 The notion of System and Model

Two important notions in enterprise ontology are the notion of System and the notion of Model. While we use these notions when explaining the axioms and theorems of enterprise ontology, we will first explain these notions.

Two different system notions exist, the teleological and the ontological system notion [34]. The teleological system notion is concerned with the function and the (external) behavior of a system. This notion is identical to one of the model types that are applied to systems, the black-box model. We will explain the notion of model in the remainder of this section.

According to Dietz [34] the ontological system notion is the only true system notion. In this notion something is a system if an only if it has the following properties [34]:

- *Composition*: a set of elements of some category (physical, social, biological, etc.).
- *Environment*: a set of elements of the same category; the composition and the environment are disjoint.
- *Production*: the elements in the composition produce things (e.g., goods or services) that are delivered to the elements in the environment.
- *Structure*: a set of influence bonds among the elements in the composition, and between them and the elements in the environment.

The composition, the environment, and the structure, are collectively called the *construction* of the system [34]. The collective activity of the elements in the composition and the environment is called the *operation* of the system [34].

For the notion of model in enterprise ontology Dietz reuses the definition of model given in [9], which states:

> *Any subject using a system A that is neither directly nor indirectly interacting with a system B, to obtain information about the system B, is using A as a model for B.*

In general, three categories of systems can be distinguished: concrete systems, symbolic systems, and conceptual systems. The following relationships among these systems are exhibited in Figure 3.1 [34]: *imitation*, a concrete model of a concrete system; *conceptualization,* a conceptual model of a concrete system; *implementation,* a concrete model of a conceptual system; *conversion*, a conceptual model of a conceptual system; *formulation,* a symbolic model of a conceptual system; *interpretation,* a conceptual model of a symbolic system; and *transformation,* a symbolic model of a symbolic system.

Two fundamental different types of models can be distinguished: the white-box model and the black-box model [34]. The white-box model is a direct conceptualization of the ontological system definition, it captures the construction and the operation of a system, while abstracting away from implementation details. The white-box model can be seen as the *construction perspective* on a system and is adequate for the purpose of building or changing a system [34].

The black-box model only takes the interactions between the composition and the environment into account. A black-box model of a system can be made and used without knowing its construction and operation. The black-box model can be seen as the *function perspective* on a system and is adequate for the purpose of using a system [34].

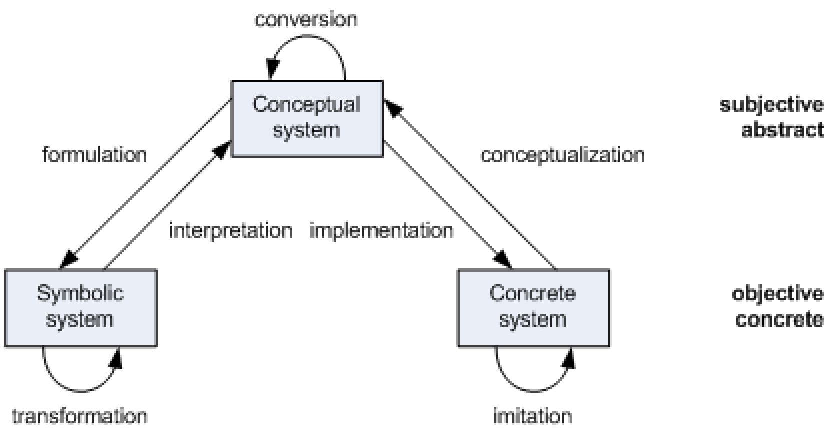


Figure 3.1: The model triangle [34]

3.3 The four axioms of Enterprise Ontology

The overall goal of the Ψ-theory is to extract the essence of an organization from its actual appearance. It presents four axioms that help to achieve this goal [34].

The *operation axiom* tells us that the implementation independent essence of an organization is that it consists of subjects fulfilling actor roles. A subject fulfilling a certain actor role is called an actor. Actors constitute the operation of an organization by performing two kinds of acts: production acts and coordination acts.

By performing production acts (P-acts for short) the subjects contribute to bringing about the goods and/or services that are delivered to the environment of the organization [34]. The results of P-acts are production facts (P-fact for short), which can be divided into material (something is manufactured, stored or transported) and immaterial (decisions or judgments) facts.

By performing coordination acts (C-acts for short) subjects enter into and comply with commitments towards each other regarding the performance of production acts [34]. A C-act is performed by one actor, the performer, and directed to another actor, the addressee. C-acts consist of an intention (e.g. request, promise, question, assertion) and a proposition (the performer proclaims the fact and the associated time the intention is about) and result in coordination facts (C-facts for short).

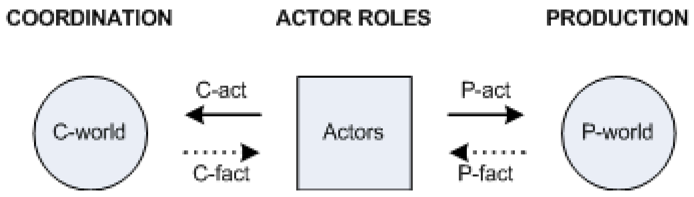


Figure 3.2: Graphical representation of the operation axiom [34]

A graphical representation of the operation axiom is presented in Figure 3.2. As visualized, P-acts have effect in the so-called production world (P-world for short), C-acts have effect in the coordination world (C-world for short). The state of a world at a particular point in time is the set of facts created up to that point of time [34]. The creation of a fact of some type is a state transition in one of the two worlds.

The *transaction axiom* tells us that C-acts are performed as steps in universal patterns, called transactions. This axiom reveals universal socionomic patterns of coordination that hold for all organizations [34]. The standard pattern of a transaction is shown in Figure 3.3. A white box represents a C-act type and a white disk represents a C-fact type. A gray box represents a P-act type and a gray diamond a P-fact type.

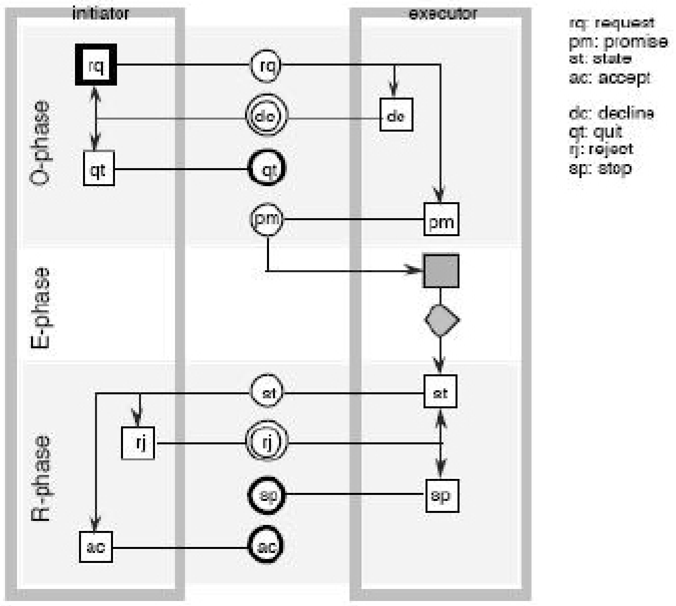


Figure 3.3: The standard pattern of a transaction [34]

Two actors are involved in a transaction, the initiator and the executor. A transaction evolves in three phases: the order phase (O-phase for short), the execution phase (E-phase for short), and the result phase (R-phase for short) [34].

In the O-phase the initiator and executor work to reach an agreement about the intended result of the transaction, i.e., the P-fact the executor is going to create as well as the intended time of creation [34]. As shown in Figure 3.3 the initiator starts the transaction by requesting something from the executor. The C-fact ’rq’ is created and the executor can either decline the request (after which the initiator can request again or quit the transaction) or accept the request by promising to deliver the requested P-fact (the C-fact ’pm’ is created).

In the E-phase, the P-fact agreed upon in the O-phase is actually brought about by the executor [34]. The gray box and diamond in Figure 3.3 reflect this phase.

A transaction ends with the R-phase in which the initiator and the executor work to reach an agreement about the P-fact that is actually produced, as well as the actual time of creation (which may both differ from what was agreed upon in the O-phase). Only if this agreement is reached the P-fact will come into existence [34]. In figure 3.3 this phase starts when the executor states that the delivery of the P-fact has been done (C-fact ’st’ is created). The initiator can either reject (after which the executor can state again or stop the transaction) or accept (thereby ending the transaction). The result of a successful transaction is the creation of a P-fact.

The *composition axiom* tells us how P-facts are interrelated. It states that every transaction is enclosed in some other transaction, or is a customer transaction of the organization under consideration, or is a self-activation transaction [34]. According to Dietz this axiom provides the basis for a well-founded definition of the notion of business process. Which states [34]:

> A business process is a collection of causally related transaction types, such that the starting step is either a request performed by an actor role in the environment (external activation) or a request by an internal actor role to itself (self-activation).

The *distinction axiom* tells us that actors exert three basic human abilities: performa, informa, and forma. Through the distinction axiom a substantial reduction of complexity and diversity is achieved, regarding both the coordination and the production in an organization [34].

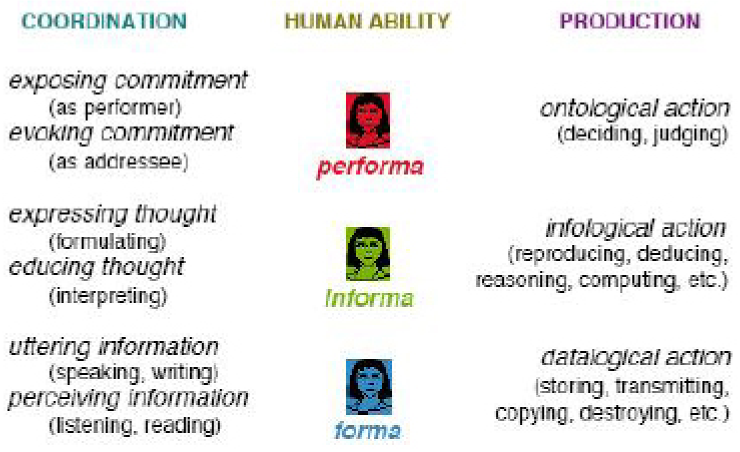


Figure 3.4: Summary of the distinction axiom [34]

The *forma* ability concerns the form aspects of communication and information [34]. As shown in Figure 3.4 communicative acts at the forma level are about uttering (speaking, writing) and perceiving (listening, reading). Production acts at the forma level are datalogical in nature, meaning that they store, transmit, copy, destroy, etc. information.

The *informa* ability concerns the content aspects of communication and information [34]. Communicative acts at the informa level express thoughts (formulating) or educe thought (interpreting). Production acts at the forma level are infological in nature, meaning that they reproduce, deduce, reason, compute, etc. information. Note that on this level we deal with communication and information while fully abstracting from the form aspects. Figure 3.4 summarizes the different acts on this level.

The *performa* ability concerns the bringing about of new, original things, directly or indirectly by communication [34]. As can be seen in Figure 3.4 communicative acts at the performa level are about exposing or evoking commitment. Production acts at this level are ontological in nature, meaning that they decide or judge about something. The performa ability is considered as the essential human ability for doing business of any kind [34]. This additional layer is important as it explains and clarifies the organizational notions of collaboration and cooperation, as well as notions like authority and responsibility.

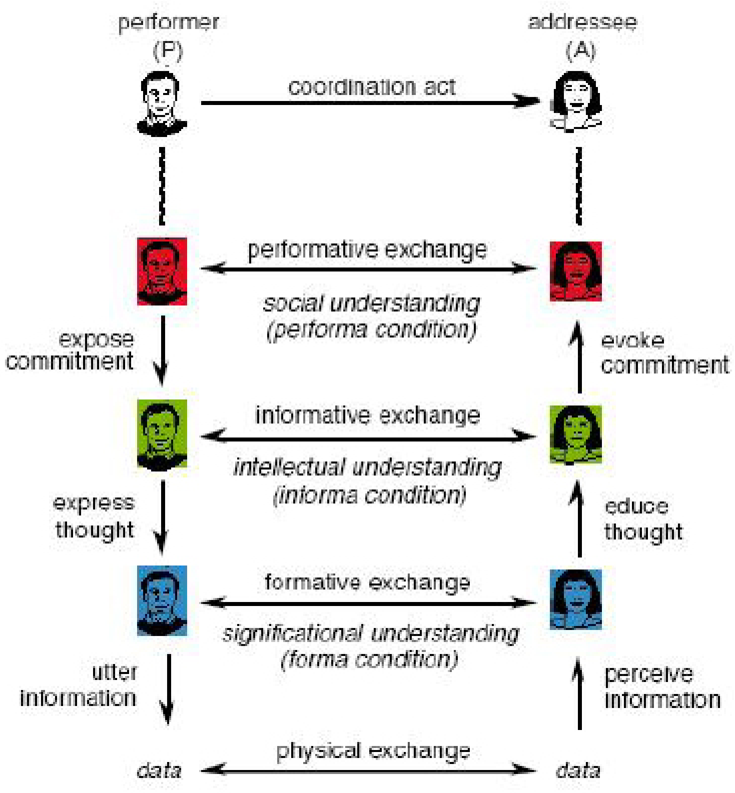


Figure 3.5: The process of performing a coordination act [34]

In order to successfully perform a C-act (as presented before in the operation axiom) the communicative acts at all three levels have to be performed, as visualized in Figure 3.5. The performer (P) has to expose his commitment to the addressee (A) to perform the C-act, i.e., P and A have to come to a social understanding. The only way to gain that understanding is by reaching an intellectual (or semantic) understanding. P has to express his thoughts, while A has to educe this thoughts and to confirm that she has (intellectually) understood P's information. P can only express a thought by formulating it in a sentence in some language, and by uttering it in some form, such as speaking or writing. A, on the other hand, can only educe the thought by listening or reading it. So, another level of understanding is needed, called significational (or syntactic) understanding. Lastly, we of course need some physical exchange of data, i.e., the field of (tele)communication technology. Dietz assumes the condition of correct physical transmission to be included in the forma condition [34].

We can make the similar distinction between the three levels of abstraction on the production side. Examples of acts on each level are mentioned before and summarized in Figure 3.4. Transactions having an ontological P-act as result are called ontological transactions. Infological transactions, i.e., transactions having an infological P-act as result, are considered to be a matter of realization of the ontological transactions [34].

Infological transactions only reproduce or derive knowledge that is needed by the ontological actors. Datalogical transaction, i.e., transactions having an datalogical P-act as result, are in their turn, considered to be a matter of realization of the infological transactions. They only care for the storage and retrieval, the copying, and the transmission of documents containing information that is needed by the infological actors [34].

3.4 The Organization Theorem

In the previous section we have presented the four axioms of enterprise ontology. As stated before they all serve the overall goal of the Ψ-theory to extract the essence of an organization from its actual appearance. The operation axiom abstracts the implementation independent essence of an organization to actors, acts and facts. The transaction axiom brings another reduce in complexity by combining acts in universal patterns holding for each organization. The composition axiom brings us the notion of business process and lastly the distinction axiom reduces complexity even more by providing a distinction between acts on different layers of human ability.

The *organization theorem* combines the benefits of these axioms into one concise, comprehensive, coherent, and consistent notion of enterprise [34]. This theorem states that the organization of an enterprise is a heterogeneous system that is constituted as the layered integration of three homogeneous systems: the B-organization (from Business), the I-organization (from Intellect), and the D-organization (from Document) [34]. We call these three systems the aspect systems of the (total) organization. Figure 3.6 visualizes the organization theorem and shows us that the D-organization supports the I-organization, which supports the B-organization. The coordination parts of these three systems are similar, they only differ in the kind of production: the production of the B-organization is ontological, the production in the I-organization is infological, and the production in the D-organization is datalogical.

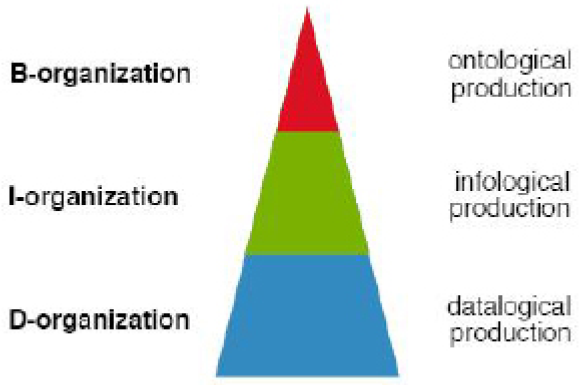


Figure 3.6: Representation of the organization theorem [34]

Dietz draws a clear distinction between the realization and the implementation of an organization. By the realization of an organization he means the thorough integration of the three aspect organizations. The implementation of an organization is the making operational of the organization's realization by means of technology [34].

The integration of the three aspect organizations is established through the layered nesting of them. As said before a system in some layer supports the system in the next higher layer, e.g., the D-organization supports the I-organization, which supports the B-organization. Controversily, a system in some layer uses the system in the next lower layer [34]. When a layer supports another layer its function supports the construction of the other layer, e.g., the function of the I-organization supports the construction of the B-organization. The distinction of the function perspective (F) and the construction perspective (C) provides a more elaborated view on the layered integration of organizations and is visualized in Figure 3.7.

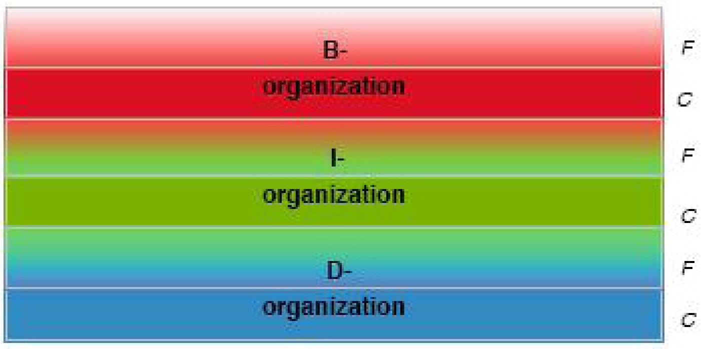


Figure 3.7: The layered integration of an organization [34]

As said before, Dietz defines the implementation of an organization as the making operational of the organization's realization by means of technology. He refers to the possible ways of implementation of actor roles, C-acts, and P-acts as *actor role technology*, *coordination technology*, and *production technology* [34].

Figure 3.8 shows how an organization is related to an ICT system. As exhibited production and coordination technology consist of different kinds of applications, B-applications, I-applications, and D-applications, supporting the B-organization, I-organization, and D-organization respectively.

Since we see a lot of ICT related technology to *support* human actors, the question is to what extend it is possible to *replace* these human actors and use ICT instead of humans to implement the organization. Let's look at coordination acts, production acts, and actor roles respectively.

The process of performing a coordination act involves exchange of information on several levels (recall Figure 3.5). These levels can be fulfilled by ICT in the following way [34]:

- Physical exchange: can very well be implemented with hardware.
- Formative exchange: can be implemented by D-applications (see the examples mentioned before).
- Informative exchange: this means that ICT systems have to establish intellectual understanding. This can be faked by making information systems interoperable using formal languages instead of natural languages.

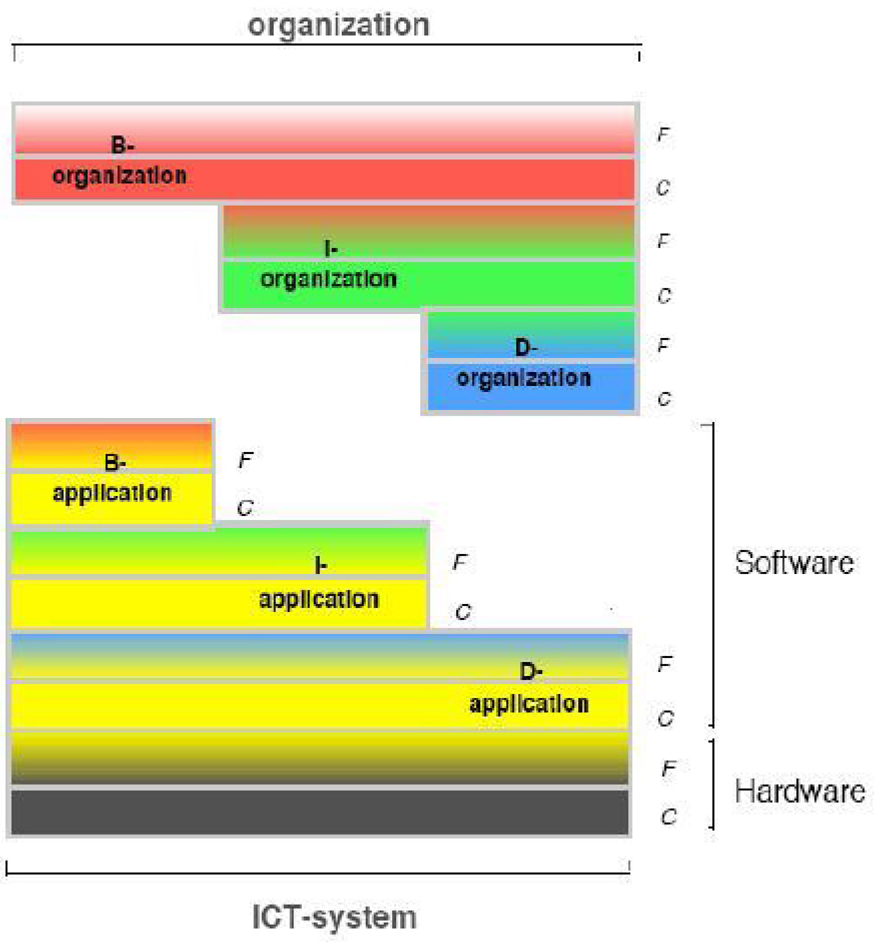


Figure 3.8: Organization and ICT system [34]

- Performative exchange: the main question for this level is, could one hold machines responsible, in the real social sense? According to Dietz this is not possible. However, it is possible to use the same trick as with informative exchange, but then there would no longer be any distinction between the two levels. This is happening a lot in modern integrated systems. Dietz sees the confusion about responsibilities, data ownership, etc. as a result of this lack of distinction.

Production acts on the different levels can be fulfilled by ICT in the following way [34]:

- D-organization: P-acts on this level are storing, transporting, copying, etc. of documents. If these documents are digital, lots of applications exist doing these jobs.

- I-organization: P-acts on this level are the computing or deducing of new knowledge from existing knowledge. With use of formal languages, in particular, of algorithmic languages, this can be supported by ICT systems.

- B-organization: P-acts on this level are original actions, like making decisions or judgments. According to Dietz it is principally impossible to have machines perform such actions.

Actor roles can not be taken over completely by ICT systems because of the performative exchanges in which they inevitably participate, and because of the ontological production acts they have to perform. As we concluded before, both things cannot be performed by ICT systems [34].

Regarding D-actor roles and I-actor roles the conclusion is that almost all of their acts can be taken over by ICT systems, only the responsibilities for the well-functioning of the systems are human roles [34]. B-actor roles on the other side can only be implemented with human resources [34].

Hence B-applications in the sense that they take over acts at the ontological level of an organization cannot really exist. Numerous ICT applications seem to be 'real' B-applications, like Business Process Management Systems and ERP systems. However, they are just I-applications, but Dietz gives them the separate status of B-application in order to distinguish them from the pure (I-organization supporting) I-applications.

So, B-applications are precisely what Figure 3.8 displays: B-organization supporting ICT applications. The only difference with I-applications is that the results of their calculations are taken as 'real' decisions or judgments. According to Dietz this is acceptable as long as it is fully clear who is responsible for making these decisions or judgments [34].

Chapter 4

Model Driven Enterprise Engineering requirements

Before we are going to define a formal MDEE approach based on the theory of enterprise ontology presented in the previous chapter, we need to get a better understanding of the requirements for MDEE. In this chapter we perform an in-depth requirements analysis after we first define MDE formally (4.1) and give some background explanation about MDE (4.2). We start our requirements analysis with an actor analysis (4.3). Based on this analysis we research two subjects in more detail: modeling complex systems (4.4) and implementing complex systems (4.5). This chapter is concluded with an overview of the requirements for an MDEE approach (4.6).

4.1 Model-Driven Engineering

In section 3.4 we have seen how the organization theorem classifies software as an ICT system. While MDE is not tied specifically to ICT systems, we can see MDE as an approach for developing systems. System development is the bringing about of a new system or of changes to an existing system [31]. The system development process comprises all activities that have to be performed in order to arrive at an implemented system [31]. Such a development process can be described by the Generic System Development Process (GSDP) as introduced in [32] and exhibited in Figure 4.1.

The GSDP always concerns two systems, the object system (i.e. the system which is going to be developed) and the using system (i.e. the system that is going to use the functionality of the object system). The GSDP can be divided into four sub processes: reverse engineering, designing, engineering, and implementation.

While in a lot of cases no construction model of the using system is available, we need to reconstruct these higher level models from the implementation model. This process is known as reverse engineering.

The design process is split into function design, which starts from the construction of the using system and ends with a functional (black-box) model of the object system, and construction design, which starts with the function of the object system and ends with a construction (white-box) model of the object system.

The engineering of a system is the activity in which the highest level construction model is converted into lower level models until the implementation model is

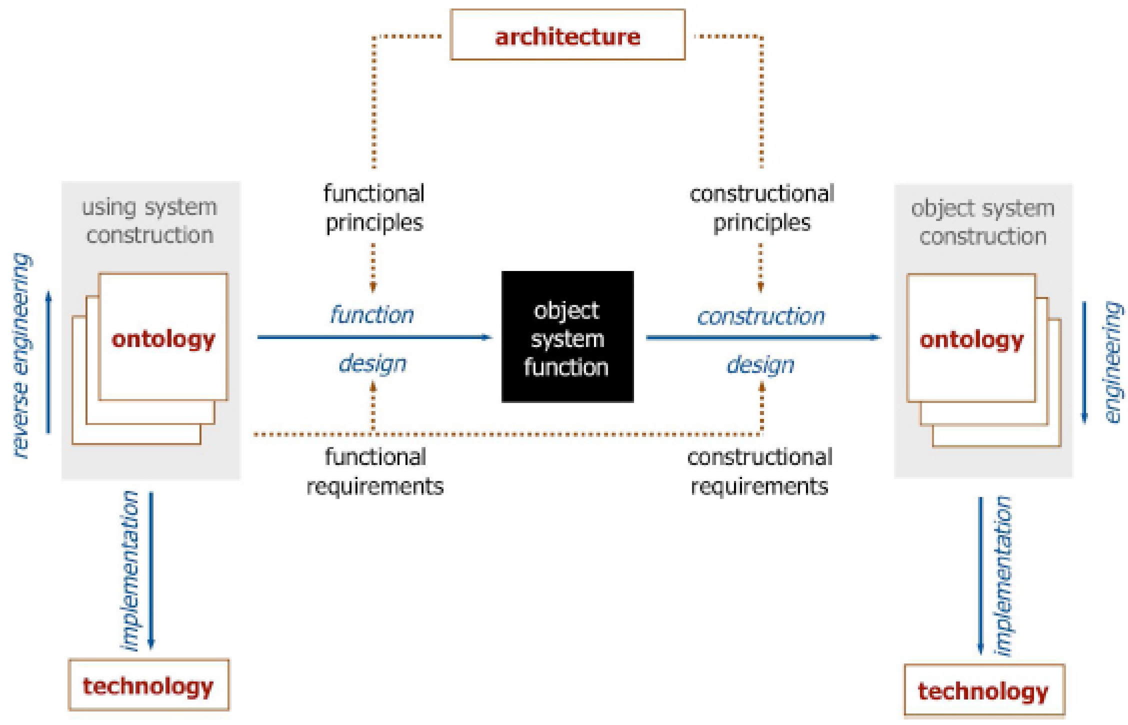


Figure 4.1: The generic system development process [31]

constructed. Dietz defines the highest level construction model as the ontology of a system, i.e. a model of the construction of a system that is completely independent of the way in which it is implemented [31].

Implementing a system means assigning technological means to the constructional elements in the implementation model [31]. In case of an implementation model expressed in a programming language, the implementation is the compilation of that model to a specific platform. Once a system is implemented it can be put into operation.

The last part of the GSDP, as exhibited in Figure 4.1, is the architecture. Dietz uses the prescriptive notion of architecture, in contrast to the descriptive notion of architecture. The descriptive definition defines architecture as 'blue-prints', for example the definition of Zachman [101] which states: "Architecture is that set of design artifacts, or descriptive representations, that are relevant for describing an object, such that it can be produced to requirements as well as maintained over the period of its useful life". Dietz states that this definition already has a name: system ontology [31]. He therefore adopts the prescriptive notion of architecture which he defines as [32, 31]:

> Theoretically, architecture is the normative restriction of design freedom.
>
> Practically, architecture is a consistent and coherent set of design principles.

In the GSDP, architecture is split into functional principles guiding the function design and constructional principles guiding the construction design.

So, what is MDE? As stated before in section 1.1, MDE aims to raise the level of *abstraction* in program specification and increase *automation* in program development [14]. The idea promoted by MDE is to use models at different levels of abstraction for developing systems [43]. The basic set of concepts in MDE are models, metamodels, and transformations [41]. Combining these ideas with the GSDP, our definition of MDE is:

MDE is a system development process consisting of:

- five main *steps*:
  - reverse engineering,
  - function design,
  - construction design,
  - engineering, and
  - implementation.
- a number of *formal models* produced in each step. At least the following models are needed:
  - a construction model of the using system (produced in the reverse engineering step),
  - a functional model of the object system (produced in the function design step),
  - a construction model of the object system (produced in the construction design step), and
  - an implementation model of the object system (produced in the engineering step).
- a set of mostly *automated transformations* connecting the models.

Following the GSDP an MDE process can contain iterations between steps. As stated in our research goal (section 1.4) we focus on an MDE process starting with an organization model and producing an implementation model which can be implemented on ICT technology. Mapping this on the GSDP we see that the using system is an organization. The object system is an ICT system supporting that organization. We refer to this process as Model Driven Enterprise Engineering (MDEE), i.e. MDE applied to Enterprise Engineering.

In the remainder of this chapter we will research the requirements for an MDEE approach. In the remainder of this thesis we will define an MDEE approach fulfilling these requirements.

4.2 MDE background

Before analyzing the requirements in detail we want to explain some common terms and definitions used in the realm of MDE. We first take a look at modeling languages. After that we explain model transformation and the implementation of models.

An MDE framework has to specify the modeling language for each used model. A language for defining specific types of models, is mostly called a Domain-Specific Language (DSL) in contrast to a General Purpose Language (GPL). A DSL can be defined as [29]:

> A DSL is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.

In practice also non-executable modeling languages are considered as DSL. The domain specificity of a language is a matter of degree. Any language has a certain scope of applicability, but some of them are more focused than others. Languages can be compared using levels based on the number of language statements needed for implementing a function point[1]. This results in a table ranging from low level languages like natural language, machine language and assembly to high level languages which are targeted at a very specific domain [86]. Although the difference between a GPL and a DSL is a matter of degree, mostly only the more focused languages are considered as a DSL, such as BNF, HTML or SQL. Other examples of often used DSLs are the Business Process Modeling Notation (BPMN) [72], Event-driven Process Chains (EPC) [57], and Object Role Modeling (ORM) [45]. As the difference between GPL and DSL is a matter of degree, which is not well-defined, we will just use the term modeling language in the remainder of this thesis. If needed, we separately specify the domain of the modeling language.

A sound language description contains an abstract syntax, one or more concrete syntax descriptions, mappings between abstract and concrete syntax, and a description of the semantics [60]. The abstract syntax defines the concepts of a language and their relationships. The concrete syntax defines the physical appearance of language. For a textual languages this means that it defines how to form sentences. For a graphical language this means that it defines the graphical appearance of the language concepts and how they may be combined into a model. The semantics describe the meaning of a sentence or model specified in some language. In the context of MDE this means that the semantics of a model describe what the effect is of implementing that model.

In MDE the process of building IT systems is one of transforming high-level models into executables (considered as yet other models) [14]. Transforming a model into another model means that a source model is transformed into a target model based on some transformation rules.

As model transformation is a critical component in the MDA [71] the OMG has been working on a model transformation standard for a while. In 2007 the final specification of the Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification [76] has been released. This specification, better known as QVT, defines metamodels for defining model transformation languages. The resulting languages can transform source models into target models where the source and target models may conform to an arbitrary MOF [74] metamodel. The transformation language itself is also a model and conforms also to a MOF metamodel.

[1] A function point is a unit of measurement to express the amount of business functionality an information system provides to a user. Function points are the units of measure used by the IFPUG Functional Size Measurement Method.

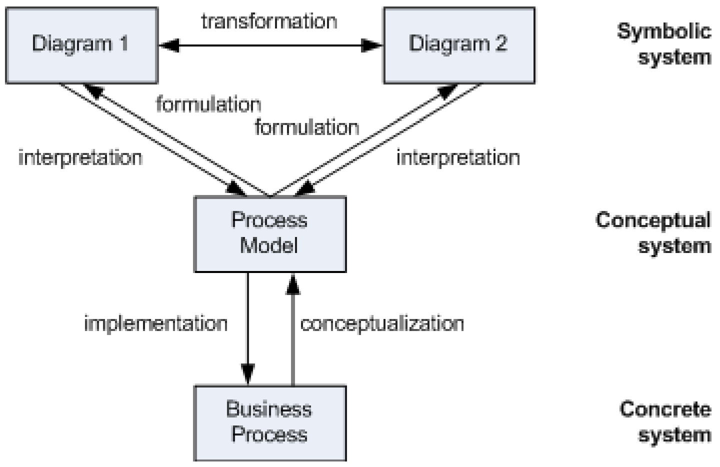


Figure 4.2: Transformation example

A formal definition of transformation is given by Dietz [34]. He states that a transformation is a symbolic model of a symbolic system (recall Figure 3.1 in section 3.2). Let's consider the example shown in Figure 4.2. A business process is an implementation of a conceptual system, the process model. This process model can be formulated with different diagrams. Diagram 1 can for example be a flow chart, while Diagram 2 can be a DEMO process diagram. A transformation can be defined between these two diagram. They are both a formulation of the same conceptual system, but they are expressed in different modeling languages. Using a set of transformation rules a flow chart can be transformed into a DEMO process diagram and back.

A transformation can be executed by an IT system or human resources. The first implementation is referred to as an automated transformation, the latter as a manual transformation. As stated before, one of the goals of MDE is to automate system development. Hence the goal is to have automated transformation as much as possible. This is of course only possible if all information needed to execute the transformation is available formulated in formal models. In the example mentioned before the transformation from a flow chart to a DEMO process diagram can only be automated if the flow chart contains additional information, e.g. is an action datalogical, infological, or ontological. The transformation from a DEMO process diagram to a flow chart can easily be automated, because no information needs to be added.

The goal of transforming models is to eventually come to an executable format, i.e. a model that can be implemented on technology. In current practice this executable format is most often source code, i.e., the model expressed in a programming language. However, it is also possible to construct engines, directly executing higher level models. So, in principle we can distinguish between:

- Code generation: the act of generating source code (in a programming language like Java or C#) from a model. This can be done in various ways, for example by using templates and rewrite rules or by constructing a metamodel of a programming language and defining formal transformations from the metamodel of the model to the metamodel of the programming language.

- Model execution: the direct execution of a model on an engine, as defined by the semantics of the model. No transformations are defined, the engine is a generic software program parameterized by the model.

In case of code generation we can consider source code as just another model (the implementation model). Hence, we still need model execution, albeit on a low level. The problem with this approach is that a model expressed in source code becomes part of the MDE process, meaning that we need to specify very detailed transformations. That's why we prefer technology consisting of engines executing higher level models. In the remainder of this thesis we will focus on model execution as the way to implement models on technology.

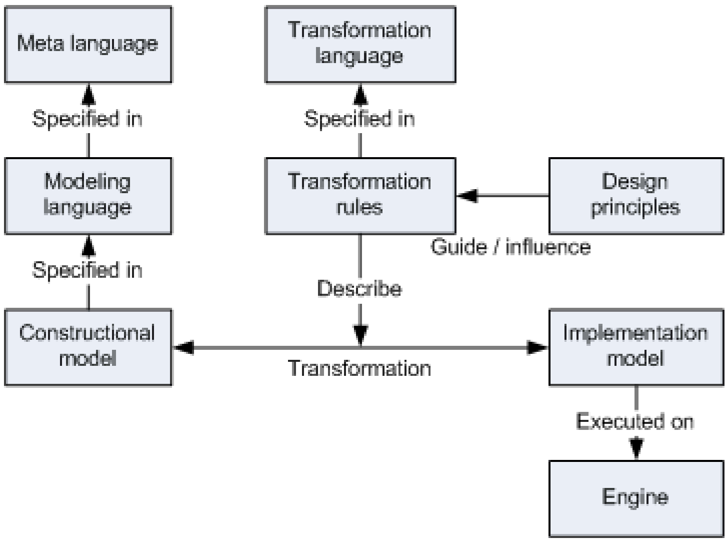


Figure 4.3: MDE overview

Figure 4.3 exhibits the concepts explained in this section. For readability purposes it only shows the last part of MDE, the engineering of a construction model into an implementation model. The full MDE process also contains the transformation from the construction model of the using system to the functional model of the object system and the transformation of this functional model to the construction model of the object system. Even the part visualized in Figure 4.3 can be extended with multiple transformations between construction models before the implementation model is created.

As shown in Figure 4.3, a construction model is specified in a modeling language, which on its turn is specified in a meta language. Recall that a language description consists of an abstract syntax, concrete syntax, and semantics. The construction model can be transformed into an implementation model using a transformation, specified with transformation rules influenced by design principles. These rules are specified in some transformation language. As we explained in this section the implementation model is implemented by executing it on an engine. While the transformation is guided by design principles, the implementation model and the engine both conform to these design principles. A consistent set of design principles is often referred to as architecture (as explained in section 4.1), which becomes important when implementing complex systems. In section 4.5 we will research this subject in more detail.

4.3 Actor analysis

In section 4.1 we have seen the elements of MDE and we introduced MDEE. In this section we perform a short actor analysis. We will analyze the actors involved with MDEE and look at what they expect from an MDEE approach based on their goals.

Regarding roles in MDE, Aagedal and Solheim [1] distinguish between two main groups of roles: the meta team and the project team. The meta team defines the MDE approach and the project team uses this approach to develop the IT system. While we are specifying an MDE approach as part of this thesis, we focus on the users of such an approach, the project team.

Aagedal and Solheim [1] distinguish three roles in the project team: application designer, system analyzer, and system tester. However, these roles are not based on a formally defined MDE process. In our MDE definition in section 4.1 we defined five steps which are part of the MDE process: reverse engineering, function design, construction design, engineering, and implementation. As MDEE focuses on an organization as using system and an IT system as object system, we can indicate the following roles:

- *Business analyst*: performs the reverse engineering step, i.e. a business analyst creates an organization model for a given organization.

- *Function designer*: performs the function design step, i.e. a function designer creates a model of the function of the IT system based on an organization model. This role can be compared to what is called a requirements engineer in some companies.

- *Construction designer*: performs the construction step, i.e. a construction designer creates a model of the construction of the IT system based on the functional model. This role can be compared to what is called a software architect in some companies.

- *Software engineer*: performs the engineering step, i.e. a software engineer creates an implementation model of the IT system based on the construction model. He can use multiple models in between these two models if needed. In principle he adds all technical details needed to come to an executable model.

- *Technology expert*: performs the implementation step, i.e. a technology expert implements the implementation model on technology. This means he configures the engines needed to execute the implementation model. This role can be split in multiple roles depending on the number and kind of engines used. In practice this role is often combined with the role of software engineer.

Note that one person can fulfill multiple roles. It depends on the size of the project team how the roles are spread among the team members. It is possible to add the role of quality engineer or system tester, as mentioned by Aagedal and Solheim, but we assume this role is contained in the software engineer and technology expert role. We leave detailed research into the quality aspects of an MDE approach for future work.

Besides the actors directly using the process the following actors are indirectly involved with the MDEE approach:

- *Business owner*: the business owner is the one paying for the resulting IT system. The business owner is affected by the alignment of the resulting IT system with his business processes, the time-to-market of an IT system, and of course the total costs of an IT system.

- *End-user*: the users of the resulting IT system are affected by the effectiveness of the MDEE approach. Aspects like quality, robustness, and fit-for-use are important for someone using the IT system.

Table 4.1 shows an overview of the identified actors, along with their goals, present or expected situation, causes for that situation, and courses of solutions. With ”Present or expected situation” we mean a situation in which no model-driven approach is used. The organization and functional model are often non-formal and the construction model does not exist. The implementation model is manually specified with a low-level programming language.

The actor analysis exhibited in Table 4.1 shows that to fulfill the needs of the different actors an MDEE approach should at least use formal models and mostly automated transformations between these models. Both needs are fulfilled by MDEE by definition (see the definition of MDEE in section 4.1).

Another important need is to use high-level models. With high-level we mean that the models are as abstract as possible and do not include every tiny implementation detail. This leads to the requirement that an MDEE approach should be able to execute high-level implementation models. As explained in section 4.2 this asks for technology (engines) being able to execute high-level models. In order to come to more detailed requirements we also need to answer the question how complex systems can be modeled with high-level, formal models. We will answer this question in section 4.4.

The last requirement which can be derived from the actor analysis is that an MDEE approach should use an architecture guiding the transformations to implementation models in such a way that they result in easy to change applications with separated business logic and infrastructure code. In section 4.5 we will research what architecture an MDEE approach should use to support the implementation of complex systems.

4.4 Modeling complex systems

Organizations are very complex systems consisting of lots of actors and transactions. Hence, an ICT system supporting such an organization may also be very complex (see for example [84]). The question we try to answer in this section is how we can create an abstract representation (i.e. high level model) of such a system which makes the designing and engineering of that system more intellectually in control.

As shown by Hessellund, Czarnecki, and Wasowski [49] large and complex systems can be accurately abstracted by the use of multiple DSLs. They state that in

	Interests / goals	Present or expected situation	Causes	Courses of solutions
Business owner	IT systems can be build fast, with high quality, and at low costs when they are needed.	IT system construction is time consuming and costs a lot of money [84].	Building and changing an IT system involves a lot of low-level manual work, which costs a lot of time and leads to bugs.	Automation in building IT systems and the use of an architecture which encourages to build easy to change IT systems.
End-user	The IT system really supports the daily work.	The daily process has to be adapted to the IT system instead of the other way around.	A big gap exists between programming languages and business requirements which is filled with all kinds of manual translations, leading to insufficiencies. It also costs a lot of time to build and change applications, in the mean time the business requirements may have changed.	Model business requirements more formal and use a higher level programming language to reduce the gap between the business requirements and implementation and to enable (partly) automated transformations. Use an architecture which encourages to build easy to change IT systems.
Reverse engineer	To capture the essence of an organization with a model easy to read and use by function designers.	Organizations are becoming too complex to understand and capture with models.	Organizations are often treated with a black-box approach [31]. Modeling techniques are not based on a well-founded theory and do not capture the organization in a coherent, comprehensive, consistent, and concise way [34].	Use the DEMO methodology to reverse engineer an organization.
Function designer	To design IT systems really supporting the daily work within an organization.	The function designer cannot meet business wishes and expectations. System requirements are defined badly [22].	No formal model is used to specify the function of an IT system and no automatic transformations exists between organization, functional, and constructional model. Hence, a gap exists between these models.	Use a formal functional model and define (partly) automated transformations between the organization model and the functional model and between the functional model and the constructional model.
Construction designer	To handle the complexity of IT systems and keep them easy to build and easy to change.	Enterprise software has become too complex to be effective [84].	According to Cynthia Rettig [80] this situation comes from a proliferation of complexity in software code. It is estimated that for every 25% increase in complexity in the tasks to be automated, the complexity of the software solution itself rises by 100%.	Use high-level construction models abstracting away from implementation details. Link these models with automated transformations to the actual implementation. Use an architecture encouraging reuse of autonomous building blocks.
Software engineer	To turn high-level models and requirements into a working IT system.	Building or changing IT systems is cumbersome and involves low-level repetitive work.	Business logic is interwoven with infrastructure code. The currently available techniques for software engineering involve low-level programming languages.	Use a higher level programming languages. Use an architecture which separates the business logic from the infrastructure code. Use and reuse building blocks.
Technology expert	To configure the implementation so that the resulting IT system is reliable and scalable.	Few IT projects truly succeed [22]. Versions of technology follow each other fast and are often incompatible with previous versions.	Among others: badly defined system requirements, use of immature technology, sloppy development practices [22].	Use a well-defined methodology, mature technology and link the implementation with automated transformations to formal requirements models.

Table 4.1: Actor analysis

complex projects multiple DSLs are usually necessary in order to cope with different concerns. In other terms: multiple domain-specific models (DSMs), specified in different DSLs are needed to accurately abstract complex systems. Warmer and Kleppe [100] advocate the use of many small, loosely-coupled models, because they are easier to handle and improve readability in comparison with a single, monolithic model. Dietz [34] also states that for keeping models of complex systems manageable the technique of constructional or functional (de)composition is often applied.

So, complex systems can be decomposed in multiple smaller systems. In the same way, the model of that system can be decomposed in multiple DSMs. A model consisting of several DSMs is often referred to as a multi-model. While these DSMs model different system aspects, they can be notated in different languages, thereby enabling the use of very specific languages.

In literature multi-modeling is defined as the act of combining diverse models, which can be done in two ways [19]:

- *Hierarchical multi-modeling*: hierarchical compositions of distinct modeling styles are combined to take advantage of the unique capabilities and expressiveness of the distinct modeling styles.

- *Multi-view modeling*: distinct and separate models of the same system are constructed to model different aspects of the system.

If we refer to multi-modeling we mean multi-view modeling as defined above. While in multi-modeling a complete system is described using multiple interdependent DSMs, the integration of these DSMs is needed to synthesize the described system.

Because complex systems ask for a lot of DSMs to model them, it is important to structure or categorize these models. We call all the models together modeling a system, the modeling space of that system. One way to structure the modeling space is to categorize model perspectives. The MDA does so by distinguishing between Platform Independent Models (PIMs) and Platform Specific Models (PSMs). A more generic way to categorize perspectives is to identify the orthogonal criteria which can be used to distinguish between one perspective and another. These criteria can be thought of as different dimensions in an n-dimensional space, and a perspective is then at the intersection of different positions along those axes [58]. To make this more clear, we will look at the different types of dimensions and an example with multiple dimensions and models based on the intersection of these dimensions.

The MDA presents the platform specific/platform independent dimension which, in principle, is an example of the more general abstract/concrete dimension. In this dimension we can only talk about whether a model is abstract (or platform independent) with respect to some other model. It depends on the viewpoint whether a model is platform independent or platform specific (abstract or concrete), but whenever a model is platform specific it must include all details needed for implementation [71].

Kent [58] defines two additional categories of dimensions. The first category contains various dimensions of concern. The first dimension of concern comes from the distinction of models on the base of the subject area they belong to. Different users of an IT system may have different viewpoints on that system, focusing on a subset of the system features. For example, the part of the IT system dealing with customers, or the part of the IT system concerned with processing orders. The second dimension of concern enumerates aspects of an IT system. Examples of aspects are data, presentation, concurrency control, security, distribution and error handling.

The second category of dimensions is about organizational issues. The dimensions in this category include authorship, version (as in version and configuration control), location (in case the system development is distributed across sites) and stakeholder (such as business expert or programmer) [58].

In IT system development projects it is necessary to determine the important dimensions for a particular project. In most projects authorship and versioning will be important, but other dimensions need the identification of points of interest, like which subject areas play a role and which system aspects are of importance. Another important decision is the levels of abstraction to be used in the process and which stakeholders are involved (for which understandable models have to be made). When building models in a system development process, each model can be placed at an intersection of the dimensions.

Dimension	Possible positions
Abstract/Concrete	CIM, PIM, PSM
Subject areas	order entry, customer portal, back-end administration, ...
Aspects	data, presentation, security, business rules, workflows, ...
Stakeholders	domain expert, programmer

Table 4.2: Example dimensions for a software development project

For the sake of concreteness, Table 4.2 shows some example dimensions and their possible positions. Examples of models for the dimensions presented in Table 4.2 are:

- A Computation Independent Model (CIM) of the workflows of the order entry part of the IT system aimed at a domain expert.
- A Platform Independent Model (PIM) of the data of the back-end administration part of the IT system aimed at a domain expert.
- A Platform Specific Model (PSM) of the security of the customer portal part of the IT system aimed at a programmer.

The various dimensions at an intersection play an important role in the choice for a modeling language for that particular model. The modeling language is, for example, influenced by the subject area, the stakeholders and the level of abstraction.

As we have seen in this section, creating a high level representation of a complex system asks for multi-models. Hence, we can can extend the requirements for an MDEE process as follows: an MDEE process should define the needed multi-models, including their dimensions and aspect models.

4.5 Implementing complex systems

In section 4.3 we have concluded that an MDEE approach needs to define an architecture guiding the transformations to implementation models. As we explained in section 4.1, we follow the prescriptive notion of architecture, meaning that we see architecture as a consistent and coherent set of design principles. In this section we research what architecture we need to fulfill the needs of the actors of MDEE as explained in section 4.3. This means the architecture should:

- enable to build complex systems in such a way that they are easy to change.
- encourage the reuse of autonomous building blocks.
- ensure the separation of business logic and infrastructure code.

The concept of Service-Oriented Architecture (SOA) aims at designing an IT system in such a way that it aligns with the business (i.e. the using system) and that it can be effectively assembled and reconfigured in response to changing business requirements [37]. From a prescriptive notion of architecture, SOA can be defined as a consistent and coherent set of design principles that need to be taken into account in the development process of services [93]. Erl [36] provides an overview of these principles.

The implementation of a set of services is mostly referred to as a component. So, a component can be seen as an IT-asset implementing one or more services. Theoretically a component is not linked to any environment. A more practical definition is given by Szyperski [89]: "*A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties*". A *business component* provides a set of services out of a given business domain through well-defined interfaces and hides its implementation [42].

The Service Component Architecture (SCA) [79], an industry standard for SOA, makes the same distinction between components and services. Within the SCA, a component consists of a configured piece of implementation providing some business function. An implementation can be specified in any technology, including other SCA composites. The business function of a component is published as a set of services.

While we are defining an MDEE process for developing an IT system supporting an organization, the architecture part of our framework can be based on SOA. Examples of functional principles guiding the function design are given by Erl [36]. The result of using these principles is that the functional model of the object system consists of service specifications. Examples of constructional principles guiding the construction design are the principles behind the SCA. We can define the highest level construction model of the object system as a set of business components.

Using such an architecture enables to build complex IT systems which are still easy to change in response to changing business requirements. This architecture also separates business logic and infrastructure code and encourages the reuse of autonomous building blocks. Hence, we can conclude this section by adding the following requirement: an MDEE process should define an architecture. This architecture should specify both functional and constructional principles. As we have seen in this section, SOA is a good out-of-the-box candidate fulfilling the needs of the actors involved in MDEE.

4.6 Conclusion

In this chapter we gave a formal definition of MDE and we called MDE applied to Enterprise Engineering MDEE. We also researched the background of MDE, its actors and their problems and needs, and the requirements for an MDEE approach. The conclusion of this chapter is that an MDEE approach should consist of five main steps: reverse engineering, function design, construction design, engineering, and implementation. In each step a formal model will be produced, at least the following models are needed:

- A construction model of the using system (produced in the reverse engineering step). As the using system is an organization, this model is an *organization model*. The model needs to be based on enterprise ontology.
- A functional model of the object system (produced in the function design step). As seen in section 4.5 complex systems may be implemented using a SOA. In a SOA the function of a system is described with services. That's why the functional model of the object system is a *service specification model*.
- A construction model of the object system (produced in the construction design step). In a SOA the services may be implemented with so-called components (see section 4.5), hence the construction model of the object system is a *business component construction model*.
- An implementation model of the object system (produced in the engineering step). This model should be directly executable on technology (i.e. engines). We call this model a *business component implementation model*.

Figure 4.4 exhibits the basic elements of an MDEE approach. In the remainder of this thesis we will define an MDEE approach consisting of these elements. Each model in this MDEE approach should comply to the following requirements:

- It should be defined in a formal modeling language. For each language the abstract syntax, concrete syntax, and semantics need to be specified.
- To accurately abstract large and complex systems it needs to be a multi-model. For each multi-model the dimensions and aspect models need to be specified.
- It needs to be derived from the previous models in the MDEE approach with mostly automated transformations.

The resulting MDEE approach fully complies to the formal definition of MDE given in this chapter and will fulfill the needs of its actors.

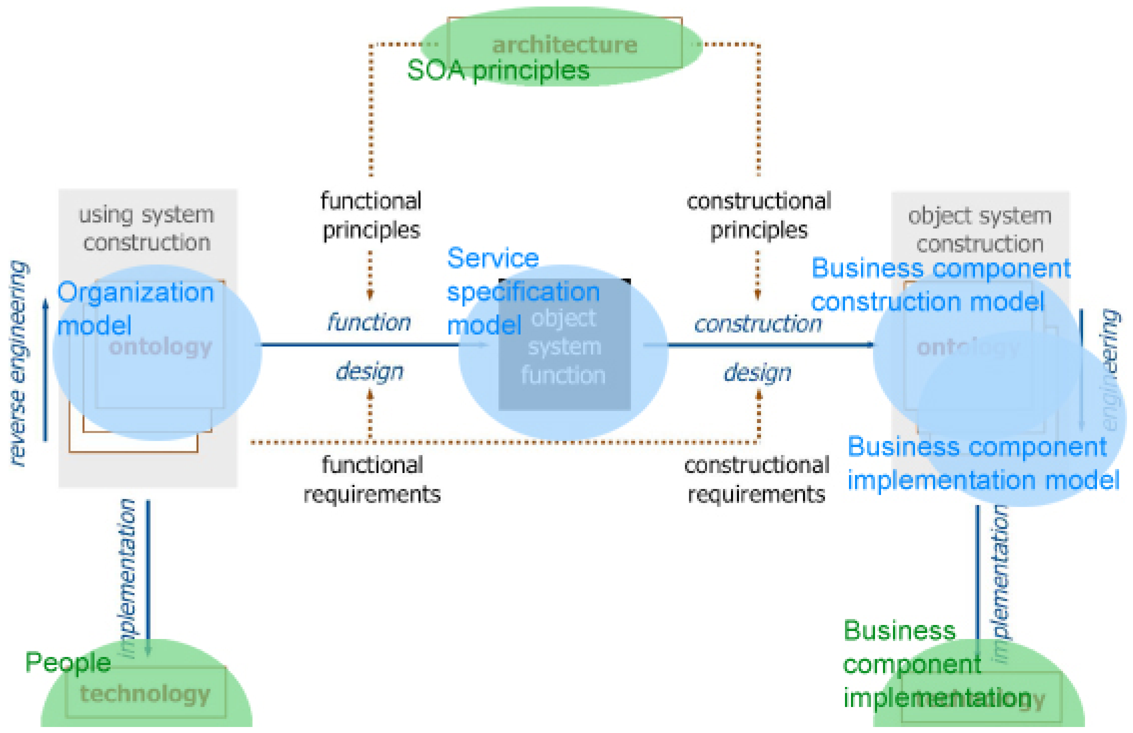


Figure 4.4: Engineering an ICT system supporting an organization

Chapter 5

An Enterprise Ontology based Organization Model

Now the theory of enterprise ontology (Chapter 3) and the requirements for an MDEE approach (Chapter 4) are clear, we are going to define the elements of our MDEE approach with use of enterprise ontology. In this chapter we look at the first model, the organization model. We first explain how the organization model can be split into multiple aspect models (5.1), what these models specify and how to come to these models (5.2). We then introduce the Protector case (5.3) and construct an example organization model for this case (5.4).

5.1 Organization multi-model

In chapter 3 we have seen that an organization can be considered as a layered integration of three subsystems: the B-organization, the I-organization, and the D-organization. The distinction between these layers is formally defined with the distinction axiom (see section 3.3). For the purpose of (re)designing the essence of an organization only a model of the B-organization is needed as shown in DEMO [34]. However, if we want to use a model of the organization for the purpose of building an ICT system supporting that organization, we also need the information of the I-organization and the D-organization.

The integration of the three aspect organizations is established through the layered nesting of them. When a layer supports another layer its function supports the construction of the other layer, e.g., the function of the I-organization supports the construction of the B-organization and the function of the D-organization supports the construction of the I-organization. The integration of these layers is established through the cohesive unification of the human being [34, 31].

De Jong [26] explains this integration in more detail. He introduces the concept of shaping, i.e. an actor shapes into an actor of another aspect organization to be the initiator of a transaction in that aspect organization. A transaction in another aspect organization cannot be directly initiated because the initiator of a transaction must be from the same category as the executor of the transaction [20]. Hence, a B-actor needs to shape into an I-actor to request information from the I-organization. Shaping is possible by the cohesive unification of the human being, i.e. a human being can fulfill the actor roles of each aspect organization.

Due to the number of actors in an organization a full model of the organization can become quite big and complex. Dietz uses the approximate proportions 1:4:7 for the number of actor roles in respectively the B-, I-, and D-organization [34]. This means that if we start with 100 actor roles in the B-organization model (which is not that much in practice), we end up with 500 actor roles in the B+I-organization model, and 1200 actor roles in the B+I+D-organization model.

As we have seen in section 4.4 a possible dimension for categorizing models is the system aspect dimension. In the DEMO approach the B-organization is modeled with four different aspect models [34]. The B-organization model can thus be seen as a multi-model consisting of a:

- *Construction model*: the construction model specifies the construction of the organization, i.e. the identified transaction types and associated actor roles, as well as the information links between actor roles and information banks;
- *Process model*: the process model specifies the transaction pattern for each transaction type, as well as the causal and conditional relationships between transactions. In principle the process model models the coordination part of an organization (C-acts and the state and transition space of the C-world);
- *State model*: the state model specifies the state space of the P-world, i.e. the object classes, fact types, result types, and the ontological coexistence rules; and
- *Action model*: the action model specifies the action rules that serve as guidelines for the actors in dealing with their agenda. In other words: the action rules guide the actors in what act to perform based on the current state of the C- and P-world.

The same aspect models can be used for the I-organization and D-organization. This means that a model of an organization consists of twelve aspect models categorized by two dimensions: the aspect dimensions and the distinction dimension. Table 5.1 shows an overview of the aspect models and what part of the organization they model.

	Construction	Process	State	Action
B	B-transactions, B-actor roles	Coordination, ontological P-acts	Objects and facts used in the B-organization	Action rules for B-actors
I	I-transactions, I-actor roles	Coordination, infological P-acts	Objects and facts used in the I-organization	Action rules for I-actors
D	D-transactions, D-actor roles	Coordination, datalogical P-acts	Objects and facts used in the D-organization	Action rules for D-actors

Table 5.1: Aspect models of an organization

The construction model of the B-organization models B-transactions (i.e. transactions with ontological P-acts) and B-actor roles (i.e. actor roles executing B-transactions). In the same way the construction model of the I-organization models I-transactions and I-actor roles, and the construction model of the D-organization models D-transactions and D-actor roles.

The process model of the B-organization models the coordination part of an organization, i.e. the details of a transaction and the relationships between transactions. It also models what P-acts are executed in each transaction. The process models of the different aspect organizations only differ in the type of P-acts they model. Actors always have to reach significational, intellectual, and social understanding using appropriate coordination acts. This means that the process models of the I-organization and D-organization are of the same nature (i.e. they consist of the same C-acts) as the process model of the B-organization. They only differ in the type P-acts executed as part of the process.

The state model, defines the objects and facts used within an organization. For the B-organization only the objects and facts used or referred to on the ontological level are modeled. The state model of the I-organization will be a more detailed model, e.g. more objects and properties are specified and facts of the I-organization are added. As the D-organization is datalogical in nature, the state model for the D-organization only defines the format for recording and retrieving data.

The last aspect model, the action model, models the action rules of an organization. For the B-organization this means that it defines how the B-actors deal with their agenda, i.e. how they order their 'to do' list. The same holds for the I-organization and D-organization.

5.2 Reverse engineering

In order to come to an organization model the first step of our MDEE approach needs to be executed: reverse engineering. Reverse engineering (i.e. how to come from an organization to an organization model) consists of three analysis and three synthesis steps [34]:

1. *The Performa-Informa-Forma Analysis.* In this step all available pieces of knowledge (i.e. documentation about the enterprise) are divided in three sets, according to the distinction axiom.

2. *The Coordination-Actors-Production Analysis.* The Performa items are divided into C-acts/result, P-acts/results, and actor roles, according to the operation axiom.

3. *The Transaction Pattern Synthesis.* The results so far, are clustered in transaction patterns, according to the transaction axiom. Next, for every transaction type, the result type is correctly and precisely formulated. The Transaction Result Table can now be produced.

4. *The Result Structure Analysis.* Based on the composition axiom, the transactions are related to each other.

5. *The Construction Synthesis.* For every transaction type, the initiating actor role(s) and the executing actor role are identified, based on the transaction axiom.

6. *The Organization Synthesis*. A definite choice has to be made as to what part of the construction will be taken as the organization to be studied and what part will become its environment. The Actor Transaction Diagram can now be produced.

With these six steps the basis for a correct and complete set of aspect models of an enterprise ontology is produced. The other aspect models of the B-organization can now be produced. The aspect models of the I- and D-organization can be produced in the same way. However, instead of only using documentation of the enterprise as input, the models of the B-organization also function as input for modeling the I-organization and the models of the I-organization function as input for the modeling the D-organization. In the remainder of this chapter we will show the result of executing the reverse engineering step on an example case.

5.3 Introducing the Protector case

As a running example for this chapter and the next chapters we use the Protector case also used in [90, 93]. Protector is an insurance company selling three types of life insurance products:

- *Term life insurance*: protects the beneficiaries of the policy from the financial damage they suffer in case the insured dies during the policy term.
- *Pension insurance*: can be seen as an insurance that protects the insured from a large income loss if he reaches his pension age or that protects his life partner and young children from large income loss after the insured (would have) reached his pension age in case of the insured's death.
- *Capital sum insurance*: an insurance for building up capital. When the end date of the policy is reached, then the benefit will be paid in a single payment. This product type is, for instance, suitable for saving money to pay off a mortgage.

Protector offers multiple products of each type and sells these products to companies (collectively) or individual persons (individually). We use the word policy for the individual policy as well as for a participation in a collective contract. An insurance policy has an insurant, one or more insured, and one or more beneficiaries. The insurant is an organization or person that is responsible for the payment of the premium of a policy. The insurant is the client of Protector. The insured is a person who is the 'insured object'. The beneficiary is a person who receives a payment if the insurant has a right to a benefit according to the product rules of a policy.

Protector reinsures policies if they cause a high risk. This means that a part of the insured amount is insured by the reinsurer in order to spread the risk. A reinsurer can be a 'regular' insurance company or an insurance company that is specialized in insuring insurance companies. Sometimes reinsurance is legally obligatory, sometimes it is a choice made by Protector itself.

We will use the individual policy part of the Protector case to explain our MDEE approach.

5.4 Example organization model

Let's apply the theory explained in this chapter to the protector case. We will first show the aspect models for the B-organization (i.e. the ontological models) following DEMO. After that we show some example models for the I- and D-organization (i.e. the infological and datalogical models). For the latter we use the same DEMO diagram types as we use to express the ontological models.

Ontological construction model

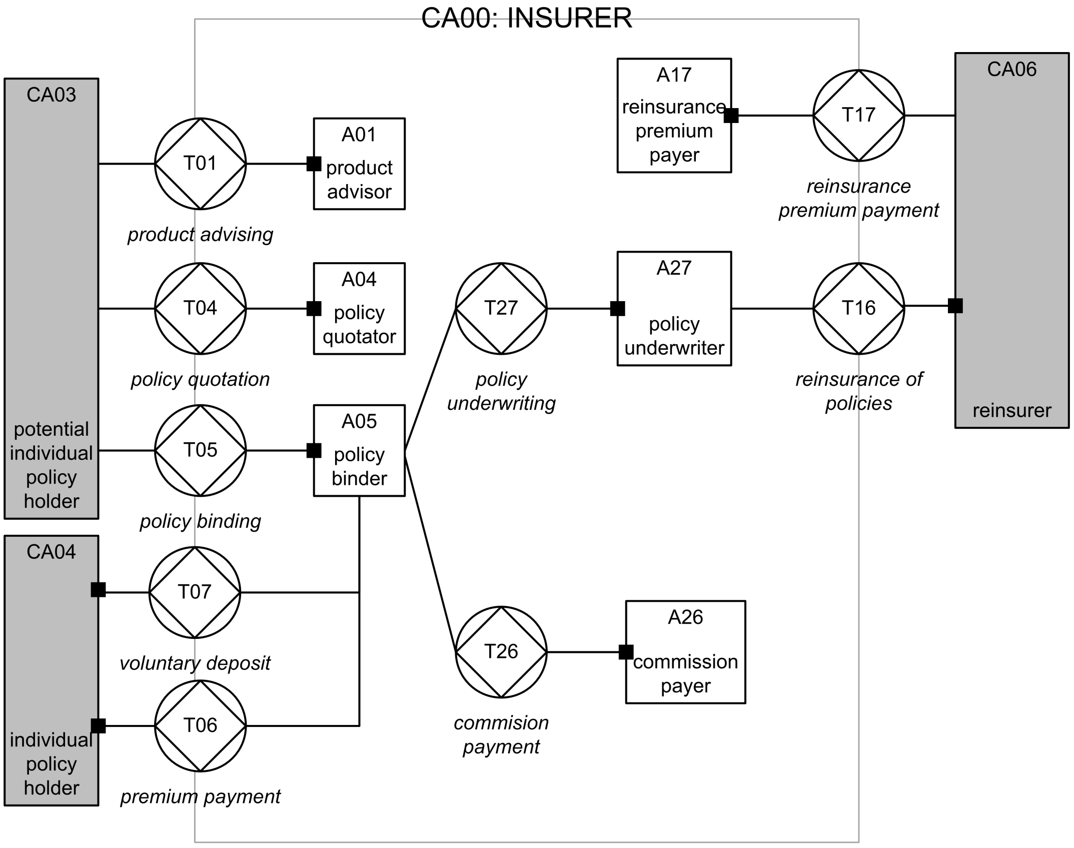


Figure 5.1: ATD for Protector [90]

The construction model (CM) of an organization specifies its composition, its environment, and its structure. DEMO specifies this model with an interaction model (IAM) consisting of an Actor Transaction Diagram (ATD) and a Transaction Result Table (TRT). Figure 5.1 shows the ATD for the individual policy part of Protector, exhibiting the actor roles and transactions. Table 5.2 shows the corresponding TRT, exhibiting the transaction types and their result types.

A potential individual policy holder can request a quotation for a product (T04) with or without getting an advice first (T01). After the promise of the policy binder (T05), the policy underwriting (T27) is requested. The policy underwriter checks if the risk is acceptable and optionally requests reinsurance of the policy (T16). If rein-

Transaction	Result type
T01 Product advising	product advice *adv* is created
T04 Policy quotation	policy *pol* is quoted
T05 Policy binding	policy *pol* is bound
T06 Premium payment	premium is paid for policy *pol* for premium period *per*
T07 Voluntary deposit	voluntary deposit is made for policy *pol*
T16 Reinsurance of policies	policy collection *pco* is reinsured for *per*
T17 Reinsurance premium payment	reinsurance premium is paid for policy collection *pco* for period *per*
T26 Commission payment	commission *com* is paid
T27 Policy underwriting	underwriting for policy *pol* has been done

Table 5.2: TRT for Protector [93]

surance is necessary and the reinsurer (CA06) states, the policy underwriter (A27) is finished and states. If no reinsurance is needed the policy underwriter directly states and in the same way A27 is finished and states. Now the policy can be bound and the policy binder (A05) states. If the potential individual policy holder (CA03) accepts, the policy binder request premium payment to the insurant (T06) and optionally requests a voluntary deposit (T07) from the policy holder and commission payment (T26) to the agent (according to the commission agreements made).

Ontological process model

object class, fact type, or result type	process steps (transactions)
PRODUCT ADVICE	T01
PRODUCT	T01
COMMISSION	T04, T26
POLICY	T04, T05, T07, T16, T27
BENEFICIARY	T04, T05
INSURED	T04, T05, T16, T27
INSURANT	T04, T05, T06
POLICY COLLECTION	T16, T17
INSURANCE PREMIUM	T05, T06, T27

Table 5.3: IUT for Protector [90]

The process model (PM) of an organization is a specification of the state space and transition space of the C-world, i.e. the set of lawful or possible or allowed sequences of states in the C-world. The process model is expressed with a Process Structure Diagram (PSD) and an Information Use Table (IUT). Figure 5.2 (DEMO 3 notation) exhibits the PSD, Table 5.3 shows the IUT. The IUT lists the object classes used in each process step.

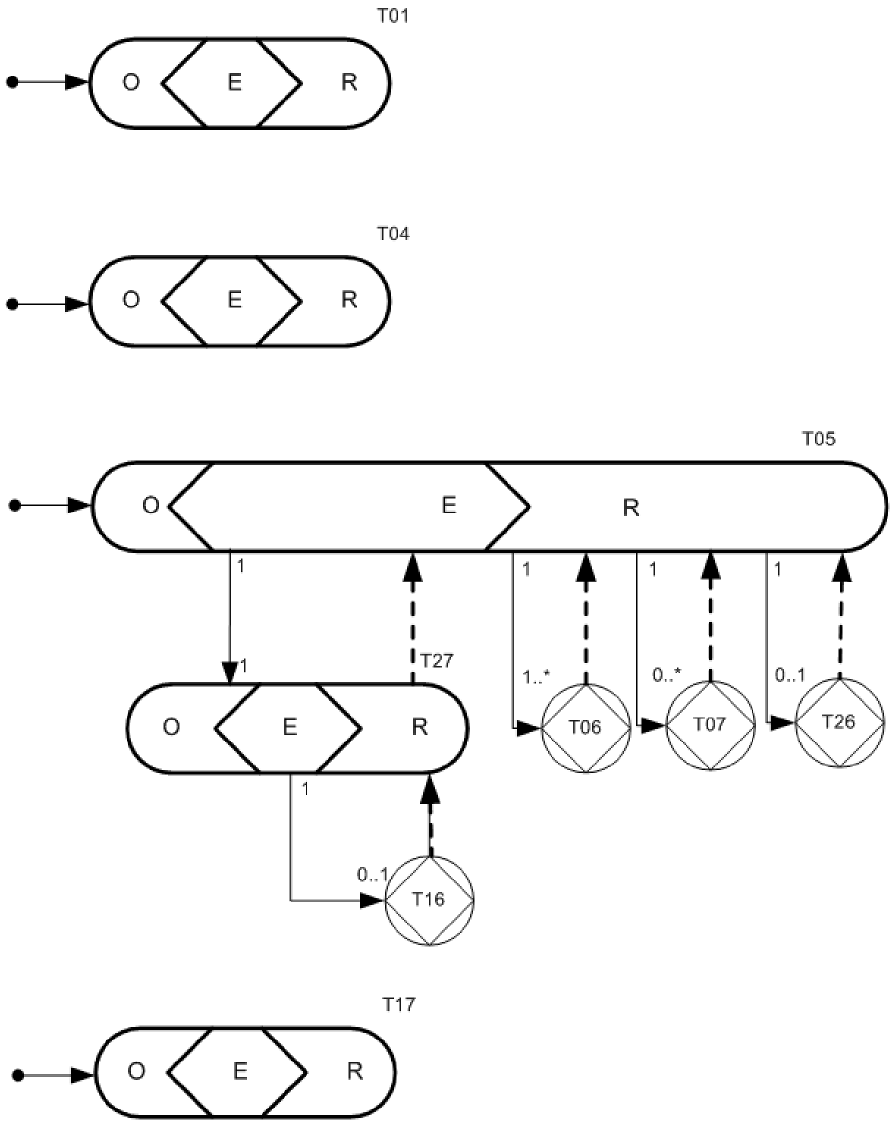


Figure 5.2: PSD for Protector [90]

In the previous section we have explained the policy binding process in detail. The PSD shows some extra information. It shows at what moment, i.e. in the order, execution, or result phase, and in which order transactions are triggered. Furthermore, the PSD exhibits the multiplicity of the relations between transactions. Note that we do not take dissent and cancellation patterns into account, these are left for future work.

Let's look at the example in Figure 5.2. T01, T04, and T17 are independent transactions. All other transactions are directly or indirectly enclosed in T05. In the execution phase of T05, T27 is triggered exactly once. T16 is optionally triggered once in the execution phase of T27. In the result phase of T05, T06 is at least triggered once, T07 is triggered zero or more times, and T26 is optionally triggered once.

Ontological state model

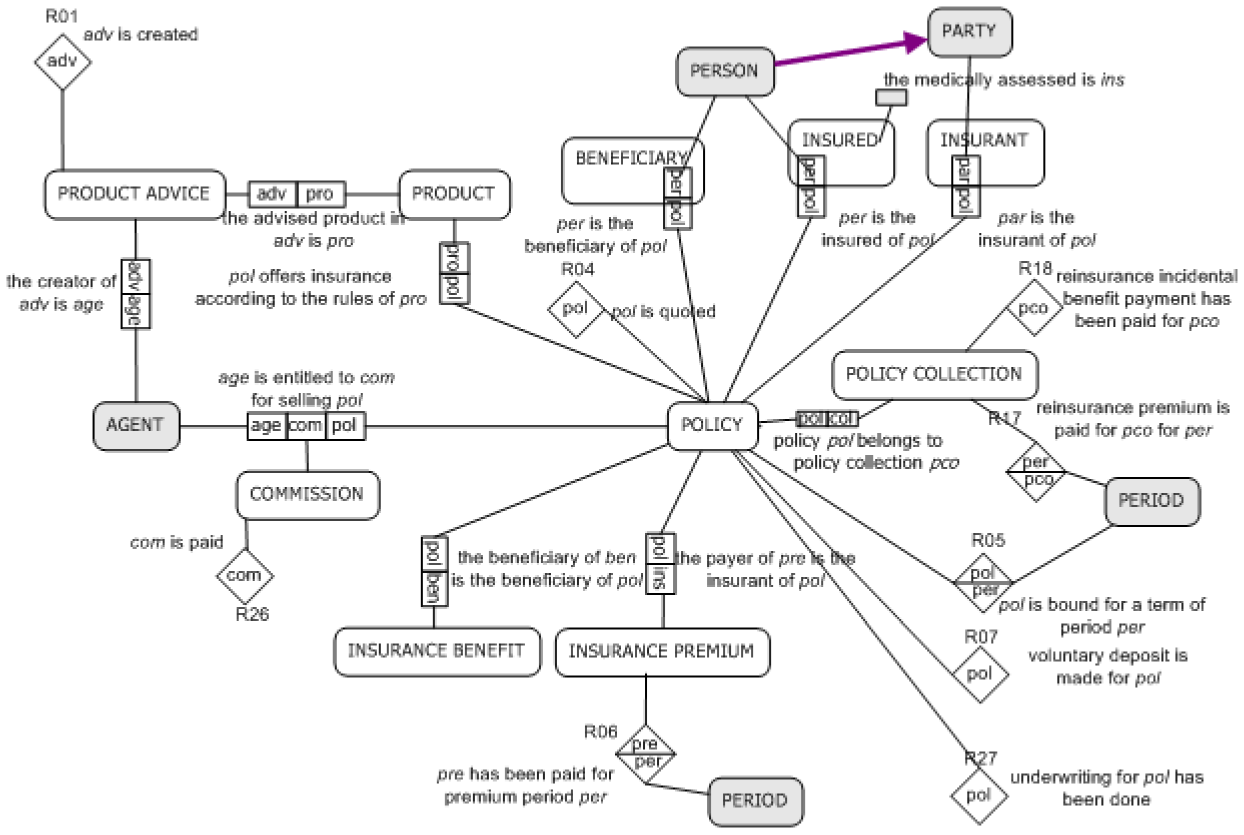

R01
adv is created
adv
PARTY
PERSON
the medically assessed is ins
BENEFICIARY
INSURED
INSURANT
per
pol
PRODUCT ADVICE
adv
pro
PRODUCT
the advised product in adv is pro
per is the beneficiary of pol
per is the insured of pol
par is the insurant of pol
R04
the creator of adv is age
adv
age
pol offers insurance according to the rules of pro
pro
pol
pol is quoted
R18
reinsurance incidental benefit payment has been paid for pco
pco
POLICY COLLECTION
age is entitled to com for selling pol
AGENT
age
com
pol
POLICY
pol
col
policy pol belongs to policy collection pco
R17
reinsurance premium is paid for pco for per
per
pco
PERIOD
COMMISSION
com is paid
com
R26
the beneficiary of ben is the beneficiary of pol
ben
the payer of pre is the insurant of pol
ins
R05
pol is bound for a term of period per
R07
voluntary deposit is made for pol
INSURANCE BENEFIT
INSURANCE PREMIUM
R06
pre
pre has been paid for premium period per
R27
underwriting for pol has been done
PERIOD

Figure 5.3: OFD for Protector

property type	object class	scale
insurance_max_sum	PRODUCT	EURO
commission_fee	COMMISSION	EURO
date_of_birth	INSURED	JULIAN DATE
period_premium	INSURANCE PREMIUM	EURO

Table 5.4: OPL for Protector

The state model (SM) is denoted in an ORM-based language called World Ontology Specification Language (WOSL) [33] and models object classes, the fact types, and the result types. The SM is expressed in an Object Fact Diagram (OFD) and an Object Property List (OPL). Figure 5.3 shows the OFD for the Protector case, which is a part of the OFD shown in [90]. We have only taken the part relevant for our case. Table 5.4 shows the OPL, exhibiting some example properties.

As exhibited in Figure 5.3 the central object is a *policy*. A *policy* has an *insurant*, one or more *insured*, and one or more *beneficiaries*. The *insurant* is an organization or person that is responsible for the payment of the *insurance premium* of a policy. The *insurant* is the client of Protector. The *insured* is a person who is the 'insured object'. The *beneficiary* is a person who receives a payment, the *insurance benefit*, if the *insurant* has a right to a benefit according to the *product* rules of a *policy*.

A *policy* offers insurance according to the rules of a *product*. A *product* can be advised by an *agent*, which is entitled a *commission* for selling a certain *policy*. A *policy collection* containing multiple policies can be reinsured.

Ontological action model

The action model (AM) is specified with a pseudo-algorithmic language. In the remainder of this section we present the action rules for the protector case, grouped on actor role.
Action rules for actor role A05:

If the policy binder (A05) has promised to bind a policy to an insured for a certain insurance premium, he requests policy underwriting (T27).

```
on promised T05(Policy, Insured, InsurancePremium) with policy(new Policy) = Po
   request T27(Policy, Insured, InsurancePremium)
no
```

If the potential individual policy holder (CA03) accepts the policy binding, the policy binder requests premium payment (T06) according to the insurance premium.

```
on accepted T05(Insurant, InsurancePremium)
   request T06(Insurant, InsurancePremium)
no
```

If the potential individual policy holder (CA03) accepts the policy binding and a commission agreement exists for the agent who advised the policy to the potential individual policy holder, the policy binder requests commission payment.

```
on accepted T05(Policy, Commission)
   if <commission agreements exists for Policy> -> request T26(Commission)
   fi
no
```

If the potential individual policy holder (CA03) accepts the policy binding and a voluntary deposit agreement exists with the policy holder, the policy binder requests the payment of that deposit.

```
on accepted T05(Policy, Beneficiary, Insured, Insurant)
   if <voluntary deposit agreement exists> -> request T07(Policy)
   fi
no
```

Action rules for actor role A27:

The policy underwriter (A27) checks whether the risk for a policy for an insured is acceptable or not. If the risk is acceptable he promises to underwrite the policy. If the risk is not acceptable the policy underwriter declines.

```
on requested T27(Policy, Insured, InsurancePremium)
    if <risk is acceptable> -> promise T27(Policy, Insured, InsurancePremium)
    not <risk is acceptable> -> decline T27(Policy, Insured, InsurancePremium)
    fi
no
```

If the policy underwriter (A27) has promised, he checks if reinsurance is necessary. Protector reinsures policies if they cause a high risk. This means that a part of the insured amount is insured by the reinsurer in order to spread the risk. If reinsurance is necessary the policy underwriter requests the reinsurance of policies (T16), if not he continues executing T27.

```
on promised T27(Policy, Insured, PolicyCollection, InsurancePremium)
    if <reinsurance is necessary> -> request T16(Policy, Insured, PolicyCollection)
    not <reinsurance is necessary> -> execute T27(Policy, Insured, InsurancePremium)
    fi
no
```

Infological and datalogical construction model

Transaction	Result type
T01 product advising	product advice *adv* is created
T30 formulate personal details	personal details are defined for person *per*
T31 compose product information	product composition *prc* is created
T60 record personal details	person *per* is recorded
T61 retrieve product data	products *pro* are retrieved

Table 5.5: TRT for ’product advising’

We express the infological and datalogical construction model in a single diagram. This gives a good insight in the shaping relations between actor roles, which are depicted with dashed arrows. It should, however, be noted that for bigger diagrams with lots of connected transactions within the same aspect organization, it can be more suitable to split the construction model for the I- and D-organization.

Let’s look at two example diagrams for the protector case. Figure 5.4 shows the ATD for the I- and D-transactions needed to support the B-transaction ’product advis-

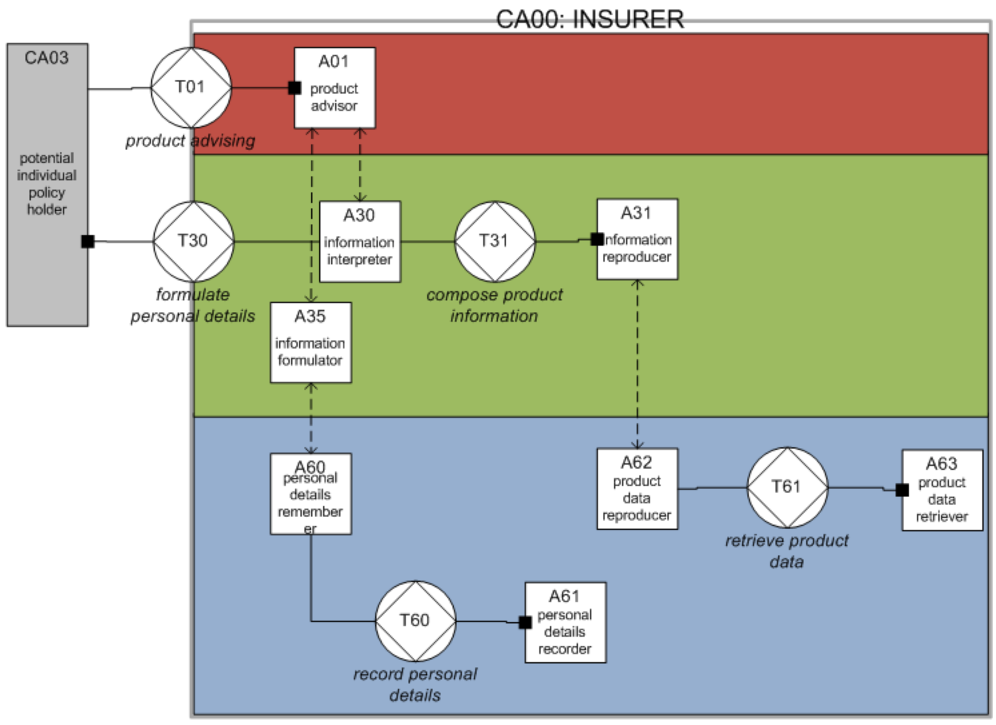

CA00: INSURER
CA03
potential individual policy holder
T01
product advising
A01 product advisor
T30
formulate personal details
A30 information interpreter
T31
compose product information
A31 information reproducer
A35 information formulator
A60 personal details rememberer
A62 product data reproducer
T61
retrieve product data
A63 product data retriever
T60
record personal details
A61 personal details recorder

Figure 5.4: CM on B, I, and D level of ’product advising’

Transaction	Result type
T05 policy binding	policy *pol* is bound
T32 combine policy request with existing information	policy *pol* is related to existing information (e.g. personal details, policy quotation, etc.)
T62 retrieve personal details	personal details *prd* is retrieved
T63 retrieve product advice	product advice *adv* is retrieved
T64 retrieve policy quotation	policy quotation *pqu* is retrieved
T65 retrieve commission agreement	commission agreement *cag* is retrieved
T66 record policy binding	policy *pol* is recorded

Table 5.6: TRT for ’policy binding’

ing’ (T01). T30 and T31 are I-transactions, T60 and T61 are D-transactions. Table 5.5 shows the TRT for ’product advising’.

When a potential individual policy holder requests a product advice, the executor of the product advising transaction, A01, will need information to create such an advice. A01 therefore shapes into the I-actor A30 and requests the potential policy holder to formulate his or her personal details, needed in order to make a suitable advice. Once this information is received A30 asks for suitable products based on these personal details. A31, the information reproducer, therefore shapes into a D-actor, A62, in order to receive the data for multiple products. When all information is available A01 decides which products he advices to the potential individual policy holder.

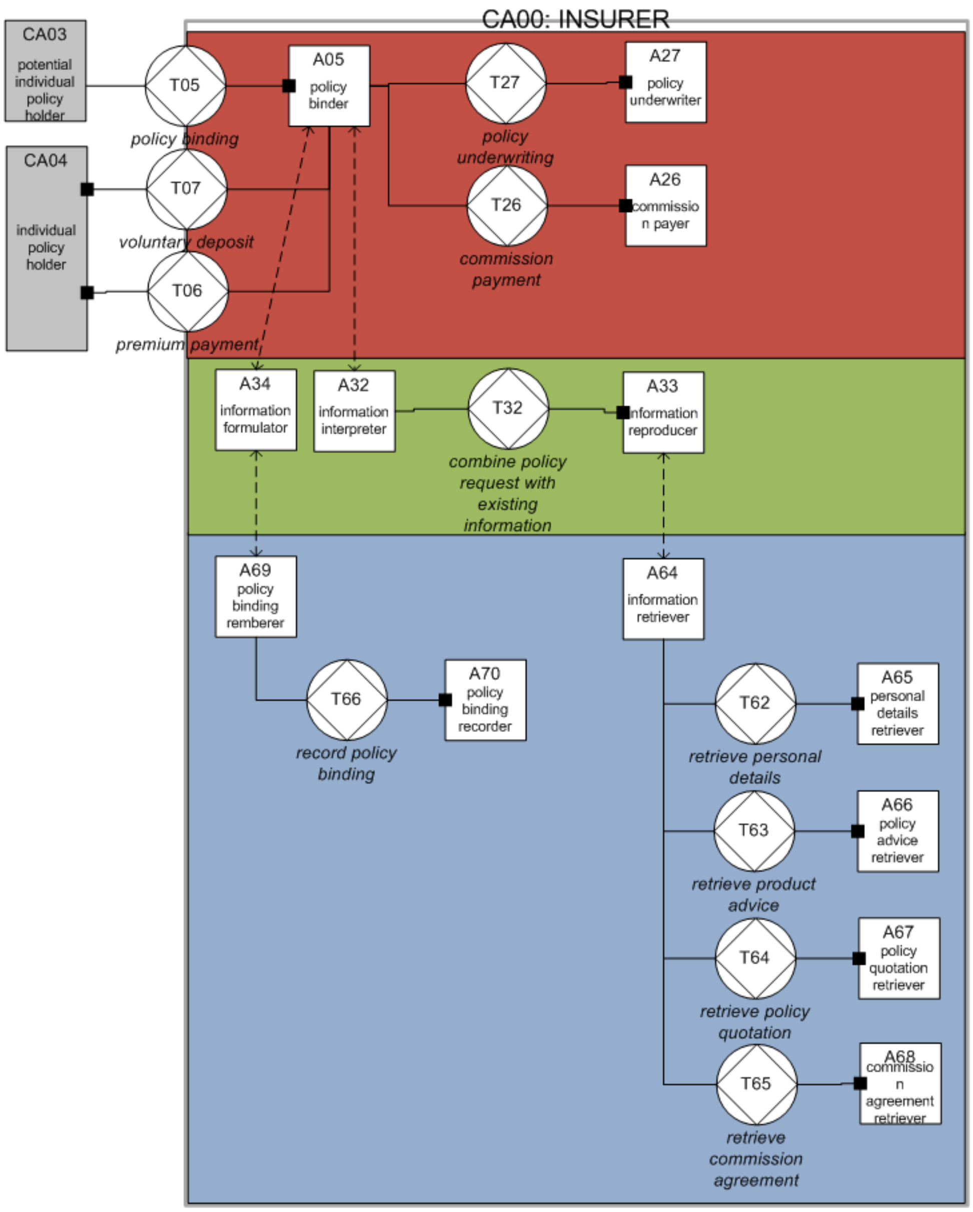

CA00: INSURER
CA03 potential individual policy holder
CA04 individual policy holder
T05 policy binding
T07 voluntary deposit
T06 premium payment
A05 policy binder
T27 policy underwriting
A27 policy underwriter
T26 commission payment
A26 commission payer
A34 information formulator
A32 information interpreter
T32 combine policy request with existing information
A33 information reproducer
A69 policy binding remberer
T66 record policy binding
A70 policy binding recorder
A64 information retriever
T62 retrieve personal details
A65 personal details retriever
T63 retrieve product advice
A66 policy advice retriever
T64 retrieve policy quotation
A67 policy quotation retriever
T65 retrieve commission agreement
A68 commission agreement retriever

Figure 5.5: CM on B, I, and D level of 'policy binding'

After giving the advice the related advice information, including the personal details of the potential individual policy holder, are stored for future use. A01 does so by shaping into D-actor A60 via I-actor A35.

Figure 5.5 shows the ATD for the I- and D-transactions needed to support B-transaction 'policy binding' (T05). T32 is an I-transaction, T62 till T66 are D-transactions. Note that we only exhibit the I- and D-transactions needed to support T05, we do not show the I- and D-transaction needed to support the other B-transactions. The TRT is exhibited in Table 5.6.

If a potential individual policy holder requests a policy binding, the policy binder needs to lookup the existing information for this person. He does so by shaping into an

I-actor, A32, and requests all policy binding information from A33, which shapes into D-actor A64. A64 retrieves all available information, i.e. personal details, optionally an earlier given product advice, the policy quotation, and optionally a commission agreement. After the policy is bound, A05 shapes into D-actor A69 (via A34) to store the policy binding information. How A05 deals with the other B-transactions was explained before when talking about the ontological construction model.

Infological and datalogical process model

As the goal of the process model is to show the possible states in the C-world and how they relate, we express the infological and datalogical process model with a diagram for each B-transaction. We show example diagrams for the two B-transactions for which we also created an ATD in the previous section: 'product advising' and 'policy binding'. Remember that transactions initiated across aspect organizations are initiated by an actor shaping into another actor.

Figure 5.6 exhibits the process diagram for 'product advising'. As visualized, once product advising (T01) is promised, the potential individual policy holder is asked to formulate his or her personal details (T30). After that, applicable products are retrieved (T61) and composed (T31). Once an advice is created it is stored (T60) and stated to the requester.

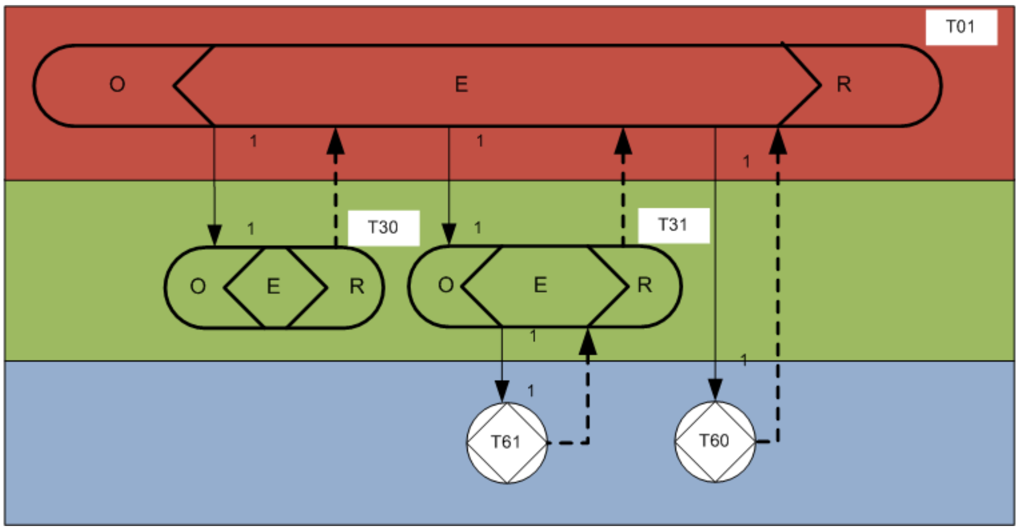


Figure 5.6: BPD on I, and D level for 'product advising'

Figure 5.7 exhibits the process diagram for 'policy binding'. As visualized, once policy binding (T05) is requested all previously recorded information is retrieved (T62 - T65) and combined with the policy binding request (T32). After the policy is bound, the policy binding is recorded (T66).

The combined IUT for both process diagrams is shown in table 5.7.

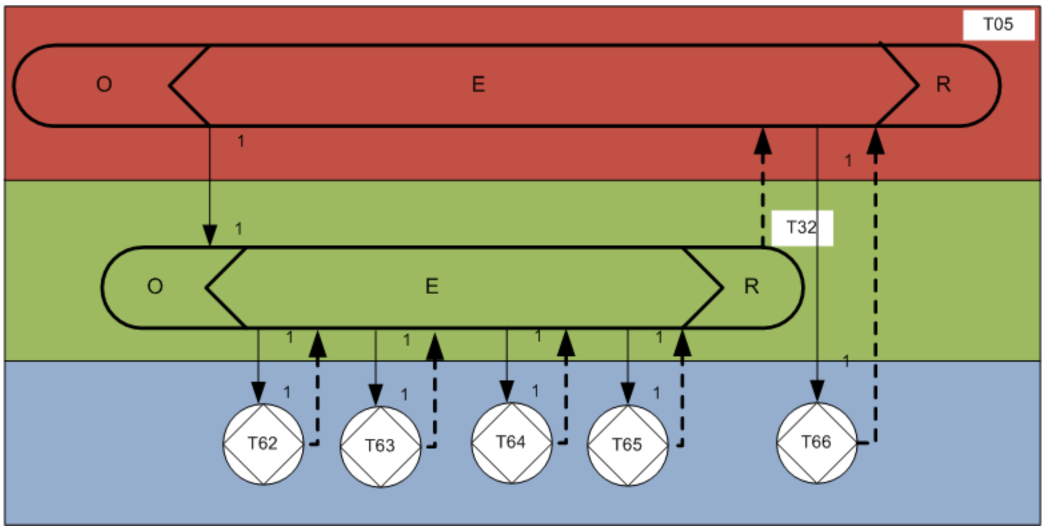


Figure 5.7: BPD on I, and D level for 'policy binding'

object class, fact type, or result type	process steps (transactions)
PERSON	T30, T60, T62
PRODUCT COMPOSITION	T31
PRODUCT	T61, T31
PRODUCT ADVICE	T63, T32
POLICY QUOTATION	T64, T32
COMMISSION AGREEMENT	T65, T32
POLICY	T66, T32

Table 5.7: IUT for 'product advising' and 'policy binding'

Infological and datalogical state model

The infological state model defines additional object classes and properties. Figure 5.8 shows an example OFD for the infological state model. It contains a part of the ontological state model, extended with the object class *policy quotation* and the result type R30. We only added the classes and result types needed for the product advising and policy binding processes. The associated OPL is shown in Table 5.8.

property type	object class	scale
price	POLICY QUOTATION	EURO
expire_date	POLICY QUOTATION	JULIAN DATE
start_date	POLICY	JULIAN DATE
percentage_fee	COMMISSION AGREEMENT	NUMBER
date_of_birth	PERSON	JULIAN DATE

Table 5.8: Infological OPL for Protector

The additional properties of Person are defined in T30 when the policy binder asks

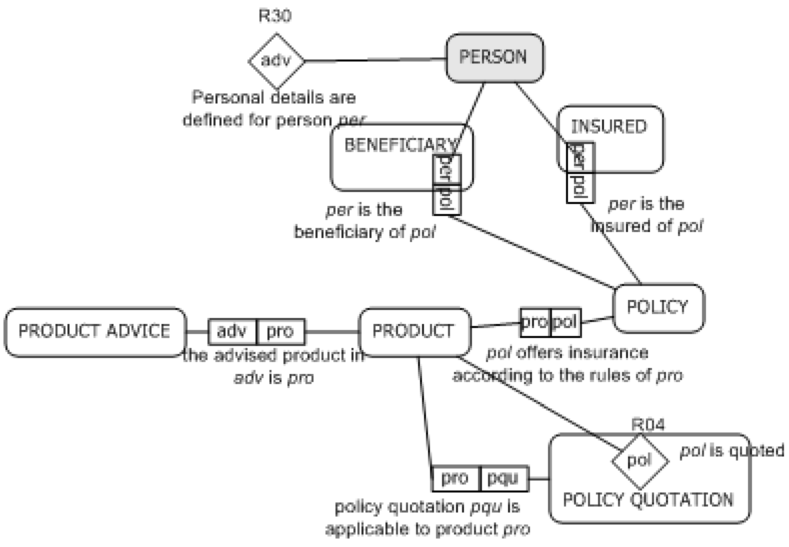

R30
adv
PERSON
Personal details are defined for person *per*
BENEFICIARY
per
pol
INSURED
per
pol
per is the beneficiary of *pol*
per is the insured of *pol*
POLICY
PRODUCT ADVICE
adv
pro
PRODUCT
pro
pol
the advised product in *adv* is *pro*
pol offers insurance according to the rules of *pro*
R04
pol
pol is quoted
pro
pqu
POLICY QUOTATION
policy quotation *pqu* is applicable to product *pro*

Figure 5.8: Infological OFD for Protector

the potential individual policy holder for additional information. A *policy quotation* is applicable to its associated *product* and contains the details for a quoted policy.

The datalogical state model only defines the format for recording and retrieving data, it does not add content. Example format properties are the language, the storage format, and format validations (like the maximum length of a property).

Let's look at two example object classes: personal details and policy quotation. Personal details are retrieved from a person on a paper form. The form is specified in dutch and archived in a file. A policy quotation is stored in a database in a table named policy_quotation with a column for each property. The table has two additional columns, one with a unique identifier (a number) and one with the unique identifier of a row in the product table (to link the quotation to a product).

Infological and datalogical action model

The infological and datalogical action model contain the same kind of rules as the ontological action model. The only difference is that they are applied to I- and D-transactions instead of B-transactions. So, I- and D-action rules describe under what conditions an actor role requests, promises, states, or accepts transactions. However, due to the fact that B- I- and D-transactions are different, the conditions in the action rules in the different aspect organizations also differ. I-action rules are focused on the content and D-action rules on the form of input data.

Action rules for I-actor role A30:

The information interpreter (A30) accepts the personal details formulated by the potential individual policy holder (CA03) if all properties are specified.

```
on stated T30(Personal Details)
    if <all properties of Personal Details are specified> -> accept T30(Personal Details)
    not <all properties of Personal Details are specified> -> decline T30(Personal Details)
    fi
no
```

If the information interpreter (A30) accepts T30 (formulate personal details), he requests to compose product information (T31) from the information reproducer (A31).

```
on accepted T30(Personal Details)
    request T31(Personal Details)
no
```

Action rules for D-actor role A70:

The policy binding recorder (A70) can only record the given policy if the property start_date is in the right date format.

```
on requested T66(Policy)
    if <start_date(Policy).format = 'MM-DD-YYYY'> -> promise T66(Policy)
    not <start_date(Policy).format = 'MM-DD-YYYY'> -> decline T66(Policy)
    fi
no
```

Chapter 6

An Enterprise Ontology based Service Specification Model

Our MDEE approach aims to design and engineer an IT system supporting an organization. The organization (the using system) is going to use the IT system (the object system). In the previous chapter we have defined the organization model, describing the highest level construction model of the organization. The next step is to design the function of the IT system based on the information in the organization model. As explained in Chapter 4 the functional (or black-box) model of the object system is a service specification model. In this chapter we research the elements of this functional model (6.1). We also define the function design step of our MDEE approach, i.e. how to identify what services we need in order to support a given organization model (6.2). This chapter is concluded with an example service specification model for the Protector case (6.3).

6.1 Service multi-model

Before thinking about how the functional model of an ICT system can be specified with a service specification model, we need to define what a service is. Terlouw and Albani [92] define the notion of service in a precise way, based on the Ψ-theory (see section 3.1):

> A service is a universal pattern of coordination and production acts, performed by the executor of a transaction for the benefit of its initiator, in the order as stated in the standard pattern of a transaction. When implemented it has the ability to get to know the coordination facts produced by the initiator and to make available to the initiator the coordination facts produced by itself.

Note that a service is not equal to a transaction. A transaction includes all acts of the initiator and the executor, the service concept emphasizes more on the executor part. Everything of the standard transaction pattern is part of the service except for the coordination acts (request, quit, reject, and accept) of the initiator.

This definition of service is very generic and, according to Terlouw and Albani [92], it holds for both human actors and IT systems. Furthermore it can be used to describe all three kinds of production acts, i.e. datalogical, infological, and ontological acts.

Looking at the service specification model, we can say that it consists of service specifications. The service specification model should describe the function of the ICT system under development (the object system) in terms of the organization (the using system). Hence, the service specification model should describe the services that support the organization. Our MDEE approach should derive the needed services from the organization model. This process should be automated as much of possible to make our MDEE approach really model-*driven*.

The first thing to consider while doing so is the size of the service specification model. As explained in chapter 5 the organization model can become quite big and is therefore treated as a multi-model consisting of multiple aspect models. We should specify the service specification model as a multi-model too for the same reasons. We can use the same dimensions as for the organization model: the distinction dimension, based on the distinction axiom, and the aspect dimension.

The distinction dimension consists of ontological, infological, and datalogical services. This compares to the service layers Terlouw distinguishes [91]. The aspect dimension is more complicated to define. Although in practice all kinds of service taxonomies are used (see for example [24, 35, 52]), none of them is based on a scientific theory. We therefore stick to the distinction between human and IT services made by Terlouw and Albani [92].

	IT	Human
B	Ontological IT services	Ontological human services
I	Infological IT services	Infological human services
D	Datalogical IT services	Datalogical human services

Table 6.1: Aspect models of the service specification model

The resulting multi-model consists of six aspect models conforming to the six service types defined by Terlouw and Albani [92]. As depicted in Table 6.1 we have the following service layers on the distinction dimension:

- *B-services*, they support B-transactions and are provided to B-actors
- *I-services*, they support I-transactions and are provided to I-actors or B-actors (via shaping)
- *D-services*, they support D-transactions and are provided to D-actors or I-actors (via shaping)

On the aspect dimension we have the following distinction:

- *IT-services*, they assist human actors in their activities. The actor role providing an IT service to the initiator of a transaction is an IT system.
- *Human-services*, they are provided by human beings.

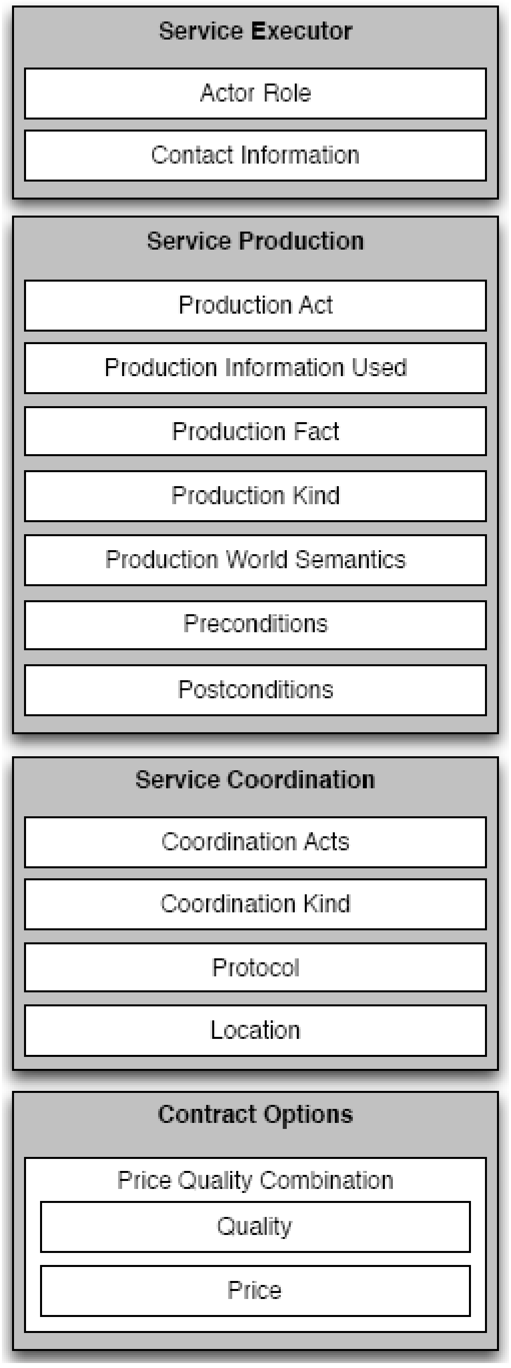


Figure 6.1: The Generic Service Specification Framework [92]

Terlouw and Albani [92] have defined a service specification framework based on the Ψ-theory (which is also the basis for enterprise ontology). Figure 6.1 exhibits this framework, which is split into four parts. For calling a service basically three things need to be known to the service consumer: who provides the service (service executor), which production act is to be performed by the provider (service production), and how to interact with the service executor (service coordination). Additionally the service consumer needs to now what he gets for which price (contract options) [92].

We use this service specification framework as a template to describe the services in our service specification model. This means that the service specification multi-model consists of six service types, all described with the service specification template presented in Figure 6.1. In the remainder of this chapter we will explain the template in more detail and describe how to derive a service specification from the organization model.

6.2 Function design

As explained in section 4.6 the step between organization model and service specification model is function design. During function design the service specification model is specified with the organization model as starting point. We can define this process with a set of rules which, if specified in a formal language, can be mostly automated.

In section 4.5 we have explained that for implementing complex ICT systems our MDEE approach need to comply to the principles of SOA. Hence, following [93] the function design step in which we identify services can be called service-oriented function design (SoFD). Terlouw [93] lists some existing SoFD methods (or methods covering SoFD): BCI-3D [4], the Business Element Approach [65], the Goal-Driven Approach [62], SOAF [38], the methodology of Papazoglou and Van den Heuvel [82], and SOMA [10].

BCI3D is focused on component identification and the services provided by components to other components. Hence, it is more about the construction of an ICT system and less about its function. We will take BCI3D into account when talking about construction design in Chapter 7. The Business Element Approach is based on business processes and information objects, but it lacks a clear, formal definition of these two concepts [93]. The Goal-Driven Approach aims to identify services directly from the goals of an enterprise using a Goal Service Graph. This approach is very subjective and informal [93]. SOAF, the methodology of Papazoglou and Van den Heuvel, and SOMA do not specify a very detailed service identification approach or refer to other approaches.

While we need a formal service identification method using our enterprise ontology based organization model as input, none of the existing methods can be used. Terlouw and Albani [92] describe in their example how they use parts of the organization model to specify the different elements of the service specification framework. Hence, we can base our service identification rules on their description.

First the transactions need to be selected which are going to be supported by the object system, i.e. the ICT system to build. For each transaction a service needs to be specified. The relation between a transaction and a service is not defined in detail yet. For now we assume a one-to-one relationship following the definition of a service given in the previous section. However, note that it can be possible that multiple services support one transaction. Hence, more research is needed into this relation. Based on our assumption, we can say that each service describes what coordination and production acts the executor of the transaction provides to the initiator. After all needed services are identified, they need to be specified using the Generic Service

Specification Framework.

Service specification item	Derivation rule
Actor Role	The executor of the transaction. If the executor is an IT system humans have the final responsibility. This information can be derived from the ATD.
Contact Information	Not available in the organization model.
Production Act	The name of the transaction in the ATD.
Production Information Used	Add each information element from the IUT associated with the transaction.
Production Fact	The result type of the transaction, specified in the TRT.
Production Kind	Corresponds with the type of transaction: ontological (B), infological (I), or datalogical (D).
Production World Semantics	Describe the elements of the OFD related to the production information and production fact and their relations.
Preconditions	The action rules associated with the transaction defining on what conditions the promise act is executed.
Postconditions	The action rules associated with the transaction defining on what conditions the accepted act is executed.
Coordination Acts	The coordination acts of the executor of the transaction defined in the BPD.
Coordination Kind	Not available in the organization model. IT or Human.
Protocol	Not available in the organization model.
Location	Not available in the organization model.
Quality	Not available in the organization model.
Price	Not available in the organization model.

Table 6.2: Service specification rules

As we have seen in Figure 6.1 the service executor area of concern consist of an actor role and contact information. The actor role specifies the role of the actor that takes final responsibility of the service. The actor role can be derived from the ATD of the organization model. Contact information is not available in the organization model and needs to be added. This information can be used to contact the executor.

The service production area of concern focuses on the actual value that the service executor offers to the service initiator. The production act can be derived from the ATD. The information needed in order to execute the actual production act can be derived from the IUT. The execution of a production act results in a production fact, which is specified in the TRT. The production kind defines the kind of service based on the type of production act, i.e. ontological, infological, and datalogical. The production kind corresponds with the type of the transaction (B, I, D) supported by the service. The production world semantics provide a common knowledge and understanding about the semantics of the service. These semantics can be derived from the OFD. The pre- and postconditions state production facts that should always hold prior to and after the execution of the service respectively. These conditions can be derived from the Action Model.

The service coordination area of concern aims at giving the consumer of the service all information required for successful communication with the provider. Therefore the coordination acts need to be described, which can be done based on the BPD. Information for the coordination kind, protocol, and location is not available in the organization model and need to be added manually. The coordination kind defines whether this service is a human or IT service (as explained in section 6.1). The protocol defines the rules governing the syntax, semantics and synchronization of communication, while the location defines where the service consumer can access the service.

The last area of concern, the contract options, provides information on the price quality combination. This information also needs to be added manually because it is not available in the organization model.

Table 6.2 exhibits the previously described service specification rules which describe how to specify a service with the organization model as input. The input for service specification is the transaction supported by the service. Based on the transaction the needed information can be derived from the organization model, except for contact information, coordination kind, protocol, location, quality, and price. Information for these items is not available in the organization model and needs to be added manually.

So, the SoFD step of our MDEE approach consists of:

- *Service identification*: define a service for each transaction in the organization model. Categorize them using the distinction and aspect dimension.

- *Service specification*: specify each identified service using the service specification framework.

6.3 Example service model

Let's look at our running example, starring the Protector case, to see what the service specification model looks like in practice and how to execute the service-oriented function design step. We have identified the services needed to support the product advising part of Protector (see Figure 5.4 for the associated organization construction model) exhibited in Table 6.3.

	IT	Human
B		ProductAdvising
I	ComposeProductInformation	FormulatePersonalDetails
D	RecordPersonalDetails, RetrieveProductData	

Table 6.3: Identified services for the 'product advising' part of Protector

As shown in Table 6.3 we have identified a service for each transaction in the organization model. We also categorized them using the distinction and aspect dimension. Each B-transaction is supported by a B-service, an I-transaction by an I-service, and a

D-transaction by a D-service. As the ProductAdvising and FormulatePersonalDetails services can not be fully automated, they are classified as human services. The ProductAdvising service supports the human actor 'product advisor' in his task to create a product advice for the potential individual policy holder. The FormulatePersonalDetails service lets the human actor 'potential individual policy holder' formulate his personal details. How the interaction of human actors with services is implemented will be described in detail in Chapter 8.

ProductAdvicing	
Actor Role	A01: product advisor
Contact Information	University Street 1A 8291 BN Insurancetown 555-492022 productadvising@protectorinsurances.com
Production Act	T01: product advising
Production Information Used	PRODUCT ADVICE, PRODUCT
Production Fact	R01: product advice *adv* is created
Production Kind	ontological (B)
Production World Semantics	the creator of PRODUCT ADVICE *adv* is AGENT *age* the advised product in PRODUCT ADVICE *adv* is PRODUCT *pro* POLICY *pol* offers insurance according to the rules of PRODUCT *pro*
Preconditions	-
Postconditions	-
Coordination Acts	the standard transaction pattern
Coordination Kind	Human
Protocol	Call the product advisor, he will ask some questions and create an advice. The advice is also send by post.
Location	555-492022
Quality	The maximum response time in 90% of the calls is 5 minutes. In most cases you need to make an appointment to get the advice, which can in 90% of the cases be planned within 2 days.
Price	Free

Table 6.4: Service specification of the ProductAdvising service

Table 6.4 shows the complete service specification for the ProductAdvising service. The specification is made using the service specification rules presented in section 6.2. No pre- and postconditions are specified because no action rules exist in the organization model associated with transaction T01 (the transaction supported by the ProductAdvising service). However, an example post condition could be that the resulting product advice is aimed at the personal details of the potential individual policy holder.

Table 6.5 shows the service specification for the RecordPersonalDetails service.

RecordPersonalDetails	
Actor Role	A61: personal details recorder, Responsible human: A01: product advisor
Contact Information	University Street 1A 8291 BN Insurancetown 555-492022 productadvising@protectorinsurances.com
Production Act	T60: record personal details
Production Information Used	PERSON
Production Fact	person *per* is recorded
Production Kind	datalogical (D)
Production World Semantics	Personal details are defined for PERSON *per* PERSON *per* is the BENEFICIARY of *pol* PERSON *per* is the INSURED of *pol*

Preconditions	**on** requested T60(Personal Details)
	    if <complete(Personal Details)> -> promise T60(Personal Details)
	    not <complete(Personal Details)> -> decline T60(Personal Details)
	    fi
	no

Postconditions	-
Coordination Acts	The standard transaction pattern
Coordination Kind	IT
Protocol	SOAP
Location	http://mdm.protector.com/RecordPersonalDetails
Quality	The response time of this service is always less than 200ms
Price	-

Table 6.5: Service specification of the RecordPersonalDetails service

As exhibited a precondition exists defining that the service only can be executed if the given personal details are complete (i.e. all attributes are filled). The implemented coordination acts by this service are the ones defined by the standard transaction pattern, as explained in section 3.3.

We also identified the services needed to support the policy binding part of Protector (see Figure 5.5 for the associated organization construction model) exhibited in Table 6.6. Following the described service identification method in the previous section we have a service for each transaction in the organization model. In the policy binding part the only human service is the PolicyBinding service. This service can not be fully implemented using an IT system, because the acts of the supported transaction are ontological. As explained in section 3.4 ontological acts can not be automated and need human involvement. Table 6.7 shows the complete service specification for the PolicyBinding service.

	ICT	Human
B		PolicyBinding
I	CombinePolicyReqInfo	
D	RecordPolicyBinding, RetrievePersonalDetails, RetrievePolicyAdvice, RetrievePolicyQuotation, Re-trieveCommissionAgreement	

Table 6.6: Identified services for the 'policy binding' part of Protector

PolicyBinding	
Actor Role	A05: policy binder
Contact Information	University Street 1B 8291 BN Insurancetown 555-492023 policybinding@protectorinsurances.com
Production Act	T05: policy binding
Production Information Used	PRODUCT ADVICE, PRODUCT
Production Fact	R05: policy *pol* is bound
Production Kind	ontological (B)
Production World Semantics	POLICY *pol* is bound for a term of PERIOD *per* PERSON *per* is the beneficiary of POLICY *pol* PERSON *per* is the insured of POLICY *pol* PARTY *par* is the insurant of POLICY *pol*
Preconditions	-
Postconditions	-
Coordination Acts	the standard transaction pattern
Coordination Kind	Human
Protocol	Make an appointment with the policy binder to meet face-to-face.
Location	University Street 1B 8291 BN Insurancetown 555-492023 policybinding@protectorinsurances.com
Quality	In 90% of the cases an appointment can be planned withing 1 week.
Price	Free

Table 6.7: Service specification of the PolicyBinding service

Chapter 7

A Business Component Construction Model

The service specification model specifies all the services needed to support the organization model. The service specification model is a black-box model, modeling the function of the object system. The next model in our MDEE approach is the business component construction model, it describes the construction of the object system. In this chapter we will first research the elements of such a model (7.1). After that we explain how to define the business component construction model based on the previous models (7.2). We finish this chapter with an example business component construction model starring the Protector case (7.3).

7.1 Construction multi-model

The services in the service specification model describe the function of the object system. The construction model needs to describe the construction of the object system, i.e. the different components of the object system and their interaction relationships [34]. As we have seen in section 4.5, a component implements a set of services. A business component implements a set of services within a certain business domain. The idea behind business components is to design them in such a way that they can be reused within the same domain and possibly across domains.

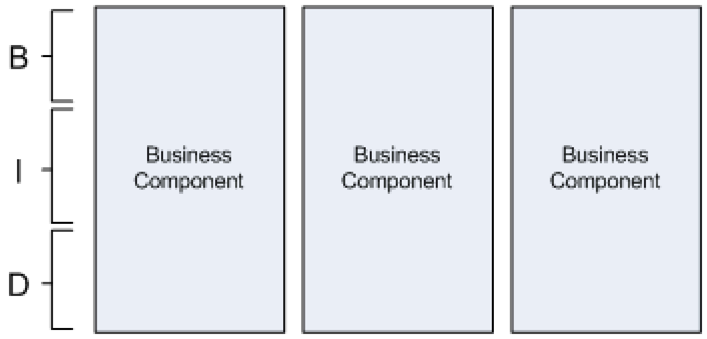


Figure 7.1: Component scenario 1

The way the object system is composed from business components can differ. Basically two scenario's can be identified, depicted in Figure 7.1 and Figure 7.2. In scenario 1 services are grouped in components based on their relations, no matter what type of services they are. Hence, components in such a scenario exist of B-, I-, and D-services.

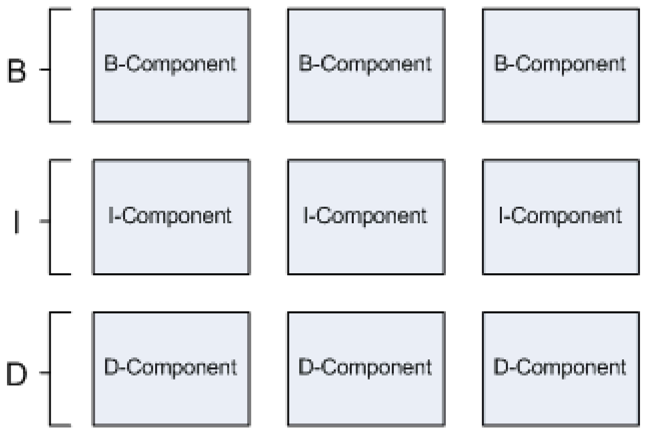


Figure 7.2: Component scenario 2

In scenario 2 we distinct between B-, I-, and D-components, publishing B-, I-, and D-services respectively. In this scenario the services are first grouped based on their production kind. After that they are grouped in components based on their relationships.

The choice between scenario 1 an 2 often depends on human choice and the existing IT landscape. While there is no 'best approach', we do not restrict our MDEE approach to one scenario. Within the MDEE approach one can choose the best fitting component design. We will explain in the next section how this choice can be made during the business component identification step.

Depending on the chosen scenario a business component construction model should be specified. This model consists of component specifications which specify what components exist, what services they provide, what services they consume (i.e. their references to services of other components), and what information objects they contain.

All services defined in the service specification model need to be grouped in components. In our service model we have made a distinction between human services and IT services. Although human services are provided by human beings, in most cases part of a human service is supported by an IT system. For example, the resulting fact (of a service execution) is registered in an IT system or the request for the human service is issued via a task list provided by an IT system. Hence, we also need to group human services in components. What part of these human services is implemented by IT is an implementation decision, which is left for the implementation model described in Chapter 8. So, we do take human services into account when executing the business component identification step.

Grouping information objects into components has two purposes. First, the grouping of information objects in components is important for the component identification step. As information objects have relations among each other they will influence the component identification step (see section 7.2). They will therefore influence the way D-services are grouped in components. D-services do not have dependencies among each other, they only have dependencies on information objects. Second, the grouping of information objects in components can say something about where instances of these information objects are stored. In Chapter 8 we will explain this subject in more detail.

The diagram we use to express the business component construction model is a UML component model [75]. We use the UML component notation because it is the most used, standardized component notation and it describes most of the elements we need to express (e.g. components, provided and consumed services, component relations). Figure 7.3 shows the UML notation we use to express the business component construction model. As exhibited provided services are expressed using a line ending in a circle, consumed services are expressed with a dashed line. Information objects have to be added as documentation because they are not part of the official notation. In section 7.3 we will show a full example diagram.

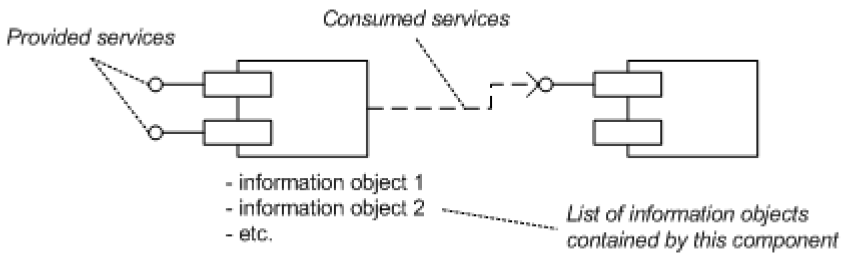


Figure 7.3: Business component specification

Until now we have focused on the part of the object system supporting the production of the organization. The service specification model describes the function of the object system, i.e. what services are published by the object system to support production acts in the organization model. The service specification model also describes a part of the coordination involved in calling a service, it defines for each service what coordination acts are supported and what protocol and location are involved. However, it does not say anything about interactions among services and in what order services need to be called. We see the interaction among services as part of the construction of the object system. Hence, the business component construction model needs a coordination part modeling these interactions.

As each service supports a transaction, we first analyze the relations between transactions before we come to a service coordination model. According to Dietz [34] (as explained in Chapter 3), every transaction is either:

- enclosed in some other transaction,
- is a transaction initiated by an actor from the environment of the enterprise,
- or is a self-activation transaction.

While we also take the I- and D-organization into account we can add another type of transaction:

- a transaction supporting (i.e. needed for the implementation of) a transaction in another aspect organization.

For example, for executing a B-transaction the executor of this transaction, a B-actor, needs to shape into an I-actor to initiate an I-transaction. So, in principle transactions can be enclosed in some other transaction in two different ways: within the same aspect organization (defined by action rules) or across aspect organizations (defined by actor shaping).

We can translate these two types of enclosed transactions in terms of services. The first type of enclosing transactions, transactions enclosed in some other transaction in the same aspect organization, can be translated into a sequence of service calls. We call such a sequence of service calls an orchestration [91]. In the organization model the relation among transactions is shown in the construction model and detailed in the process model and action model. The action model contains all the information of the construction and process model and adds more details. The action rules define the relations among transactions in detail with additional conditions. Hence, these action rules can be translated into service orchestrations defining in what order services are called and on what conditions.

The second type of enclosing transactions, transactions initiated across aspect organizations, can be translated in a service call from a service consumer to a service provider on another service layer. A service whose implementation calls other services is defined as composite service. Hardjosumarto [47] defines a composite service based on enterprise ontology as follows:

> A *composite service* is a service from which the creation of the production fact depends on the creation of production facts from underlying services. After the composite service promises the service consumer to create the production fact, it requests the underlying services. If the composite service has accepted the production fact from the underlying services, it creates its own production fact and presents this to the service consumer.

For example, a B-service calls one or more I-services to fulfill its task. From a composite service perspective we can say that in this case the B-service is the composite service, the I-services are service parts needed to implement the composite service. So, the implementation of the composite service depends on service calls to service parts (i.e. underlying services).

Transaction enclosing types	Transactions related through	Service interactions
Within the same aspect organization	Action rules	Orchestration
Across aspect organizations	Actor shaping	Service *parts* are called by the implementation of the *composite* service

Table 7.1: Service interaction types

Table 7.1 shows an overview of the different service interaction types based on the different transaction enclosing types. As exhibited we distinguish between orchestrations and composite-part relationships. Orchestrations coordinate sequences of service calls within the same service layer, thereby implementing a business process. Figure 7.4 shows an example orchestration. Each activity in the process calls one service, thereby creating a sequence of service calls. An orchestration is defined using the UML activity diagram notation [75]. We use this notation because it is standardized and defines the basic elements for defining flow diagrams. The coordination part of the business component construction model consists of a set of such orchestrations. These orchestrations reflect business processes and thus need to be flexible because they have to be adapted as fast as possible when the organization wants to change.

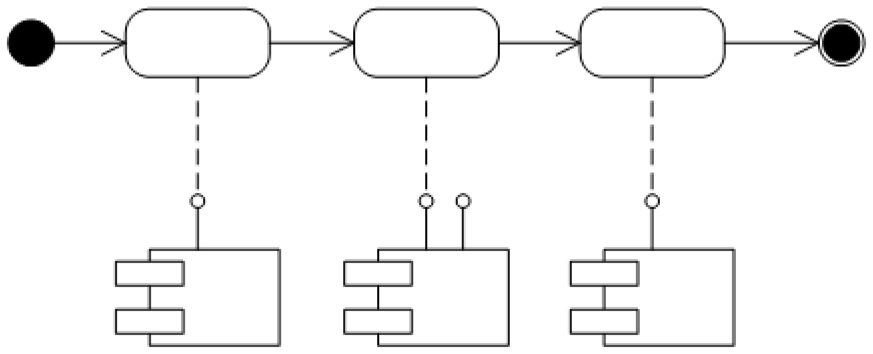


Figure 7.4: Orchestration

Composite-part relationships, on the other hand, are much less flexible. The connection between a composite and its parts is relatively strong because parts are not just triggered, they are an essential part of the implementation of the composite. One composite is implemented by one or more parts on another ("lower") service layer. While the composite-part relationship is part of the service implementation, it is not modeled in the business component construction model. In the business component construction model we only define for each component for each of it's published services what service dependencies it has, i.e. what services are needed to implement it. All these services together are the consumed services of a component (see Figure 7.3). The definition of the implementation and at what points the consumed services are called will be described in the business component implementation model (see Chapter 8).

Concluding we can say that the business component construction model is a multi-model consisting of the following aspect models:

- *Component diagram*: models all components and their relations, expressed in the UML component diagram notation. For each component we define:
  - *Provided services*: the services grouped in the component. In the diagram the service names are used, they refer to the service specifications in the service specification model.
  - *Information objects*: the information objects grouped in the component.

  - *Consumed services*: the services needed for the implementation of the component (i.e. needed for the implementation of the published services of the component).

- *Service orchestrations*: models sequences of service calls on the same service layer, expressed in the UML activity diagram notation. A service orchestration supports a business process.

7.2 Service-oriented construction design

As explained in section 4.6 the step between service specification model and business component construction model is construction design. In section 4.5 we have explained that for implementing complex ICT systems our MDEE approach needs to comply to the principles of SOA. Hence, following [93] the construction design step in which we identify components can be called service-oriented construction design (SoCD).

During construction design the business component construction model is specified using the organization model and service specification model as starting point. As function design consists of service identification and service specification, construction design consists of component identification and component specification. In addition we also have to define the service orchestrations. This means that the SoCD step in our MDEE approach consists of the following steps:

- Service orchestration design
- Business component identification
- Business component specification

As we explained in the previous section, service orchestrations are based on the action rules defined in the organization model. The action rules define in what order and on what conditions transactions are executed. Another part of the organization model, the process model, gives an overview of a subset of this information in a more readable way. From the service specification model we can derive the relations between services and transactions.

In principle a service orchestration is created for each process definition. Each transaction in the process model will be translated into an activity in the service orchestration representing a service call to the service supporting the transaction. If the action rule defining the relation between two transactions contains conditions, these conditions have to be added to the service orchestration as decisions in the flow. We will show an example services orchestration and how we derived it from the organization model and service model in section 7.3.

The service orchestration design step is based on two assumptions:

- Each transaction is supported by exactly one service. We have made this assumption in section 6.2.
- In section 5.4 we stated that dissent and cancellation patterns are outside the scope of this thesis.

Both assumptions affect the way service orchestrations are modeled. The first assumption means that we always have exactly one invoke activity for each transaction. The second assumption means that we always assume that a transaction will be promised, stated, and accepted. This means we do not need additional decisions and service calls in our service orchestrations to handle the cancellation patterns. Although we need these assumptions for scoping our work, we will evaluate the impact and possible directions for future research in the evaluation of our MDEE approach (section 9.3).

After the service orchestrations are derived from the organization model and service specification model the business component identification step is executed. Two types of business component identification exist: forward identification, i.e. identifying business components from requirements models, and reverse identification, i.e. identifying components from existing software [99]. We focus on forward identification, in which the Cohesion-Coupling based Clustering Analysis Methods [99] obtained widespread attention. In these methods researchers try to cluster business models according to the 'high cohesion and low coupling' principle. The basic idea is to transform business models into a weighted directional graph, in which business elements are nodes and the relations between these elements are edges. The key difficulty in these methods is the calculation of the dependency strength for each edge, which is often based on predefined values for the different dependency types.

Example methods following this approach are presented in [55] and [40]. BCI3D, mentioned before in section 6.2, also follows this approach. We use BCI3D for the business component identification step of our MDEE approach because it is general enough to be used for all kinds of models and its usefulness for identifying business components based on DEMO models has already been shown in [5, 4].

BCI3D aims at grouping functions and their corresponding information objects into business components using clustering algorithms [6]. The input for the clustering algorithms in BCI3D can be an organization model based on DEMO as shown in [5, 4]. After setting weights for the different relationship types between transactions, between object classes and transactions, and between object classes, the algorithms can determine the optimal clustering of transactions and information objects in components.

Our approach slightly differs from the method shown in [5, 4], because we also take the I- and D-organization into account and we already have specified a service model. However, most of the mappings between the organization model and the input of BCI3D can be used in the same way. The input for BCI3D consists of a set of tables which are transformed in a graph. After that the clustering algorithms can be executed and a set of business components is calculated.

Table 7.2 exhibits the first part of the set of input tables and how they can be formulated based on the organization model and the service model. The actor table lists the actors and what services they provide and consume. The extern table lists the external components and their shortcuts for reference from other tables. The io table lists all information objects, their shortcuts, and optionally the external component they are part of. Last, the fun table lists all services, their shortcuts, and optionally the external component they are part of.

Table 7.3 exhibits the second part of the set of input tables and how they can be formulated based on the organization model. The ioio table lists the relations among information objects, the funfun table lists the relations among services, and the iofun

Table	Column	Explanation	Derivation rule
actor	actor	actor name	Name of actor in the Construction Model of the organization model
	shortcut	shortcut for graph visualization	-
	performs	shortcut of functions this actor consumes and provides	Take each transaction from the Construction Model associated with the actor. Search for the service supporting the transaction in the service model. Lookup the shortcut for the service in the fun table.
extern	extern	external components	Name of external actor in the Construction Model.
	shortcut	shortcut for graph visualization	-
io	io	the name of the information object (object class or result type)	Name of object class in the State Model of the organization model or the name of a result type from the TRT.
	shortcut	shortcut for graph visualization	-
	is part of external component	optionally the name of an external component	If the executor of the transactions retrieving and storing this information is an external actor, the shortcut of the associated external component from the extern table needs to be added.
fun	function	function name	Name of service in the service model.
	shortcut	shortcut for graph visualization	-
	is part of external component	optionally the name of an external component	If the executor of the transaction supported by this service is an external actor, the shortcut of the associated external component from the extern table needs to be added.

Table 7.2: BCI3D input table derivation rules, part 1

table lists the relations between information objects and services. Note that in the funfun table both service relation types (as explained in the previous section) are used to define the relationships between functions. From the service orchestrations it can be derived which services are called after each other. From the process model (part of the organization model) it can be derived which services are called by which service implementation (i.e. composite-part relationships). This relationship is based on the shaping relations between actors in the organization model and shown in the process model by relations between transactions in different aspect organizations.

Table	Column	Explanation	Derivation rule
ioio	io-io	shortcut of information object	-
	related-to	the information objects this object is related to	List the related objects based on the information in the State Model.
	part-of	the information object this object has a whole-part relationship with	List the appropriate objects based on the information in the State Model. The choice between related-to and part-of needs to be made manual, because this information isn't specified in the State Model.
	evolution-of	shortcut of the information object this object is a specialization of	If this object is a derived statum type the element it is derived of should be listed. This information can be derived from the State Model which contains four types of derivation: partition, aggregation, specialization, and generalization.
	state-of	shortcut of the information object this object is a state of	If the information object is a factum type (defined in the State Model), the object it is associated with should be put here.
funfun	fun-fun	shortcut of function	-
	standard (1..1)	the functions always triggered once by this function	The services always called once after the execution of this service (can be derived from the service orchestrations) or by the implementation of this service (can be derived from the process model)
	optional (0..1, 0..n)	the functions optionally triggered by this function	The services optionally / conditionally called after the execution of this service (can be derived from the service orchestrations) or by the implementation of this service (can be derived from the process model)
	frequent (1..n)	the functions triggered one or more times by this function	The services called one or more times after the execution of this service (can be derived from the service orchestrations) or by the implementation of this service (can be derived from the process model)
	notify	the functions notified by this function	The services called asynchronously (notify) after the execution of this service (can be derived from the service orchestrations)
iofun	io-fun	shortcut of information object	-
	create	the functions in which this object is created	The services in the service model having this object as production fact or the production fact is a state of this object.
	use	the functions in which this object is used	The service in the service model having this object included in their production information used field.

Table 7.3: BCI3D input table derivation rules, part 2

While BCI3D handles multiple relationship types between functions (i.e. standard, optional, frequent, notify) which can all get different dependency strengths, we can precisely define the dependency strength for the different service relationships.

Figure 7.5 shows the full business component identification approach. After the tables are filled based on the organization model, service model and service orchestrations (defined previously in the construction design step), a graph is created with information objects and functions (in our case services) as nodes and the defined relations as edges. After a weight is specified for each type of relation, a starting solution (decomposition of the graph) is generated using a start algorithm (e.g. the greedy graph partitioning algorithm [56]). Next an improvement algorithm (e.g. the Kernighan and Lin graph-partitioning algorithm [59]) improves this starting solution to an optimal solution. As explained in [16], multiple algorithms can be chosen. The result is an optimal decomposition of the given graph into sub graphs with high intern costs and low extern costs, i.e. components with high cohesion and low coupling.

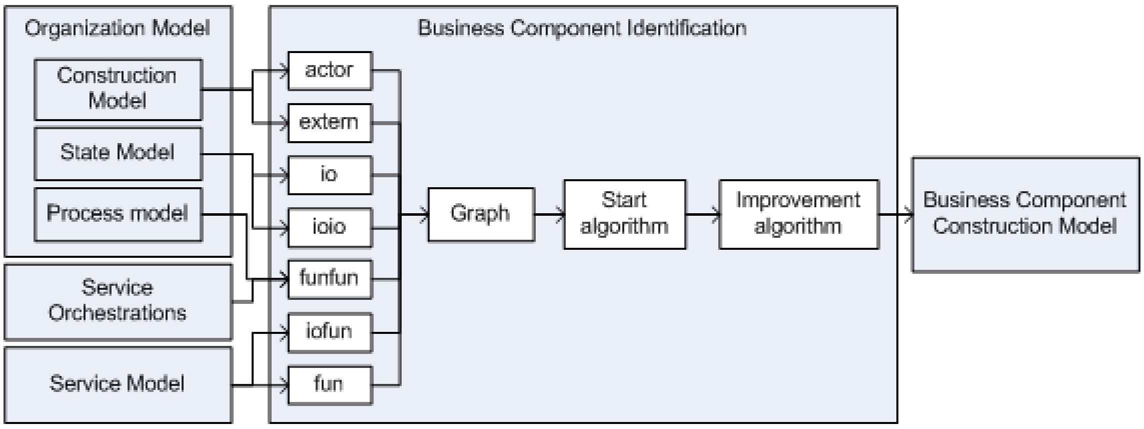


Figure 7.5: Business Component Identification

In section 7.1 we distinguished between two different scenario's to group services in components. In scenario 1 the component identification step is executed once with all services and information objects as input. In scenario 2 the component identification step is executed three times. With B-services as input leading to B-components, with I-services as input leading to I-components, and with D-services and information objects as input leading to D-components.

After executing the business component identification step, the components need to be specified. As stated in the previous section a component is described by the services it provides, the information objects it contains, and the services it consumes. The provided services and information objects are a direct result of the component identification step, i.e. in the identification step we have grouped the services and information objects in components. Consumed services are also a result of the component identification step. However, not each edge in the graph output of BCI3D is a consumed service. In the BCI3D output three different relations between components can be distinguished:

- *Relations between information objects (io-io)*: these relations have been used to identify the optimal component arrangement. They are not part of the interface of a component because information objects do not *use* each other. It can happen that the relation between information objects needs to be defined and stored, but that's always done at the service level.

- *Relations between services and information objects (fun-io)*: these relations mean that a service uses an information object. These relations are used by BCI3D to identify the optimal components but they do not define the consumed services of a component. Information objects can be created and used by services without storing them into a datastore. If they are stored into or retrieved from a datastore an explicit call to a D-service exists. So, a fun-io edge means the services need to know the definition of the information object, it does not mean the service calls a D-service for storing or retrieving the information object. Such relations are defined as fun-fun relations.
- *Relations between services (fun-fun)*: only these relations are used to define the consumed services of a component. However, as explained before only the composite-part relations are consumed services. Other service relations are defined in the orchestration model.

The end result of the service-oriented construction design step is a business component construction model consisting of a component diagram and a set of service orchestrations. This model is described in detail in the previous section.

7.3 Example construction model

Let's execute the construction design step for the Protector case to make things a bit more clear. In section 7.1 we have shown two possible scenario's for constructing a business component construction model. In this section we will show an example for each scenario based on the Protector case. In scenario 1, all services are grouped in components based on their relationships. Hence we have components containing B-, I-, and D-services (see Figure 7.1). In scenario 2 we make a distinction between three types of components, B-components, I-components, and D-components, containing respectively B-, I- and D-services (see Figure 7.2).

Example based on scenario 1

The first construction design step is to design the service orchestrations. We will execute this step for the policy binding process in the B-organization shown in the process model in Figure 5.2 (part of the organization model). We will explain how elements of the organization model (the process model and action model for the policy binding part of the organization) can be transformed into the service orchestration exhibited in Figure 7.6.

The policy binding process starts when a potential individual policy holder requests a policy binding (transaction T05). This means the service orchestrations starts with invoking the service supporting transaction T05 (we refer to the service in the orchestration with the shortcut T05).

After T05 has been promised, transaction T27 is requested. This is described in the following action rule from the organization model (described in section 5.4):

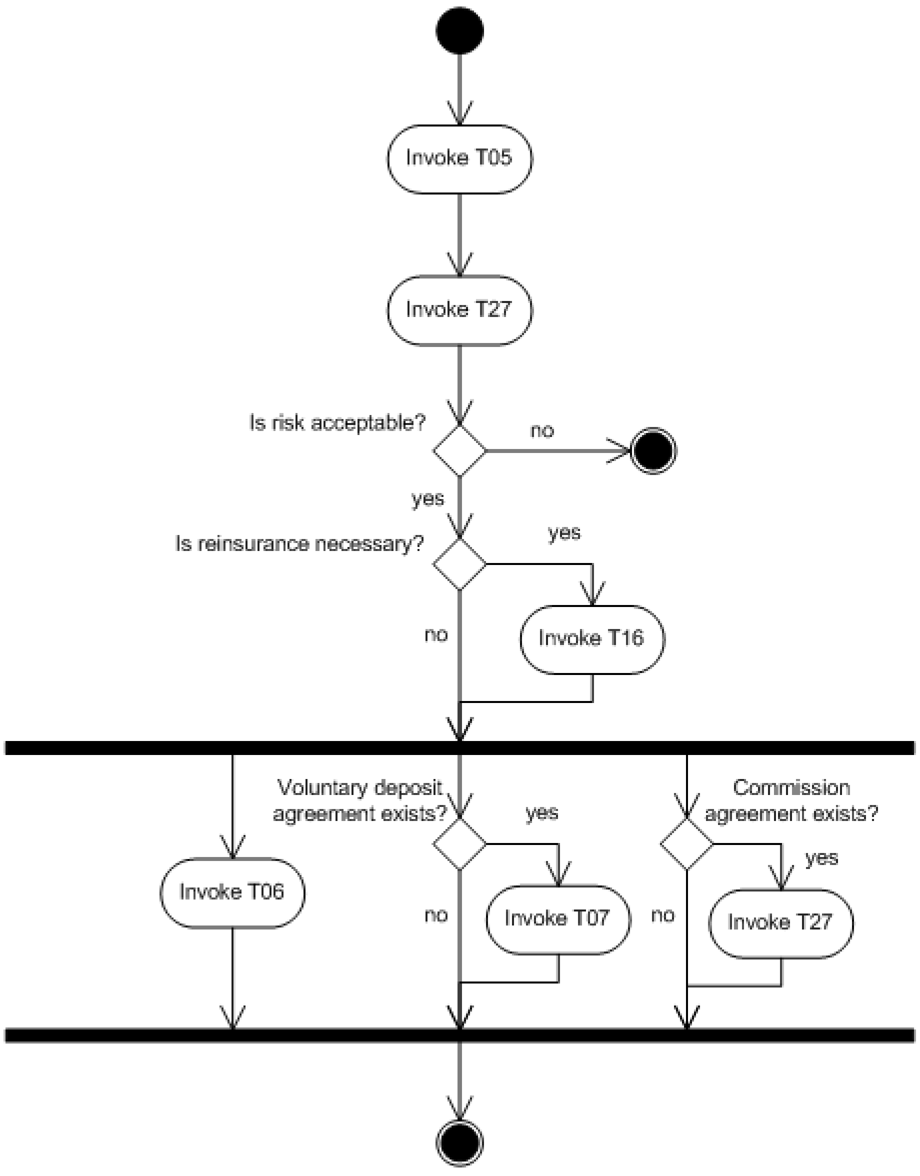


Figure 7.6: Service orchestration for the policy binding process

on promised T05(Policy, Insured, InsurancePremium) **with** policy(new Policy) = Po
  request T27(Policy, Insured, InsurancePremium)
no

As no condition is specified in this action rule we do not include a decision in the service orchestration, we add an activity invoking service T27. Once transaction T27 is requested the following action rule is triggered:

on requested T27(Policy, Insured, InsurancePremium)

```
    **if** <risk is acceptable> -> promise T27(Policy, Insured, InsurancePremium)
    **not** <risk is acceptable> -> decline T27(Policy, Insured, InsurancePremium)
    **fi**
**no**
```

As shown in the action rule a condition is defined checking if the risk is acceptable, if so the process continues, otherwise the process stops. Hence, we need to add a decision to the service orchestration performing the same check.

If the process continues, i.e. transaction T27 is promised, the following action rule is triggered:

```
on promised T27(Policy, Insured, PolicyCollection, InsurancePremium)
    if <reinsurance is necessary> -> request T16(Policy, Insured, PolicyCollection)
    not <reinsurance is necessary> -> execute T27(Policy, Insured, InsurancePremium)
    fi
no
```

Again we see a condition in the action rule, this time checking if reinsurance is necessary. We add a decision to the service orchestration checking if reinsurance is necessary, if so service T16 will be invoked, if not the orchestration continues without invoking T16.

Looking at the process diagram in Figure 5.2 we see that the execution phase of transaction T05 has been completed. The result is stated to the potential individual policy holder. As dissent and cancellation patterns are outside the scope of this thesis we assume that the result always will be accepted. This means that the following action rules will be triggered:

```
on accepted T05(Insurant, InsurancePremium)
    request T06(Insurant, InsurancePremium)
no

on accepted T05(Policy, Beneficiary, Insured, Insurant)
    if <voluntary deposit agreement exists> -> request T07(Policy)
    fi
no

on accepted T05(Policy, Commission)
    if <commission agreements exists for Policy> -> request T26(Commission)
    fi
no
```

As all three action rules are triggered at the same moment we add a fork to the service orchestration. This means that we can define a parallel flow for each action rule listed above. The first action rule does not contain a condition, hence we just add an activity to invoke service T06 as the most left flow. For the second action rule we add a

Table	Relationship	Weight
ioio	related-to	5
	part-of	10
	evolution-of	10
	state-of	5
funfun	standard (1..1)	15
	optional (0..1, 0..n)	5
	frequent (1..n)	17
	notify	2
iofun	create	15
	use	10

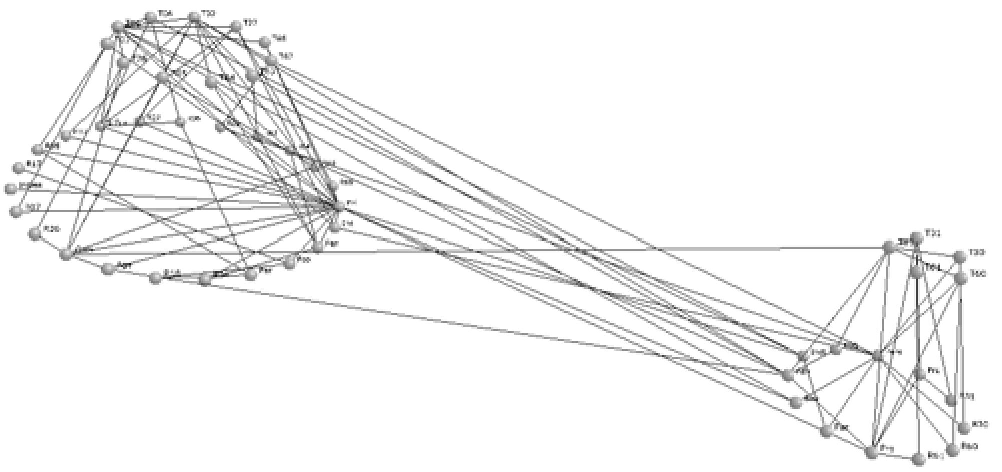
Table 7.4: BCI3D relationship weights for scenario 1

decision checking if a voluntary deposit agreement exist, if so service T07 is invoked, otherwise the flow continues without invoking T07. The third action rules contains a condition, hence we add a decision checking if a commission agreement exists, if so service T27 will be invoked, otherwise the flow continues without invoking service T27. The resulting orchestration for the policy binding process is exhibited in Figure 7.6.

The second construction design step is component identification. Following the process described in section 7.2 we have created the BCI3D input tables based on the organization model and the service model, which we used as input for the BCI3D tool described in detail in [16]. Table 7.4 shows the weights we used for each type of relation. The choice for these weights are quite arbitrary. In practice these weights are based on experience and tuning of the result.

Using the greedy graph partitioning algorithm [56] for generating a start solution and the Kernighan and Lin graph-partitioning algorithm [59] for improving the solution, BCI3D generated the result shown in Figure 7.7a. We have used the transaction numbers as shortcut to refer to service names. As exhibited in Table 7.7b we have identified two components, component 1 and 2, corresponding with respectively the 'policy binding' and 'product advising' parts of the organization model. Note that using other weights will lead to other components, for example more and smaller components.

The final construction design step is component specification, leading to the component diagram shown in Figure 7.8. Table 7.7b already exhibited the provided services, information objects, and consumed services for each component. Because the composite-part relationship is rather strong the components do not have consumed services. Each service consumed by another service (i.e. a composite-part relationship) is put in the same component (by BCI3D) as the consuming service. Figure 7.7a shows edges between the two components, but these are all fun-io or io-io relationships. As explained in section 7.2 these relationships do not translate into consumed services.



(a) BCI3D output

Component 1	Provided services	T05, T06, T32, T27, T66, T62, T63, T64, T65, T26, T07
	Information objects	R05, R32, R27, R06, R62, R63, R64, R65, R66, R04, R18, R26, R07, R17, Insurance Premium (InPre), Policy (Pol), Insured (Ins), Policy Quotation (Pqo), Policy Collection (Pco), Period (Per), Agent (Age), Commission (Com), Insurance Benefit (InBen)
	Consumed services	-
Component 2	Services	T01, T31, T30, T60, T61
	Information objects	R01, R31, R30, R60, R61, Product Advice (Adv), Insurant (Inst), Person (Pers), Product Composition (Prc), Product (Pro), Party (Par), Beneficiary (Ben)
	Consumed services	-

(b) Component specification

Figure 7.7: Identified components for scenario 1

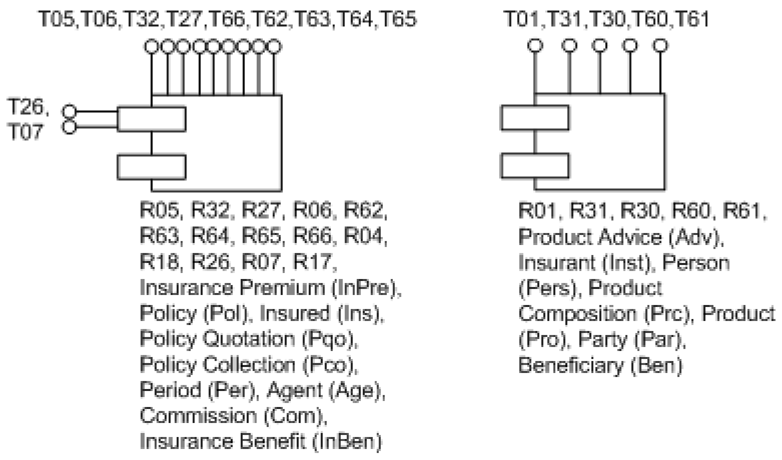

T05,T06,T32,T27,T66,T62,T63,T64,T65
T26,
T07
R05, R32, R27, R06, R62, R63, R64, R65, R66, R04, R18, R26, R07, R17, Insurance Premium (InPre), Policy (Pol), Insured (Ins), Policy Quotation (Pqo), Policy Collection (Pco), Period (Per), Agent (Age), Commission (Com), Insurance Benefit (InBen)
T01,T31,T30,T60,T61
R01, R31, R30, R60, R61, Product Advice (Adv), Insurant (Inst), Person (Pers), Product Composition (Prc), Product (Pro), Party (Par), Beneficiary (Ben)

Figure 7.8: Component diagram for scenario 1

Example based on scenario 2

The different scenario's only affect the arrangement of services and information objects in components. The service orchestrations are the same for both scenario's. This means the service orchestration exhibited in Figure 7.6 is also part of the business component construction model for scenario 2.

The business component identification step is different for scenario 2. We created a set of BCI3D input tables for each type of component, i.e. we executed the component identification process for B-components, I-components, and D-components. All information objects and result types are part of D-components because on the D-level the data is transported an stored. The B- and I-level make use of D-services to access the information objects and result types.

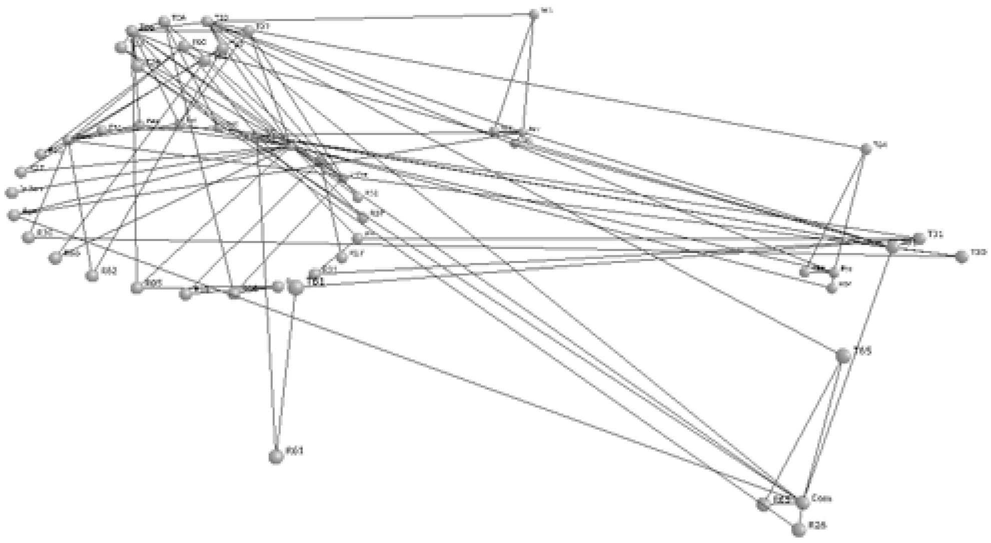
For identifying the D-components we have created BCI3D input tables containing all D-services, information objects, and result types, and their relationships. We have used the same weights as for scenario 1 (see Table 7.4) which resulted in the BCI3D output exhibited in Figure 7.9a representing the five D-components shown in Table 7.9b. Because D-services do not have relations among each other in our scenario, the arrangement in components is fully based on the relationships between the information objects. D-services also do not have consumed services, because their implementation does not call other services. D-services are only used by other services (B- and I-services).

It is notable that component 1 is much bigger than the other ones. This is because we did not identify all D-services, we have only identified the ones needed in the 'product advising' and 'policy binding' part of the organization model. So, all information objects not connected to any service are put in component 1 because of their relations among each other.

Component 1	Provided services	T30, T31, T32
	Information objects	-
	Consumed services	T61, T62, T63, T64, T65

Table 7.5: Identified I-component for scenario 2

On the I-level we only have three I-services: T30, T31, T32. Because they do not have relationships among each other they can be arbitrary grouped in components. In this scenario (i.e. scenario 2) shaping relations between services (i.e. relations between services supporting different aspect organizations) do not play a role because we identify B-, I-, and D-components separately. If we put the three I-services in one component, we can specify that component as exhibited in Table 7.5. The consumed services are the services needed for the implementation of the provided services (T30, T31, T32). This can be derived from the infological and datalogical construction model exhibited in Figure 5.4 and Figure 5.5. The shaping relations between actors indicate a composite-part relationship between the services supporting the related transactions (as explained in section 7.1).

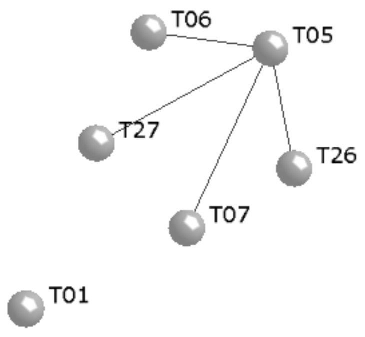


(a) BCI3D output

Component 1	Provided services	T60, T62, T66
	Information objects	R60, R32, R07, R17, R31, R06, R18, R05, R62, R66, R30, R27, Person (Pers), Party (Par), Beneficiary (Ben), Insured (Ins), Insurant (Inst), Product (Pro), Policy (Pol), Policy Collection (Pco), Insurance Premium (InPre), Product Composition (Prc), Period (Per), Agent (Age), Insurance Benefit (InBen)
	Consumed services	-
Component 2	Provided services	T63
	Information objects	R63, R01, Product Advice (Adv)
	Consumed services	-
Component 3	Provided services	T64
	Information objects	R64, R04, Policy Quotation (Pqo)
	Consumed services	-
Component 4	Provided services	T65
	Information objects	R65, R26, Commission (Com)
	Consumed services	-
Component 5	Provided services	T61
	Information objects	R61
	Consumed services	-

(b) Component specification

Figure 7.9: Identified D-components for scenario 2



(a) BCI3D output

Component 1	Provided services	T05, T06, T27, T07, T26
	Information objects	-
	Consumed services	T32, T66
Component 2	Provided services	T01
	Information objects	-
	Consumed services	T30, T31, T60

(b) Component specification

Figure 7.10: Identified B-components for scenario 2

Identifying B-components is a bit more interesting for this scenario because the identified B-services have different types of relations among each other. However, due to our small example the component arrangement stays quite straightforward. We have again executed the process described in section 7.2 to translate the organization model and service model into BCI3D input tables. Using the weights presented in Table 7.4 BCI3D identified the components as exhibited in Figure 7.10a and Table 7.10b. We again see that the components correspond with the 'policy binding' and 'product advising' part of the organization model. Note that for component 1 only the consumed services of the provided service T05 are defined and not the consumed services of the other provided services (T06, T27, T07, T26), because only that part of the organization model has been specified in detail as example in section 5.4.

The final component diagram for scenario 2 is shown in Figure 7.11. As exhibited we have three layers of components: B-components, I-components, and D-components. Each component consumes the provided services of components on lower layers.

We have shown an example component diagram for two different scenario's. Although these examples exhibit the use of the construction design step of our MDEE approach, they are to small to draw conclusions regarding the applicability of the different scenario's. As this applicability also depends on the existing IT landscape we leave the question when to use which scenario for future research.

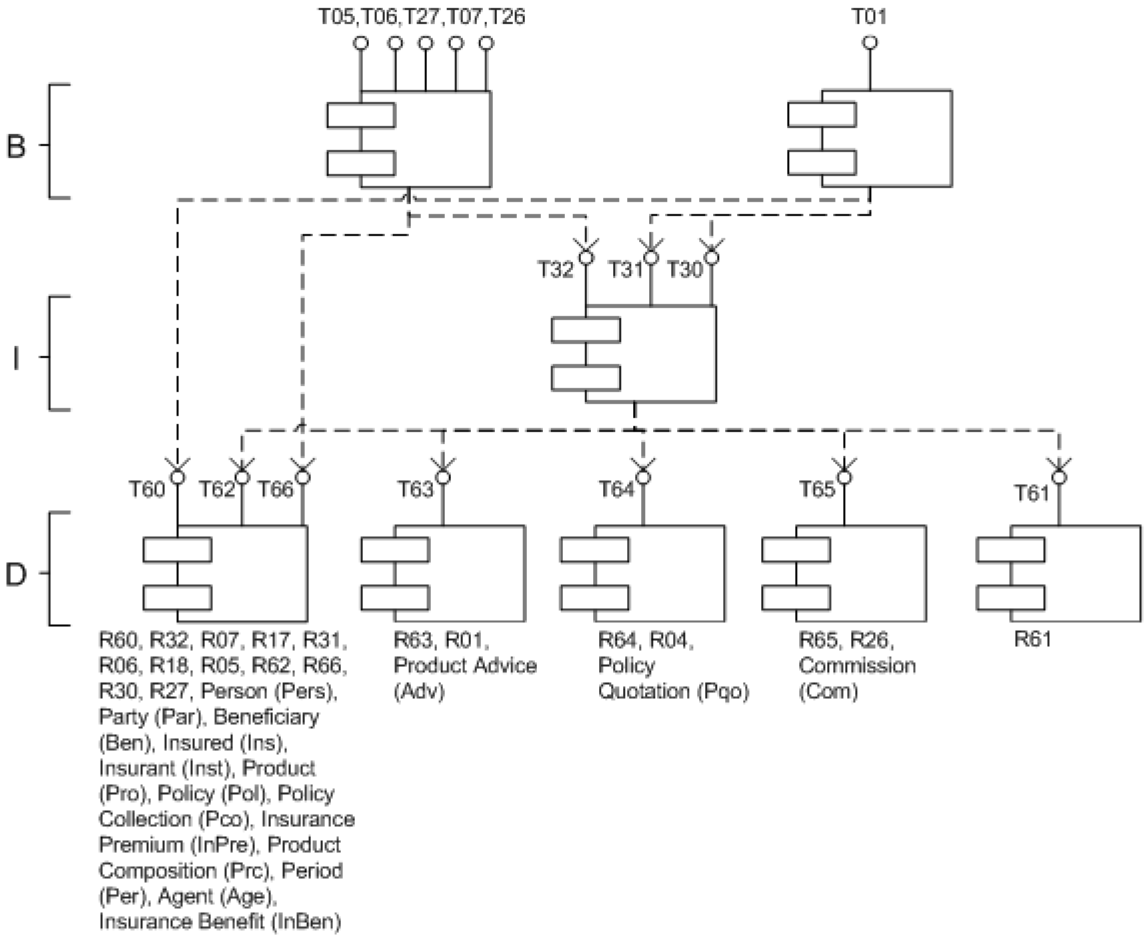

T05,T06,T27,T07,T26
T01
B
I
D
T32
T31
T30
T60
T62
T66
T63
T64
T65
T61
R60, R32, R07, R17, R31, R06, R18, R05, R62, R66, R30, R27, Person (Pers), Party (Par), Beneficiary (Ben), Insured (Ins), Insurant (Inst), Product (Pro), Policy (Pol), Policy Collection (Pco), Insurance Premium (InPre), Product Composition (Prc), Period (Per), Agent (Age), Insurance Benefit (InBen)
R63, R01, Product Advice (Adv)
R64, R04, Policy Quotation (Pqo)
R65, R26, Commission (Com)
R61

Figure 7.11: Component diagram for scenario 2

Chapter 8

An executable Implementation Model

In the previous chapters we have described the function design step and the construction design step of our MDEE approach. These steps have resulted in a service specification model and a business component construction model. The main question for this chapter is how we can turn these models into an executable model, thereby leading to the end result of our MDEE approach: a working software application. To come to this result we have to describe the last two steps in our MDEE approach, engineering and implementation.

In this chapter we first explain how we can construct an executable model (8.1). This executable model, the business component implementation model, consists of component implementations. Therefor we research how to implement a component containing the service orchestrations (8.2). We also research how to define the implementations for the different service types, together forming the implementation of the other components (8.3, 8.4, and 8.5). We also need to define the SCA configurations needed to deploy the components.

After the full business component implementation model has been defined, we will explain how this model can be derived from the previous models in our MDEE approach, i.e. the engineering step (8.7). Implementing the business component implementation model means executing it on engines. We will cover the needed engines briefly and explain how to derive the input for these engines from the business component implementation model (8.8). This chapter is concluded with an example business component implementation model based on the Protector case (8.9).

8.1 How to come to an executable model?

The combined service specification model and business component construction model give a complete high-level model of an IT system. It is high-level because a lot of details are not fully specified yet. The question at hand is how we can transform this model into a model which can be implemented on technology, i.e. a model which is executable on engines. This means that additional implementation details need to be specified. What specific details are needed depends on the used engines. However, what engines we need depends on the architecture and the way the models are structured.

As explained in section 4.5 our MDEE approach follows the architecture principles of SOA. The service oriented paradigm is, at its core, a model of distributed software components, built around the idea of multi-protocol interoperability and standardized component contracts [25]. The functional model and the constructional model of the object system therefore respectively model services and components. In the previous Chapter we have seen that the business component construction model consists of service orchestrations and a component diagram. Each component in the component diagram is defined by its provided and consumed services and the information objects it contains. Each provided service refers to a service specification in the service specification model.

In section 4.5 we have defined that a component has a contractually specified interface and that it can be deployed independently. In Chapter 7 we have seen that the contract of a component defines its provided and consumed services. The question is how we can engineer such component descriptions into an executable component. Bachman et al. [13] give an overview of the technical elements needed for Component-Based Software Engineering (CBSE) including the elements needed to deploy a component. The two main elements needed to deploy components are a component model and a component framework [13]:

- A *component model* is the set of component types, their interfaces (i.e. contract), and, additionally, a specification of the allowable patterns of interaction among component types. In principle component models specify the design rules that must be obeyed by component implementations.

- A *component framework* provides a variety of runtime services to support and enforce the component model (e.g. communication services, security services, etc.). In many respects component frameworks are like special-purpose operating systems, although they operate at much higher levels of abstraction. Components are deployed on a component framework.

Example component models are Sun Microsystems' Enterprise JavaBeans (EJB) [27], Microsoft's COM+, OMG's CORBA Component Model (CCM) [73], and OASIS' SCA (described before in section 4.5). Both CCM and SCA are vendor-independent industry standards. However, the SCA is more recent and has the ability to 'bind' legacy components or services, accessed normally by technologies like for example EJB and CORBA. We will therefore target the SCA as the component model for our MDEE approach.

Based on the previous we can say that we need the elements exhibited in Figure 8.1 to make our business component construction model executable. We need a component framework implementing the SCA standard. On this component framework a component can be deployed. A component consists of a contract and an implementation and is specified according to the SCA standard.

In the SCA a component consists of a configured piece of implementation providing some business function. An implementation can be specified in any technology (e.g. Java, .Net, other SCA components, WS-BPEL). The business function of a component is published as a set of services. The implementation can have dependencies on the services of other components, the SCA calls these dependencies references. Im-

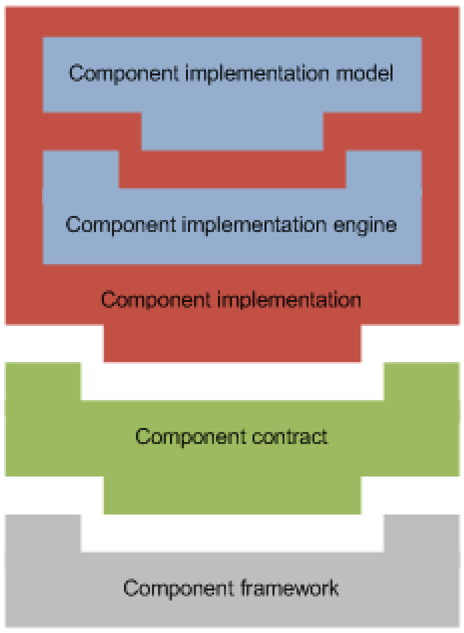

Component implementation model
Component implementation engine
Component implementation
Component contract
Component framework

Figure 8.1: Component implementation

plementations can also have properties which are set by the component (i.e. they are set in the XML configuration of a component).

So, the component contract shown in Figure 8.1 consists of the definition of services, references, properties, and it points to an implementation. All component contracts together form the component model. The communication among components is implemented by the component framework. Most SCA implementations support multiple communication protocols.

Although the component implementation in SCA can be in any technology, in our MDEE approach we want to have a model-driven implementation. As explained in section 4.1 this means that the implementation consist of a model which can be executed (i.e. interpreted) on an engine. Hence, as exhibited in Figure 8.1, a component implementation consists of a component implementation engine and a component implementation model.

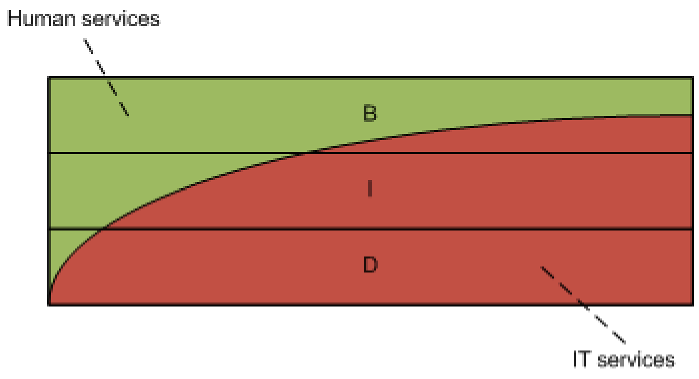


Figure 8.2: Service layers and the distinction between Human and IT services

The implementation of a component can be seen as a combination of the implementations of the services provided by that component. If we look at the implementations of the different service types we can distinguish between three different implementation types. These types are based on the observation shown in Figure 8.2. This Figure shows the three service layers (B, I, and D) and the distinction between human and IT services. The intention of the curve is to give an idea of the approximate proportions of human and IT services in each layer.

B-services support ontological transactions. As explained in section 3.4 ontological acts cannot be fully automated. Hence, B-services are almost always human services. I-services can be automated and thus are mostly IT services. However, some of them can be human services. See for example the identified I-service Formulate-PersonalDetails shown in table 6.3, which is a human service. D-services are almost always IT services, in some rare cases D-services can be human services. For example a human sends a letter to a customer. Transmitting a message is a D-service, which is in this case not automated by IT technology.

We can group service types with similar implementations, thereby reducing the number of different service implementation types. The implementation of human services whether they are B-, I-, or D-services will be the same. They have to represent a task for a human and capture the resulting fact. IT service implementations on B and I level are also the same. Based on some data a calculation or decision needs to be made. As D-service implementations only store or retrieve data they differ from the other IT-services. That's why we distinguish between IT service implementations and data service implementations.

Based on the previous we can make a distinction between the following service implementation types:

- Human service implementations: providing an implementation for almost all B-services and maybe a few I-services and D-services.
- IT service implementations: providing an implementation for almost all I-services and maybe a few B-services.
- Data service implementations: providing an implementation for all D-IT-services.

Besides component specifications (including the service specifications) the business component construction model also contains service orchestrations. Hence, we also need to define how these orchestrations are implemented. While we target a component-based runtime environment each part of the IT system is represented by one or more components, meaning that service orchestrations also need to be executed as a component implementation. Such an 'orchestration component' consists of the same elements as the other components, i.e. a contract (defining the provided services and references to other services) and an implementation consisting of an engine and an implementation model (see Figure 8.1). Each service orchestration will become a provided service, the service calls from the orchestrations to the provided services of other components will become the references of the orchestration component.

Concluding we can answer the question how to come to an executable model. To come to an executable model based on the previous models in our MDEE approach (i.e. the service specification model and the business component construction model) we need to define:

- the contract, engine, and implementation model of the orchestration component. We will do this in section 8.2.
- the contract, engine, and implementation model of each component defined in the business component construction model. This means that we need to define human service implementations (we will do this in section 8.3), IT service implementations (we will do this in section 8.4), and data service implementations (we will do this in section 8.5).
- the engineering step of our MDEE approach, i.e. how to derive the SCA component model and the previously listed contracts and implementation models (together forming the business component implementation model) from the service specification model and business component construction model. We will do this in section 8.6 and 8.7.
- the implementation step of our MDEE approach, i.e. how to execute the implementation model on engines. We will do this in section 8.8.

8.2 Implementing the orchestration component

As shown in the previous section a deployed component consists of a contract and an implementation. The contract is an SCA component description. The implementation consists of an engine executing an implementation model. The implementation model should describe the service orchestrations in such a way that they can directly be executed on an orchestration engine. To research the needed implementation model and engine for a service orchestration we assume that we have a component consisting of one service orchestration. In section 8.7 we will explain how to combine multiple service orchestrations in one component.

In the business component construction model we used a UML activity diagram to express the service orchestrations. An example service orchestration has been exhibited in Figure 7.6. As an UML activity is not directly executable we need to translate the service orchestration expressed in this language to an executable language. The example in Figure 7.6 and the description of the service orchestration model part of the business component construction model in section 7.1 show that an executable language for the implementation of the orchestration needs to support:

- Sequential and parallel flows.
- Decision points in these flow.
- Activities to invoke other services.

Several standards exist for describing service interactions. However, for orchestrating services the Web Services Business Process Execution Language (WS-BPEL) [70] is embraced by the industry as vendor-independent standard [83] and almost all commercial engines nowadays support this standard.

It is possible to use WS-BPEL to define an executable service orchestration. The language effectively defines a portable execution format for business processes that rely exclusively on web service resources and XML data [70]. All three points mentioned above are supported by WS-BPEL. WS-BPEL contains, among others, the following activities:

- *Invoke*: the invoke activity is used to call web services offered by service providers.
- *Receive and Reply*: the receive activity waits for incoming messages. The corresponding reply activity sends a response back.
- *if*: the if activity provides conditional behavior and consists of an ordered list of one or more conditional branches. The first branch whose condition holds true is taken, and its contained activity is performed.
- *Sequence*: a sequence activity contains one or more activities that are performed sequentially, in the lexical order in which they appear within the sequence element.
- *Flow*: the flow activity provides concurrency and synchronization. A flow activity contains one or more activities that are performed concurrently.

These are the basic activities needed to model a service orchestration using WS-BPEL. We can quite straightforwardly translate a UML activity diagram into a WS-BPEL specification, thereby constructing the implementation model which is executable on a WS-BPEL engine (we will describe this process in detail in section 8.7, in section 8.9 we will show an example). Hence, the component implementation for the orchestration component consists of an implementation model specified in the WS-BPEL language and executed on a WS-BPEL engine.

The last part we need to define to deploy and run the orchestration component is the component contract. As explained in the previous section a component contract is an SCA component configuration consisting of services, references, and properties. The SCA configuration points to the implementation of the component, which is in this case a WS-BPEL process. In SCA, both services and references correspond to WS-BPEL's concept of partner link. A partner link in WS-BPEL represents both a consumer of a service provided by the business process and a provider of a service to the business process. A WS-BPEL invoke activity, for example, points to a partner link which points to a service provided by another party.

In order to use a WS-BPEL process as a component implementation we need to define a service or reference for each partner link in the WS-BPEL process. Figure 8.3 exhibits how a service is connected to a WS-BPEL receive and reply activity via a partner link. Invoke activities are connected to references. The SCA Client and Implementation Model Specification for WS-BPEL [80] describes in detail how an SCA component configuration can be defined for a WS-BPEL process. In our case we can say that each service orchestration in the orchestration component will lead to a service in the SCA configuration and each invoked service by any of the service orchestration will lead to a reference.

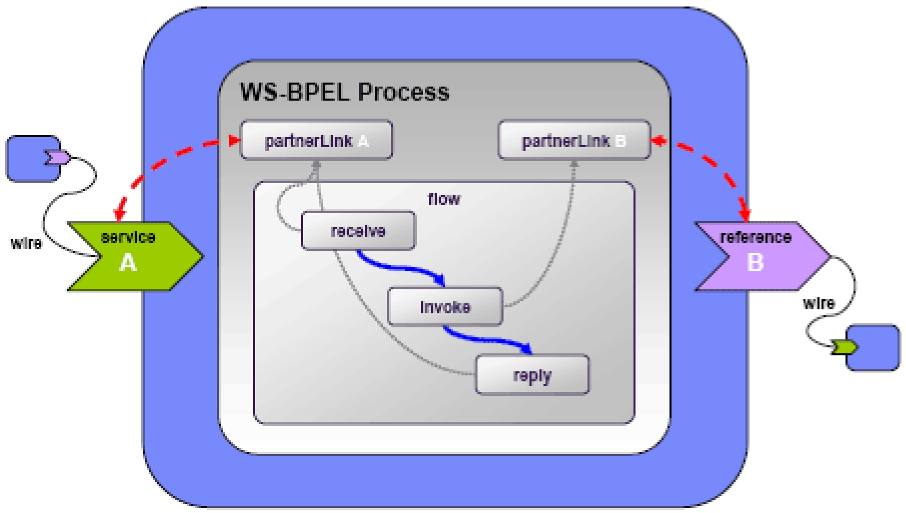


Figure 8.3: A WS-BPEL process as SCA component implementation [80]

8.3 Implementing human services

In section 8.1 we have seen that one of the service implementation types is the human service. This implementation type implements all human services specified in the service specification model, which will, in practice, mostly be B-services. To research the needed implementation model and engine for a human service we assume that we have a component consisting of one human service. This means we can reason about the implementation of a Human service as a component implementation, i.e. we can define an SCA component description (i.e. the component contract), an engine, and an implementation model for this component. In section 8.7 we will explain how the implementations of multiple services can be combined in one component implementation, depending on the component arrangement chosen in the business component construction model described in Chapter 7.

We will first research what implementation model and corresponding engine we need to implement a human service. Although human services are provided by human beings, parts of the service implementation can be automated. Different choices can be made regarding the parts implemented by IT technology. We will focus on an implementation model automating human services as much as possible.

Figure 8.4 exhibits the standard transaction pattern with a human service consumer and a human service provider. The coordination acts (e.g. request, promise, state, and accept) are executed by these human beings. However, the production act, actually creating the requested production fact, can be supported by an IT system. The IT system can present a task to the human service provider describing what he needs to do to produce the production fact. The resulting fact is registered in the IT system. The IT system can also provide the connection to the I- and D-services needed to implement the human service. As shown in Figure 8.4, once the result is promised, the service provider requests a task from the IT system. Requesting a task means claiming an existing task or creating a new task. If the task has been completed the service provider states the result and the consumer can accept the result.

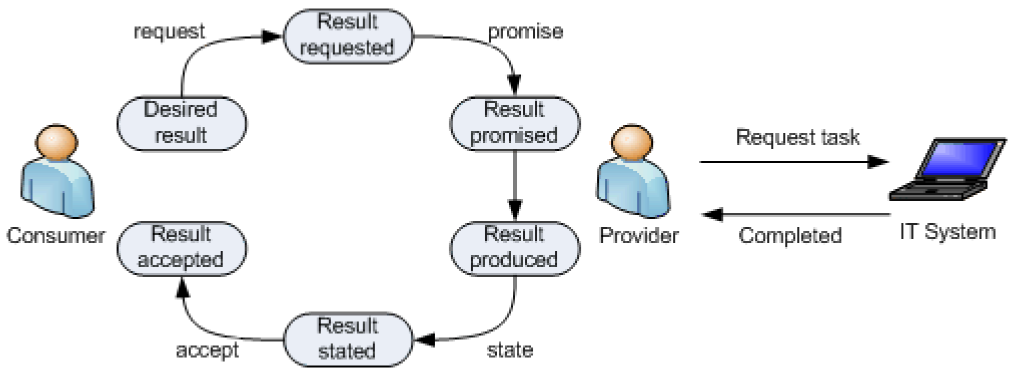


Figure 8.4: The transaction pattern and human tasks

Human tasks, provided by an IT system (as shown in Figure 8.4), are services 'implemented' by people, they allow the integration of humans in service-oriented applications [2]. A human task has two interfaces, an interface to IT systems and an interface to people.

The IT interface exposes the service offered by the task to the IT system. This service can thus be triggered from within the IT system, for example from a service orchestration. Triggering a task service means creating a human task and putting it in the inbox of the associated user(s). The IT interface is bi-directional, meaning that a task can also access IT services (e.g. I- and D-services) to request information needed to fulfill the task.

The human interface allows people to deal with tasks, for example to query for human tasks waiting for them (e.g. using a task inbox), and to work on these tasks. In principle the 'human part' of a task describes what people should do to fulfill the task. A human task has people assigned to it. These assignments define who should be allowed to play a certain role on that task. Human tasks may also specify how task data is rendered on graphical user interfaces of different devices. The information needed to describe and fulfill a task can be requested from the IT system using I- and D-services.

As far as we know the only standard in the field of human tasks is the WS-HumanTask specification [2]. This specification describes tasks in a standard way in which they are directly executable on an engine. WS-HumanTask is part of the BPEL4People [3] extension of WS-BPEL and supported by most WS-BPEL engines.

The main element of WS-HumanTask is a Human Task which is described by the following mandatory elements:

- *name*: uniquely identifies the task.
- *interface*: specifies the operation used to invoke the task (i.e. the IT interface of the task). Invoking a task means creating an instance of it. The needed data is specified with XML Schema [95, 17] (within the WSDL [23] definition of the interface) and needs to be provided in the input message when invoking the task.
- *peopleAssignments*: defines which people or roles are assigned to the task.

- *presentationElements*: definition of the user interface elements needed to allow users to deal with their task (i.e. the human interface of the task).
- *outcome*: defines the business result of the task.
- *deadlines*: defines the timeouts and escalations for the task. Timeouts and escalations allow the specification of a date or time before which the task must reach a specific state. A distinction is made between a start deadline (specifies the time until the task must start) and a completion deadline (specifies the due time of the task).

As shown in Chapter 5 B-transactions are supported by I-transactions and D-transactions via the shaping of actors. In Chapter 6 we have seen that in the same way B-services are supported by I- and D-services. We defined the relationship between a service whose implementation depends on other services as a composite-part relationship.

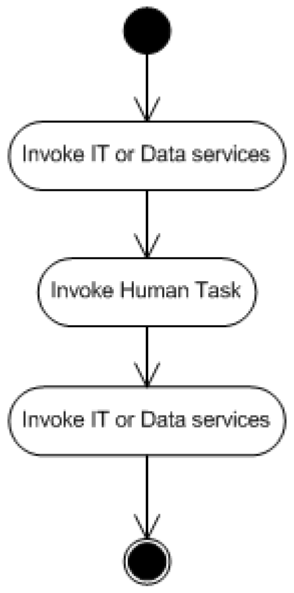


Figure 8.5: Human service invoke pattern

The implementation of a human service always depends on IT or data services. This is reflected by the definition of a Human Task given before. When a Human Task is invoked via its interface, input data needs to be provided. The outcome of a task also needs to be stored using IT or data services. Figure 8.5 exhibits the basic pattern executed when a human service provider claims or creates a task. First one or more IT or data services are invoked. The data retrieved from these services is used as input data for the Human Task invoke. After the Human Task has been completed one or more IT or data services are called with the output of the Human Task as input, i.e. the outcome of the task is stored using IT or data services.

The pattern described in Figure 8.5 is the basis for the human service implementation model. While the WS-HumanTask standard is part of BPEL4People which is an extension of WS-BPEL we can define this implementation model using these executable languages. The following elements need to be defined:

- *WS-BPEL process*: a process containing invoke activities for calling IT or Data services.
- *People activity*: an additional activity in the WS-BPEL process for invoking the Human Task. The people activity can be used if the BPEL4People extension is included in the WS-BPEL process.
- *Human task*: definition of the human task following the WS-HumanTask standard.

The result is a WS-BPEL process which can directly be executed on a WS-BPEL engine supporting BPEL4People. Hence, the component implementation for a component containing a single human service consists of a WS-BPEL process running on its associated engine. The SCA contract of the component can be defined in the same way as described in section 8.2 (i.e. it can be derived from the process definition).

8.4 Implementing IT services

To research the needed implementation model and engine for an IT service we make the same assumption as in the previous section, i.e. that we have a component consisting of one IT service. This means we can reason about the implementation of an IT service as a component implementation, i.e. we can define an SCA component description (i.e. the component contract), an engine, and an implementation model for this component.

We will first research what implementation model and corresponding engine we need to implement an IT service. Several SCA specifications exist describing how to implement a component. One of them is to use WS-BPEL for defining the implementation [80]. We used that approach to implement the orchestration component. To implement a component providing a human service we used an extension of WS-BPEL, BPEL4People. Other SCA specifications include the SCA Java Component Implementation [81]. This specification defines how to implement a component using Java.

The default way to define SCA component implementations is to use Java. However, as we explained in Chapter 4, MDE aims to improve productivity in software development by raising the level of abstraction and automation. Luoma et al. [63] have researched over 20 real-world cases in which MDE is applied with use of DSLs (see section 4.4), they call these approaches Domain-Specific Modeling. They conclude that in all cases, Domain-Specific Modeling had a clear productivity influence due to its higher level of abstraction: it required less modeling work, which could often be carried out by personnel with little or no programming experience.

In principle, visual languages (as opposed to textual ones) would often be ideal to use as domain-specific notations. For a large number of problem domains there exists a natural and intuitive visual representation of artifacts in these domains. Practitioners in these fields have been using these notations for a long time, which reduces training costs and lowers the barriers to the acceptance of a new language [39]. Visual languages are often graph-based, the vertices denote activities, while the edges show the relations between the activities. The notion of an activity is alike in all workflow-like languages, i.e., an activity is atomic and corresponds to a logical unit of work [96].

Based on the previous we can say that we can use a graph-based language for defining the implementation of IT services in a way fulfilling the goals of our MDEE approach (i.e. defining the implementation on a higher abstraction level, more automation by using executable modeling languages, and a higher productivity by enabling domain experts without programming experience to define the implementation). However, the full definition of such a language, including it's abstract syntax, concrete syntax, and semantics, is outside the scope of this thesis. We will define the basic structure and some example activities to show the idea.

The requirements for the language we need are as follows:

- We need to be able to define a flow including decision points.
- We need to be able to invoke other IT services or data services to support composite-part relations between services (as explained in section 7.1).
- We need to be able to define how the production fact is created. While IT services support I-transactions, production acts supported by IT services are infological in nature, meaning that they reproduce, deduce, reason, compute, etc. information (see the distinction axiom in section 3.3). Hence, we need to have activities in our language to manipulate information, to execute calculations, to evaluate conditions based on information, and to validate the content of information.

We can take WS-BPEL as starting point for our language. WS-BPEL is easy to visualize graphically and it contains activities for defining a flow (e.g. the sequence and flow activity), to invoke services (e.g. the invoke, receive, and reply activities), and activities describing decision points or conditions (e.g. the if activity). However, for actually manipulating data or to do calculations WS-BPEL needs to invoke a service providing that functionality. As explained before we want to have an executable implementation model without further refinements. Hence, we need to specialize WS-BPEL to come to a language which includes the needed activities and is directly executable. This is a same kind of specialization as the BPEL4People extension we used to define the implementation of human services. The difference is that no industry standard exists for the specialization we need and we thus have to define this WS-BPEL extension ourselves.

While we need to be able to define how the production fact is created, we need to add the following activities to WS-BPEL:

- *Validate*: used to validate the content of input based on a set of validation rules. The definition of this activity contains a list of data attributes to check and a validation rule for each attribute (e.g. a regular expression).
- *Calculate*: used to do mathematical calculations. The definition of this activity contains a mathematical expression consisting of data attributes and operators.
- *Change*: used to change information objects. The definition of this activity contains an input object and a list of changes. A change consists of the attribute to change and an expression describing the new value.

With these activities in addition to the activities already defined in WS-BPEL we have a language to define the implementation model of IT services. In section 8.9 we will show an example IT service implementation model.

While we defined our own language to specify the implementation model, we also need to define our own engine to execute the implementation model. However, the IT-service implementation language is a specialization of WS-BPEL. This means we can create an engine by specializing a WS-BPEL engine. Multiple open source WS-BPEL engines exists which can be used as starting point. We leave the actual implementation of such an engine for future work.

The contract for a component containing a single IT service, i.e. the SCA component configuration, can again be derived from the WS-BPEL process. The services and references defined in the contract can be derived from the invoke, receive, and reply activities in the WS-BPEL process as described in section 8.2.

8.5 Implementing data services

Data service implementations only differ from IT service implementations in the kind of activities they execute. While data services support D-transactions, production acts supported by data services are datalogical in nature, meaning that they store, transmit, copy, and destroy data (see the distinction axiom in section 3.3). Hence, to define the implementation model of data services we need a language able to describe these kinds of activities. As with IT services we can extend WS-BPEL. However, instead of the validate, calculate, and change activity used for the IT service implementation model we need the following activities in addition to the WS-BPEL activities:

- *Validate*: used to validate the format of input based on a set of validation rules. The definition of this activity contains a list of data attributes to check and a validation rule for each attribute (e.g. a regular expression).
- *Create*: used to create objects. The definition of this activity only contains the type of information object to create.
- *Retrieve*: used to read objects from data sources. The definition of this activity contains a query, i.e. the language string used to query data sources for the requested information.
- *Store*: used to store changed objects. The definition of this activity contains an input object, which is the object to store.
- *Delete*: used to delete objects. The definition of this activity contains an input object, which is the object to delete.

With these activities in addition to the activities already defined in WS-BPEL we have a language to define the implementation model of data services. However, we also need information about the data structure for actually creating, retrieving, storing, and deleting data. The implementation of a data service can execute requests on a database for manipulating data, but that database need to be configured with a database schema defining the structure of the data.

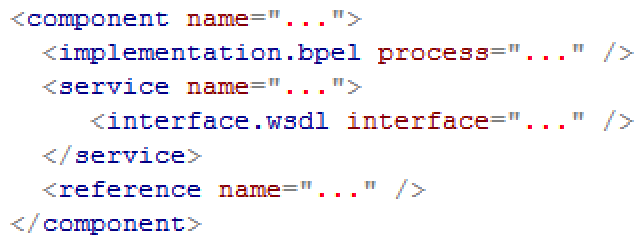
In the organization model the state model (modeling the information objects, their attributes, and their relations) has been defined using a language based on ORM [45] (see section 5.1). As Halpin and Proper show in [46] ORM can directly be used to define optimized database structures. Hence, we do not need to define additional information in the data service implementation model, we can directly derive the database schema from the state model. As shown by Vermolen and Visser [97] it is even possible to handle data structure changes over time while preserving the data in the database. The data stored in the database according to the database schema can be evolved automatically following the changes in the model.

Based on the previous we see that we need to define a custom engine to execute the data service implementation model. As with IT services this engine can be a specialization of a WS-BPEL engine. In addition to this engine we also need a Database Management System (DBMS) to create, retrieve, store, and delete data. We leave the implementation of the custom engine for future work. We do not need to implement a DBMS while there exist multiple DBMS products which are used a lot and in practice mostly already available within organizations.

In section 8.9 we will show an example data service implementation model. In section 8.7 we will explain in more detail how to derive a database schema from the state model.

8.6 Defining the SCA configuration

In the previous sections we have seen the different elements of the business component implementation model. We also briefly described an SCA component configuration, assuming we had one service or service orchestration per component. In this section we will first define the SCA component configuration in detail including its XML syntax. Afterwards we will explain that we need to define SCA composite configurations to define a component implementation for a component providing multiple services.



```
<component name="...">
  <implementation.bpel process="..." />
  <service name="...">
     <interface.wsdl interface="..." />
  </service>
  <reference name="..." />
</component>
```

Figure 8.6: SCA component configuration template

Figure 8.6 exhibits an example SCA component XML configuration template. The root element defines the component which is identified with an unique name. The implementation element defines the type of implementation for the component. In our case we always use the WS-BPEL implementation type as defined in the SCA Client and Implementation Model Specification for WS-BPEL [80]. The specification defines that the implementation element should contain a process attribute containing the name of the process used as implementation. As all implementations described in the previous sections (i.e. a service orchestration, human service, IT service, and data service) are based on WS-BPEL each of them can be described with the SCA BPEL implementation type. The SCA BPEL specification also describes in detail how to derive the service and reference elements from the WS-BPEL process. We have described this process before in section 8.2 and Figure 8.3.

The service element in the XML configuration contains a name, which can be derived from the business component construction model (i.e. the name of the provided service of the component). The service element contains an interface element which points to the definition of the interfaces specified in the Web Service Description Language (WSDL) [23]. Although multiple service elements can be defined, only one is needed because each component only represents a single WS-BPEL process implementing a service orchestration, human service, IT service, or data service.

Each service invoked from a service orchestration or from the implementation of a service will lead to a reference in the component configuration. As shown in Figure 8.6 the reference element contains a name attribute which points to the name of another service. For internal services, i.e. services in the same SCA domain, no binding information (i.e. location and protocol details) needs to be specified, the runtime ensures that services can find and communicate with each other. For services used by external parties the binding information can be specified in the runtime environment.

Until now we assumed that each component contained a single service orchestration or a single service. However, in the previous chapter we have seen that the business component construction model can contain components providing multiple services. Hence, we need to define the SCA configuration for such components.

```
<component name="component1">
  <implementation.composite name="composite1" />
  <service name="s1">
     <interface.wsdl interface="..." />
  </service>
  <service name="s6">
     <interface.wsdl interface="..." />
  </service>
  <reference name="s3" />
  <reference name="s4" />
  <reference name="s5" />
</component>
```

Figure 8.7: SCA component with composite as implementation

The default way in SCA to combine multiple implementations in one component is to use an SCA composite as component implementation. An SCA composite is used to assemble SCA elements in logical groupings. It is the basic unit of composition within an SCA Domain [79]. An SCA composite contains a set of components, services, references and the wires that interconnect them, plus a set of properties which can be used to configure components. Figure 8.7 exhibits an example component configuration using an SCA composite as implementation. The component, named component1, uses composite1 as implementation and it provides the services s1 and s6. Its implementation depends on the services s3, s4 and s5, which are defined as references. We did not specify the full WSDL definitions of the service interfaces because they do not add anything to our example.

```
<composite name="composite1">

  <service name="s1" promote="c1/s1" />
  <service name="s6" promote="c3/s6" />
  <reference name="s3" promote="c1/s3" />
  <reference name="s4" promote="c2/s4" />
  <reference name="s5" promote="c3/s5" />

  <component name="c1">
    <implementation.bpel process="process1" />
    <service name="s1">
      <interface.wsdl interface="..." />
    </service>
    <reference name="s2" />
    <reference name="s3" />
  </component>

  <component name="c2">
    <implementation.bpel process="process2" />
    <service name="s2">
      <interface.wsdl interface="..." />
    </service>
    <reference name="s4" />
  </component>

  <component name="c3">
    <implementation.bpel process="process3" />
    <service name="s6">
      <interface.wsdl interface="..." />
    </service>
    <reference name="s5" />
  </component>

</composite>
```

Figure 8.8: Example SCA composite configuration

Figure 8.8 exhibits the example configuration for composite1. It groups three components (c1, c2, and c3) each providing a single service and each implemented by a WS-BPEL process. As shown a composite also has services and references just as components. The services provided by a composite are promoted component services. For example, service s1 is the service s1 from component c1 promoted to the composite level. The same holds for references. In this way a subset of services and references of the included components will become external. As shown in the example in Figure 8.8 not all services and references are promoted. Component c1 has a reference referring to service s2 provided by component c2. So, components grouped within a composite can refer to services provided by other components or to services provided by another composite. In the latter case the reference needs to be promoted and becomes a reference of the composite.

An SCA composite is not necessarily used to implement an SCA component. Instead of using an SCA composite to define the implementation of a higher-level component it can also be directly deployed on an SCA framework.

8.7 Engineering

Now the full business component implementation model has been defined we will describe the engineering step of our MDEE approach, which defines how to derive the business component implementation model from the business component construction model.

In section 8.2 we have shown how a single service orchestration can be implemented using an SCA component with a WS-BPEL process as implementation. In section 8.6 we have seen that SCA components can be grouped in SCA composites, thereby defining the implementation of a higher level component. Although we will now assume that all service orchestrations defined in the business component construction model will be grouped in one component, they can also be grouped in multiple components based on specific implementation needs.

Based on the previous we can formulate the engineering step of our MDEE approach for the coordination part of our model as follows:

- Transform each service orchestration in the business component construction model into a WS-BPEL process. Sequential activities will become a sequence activity in WS-BPEL, activities defined in parallel will become part of a WS-BPEL flow activity. Each activity defined in the service orchestration will become a WS-BPEL invoke activity, each decision will become an if activity. Additional information needs to be added to the WS-BPEL process such as the precise definition of inputs and outputs of invoke activities and how the output of one activity can be used as the input for another activity.

- Generate an SCA component configuration for each WS-BPEL process. This configuration will contain one service for calling the process and a reference for each invoke activity in the process. We have described this in more detail in section 8.2.

- Generate an SCA composite configuration containing all generated SCA component configurations. For each component configuration the service elements are added as promoted service to the SCA composite, each reference element is added as promoted reference to the SCA composite.

The resulting SCA composite represents the orchestration component and can directly be deployed on an SCA framework, i.e. it is directly executable.

The components defined in the business component construction model also need to be translated to deployable SCA composites. In the sections 8.3, 8.4, and 8.5 we have seen the implementation model and SCA component configuration for human, IT, and data services respectively. We have defined these implementations assuming that each component only contained a single service. In the business component construction model, however, a component can contain multiple services. Therefore we need to group the SCA components representing a single human, IT, or data service into SCA composites.

Based on the previous we can formulate the remaining part of the engineering step of our MDEE approach as follows.

For each component in the business component construction model:

- Create an SCA composite.
- Create an SCA component within the composite for each provided service by the component.
- Define the SCA configuration and implementation of that SCA component based on the type of service. We will describe this process in detail for each service implementation type in the remainder of this section.
- Promote each SCA component service to a service of the SCA composite.
- Promote each reference which does not refer to a service provided by an SCA component available within the composite to a reference of the SCA composite. If an SCA composite, for example, contains an IT service which implementation uses a data service which is part of the same SCA composite, than this reference can stay internal, i.e. it does not have to be part of the interface of the SCA composite. References to services provided by other SCA composites need to be part of the interface, i.e. they have to be promoted to a reference of the composite itself.

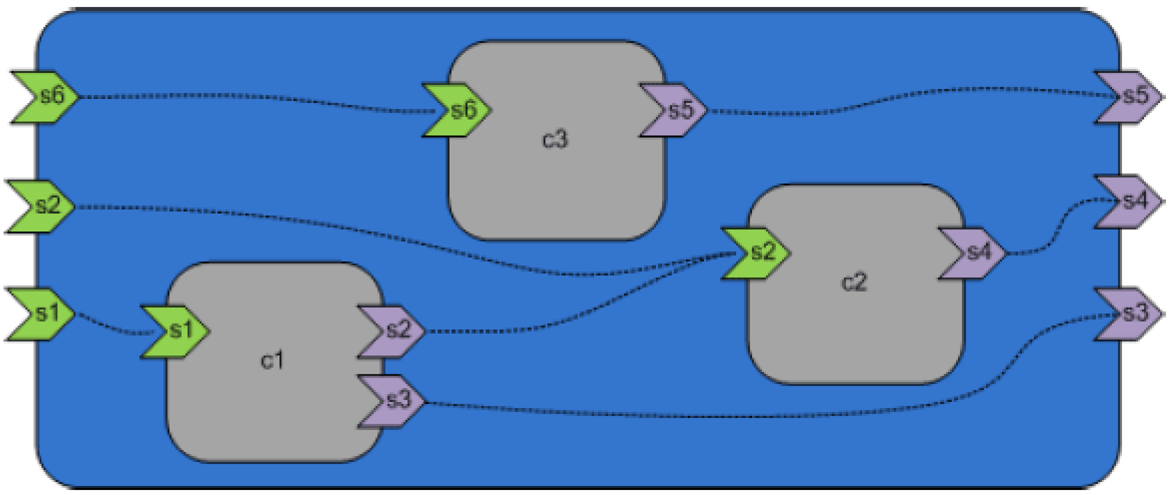


Figure 8.9: SCA composite example visualized

Let's consider an example component providing three services: s1, s2, and s6. Service s1 is an IT service which implementation invokes service s2 and service s3. Service s2 is a data service which implementation invokes service s4. Service s6 is a human service which implementation invokes service s5. Following the engineering step described above we come to the SCA composite visualized in Figure 8.9.

As exhibited the SCA composite contains three SCA components (c1, c2, and c3) each providing one service. All these services, s1, s2, and s6, are promoted to a service of the SCA composite. References s4, s5, and s6 are examples of promoted references. The wire between reference s2 of component c1 and service s2 of component c2 is an example of an internal reference. This means reference s2 of component c1 does not have to be promoted to a reference of the SCA composite.

So, following the steps listed above the full business component implementation model can be specified. Each resulting SCA composite can directly be deployed on an SCA framework and as soon as its references are available its provided services can be called. In the remainder of this section we will define in more detail how to derive the implementation model for each service implementation type. In section 8.9 we will provide an example implementation model for each service implementation type.

For human service we need to create a WS-BPEL process including a people activity invoking a Human Task (as described in section 8.3). Besides the people activity, the WS-BPEL process contains an invoke activity for each service which needs to be called to retrieve the information needed as input for the Human Task. The process also contains an invoke activity for each service needed to store the outcome of the Human Task. What services need to be invoked can be derived from the shaping relations in the organization model, i.e. each service supporting a transaction requested via shaping by the executor of the transaction supported by this human service need to be invoked by the implementation of this service. The SCA component configuration for a human service implementation contains an implementation element referring to the WS-BPEL process, one service representing the human service, and a reference for each service invoked by the WS-BPEL process.

The Human Task definition itself can be derived from the previous models as follows:

- name: the name of the service as specified in the service specification model.
- interface: WSDL definition of the IT interface of the task. Can be the same for each human service depending on the used engine.
- peopleAssignments: the actor role of the service as specified in the service specification model.
- presentationElements: can be a manually defined interface or automatically generated based on the outcome. In the latter case standardized interface elements are used for each possible attribute type of the information object defined as outcome.
- outcome: the production fact of the service as specified in the service specification model.
- deadlines: can be derived from the quality attribute of the service as specified in the service specification model.

We can summarize the engineering step for a human service with the following steps:

- Create a WS-BPEL process with one people activity calling a Human Task.
- Define the Human Task according to the WS-HumanTask standard. Derive each element from the other models in our MDEE approach as described above.
- Add invoke activities to the process for each service invoked to retrieve the data needed as input for the Human Task.
- Add invoke activities to the process for each service invoked to store the outcome of the Human Task.
- Create an SCA component configuration representing the human service.

In section 8.4 we have described the implementation of an IT service. We defined a custom language based on WS-BPEL to describe the implementation model. This language contains three activities in addition to WS-BPEL: validate, calculate, and change. The complete implementation model for an IT service consists of a process defined using this language and an SCA component configuration. The engineering step for an IT service can be defined as follows:

- Create a process using the custom language.
- Add an invoke activity for each service invoked by the IT service. This can be derived from the shaping relations in the organization model in the same way as described previously for human services.
- Model how the production fact is created using the validate, calculate, and change activities. These activities need to be placed before or after invoke activities depending on the needed input and output data of each activity. For example, a calculate activity needs data as input and hence needs to follow an invoke activity providing this data as output. The calculate activity can be followed by an invoke activity with the result of the calculation as input.
- Create an SCA component configuration representing the IT service. Its implementation element will refer to the process, its service element describes the provided IT service, and it will contain a reference for each invoked service.

For data services we have also defined a custom language based on WS-BPEL. Hence, the engineering step should describe how to derive a process defined in this language from the previous models in our MDEE approach. As explained in section 8.5 we also need to have a database to store and retrieve data to make the implementation of data services executable. The structure of this database can be based on the state model in the organization model. For now we assume that we transform the state model in a single database structure which is used by all data services. It is also possible to use multiple databases on different locations. Goethals [44] describes and evaluates data storage options in a service-oriented environment (i.e. where the data is stored and who stores it) in more detail. The combination of his work and our MDEE approach is left for future work.

The database structure can be derived from the state model. McCormack, Halpin, and Ritson [64] have developed and described an approach to transform ORM models to a relational database schema. Basically the following steps are needed (translated to the ORM based language used to define the state model):

- Create a table for each information object and result type.
- Create a column for each attribute of the information object.
- Each relation transforms to a crosstable or foreign key (column with identifier to the row of another table depending on the type of relation).

If the database is generated, the engineering step for a data service can be defined as follows:

- Create a process using the custom language.
- Model how the production fact is created using the validate, create, store, retrieve, and delete activities. For example, the input of a data service is an information object. The first activity validates if the format of the input data is correct. If the input data is correct then a second activity stores the information object in the database.
- Create an SCA component configuration representing the data service. Its implementation element will refer to the process, its service element describes the provided data service. The reference to a database is not part of the SCA standard and needs to be configured in the SCA framework in a vendor-specific way.

In this section we have seen how we can create an executable implementation model based on the previous models in our MDEE approach. We first described how to create an SCA composite representing the orchestration component. Afterwards we explained how to create an SCA composite for each component defined in the business component construction model.

8.8 Implementation

In the previous sections we described the business component implementation model and how to derive it from the previous models in our MDEE approach. The business component implementation model is specified using graphical modeling languages. In this section we will explain how the business component implementation model can be executed, i.e. how to implement this model on engines.

While the business component implementation model consists of SCA composites, we need an engine executing these composites. As explained in section 8.1 we call such an engine a component framework. Several component frameworks implementing the SCA standard exist, among them the open source Tuscany project[1].

For SCA frameworks the basic unit of deployment is a so-called SCA contribution. This is an archive file containing all information needed to run an SCA composite. It contains the configuration for the SCA composite and the configuration for each SCA component. These configurations are defined in XML files as prescribed by the SCA

[1]http://tuscany.apache.org/

standard. An SCA contribution also contains the implementation of the SCA components. An SCA framework can configure and execute these implementations using the SCA configuration files. However, the framework should support the provided type of implementation. If the implementation, for example, is defined in Java, the framework should be able to execute Java class files. If the implementation is defined in WS-BPEL the framework should contain a WS-BPEL engine.

As we explained in the previous sections we use custom-defined languages to specify the implementation of SCA components. This gives us the ability to derive part of the implementation from the previously defined models in our MDEE approach and the remaining part can be specified using a high-level graphical language. The downside is that we need to provide custom engines to execute these implementation models. We can provide these engines by extending the SCA framework or by including the engine in the SCA contribution.

The SCA aims to be interoperable among all kind of implementation types. That is why almost all SCA frameworks provide extension mechanisms to provide engines for custom implementation types. However, each SCA framework has its own way to enable the addition of extensions. This means that we do not target a standardized framework anymore, we need to provide extension implementations for each framework we want to deploy SCA composites generated by our MDEE approach on.

The second way to provide custom engines, including the engine in the SCA contribution, is much more flexible. The basic idea is to define the implementation of an SCA component as a Java implementation, because that is the default implementation type and thus supported by each SCA framework. This Java implementation is just a small wrapper which starts the custom engine and let it execute the implementation model. Each service call is redirected to the engine instead of handled by Java code. The advantage of this approach is that the SCA contributions generated by our MDEE approach can be executed on each SCA framework. Another advantage is that each contribution can contain another version of the engine. In practice this will be necessary because the custom language, and thus the custom engine, will evolve over time. However, this is also a drawback of this approach. If the engine need to be updated each SCA contribution need to be updated instead of one extension of the SCA framework.

Which of these two approaches to use depends on multiple factors, a lot of them non-technical. While we leave the actual implementation of our MDEE approach as future work, we also leave this question open for future research.

Concluding we can say that the implementation step of our MDEE approach consists of:

- Transforming all configurations and models into a XML files which can be interpreted by the engines.
- The bundling of these files in SCA contributions.
- Deploying the SCA contributions on an SCA framework.

The following engines are needed to execute the result of our MDEE approach:

- An SCA framework for deploying the SCA contributions and the wiring of all SCA composites and SCA components.

- A WS-BPEL engine for executing the service orchestrations.
- A WS-BPEL engine with WS-HumanTask extension for executing human service implementations.
- A WS-BPEL engine with an extension defining a validate, calculate, and change activity for executing IT service implementations.
- A WS-BPEL engine with an extension defining a validate, create, retrieve, store, and delete activity for executing data service implementations.
- A DBMS configured with the generated database schema to create, retrieve, store, and delete data.

8.9 Example implementation model

We will conclude this chapter by showing some example parts of the business component implementation model. We will show an example implementation model for a service orchestration, a human service, an IT service, and a data service. We will also show an example SCA composite implementing a component described in the business component construction model. We have drawn the WS-BPEL processes for the service implementation models with a trial version of ActiveVOS Designer[2].

In section 7.3 we have shown an example service orchestration for the policy binding process in Figure 7.6. Figure 8.10 exhibits the implementation model expressed in WS-BPEL for this service orchestration. We created this implementation model following the engineering step described in section 8.7. We described the service orchestration in detail in section 7.3. The only difference worth mentioning is that instead of the start and end activity used in the UML activity diagram, the WS-BPEL contains a receive activity waiting for the incoming message triggering the process and a reply activity sending a message back.

[2]www.activevos.com

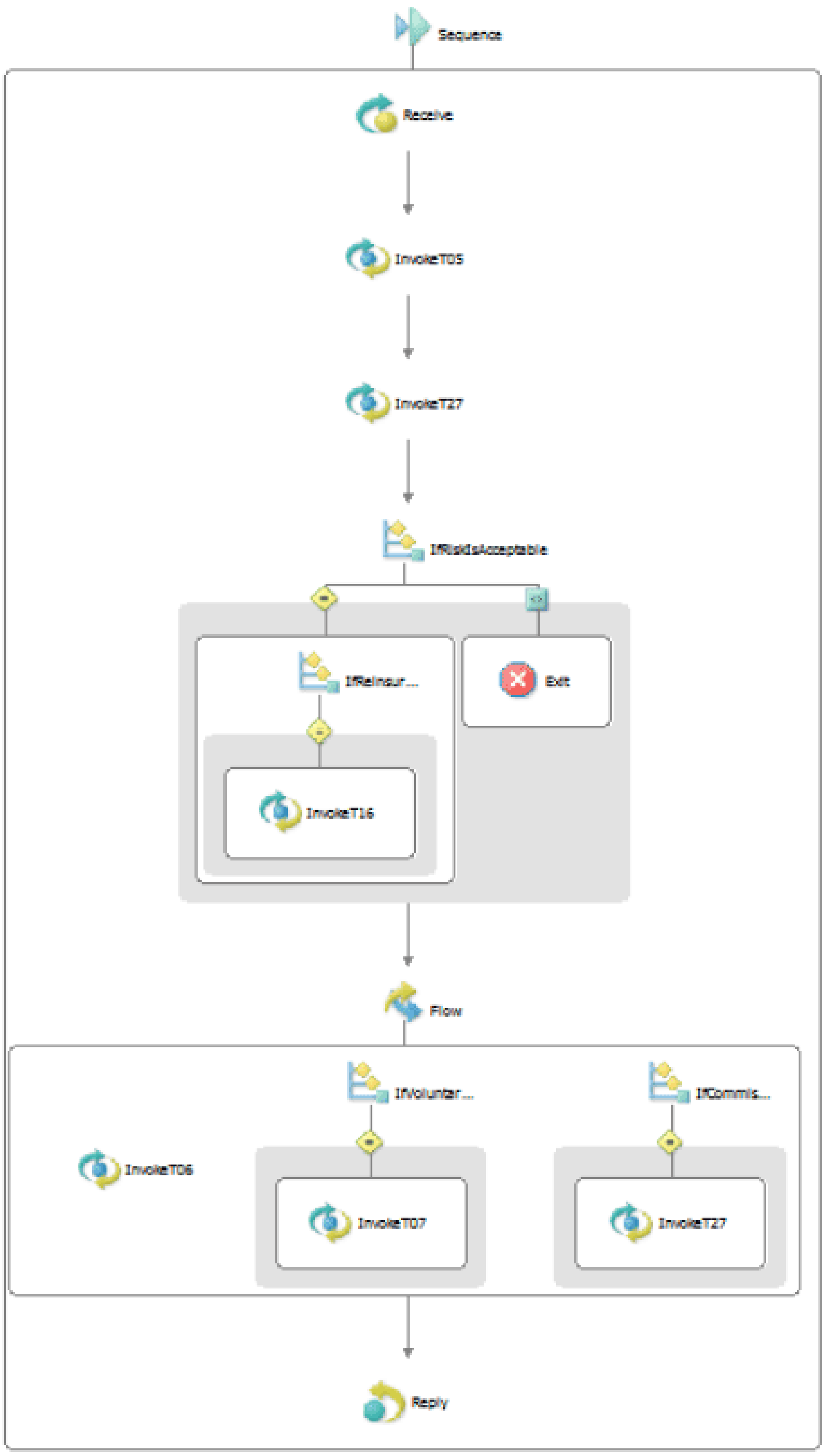


Figure 8.10: Implementation model of the policy binding service orchestration

Figure 8.11 exhibits an example implementation of human service T01 (ProductAdvicing). All activities in the WS-BPEL process are placed in a sequence activity, denoting that they are executed in sequential order. The first activity, a receive, waits for incoming messages. Once T01 is triggered the receive activity receives the incoming message and the engine executes the next activity, an invoke activity which invokes the IT interface of human service T30 to retrieve the personal details from the potential individual policy holder. A task will be created for the associated actor role and once the task is completed the result will be returned to the invoking party (in this case the implementation of T01). After that the process implementing service T01 can proceed with invoking IT service T31 which returns the needed product information to create a product advice. Once this information is received the people activity CreateProductAdvice is executed which create the CreateProductAdvice task which is described with the following WS-HumanTask elements:

- *name*: CreateProductAdvice.
- *interface*: WSDL interface with portType 'createAdvice' and operation 'advice'. The input message for this operation contains a PERSON object containing the personal details of the potential individual policy holder and a PRODUCTCOMPOSITION object containing the relevant information of the relevant products.
- *peopleAssignments*: product advisor (A01).
- *presentationElements*: graphical user interface containing a task description for the product adviser including the personal details and product information.
- *outcome*: product advice is created (R01).
- *deadlines*: the start deadline is five minutes, the completion deadline two days.

Note that we did not provide the full WS-HumanTask XML specification for the CreateProductAdvice task. We described the mandatory elements of the task in an informal way to explain them. A complete executable specification of the implementation model and testing it on appropriate engines is left for future work.

After the CreateProductAdvice task is completed the people activity is finished. The next activity in the process exhibited in Figure 8.11 invokes data service T60 to store the outcome of task. The process finishes with a reply activity which sends a message back to the sender of the incoming message received by the receive activity.

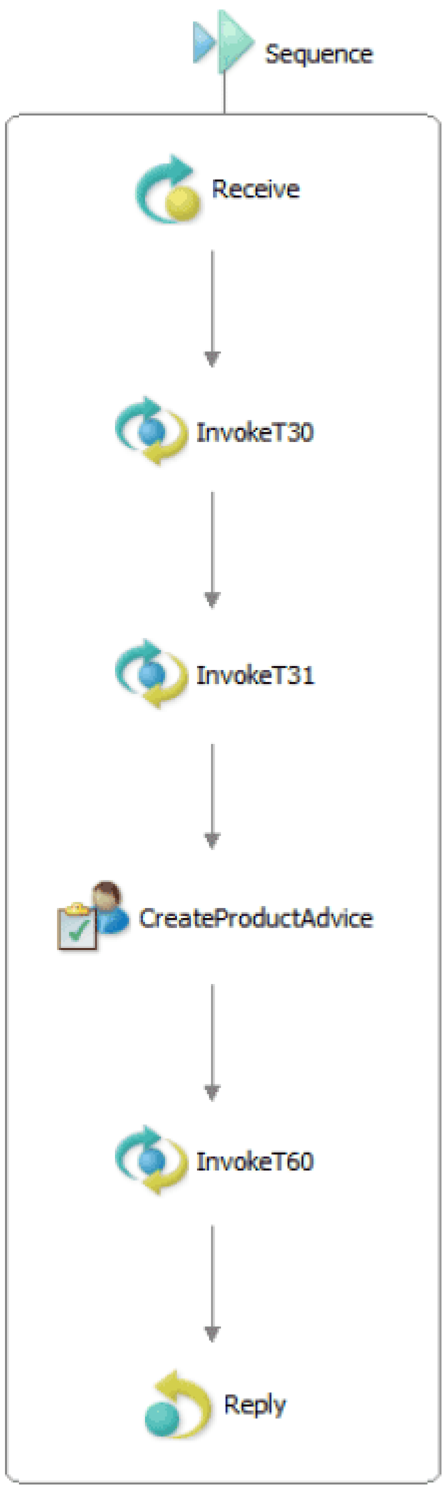


Figure 8.11: Implementation model of human service T01

Figure 8.12 exhibits the implementation of IT service T31 (ComposeProductInformation). The implementation process starts with a receive activity waiting for an incoming message. This message should contain a PERSON object containing the personal details of the potential individual policy holder. While service T31 has a composite-part relationship with service T61 (as we can derive from the shaping relations in Figure 5.4) its implementation contains an invoke activity invoking service T61. This service returns the relevant products based on the given personal details.

The next activity in the process, the for each activity, takes the resulting list of products as input and executes the activities in the scope it contains for each product. A scope provides the context which influences the execution behavior of its enclosed activities. So, for each product the CalculateProductInformation activity will be executed. This activity is a calculate activity (see section 8.4) and calculates the price of a product based on the personal details of the potential individual policy holder. This activity is configured with a mathematical expression like for example:

$$product.policy.insurancePremium.price + product.policy.commission.price + \frac{100}{person.age}$$

We just created a mathematical expression as example, this is not a realistic way to calculate the costs of a certain insurance product for a given potential individual policy holder. As exhibited we add the price of the INSURANCEPREMIUM associated with the POLICY associated with the PRODUCT to the price of the COMMISSION associated to the POLICY associated to the PRODUCT. Furthermore we add the result of 100 divided with the age of the given PERSON. Once the price for each product has been calculated the service returns the results to the invoking party with the reply activity.

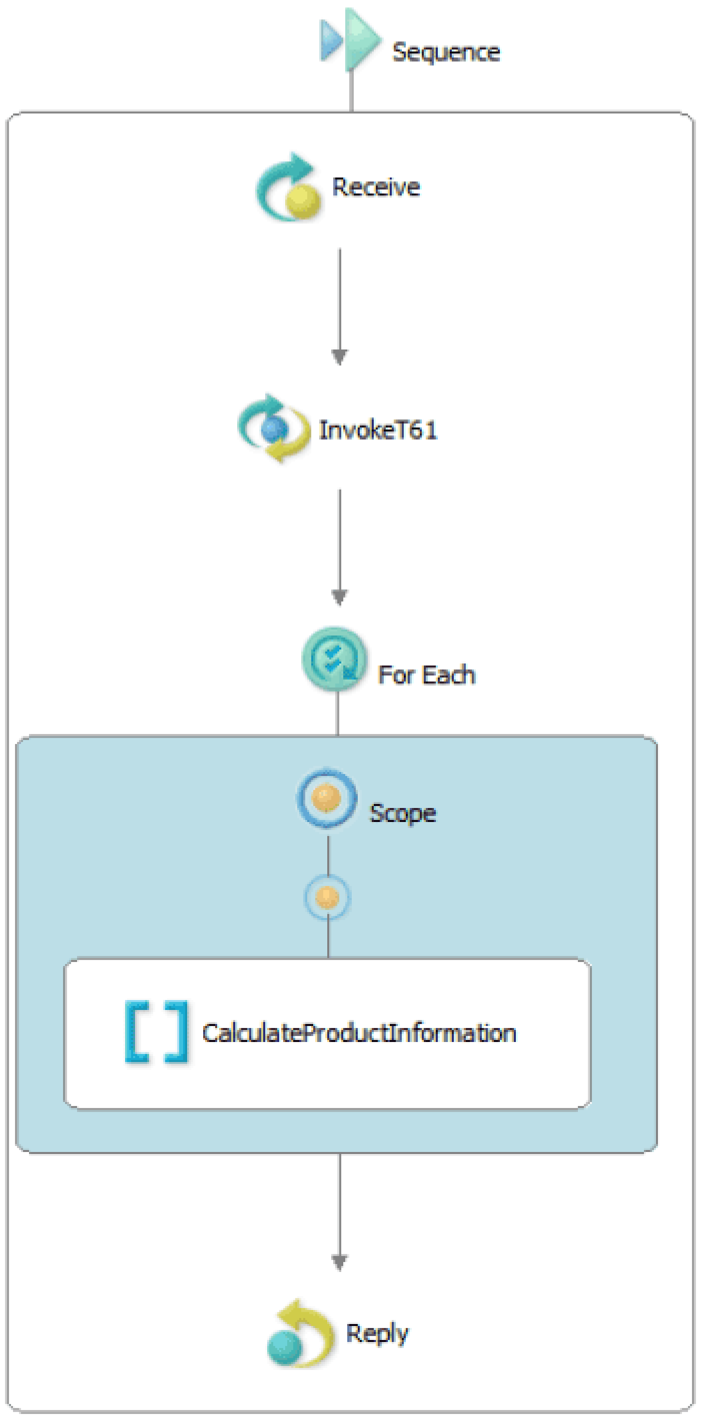


Figure 8.12: Implementation model of IT service T31

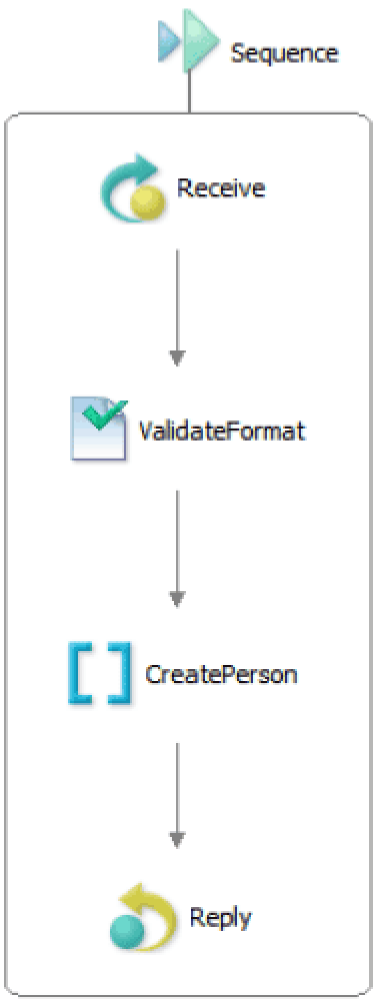


Figure 8.13: Implementation model of data service T61

Figure 8.13 exhibits the implementation of data service T61 (RecordPersonalDetails). As shown the process waits for an incoming message using the receive activity. This message should contain a PERSON object containing the personal information of the potential individual policy holder. The ValidateFormat activity is a validate activity (see section 8.5) and it validates the format of the incoming PERSON object. This activity is configured with, for example, the following validation rules:

- Attribute age should be greater than 0.
- Attribute name should apply to the following regular expression: [a-zA-Z]+

This list can of course be extended with all kinds of validation rules each attribute of the PERSON object should comply to. If the format is valid the CreatePerson activity executes a query on the database to insert a new PERSON object, i.e. it will execute an SQL insert query (depending on the type of database) to insert a record in the PERSON table. So, CreatePerson is a create activity (see section 8.5) configured with a specific query.

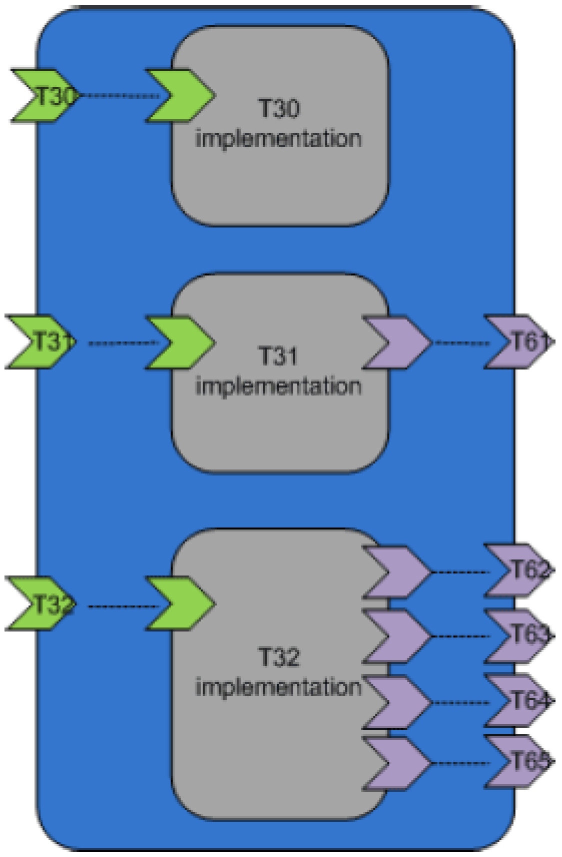


Figure 8.14: Example component implementation

We have shown an example implementation model for a service orchestration, a human service, an IT service, and a data service. We will finish this section by showing an example SCA composite implementing the I-component of the business component construction model for scenario 2, shown in Figure 7.11. Figure 8.14 exhibits a visual representation of this composite, which we created following the engineering step of our MDEE approach as described in section 8.7. As exhibited the component provides three services (T30, T31, and T32) and has references to five services (T61, T62, T63, T64, and T65). Each provided service is implemented with an SCA component, as explained in detail in this chapter. T30 is a human service, T31 and T32 are IT services. We have shown example implementation models for both service types previously in this section. The SCA composite shown in Figure 8.14 can be implemented on technology following the implementation step of our MDEE approach described in section 8.8. This will result in a working IT system directly supporting the organization (as described by the organization model) with its provided services.

Chapter 9

Conclusions

We started this research with a set of requirements for an MDEE approach. Our goal was to design an MDEE approach fulfilling these requirements. The DEMO methodology provided us with a formal ontological organization model and a description of the reverse engineering step to start the design of the MDEE approach with. We have described a complete MDEE approach turning such an ontological organization model into a working IT system with use of multiple steps and models in between. In this Chapter we will explain what we contributed (9.1) and we will draw some conclusions (9.2). We will also evaluate our research (9.3) and give an overview of what we see as future work (9.4).

9.1 Contributions

This report contributes by giving a formal definition of MDE. We also applied MDE to Enterprise Engineering, thereby defining an MDEE approach. This MDEE approach is founded on the theory of enterprise ontology and is the first attempt of combining the theory of enterprise ontology and MDE. The presented MDEE approach is the first formal end-to-end approach describing how to model an organization and how to transform and refine that model until an executable model of an IT system has been defined.

The DEMO approach focuses on the ontological model of an organization for the purpose of (re)designing the essence of an organization. We extended DEMO by providing a well-founded model-driven approach to implementing an ontological model of an organization. We did this by combining the work on enterprise ontology and DEMO [34, 31], the work on modeling the I- and D-organization [26], the work on enterprise ontology based service specification [91, 93, 92], the work on BCI-3D [5, 4, 6], and a lot of theory on MDE and DSL approaches.

The organization model could be partly based on DEMO. We described this model (and all other models) from an MDE perspective, i.e. we described it as a multi-model. While only a first research paper on the integration aspects between the B-, I-, and D-organization [26] was available during our research, we had to define what aspect models we needed to model the I- and D-organization, how these models are expressed and how they are related to each other and the aspect models of the B-organization.

For the service specification model we could fully rely on the work of Albani and

Terlouw [91, 93, 92]. We contributed by describing the service specification model as a multi-model and fitting it into our MDEE approach. We also contributed an example service specification for each service type based on a real-life case.

The business component construction model has also been defined based on enterprise ontology. We contributed by defining this model including different types of components and service orchestrations. For the component identification step we used BCI-3D, which we slightly adapted to take the previously defined services into account. The other parts of the construction design step, service orchestration design and component specification, have been defined by ourselves.

The last part of the MDEE approach, the business component implementation model and the engineering and implementation step, is an important contribution in the field of MDE. We have shown that it is possible to implement components and service orchestrations based on high-level models derived from the previously defined models in the MDEE approach. The implementation fully complies to the principles of SOA and provides autonomous components which are easy to scale and reuse.

Concluding we can state that we contributed by describing a formal and complete MDEE approach by combining existing research, filling the gaps in this research, and extending this research with our own research, especially in the field of MDE. We also contributed by providing a full example following our MDEE approach based on a real-life case.

9.2 Conclusions

We started this research with the following main research goal:

> *Design an MDEE approach based on a sound theoretical foundation, providing end-to-end guidance to refine and transform an organization model into an IT system supporting that organization.*

In order to achieve the research goal, a number of research questions have been formulated. The first question, *what are the requirements for an MDEE approach*, has been answered in Chapter 4. We have defined MDE formally based on the GSDP. We also listed the requirements for an MDEE approach based on a short actor analysis. The requirements for an MDEE approach are:

- an MDEE approach should consist of five main steps: reverse engineering, function design, construction design, engineering, and implementation.
- In each step a formal model will be produced, at least the following models are needed: an organization model, a service specification model, a business component construction model, and a business component implementation model.
- Each model in this MDEE approach should comply to the following requirements:
    - It should be defined in a formal modeling language. For each language the abstract syntax, concrete syntax, and semantics need to be specified.

- To accurately abstract large and complex systems it needs to be a multi-model. For each multi-model the dimensions and aspect models need to be specified.
- It needs to be derived from the previous models in the MDEE approach with mostly automated transformations.

The second question, *how can enterprise ontology provide a sound theoretical foundation for an MDEE approach*, has been answered by describing enterprise ontology in detail in Chapter 3. Since enterprise ontology provides a very sound theoretical foundation for a layered view on organizations, its supporting IT systems, and how to distinguish these layers, this theory provides us with a formal foundation for an MDEE approach providing end-to-end guidance to refine and transform an organization model into an IT system supporting that organization.

The third question, *how can an MDEE approach be designed with use of enterprise ontology*, has been answered in the Chapters 5, 6, 7, and 8. In these chapters we researched and defined the steps and models of our MDEE approach. Figure 9.1 exhibits an overview of the resulting MDEE approach.

The first step in our MDEE approach starts with interviews and the documentation of an organization and results in an organization model. Using service identification a set of services is identified which are needed to support this organization. These services are specified in the service specification step, resulting in a service specification model. Service identification and service specification together form the function design step of our MDEE approach.

The next step in our MDEE approach is construction design. As shown in Figure 9.1 this steps consists of two parallel tracks. Service orchestrations are derived from the previous models based on a set of derivation rules. Another set of derivation rules defines how to derive BCI3D input tables from the previous models. The component identification step, implemented with BCI3D, identifies a set of components based on these tables. The identified components are specified in the component specification step. The model created in the construction design step is a business component construction model and consists of service orchestrations and a component diagram.

In the engineering step the business component construction model is transformed and refined into a business component implementation model. As exhibited in Figure 9.1 this is done by defining an implementation for the orchestrations, human services, IT services, and data services. These implementations are combined into SCA composites which form the business component implementation model. This final model can be implemented by executing it on engines, resulting in a working IT system.

As enterprise ontology is a way of thinking, different modelers create the same model for the same organization. This means our MDEE approach has a stable basis. While our MDEE approach is also model-driven and consists of steps which are partly automatable the quality of the resulting IT system can be very high depending on the quality of the MDEE approach itself. In other words: as the IT system is derived automatically (for a big part) from the organization model and the organization model is stable because of the underlying theory, the quality of the IT system can be ensured by improving the quality of the MDEE approach itself.

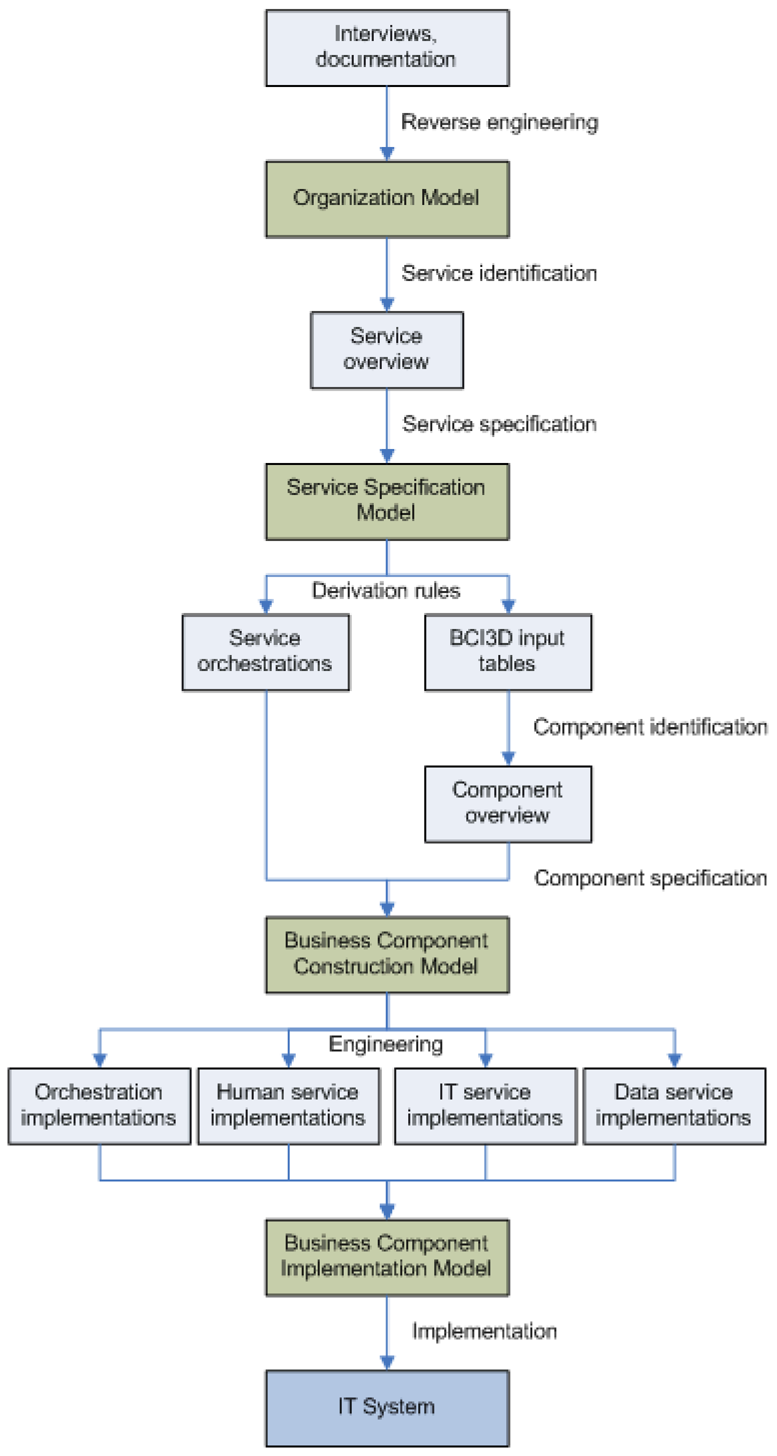


Figure 9.1: Overview of the MDEE approach

In our definition of MDE the automation of transformations is an important aspect. Hence, the question is to what extend the steps in our MDEE approach are automatable. The steps in our MDEE approach consist of two parts: a transformation part and a refinement part. The transformation part is defined with derivation rules and specifies what elements of the output model can be directly derived from the input model. This part can be fully automated with model transformations. The refinement part describes what additional elements have to be added to produce the output model. In this part decisions have to be made about what elements to add and how they are modeled. These decisions can be 'hard-coded' in transformations, thereby also automating this part of an MDEE step. However, a part of the decisions will always stay human decisions. The more an MDEE approach is tailored to a specific domain (e.g. the approach is only applicable to insurance companies), the more decisions can be taken beforehand and thus 'hard-coded' in automated transformations.

Because a significant part of the steps in our MDEE approach can be automated and the implementation model (which is directly executable) is defined in a much higher level language than currently existing programming languages, we can conclude that our approach fully complies to the definition of MDE. As our MDEE approach also complies to the requirements listed before it will fulfill the needs of its actors. Our MDEE approach does raise the level of abstraction in program specification and it increases the automation in program development. Hence, our MDEE approach can finally deliver the increase in productivity we are waiting for quite some years.

Concluding we can state that we succeeded in designing an MDEE approach based on a sound theoretical foundation, providing end-to-end guidance to refine and transform an organization model into an IT system supporting that organization. As we have shown with our running example, starring the Protector case, our MDEE approach is applicable to real-life cases. However, due to the limited scope of our research there are some research limitations. We will evaluate our research including its limitations in the next section.

9.3 Evaluation

The research goal we started our research with, *design an MDEE approach based on a sound theoretical foundation, providing end-to-end guidance to refine and transform an organization model into an IT system supporting that organization*, was quite ambitious. We have succeeded in this ambition, but due to the limited scope of our research and the broad subject we could not specify every aspect of our MDEE approach in detail. Hence, our research was limited on the following points:

- The MDEE approach is a top-down approach. Although it is necessary in practice to combine a top-down approach with a bottom-up approach to incorporate existing IT systems in the resulting implementation, we have left this as future work. We did, however, define two possible scenario's for component identification in Chapter 7, which can be used as a starting point for incorporating existing systems into our MDEE approach. With use of the right scenario (or a mix of scenario's) existing systems can also be modeled as components.

- To specify the needed models we have used existing modeling languages as much as possible. If no modeling language existed fulfilling our needs we did define a custom language (e.g. by specializing an existing language). We only defined the basic elements of the language, we did not provide a full formal language definition.

- The end result of our MDEE approach is an executable implementation model. We did define this model and explain how to execute it. However, we did not specify a full example including all needed details to execute it. We also did not implement the needed technology to execute the implementation model. We have only shown that it is possible to directly execute the implementation model and we gave some first ideas about implementing the needed technology to do so.

Our MDEE approach combines a lot of research subjects, some of them still work in progress, which has made our work more difficult. Both the research in modeling the I- and D-organization and enterprise ontology based service specification are still running. The first paper touching the subject of modeling the I- and D-organization [26] was published during the time we worked on our research. The same holds for the paper describing the enterprise ontology based service specification framework [92]. We did take this research into account, but future changes in these works, some of them already available when finishing our research, need to be incorporated in our MDEE approach accordingly.

As a lot of information is derived from the organization model it is of utmost importance that this model is correct and stable. For the DEMO models of the B-organization this has been proved. More work on the I- and D-level is needed to prove that these models are also stable like the models of the B-organization.

In the organization model as defined as part of our MDEE approach, the models of the I- and D-organization are presented in a combined model to show the shaping relations between the actors. However, while transactions in the I-organization can be (re)used by multiple B-actors and transactions in the D-organization can be (re)used by multiple I-actors, it can be useful to model a single ATD for each aspect organization. This will give more insight in the coherence of the different I- and D-transactions.

It is also possible to skip the models of the I-organization and the D-organization. In that way only B-services are identified and the decision how to support these services is deferred to the construction part of our MDEE approach. However, this means that less functionality of the IT system is specified in terms of the using system and more decisions are made in the construction phase of the system development process. This means less automation in the development process and less (explicit) alignment between the organization and the resulting IT system.

Our MDEE approach can be used to create IT systems just supporting a part of the organization. In such cases the organization model will only model a part of the organization (e.g. a department) and all actors outside this part of the organization will be considered as external actor.

The service orchestrations in our MDEE approach are quite readable because we did not take cancellation and dissent patterns into account. If cancellation and dissent patterns are added the orchestrations can become cumbersome. If so, it can be more convenient to use a rule approach instead of an orchestration approach. In such a rule approach no service orchestrations are defined but each action rule in the organization model is supported by a separate system rule. These rules can be executed on a rules engine. Each service execution will lead to a couple of events (e.g. requested, promised, stated, accepted) which are send to the rules engine. Based on these events and the applicable rules additional services will be requested. This approach is more flexible but also more difficult to read because business processes are spread amongst multiple, separately defined rules.

In our MDEE approach reuse at runtime is only done at the service level. B-, I-, and D-services can easily be reused. However, in programming languages like Java all kinds of lower level implementation elements are reused. We see it as an advantage that runtime reuse is limited to services with well-specified interfaces thereby reducing the number of dependencies. Reuse of lower level implementations elements (i.e. parts of service implementations like GUI parts or calculations) is however possible at design time by using a central model repository and appropriate tooling which enable the reuse of model parts.

We see tool support as an essential element for the success of our MDEE approach. An integrated tool is needed supporting the full MDEE approach. This tool needs to be able to check the syntax of each model, enable the visual definition of models, and support and automate the MDEE steps as much as possible. All models need to be saved in a central repository to enable consistency checking between all models. Using the automated transformations between models the tool should propagate the changes in one model to all associated models.

Tool support is also necessary to make it easy to change IT systems build using our MDEE approach. In our research we mainly focused on building new IT systems, but the real advantage of MDE will come up when maintaining and changing applications. MDE improves the long-term productivity of developers by reducing the rate at which primary software artifacts become obsolete. To do so, our MDEE approach makes the resulting IT systems less sensitive for changes in personnel (by using high-level models which are easy to understand), requirements (by providing partly automated transformations from the organization model to executable models), and deployment platforms (the executable models are as high-level as possible and mostly specified in broadly supported industry standards).

9.4 Future Work

As we stated in the previous section our research had a limited scope. We therefore could not specify every aspect of our MDEE approach in detail. The main research limitations as described in the evaluation are also the main subjects for future research. Future research should hence focus on incorporating existing IT systems in the MDEE approach. This means the service identification step needs to be extended by identifying the existing systems and the services they provide. This will also affect the component identification step: existing services are already grouped in components. We already presented two different scenario's for component identification. Future work should research which scenario can be best used in what situation.

Future research should also focus on more extensive research into the details of the needed modeling languages. For each language the abstract syntax, concrete syntax, and semantics needs to be defined [60]. Defining the semantics of the modeling languages of the implementation model (i.e. how language elements behave when they are executed) leads to the last research limitation which should be tackled in future research, that is, the needed engines need to be implemented and a full working example needs to be constructed.

We did evaluate our MDEE approach with a single example case. More validation of our MDEE research is needed with more case studies in different domains. The cases should also be bigger (i.e. more services) to validate if the approach can handle such cases and if the models stay well-organized. The MDEE approach should also be validated with all involved actors. Does our approach really fulfill the needs of all actors in all cases?

Another important point to research in future work is how actors should use our MDEE approach. This research should answer questions like: what actors should perform what step in what order? How can actors verify if their models are correct? In close relation to the last question are the quality aspects of an MDEE approach. It should be researched how techniques like model validation [15], model checking [21], and model-based testing [85] can be used in combination with our MDEE approach.

The architecture part of our MDEE approach has not been described in detail. We referred to the work of Earl [36] as the principles of our architecture. Future work could be done researching the completeness of these principles and validating our MDEE approach against these architecture principles.

More research is needed on the implementation of transformations. We described each MDEE step with informal derivation rules. Future research should define each MDEE step as a formal, executable model transformation. For example in a language like QVT [76].

Future work should also focus on including dissent and cancellation patterns (as defined in enterprise ontology [34]) in our MDEE approach. It can also be researched if another part of enterprise ontology, information links and banks (modeled in the Actor Bank Diagram), can be included in MDEE. Goethals [44] describes and evaluates data storage options in a service-oriented environment (i.e. where the data is stored and who stores it) from a more technical perspective. The combination of his work and the fact banks in enterprise ontology can be a useful addition to our MDEE approach.

One of the engines we used to execute the implementation model is a component framework. As we explained a component framework provides a variety of runtime services to support and enforce the component model (e.g. communication services, security services, etc.). Future work is needed to research if the policies in the component framework and model like security can be derived from the organization model. Future work should also decide on how to provide custom engines, using component framework extensions or by including them in the components themselves.

Last, but definitely not least, tools supporting our MDEE approach should be implemented. We see tool support as an essential element for the success of our MDEE approach.

Bibliography

[1] J. O. Aagedal and I. Solheim. New roles in model-driven development. In *Proceedings of Second European Workshop on Model Driven Architecture (MDA), Canterbury, England*, 2004.

[2] Active Endpoints Inc., Adobe Systems Inc., BEA Systems Inc., IBM Corp., Oracle Inc., and SAP AG. Web Services Human Task (WS-HumanTask), version 1.0. Technical report, June 2007.

[3] Active Endpoints Inc., Adobe Systems Inc., BEA Systems Inc., IBM Corp., Oracle Inc., and SAP AG. WS-BPEL Extension for People (BPEL4People), version 1.0. Technical report, 2007.

[4] A. Albani and J. Dietz. The benefit of enterprise ontology in identifying business components. In *WCC '06: Proceedings of the IFIP World Computer Congress*, Santiago de Chile, Chile, 2006.

[5] A. Albani, J. Dietz, and J. Zaha. Identifying business components on the basis of an enterprise ontology. In *Interoperability of Enterprise Software and Applications*, pages 335–347. Springer, 2006.

[6] A. Albani, S. Overhage, and D. Birkmeier. Towards a systematic method for identifying business components. In *CBSE '08: Proceedings of the 11th International Symposium on Component-Based Software Engineering*, pages 262–277, Berlin, Heidelberg, 2008. Springer-Verlag.

[7] P. Amaya, C. González, and J. M. Murillo. Towards a subject-oriented model-driven framework. *Electr. Notes Theor. Comput. Sci.*, 163(1):31–44, 2006.

[8] A. Anaby-Tavor, D. Amid, A. Sela, A. Fisher, K. Zhang, and O. T. Jun. Towards a Model Driven Service Engineering Process. *Congress on Services - Part I, 2008. SERVICES '08. IEEE*, pages 503–510, July 2008.

[9] L. Apostel. Towards the formal study of models in the non-formal sciences. In H. Freudenthal, editor, *The concept and the role of the model in mathematics and natural and social sciences*, Dordrecht, the Netherlands, 1960. D. Reidel Publishing Company.

[10] A. Arsanjani and A. Allam. Service-oriented modeling and architecture for realization of an soa. In *SCC '06: Proceedings of the IEEE International Conference on Services Computing*, page 521, Washington, DC, USA, 2006. IEEE Computer Society.

[11] C. Atkinson and T. Kühne. Rearchitecting the uml infrastructure. *ACM Trans. Model. Comput. Simul.*, 12(4):290–321, 2002.

[12] C. Atkinson and T. Kühne. Model-driven development: A metamodeling foundation. *IEEE Software*, 20(5):36–41, 2003.

[13] F. Bachman, L. Bass, S. Buhman, S. Comella-Dorda, F. Long, R. Seacord, and K. Wallnau. Technical Concepts of Component-Based Software Engineering. Technical Report CMU/SEI-2000-TR-008, Software Engineering Institute, Carnegie Mellon University, 2000.

[14] D. Batory. Multilevel models in model-driven engineering, product lines, and metaprogramming. *IBM Syst. J.*, 45(3):527–539, 2006.

[15] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 2000.

[16] D. Birkmeier. Graphpartitionierungsalgorithmen zur Komponentenidentifikation - Entwicklung, Implementierung und Analyse, sowie Validierung anhand einer Fallstudie aus der Versicherungsbranche. Master's thesis, Universität Augsburg, 2008.

[17] P. V. Biron and A. Malhotra. XML Schema Part 2: Datatypes Second Edition. Technical report, W3C, 2004.

[18] M. Bräuer and H. Lochmann. Towards semantic integration of multiple domain-specific languages using ontological foundations. In *Fourth International Workshop on Software Language Engineering, Nashville, USA*, Grenoble, France, October 2007. megaplanet.org.

[19] C. Brooks, T. H. Feng, E. A. Lee, and R. von Hanxleden. Multimodeling: A preliminary case study. Technical Report UCB/EECS-2008-7, EECS Department, University of California, Berkeley, Jan 2008.

[20] M. Bunge. *Treatise on Basic Philosophy: Volume 4: Ontology II: A World of Systems*. Springer, 1 edition, April 1979.

[21] W. Chan, R. J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, 1998.

[22] R. N. Charette. Why software fails. *IEEE Spectrum*, 42(9):42–49, September 2005.

[23] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. Technical report, W3C, 2001.

[24] S. Cohen. Ontology and Taxonomy of Services in a Service-Oriented Architecture. *Microsoft Architect Journal*, (11), April 2007.

[25] F. Curbera, D. F. Ferguson, M. Nally, and M. L. Stockton. Toward a Programming Model for Service-Oriented Computing. In B. Benatallah, F. Casati, and P. Traverso, editors, *Service-Oriented Computing - ICSOC 2005, Third International Conference, Amsterdam, The Netherlands, December 12-15, 2005, Proceedings*, volume 3826 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2005.

[26] J. de Jong. Integration aspects between the B/I/D organizations of the enterprise. In A. Albani, J. Barjis, and J. L. Dietz, editors, *5th International Workshop, CIAO! 2009, and 5th International Workshop, EOMAS 2009, held at CAiSE 2009, Amsterdam, The Netherlands*, volume 34 of *Lecture Notes in Business Information Processing*, pages 187–200. Springer-Verlag Berlin Heidelberg, June 2009.

[27] L. DeMichiel and M. Keith. JSR 220: Enterprise JavaBeans,Version 3.0 - EJB Core Contracts and Requirements. Final release, Sun Microsystems, May 2006.

[28] T. Denton, E. Jones, S. Srinivasan, K. Owens, and R. W. Buskens. NAOMI - An Experimental Platform for Multi-modeling. In K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Völter, editors, *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings*, volume 5301 of *Lecture Notes in Computer Science*, pages 143–157. Springer, 2008.

[29] A. v. Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.

[30] J. Dietz. Enterprise ontology - understanding the essence of organizational operation. *Enterprise Information Systems VII*, pages 19–30, 2006.

[31] J. Dietz. *Architecture - Building strategy into design*. Academic Service, The Hague, The Netherlands, 2008.

[32] J. Dietz and J. Hoogervorst. Enterprise Ontology and Enterprise Architecture - how to let them evolve into effective complementary notions. *GEAO Journal of Enterprise Architecture*, 1, 2007.

[33] J. L. G. Dietz. A world ontology specification language. In *On the Move to Meaningful Internet Systems 2005: OTM 2005 Workshops, OTM Confederated International Workshops and Posters, AWeSOMe, CAMS, GADA, MIOS+INTEROP, ORM, PhDS, SeBGIS, SWWS, and WOSE 2005, Agia Napa, Cyprus, October 31 - November 4, 2005, Proceedings*, Lecture Notes in Computer Science, pages 688–699. Springer, 2005.

[34] J. L. G. Dietz. *Enterprise Ontology*. Springer-Verlag, Berlin Heidelberg, 2006.

[35] J.-J. Dubray. *Composite Software Construction*. InfoQ.com, C4Media, 2007.

[36] T. Erl. *Principles of Service Design*. The Prentice Hall Service-Oriented Computing Series from Thomas Erl. Prentice Hall, Upper Saddle River, NJ, USA, 2007.

[37] T. Erl. *SOA Design Patterns*. The Prentice Hall Service-Oriented Computing Series from Thomas Erl. Prentice Hall, Upper Saddle River, NJ, USA, 2009.

[38] A. Erradi, S. Anand, and N. Kulkarni. Soaf: An architectural framework for service definition and realization. In *SCC '06: Proceedings of the IEEE International Conference on Services Computing*, pages 151–158, Washington, DC, USA, 2006. IEEE Computer Society.

[39] R. Esser and J. W. Janneck. A framework for defining domain-specific visual languages. In *Workshop on Domain-Specific Visual Languages*, Tampa Bay (Fl), October 2001.

[40] M. Fan-Chao, Z. Den-Chen, and X. Xiao-Fei. Business Component Identification of Enterprise Information System: A hierarchical clustering method. *IEEE International Conference on E-Business Engineering (ICEBE'05)*, pages 473–480, 2005.

[41] J.-M. Favre. Towards a basic theory to model driven engineering. In *Third Workshop in Software Model Engineering (WiSME@UML)*, 2004.

[42] K. J. Fellner and K. Turowski. Classification framework for business components. In *HICSS '00: Proceedings of the 33rd Hawaii International Conference on System Sciences-Volume 8*, page 8047, Washington, DC, USA, 2000. IEEE Computer Society.

[43] F. Fondement and R. Silaghi. Defining Model Driven Engineering Processes. In *Third International Workshop in Software Model Engineering (WiSME), held at the 7th International Conference on the Unified Modeling Language (UML)*, 2004. Available as Technical Report IC/2004/94, Ecole Polytechnique Fédérale de Lausanne (EPFL), School of Computer and Communication Sciences, November 2004.

[44] F. Goethals. *Classifying and Assessing Extended Enterprise Integration Approaches*. PhD thesis, Katholieke Universiteit Leuven, December 2006.

[45] T. Halpin. *Conceptual schema and relational database design (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[46] T. A. Halpin and H. A. Proper. Database Schema Transformation and Optimization. In M. P. Papazoglou, editor, *OOER'95: Object-Oriented and Entity-Relationship Modelling, 14th International Conference, Gold Coast, Australia, December 12-15, 1995, Proceedings*, volume 1021 of *Lecture Notes in Computer Science*, pages 191–203. Springer, 1995.

[47] G. Hardjosumarto. An Enterprise Ontology based Approach to Service Specification. Master's thesis, Delft University of Technology, Delft, the Netherlands, October 2008.

[48] Z. Hemel, R. Verhaaf, and E. Visser. Webworkflow: An object-oriented workflow modeling language for web applications. In K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Völter, editors, *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings*, volume 5301 of *Lecture Notes in Computer Science*, pages 113–127. Springer, 2008.

[49] A. Hessellund, K. Czarnecki, and A. Wasowski. Guided development with multiple domain-specific languages. In *In ACM/IEEE 10th International Conference On Model Driven Engineering Languages and Systems (MODELS 2007*, 2007.

[50] A. Hessellund and A. Wasowski. Interfaces and metainterfaces for models and metamodels. In K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Völter, editors, *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings*, volume 5301 of *Lecture Notes in Computer Science*, pages 401–415. Springer, 2008.

[51] A. R. Hevner, S. T. March, J. Park, and S. Ram. Design science in information systems research. *MIS Quarterly*, 28(1):75–105, 2004.

[52] R. High, S. Kinder, and S. Graham. IBM's SOA Foundation: An architectural introduction and overview. Version 1.0. Technical report, IBM, November 2005.

[53] J. A. P. Hoogervorst. Enterprise Architecture: Enabling Integration, Agility and Change. *International Journal of Cooperative Information Systems*, 13(3):213–233, 2004.

[54] J. A. P. Hoogervorst. *Enterprise Governance & Architectuur - Corporate, IT en enterprise governance in samenhangend perspectief.* Sdu Uitgevers bv, Den Haag, 2007.

[55] H. Jain, N. Chalimeda, N. Ivaturi, and B. Reddy. Business Component Identification - A Formal Approach. In *EDOC '01: Proceedings of the 5th IEEE International Conference on Enterprise Distributed Object Computing*, page 183, Washington, DC, USA, 2001. IEEE Computer Society.

[56] D. Jungnickel. *Graphs, Networks and Algorithms*, chapter The Greedy Algorithm, pages 123–146. Springer, Berlin, 2005.

[57] G. Keller, M. Nüttgens, and A. Scheer. *Semantische Processmodellierung auf der Grundlage Ereignisgesteuerter Processketten (EPK).* Veröffentlichungen des Instituts für Wirtschaftsinformatik, Heft 89 (in German), University of Saarland, Saarbrücken, 1992.

[58] S. Kent. Model driven engineering. In *IFM '02: Proceedings of the Third International Conference on Integrated Formal Methods*, pages 286–298, London, UK, 2002. Springer-Verlag.

[59] B. Kernighan and S. Lin. An efficient heurisitc procedure for partitioning graphs. *Bell Systems Technical Journal*, 49:291–307, 1970.

[60] A. G. Kleppe. A language description is more than a metamodel. In *Fourth International Workshop on Software Language Engineering, Nashville, USA*, Grenoble, France, October 2007. megaplanet.org.

[61] T. Kühne. Matters of (meta-)modeling. *Software and System Modeling*, 5(4):369–385, 2006.

[62] K. Levi and A. Arsanjani. A goal-driven approach to enterprise component identification and specification. *Commun. ACM*, 45(10):45–52, 2002.

[63] J. Luoma, S. Kelly, and J. pekka Tolvanen. Defining domain-specific modeling languages: Collected experiences. In *In Proceedings of the 4th OOPSLA Workshop on Domain-Specific Modeling (DSM04)*, 2004.

[64] J. I. McCormack, T. A. Halpin, and P. R. Ritson. Automated Mapping of Conceptual Schemas to Relational Schemas. In *CAiSE '93: Proceedings of Advanced Information Systems Engineering*, pages 432–448, London, UK, 1993. Springer-Verlag.

[65] J. McGovern, O. Sims, A. Jain, and M. Little. *Enterprise Service Oriented Architectures: Concepts, Challenges, Recommendations*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[66] S. J. Mellor, K. Scott, A. Uhl, and D. Weise. Model-Driven Architecture. In *OOIS '02: Proceedings of the Workshops on Advances in Object-Oriented Information Systems*, pages 290–297, London, UK, 2002. Springer-Verlag.

[67] T. Mens and P. V. Gorp. A taxonomy of model transformation. *Electr. Notes Theor. Comput. Sci.*, 152:125–142, 2006.

[68] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[69] J. Mulder. *Rapid Enterprise Design*. PhD thesis, Delf University of Technology, 2006.

[70] OASIS. Web Services Business Process Execution Language Version 2.0. Technical report, OASIS Web Services Business Process Execution Language (WS-BPEL) TC, 2007.

[71] OMG. MDA Guide Version 1.0.1, June 2003.

[72] OMG. Business Process Modeling Notation (BPMN) Specification v. 1.0. OMG Final Adopted Specification dtc/06-02-01, Object Management Group, http://www.bpmn.org/, 2006.

[73] OMG. CORBA Component Model Specification Version 4.0. OMG Available Specification formal/06-04-01, Object Management Group, April 2006.

[74] OMG. Meta Object Facility (MOF) Core Specification version 2.0. OMG Available Specification formal/06-05-01, Object Management Group, 2006.

[75] OMG. OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2. OMG Available Specification formal/2007-11-02, Object Management Group, 2007.

[76] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. OMG Final Adopted Specification formal/2008-04-03, Object Management Group, 2008.

[77] M. Op 't Land. *Applying Architecture and Ontology to the Splitting and Allying of Enterprises*. PhD thesis, Delft University of Technology, Schildmos 13, 3994 LS Houten, Netherlands, June 2008.

[78] ORMSC Architecture Board. Model Driven Architecture - A Technical Perspective, Jule 2001.

[79] OSOA. SCA Service Component Architecture - Assembly Model Specification v1.00. Technical report, Open SOA Collaboration, March 2007.

[80] OSOA. SCA Service Component Architecture - Client and Implementation Model Specification for WS-BPEL v1.00sca c client and implementationsca c client and implementationsca c client and implementationsca c client and implementation. Technical report, Open SOA Collaboration, March 2007.

[81] OSOA. SCA Service Component Architecture - SCA Java Component Implementation v1.00. Technical report, Open SOA Collaboration, February 2007.

[82] M. P. Papazoglou and W.-J. V. D. Heuvel. Service-oriented design and development methodology. *Int. J. Web Eng. Technol.*, 2(4):412–442, 2006.

[83] C. Peltz. Web Services Orchestration and Choreography. *Computer*, 36(10):46–52, 2003.

[84] C. Rettig. The trouble with enterprise software. *MITSloan Management Review*, Internet Edition, August 2007.

[85] P. Santos-Neto, R. Resende, and C. Pádua. Requirements for information systems model-based testing. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 1409–1415, New York, NY, USA, 2007. ACM.

[86] Software Productivity Research. SPR Programming Languages Table. Technical report, Software Productivity Research, 2005.

[87] R. Soley. Model Driven Architecture, November 2000.

[88] S. Stein, S. Kühne, and K. Ivanov. Business to IT Transformations Revisited. In C. Pautasso and J. Koehler, editors, *1st International Workshop on Model-Driven Engineering for Business Process Management*, pages 1–12, Milano, Italy, September 2008.

[89] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.

[90] L. Terlouw. Insurance Company Case - Enterprise Ontology. May 2008.

[91] L. Terlouw. Towards a Business-Oriented Specification for Services. In J. L. G. Dietz, A. Albani, and J. Barjis, editors, *Advances in Enterprise Engineering I, 4th International Workshop CIAO! and 4th International Workshop EOMAS, held at CAiSE 2008, Montpellier, France, June 16-17, 2008. Proceedings*, volume 10 of *Lecture Notes in Business Information Processing*, pages 122–136. Springer, 2008.

[92] L. Terlouw and A. Albani. An Enterprise Ontology-Based Approach to Service Specification. Technical report, Delft University of Technology, 2009.

[93] L. Terlouw and J. Dietz. Comparing Methodologies for Service-Orientation using the Generic System Development Process. Technical report, Delft University of Technology, 2009.

[94] X. Thirioux, B. Combemale, X. Crégut, and P.-L. Garoche. A Framework to formalise the MDE Foundations. In R. Paige and J. Bézivin, editors, *International Workshop on Towers of Models (TOWERS), Zurich, 25/06/07*, pages 14–30, http://planetmde.org, juin 2007. Model Driven Engineering (MDE).

[95] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML Schema Part 1: Structures Second Edition. Technical report, W3C, 2004.

[96] W. van der Aalst and M. Pesic. Decserflow: Towards a truly declarative service flow language. In F. Leymann, W. Reisig, S. R. Thatte, and W. van der Aalst, editors, *The Role of Business Processes in Service Oriented Architectures*, number 06291 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[97] S. D. Vermolen and E. Visser. Heterogeneous coupled evolution of software languages. In K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Völter, editors, *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems (MODELS 2008)*, volume 5301 of *Lecture Notes in Computer Science*, pages 630–644, Heidelberg, September 2008. Springer.

[98] E. Visser. WebDSL: A case study in domain-specific language engineering. In R. Lammel, J. Saraiva, and J. Visser, editors, *Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, Lecture Notes in Computer Science. Springer, 2008.

[99] Z. J. Wang, X. F. Xu, and D. C. Zhan. A Survey of Business Component Identification Methods and Related Techniques. *International Journal of Information Technology*, 2(4), 2005.

[100] J. B. Warmer and A. G. Kleppe. Building a flexible software factory using partial domain specific models. In *Sixth OOPSLA Workshop on Domain-Specific Modeling (DSM'06), Portland, Oregon, USA*, pages 15–22, Jyvaskyla, October 2006. University of Jyvaskyla.

[101] J. Zachman. *Concepts of the Framework for Enterprise Architecture*. Zachman International Inc., 1997.

[102] L. Zhang, J. Zhang, and H. Cai. Services Computing: Core Enabling Technology of the Modern Services Industry. *Tsinghua University Press*, 2007.