4 Service Modelling

M.W.A. Steen, M.E. Iacob, M.M. Lankhorst, H. Jonkers, M. Zoet, W. Engelsman, J. Versendaal, H.A. Proper, L. Debije, K. Gaaloul

The development of enterprise services involves making design decisions at different levels, ranging from strategic to infrastructural choices, and concerning many different aspects, ranging from customer interaction to information registration concerns. In order to support an agile development process with short iterations through each of these levels and aspect, we need to manage the inherent complexity and support rapid feedback on the impact of design decisions across the various aspects of service development. The use of models can help to manage the coherence among the different aspects in service design, and in facilitating and accelerating changes. Therefore, we propose a comprehensive framework and method for service modelling and model integration as an important ingredient of an agile service development methodology. This method is aimed at providing a shorter path between requirements and execution through the use of models to feed run-time execution engines, fast validation at the model level, support for communication with stakeholders, integration of different aspects, domains and fields of expertise, and consistency across the enterprise.

4.1 Introduction

Enterprise services are provided by a complex socio-technical system – the service system – comprising both human and technological resources. These resources need to be instructed on what to do, when and how, in order to deliver the required service. Service development is the entire process of designing, implementing, maintaining and adapting services. The development of enterprise services involves making design decisions at different levels, ranging from strategic to infrastructural choices, and concerning many different aspects, ranging from customer interaction to information registration concerns. Think of information to be managed, partners to involve, channels to be used and processes to execute in order to deliver the overall service. In addition, the distinction between business and IT services is blurring, with the ongoing shift from people-delivered, possibly IT-supported services to IT-delivered, possibly people-supported services.

Service providers need to address the question of how to align their business operations and information technology to market demands, legal and regulatory requirements, and business strategy. There are many stakeholders involved, each with their own interests and concerns. In addition, services have to fit with the needs of customers, the organizational context and the technological infrastructure. Marketeers will be interested in targeted market segments, channels and proposed customer value; business operations managers in the impact on business processes; IT managers in the impact on applications and technology infrastructure; line of business managers in the division of roles and responsibilities; partner managers in the involvement of key partners; and so on, and so forth.

Next to the many aspects that need to be addressed when developing services, the other main challenge of service organizations is to deal with change (also see Chap. 2). They are continuously confronted with changes, such as changing market conditions, changing legislation, technological changes, changing volumes, changing partnerships, and the introduction of new channels. Therefore, we advocate an agile way of working, which is detailed further in Chap. 6. However, in order to support such an agile development process with short iterations through each of the design levels and aspect, we need to manage the inherent complexity and support rapid feedback on the impact of design decisions across the various aspects of service development.

It is simply not possible to be agile in such a complex endeavour without the use of suitable and coherent abstractions. In this chapter, we therefore propose a comprehensive framework for service development that takes the various aspects of services into consideration. This framework can serve as a map for plotting and relating the various concerns of stakeholders. The approach for composing a way of working as outlined in Chap. 6 can then be used to plan a route through this service development landscape. In particular, we can on the one hand position the various development artefacts within this framework, and on the other hand use these to select and combine relevant agile practices (see in particular Sect. 6.7.2).

The framework is complemented with a method for integrating the different aspects. In this way, we obtain an integrated, model-based, agile approach to service development. This method enables a shorter path between requirements and execution, through the use of models to feed run-time execution engines, fast validation at the model level, support for communication with stakeholders, integration of different aspects, domains, expertises, and consistency across the enterprise.

4.2 The Role of Models in Agile Service Development

As we explained in Chap. 1, we strive for an agile engineering approach to service development. Most mature engineering disciplines are firmly rooted in the use of formal, mathematical models for predicting the various properties of their design

artefacts, in order to make the right decisions. In this context, we use the following definition of a model:

A *model* is a purposely abstracted and unambiguous conception of a domain.

This definition is taken from (Lankhorst et al. 2009), and is originally based on (Falkenberg et al. 1998). In this definition, a 'domain' is any subset of a conception of the universe – i.e., the service world we are talking about – that is viewed as being some 'part' or 'aspect' of the universe. For complex worlds, such as the world of enterprise services, many different domains or abstractions can be envisioned, ranging from the financial and economic structure of the service network, via the individual organizations involved and their business processes and functions, to the IT implementations and infrastructures.

Models in general serve many purposes. By means of their abstraction from details, they help us focus on the essence, a specific purpose. This way, they provide us with more insight into a situation. This insight might be needed towards an informing purpose, analysis, decision making, etcetera. This is the *descriptive* use of models.

Models may also be used *prescriptively*, to provide guidance towards the execution of work. This could for example concern design activities or operational work processes. Because of the unambiguous nature of models, the guidance they provide is often clearer and more explicit than when natural language or simple pictures are used. In particular when formalized rules or processes must be followed, using models may help avoid miscommunication or differences in interpretation, and thus are a great help in project communication.

In Chap. 2, we outlined the various attributes of agility, both of agile processes and of agile systems. The use of models contributes to the realization of many of these agility aspects. Models help us in clearly establishing and prioritizing requirements, and in achieving Traceability between business goals, requirements and design models, which is important to ensure that our designs really fit the needs of the business. Models also help in estimating the effort needed for a specific requirement, for example because they give insight in the size of the functionality needed or in the complexity of a system's interactions. This aids in prioritizing requirements, and improves the competency of the service development team in delivering on its promises.

Assuming that models are easier to create and maintain that software code, the use of models may help to accelerate and shorten the path between requirements and execution. If we automatically convert models to implementations, or even better, if we have an infrastructure that can directly interpret and execute these models, we can build and change services with less software coding or even no coding at all. Considering that various kinds of 'engines', for example for business process or business rule execution, are becoming increasingly popular because of this. Thus, making changes becomes much easier and the development process is accelerated. The figure below illustrates this.

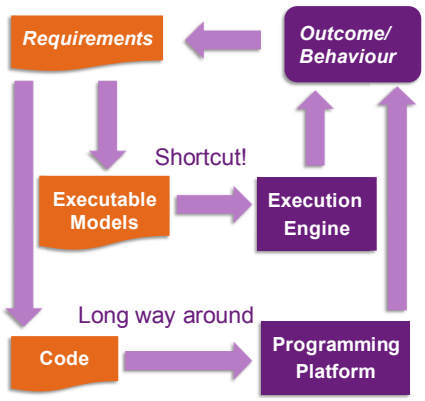

Fig. 15. Agility through model execution. ; when models are executable, one can take a shortcut and obtain faster feedback on design decisions.

Furthermore, models may serve as an important means for communication with business stakeholders (also see Chap. 7). These stakeholders can be more closely involved in designing or changing a service, or in some cases they may even be able to make changes themselves. In particular when the infrastructure is built using engines like those mentioned above, some types of changes may be made directly by end-users, such as changing business rules or workflows. Thus, using models may improve stakeholder involvement, increase responsiveness to business needs, and speed up the development process.

Models also facilitate the deployment of a change, by shortening the development, testing, acceptance and production process. Since less coding is involved, the development process takes less effort. But you can also use models to validate or verify a service design offline, for example, by means of simulation and model-based analysis techniques. You can detect errors at an earlier stage, when they are usually cheaper to fix, and you can predict behaviour, for example resource consumption under heavy usage.

The usage of models that show the dependencies between different service aspects, also helps you in assessing the effects of changes, for example by seeing how they propagate through the models. This ensures consistency and avoids unexpected and unwanted side-effects.

Finally, models may facilitate the integration and reuse of services. If you have model-based descriptions of service interfaces, finding and integrating these services is easier and can sometimes even be automated. Service composition and bundling, offering new combinations of existing services, is also facilitated if you can already check at the model level whether these services are compatible.

Importantly, and as discussed in Chap 6, we do not advocate ‘big design upfront’ of the entire service landscape. Rather, the modelling efforts themselves should be iterative and should follow good practices for agile modelling, as for example provided by Ambler (2002) and others. But as we already outlined in

Chap. 3, an investment in a model-based architecture may well pay off in a much quicker and cheaper development afterwards. Some design up-front is therefore required.

4.3 Adoption Levels of Modelling

Although the use of models in service development clearly has advantages in terms of agility, coherence, consistency and quality, there are also costs involved. Modelling requires effort, skills, and specialized tools. And scenarios in which models are directly executed, require an infrastructure consisting of appropriate execution engines. Unfortunately, the benefits of modelling can usually only be obtained after these investments in infrastructure, tools, education and human resources have been made. Therefore, a model-driven approach should only be adopted when the benefits outweigh the costs. Below, we describe a number of adoption levels for the use of models, each with their own set of benefits and costs. This can help organizations to choose the right level of adoption.

Table 2. Adoption of modelling.

Level of model adoption	Benefits	Costs
1. no models	none	none
2. informal models	improved communication	low
3. isolated formal models	unambiguous specification, analysis support	know-how and tools for specific technique(s)
4a. horizontally integrated formal models	cross-aspect impact of change analysis, consistency across domains, reuse	integrated tool-suite for modelling or model integration support, cross domain modelling expertise
4b. vertically integrated formal models	traceability to requirements, impact of change analysis, forward and backward engineering support, e.g. code generation	dedicated tool-chain and target platform, model transformation expertise
5. integrated formal models	Combined benefits of 3 and 4.	integration of tools and infrastructure components, combined expertise and know-how from 3 and 4.

Level 1 – no models – speaks for itself. If no models are used in the development process, there are obviously also no benefits and no costs for making models. Be aware, however, that overall development costs may increase, because there are limited means to manage the inherent complexity of service design.

At level 2 – informal models – service developers enjoy improved communication, while the modelling costs are still low. An ‘informal’ model has no formally

defined syntax or semantics. Examples include Visio diagrams and PowerPoint drawings.

At level 3 – isolated formal models – proper modelling tools are employed to model some aspects of the service design. We use 'isolated' here as opposite of 'integrated', meaning that multiple models may be used, but without being formally related to one another. The minimum requirement for a model to be 'formal' is that its syntax conforms to a metamodel. Examples include, BPMN process models and UML class diagrams. The advantage of formal models over informal models is that they are unambiguous and amenable to formal analysis, such that they can be used to predict properties of the service before it is implemented. Obviously, this level requires know-how and dedicated tools for the selected modelling techniques.

At the highest adoption level, 5 – integrated formal models – it is assumed that models are used at most abstraction levels and across most of the aspects of the service design. Moreover, these models are all assumed to be views on one, usually left implicit, integrated underlying model of the service. This level demands a lot in terms of skills, tools and infrastructure. Therefore, it may not be appropriate for every organization. There are two possible routes to achieve level 5: horizontal integration first (4a) and vertical integration first (4b). The desired benefits and the priorities of the organization determine which route is most appropriate and to what extent the other path is followed.

At level 4a – horizontally integrated formal models – service developers can reuse elements from one aspect model in another, check and enforce consistency across the aspects and perform cross-aspect impact-of-change analyses. Models are 'horizontally' integrated when they are at the same level of abstraction, but possibly addressing different aspects, and consistently referring to each other's elements. This level of integration requires an integrated tool-suite for modelling or another form of model integration support and cross-domain modelling expertise. In Sect. 4.7, we detail further the specific requirements this scenario places on tools support.

At level 4b – vertically integrated formal models – service developers can trace design and implementation artefacts back to the requirements that motivate their existence, analyse the impact of changes in requirements or designs on the lower abstraction levels, and make use of automated support for forward and backward engineering, such as code generation. Models are 'vertically' integrated when they are at different levels of abstraction, addressing the same aspects, and have relationships defined between semantically conformant elements. This level of integration requires a dedicated tool-chain and target platform as well as model transformation expertise.

4.4 The ASD Framework

In this section, we present a model-based framework for agile service development. We focus in particular on identifying the kinds of abstractions that are required to support an integral and coherent service development process. In developing this framework, our aim was not to develop "yet another" framework, but rather to combine the features of relevant existing frameworks that are relevant to (agile) system development.

Of course, we are not the first to propose a framework for enterprise service development. One of the best-known and oldest frameworks for the describing the design space of enterprises is the Zachman Framework for Enterprise Architecture (Sowa & Zachman 1992). It was first introduced in 1987 as the 'Framework for Information Systems Architecture' (Zachman 1987). The Zachman framework is a logical structure for classifying and organizing the elements and aspects of an enterprise (its ontology) that are significant to the management of the enterprise as well as to the development of the enterprise's systems. In its most simple form the Zachman framework depicts the concepts on the intersections between the roles in the design process, in particular the *planner*, *owner*, *designer*, and *builder*; and the product abstractions: that is, *what* (data) it is made of, *how* (function) it works and *where* (network) the components are located with respect to one another. Three additional columns of models depict *who* does what work, *when* do things happen, and *why* are various choices made?

A more recent framework with a strong impact on international standardization is the framework embedded in the ArchiMate language (The Open Group 2012; Lankhorst et al. 2009). The core of the ArchiMate language distinguishes between the structural or static aspect and the behavioural or dynamic aspect of enterprises. The structural aspect is further subdivided into active structural elements (the business actors, application components and devices that display actual behaviour, i.e., the 'subjects' of activity), and passive structural elements, i.e., the objects on which behaviour is performed.

In addition, ArchiMate makes a distinction between an external view and an internal view on systems. The service concept represents a unit of essential functionality that a system exposes to its environment. For the external users, only this external functionality, together with non-functional aspects such as the quality of service, costs, etc., are relevant. Services are accessible through interfaces, which constitute the external view on the structural aspect.

Finally, ArchiMate distinguishes three layers: The Business layer offers products and services to external customers, which are realized in the organization by business processes (performed by business actors or roles); the Application layer supports the business layer with application services which are realized by (software) application components; the Technology layer offers infrastructural services (e.g., processing, storage and communication services) needed to run applications, realized by computer and communication devices and system software.

There are many more framework and reference architectures with some relevance to service development. Standardization organizations, including OASIS, The Open Group, W3C and OMG, have produced various standards and whitepapers containing guidance for developing service oriented solutions, see (The Open Group 2009b) for an overview. However, these generally are of a technical nature and pay less attention to the business, organizational, decision and interaction aspects of enterprise service development. Most methodologies and development tools also boast their own world views. TOGAF, for example, has its own Content Framework, which categorizes architecture artefacts according to the TOGAF development phases (The Open Group 2011). The Design and Engineering Methodology for Organizations (DEMO) takes a language-action perspective and looks at organizations at an ontological, an infological and a datalogical level, and further distinguishes the construction, process, state, and action aspects (Dietz 2006).

While each of these frameworks has its merits, none of them covered all the aspects and perspectives that we encountered in agile service development. Nevertheless, they are complementary and contain many useful concepts that we can reuse in service development. We therefore saw the need to combine those features of the existing frameworks that are relevant to agile service development, into a framework that is comprehensive and specific to agile service development.

4.4.1 Service Aspects

Our framework aims to support agility and flexibility in realizing service requirements, while managing the inherent complexity. A good practice for achieving such flexibility is 'separation of concerns'. Following Zachman, our framework is structured along two axes: service aspects and abstraction levels. By dealing with each aspect separately before dealing with the bigger picture, we can maintain a grip on the complexity and avoid that design concerns get mixed up. This supports agility by providing a single point of definition and change for each aspect. A well-known example of this principle from web application development is the use of the so-called three-tier architecture, separating presentation, business logic and data. A similar subdivision applies to service development, where we need to address the interaction with customers, the provided functionality and the information being managed.

However, the functionality or business logic of enterprise services is usually not so easily captured. Other principles help us to divide this aspect further. Service-oriented architecture (Erl 2009) and structured analysis and design techniques (Marca and McGowan 1987) suggest us to separate, decompose and encapsulate groups of coherent functionalities into reusable building blocks, providing again services to their environment, resulting in a hierarchy of functional building blocks. Workflow thinking has taught us to separate activities or tasks to be executed from the (human or system) actors executing them. Business process management (BPM) (Brocke and Rosemann 2010) thinking suggests to separate the

coordination of such functional building blocks. And a final good practice is to 'separate the know from the flow', i.e., do not mix decision logic with coordination logic. Fig. 16 shows the six resulting service aspects and loosely relates them to the ArchiMate and Zachman frameworks.

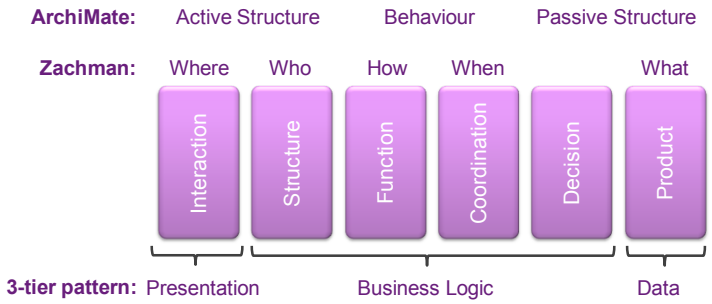

Fig. 16. The framework's aspects and their relationship to other frameworks.

Interaction

The Interaction aspect is concerned with the way in which the enterprise interacts with its environment. It includes the enterprise's collaboration with its various partners and how its clients interact with the business services it provides. These services may be delivered through an online channel, but traditional, human-centric services, delivered e.g., via the telephone or over the counter, are also part of this. Hence, the interaction between user and service may involve graphical user interfaces, online forms, etc., but also the classical person-to-person interaction and service.

Structure

The Structure aspect concerns the way in which the enterprise organizes its human and technological resources. This includes the organizational structure, comprising the definition and allocation of roles, responsibilities, authorizations, reporting lines, etc., but also the information system structures, i.e., the technical and application architectures.

Function

In the Function aspect, we address the individual elements of business and application functionality that, orchestrated and coordinated together, deliver the actual substance of a service. This comprises both the (manual) tasks of employees and the (automated) service logic of applications. Individual functions (and the services they deliver) are coordinated via the Coordination aspect, they use and produce information from the Information aspect, and they employ rules and calculations from the Decision aspect.

Coordination

The Coordination aspect focuses on the various dependencies between the activities needed to deliver services. This includes, for example, the specification and (possibly automated) orchestration of business processes, workflow support, etc. It comprises both the coordination within an individual organization and the coordination of activities with other organizations, which may be users of the service or partners in delivering it.

Decision

The Decision aspect captures the logic of reasoning used in the service domain, to reach decisions, i.e., how decisions are (to be) made. For example, in the domain of insurance policies or banking products, this pertains to decisions based on calculations, and other (logical) derivations. Part of this logic may take the form of executable specifications, such as decision tables or executable business rules; other elements are typically used by people, both in delivering the service and in defining, checking and enforcing an organization's 'rules of conduct'. However, the logic specified here should not include coordination, interaction or organization logic, which belong to the other aspects.

Product

Finally, the Product aspect is concerned with the things that the service produces and consumes, and the way in which these products are registered and managed. Products can refer both to tangible business objects, such as cars and pizza's, but also to intangible information items, such as insurance claims and pizza orders.

4.4.2 Abstraction Levels

Each of the service aspects can be considered at different abstraction levels (Fig. 17). We distinguish between the specification space and the human and technical infrastructure on which specifications are realized and deployed. The specification space can be divided further into a requirements level, a design level and an implementation level. These abstraction levels are detailed further below.

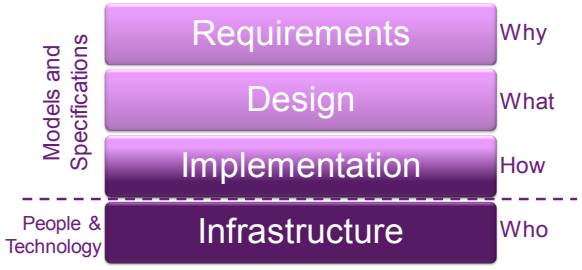

Fig. 17. The framework's abstraction levels.

Requirements level

The Requirements level deals with the motivation and rationale behind the service, i.e., its 'why', and comprises the service requirements from the business perspective. This level not only contains specifications of the requirements on the specific service under development, but also includes specifications of the context in which the service is to operate. Therefore, at this abstraction level, we recognize the need for at least two types of models: a context model and a requirements model.

Design level

The Design level contains the 'what' of the service: the interactions, processes, functions, rules and objects that are needed to realize the service. Designs are typically denoted in the form of some kind of model. Models, being formalized abstractions of reality that cover specific aspects of that reality and abstract from the rest, are a precise way of specifying services. The use of models as executable specifications is especially valuable from the perspective of agility. Because models can be checked in various ways before they are implemented, risks of changes can be managed and their effects can be predicted (within limits) before implementation. Furthermore, if implementing a change in the IT domain merely amounts to changing some models, an organization may react much more quickly to changing requirements than for example when large-scale software changes are needed.

Implementation level

The Implementation level describes the 'how', i.e., how the service will be implemented, in terms of both the people and the technology involved. Ideally, this level can be skipped, i.e., if the design models are directly executable on the infrastructure. However, more often than not this is not realistic, which makes it necessary to also look into the implementation artefacts.

Infrastructure level

Finally, we have the Infrastructure level. This is where the rubber meets the road: the people and technology actually delivering the service. One the one hand we find here people with suitable capabilities who deliver services through physical channels. To be able to deliver these services, they execute tasks, coordinate activities, manage other people, and enforce that rules are obeyed. One the other hand, there is the IT infrastructure which delivers services through online channels and comprises both generic hard- and software infrastructure and specific applications on top of that, such as DBMSs, BPMSs, rule engines, and web and application servers.

4.4.3 Overview and Use of the Framework

Putting all of the abstraction levels and aspects together results in the framework shown in Fig. 18.

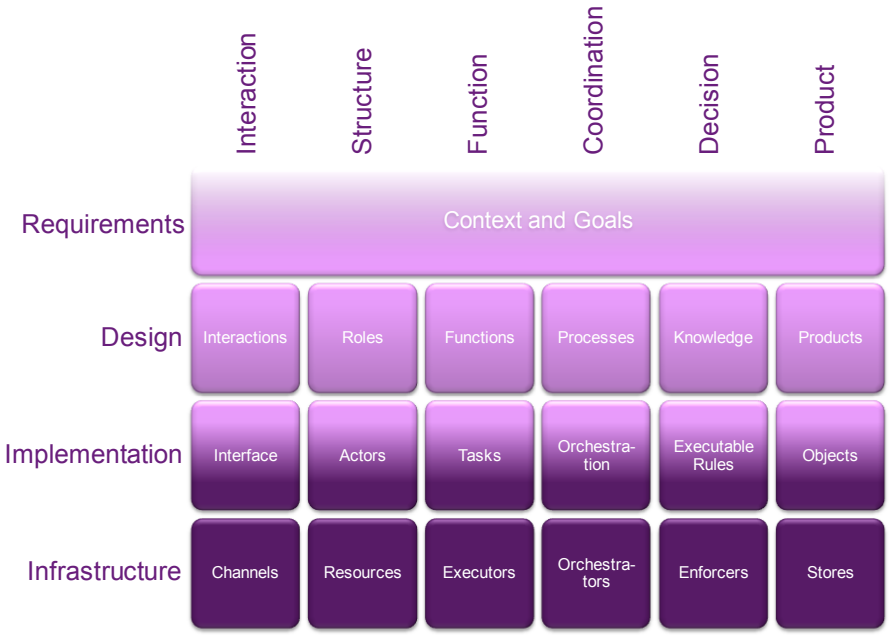

Fig. 18. Framework overview.

The framework illustrates that enterprise services are realized through combinations of business functions, processes, IT and more, all of which should be developed jointly. Of course, the framework is no more than that, a frame of reference. It does not specify a way of working and there is no requirement to fill each cell of the model individually. Rather, it provides a way to position and relate the various

design artefacts, where individual artefacts may cover more than one cell. For example, in the infrastructure layer, people (human resources) will often fill multiple positions at the same time, e.g., being both ‘manager’ and ‘coordinator’. Similarly, the more advanced IT systems, such as for case management, business process management or business rule management, cover multiple levels and aspects. Conversely, some cells may remain empty for a specific service. For example, if no significant coordination with others is required, business process specifications may be superfluous.

Service organizations can use the framework to plot the models and abstractions they are already using in their service development process and highlight white spots that they currently do not cover. To give an idea on what to put where, we return to the AgiSurance case study introduced in Chap. 2.

4.4.4 Modelling AgiSurance

While AgiSurance offers many different insurance products, each with their own unique properties and rules, they also share some characteristics. For each product there has to be an acceptance process and a claim handling function. In order to cope with the regularly changing product offering, AgiSurance wants to establish an agile service architecture, allowing easy configuration of new insurance products on a stable infrastructure. In the following, we focus on the redevelopment of the claim handling service. Currently, claims are received on paper and handled manually – a costly and error-prone process. Due to increased sales, AgiSurance expects a sharp rise in claims. Therefore, they want to optimize and partly automate the claim handling. The idea is to develop generic business and IT functions, and processes for handling insurance claims, that are fed with specific rules for decision making and interaction for each product.

In section 3.2.4, AgiSurance had already decided for a flexible, model-driven infrastructure, consisting of a database management system (DBMS), a business rule management system (BRMS) and a business process management system (BPMS). However, because of the relatively simple and stable processes for claim handling, they now decide not to use the BPMS, but to implement the processes directly onto a standard application server.

AgiSurance first constructs a *context model* and a *product model*, based on an analysis of contracts, policies and insurance legislation. The context model is complemented with a *business requirements model* to define the stakeholders, their goals and the requirements on the claim handling service. The organizational structure and the division of tasks and responsibilities does not really change, so these models can be copied from the corporate *business architecture*, defining organizational units, roles, functions and high-level processes, and the *handbook* with guidelines and procedures for employees. Next, the context and product models are detailed further into a *rule model* and an *information model*. The rule model will be executed directly on a rule engine; the information model is auto-

matically transformed into a *database schema*. In parallel, a *user interface model* and a *service model* are devised. The service model lays the foundations for the application code and the process model. The user interface model is used to generate the *web pages* for the claim handling service.

Fig. 19 plots the identified models and infrastructure elements on the ASD Framework. Here we can see that all the service aspects are covered to some extent, and that detailed implementation level models can be traced back to higher level requirements models. Design artefacts can also be related horizontally, such as the service, rule and information models, signifying that they refer to each other. Several models cover more than one aspect and/or level. The product model, for example, floats between the requirements and design layers, and covers parts of the information, decision and process aspects. By plotting the models on the framework, AgiSurance can identify where specifications are missing and where they should put their effort in verifying the consistency between models.

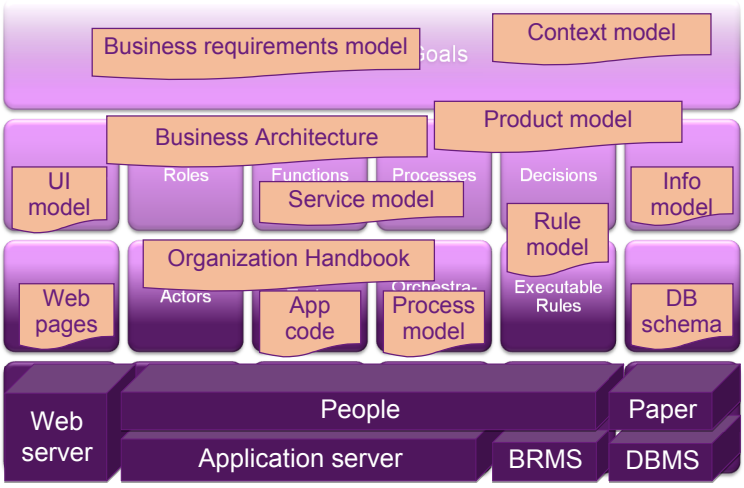

Fig. 19. Positioning AgiSurance models on the ASD Framework.

4.5 The ASD Conceptual Model

In this section, we describe the concepts underlying the various service aspects and their relationships. This gives body to the ASD Framework and is a first step towards integrating different models and specifications. The conceptual model is structured along the dimensions of the ASD Framework (see Sect. 4.4.3). First we describe the concepts pertaining to the Requirements level. At this level models are increasingly used to formalize service requirements and to use them to relate design artefacts to organizational goals. Service requirements, in turn, depend on a conceptualization of the business domain, the context in which the service is to

operate. Therefore, we introduce two modelling domains at this level: Context and Requirements. Second, we define concepts for each of the six design and implementation aspects. Each aspect constitutes a modelling domain. Finally, we define the relationships between the defined modelling domains.

The metamodels for each of the modelling domains were constructed by analysing a number of relevant and popular modelling techniques for the given domains. In a sense, we attempted to extract the best practices in conceptual modelling from a large number of existing techniques. Table 2 lists the modelling techniques that were analysed for each of the modelling domains. Each technique defines its own set of modelling concepts. Concepts that we encountered more than once (also those which could be considered synonymous) or which represent a key abstraction for the given domain, we selected as key concepts for that modelling domain.

Table 3. Relevant modelling techniques for each of the modelling domains.

Modelling domain	Modelling techniques
Context	ERD, ORM, OWL, SBVR, UML Class diagram
Requirements	ArchiMate Motivation extension, i*, KAOS
Interaction	ConcurTaskTrees, Diamodl, UML Use Case diagram, UsiXML
Structure	ArchiMate, BPMN, DEMO, e^3value, UML
Function	ArchiMate, DEM, DFD, e^3value, IDEF0, SoaML
Coordination	ArchiMate, BPMN, DEMO, UML Activity diagram
Decision	Decision tables, DMN, ECA rules, SBVR
Product	ERD, ORM, UML Class diagram

4.5.1 The Context Domain

A *context model* is a conceptual model of the domain in which an enterprise conducts its business and in which the service(s) under development should operate. It describes the vocabulary and key concepts of the business domain, as well as their properties and relationships. Usually, the context or domain model is only associated with a structural view of the business domain, but it can equally well contain dynamic views describing the main business activities and their constraints and dependencies. A context model is often complemented with the constraints that govern the integrity of the model. Sometimes the context model is referred to as *knowledge model*, because it defines the basic facts and rules of the business domain. Thus it is more than a list of dictionary-style definitions of terms.

The context model can be effectively used to verify and validate the understanding of the business domain among various stakeholders. It is especially helpful as a communication tool and a focusing point both amongst the different members of the business team as well as between the technical and business teams. In

addition, the context model forms the basis for defining requirements on the service(s) to be developed. Therefore, it is important that a context model itself is independent from design or implementation considerations.

The importance of context or domain modelling has been highlighted by many before us. As a consequence, many methods and techniques exist for it. The structure of a domain is often modelled using object-oriented techniques, such as UML class diagrams, or the more basic Entity Relationship Diagrams (ERD) (Chen 1976). In case UML (OMG 2011a) is used, domain concepts are represented as classes, their relationships as associations, and their properties as attributes. Constraints can be specified using the Object Constraint Language (OCL) (OMG 2010). However, the UML has been criticized a lot for being incomprehensible by domain experts that are not software engineers.

There are several alternatives from the data and knowledge representation community, such as the Web Ontology Language (OWL)) (W3C 2009), Object Role Modeling (ORM)) (Halpin and Morgan 2008) and the Semantics of Business Vocabulary and Business Rules standard (SBVR)) (OMG 2008), that take a more fact-oriented approach to modelling the domain. In these techniques, properties, relationships and other rules or constraints are all seen as 'fact types'. Sometimes these techniques enable automatic reasoning, for example to derive new facts.

By analysing and comparing the modelling techniques mentioned above, we have derived the following set of key concepts for context modelling:

Concept: an abstraction or generalization of a phenomenon that may occur in the domain.

Property: an attribute, characteristic, or quality of a Concept. Each Property has a type specifying the range of values it can take.

Relation: an association between two or more Concepts, each having a particular Role in the Relation.

Value Type: a range of values that Properties can take.

Role: the position of a Concept or the part played by a Concept in a Relation.

Constraint: a limitation, restriction or rule controlling the possible instances of concepts and relations, and values of properties.

The figure below depicts the context modelling metamodel, relating the key concepts defined above.

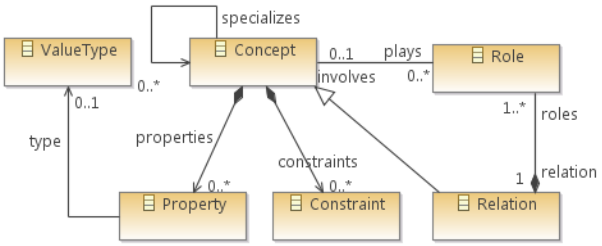

Fig. 20. The context metamodel.

An example context model

In finance, context models are to a large extent defined by the legal 'space' in which the financial institution operates. They need to comply with legislation based on international directives and standards, such as Solvency II, Basel III, and IFRS. Within this legal space, or more precisely their interpretation of the legal space, financial institutions define a product model defining their products and services.Both legislation and financial products and services are generally defined in legal documents (laws, contracts, policies). Drawing up a context model then consists of interpreting these documents, formalizing the definitions and rules contained within them.

Below, we show some of the results of such a context modelling exercise for AgiSurance, our example insurance company. A UML class diagram is used to model the domain concepts and their relations.

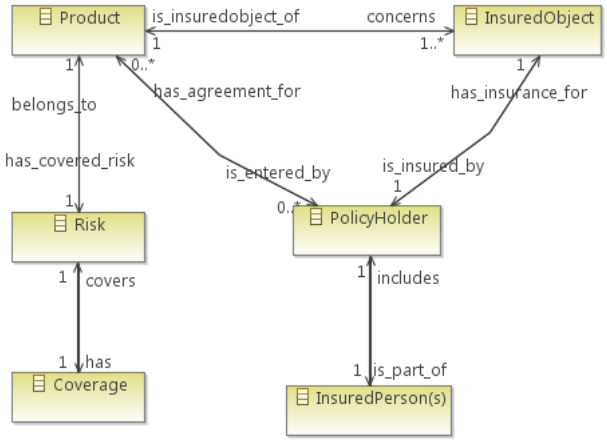

Fig. 21. Part of the AgiSurance context model in UML.

4.5.2 Requirements Modelling

Whereas the context model defines the inherent structures and constraints of the environment in which the enterprise has to operate, the *requirements model* captures the strategic direction and desires of the enterprise itself. A requirements model describes the motivation for the service(s) under development. It identifies the stakeholders and their concerns, and defines their goals or objectives.

In addition to clarifying and formally specifying the requirements for a service, requirements modelling is useful to achieve forward and backward traceability between objectives and design artefacts. Forward traceability is the ability to analyse the impact of a change in requirements. For example, when a business objective changes it becomes possible to analyse which services, and the components realizing those services, are affected by this change. Backward traceability can be used to determine the value or *raison d'être* of a design artefact. Backward traceability answers questions like: 'why was this service here? Who was responsible for this service? What is the added value of this service to the enterprise??'

Within the Goal-Oriented Requirements Engineering literature we can identify a number of relevant modelling techniques, such as KAOS (Van Lamsweerde 2003) and i* (Yu 1997). Several of these were analysed and compared in the design of the ArchiMate Motivation extension (Engelsman et al. 2011). KAOS is a language that refines system goals to concrete requirements. In i*, intentions of stakeholders (goals, beliefs) and their dependencies are modelled. Intentions are refined into tasks an actor has to perform to realize them.

Since the ArchiMate Motivation extension (The Open Group 2012) is based on these earlier techniques, we adopt the most essential concepts from ArchiMate as key to requirements modelling:

> *Stakeholder*: the role of an individual, team, or organization (or classes thereof) that represents their interests in, or concerns relative to, the outcome of the architecture.
>
> *Goal*: an end state that a stakeholder intends to achieve
>
> *Requirement*: a statement of need that must be realized by a system.

A number of different links are possible between these constructs. A goal can be associated with one or more stakeholder. There are two relations available for goal refinement. First, we have the goal decomposition relation. A decomposition of a goal is the conjunction of the set of subgoals that constitute the goal, in such a way that an immeasurable goal is decomposed into goals with measurable indicators, and a goal with measurable indicators is decomposed in subgoals with subindicators. The decomposition relation is used to operationalize goals. A goal influence relation is used to demonstrate that the satisfaction of one goal positively or negatively influences another goal.

A third link is the goal conflict relation. Two goals are conflicting if the satisfaction of one goal prevents the satisfaction of the other and vice versa. In this case, both goals are mutually exclusive.

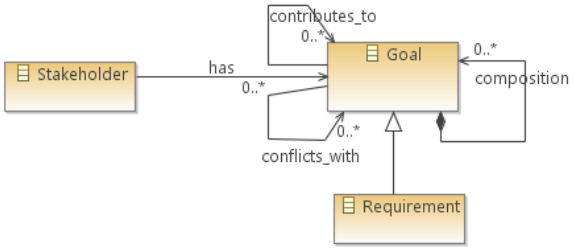

Fig. 22. The requirements metamodel.

Fig. 22 illustrates the underlying metamodel for requirements modelling. The main idea behind this metamodel is that a stakeholder may have a number of desires or intentions. An *intention* can be anything, most likely a desired state of the world. An intention only becomes a goal when a stakeholder is willing to commit resources to reach that state of the world. The previously discussed relations are used to refine goals into requirements. A *requirement* is a concrete goal that can be assigned to a single actor. A *goal* is a desired state of the world which is not yet concrete enough to be assigned to a single actor.

An example requirements model

Fig. 23 shows part of the business requirements model for AgiSurance using the ArchiMate Motivation extension. It shows three stakeholder roles: the Chief Operating Officer (COO) of AgiSurance, whose main goal is cost reduction; an Intermediary, whose main goal is to reduce the manual work he has in filing claims on his customers' behalf; and a Customer, who wants claims to be processed faster such that he will receive the insurance money in time to cover the incurred damage. The diagram further shows that the main goals and contributing goals will be realized by the two requirements 'Provide on-line claim submission service' and 'Automate claim handling'.

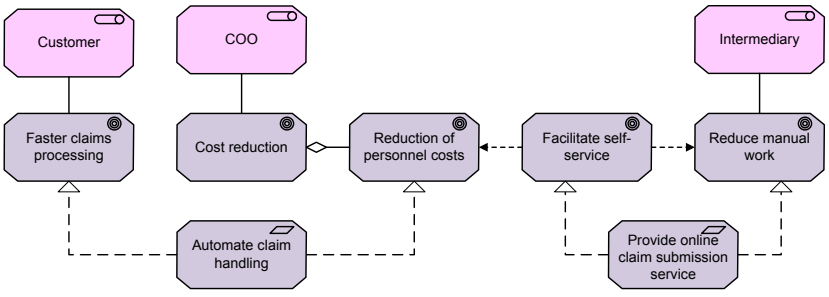

Fig. 23. A business requirements model for AgiSurance.

4.5.3 Interaction Modelling

In the interaction aspect we design and model the way in which the various parts of the enterprise interact with customers, partners, and each other to deliver the service. Such interactions can be specified at various levels, ranging from an identification of collaborations and channels to the detailed design of user interfaces.

Although the scope of interaction design for enterprise services is much broader, involving for example also physical channel design, many insights can be gained from the more established discipline of user interface engineering for software applications. In fact, user interfaces are a kind of service interface. User interface engineering involves human factors engineering, user interface design and graphics design (Nielsen 1993). For each of those subdisciplines, separate developers and designers with different competences are needed. Such perspectives and competences are clearly relevant to service interaction design as well (see also (Dividino et al. 2009)).

In user interface engineering literature, authors have distinguished multiple levels of interaction modelling (Nielsen 1993; Aquino et al. 2008; Calvary et al. 2003; Vanderdonckt 2005; Versendaal 1991):

- task level;
- concept level;
- interface level (abstract (user) interface);
- navigation and presentation level (concrete (user) interface);
- the implemented (user) interface

In our framework, the task level is largely covered by the structure and function aspects, while the concept level is covered by what we call the context model. What remains are abstractions for modelling the actor-actor, actor-system and system-system interactions. Most literature in this area focuses on actor-system interactions, i.e., on how to model human-computer interfaces, e.g., (Vanderdonckt 2005) and (Trætteberg 2009). Actor-actor interactions can be modelled in languages such as ArchiMate and UML, that both support the Collaboration concept.

> *Collaboration*: a (possibly temporary) configuration of two or more Roles (see Structure aspect) that cooperate to jointly perform certain collective behaviour.
>
> *Interface*: a point through which a Role offers access to its Services.
>
> *Interaction Element*: part of a user interface, for example, a window, button, or checkbox. Also non-visible parts of the interface, such as an input event and a command, are interaction elements.

Since our aim is to integrate multiple modelling techniques, we have chosen here for the most abstract definition of the interface concept. It can be used to represent business interfaces or channels, but also to represent user interfaces (screen dia-

logs) and application-to-application interfaces (e.g., WSDL). When we dive deeper into the implementation level, we may also need concepts to model page navigation and presentation, such as page, field, command, and page flow.

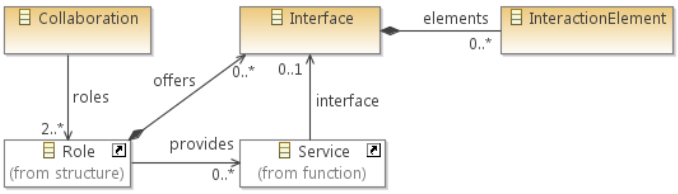

Fig. 24. The interaction metamodel.

In Fig. 24, we present a metamodel for the key concepts of the interaction aspect and their most important relationships to concepts from other aspects, i.e., the Role concept from the Structure aspect and the Service concept from the function aspect.

4.5.4 Structure Modelling

In the structure aspect we design and model the way in which the enterprise is organized internally to deliver its services. This involves the definition of the human, organizational and system actors within the enterprise, as well as the relationships between them. A simple example of a structure model is the organizational chart, depicting the hierarchy of organizational units and positions within an enterprise.

Often, the structural aspects of an organization are covered by models that have a broader scope. Interaction, process, function, and value models usually include a partial specification of the structure of the service system. In the Business Process Modelling Notation (BPMN (OMG 2011c), for example (also see Sect. 0), activities are assigned to pools and lanes representing organizational units, roles and role hierarchies. In ArchiMate, the Open Group standard for enterprise architecture modelling (The Open Group 2012), the structure aspect is covered by the static structure aspect of ArchiMate. This includes business level concepts, such as Business Actor and Business Role, but also application and infrastructure level concepts, such as Application Component, Device and Node.

The structure of a service system can be captured using the following key concepts:

Actor: an entity within the enterprise that can be assigned behaviour and responsibilities, such as a person, organizational unit or application component.

Role: an abstract kind of Actor and a collection of responsibilities and potential behaviours. An Actor can be assigned to a Role, indicating that the Actor will fulfil all responsibilities and behaviours specified by the Role. An Actor may be assigned to multiple Roles; and a Role may be assigned to multiple Actors.

Location: a logical or a physical location relevant to the enterprise (such as branch office, city, or country).

Actors can be related to each other through composition: one actor can contain other actors. Other relationships can be imagined, such as reporting, ownership, or assignment (of a role). In addition, actors can be assigned a location. Fig. 25 illustrate the key structure modelling concepts and their relationships.

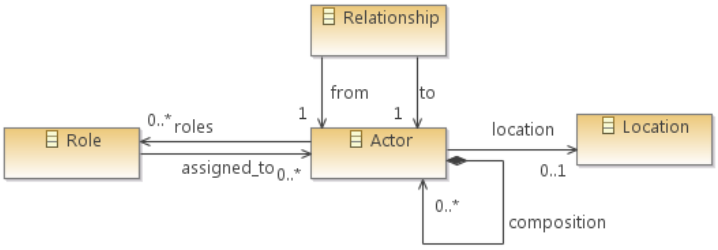

Fig. 25. The structure metamodel.

An example structure model

AgiSurance's corporate business architecture consists of three models: the organization model, the business function architecture model and the high-level business process architecture model. The organization model specifies the organizational structure and the hierarchical relationships between departments (see Fig. 26). In this figure, hierarchy is modelled as containment (nesting).

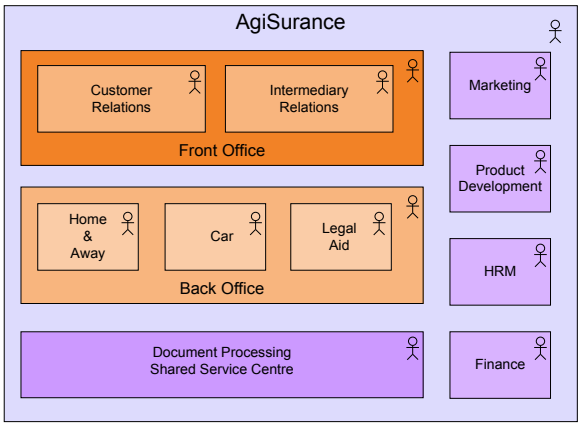

Fig. 26. An organizational structure model for AgiSurance in ArchiMate.

4.5.5 Function Modelling

In the functional aspect we design and model the activities or functional building blocks that are required to deliver the service under development. Together with the coordination aspect, it specifies the behaviour of the service. Where the coordination aspect focuses on the flow (the logical or temporal ordering of activities), the functional aspect focuses on the decomposition of complex behaviour into smaller, manageable and reusable functions, and their interconnection through input/output-relationships. Functional decomposition makes it possible to structure the complexity of organizations and systems. The decomposition gives structure to the tasks and activities in an orderly manner, independent from the executing mechanism.

Functional decomposition, as a technique for describing systems as a hierarchy of functions, is a widely applicable principle. It was first introduced in the area of information systems engineering as the Structured Analysis and Design Technique (SADT) (Marca and McGowan 1987), later formalized by the IDEF0 (Integration Definition for Function Modelling) (IDEF 1981) standard. Data flow diagramming (DFD) (Stevens et al. 1974) is another technique often used in information system analysis, which focuses more on the flows of information between functions (called 'processes' in DFD).

The principle of functional decomposition is also present in service-oriented architectures and business function architectures. Therefore, techniques used to model these, such as SoaML (OMG 2009) and Dynamic Enterprise Modelling (DEM) (Es and Post 1996), are also relevant to the functional aspect.

The e^{3}value methodology models a network of enterprises creating, distributing, and consuming things of economic value (Gordijn and Akkermans 2001). The e^{3}value technique can be used to model value exchanges between enterprises. This may result in a business value model, clearly showing the enterprises and final customers involved and the flow of valuable objects (goods, services, and money). Such models can be used to analyse the economic viability of each enterprise within a service network.

Central to the functional aspect are the concepts of 'function' and 'flow'.

> *Function*: a coherent unit of behaviour with the purpose of performing and/or fulfilling one or more missions or objectives, and identified by a verb or verb phrase that describes what must be accomplished.
>
> *Flow*: a steady, continuous stream or supply of something. Different types of flow may be distinguished, such as information, physical, and value flows.

Functions consume and produce flows, respectively their inputs and outputs. Each function can be decomposed into 'subfunctions', thus creating a hierarchy of functions. Functions are executed by 'mechanisms', which can be automated systems, individuals, a group of people or a combination of systems and people. In the

metamodel, we model this as an assignment to a role (from the structure aspect). The execution of a function and its subfunctions takes place under 'control' of something or someone. This can be a workflow, a set of rules or some other kind of control function that is associated with the function.

Once the functions have been named and defined, we can start to think about the services they realize. We repeat the definition from Chap 1:

> *Service*: a unit of functionality that a system exposes to its environment, while hiding internal operations, which provides a certain value (monetary or otherwise).

The key concepts for the function aspect and their relationships are illustrated in Fig. 27.

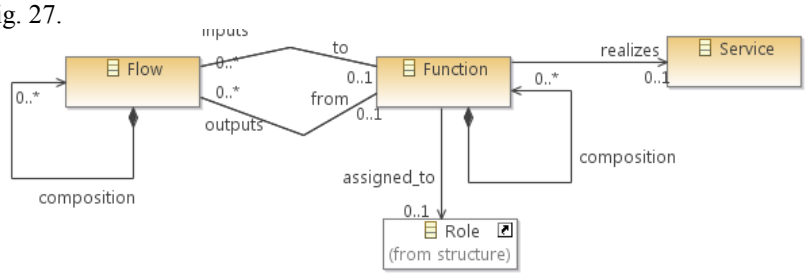

Fig. 27. The function metamodel

An example function model

Fig. 28 shows the business function architecture model for AgiSurance in ArchiMate. It is an example of a model that is mainly concerned with the functional aspect: it defines the functions, their decomposition, and the information flows relating them to each other and to external roles.

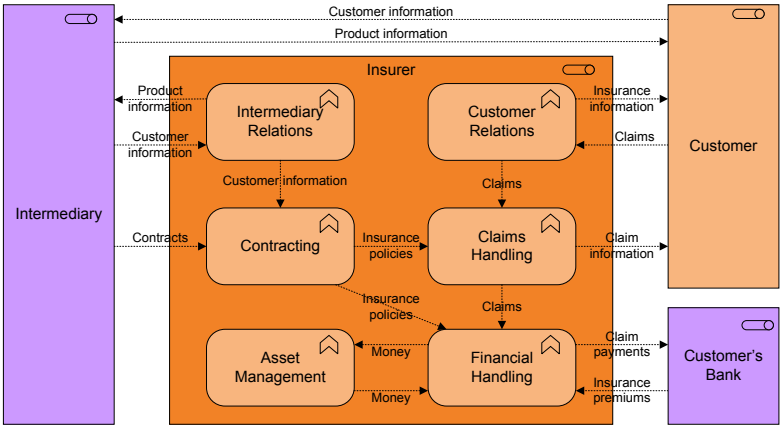

Fig. 28. A business function model for AgiSurance.

4.5.6 Coordination Modelling

In the coordination aspect, we design and model the way in which activities, both automated and human activities, are coordinated to deliver the service under development. It comprises both the coordination of activities within the enterprise and the coordination of the interaction with other organizations, which may be users of the service or partners in delivering it. The literature on service coordination generally distinguishes between a centralized form of coordination, called orchestration, and a decentralized, emergent form of coordination, called choreography (Papazoglou and Heuvel 2007).

Many modelling techniques, both tool-independent and proprietary, are available to model activities and their dependencies. The concepts that are used in these techniques show a lot of overlap, although there are some differences in focus. Some techniques are limited to modelling the processes of a single system or organization, while others explicitly address the interactions between parties.

The Business Process Modelling Notation (BPMN) is a standardized business process notation which is defined and specified by the Object Management Group (OMG 2011c) and has become the *de facto* standard for graphical process modelling. BPMN process models are composed of flow objects such as routing gateways, events, and activity nodes. Activities, commonly referred to as tasks, represent items of work performed by software systems or humans. Routing gateways and events capture the flow of control between activities. The Unified Modeling Language (UML) offers Activity diagrams, to model the flow of activities within a process, and Sequence diagrams, to model the detailed interactions between actors in a specific scenario.

Central in coordination modelling are a set of behaviour elements, that express the way in which an actor (for example, a system, person, or organization) acts in relation to its environment. Typically, behaviour elements can be defined at different levels of granularity. We distinguish two levels of behaviour elements:

Process: a grouping of behaviour based on an ordering of activities. It is intended to produce a defined set of (internal or external) products. A process may be decomposed into more fine-grained (sub)processes.

Activity (or *action*): an atomic behaviour element, performed by a single role within a certain time frame at a certain location. It can represent a function or task (from the function aspect) that is subject to coordination.

Some coordination modelling formalisms explicitly discern collective behaviour of two or more roles:

Interaction: a common behaviour element, carried out by two or more roles, in which each role is responsible for its part in the interaction. A *transaction* is an interaction that is treated as a unit to satisfy a specific request.

In general, a process does not consist of a single sequence of activities. It may contain, for example, branches (choices) or parallel activities. For this purpose, all process modelling languages provide several types of *gateways*:

> *Gateway*: a coordination element that controls the flow of a process, handling the forking, merging and joining of paths within a process.

A process can be influenced by internal or external *events*, which may for example trigger a new process instance or interrupt a running process. A process may also raise events.

> *Event*: something that happens (internally or externally) and triggers a processs or activity.

Finally, some process modelling formalisms explicitly model *states*. Behaviour elements then result in *transitions* between states.

Coordination elements can be related in different ways, depending on the particular modelling technique that is used. We distinguish two main types of relationship: triggering and dependency:

> *Triggering*: a relationship that defines the control flow, i.e., an explicit ordering of activities within a process.
>
> *Dependency*: a relationship that defines how the execution of one activity depends on the completion of other activities or on the availability of certain product items.

The coordination aspect is closely related to other aspects. Processes or activities may be *assigned to* roles from the structure aspect. They may *access* (create/read/write/update) product items, and they may *refer to* decisions or rules.

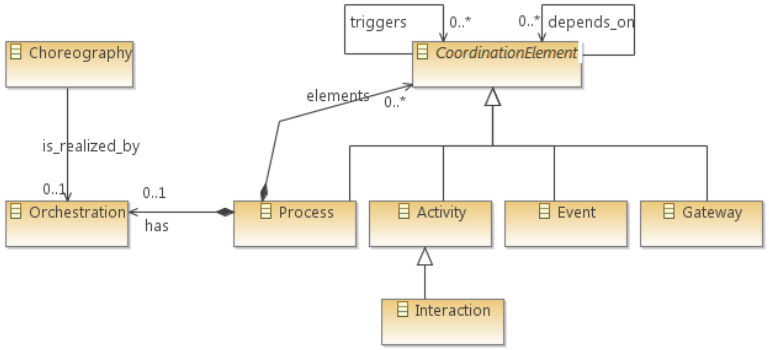

Fig. 29. The coordination metamodel.

In the graphical notation of most process modelling languages there are 'placeholders' for elements from the other aspects. For example, *items* or *data objects* that can be accessed by behaviour elements, or *decision activities* that refer to decisions.

An example coordination model

One of the processes within the claim handling function of AgiSurance is the acceptance process (see Fig. 30). AgiSurance first determines the admissibility of the claim and then the amount of coverage, upon which an acceptance or rejection letter is sent.

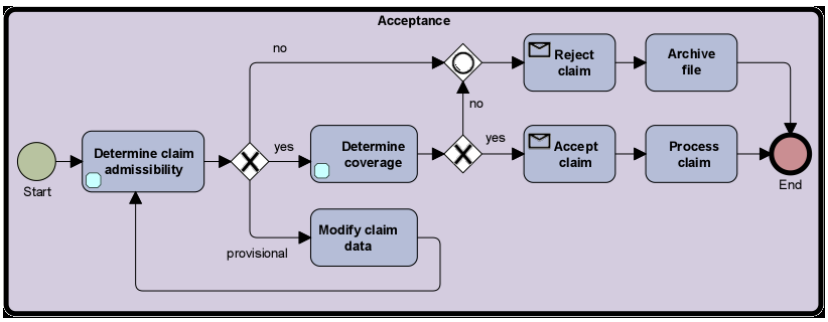

Fig. 30. A coordination model for AgiSurance's acceptance function in BPMN.

4.5.7 Decision Modelling

As mentioned before, the Decision aspect captures the logic of reasoning used in the service domain to reach decisions, i.e., how decisions are (to be) made. In this aspect we therefore design and model the (business) logic to be used by the service under development. A decision is made to determine a conclusion regarding a specific case, based on domain-specific norms (Breuker and Van de Velde 1994). We illustrate the process of deriving a conclusion from domain-specific norms by an example based on the AgiSurance case. Consider, for example, the claim acceptance process at AgiSurance in which the activity 'determine claim admissibility' is executed. First, data is collected from and about the claim and the incidents that are reported. Second, this data is compared to predefined norms defined by AgiSurance. Once the data has been compared, a conclusion is derived.

The decision described is a straightforward operational decision. Other kinds of decision exist, such as strategic/chaotic or strategic/complex decisions. Examples of such decisions are crisis management and merger and acquisition decisions. These kinds of decision are outside the scope of this book; here we focus on operational patterns and fact-based decisions.

Elaborating on the previous paragraph, we further detail a decision by identifying the key concepts it consists of or to which it is closely related. A decision con-

sists of a combination of conditions and conclusions. Both conditions and conclusions are represented by fact types. A fact type is a general classification of a real-world fact, e.g., age, caring criteria, number of accidents and credit rating. Depending on the modelling language used, a specific combination of conclusions and conditions is allowed.

Currently there are multiple techniques within the professional as well as the scientific domain to describe decisions and underlying facts. Six of the most common languages to model decisions are (Zoet and Ravesteyn 2011):

1. if-then sentences;
2. decision tables;
3. decision trees;
4. score cards;
5. event-condition-action rules;
6. event-condition-action-and-alternative rules.

Although the six languages display many similarities, differences exist regarding the underlying concepts as well as the relationships they allow. Nevertheless, Zoet and Ravesteyn show how the languages can be translated to each other.

In addition to the actual modelling languages, an important topic currently emerging is the manageability of decision models. The expert system community long wrestled with this problem, but according to Arnott and Pervan (2005), this research is focusing on the wrong application areas and has no connection with industry anymore. Van Thienen and Snoeck (1993) came to the same conclusion almost a decade earlier, proposing a first solution based on normalization theory. Currently, multiple decision management methods are being developed. A management method that is industry-based is The Decision Model (Halle and Goldberg 2009). A method emerging from the scientific community is described in Zoet and Ravesteyn (2011). We discern the following key concepts for modelling the decision aspect:

Decision: a conclusion reached after consideration of a number of facts and the way in which that conclusion is drawn from those facts.

Fact Type: a general classification of a fact.

Rule Set: a group of statements that defines or constrains a specific aspect of the business.

Rule: a logic statement connecting one or more conclusions to a set of conditions.

Condition: an assertion used as antecedent in a rule.

Conclusion: an assertion used as consequent in a rule.

The key decision modelling concepts are closely related to the context modelling concepts, as we can see in Fig. 31.

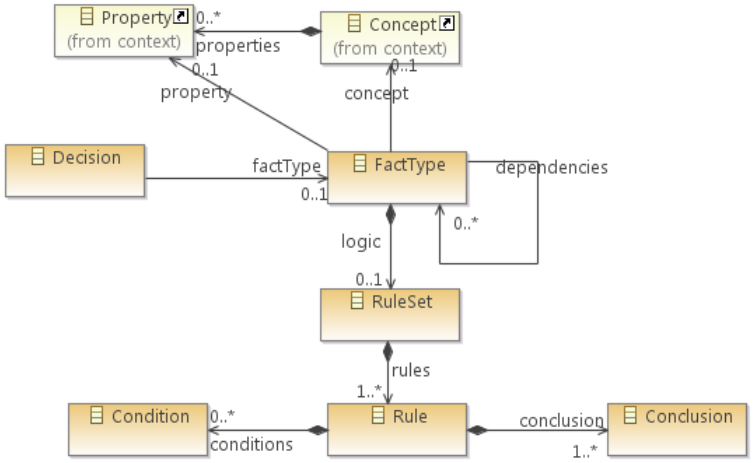

Fig. 31. The decision metamodel.

An example decision model

The acceptance process of AgiSurance contains two decision activities: 'determin claim admissibility' and 'determine coverage'. The first is modelled in Fig. 32 using the Decision Modelling Notation (DMN) from Von Halle and Goldberg (2009). It shows that the claims admissibility depends on the customer's payment behaviour (has he paid his insurance premium) and the correctness of the data in the claim form. In reality there will of course be many more conditions for the acceptability of a claim. The picture only shows the graphical representation of the decision model. Not visible are the decision tables for each of the fact types.

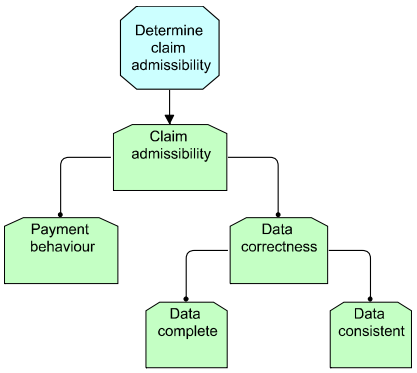

Fig. 32. Decision model for claim admissibility in the Decision Modelling Notation.

4.5.8 Product Modelling

In the product aspect of a service, we design and model the products that are produced and consumed by the service under development. These can be physical products, but more often these will be informational products. Every service uses and manages a certain amount of information. For insurance services this can include information on customers, sold policies and claims received. The product modelling concepts are highly interwoven with almost all other modelling concepts. The interaction modelling concepts display the information products and when modelling decisions the fact type refer to information types.

Product modelling is very similar to context modelling and many of the same techniques can be used, e.g., ERD, ORM, and UML (also see Sect. 4.5.1). A product model is usually more detailed and more concrete than a context model, because it is the basis for implementation in database and message schemas. Therefore, we adopt a more restrictive set of concepts close to those of ER diagrams, with the addition of concepts for modelling physical products:

Product: a thing that is produced or consumed by services. There are two kinds of product: physical *Objects*, such as people or cars, and informational *Items*, such as orders and claims. Often items are used to represent real-world objects in an information system. Products can be composed of other products, their parts.

Entity: a specification of a class of information items. One entity can specialize another, i.e., inherit the more general entity's properties.

Attribute: a property belonging to an entity, e.g., its name, age, length, or amount. Attributes have a type, which defines the values it can take.

Reference: a relationship from one entity to another entity, e.g. a car entity refers to its owner (a person entity).

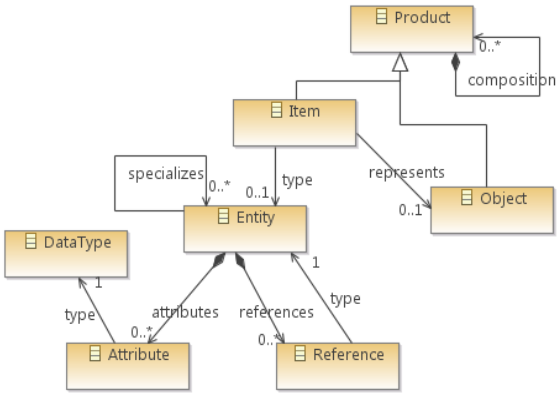

Fig. 33. The product metamodel.

An example product model

Fig. 34 shows a small part of the AgiSurance information model (the entities, their attributes and references) pertaining to the handling of insurance claims. Here a UML class diagram is used to model these information products.

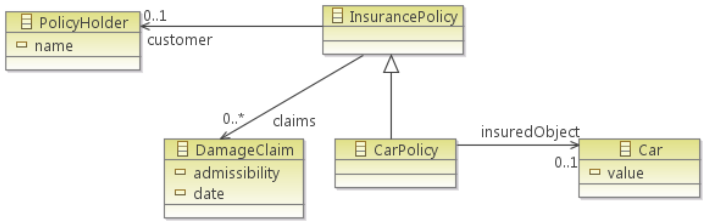

Fig. 34. Fragment of the Product model for AgiSurance.

4.5.9 Integrated Service Metamodel

The analysis of existing modelling techniques above has resulted in the identification of the key concepts for each of the modelling domains in the ASD framework. These concepts are summarized in Table 4.

Table 4. Key concepts for each modelling domain.

Modelling domain	Key concepts
Context	Concept, Property, Relationship, Value Type, Constraint
Requirements	Stakeholder, Goal, Requirement
Interaction	Collaboration, Interface
Organization	Actor, Location, Relationship
Function	Function, Flow, Service
Coordination	Process, Activity, Interaction, Gateway, Event
Decision	Decision, Fact Type, Rule, Rule Set
Product	Product, Object, Item, Entity, Attribute, Reference

Most existing modelling techniques cover more than one of the modelling domains. This helped us in identifying relationships between concepts across the domains. ArchiMate in particular covers many of the identified modelling domains. The ArchiMate core language defines and relates concepts for the interaction, organization, function, coordination and information domains. In version 2.0 (The Open Group 2012), this core is extended with concepts for modelling also the requirements domain. Other languages, such as BPMN (OMG 2011c) and the Decision Model Notation, cover a smaller intersection of the ASD conceptual model, but still identify relationships between their concepts and other languages. In The Decision Model (Halle and Goldberg 2009) the authors clearly specify how

decisions are related to *activities* in BPMN and how *fact types* are related to *concepts* and *properties* in a context model, or, similarly, to *entities* and *attributes* in a product model. Fig. 35 provides an overview of the metamodel for the ASD conceptual model including these relationships. For readability we have left out those concepts that do not have relations with concepts from other aspects.

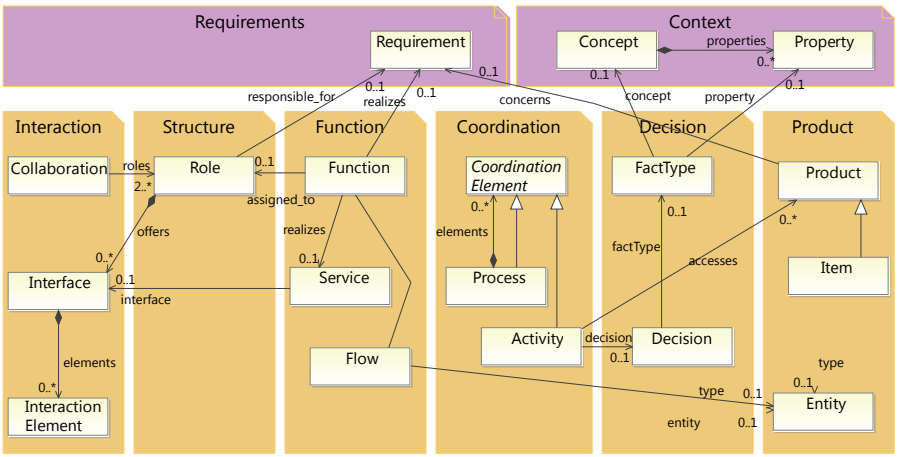

Fig. 35. Integrated metamodel.

4.6 Model Integration

The ASD framework is an instrument to divide a service design into smaller set of more manageable abstractions. However, '... having divided to conquer, we must reunite to rule' (Jackson, 1990). Therefore, the framework must be complemented with a method for integrating the different aspects and abstractions. To this end, we propose to use a metamodel-based approach akin to, for example, the approach suggested by De Lara, Vangheluwe and Alfonseca, 2004. The basic idea is to relate different models via a common, integrated metamodel or ontology, in our case the ASD conceptual model presented above. The conceptual model defines and relates the key concepts for each of the modelling domains. There is no need to create some kind of super metamodel that incorporates all possible concepts for all possible aspects. It is sufficient to focus on the key concepts, because our objective is to support consistency checking and traceability, not to do fully semantics-preserving transformations. We presume the latter to be supported by specific tools, such as BPM suites and model-driven code generators.

The relations between the concepts from the various modelling domains enable us to relate actual models used in a service design. We use the following procedure to do so:

1. First, we define mappings from the used domain-specific modelling languages (DSMLs) to the ASD integrated metamodel.
2. Second, we use the defined mappings to translate each of the models to a corresponding ASD model, i.e., a model that conforms to the ASD integrated metamodel.
3. Third, the resulting models are merged into one integrated model. Model elements that represent the same real world object or phenomenon are matched and merged into one model element. For example, when a process model refers to a particular actor and the organizational model contains an actor with the same name, then these actors are candidates for being merged. Model merging can be done in a naïve name-matching manner, possibly augmented with the help of a thesaurus to match synonyms, but it can also be based on more elaborate semantic matching algorithms. In any case, it is sensible to make this an interactive, user-controlled process.
4. Finally, the integrated model can be used to query for the existence and consistency of relations between elements from different aspect models.

Let us illustrate this procedure using the AgiSurance case study. In Sect. 4.4.4, we already introduced the various models that AgiSurance made for redeveloping its claim handling service (see Fig. 19). Subsequently, we showed parts of these models in Sect. 4.5, when we introduced the conceptual model. Different modelling languages were used: ArchiMate and its motivation extension, BPMN, the Decision Modeling Notation (DMN), and UML. The first step now is to map the used modelling languages onto the ASD integrated metamodel. These mappings are summarized in Table 5 below. For the sake of simplicity, we present only those parts of the mappings that are relevant to the case study.

Table 5. Language mappings.

ASD Concept	ArchiMate	BPMN	DMN	UML
Role	Business Role	Pool/Lane		
Function	Business Function			
Flow	Flow-relation			
Process	Business Process	(Sub)Process		
Activity	Business Activity	Activity		
Gateway	Junction	Gateway		
Decision			Decision	
Fact Type			Fact Type	
Rule Set			Rule Family	
Concept/Entity	(Business) Object			Class
Property/Attribute				Attribute

A Business Role in ArchiMate, and Pools and Lanes from BPMN are mapped to the Role concept. Processes, Activities and Gateways also occur in both these lan-

guages and are mapped to their corresponding concept from the Coordination aspect. The Decision aspect concepts, however, only occur in the Decision Modelling Notation in this case study. UML was used for context and product models.

Next, we translate the given ArchiMate, BPMN, Decision and UML models using these mappings (step 2) and integrate the resulting models (step 3). The integrated model we obtained is illustrated in Fig. 36. We use the UML Object diagram notation, with rectangular boxes representing the model elements. The boxes are labelled with the name of the element and followed by their type (the name of the corresponding metamodel concept). Due to space constraints, the figure only shows an excerpt of the model, highlighting the elements related to the determination of the admissibility of an insurance claim.

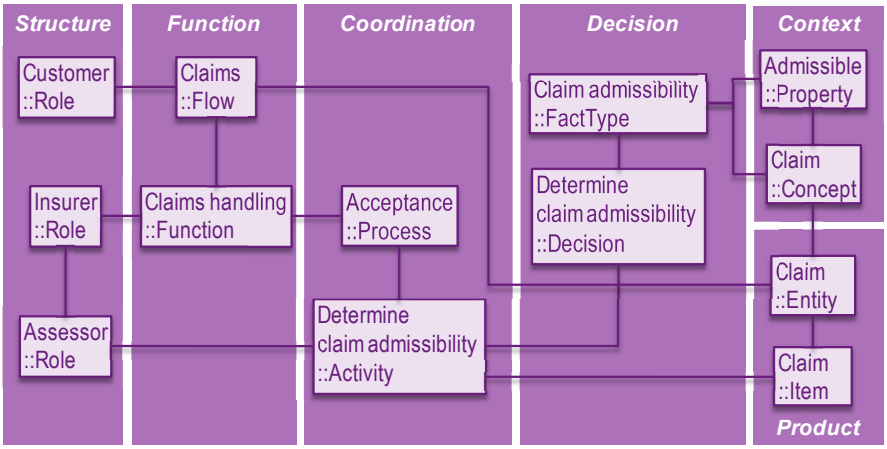

Fig. 36. Part of the integrated model for AgiSurance.

The model should be read as follows: Three roles have been defined in the structure domain as part of the business function architecture (see Fig. 28): 'Customer', 'Insurer' and 'Assessor'. The Assessor role is contained within the Insurer role (not depicted in Fig. 36). From the business function architecture, we can also derive the following elements in the function domain: a 'Claims handling' function, which is the responsibility of the Insurer role, and a 'Claims' flow from the 'Customer' role to the 'Claims handling' function. Within the Coordination domain, a process called 'Acceptance' has been defined which belongs to the Claims handling function. This 'Acceptance' process contains an activity 'Determine claim admissibility', which has been assigned to the 'Assessor' role. The latter activity refers to a decision called 'Determine claim admissibility' in the Decision domain. The corresponding 'Claim admissibility' fact type refers in turn to the 'Claim' concept and its 'Admissible' property from the context domain. In the product domain, we also find a 'Claim' entity and a 'Claim' information item, which are associated with the corresponding context model concept.

4.7 Requirements for Tool Support

The model integration approach presented in the previous sections of course poses important requirements to service development tools and operational infrastructures. As we already described in Sect. 4.3, we can distinguish different levels of adoption of modelling. Up to level 3, 'isolated formal models', no additional functionality is needed beyond what individual modelling tools already offer. For the two highest levels, however, more is required.

At level 4b, 'vertically integrated formal models', it should be possible to relate models from different abstraction levels. This means that the relations from requirements via design and implementation down to the operational infrastructure, and vice versa, can be traced, and that models are used to configure individual infrastructure elements. An example of this would be the use of a business process management engine that is configured with BPMN 2.0 (OMG 2011c) models, which in turn are related to more abstract architecture and requirements models.

This type of functionality is already offered by many integrated tool suites. However, if the upper-level models are designed in different tools than the lower-level models and the execution environment, a clear interface between these levels needs to be established. Existing standards such as XMI (ISO/IEC 2008) can be used to specify the necessary interchange formats and many modelling tools support this (although they are often better at importing than at exporting models, for obvious reasons). Usually, however, this is a unidirectional transformation, down towards the implementation and infrastructure; if we also want traceability back up the chain, a feedback mechanism needs to be implemented. Such traceability is currently offered only by single-vendor, integrated tool solutions.

At level 4a, 'horizontally integrated formal models', we want to be able to relate and integrate models from different aspects. This helps in ensuring consistency and coherence between these aspects and allows for various kinds of analyses of these models, as explained before. This integration implies that we must relate elements in different models with each other, and hence that we need to 'address' such elements.

One approach to this is the use of a single tool environment that 'owns' the various models covering different aspects. This is the approach taken by many business- and architecture-oriented modelling solutions, such as Be Informed Studio, Aquima, BiZZdesign Architect and IBM Rational System Architect.

However, at the lower levels of abstraction, different types of models are often managed by different tools; for example, process models are tied to BPM suites, business rule models to BRM tools and engines, class and object models to software development environments, et cetera. This implies that these tools need to talk to each other, or at least to some common environment that links them together. In that case, each model element would ideally have a globally unique identifier that any tool can use as a reference, even if that particular element is in a model managed by a different tool. However, at the moment the only realistic solution is

to store these models in a single shared repository, on which these different tools operate. This is the route taken by most vendors of larger modelling and requirements management tool suites.

Unfortunately, such repositories are often not open to third-party tools. Although many repositories are based on standards such as EMF (Steinberg et al. 2008) or MOF (OMG 2011b), this is not enough. We also need open, standardized interfaces and (semantic) standards for relating concepts, based on a clear metamodel such as the one presented in the previous sections.

To demonstrate the feasibility of this integration approach, we have already connected different tools in both the horizontal and vertical direction. BPMN models created with the business process tool BiZZdesigner (from BiZZdesign) were related with the context and product models from the knowledge modelling solution of RuleManagement Group, via decision models in the Decision Modelling Notation. In the previous section, we already showed part of the integrated AgiSurance model that was developed in this case study (**Fig. 36**).

The hardest part, however, may turn out to be the integration between the different elements and platforms at the infrastructure level. The complexity of integrating various components, for example via an enterprise service bus, should not be underestimated. Not only does this integration require clarity about the functionality and semantics of each of these elements, but it must also conform to various non-functional requirements. If high or bursty volumes must be processed, for example, the performance may become a bottleneck. The complexity of such a landscape may require extensive proofs-of-concept. Models may also be of help here, for example to perform quantitative analyses or simulations. For more on such analyses, see e.g. (Lankhorst et al. 2009, Chap. 8).