DEPARTMENT OF COMPUTER SCIENCE
SERIES OF PUBLICATIONS C
REPORT C-2009-9

Modelling framework for interoperability management in collaborative computing environments

Toni Ruokolainen

UNIVERSITY OF HELSINKI
FINLAND

Modelling framework for interoperability management in collaborative computing environments

Toni Ruokolainen

Department of Computer Science
P.O. Box 26, FIN-00014 University of Helsinki, Finland
Toni.Ruokolainen@cs.Helsinki.FI

Technical report, Series of Publications C, Report C-2009-9
Helsinki, June 2009, iv + 159 pages

Abstract

Modern inter-enterprise computing imposes demanding requirements on the concepts used for business service modeling and development, and operational management facilities targeted towards establishing the corresponding inter-enterprise collaborations. Current business models based on highly specialised core competencies and outsourcing of secondary functionality stress the openness and agility of the business support infrastructure. On the other hand, the legacy systems, customs and business models used in individual enterprises determine in many respects the properties and capabilities of business services that are offered to the clientele. As a result, the individual components of an electronic inter-enterprise collaboration are typically heterogeneous both technically and semantically. More over, each of the components reside in an autonomic administration domain and their operation is governed by the holder's own interests. This autonomy further increases the level of heterogeneity encountered and introduces dynamism to the system.

Because of the inherent properties of inter-enterprise collaborations, establishing interoperation becomes very problematic. While in the past approaches interoperation has been achieved using tightly controlled integration approaches techniques, such as application level integration or model unification, that hinder the openness and agility of the system are not feasible anymore. Instead, interoperation should be established using a federated approaches where collaboration models are established dynamically with utilization of interoperability facilities and mechanisms for electronic contracting. Such a federated interoperability management demands for rigorous models defining the different aspects of collaboration and infrastructure services for validating and maintaining interoperability.

This thesis provides a study and conceptualization of collaborative software systems and a framework for interoperability management. The conceptualization is provided as a set of metamodels that define the concepts and relationships of collaborative systems. A metamodel for a special kind of collaborative system, namely federated service communities, is then constructed. The concepts of federated service communities are constructed on the basis of the requirements for modern inter-enterprise collaborative computing. The foundations for establishing interoperable federated service communities are laid on a formal service typing discipline that provides for methods validating business service interoperability. In addition to the conceptual and theoretical elaboration of collaborative systems and interoperability, a design for service type management infrastructure is described. Service types and the corresponding metainformation management infrastructure developed as part of this thesis provide foundations for delivering service trading facilities for open service markets.

Computing Reviews (1998) Categories and Subject Descriptors:
D.2.1 Requirements / Specifications: Methodologies
D.2.11 Software Architectures: Domain-specific Architectures
D.2.12 Interoperability
I.6.5 Model Development

General Terms:

Additional Key Words and Phrases:
service-oriented software engineering, service ecosystems, domain ontology, metamodelling, linguistic metamodel, ontological metamodel

Contents

Chapter 1

Introduction

Modern networked enterprises demand flexibility and openness from computing systems to tolerate changes in ever-changing technology and business domains, and to gain a competitive edge. Such requirements, however, make it difficult to achieve and maintain interoperability between the distinct enterprise-information systems owned by partners willing to collaborate. This thesis discusses facilitating concepts and mechanisms for establishing inter-enterprise collaborative computing. A multi-disciplinary study of interoperability that spans modern middleware platforms and open distributed computing, as well as forward-looking disciplines for software-engineering is presented. The contributions of this thesis include a rigorous conceptualization of collaborative systems and so-called federated service communities. The resulting concepts are formalized as metamodels which provide means for facilitating interoperable collaborations in open collaborative systems. The metamodels enable sharing and management of interoperability and engineering knowledge within the corresponding service ecosystems. A formalization of interoperability that is laid on the foundations of service typing is utilized for establishing dependable business service delivery.

This introductory chapter provides motivations and context for the rest of the thesis. In Section 1.1 the primary challenges confronted in modern inter-enterprise computing are discussed. Modern model-oriented software engineering principles giving a frame of reference and realization environment for collaborative systems are presented in Section 1.2. After that, the concept of collaborative computing and the special characteristics of interoperability it involves are introduced in Section 1.3. Finally, the contributions and structure of the thesis are elaborated in Section 1.4.

1.1 Challenges of modern inter-enterprise computing

Inter-enterprise computing imposes demanding requirements on the concepts used for modelling and development of business services, and for the operational management facilities targeted towards establishing inter-enterprise collaborations. Openness and agility of the business support infrastructure are the most prominent of these requirements and are stressed by the current business models based on highly specialised core competencies and outsourcing of secondary functionality.

Openness of an enterprise computing environment is manifested on one hand by low degree of dependence and coupling between business partners and their information systems, and on the other hand by the absence of superfluous (technical) constraints for joining and establishing business collaborations. In the best scenario, the environment for facilitating inter-enterprise electronic business comprises an “open service market” where market makers, service providers and

clientele can interconnect to establish business collaborations on-demand. This kind of openness enhances possibilities for exploiting business opportunities as a reaction to market changes. More over, openness of the collaboration environment enables opportunistic formation of collaborations between enterprises acting in different business domains or introduction of new kinds of business collaboration networks.

Agility becomes very important in the context of rapidly changing networked business for tolerating and managing the ever-changing requirements in technology and business domains. Agility is a property of individual information systems and their administration domains which refers to the pace and lightness of the processes used for providing new kinds of solutions, and for managing and configuring the existing ones. While agility is needed in all sizes of enterprises, it offers competitive edge especially for small and medium size enterprises (SMEs) and may constitute an important factor in their business models and competitiveness. Agility of operation is required from an enterprise information system (EIS) throughout its whole life-cycle. During development of new business services, the developers must be provided with tools that support agile development methods. Functionality such as generation of business logic implementations from high-level models, generation of automated test-cases, or prototyping and simulation mechanisms that are applicable in the early stages of business service development are needed. Agile facilities are also required for maintenance of enterprise information systems such that it would be relatively easy to configure the systems for the requirements of different kinds of collaboration scenarios.

Attaining flexible and effective networked business necessitates methods and technologies that align enterprises' business strategies and goals with the underlying information technology. Evidently there is a semantic gap between any real-life problem domain and a corresponding technological solution. In the context of enterprise computing this semantic gap is even more evident. The business vocabulary uses high-level concepts such as "partner", "contract" or "business rule" to describe how business is made. Mapping these concepts into a computing infrastructure terminology and implementations is a challenging problem that requires knowledge of both conceptual domains. Use of domain specific vocabulary for describing enterprise computing systems and business services helps to bridge this semantic gap between business and technology. The vision is, that when equipped with suitable modelling concepts, tool-chains and computing platforms, the enterprise computing environment can be configured by even non-technical experts such as business analysts. Given such facilities, the reactivity of an enterprise computing environment with respect to business changes and new business opportunities can be dramatically improved.

One of the most characterizing features and source of complications of modern inter-enterprise computing environments is the autonomy of the participants. Services used for realising collaborations reside inside autonomous administration domains and their properties are governed by participant's own interests and policies. The legacy systems, as well as the business customs and models used in individual enterprises determine in many respects the properties and capabilities of business services that are offered to the clientele. Autonomy manifests itself as different degrees and quality of self-governance with respect to design and implementation of business services, willingness to collaborate, and decisions concerning the operation and maintenance of the business services. Autonomy has severe implications on the nature of interoperation in collaborative systems and imposes prerequisites for the kinds of facilities needed from the operational environment. Before a collaborative system can be conceptualised, the implications stemming from the autonomic nature of the collaboration environment have to be identified.

Since participants of a collaborative system have the freedom to design and implement their computational services as they wish, technological heterogeneity must be tolerated. Some of the technological heterogeneity can be dealt with quite easily using standardised communication protocols and middleware platforms. However, freedom of design inflicts also technical heterogeneity

related with the selection and configuration of collaboration capabilities, such as properties related to security, trust, and quality of service.

Autonomy inflicts also semantic heterogeneity between enterprise information systems which manifests itself as conflicts related to interpretation of business document contents and service behaviour. Some kind of means for expressing the semantics of business documents and services is needed to give accurate interpretations for services and to validate compatibility of semantics during interoperation. Standardised taxonomies (e.g. NAICS [171]), vocabularies (e.g. ebXML [133]) and ontologies (defined for example with OWL [259]) are utilised to overcome semantic heterogeneity related to static business artifacts. In addition to the static semantics of business documents, the dynamic semantics of behaviour has also to be considered. For this purpose, different kinds of behaviour description languages are used to attach a definition of service behaviour into business service descriptions. For expressing the behaviour of service method invocations, logical characterisations of method pre- and post-conditions, and effects can be attached to their descriptions, as in OWL-S [188] for example. Description languages such as WS-BPEL (Web Service Business Process Execution Language) [242] can be used to express process-like behaviour of services.

Autonomy of individual enterprises causes heterogeneity and dynamism into inter-enterprise collaboration environments which again cause severe interoperability problems when collaborations should be established. *Interoperability* means the capability of (enterprise information) systems to collaborate in such a fashion that eventually either their mutual goals become fulfilled or their co-operation is dissolved in a controllable manner in case of an error. Interoperability problems range from simple technological incompatibilities to conflicts between business strategies. Interoperability can be characterized as having technical, semantic and pragmatic concerns [141]. *Technical interoperability* means that the technological facilities underlying the applications and business services are compatible such that for example communication paths can be established. *Semantic interoperability* deals with the meaning of exchanged information and message exchange patterns. Finally, *pragmatic interoperability* is achieved if the intentions, business rules, and organizational policies of collaborating parties are compatible with each other. As part of the contributions of this thesis, a multi-disciplinary study of interoperability addressing especially the semantic and pragmatic aspects is given. The results of these studies contribute to the metamodels developed for management of collaborative systems.

1.2 Model-orientation in distributed systems engineering

The challenges of modern distributed computing and software engineering therein have been pushing the development in these areas towards model-oriented and description-centric approaches. The disciplines of service-oriented computing (SOC) [191, 231] and model-driven engineering (MDE) [225] are now widely recognised as efficient means for designing, developing and maintaining complex distributed systems. The former provides a framework for loosely coupled interactions in open distributed computing environments while the latter provides efficient software engineering methodologies for development of business services. The disciplines are complementary to each other and rely extensively on explicit and formal domain-specific models and languages defining the relevant properties about the universe of discourse.

Service-oriented computing is a paradigm for designing and implementing complex distributed systems [191, 231] that is based on the concept of services. A service is considered as a well-defined and demarcated unit of operation that delivers some abstract behaviour meaningful for its potential clientele. The SOC-paradigm comprises four conceptual elements: services, service

descriptions, service composition, and service-oriented architectures (SOA). Services are advertised by publishing their descriptions in service brokers, which form an essential part of the SOA. Service descriptions are produced by service providers and they characterize the properties and capabilities of corresponding services. Service consumers use service discovery mechanisms provided by a service broker infrastructure to locate appropriate services.

The movement towards service-centric model in enterprise computing environments has already been witnessed: the Enterprise Service Bus (ESB) [226] is a recent technological framework that emphasizes the use of service-oriented architectures and service-oriented computing as the basis for enterprise computing. Adoption of service-orientation in the context of enterprise information systems is motivated by isolation between enterprise's business models and underlying technology. As a result, more flexible EIS with increased return-on-invest can be achieved. In addition, necessary changes due to evolution of EIS technology or business models can be introduced to the system incrementally and in a controllable manner.

Model-driven engineering (MDE) is a software engineering discipline which considers models as first-class entities and primary objects of engineering. Whereas in archetypical software engineering practices models, such as UML diagrams [182], are used informally and only for documentary purposes, MDE emphasizes the importance of formal machine-processable models. The main challenges that MDE approach pursues to respond are induced by the complexity and evolution of computing platforms as well as complexity of system integration and configuration [225]. Domain specific languages, model transformations, and code generation are utilised to bridge the semantic gap between problem domains and technology, and to efficiently produce software artifacts and systems that are "correct-by-construction" [225].

More recently the specific requirements of service-oriented computing have been addressed also from the software engineering perspective. Some preliminary research has been conducted in the emerging area of service-oriented software (system) engineering (SOSE) [248, 236] which utilises constructs and concepts conforming with the service-oriented computing paradigm [231, 191] for designing, modelling, developing and managing open distributed computing systems. Due to the inherent nature of services and service-oriented computing, the development process is highly description centric. In the design and implementation phase, the principles of MDE are applied but in addition, the meta-information describing properties of the system is utilised extensively also later in the system life-cycle. Service discovery facilities and service composition are used as the primary means for delivering the system functionality instead of traditional design and coding [248].

On the basis of recent developments it can be argued that the role of models and meta-information in general is changing and their importance is growing as part of software development processes and open distributed systems. Whereas in the traditional object and component oriented software engineering disciplines models were typically serving as documentary artifacts, the emergence of model-driven engineering has elevated their importance. In the prevailing practise of MDE the model information is only applied during the development phase and primarily for code generation; models do not have a role during the operation of the system. The SOC and SOSE disciplines however extend the life-time of models from the development time all the way to the operational time. The models are used for defining the properties and capabilities of services and service-oriented collaborations.

Accompanying the model-orientation and emergence of service-oriented software engineering, an evident convergence is happening on one the hand between the different phases of software-engineering processes (i.e. analysis, design, implementation, deployment), and on the other hand between the development and operational time facilities. The same meta-information repositories and models representing services can be utilised equally by development tools for service-oriented

software engineering and infrastructure facilities for service-oriented computing, for example. To fully utilize the potential of this trend, complete tool-chains with consistent concepts and semantics for service-oriented software engineering are required. This thesis provides a contribution towards this objective in form of metamodels and repository designs.

1.3 Collaborative and interoperable computing

Recently, the notion of collaboration has been highlighted as an ambitious objective for distributed computing environments. Whereas the previous generations of distributed systems have been based on a quite closed and specialised solutions, such as integrated application frameworks, collaborative computing systems are targeted towards providing interoperation and collaboration platforms for open environments. Based on the idea of the peer equality and shared responsibilities, collaborative systems provide computing environments for autonomous, active and heterogeneous components [231]. Modern collaborative computing can be based on the notion of collaboration, service-oriented computing paradigm, principles of model-driven engineering, and explicit interoperability management. While the frameworks of service-oriented computing and model-driven engineering were discussed in the previous section, the concepts of collaboration and interoperability management are elaborated briefly below.

Collaboration is a process of shared creation among a group of entities which share information, resources, responsibilities and rewards to achieve a common goal [46]. Collaboration transcends the notion of co-operation. In co-operation the participants of a community are involved with each other only to work together for each entity's own benefit or to serve a "public good". As opposed to real collaboration, there is no added-value in co-operation: the individual's work contributes only as a part of a larger assignment. The role and meaning of co-operation is usually implicit and functionally irrelevant to an individual entity. Co-operation is found typically in scenarios that contain subcontracting or resource sharing, such as in the context of supply-chains or Grid-computing [81]. However, in collaboration the work of an individual entity constitutes a part of an assignment with an added value. The entities in collaboration share an objective which cannot be achieved without the corresponding form of collaboration. In such cases, collaboration may even constitute a necessity for the existence or functioning of the entities.

To reach interoperation in a collaborative computing environment, the contents and intention of the collaboration must first become unambiguously understood among all the participants. There are different means to establish this understanding and they possess their characteristic properties with respect to the level and quality of tolerating heterogeneity, autonomy and dynamism. In current software development frameworks and operational platforms for inter-enterprise computing, interoperation is achieved using tightly controlled and constraining techniques such as application level integration or model unification. However, these kind of techniques are not feasible for fulfilling the promise of electronic commerce, since they do not provide the kind of openness and agility needed. Instead, interoperation should be established using approaches where collaboration models are established dynamically and on-demand with utilization of generic interoperability facilities and mechanisms for electronic contracting. Such an interoperability management framework demands for rigorous models and theory defining the different aspects of collaboration, and infrastructure services for validating and maintaining interoperability. The metamodels and metainformation repository designs developed in this thesis are addressing especially the challenge of interoperability management.

1.4 Thesis contributions and structure

This thesis provides a study and conceptualization of collaborative systems and introduces a framework for interoperability management. The conceptualization is provided as a set of metamodels that define the concepts and relationships of collaborative systems.

The importance of such formal conceptualization becomes evident when considering its usage scenarios. First of all, the metamodels provide means for sharing and maintaining *interoperability knowledge* within an open collaborative system during its life-cycle. We categorize the means for achieving interoperation to three approaches, namely integration, unification and federation [141]. Since integration and unification approaches do not preserve all relevant information about interoperation [216], features related to interoperation can not in these approaches be considered dynamically. In the federated approach interoperability is achieved by utilizing a shared metamodel and making the relevant information about features affecting interoperation explicit. This information may comprise different kinds of artifacts, such as prescriptive models, ontologies or service interface descriptions.

We call the set of artifacts conforming to the shared metamodel as interoperability knowledge, as its primary purpose is to provide sufficient grounds for establishing interoperable collaborations by declaring consistency and conformance criteria for knowledge elements. Each collaboration participant may have their own representations of the artifacts describing their business domain and services. The infrastructure services provided by the operational environment of a collaborative computing system are used for maintaining the consistency of interoperability knowledge, and for instrumenting collaboration establishment processes based on this knowledge.

Secondly, the metamodels introduced in this thesis can be used to facilitate sharing of *engineering knowledge*. Services, service-based systems and other artefacts instrumenting service-based collaborations are likely to be produced in open service ecosystems by globalised software engineering processes. However, in such *global software engineering* (see for example [59]) communities, the use of heterogeneous and varying modelling notations also become a hinder for exchanging modelling knowledge and conventions between partners. For enabling collaborative and distributed software engineering activities, information regarding the corresponding processes and engineering artefacts need to be shared among the participants of the engineering domain; the metamodels presented in this thesis provide means for unifying such knowledge. While not in the focus of this thesis, such usage of the metamodels has been kept in mind while designing the metamodels.

The general concepts of collaborative computing systems are formalized in a set of metamodels. These top-level metamodels are then extended for characterizing concepts in more specific systems. For providing a preliminary validation case for the soundness of the top-level metamodels we construct a metamodel for so-called federated service communities. The concepts of federated service communities are constructed on the basis of the requirements for modern inter-enterprise collaborative computing. The conceptualization is based on previous research in the context of the Pilarcos framework [143, 141]. The foundations for establishing interoperable federated service communities are laid on a formal service typing discipline that provides for methods validating business service interoperability. In addition to the conceptual and theoretical elaboration of collaborative systems and interoperability, a design for service type management infrastructure is described. Service types and the corresponding metainformation management infrastructure developed as part of this thesis provide foundations for delivering service trading facilities for open service markets.

The structure of this thesis is as follows. Chapter 2 introduces collaborative software systems in a generic level. First the characteristics of collaborative software systems are presented.

The concept of collaborative systems is elaborated by introducing the related engineering methods and conceptual frameworks that are considered as foundational for the development of such systems. The special characteristics of interoperation in the context of inter-enterprise computing are identified in Chapter 3. We develop a modelling framework that provides the foundations for establishing collaborative computing domain ontologies in Chapter 4. Based on the premises given by the previously mentioned chapters, a metamodel and domain-ontology for collaborative systems is defined in Chapter 5. Finally, in Chapter 6 we introduce the domain ontology and corresponding metamodels describing federated service communities. We also give a preliminary description for the implementation of the necessary knowledge repository infrastructure that is needed to establish federated service communities. Also foundations for validating and managing interoperability in this context is provided. The infrastructure facilities needed for interoperable service delivery in federated service communities are part of the Pilarcos interoperability middleware [141]. These infrastructure facilities comprise a distributed metainformation management framework that is utilised for business service trading and for dynamic establishment of business communities. Finally, a discussion of the applicability of the approach, theoretical and pragmatic issues not addressed in this thesis, and future research directions are given in Chapter 7.

Chapter 2

Collaborative computing

Collaborative computing is a continuation in the evolution of information systems from centralized mainframe solutions to client-server systems and peer-to-peer environments. Whereas the previous generations of information systems were composed out of homogeneous components and constituted closed and tightly integrated systems, the components of collaborative computing environments are heterogeneous, loosely coupled and subject to their owners' autonomic intentions. In distinction to traditional client-server computing, the communication paradigm is typically based on loosely coupled messaging instead of the remote procedure calls (RPC) [34] prevalent in the client-server systems. Instead of predetermined and static bindings, interoperation between the components of a system are governed by dynamically negotiated service-level agreements and collaboration contracts. Based on the similar idea of the peer equality and shared responsibilities of peer-to-peer systems [235], collaborative computing environments provide a cooperation platform for autonomous, active and heterogeneous components [231].

This Chapter defines the essential concepts of collaborative computing environments, and introduces engineering methodologies and frameworks useful for realizing such systems. Collaborative computing environments are characterized in Section 2.1 where we identify the essential properties of such environments and provide a comparison between a selection of collaborative computing systems. After that in Section 2.2.1 we elaborate on the notion notion of loose coupling and how it can be achieved in collaborative computing environments. Finally, Section 2.3 introduces conceptual and technological frameworks suitable for realising collaborative computing environments.

2.1 Characterizing collaborative computing environments

A collaborative computing environment is a distributed computing system comprised of autonomous *collaboration agents* and a set of infrastructure services provided by the *operational environment*. A collaboration agent provides a technical representation of the entity, such as an individual or an organization, willing to collaborate and mediates their activities and decisions towards other participants. Collaboration agents are active during collaboration establishment process and actual operation of the collaboration. The agents are loosely coupled and interoperate with each other to meet some mutual objective. The collaboration is facilitated by a set of infrastructure services provided by the operational environment for establishing and controlling the collaborations.

The mutual objective of a collaboration defines the results expected from successful fulfillment of the corresponding collaborative activities. The objective might be as concrete as delivery of items through a supply-chain from sub-contractors to the client, or more abstract like delivery

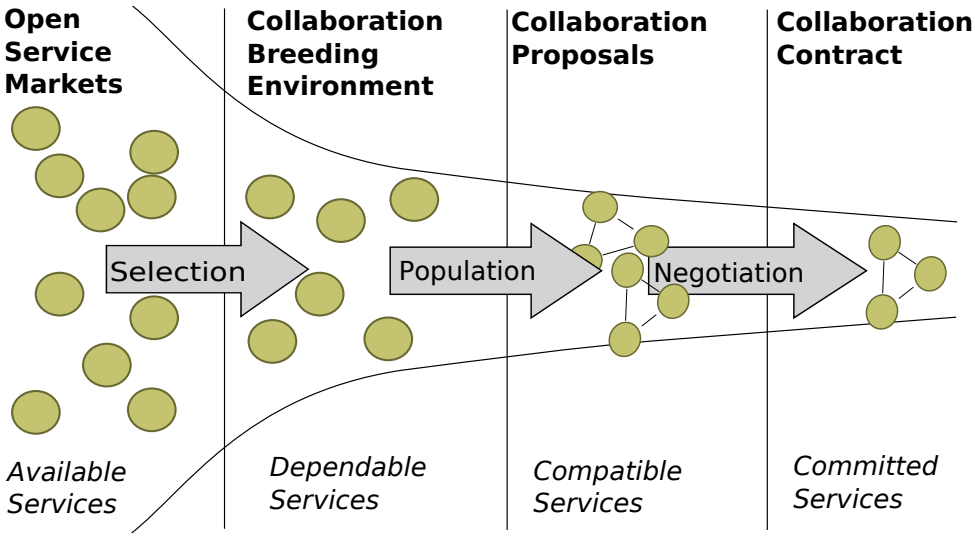

Figure 2.1: Illustrating the process of collaboration establishment.

of some conceptual service towards clients through the underlying collaboration. As the collaboration objective can be very high-level and abstract description of the collaboration intention, the objective is usually defined by a set of explicit *collaboration goals* which prescribe concrete steps or tasks that are need to be performed for realizing the objective. In a sense, the set of goals gives an operational interpretation for the collaboration objective which can sometimes be very difficult to put into a well-defined, specific form otherwise.

The infrastructure services contained in a collaborative computing environment deliver functionality for enabling formation of loosely coupled collaborations, and for arbitrating the collaborative actions and commitments between the collaboration agents. During the formation of a collaboration, infrastructure services are used for discovering and selecting appropriate services from an open service market, further refining the selection of services using semantic and pragmatic criteria, and finally establishing a collaboration contract between the set of selected service providers. This process involves service selection, and collaboration population and negotiation phases [141] and is illustrated in Figure 2.1. The distinct phases and the corresponding functionalities required from the operational environment are discussed in the following.

During service selection phase appropriate services are located and selected from the ones available in open service markets. The selection is based on criteria set by the form of collaboration and the requirements set by the initiator of the collaboration. The selection criteria especially consider the suitability of services for the corresponding form of collaboration, that is their potential for fulfilling the collaboration goals. Such criteria address especially the technical, and partly semantic interoperability requirements. In addition, both collaboration itself and its initiator may require certain level of initial trust and reputation from the services and their providers. A collaboration may require, by its very nature, that a service in the specific role of the collaboration can be accepted only from providers fulfilling a certification standard, for example. In addition, a collaboration initiator typically has individual trust and reputation criteria for service providers.

Conceptually there is an invisible and impenetrable border uphold by a “gate keeper” between the open service markets and a collaboration breeding environment. This gate keeper consists of infrastructure services for discovery and selection of services, as well as trust [213] and reputation management [214] mechanisms. The collaboration breeding environment provides means for es-

tablishing certain kinds of collaborations from a set of *dependable* services. The services selected to the breeding environment are dependable in sense that they fulfill the necessary technical and semantic interoperability requirements of the corresponding kind of collaboration and they are provided by trusted partners.

The collaboration breeding environment acts as a catalysis platform for potential collaborations using the set of pre-selected services. During the corresponding collaboration population phase especially the semantic aspects of interoperability are addressed. The population phase produces a set of collaboration proposals from the dependable services and a model characterizing the structure and requirements of the collaboration [141, 139]. These collaboration proposals are further refined using negotiations taken between the collaboration agents. The negotiations result in formulation of a collaboration contract which states the responsibilities for each participating entity, the structure of the collaboration, and the non-functional features expected from the corresponding business services and cooperation facilities. The collaboration contract is then used for managing the operation of the collaboration [165].

The phases of the collaboration establishment process illustrated in Figure 2.1 can be paralleled with the degrees of interoperability, technical, semantic and pragmatic, introduced already in Section 1.1. The primary purpose of the service selection phase is to guarantee technical and semantic interoperability, that is each service passing the service selection criteria should be at least technologically and behaviourally compatible with the given form of collaboration. Semantic interoperability is addressed in full by the population phase: compatibility between non-functional property metrics and service usage policies should be addressed at this stage, for example. Finally, the pragmatic aspects of interoperability, such as expression of the agents' willingness to collaborate, are considered during the negotiations.

In the following Sections we discuss in more detail the characteristics of collaborative computing environments. In Section 2.1.1 we discuss the general properties of operational environments. The properties of individual collaboration agents are addressed in Section 2.1.2. Finally, different means for establishing interoperation in collaborative computing environments are discussed in Section 2.1.3.

2.1.1 Properties of operational environments

An operational environment for collaborative computing can be characterized by considering the *variability* of available services supported, the *openness* of the collaboration breeding environment, and the level of *autonomy* allowed for the service providers. Variability measures the degree of similarity required from services while still being able to establish collaborations. The type of service variability supported by an operational environment is strongly related with the interoperation model adopted; different means for achieving interoperation are discussed in Section 2.1.3. Openness of the collaboration breeding environment determines the manner in which services and their providers are accepted to participate a certain collaboration. Finally, the level of autonomy allowed for service providers is an important feature when considering the applicability of an operational environment to a certain collaborative computing scenario. The level of autonomy allowed for service providers can be considered from the viewpoints of service implementation, design and involvement.

Variability in the context of collaborative computing environments can be measured with respect to agent *1)* implementation and *2)* interface heterogeneity. At the implementation level variability states if the applications underlying the collaboration agents are expected to be constructed using same unifying technology. Implementation level variability can be tolerated using for example different middleware technologies such as CORBA [178] or Web Services [262],

adapters [270] or wrappers [38].

Interface variability considers the tolerance of structural heterogeneity of the collaboration agent service interfaces that spans the structure of exchanged messages and provided methods. In the archetypical Web Services [262] development scenario where service client stubs are statically generated from the provided WSDL [55] descriptions the interfaces typically need to be homogeneous between the collaborating parties. Interface heterogeneity can be tolerated to some extent by using so-called dynamic invocation interfaces (DII, see for example [178]) where method requests and messages are programmatically generated during runtime based on dynamic introspection of available interface structures. Also more flexible matching criteria between interface structures, such as using structural subtyping (see for example [5, 106]) instead of nominal subtyping, can be utilized for enhancing the tolerance over interface heterogeneity.

The environment openness measures the level of dynamism with respect to the selection of services available for collaboration establishment. The degree of environment openness can be characterized as *1)* closed, *2)* semi-open, or *3)* open. In the closed case, the set of services in the collaboration is static. That is, the phases of service selection and collaboration population are predetermined. In the semi-open case, the set of dependable services located in the breeding environment is static (closed world), but they can be selected dynamically for collaborations from this pre-determined pool of services (open world with respect to the actual collaboration). In this scenario, the service selection phase is predetermined, but collaboration population can be done dynamically.

An open collaboration environment provides most flexibility as the set of agents available for collaborations is not limited. Agents can be introduced to and withdrawn from the collaboration environment as needed and a breeding environment matching the requirements and properties of a specific collaboration is established dynamically from the services available in open service markets. However, this freedom needs to be regulated with feasible infrastructure mechanisms that provide trusted and interoperable computing support.

Autonomy in the context of collaborative computing environments refers to existence of administrative domains and self-governing providers of available services. Autonomy is manifested by the freedom of service providers to decide about the governance of their exported services. Service design, implementation and operation is determined by the local stakeholder intentions and administration policies. Under such presumptions, usability of a service is prescribed by not just technological issues, such as availability and timeliness of service invocations, but also by willingness of the corresponding bearer to take the requested actions and compatibility between the policies of collaborating parties.

2.1.2 Properties of collaboration agents

Collaboration agents are technological artifacts that represent the entities willing to collaborate in a collaborative computing environment and the services they provide. The collaboration agents take part actively in the collaboration establishment process illustrated in Figure 2.1 by making context-aware decisions about the properties of offered services during service selection and population phases (if the services have so-called dynamic properties), and expressing commitments and willingness to collaborate during the negotiation phase.

Collaboration agents are in many respects similar to the kinds of agents found in multi-agent systems [227] and artificial intelligence research [218] where agent operation and properties are characterized by their autonomy, situatedness, and sociality [116, 278]. In the following, autonomy and sociality are taken as such as characterizing factors of collaboration agents; the situatedness however becomes a classifying factor of the agent activity aspect.

When considering the type of agent *activity*, collaboration agents systems can be classified either as *1)* passive, *2)* active or *3)* pro-active. Operation of passive agents depend on external stimuli, decision making procedures and processes. Passive agents serve only for providing uniform interaction interfaces and are valuable in so-called computer supported collaborative working (CSCW) applications [207] and traditional workflow management systems (see for example [50] for an overview of workflow management systems) for intervening and controlling operations between human resources, for example.

Active agents perform collaboration activities that have been prescribed by some means before the actual operation. An active agent thus operates and reacts in a strictly predetermined manner and does not possess the capabilities for autonomous decision making or negotiation, for example. Typical inter-enterprise integration approaches and early academic projects on that field, such as WISE, MASSYVE, or CrossFlow [146, 204, 94], can be considered as consisting of active agents representing the intra-enterprise business processes and providing uniform integration interfaces between such processes.

Pro-active collaboration agents align strongly with the concepts found from agent-based systems research [227, 116] and artificial intelligence (AI) [218]. The pro-active collaboration agents extend the active ones by introducing configurability, context awareness and self-reflection capabilities. *Configurability* refers to the capability to adapt the properties and behaviour of agents using "light" and "agile" methods. Such configurability can be attained using mechanisms such as runtime interpretation of a domain-specific (scripting) languages, declarative rule-based systems or dynamic aspect weaving (see for example [202, 37]).

Context awareness refers to the capability of an agent to make itself aware of the relevant properties of the surrounding environment affecting its operation. The term "context awareness" is preferred over the term situatedness (as used for example in [116, 278]); this is to underline the explicit and partially pre-determined nature of the contextual knowledge in distinction to the more open world assumption implied by the term "situatedness". To establish context awareness, some kind of sensors are needed for the perception of the environment. In the context of distributed systems, these sensors are usually implemented as message interceptors attached to the communication paths. Communication interceptors can then be used for example to monitor contextual information concerning security, timeliness, and conformance of collaboration agent interactions.

Self-reflection provides the facilities for an agent to adapt itself to the changes happening in its working environment. Self-reflection presupposes configurability and context awareness. While an agent can be considered configurable if it is specializable to different contexts using for example specialized administration tools, the self-reflection capability implies that the agent is capable of automatically regulating and adapting its behaviour on the basis of the properties of the surrounding environment. Using the terminology of computational reflection [159], a pro-active agent is causally connected with its operational environment.

When considering the model of *sociality* of collaboration agents, they can be identified either as co-operative or competitive. In co-operative agent systems [65] the agents work independently but in coordinated fashion to achieve the shared objective of the agent community. While co-operative agents are the most prevalent type of agents needed in collaborative computing environments, some level of competitiveness can be expected especially during the collaboration negotiation phase. There are three different kinds of negotiation strategies that have been researched in the context of multi-agent systems and are utilizable during collaboration establishment, namely bidding, auctioning and bargaining [238]. In bidding a client declares properties for the wanted service and then asks for bids from service providers. After receiving service offers (bids) from providers, client decides which service provider is chosen using heuristics like cost-benefit analysis [152] or multi attribute utility theory [17]. In auctioning a pre-fixed auctioning protocol is

followed by all the participants [238]. Bargaining is the most complex negotiation strategy which may involve multiple proposal-counterproposal–rounds before a mutual agreement or disagreement is established [238].

2.1.3 Means for achieving interoperation

Interoperability means the capability of computational entities to collaborate in such a fashion that eventually either their mutual goals become fulfilled or their co-operation is dissolved in a controllable and pre-determined manner in case of an error. Interoperability can be characterized as having technical, semantic and pragmatic aspects [141] and it can be achieved by using different approaches. These different aspects of interoperability and the kinds of approaches available for establishing interoperation are discussed in the following.

Technical interoperability means that the technological facilities underlying the collaborating parties are compatible such that for example communication paths can be established. When considering technical interoperability, that is the connectivity, communication and encoding related aspects, incompatibilities between languages, interfaces and operational environments can be solved quite efficiently. Methods and techniques like interface description languages [144, 177], adaptors [277, 210], wrappers [162], middleware [185, 239] and middleware bridges [76] have quite successfully been applied for tightly coupled enterprise integration purposes.

Semantic interoperability deals with the meaning of information and message exchange patterns. When considering information semantics, domain specific vocabularies and ontologies have been used for establishing semantic compatibility through homogenization. Standardized vocabularies, such as given by ebXML [133] or RosettaNet [211], and common taxonomies such as the North American Industry Classification System (NAICS) have been used for such purposes. More recently, ontologies based on logical frameworks have been utilized. In this context, an ontology is a considered as logical theory that defines a vocabulary and a logical language [158]. Vocabulary for an ontology defines a set of basic symbols which can be composed to more complex objects with the operators of the ontology language. An ontology language is usually defined by triples which indicate $(subject, predicate, object)$-type of relations between the basic terms. The most prominent ontology definition language currently is the Web Ontology Language (OWL) [259] that provides a markup-language with reasoning capabilities based on the logical framework of Description Logics [174].

Behavioural semantics considers the compatibility and substitutability of interaction patterns taking place between services. Behavioural compatibility means that the individual actions between a set of interaction patterns form dual pairs in a sense that each input activity of a certain type is provided with a corresponding output activity with a similar type and vice versa. Behavioural substitutability means a relationship between two behavioural patterns that respects the Liskov's substitution principle [151] which states that any relevant property that holds for the original behaviour must also hold for the substitute behaviour. The notions of behavioural compatibility and substitutability between business services can be addressed using formal methods such as pi-calculus [166] or Petri Nets [196] and validated using corresponding analysis methods, such as model checking or theorem proving [56]. Using such formal methods, the notions of similarity, correct termination, or deadlock freedom of service interactions can be analyzed and validated (semi-) automatically.

Finally, pragmatic interoperability is achieved if the intentions, business rules, and organizational policies of collaborating parties are compatible with each other. Pragmatic aspects of interoperation reflect the social circumstances behind the technical systems and must be considered explicitly in collaborative computing environments. Willingness of participants to collabo-

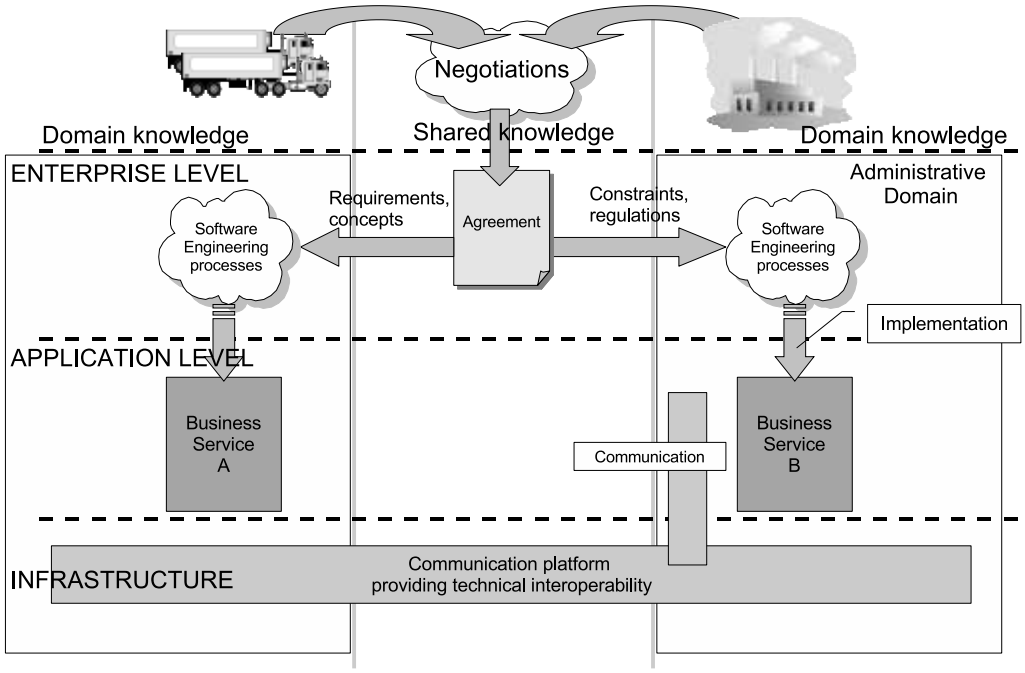

Figure 2.2: An illustration of the tightly coupled integration approach.

rate and trust between peers are few of the issues that must be decided upon before actual collaboration may take place. Negotiation of collaboration contracts, contract-based management of loosely-coupled collaborations and reputation based trust-establishment [213, 214] systems are few of the mechanisms that can be utilized for addressing the pragmatic interoperability aspects [215, 216, 165, 138].

The means for achieving interoperation can be broadly categorised to three approaches, namely integration, unification and federation. Integration aligns with the traditional model of software systems development where interoperability is ensured by pre-development and pre-operational agreements about the properties of collaboration components, and basically handcrafting the corresponding software artifacts to fulfill the prerequisites for interoperability. The principle of integration as means for interoperation is illustrated in Figure 2.2. First some kind of negotiations are held between the collaborating parties to come into conclusion about the requirements and properties of the collaboration. The requirements engineering process results in documentation providing the description for the collaboration properties. The resulting documentation can be represented using natural language or UML [182] diagrams, for example. This interoperability knowledge is then implicitly injected into the components of the computing system during the system design and development processes taking place in the individual administration domains. Explicit knowledge about the prerequisites for interoperability does not exists anymore after the software engineering processes at the application level. Tightly coupled ad-hoc solutions, software adaption and use of common computing frameworks (middleware) are typical integration approaches [216].

This model of application integration provides solutions for establishing technical interoperability. Technological heterogeneity can be dealt with, but typically there are very strict bindings between the collaborating business applications and underlying computation and communication platforms. Heterogeneity in higher levels of abstraction, such as when considering service behaviour or information representation, is not usually tolerated due to the rigid development pro-

cesses and non-existing interoperability knowledge. Integrated collaboration model does neither tolerate dynamism or autonomy of participants: as the information about interoperation prerequisites is hidden inside business application and infrastructure components, dynamic changes in the environment can not be coped with. While integration can be very effective means for achieving technical interoperation, it is not flexible enough to be used in modern collaborative computing environments and does not support semantic or pragmatic aspects of interoperability [216]. More over, rigid and tight bindings between the integrated components degrade their reuse in different contexts and hinder their evolution.

In unification a shared model describes the functionality and responsibilities of each collaboration participant and provides the knowledge needed for attaining interoperability. Interoperability knowledge is then used for generating the actual components of the collaborative system. As the components implemented by generative methods are based on the same platform independent model, interoperation between components generated by different vendors should be possible, given appropriate code generation tools. The unification approach is typically utilized in conjunction with a development framework following the principles of the Model-Driven Engineering paradigm [225] such as provided by OMG's Model-Driven Architecture initiative [84].

Unified collaboration model provides support for both technical and semantic interoperability. Heterogeneity of computation and communication platforms is also supported as the meta-models are platform independent. Unified collaboration based on shared models is however inflexible due to the fact that although the design entities in the models are reusable, the actual service components are typically specialised for the specific architecture and use-case described in the model. Model evolution is effectively supported but real dynamism, that is awareness and adaption to changes happening dynamically is not supported by the unified collaboration model, unless explicitly modeled in the corresponding meta-information elements [216]. As in the integration approach, interoperability knowledge is implicit at the application level.

Especially the pragmatic aspects of interoperation are not respected by approaches based on integrated or unified interoperation methods: since both integration and unification do not possess the information regarding interoperation between the components during their operation, parameters related to interoperation can not be changed or negotiated. In addition, components based on these approaches can not cope with dynamic aspects of business networks as all the properties regarding interoperation are predetermined during design and implementation of the corresponding components.

To establish support for pragmatic aspects of interoperation, the inter-dependencies between components have to be relaxed. In federated collaboration, illustrated in Figure 2.3, no shared model describing the operation of a collaboration is necessarily needed. Each participant may have their own models describing their business domain and services. To achieve interoperability, a shared and unified metamodel is exploited. The metamodel provides concepts and constraints to be used by the modelling languages and methodologies during enterprise modelling and provides the rules for deciding if models are interoperable. A metamodel provides thus a typing discipline for the corresponding models.

Federated collaboration provides support for dynamism and autonomy since the actual collaboration models are constructed dynamically by utilizing negotiation mechanisms, model verification and monitoring of collaboration behaviour with respect to the collaboration contract [142, 140]. Even model evolution is supported, as interoperability between business applications is achieved using the shared metamodel and corresponding interoperability validation facilities [216] thus removing the need for static collaboration models. In contrary to integration and unification, models exists as first-class and explicit knowledge elements during the operation of the collaboration.

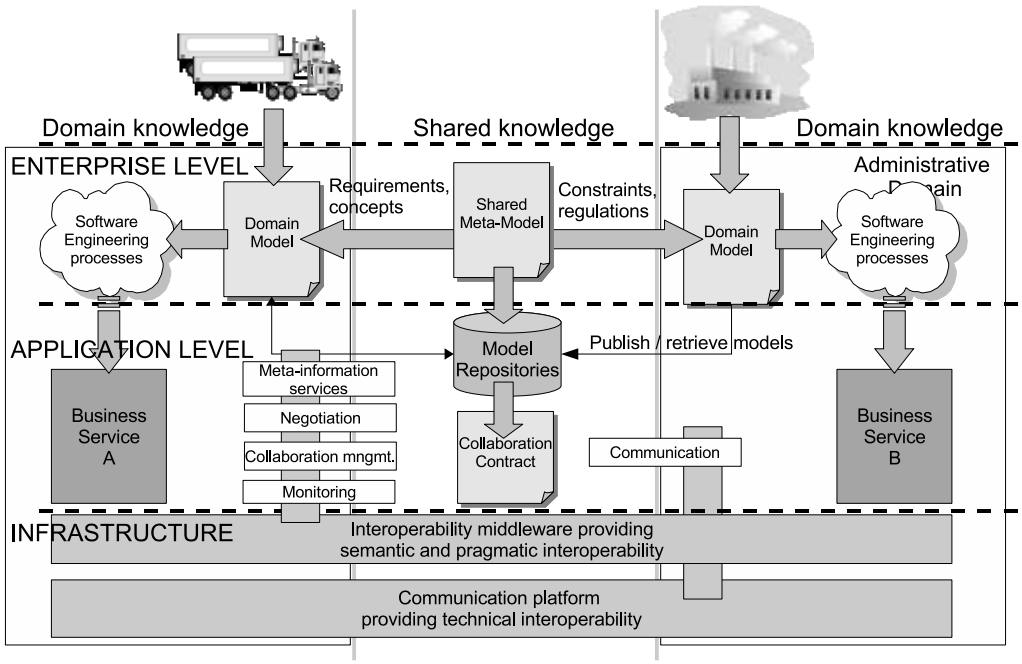

Figure 2.3: An illustration of the federated interoperability approach.

2.1.4 Comparison of collaborative computing environments

In the following we identify and compare different kinds of collaborative computing environments found currently both in industrial use and academic research. The exemplars chosen provide a cross-cutting view of the prevailing practices of collaborative computing and reveal the weak points to be addressed when striving for next generation collaborative computing environments. The properties of operational environments and collaboration agents identified in Section 2.1.2 and 2.1.1, and the means for achieving interoperability described in Section 2.1.3 provide the basis for the following discussion and comparison. In the following, we first introduce a selection of archetypical systems for collaborative computing and conclude with a comparison of their properties.

Computer-Supported Cooperative Work (CSCW) systems provide simple tools and services for arranging collaboration between people and organizations. CSCW systems were developed in the 1980s to address collaboration needs among product and organizational system developers, and researchers working in multi-disciplinary cooperative projects [96]. The CSCW systems concentrate on providing facilities for supporting human-centered group communication and co-operative work, typically in an environment with pre-determined working habits, processes and tools. CSCW environments are also known as “collaborative computing systems” or “collaborative software”, and the operational environment is commonly known as “group-ware”.

The operational environments for CSCW allow only restricted kind of platform heterogeneity by providing corresponding application implementations on top of platform independent run-times such as the Java Runtime Environment (JRE) [9]. Otherwise, service variability is not supported. Group-ware aimed for assisting joint authoring projects are based on tightly closed environments with static communities of collaboration participants [207]. Typically the collaboration agents used in CSCW environments are passive and co-operative and their autonomy is very restricted: only pre-defined activities provided for the users to express their willingness to collaborate (i.e.

accept or decline a collaboration activity) are allowed.

When considering integration of business functions and divisions within a single enterprise, two most prominent approaches provided are enterprise resource planning (ERP) and enterprise application integration (EAI) systems. Both kinds of platforms are directed towards large and medium size enterprises that typically consist of distinct sub-departments with heterogeneous and possibly autonomic administration domains. ERP systems focus on integrating the internal business functions, such as customer relationship management, order management and accounting, through an integrated enterprise wide application solution. In the ERP-approach companies must typically re-engineer their business processes to adopt the predefined ERP standard business processes [147]. An example of an enterprise resource planning software is the SAP ERP system provided by SAP AG [223].

Typical ERP systems closed environments based on tight integration. Usually an ERP system utilized in an enterprise is provided as a whole by a single system provider. The collaboration agents in ERP systems are passive and co-operative. Due to tight integration approach, autonomy of individual subsystems is either very limited or totally forbidden. Interoperability between subsystems and components is created at technical level using tightly coupled application integration.

EAI systems provide utilities and infrastructure services for integration of enterprise applications that enables information sharing and business processes [74]. EAI systems take a less pervasive approach on enterprise subsystem integration when compared to ERP by providing a set of generic middleware services for integration of enterprise legacy systems into unified, enterprise wide business systems. In addition for providing a communication and distribution middleware services, an EAI system includes typically a workflow management system (WfMS) for realizing and controlling business processes between the subsystems. A workflow is a collection of tasks to accomplish a business process [91]. A workflow management system is based on workflow specifications and their execution. A complete workflow management systems provides tools for process modelling and workflow specification, process re-engineering, and for automating and implementing the workflows [91]. When compared to an prefabricated applications of ERP systems, workflow management systems (WfMS) provide more flexibility and controllability over the enterprises' business processes. WfMSs are more suitable for implementing business processes involving humans and software systems in heterogeneous environments, whereas EPRs are suitable for transactional workflows in homogeneous environments [50]. Workflow management systems are used in service-oriented computing to implement business processes with orchestration and choreographies. In smaller enterprises, middleware platforms such as J2EE [239] or CORBA [178] are utilised as an inter-enterprise integration technology.

EAI are collaborative computing systems that are directed for environments where the components providing business functions are implemented using heterogeneous technology and may have substantial differences in their service interfaces. EAI systems do not however support service interface heterogeneity, but such heterogeneity is hidden behind wrappers and adapters. A typical environment targeted by an EAI system is closed with respect to the set of dependable services. The collaboration agents in EAI systems are passive and co-operative. The autonomy of the subsystem service providers (in enterprise subdivisions, for example) is slightly more supported when compared to ERP systems. Interoperability is still enabled with technology level integration solutions.

Inter-enterprise integration (I-EAI) refers to integration of information systems residing in distinct enterprises for enabling supply-chain management and sub-contracting relationships. In inter-enterprise integration properties such as technological heterogeneity, dynamism of the provided services and autonomic administration domains have to be addressed more carefully than in intra-enterprise integration. The key ingredients of inter-enterprise, or business-to-business (B2B)

integration are business vocabulary and process unification, and distributed workflow management. Technological and partly service interface heterogeneity can be addressed using Web Services [262] technology, for example. Standards such as ebXML [249, 133] and RosettaNet [60, 211] defining business related vocabulary and standardized business processes are used for achieving limited kind of semantic interoperability between enterprises. The emphasis of distributed workflow management is on the division of work to sub-workflows that can be engaged within the individual enterprises [163]. Coordination of the distributed workflow is provided by a centralized entity that is typically the owner of the corresponding supply-chain.

The collaborative activities are enabled by service-level agreements (SLA) [126]. However, these contracts and agreements are typically negotiated and agreed in person and left implicit at the level of collaboration agents. The enactment of services with respect to SLA:s must however be monitored using service level monitoring facilities. Another peculiarity rising in the inter-enterprise collaborations is the need for long-running transactions. Long-running transactions provide concurrency control and recovery mechanisms similar that developed in database management community for managing and ensuring consistency within the distributed business processes [79]. Such techniques as linear Sagas, flexible transactions and compensations that were developed to address the problems related to long-running transactions can be applied to workflow systems [4] used in I-EAI context.

Collaborative computing environments for inter-enterprise application integration have to endure implementation heterogeneity and variability of the service interfaces. The collaboration breeding environment is a closed one with the set of potential partners having pre-agreements before the actual collaboration operation. The collaboration agents in I-EAI context are passive and targeted only for providing uniform interfaces for collaboration management activities. Interoperability is achieved at the technical level using application integration, and partly and the semantic level using unification of business vocabulary and processes with standards such as ebXML [133] and RosettaNet [211].

Collaborative Networks (CN) [46] are loosely coupled collaborations between entities bound together by an explicit collaboration contract for achieving common or compatible goals. Collaborative networks Virtual enterprises, virtual organizations or virtual professional communities are examples of the different manifestations of collaborative networks that distinguish from each other on the kinds of goals and co-operation structures that are striven for. Virtual enterprises encountered in automotive industries take the form of supply chains with a dominant player, for example [240]. In other contexts, such as formal joint ventures [240], professional virtual communities, or collaborative engineering environments [164], the collaborations are formed in a more loosely coupled and democratic way to share risks, development costs or intellectual capital between the organizations, or to complement each others skills [240, 46]. In many projects establishing collaborative networks, like PRODNET [2], MASSYVE [204] and FETISH-ETF [45], the operational environment provides facilities for negotiating and modelling the collaboration processes, and for controlling the enactment of the processes.

Typical Collaborative Networks, such as Virtual Organizations (see [47]), can be characterized as heterogeneous collaboration environments comprising a semi-open breeding environment and facilities for dynamically establishing opportunistic virtual organizations from the set of dependable services. For guaranteeing semantic interoperability, a unifying shared model to which all collaborating participants have to adapt their local services is used. In addition, standardized business vocabularies and processes are typically used. The co-operation in Collaborative Networks is governed by explicit collaboration contracts and service-level agreements.

The technology of intelligent agents and multi-agent systems (MAS) have been used for establishing loosely coupled collaboration environments. For example, in the ADEPT (Advanced

Decision Environment for Process Tasks) project [118, 117] agents were used for negotiating on services providing means for business process management and electronic commerce. Such agent-based systems are especially useful in contexts involving an inherent distribution of information, autonomous actors, social interactions and pro-activeness, unpredictable processes and opportunistic behaviour [118].

An intelligent agent in this context is considered as an autonomous decision-making system, which senses and acts in some environment [276]. Agents make autonomous decisions about their future activities without user intervention, are able to react to changes happening in their operational environment and take initiative to change their environments, and have social abilities to interact with other agents [276]. A multi-agent system comprises a set of agents and an operational environment providing the necessary infrastructure services, such as communication, interaction management, and situation assessment [118], for the agents. The agents use negotiations for establishing the service-level agreements, SLA:s, between the service providers [117]. During these service negotiations semantic mappings and transformations might be needed to create a mutually comprehensible information sharing language [117].

Collaborative computing environments based on intelligent agents and multi-agent systems tolerate both service implementation and interface variability, due to negotiable interaction establishment. The corresponding breeding environment can be considered as open. Intelligent agents are pro-active and manifest both co-operative and competitive behaviour. Interoperability is established at technical and semantic levels. The existence of explicit negotiation mechanisms enable attaining a limited form of pragmatic interoperability: the values of pre-defined service-level agreement templates can typically be dynamically negotiated [117]. Unification is used as primary means for establishing interoperation. The corresponding abstract models typically prescribe the protocols to be used in agent negotiations, the architecture of the multi-agent system and the generic responsibilities of the (different kinds of) agents.

The collaborative computing environments discussed above are compared in Figure 2.4. The comparison is based on the properties of collaboration breeding environments, collaboration agents, interoperability approach, and the level of autonomy provided by the corresponding collaborative computing environment for service providers. Collaboration breeding environment properties were discussed in Section 2.1.1. Collaboration agents are compared with respect to their activity and sociality, as discussed in Section 2.1.2. The approach provided for enabling and managing interoperability is compared with respect to the level of interoperability provided (technical, semantic, pragmatic) and the means for establishing interoperability (integration, unification, federation). These aspects of interoperability were discussed in Section 2.1.3. Existence of electronic contracting mechanisms and explicitness of collaboration contracts are also taken as comparison criteria.

Finally, as a summarizing quality of the collaboration environments the autonomy provided by a collaborative computing environment is characterized by a set of numbers between zero (0) and five (5). The numbering characterizes the different aspects of freedom or abilities for increasing the autonomy of services and their providers. The numbering is cumulative in a sense that autonomy with respect to *a)* service implementation is addressed using number 0-1, *b)* service design with numbers 2-3, and *c)* service involvement with numbers 4-5. Autonomy ability at level 0 means that services have to be implemented using a homogeneous technological framework; ability at level 1 means that simple technological heterogeneity is tolerated with help of wrappers or adapters, for example. Autonomy levels 2 and 3 imply service design autonomy. At level 2 interface variability can be supported by application of dynamic binding mechanisms and semantic matching, for example. Autonomy level 3 corresponds to the ability of encapsulating services dynamically with non-functional aspects and collaboration specific configurations, for example.

			CSCW	ERP	EAI	I-EAI	CN	MAS	*FedSC*
Collaboration Breeding Environment		Closed	X	X	X	X			
		Semi-Open					X		
		Open						X	*X*
Collaboration Agents	**Activity**	Passive	X	X	X	X			
		Active					X		
		Pro-active					?	X	*X*
	Sociality	Co-operative	X	X	X	X	X	X	*X*
		Competitive					?	X	*X*
Interoper-ability	**Interop. Level**	Technical	X	X	X	X	X	X	*X*
		Semantic				X	X	X	*X*
		Pragmatic						?	*X*
	Interop. Means	Integration	X	X	X	X	X	X	*X*
		Unification				?	X	X	*X*
		Federation							*X*
Autonomy		(0-5)*	0	0	1	1,2	1,2,4	1,2,4	*1-5*

** 0 / 1 = homogeneous / heterogeneous service implementations, 2 = interface variability, 3 = late encapsulation of properties (configurability), 4 = negotiable contracts, 5 = negotiable contracts and breach management*

Figure 2.4: Comparing collaborative computing environments.

Autonomy at levels 4 and 5 is provided by abilities enabling autonomic decisions about service involvement during collaboration enactment. At autonomy level 4 participants are provided with negotiation mechanisms during collaboration establishment processes. At autonomy level 5 the negotiation mechanisms are in place and in addition the operational environment provides facilities for managing contract breaches during the operation of collaborations. That is, at autonomy level 5 a collaboration participant may decline the use of its services even if previously negotiated collaboration contracts exist; at level 5 this is not however considered as a unrecoverable error, but the situation is managed in an interoperable manner using the infrastructure services provided by the operational environment.

Finally, there is a column marked in Figure 2.4 with *FedSC* (for "Federated Service Communities") characterizing the vision we are striving for. The next generation of collaborative computing environments should be targeted for open service markets, thus only an open breeding environment will suffice. The collaboration agents should be pro-active when considering "routine" decisions during collaboration enactment. Both co-operative and competitive characteristics are needed from the collaboration agents taking part in next generation electronic business ecosystems. Interoperability mechanisms must increasingly address also the pragmatic aspects of interoperation and for fulfilling this purpose, a federated interoperability approach has to be supported. Support for electronic contracting and breach management activities are required from the corresponding operational environment. When considering allowing as much autonomy for service providers as possible, mechanisms for enabling late encapsulation of service properties and dynamic binding are required from the software engineering processes and operational environments. In the following Sections of this Thesis we will concentrate on the formalization of such federated service

communities.

2.2 Necessities for loosely coupled collaborations

The level of autonomy required by next generation collaborative computing environments can only be achieved if the information systems, services and collaboration agents participating in collaboration enactment are loosely coupled. A loosely coupled system enables its individual components to operate independently and to possess autonomy over local decision making, provides facilities for expressing and utilizing the capability of interworking (e.g. languages for service description, and service discovery facilities), and delivers infrastructure services for managing the mutual inter-dependencies when establishing collaborations (e.g. negotiated service-level agreements and collaboration contracts, and dynamic service binding and configuration mechanisms).

Loose coupling is not something that can be defined explicitly but is implied by utilisation of techniques that are known to lower the degree of dependency between software artifacts. In the context of expert database systems, loose coupling is achieved if the individual subsystems (AI and DBMS) can operate autonomically and communicate through well-defined interfaces [127], for example. In service-oriented B2B systems the use of document based communication caters for loosely coupled communication relationships between services [163]. In the context of inter-organizational workflows, loosely coupled processes *"work independently, but have to synchronize at certain points to ensure the correct execution of overall business process"* [256].

In collaborative computing environments appropriate methods and mechanisms must be provided for establishing bindings between service interfaces, and resolving properties of interactions and mutual behaviour in a loosely coupled manner. In addition, loose coupling necessitates contract-based government of the established collaborations. The collaboration contract negotiated between the participants prescribes the obligations for participants, activities required for attaining collaboration goals and the qualities of the mutual interactions. In the following, we discuss these necessities for loosely coupled collaborative computing. The elements for enabling loosely coupled collaborations are described in Section 2.2.1 while the contract-based approach for managing loosely coupled collaborations is briefly discussed in Section 2.2.2.

2.2.1 Elements for loose coupling

Loose coupling is facilitated with use of methods and mechanisms that explicates the nature of inter-dependencies and relationships between the collaborating entities, and more over let entities postpone the decisions about the collaboration properties as late as possible. The methods for attaining loose coupling include for example design ideologies that stress low coupling and high cohesion of individual services, well-defined service interfaces, and separation of concerns. Dynamic invocation mechanisms, name based resolution of service locations, runtime binding of service interfaces, and dynamic configuration of service properties and behaviour are some examples of the mechanisms feasible for establish loose coupling. In the following, we concentrate on such mechanisms, and leave the methodological and design perspectives aside.

Loose coupling in collaborative computing environments requires openness of *1)* service interfaces and behaviour, *2)* interactions and communication, and *3)* mutual behaviour and business processes. By openness we mean that the involved artifacts (service descriptions, communication channels etc.) are well-defined, have clear criteria for determining the consistency of their usage in a collaboration context, and are dynamically configurable and extendable with collaboration specific features.

Well-definedness of models describing collaboration artifacts means that their interpretation does not need to rely on human intuition but is "machine understandable". In the case of service interfaces their structural and behavioural properties must be defined using a rigorous typing discipline that provides means for checking service substitutability [151] or behavioural subtyping, for example. Standardized vocabulary such as ebXML [133] or Rosettanet [211] and generic ontology description languages [259, 261, 260] can be utilized for giving unambiguous semantics for the domain concepts used for describing information contents of communication. Formal semantics for interactions, mutual behaviour and processes can be given using finite state machines [21], Petri nets [97], or process algebras [220], for example.

Formal criteria for the consistency of artifacts involved in a collaboration must be provided to enable dynamic binding and encapsulation of services. The consistency criteria must set rules at least for behavioural interoperability between services, conformance of business processes with respect to behaviour manifested by the individual services, and feasibility of feature combinations attached to interaction relationships. Behavioural interoperability or compatibility has been researched for example in [49] in the context of pi-calculus [166]. Business process conformance criteria of various forms have been recently studied for example in [13, 54] and [149]. The various behavioural consistency rules emerging in a collaborative computing environment need to be formalized using applicable methods and included as an integral part of the conceptualizations, models and metamodels.

The capability to dynamically encapsulate service interactions with non-functional requirements such as *"security"* or *"non-repudiation"* pre-requires mechanisms to inject the corresponding properties into the models and technological artifacts representing the interactions. In an engineering framework that is based on model-driven engineering [225] discipline, such features are expressed using various *aspect languages*. Each aspect language can be considered as a language designed for expressing a demarcated feature of a system, such as communication security or timeliness of service interactions. However, in such a setting involving simultaneous use of several aspect languages the identification of *feature interactions* [99] and resolution of conflicts emerging from such interactions becomes problematic. The ordering between different aspects and their other mutual dependencies can not be addressed in an open, loosely coupled collaboration framework during the design of service interactions or individual aspects. Thus the feature interactions must be identified and the application possibilities of different features must be constrained correspondingly. A lot of work has been conducted on feature interactions especially in the domain of telecommunications systems []. In [66] the authors present a language-independent technique for detecting semantic conflicts between features attached in same artifact (or join-point in this case). In [221] an expert system for identifying and managing different feature interaction types (assistance, choice, conflict, dependency and mutex) in an collaboration environment based on a component middleware platform. The expert system in [221] captures domain specific concern interaction knowledge and can be used for reasoning about interactions in component-based systems.

Dynamic configurability of collaboration artifacts can be distinguished to three mechanisms addressing the levels of service interfaces, service interactions and business processes. At the service interface level late binding can be used to establish interoperation between services during collaboration enactment. Late encapsulation of service interactions with non-functional properties provides for dynamic configurability at the interaction level. Finally, negotiation of the properties addressing mutual behaviour and processes taken between collaboration participants is used for further increasing loose coupling of collaborations. The mechanisms of late binding and encapsulation are discussed below, while negotiations mechanisms are discussed in Section 2.2.2.

Late binding, also known as dynamic or runtime binding, of service interfaces is one of the

principal tenets of service-oriented computing. In a collaboration involving late binding between a service customer and a suitable service provider a *binding process* is initiated for deciding the properties and capabilities, especially the identity and location of service endpoints, of forthcoming service interactions. Essentially, the binding process establishes a contractual context (a binding) between service interfaces to enable service interactions [113]. The binding process may comprise refinements of communication channel models and negotiations about the properties of the communication channels. Communication channel properties such as support for different distribution transparencies [113] must be decided during the binding process. After a successful binding process, the participants share a common view about the properties of the corresponding service interactions and are provided with the platform specific models needed to configure their communication infrastructure correspondingly. Pre-defined communication channel templates prescribing the best practices of the domain and pre-validated communication channel property combinations can be utilized when establishing the bindings.

In late encapsulation the communication channel models provided by a binding process are further refined with non-functional properties. Late encapsulation involves selection and negotiation about the features of service interactions, such as security, privacy or non-repudiation, for example. In a collaboration environment that is based on the model-driven engineering [225] approach, aspect models and weaving models are utilized for expressing the non-functional aspects and their effects on the communication channels, correspondingly. After a successful negotiation process, the participants are provided with a shared service-level agreement, or SLA (see for example [157, 232, 233]), expressing the mutual commitments and expectations about the non-functional properties of service interactions. In addition, the late encapsulation process provides the participants with communication channel models refined with information included in the aspect models defining the semantics of the corresponding non-functional aspects. Horizontal model transformations are used for describing the effects of aspect models to the communication channel models residing the same level of abstraction. Such an approach has been proposed for example in [271, 131] and can be regarded as providing "translational semantics" of non-functional aspects in a model-driven engineering framework.

2.2.2 Contract-based collaboration

In a loosely coupled collaborative computing setting, interoperation has to be based on contractual relationships between participants to enable late decisions about collaboration properties. Before a collaboration can be initiated, each participant of a community has to express their willingness to collaborate, and make a commitment to take the actions requested and agreed upon during the negotiations. A contract describing the properties of collaboration and commitments for partners is provided as a result of a successful negotiation. Based on these expressions on commitments, the participants may govern their own and other participants collaborative activities. A collaborative computing environment must provide means for electronic contracting, including mechanisms for contract establishment, contract enforcement and supporting trustful decision making [167].

The mechanisms for contract establishment include repositories for storing contract templates and contracts, mechanisms for contract negotiation and validity checking [167], for example. Contract templates provide standard contract forms to facilitate the drafting of collaboration contracts. Repositories storing contracts are needed for keeping signed copies of contracts as evidence for possible dispute settlement processes.

Negotiation is *"a process by which a group of (collaboration) agents communicate with each other and try to come to a mutually acceptable agreement on some matter"* [153]. Negotiation provides a mechanism for expressing the autonomic intentions of partners during collaboration

establishment. Negotiations are used primarily to come into conclusion about shared properties among the partners through a process of gradual refinement of the collaboration properties. However, negotiation is also a mechanism to introduce autonomic decision procedures to collaboration enactment. When provided with functionality enabling negotiations between collaboration agents, an enterprise can keep its decision procedures and preferences private during collaboration establishment, since a negotiation about joining the collaboration leaves an option to decline the invitation to join the community. Collaboration contract comprises declarations about the commitments formed between the partners, service-level agreements describing more technical properties of interactions, and policies and protocols specifying the agreed models for collaboration dissolution and contract breach handling.

Contract enforcement comprises infrastructure services that provide means for monitoring the compliance between service interactions and a collaboration contract, contract enforcement when a collaboration partner deviates from the behaviour prescribed by the contract, and dissolution of collaborations. The essentials for enabling contract enforcement and dispute settlement in electronic collaborations are typically borrowed from conventional (non-electronic) environments: compensations, insurances, fines and trusted third parties acting as notaries are used for giving the motivation for the partners to follow the collaboration contract.

A partner may either involuntarily (for example due to *force majeure* situation) or by purpose (due to changed priorities, business rules or policies) deviate from the contract. Once such a deviation is noticed by the contract monitoring infrastructure, the contract enforcement mechanisms come into operation. Contract notification, mediation and arbitration mechanism are examples of typical contract enforcement facilities [167] used for informing the parties about (possible) contract deviations, and settlement and resolution of actual contract breaches.

2.3 Engineering frameworks for collaborative computing

Service-oriented computing (SOC) [191, 231] paradigm and model-driven engineering (MDE) [225] discipline provide two essential frameworks for engineering collaborative computing systems. Service-oriented computing provides an ideological framework which emphasizes the concept of a service as the primitive computational entity in collaborative computing environments. Model-driven engineering enables efficient design and production of services and service compositions by considering models, modelling languages and model transformations as the essential primitives for software engineering. Both SOC and MDE emphasize the role of well-defined descriptions and models for facilitating distributed computing and software development. The essentials of these frameworks are discussed in the following.

2.3.1 Service-oriented computing paradigm

Service-oriented computing (SOC) has been entitled as the new paradigm for designing and implementing complex distributed systems [191, 231]. Service-oriented computing is based on the notion of services which are independently developed, autonomous software components with well-defined platform independent interfaces. Another important point of the paradigm is the inherent composability of services: new composite services with added value can be designed and implemented using the already available services. Service-oriented computing is a description intensive paradigm: each service must have an explicit description of its capabilities. Service-orientation manifests document-oriented computing semantics meaning that the purpose of computation is defined in conjunction by the visible document exchange patterns (protocols, processes) and the meaning of the individual documents.

Currently the most renown technological platform and standardization framework for establishing service-oriented computing is based on the Web Services architecture [262]. A web service is a software component which is identified by an URI [23] and provides its functionality using the standard Internet protocols, such as SOAP over HTTP. Web service interfaces and bindings are described using XML, typically with WSDL [55, 53]. Web services standards address such areas of distributed computing as communication and messaging [265], interface descriptions [55], and business process descriptions [242, 125]. The Web Services standardization is not the first service-oriented computing initiative: standardization on reference model for open distributed processing [113] and CORBA [178] have provided facilities for service-oriented computing since the 90's. However, the advent of Web Services technology has considerably increased the use of service-oriented computing techniques and technologies due to its open and low-cost (e.g. use of TCP/IP and HTTP) approach for distributed computing.

The primary goals of service-oriented computing address problems related both to software engineering and infrastructure support for loosely coupled collaborative computing. The SOC-paradigm essentially provides a framework for distributed software components and the basis for "open service markets". Attaining these goals requires re-usability, flexibility and composability from the service artifacts used during design and runtime, as well as infrastructure services for realizing an ecosystem for interoperable service delivery. Service-oriented computing provides for loosely coupled collaborative computing, as it emphasizes the use of self-descriptive, independent and composable software entities and loosely coupled collaboration enactment based on contractual relationships.

The main concepts of service-oriented computing are those of *1)* services, *2)* service declarations, *3)* service collaborations and *4)* service-oriented architectures. A service is a self-descriptive, independent software component with a well-defined interface [231]. Self-descriptiveness in this context means that all relevant information concerning the use and behaviour of a service is available explicitly for the users of the service. Services are independent software components in a sense that their functionality does not depend on any other services and especially they do not have any implicit dependencies on their operational environment. If a service needs other services to implement its functionality and this information is relevant to know, these dependencies must be given in the service declarations.

In the context of SOC two kinds of service declarations can be distinguished: service definitions and service descriptions. Service definitions are formal specifications of service capabilities and their primary purpose is to introduce means for attaining service interoperability and to categorize available services. Service definitions are models that characterize the essential properties and functionality of conceptual services from service user perspective. Service definitions are designed based on the requirements identified using domain analysis methods such as use case analysis, customer feedback or interviews. The service definitions must be formal enough to preempt unambiguous interpretation about the purpose, applicability and behaviour of the service.

Service definitions must not constraint the choice of technologies that can be used for implementing the corresponding kind of services. That is, they must be described in a technology independent manner. Decisions about the use of specific kind of communication technology or even paradigm (such as RPC [6] or publish-subscribe [14]) needed to realize the service must be left open to promote reuse of the conceptual service and effort put into its analysis and design. The further these decisions can be put back in the service provisioning process, the more loosely coupled interaction relationship can be achieved. In the best case scenario, these decisions are made during a dynamic service binding process.

Service descriptions are more technology specific declarations advertising properties of service instances and they are used for establishing communication paths between the corresponding

service endpoints. Service description takes place when a service provider has decided to publish some of its services. In the description phase the corresponding service definition provides a template that is refined with technical details, provider specific properties and service location information. More over, the service definition constitutes a conformance criterion that must be met by the service implementation and the service description used for advertising the service. Service descriptions provide also binding information needed to establish a connection with the service implementation. The binding information typically prescribes such properties as communication protocols and serialization mechanisms to be used, addressing information, and non-functional requirements (related to trust, security or privacy, for example) over the communication channels.

The explicit distinction between platform independent service definitions and platform specific service descriptions enables efficient separation of concerns with respect to the conceptual services and their technical incarnations. This separation is to some extent accomplished by any service description language that distinguishes the actual service interfaces from the technical details. For example in WSDL [55] technical bindings are separated as extension elements of the interface descriptions. However, the service definition concepts used in Web Services framework, such as "abstract service" [55] or "abstract process" [242] are quite artificial since they are not first-class concepts of the corresponding description framework but merely syntactic constructs and constraints with no well-defined semantic difference.

In the context of service-oriented computing two different kinds of service collaborations can be identified: service orchestrations and service choreographies. An *orchestration* is a kind of service collaboration whose intention is to provide an added value composite service to be used in a specific collaboration context. An orchestration describes a locally executable business process which coordinates a set of services in such a way that the resulting composite service can fulfill the requirements of one or more choreography roles. Service orchestration uses coordination and centralized process enactment to glue together a group of services in a such way that the resulting composite service can be delivered to clientele. Service orchestrations, and basic services as well, are contextually open software artifacts: they are not functional entities by themselves but provide functionality to be used in some context. In the lines of [63], service orchestration can be thought as "service reuse in the small" since service orchestration provides a functional module (the composite service) to be used in a larger collaboration context. Service orchestration is supported in the Web Services framework by the WS-BPEL standard [242] that specifies an XML-based description language for centralized business processes. The orchestration is described as a process consisting of primitives for communication, data manipulation and controlling the flow of operation.

A *choreography* defines a distributed business process that coordinates a collaboration between a set of distributed services. A choreography is declared by a set of choreography roles and the mutual activities taking place between the roles. A choreography describes a coordinated collaboration between a set of geographically distributed services put together to achieve some common goal among the partners delivering the services. Reflecting upon the seminal work of DeRemer and Kron [63], choreographies provide "service reuse in the large", since choreographies structure together collections of modules to form a functional system. In this case the "modules" are services and the structuring is accomplished by defining interconnections and coordination relationships between the services. Service choreographies are addressed in the Web Services framework by the WS-CDL (Web Services Choreography Description Language) which is an XML-based language for describing the global, externally observable behaviour among participants of service collaborations [125].

A service-oriented architecture provides the technological support for service-oriented computing. Service-oriented architectures have been around at least since the ODP-standardisation [111]

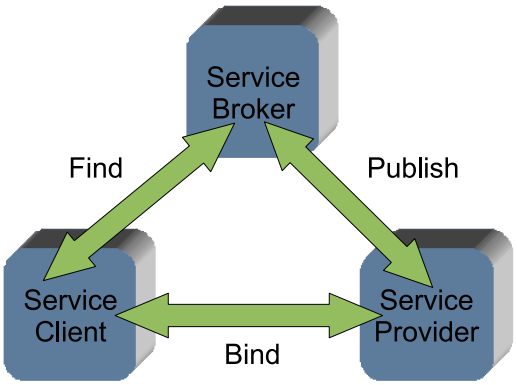

Figure 2.5: The basic principle of a service-oriented architecture.

and related implementations such as the trading services for DCE [18] and CORBA [176] platforms. The service trading mechanism is an implementation of the broker pattern variant [43] that is commonly utilised to decouple component interfaces in distributed systems. It has been used for example in the ODP standardisation [113], the related OMG CORBA middleware platform [178], and most notably is presupposed by the service-oriented computing paradigm [191, 231] and is included in the Web Services architecture [262] in form of the UDDI registries [250].

A service-oriented architecture utilizes service trading infrastructure for achieving loosely coupled service interactions between service clients and providers. The basic elements of SOA are illustrated in Figure 2.5. In a service-oriented architecture a *service broker* manages a repository containing descriptions of the service available for use. The service descriptions are published by the *service providers* which maintain the corresponding service implementations. When a *service client* wants to use a specific service, it queries the service broker about available services matching client's criteria. When such a service is found, the service broker mediates the service details given by the service provider to the client. Based on these details, which especially contain the identity and location information for the corresponding service, the service client and service provider may establish a service binding. The service binding enables further service interactions between the client and the service provider.

The service trading infrastructure provides only the basis for realizing a service-oriented architecture. However, a feasible SOA solution includes in addition to the service broker mechanism also facilities for sharing vocabularies, document formats, business process descriptions and so on. In addition, a service-oriented architecture should provide infrastructure services that are semantically richer and more meaningful to the end-users, such as functionality for business level management of services. Such an extended SOA has been discussed for example in [191, 192] and is illustrated in Figure 2.6. At the bottom level, a basic service-oriented architecture with service trading infrastructure is used for providing functionality for service publication, discovery, selection and binding. The upper levels use this basic SOA for establishing support for service composition and service management. At the highest level of the extended SOA framework, market makers manage services using terminology and tools that align more closely with the business level concepts.

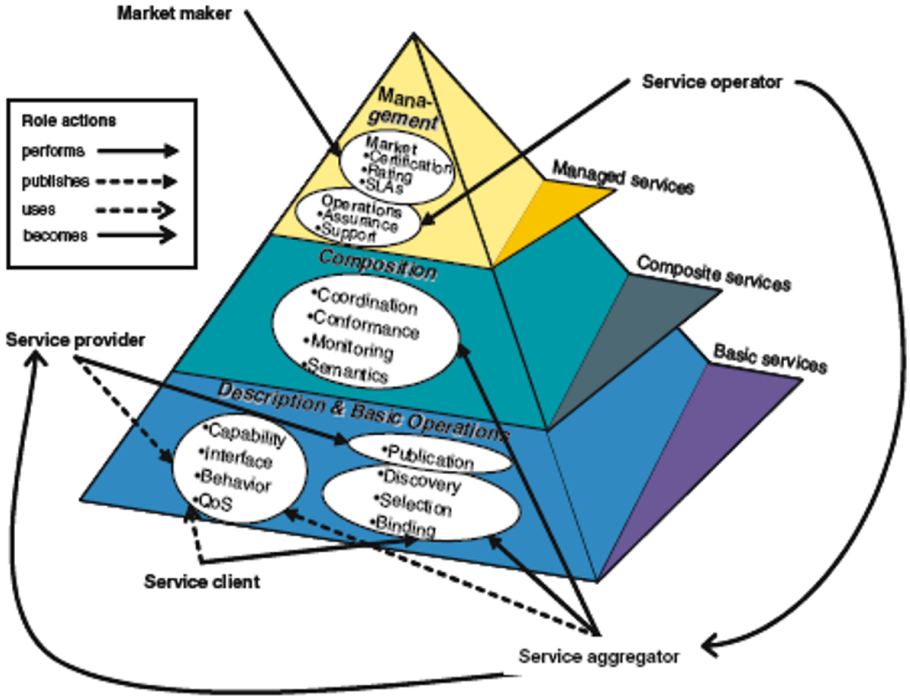

Figure 2.6: An extended SOA framework [192].

2.3.2 Model-driven engineering

Model-driven engineering (MDE) is a software engineering discipline which considers models as first-class entities and primary objects of engineering activities. The main challenges that MDE approach tries to respond to are induced by the complexity and evolution of computing platforms as well as complexity of system integration and configuration [225]. Domain specific languages, model transformations, and code generation are utilised to bridge the gap between problem domains and technology, and to produce software artifacts and systems that are "correct-by-construction" [225].

Model-driven engineering is founded on three principles, namely direct representability, automation, and open standardisation [85]. Direct representability means that instead of modeling the systems using concepts and vocabulary of the technological domain, modeling should be done using the concepts of the corresponding problem domain. Domain specific modeling languages are needed for this purpose. The principle of direct representability is emphasized also in the context of service-oriented computing: service descriptions should be representations that reflect the business models and values, processes, and use-cases occurring in the corresponding domain of communal activity. Automation is considered in model-driven engineering as the primary means of refining models and generating implementation artifacts. Explicit models and automated transformations between different abstraction levels of models reduce the risk of human interpretation errors. To establish automation, both domain concepts and implementation technology need to be modelled. Open standards and standardised computing platforms should be used as the guiding frameworks and deployment targets of model-driven engineering process. This provides technical interoperability between systems and ensures persistence of the modeling efforts. In the context of service-oriented computing, the standardisation and technology provided by Web Services ini-

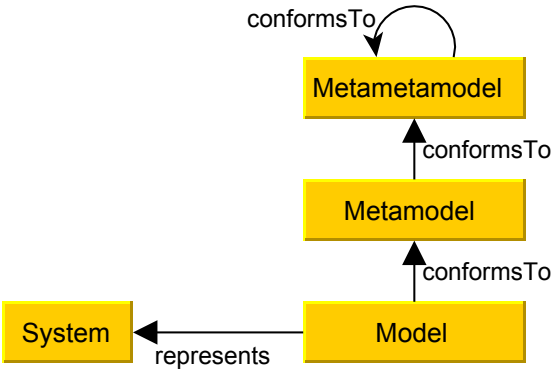

Figure 2.7: Basic modelling relationships in model-driven engineering.

tiatives [262] can be currently considered as the most feasible target framework.

A *model* is an abstraction of a real or language based system allowing predictions or inferences to be made [134]. A model is used for giving a suitable representation of reality for a given purpose [26] by the process of abstraction and projection. As such, it is not purpose or even possible to grasp all the aspects of reality into a model. At large, a system to be developed within an MDE process is not described by a single model. Instead, several models describing different viewpoints and abstraction levels of the system are used in conjunction. The OMG's MDA (Model-Driven Architecture) framework [84] advocates the use of three different types of models residing at different levels of abstraction, for example. So-called Computation Independent Models (CIM) are the most abstract kinds of models that represent business functionality, Platform Independent Models (PIM) describe the computational logic of systems, and finally Platform Specific Models (PSM) describe how the computational logic is implemented using some specific technological platform, such as Web Services [262]. In addition, Platform Description Models (PDM) are possibly used to separate and make explicit the nature of artifacts existing a technological platform. The platform description models can describe the functional as well-as non-functional properties of technological artifacts.

Model-driven engineering disciplines utilize hierarchies of modelling levels based on linguistic abstraction layers. Typically a three-level hierarchy is used consisting of *(terminal-) models*, *metamodels* and *metametamodels* [84, 30]. A model is a *representation* of the system under study. Each model *conforms* to a metamodel which provides an abstract syntax and typing rules for describing models. A model M conforms to a metamodel MM if and only if each model element has its metaelement defined in the metamodel [30]. Finally, a metametamodel provides a language for defining new metamodels. A metametamodel is a model that conforms to itself. The different model kinds found in model-driven engineering frameworks and their relationships with the system under study and other models are illustrated in Figure 2.7.

Models are applied in the context of model-driven engineering using three approaches: forward engineering, backward engineering, and "models at runtime" [27]. In forward engineering models are created before the implementation of the system. The models represent and characterize the properties of the system, and are typically then used for generating software artifacts needed for implementing the system. In the backward engineering approach a system exists before models. Models created from the system are used in this approach to better understand its properties. Models are then used for visualization, analysis or re-engineering purposes, for example. In the ap-

proach termed "models at runtime" [27] the models and the system coexist simultaneously during the operation of the system. Reflection mechanisms are used for animating this co-existence and inter-relationships between the system and the corresponding models. This approach is the most interesting and advanced one, and is essential for enabling loosely coupled and service-oriented collaborative computing environments.

Besides models, MDE emphasizes the use of *model transformations* for facilitating model-based software development. Model transformations are used for various purposes, such as model synchronization, consistency validation, model refinement, or code generation. In an MDE-based framework software artifacts such as components, objects or procedures are largely produced using model transformations and code generation; manual coding is used only to implement the necessary business logic fragments inside the generated code.

Model transformations can be roughly classified to three different categories: *1)* model-to-model transformations, *2)* model-to-text transformations, and *3)* model weaving. In a model-to-model (M2M) transformation a (set of) source model(s) and a transformation is used to generate a target model. In model-to-text transformation, the target of the transformation is an artifact in some technological space. Model-to-text transformations can be used for generating Java code, WSDL descriptions and deployment descriptors for enterprise application components, for example. In model weaving two or more models related by the rules given in a weaving model or specification which establishes typed links between the model elements [64]. In addition, model transformation approaches can be characterized by their capabilities and properties using the classification model given in [58], for example. Important criteria when considering the applicability of a model transformation environment to task at hand include the directionality of transformations, tracing of transformations, and transformation rule properties, among others [58].

Separation of concerns and loose coupling at level of modelling artifacts can be supported in a MDE framework using so-called *aspect oriented modelling* [93, 82, 229] approach. Aspect-oriented modelling establishes horizontal separation of concerns by encapsulating cross-cutting concerns of a system are into separate aspect models. Aspect models are then weaved [64] with primary design models describing the core functionality of the system to form complete descriptions of the system under study. An aspect-oriented modelling framework comprises four essential artifacts [229]:

1) A *primary model* describing the core functionality of the system;

2) A set of *generic aspect models*, where each model is a generic description of a cross-cutting feature;

3) A set of bindings that determine where in the primary model the aspect models are to be composed; and

4) A set of composition directives that influence how aspect models are composed with the primary model.

Aspect models are typically represented as pattern templates that describe common structural and behavioral characteristics of the system models the corresponding aspect is applicable to. Such pattern templates can be described using UML-based pattern language, the Role-Based Metamodeling Language (RBML) [82], for example. In a RBML-based model each role prescribes the properties that a model element must have if its is to be part of the corresponding pattern [83]. Roles can characterise the properties of both actual model elements (i.e. classes) and their inter-relationships (i.e. associations). In an aspect-oriented modelling framework based on the RBML-approach composition of an aspect model with a primary model is manifested by an

instantiation of the pattern by binding template parameters to system specific values existing in the primary model [82].

Chapter 3

Enabling inter-enterprise collaborative computing

Globalization of business and maturation of information technology has changed the way successful business is made. To maintain competitiveness, enterprises are focusing on their core competencies and outsourcing their supporting functions. As a result, an increasing amount of enterprises' income is gained inside varying kinds of business networks and as participants in inter-enterprise value and supply chains. In addition, new business models emphasizing the interworking capabilities of enterprises, such as virtual [175] and extended enterprises [274], have emerged.

Previous approaches on enterprise integration have focused on connecting enterprise information systems at technological level. Technologies that have been used for integration of enterprises' internal systems (Enterprise Application Integration, EAI [147]) or inter-enterprise application integration (Business-to-business integration, B2Bi [150]) do not however provide the necessary facilities or flexibility for operating in modern networked business environments. For this reason, the current trend is towards establishing loosely coupled collaborations between the autonomous enterprise information systems.

This chapter introduces concepts for inter-enterprise collaborative computing and addresses the challenges and requirements of modern networked business. Federated service community is a specific form of inter-enterprise collaborative computing based on service-oriented computing paradigm and emphasizes especially the autonomy of participants and loose coupling between business services. Attaining such a collaborative computing approach requires conceptual development for managing the complexity of collaboration establishment. In addition, the federated service communities approach has to be equipped with applicable development processes and collaboration environment infrastructure. These essential elements of federated service communities are discussed in the following sections. First, the essential concepts of collaborative inter-enterprise environments are defined in Section 3.1. Section 3.2 identifies issues related with interoperability management in federated service communities. In Section 3.3 a preliminary discussion of a service-oriented software engineering (SOSE) framework is given. The SOSE framework provides the means for delivering new business services and service-based collaborations using loosely coupled software-engineering processes. Finally, Section 3.4 introduces the facilities, that is collaboration environment infrastructure services and SOSE tools, required by the framework laid by the preceding discussions.

3.1 Conceptualising inter-enterprise environments

The concepts to be used for describing enterprise computing must reckon with the inherent properties of networked electronic business, such as differences in business strategies, operation models or legacy systems and especially, autonomy. A loosely coupled model for inter-enterprise collaboration can be established by relying on explicit models describing the properties of business services and networks, service trading facilities that uphold interoperability, dynamically negotiable collaboration parameters and contract-based governance of the resulting business network [140, 165]. This kind of a collaboration framework is called a *federated service community*. Relying on the aforementioned mechanisms and service-oriented business networks, federated service communities tolerate the autonomy and dynamism inherent in inter-enterprise computing as well as mask heterogeneity of systems and processes.

This Section introduces the concepts of federated service communities in a bottom-up manner. First, the elementary concepts needed for characterizing inter-enterprise collaborations are introduced. Section 3.1.1 introduces the notion of business services and related concepts. Section 3.1.2 discusses the different forms of behaviour encountered in business networking environments, namely business protocols and processes, and choreographies. Business networks are then characterized in Section 3.1.3.

3.1.1 Business services

A business service represents business functionality provided by an enterprise to its clientele and partners. A business service is an abstraction that is provided by an enterprise with conjoinment of a *computational service* (or business application) providing the core business logic, and a *network management agent* providing a business context aware representative for the business service. Functionality of a business service is controlled and affected by the enterprise's own business rules, organizational policies, and contracts regulating its operation and future commitments. The concept of a business service and its realizing components are illustrated in Figure 3.1 and discussed in the following.

Computational service implements the business logic by utilising the intra-enterprise infrastructure services and provides a service-oriented interface for accessing the corresponding business functionality. Computational service is implemented using local technology and over a distributed object computing middleware, such as the J2EE [239], for example. Typically technologies such as workflow management systems [91], databases, and ERP-platforms (Enterprise Resource Management) [50] are also used for realising the computational services.

The Network Management Agent (NMA) is part of a B2B middleware [141, 165] and provides business context awareness and regulates the operation of the computational service on the basis of local business rules, policies and the contractual information in hand. The NMA utilises a monitoring infrastructure to observe and control the operation of the computational service. Local contract and policy repositories are used by the NMA to store information concerning the contracts, business rules and policies effective in the context of the collaboration.

For controlling and monitoring the operation of the computational service, the network management agent utilizes interception mechanisms specific for the technological domain in question (e.g. an interceptor for SOAP-based [265] service interactions). A *monitor* is an interceptor residing at the interface between the computational service and the client. The task of a monitor is to intercept the communication and to extract relevant information with respect to the collaboration contract, business rules and policies. The monitoring element implementation is partly technology dependent, since it has to be linked with specific communication protocols, such as TCP/IP and

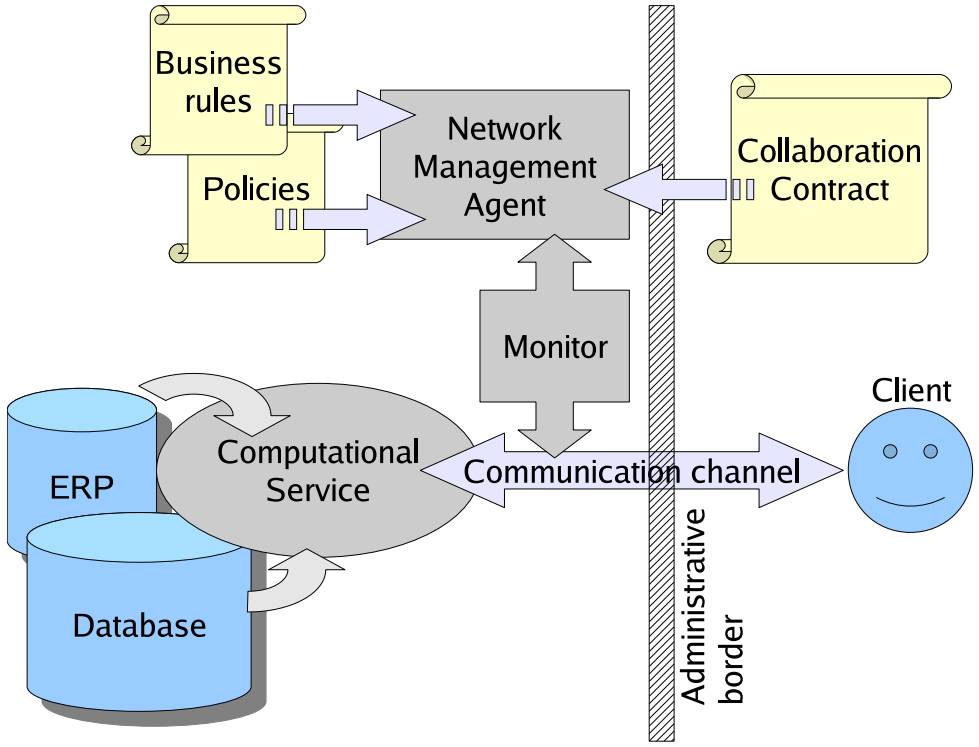

Figure 3.1: An schematic overview of the business service concept.

SOAP. However, the monitoring element should provide a technology independent interface that can be used by the Network Management Agent to inject monitoring rules and to query about the status of the monitor.

As enterprises are likely to join simultaneously multiple business networks with different partners, constraints and requirements, the awareness and management of the corresponding business context must be isolated from computational services as to increase the re-usability and flexibility of services. The *business context* related to a specific collaboration comprises of a business objective, and legislation, business strategies and contracts effective in the corresponding context.

Business objective defines the purpose and goal of action for an entity, individual or community. In the context of inter-enterprise computing, business objective is defined by a set of *business transactions* to be taken, and *organizational policies* and *business rules* restricting and constraining the business transaction activities. A transaction is "*an action or a set of actions occurring between two or more persons relating to the conduct of business, commercial, or governmental affairs*" [251]. A business transaction then defines the set of activities that must be taken by a business entity to achieve a certain business objective. A business transaction is manifested by business protocols and processes that prescribe a certain way to accomplish the intension of a business transaction; these behavioural artifacts are discussed in Section 3.1.2.

An *organizational policy* is a constraint or regulation that directs and influences the decisions and activities of a business entity. Thus policies may affect on the formulation of collaborations, structure and characteristics of business transactions, or interoperation between enterprise information systems. Policies can be specific for a business transaction, an organization (for example access and security policies), or a specific community of collaboration. Organizational policies prescribe rules that take the form of obligations, prohibitions or permissions, for example.

Business rules are declarative rules that define or constrain some aspect of business [86]. Business rules are considered as meta-information describing additional functionality, regulations and constraints over the business transactions and services. They can exist independently of the specific business transactions or services. A business rule may prescribe that certain customers are prioritized over others, for example.

In addition to business objectives, operation of each enterprise or organization in an inter-enterprise community is constrained by legislation, business strategies and effective contracts. Legislation sets the boundaries for doing business by setting regulations, constraints and requirements that state the form of legitimate business transactions in corresponding scenarios. Business strategy of an enterprise may impose additional constraints on the provisioning of services and the nature of collaborations to join. For example, an enterprise may offer different quality of services for different customers depending on their "value" for the enterprise. Direct competitors might not want to collaborate under any circumstances. Finally, the contracts that are already effective within an enterprise might have an impact on the realizability and properties of forth-coming contracts.

Finally, non-functional features can be attached to the communication channels or other co-operation facilities connecting business services and clients. Typical non-functional features are related to communication security, trust, or quality-level of service invocations. Non-functional features can be described using different levels of detail in service definitions. Either platform independent (*"secure communication is required"*) or platform (technology) dependent vocabulary (*"Secure Sockets Layer v3 protocol is required"*) is used for prescribing the types of non-functional features required. Use of a platform independent vocabulary for non-functional features necessitates existence of a shared ontology describing the kind of features available and their relationships with communication channels and other non-functional features. Such an ontology is needed for guaranteeing compatibility of non-functional features with the types of communication channels available, and more over, compatibility of non-functional features with each other.

3.1.2 Business protocols and processes

The primary purpose of a business service is to offer a computational interface for realising one or more business transactions. There are two kinds of descriptions for defining service-oriented business transaction behaviour: business protocols and business processes. While business protocols specify the behaviour of individual business services, business processes are used for defining the behaviour of business service compositions. In this framework both kinds of behavioural artifacts comprise a series of *actions* where an action *"is any activity that is considered as a conceptual entity at the given level of abstraction"* [257]. Business protocols and processes need to be provided with concepts for enabling business protocol evolution, efficient and agile business process development, as well as criteria for service interoperability and consistency of service compositions.

A *business protocol* prescribes the form of business transactions supported by the business service by defining the acceptable bilateral behavioural patterns supported by the corresponding business service. Business protocol is purely behavioural description in a sense that it does not expose the business logic behind the behaviour or the internal state of business service. Business protocols are modular and platform independent characterisations of business transactions that *"represent logically self-contained interaction"* [230].

A *business process* is an abstract (not implementable as such) or concrete (executable) definition of activities taken between a set of participants. Depending on the level of abstraction, business process participants can be such entities as enterprises, roles or technological services, for example, and correspondingly a business process may expose some or all of the internals of business logic and state-handling, such as assignments of values to variables or internal choices

between alternative execution paths. In service-oriented computing two kinds of business processes are distinguished, namely choreographies and orchestration. A *choreography* provides a global description of a distributed business process involving two or more parties. The parties in a choreography are represented using roles and choreography behaviour is then defined as a set of mutual interactions taking place between the roles. An *orchestration* describes a local, centralized business process that implements the behaviour expected from a participant taking a role in some choreography. An orchestration typically involves a mixture of control and data flow dependencies, whereas a choreography usually is more concentrated on the control flow of a distributed business process. Different business process descriptions languages and notations can be used, such as UML sequence or activity diagrams [179], WS-CDL [125], BPMN [180] or WS-BPEL [242].

In the context of federated service communities a business transaction is realised by the business document exchange patterns defined by the business protocols and business processes. A *business document* represents a piece of information that has business value and is used for realizing a business transaction. Definition of a business document type describes the structure and semantics of corresponding business documents. The structure is defined via XML-Schema [263] or other commonly acknowledged document definition language. The semantics for a business document can be given by semantic annotation of the structural definition. Such semantic annotations use vocabulary from shared ontologies or business standards to distinguish concepts that are syntactically similar but should be distinguished semantically.

3.1.3 Business collaboration networks

Business services are utilized in business collaboration networks of varying kinds. The structure, properties and purpose of a business collaboration network is defined in a *business network model*. A business network model comprises basically a set of business roles, a set of relationships between the roles and the collaboration specific properties set for the relationships. More over, the business network model is attached with a choreography that defines the global behaviour of the network in an abstract, non-technical level.

A *business role* prescribes the capabilities and properties that a business partner must fulfill to become a member of a certain collaboration. Business role prescribes the behaviour expected from a participant in form of a business process. In addition, a business role definition may prescribe role specific constraints and requirements for the participant and connections the role is involved with. Such properties may involve business network specific business rules, policies and non-functional properties, for example.

The existence of an explicit business network model is essential for establishing and managing dynamic business service collaborations. The business network model provides the criteria for selecting services from the open market of available services and during the population, as discussed in Section 2.1.1. The role of business network models during collaboration establishment and management is more thoroughly discussed in [141, 139].

3.2 Managing interoperability

Due to the characteristics of modern business networks, interoperation of business services becomes problematic: interoperation must be reached simultaneously at different abstraction levels and between varying features of service-based communities [216]. Especially, interoperability is a multifaceted problem caused by issues surpassing those of technological incompatibilities. The

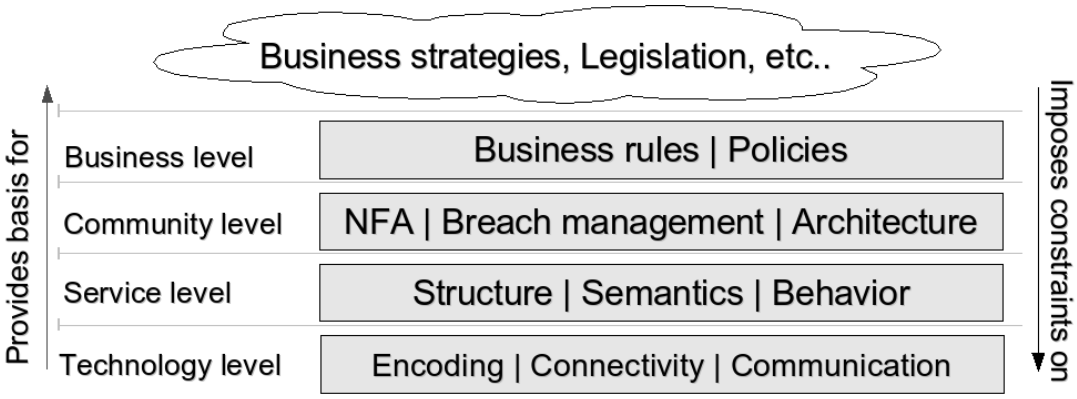

Figure 3.2: Aspects of interoperability

real interoperability challenges are stemming from various sources, such as organisational incompatibilities buried deeply into the structures of collaborating enterprises, architectural mismatches and defective assumptions about business application behaviour, or from the inherent properties of business collaboration models. To establish and support an open model of interoperability, the different features of collaborative service-oriented computing must be identified and analysed. The notion of interoperability must be separated into corresponding independent features, each feature grasping a different need and orthogonal viewpoint on collaborative computing. A clean separation of such features is important for identifying the requirements interoperability imposes on modelling concepts and infrastructure facilities, and for constructing corresponding aspect languages and mechanisms for validating interoperability and consistency of service interactions.

For the purpose of managing interoperability, the features of collaborative computing are classified to distinct layers, each grasping more abstract concepts of interoperation than the previous one. The classification is based on previous studies of interoperability (see for example [253, 75]) and on conceptualisation of enterprise computing environments [142]. Interoperability is addressed conceptually at four distinct layers: *1)* technology, *2)* service, *3)* community, and *4)* business layer. This division is based on identification of the subjects responsible for deciding if interoperation can be achieved. Each level is further divided into different features. The classification as a whole is illustrated in Figure 3.2 and it is discussed in the following subsections.

3.2.1 Technological interoperability

At the technology level, technical interoperability must be achieved between communication and computing platforms by selecting and configuring appropriate middleware services and their parameters. At the technology layer interoperability must be addressed with regards to three different viewpoints of collaborative computing technology, namely *1)* information encoding, *2)* connectivity, and *3)* communication. These viewpoints are strongly aligned with *distribution transparencies* defined in the ODP standardization [112, pp. 5] which can be characterised as the level and quality of distributed computing support provided by the underlying computing platform. Some of the transparencies describe fundamental functionality that are provided by most middleware platforms while others are applied in more specialised scenarios, such as in context of mobile computing.

Information encoding interoperability is achieved when data can be delivered from collaboration participant to another in a way that the contents of the delivery can be further processed by the receiver and attached with meaningful semantics. Such things as serialization and deserializa-

tion, structuring and localization of data need to be taken into account. Encoding interoperability is related to ODP *access transparency* [112, pp. 46] which masks differences in data representation and service invocation mechanisms. Access transparency provides a homogenized view of the basic communication primitives for distributed applications via unification of data representation and messaging formats, and transport mechanisms. In the Web Services technology framework [262] access transparency is provided by XML [267], SOAP [265] and WSDL [55] standards. In the CORBA [178] middleware, access transparency is provided by General Inter-ORB Protocol (GIOP) and its most commonly used refinement, the Internet Inter-ORB Protocol (IIOP). Simple transformations affecting the structure or data value representation of information can be used to resolve identified incompatibilities between encoding schemes; XSL-T [258] is applicable for this purpose, for example.

Connectivity between collaboration participants has to be established before communication can take place. When considering interoperability from the viewpoint of connectivity, such issues as messaging protocol compatibility at different levels (e.g. TCP/IP and SMTP or HTTP), or establishment of routing in ad-hoc networks have to be addressed. Especially in so-called multi-channel service-oriented collaboration networks the potential mobility of both service client and provider has to be addressed. When put in par with the ODP distribution transparencies [112], interoperability with respect to technological connectivity reduces to establishing a shared understanding about the location, relocation and failure transparencies. Location and relocation transparencies detach service clients and providers from static addressing schemes and network topologies. Location transparency *"masks the use of information about location in space when identifying and binding to interfaces"* [112, pp. 47]; this is typically achieved with name-based resolution of resources and service trading mechanisms. *Relocation transparency* is achieved if the relocation operation preserves the bindings between service interfaces such that service clients may interact with the service as before. In service-oriented computing relocation transparency is valuable, since relocations of services may occur due to organizational or technical changes in the enterprises providing the services especially during long-running (business) processes, for example. A system is *failure transparent*, if the failures and recovery mechanisms on the side of service provider are masked from clients. In the context of service-oriented computing, all primitive failures related to the internals of the service should be masked as long as possible; only failures that are declared in the contracts regulating the service interactions are allowed to occur.

Communication interoperability considers the compatibility between the properties attached to communication channels, endpoints and their abstract representations. When considering interoperability from the communication perspective such things as compatibility between platform specific artifacts manifesting the non-functional properties attached over service interactions have to be addressed. More over, the consistency of communication channels has to be validated, that is there should not be conflicting features (e.g. privacy versus monitoring) attached to a channel. Even incompatibilities between different communication schemes, such as RPC [6] or publish-subscribe [14], can be accounted for by attaching appropriate communication protocol adapters to communication endpoints.

Purely technological incompatibilities between languages, interfaces or operational environments can be solved quite efficiently. Methods and techniques like interface description languages [144, 177], adaptors [277, 210], wrappers [162], middleware [185, 239] and middleware bridges [76] have quite successfully been applied for enterprise integration. However, while providing the necessary means for collaboration, technological interoperability and the methods for achieving it are only the basis for providing interoperable business service collaborations.

3.2.2 Service interoperability

At the service level, both technical (compatibility between service signatures) and semantic interoperability (semantics and behaviour of services) between service end-points must be established. Service discovery mechanisms are used for this purpose and the decision making procedures are bilateral. Service level interoperability means capability of interoperation between electronic services with well-defined, self-descriptive interfaces.

Issues regarding the features of service level interoperability have been previously studied among object-oriented and component based approaches [132, 253, 73, 254], for example. Object-oriented interoperability was first addressed in [132] which recognised that interoperability conflicts in object-oriented platforms can not be solved by simple adaption or procedure parameters between heterogeneous objects with use of unifying type systems. It is the overall functionality and semantics of an object which is important [132].

Interoperability at the service level is characterised by three features of services, namely syntactic, semantic and behavioural properties of service interfaces [253, 254]. Validation of syntactic interoperability, that is substitutability of syntactic structures, reduces to a *type matching*. Type matching problem is about finding and defining bindings and transformations between the interface a client wants to use and the interface provided by a service [132]. Type matching problem in general is impossible, since identification of operation semantics and information contents used in the operations or attributes can not be fully automated. However, if two interface signatures are described using the same language (type system) or the interface descriptions can be unified and the problem is restricted strictly to syntactic properties, efficient type matching methods and algorithms can be used, such as developed in [189, 119].

When considering type matching in Web Services based environments, the notion of schema matching emerges (see for example [205, 206]). Schemas describe the structures of business documents and information exchanged in service interactions. When considering Web Services based environments, XML-Schemas [263] are used for describing such document structures. The type system behind XML-Schema mixes both structural and name-based features [228]. This makes XML-Schema matching a bit complicated and the type system less elegant, since purely structural matching methods can not be used.

Semantic feature of service level interoperability is concerned with the meaning of service operations and documents. Matching of service interfaces based on their operational semantics have been addressed for example in [279]. Operational semantics are usually attached to a service as operation-specific pre- and post-conditions (or effects). These conditions are definitions given in appropriate logic describing the assumptions and results of the operations.

Semantics are use also for attaching meaning for information contents exchanged between services. In tightly coupled and closed systems interpretation of semantics is implicitly coded into the applications, since the operational environment is known during development of the application. Exploiting explicit shared ontologies for description of operation and information semantics provides a more loosely coupled approach. Attaching semantics to service operations and messages for establishing interoperation of services sharing a common ontology is the approach taken for Semantic Web services [161, 195]

Interoperation of behaviourally typed artifacts reduces to inter-related concepts of *substitutability* and *compatibility* [255]. Substitutability means that two business services can be replaced by each other. Compatibility means that two services can co-operate in a meaningful way if they are connected together: their interplay does not lead to a locking situation and the types (structure and semantics) of communicated messages are the expected ones. The concepts of substitutability and compatibility, and rules for verifying these properties have to be provided by

the service-typing discipline. Substitutability is manifested in the service-typing system by type equivalence and subtyping rules. Compatibility of business service behaviour is provided with use of the notions of session type duality and subtyping [255], for example.

Attaching behavioural descriptions to interface signatures provides stricter guarantees of application interoperability. When only syntactic and semantic features are considered, we cannot clearly specify how the service should be used. If a formal specification of application behaviour is attached to its interface, we can both validate that the interoperation between two applications is compatible or that service behaviours implemented by two distinct components are behaviourally equivalent [277, 48]. For example process algebras can be used to formalise service behaviour. Given formal semantics for service behaviour, several interesting properties from a service and service interoperation can be verified, such as absence of deadlocks or behavioural similarities.

3.2.3 Community level interoperability

Interoperation between distinct services does not guarantee that functionality of the whole system is consistent and flawless, but requirements and constraints for interoperation are induced also by the global properties of the community. For establishing community level interoperability the features of architectural properties, failure handling procedures and compensation processes, and non-functional properties of service-oriented communities must be addressed. Decision making at the community level is multi-lateral since the properties of all the participants must be taken into consideration. Negotiation mechanisms are used for populating communities with compatible services. Both technical (non-functional properties) and semantic interoperability (breach management and community architecture) is addressed at the community level.

Interoperability at the community level must be guaranteed with respect to non-functional properties, breach management mechanisms and architectural properties of the community. Interoperation is established by multi-lateral negotiations during the population process which results in a mutual agreement between the community participants. Community level interoperability grasps rest of the semantic interoperability features in enterprise computing environment, in addition to those addressed at the service level.

Non-functional properties, such as quality of service, security, trust, location or availability is are important features of service-oriented communities that need to be addressed for achieving interoperability at the community level. Mutually agreed values for non-functional properties are used for configuring communication channels and middleware services as well as supervised during community operation by the monitoring facilities. Simple error handling, such as service exception handling, is usually provided and agreed in the technology and service levels. There are also more severe, business related errors that manifest themselves as contract breaches. For managing the different kinds of failures that may contradict with the objectives or success of a service-oriented community, the breach management mechanisms have to be agreed between the participants of a community. Typical failure cases and handling mechanisms are failures in meeting deadlines or business objectives, and compensation processes over contract breaches, for example.

Architectural features contain such properties as the topology of a community, composition of services into business roles and coordination of services across the community. Mismatches in architectural properties of communities can be caused by faulty assumptions about other components, connections between components or topology of the community [88]. Architecture description languages such as Wright [3], Darwin [160] or Rapide [156] have been developed for defining software architectures. These languages formalise architectural properties, thus making it possible to automate validation of architectural interoperability. Standardisation of business community ar-

chitectures and business cases has also been used for providing architectural interoperability. This is the approach taken in ebXML [133] or RosettaNet [211], for example.

3.2.4 Business level interoperability

While technical and semantic interoperability is addressed by the technological, service and community layers, pragmatic interoperability manifesting the needs and intentions of autonomic enterprises is addressed at the business layer. Properties stemming from enterprise strategies, legislation and intentions of different organisations manifest themselves at the business layer as concrete *business rules* and *organisational policies* (or just policies). At this layer, compatibility between business rules and policies affecting service interactions and collaboration processes must be agreed upon. Both business rules and policies are primarily organisational knowledge that are independent of community or service life-cycles; thus it is necessary to separate these elements from the other features related to community and service level interoperability.

Business rules are declarative rules that constraint or define some aspects of business [86] and thus may modify functionality of the services offered by an organisation. They are part of the organisation's business knowledge which direct and influence the behaviour of the organisation [7]. Typical examples of business rules are different kinds of service pricing policies or regulations on service availability based on customer classifications. A business rule may affect the non-functional or behavioural properties of services by constraining the choices for non-functional properties of a service or by introducing new kind of behaviour during service operation. To achieve automated validation of business rule interoperation, the business rules must be expressed using a feasible logical framework. Conceptual graphs [252] and defeasible logic [7] have been used for modeling of business rules, for example.

Organizational policies regulate the use of business functionality and knowledge provided by an enterprise and may also internally control the observable behavioural of provided business services. More over, organizational policies constraint community behaviour such that the common objective of the community can be achieved [234]. Rules addressing accessibility, authorization, trust and privacy with respect to the provided business services and information are typical examples of organizational policies.

Organisational policies declare autonomic intentions of organisations and they are typically specified by using concepts of obligation, permission and prohibition [234]. An obligation is expresses that certain behaviour is required whereas permissions and prohibitions express allowable behaviour [234]. Policies may thus modify behaviour of services by requiring certain actions to be taken instead of the others, or by prohibiting certain actions to be taken. When organisational policies of collaborating participants are known beforehand, policy conflicts can be identified before community operation. If behavioural descriptions are given using an appropriate logic, interoperation of organisation policies with respect to the behavioural descriptions can be verified for example with model checking. However, organisational policies are inherently dynamic entities and not even necessarily published outside the organisations. Organisational policies are one primary cause for the dynamism in enterprise computing environments.

3.2.5 Establishing horizontal and vertical interoperability

Identification of orthogonal features of interoperation is a crucial for managing interoperability in federated service communities. However, it is not sufficient by itself and requires as a companion a characterization of how interoperation can be maintained within and between the distinct domains

of concern. For this purpose, the concepts of horizontal (or intra-feature) and vertical (inter-feature) interoperability is discussed in the following.

Horizontal interoperability characterizes the relationships guaranteeing interoperation between resources in the same domain of concern, such as between business rules of two collaborating partners. Horizontal interoperability problems are typically incompatibilities induced by heterogeneity between the corresponding resources. When addressing interoperability issues at the service layer, behavioural compatibility is an evident intra-feature interoperability criteria, for example.

At the technology level, technical interoperability must be achieved between communication and computation platforms by addressing issues related with connectivity, communication and encoding. Interoperability is established at the technology level by integration and unification of technologies and provides the foundation for interoperation at the higher levels. When a technology level interoperability problem is encountered during the operation of a community, a compensation or error handling action is needed to be performed outside the conflicting technology domain to fulfill the objectives of the community.

At the service level, both technical (compatibility between business service and document structures) and semantic interoperability (semantics of information and behavior of business services) between service end-points must be established. Considering the information exchanged between business services, technical and semantic interoperability can be established with homogenization of the vocabulary. Standardised document description languages such as XML [267], vocabularies (e.g. ebXML [133] or RosettaNet [211]), common taxonomies such as North American Industry Classification System (NAICS) or shared ontologies such as those based on the OWL [259] can be used for this purpose. The behavioral features of business services can be formalized using methods such as pi-calculus [166] or Petri-nets [196]; accompanying analysis methods and tools can be applied for *a priori* interoperability validation.

For establishing interoperability at the community level, the features of architectural properties, breach handling, and non-functional properties of communities must be addressed. Interoperability of technical (non-functional properties) and semantic kind (breach handling and community architecture) are addressed at the community level. Non-functional properties addressed at the community level contribute to the technical and semantic features of interoperation. For example, requirements for security or quality of service are addressed by the non-functional properties at the community level. Non-functional properties of collaborations contributing to the pragmatics of interoperation, such as organisational policies, are addressed at the business level.

At the community level, deciding about the interoperability management methods has to be a joint process, since the properties of all the participants must be taken into consideration. Negotiation mechanisms are used for this purpose and the mutual agreements about the properties concerning the features of interoperation are formalized into a collaboration contract. Negotiation is used in this scenario as an *a priori* solution for homogenizing the view on the collaboration among the participants; the resulting contract provides a homogenized model of collaboration with respect to the community operation. Compensation procedures agreed with during the negotiations are used as mechanisms for resolving incompatibilities encountered during the operation of the community.

At the business level, pragmatic interoperability which considers business rules and organizational policies, is addressed. These rules comprise the autonomic intention and character of enterprises. Both policies and business rules are metainformation entities that are independent of community or service life-cycles; thus it is necessary to separate these features to an independent layer distinguished from community and service level interoperation. Constraint satisfaction algorithms [135] can be used for finding a compatible intersection of policy values between enterprises, for example. Horizontal interoperability problems at the business level are instances of

more general policy conflicts. Policy conflicts can be resolved if *meta-policies* which state modality precedences (ordering between negative and positive predicates) or priorities among business rules [124, 95] have been defined. This prioritization approach can be applied to both business rules and organizational policies.

Vertical interoperability concerns inter-dependent resources of different kind. Characterization of vertical interoperability is provided by a set of relationships, consistency rules, and conformance criterion between domains of concerns residing at different abstraction levels. Vertical interoperability is managed in general using conformance validation procedures where an interoperability feature residing at the higher level of abstraction is considered as a specification that must be fulfilled by some interoperability feature at the lower level of interoperability management abstraction. For example, when organizational policies are represented using deontic logic, conformance between a policy and service behavior can be validated using model checking [186]. Concretely, these specifications could be formalized as rules that a monitoring system will use at runtime to check a whole system consistency. As soon as each requirement expressed within an feature is correctly identified and formalized, conformance points allow to check that there are no predictable misalignments among different features, leading to interoperability problems.

3.3 Service-oriented engineering of collaborative systems

Service-oriented software engineering (SOSE) is a software engineering discipline which utilises constructs and concepts conforming with the service-oriented computing (SOC) paradigm [231, 191] for designing, modelling, developing and managing open, highly distributed computing systems. The key ingredients of service-oriented computing are services, service descriptions, service composition, and service-oriented architectures (SOA). In this context, services are considered as autonomous software components with well-defined interfaces that are advertised by publishing their descriptions in service brokers. Service descriptions are produced by service providers and they characterize the properties and capabilities of corresponding services. Service consumers use service discovery mechanisms provided by service trading infrastructure to locate appropriate services. Service trading and discovery mechanisms are few of the utility services provided by a SOA middleware.

The development processes of SOSE are description centric: models describing the behaviour and properties of individual services and service-based systems are not used just in design and modelling phases but they are utilised extensively also later in the engineering processes and during runtime. Service discovery facilities provided by a SOA middleware and service composition with choreographies and business processes are used as the primary means for delivering the system functionality. More over, the actual development processes in a SOSE framework necessitate tools that provide agile and flexible development of interoperable services and their compositions. For this purpose, formal models of behaviour and relationships between such models characterizing service interoperability and business process consistency are needed.

A SOSE framework necessitates four essential elements: *1)* infrastructure services for knowledge sharing and facilitating a service-oriented computing environment, *2)* a selection of service and business process engineering tools, *3)* metamodels and ontologies for conjoining engineering processes, infrastructure and different actors within the framework, and *4)* formal methods for guaranteeing the consistency and correctness of service and business process development activities.

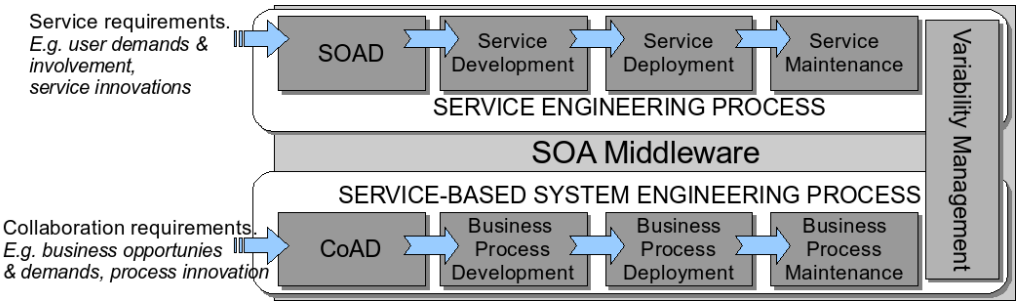

Figure 3.3: A framework for Service-Oriented Software Engineering.

3.3.1 A framework for service-oriented software engineering

A service-oriented software engineering framework comprises software engineering processes delivering artifacts such as service specifications, service components, and business processes. The engineering processes in SOSE framework are model-centric in a sense that the abstract models of systems, their components and properties are considered as first-class and primary engineering artifacts. Model-Driven Engineering (MDE) [225] paradigm is thus a natural choice for facilitating the SOSE processes. Modelling notations and languages familiar from MDE approaches (such as UML [179]) are utilized in SOSE processes for defining the models. Similarly, model transformations and weaving, aspect-oriented modelling and code generation can be applied during the SOSE processes for refining the models and generating implementation artifacts from them. The permanence of models through service-oriented system life-cycle, their inseparable role in SOA middleware functionality, as well as their role in managing the dynamism and heterogeneity of service-oriented computing environments [141] emphasize the importance of models even more than in the context of traditional MDE.

A SOSE framework necessitates a selection of supporting activities that provide horizontal elements of functionality and knowledge that can be utilized for tailoring available services and business processes for reuse in different contexts. Functional elements include for example transformation and weaving models [28] for service development and adaption, as well as feature or aspect models characterizing non-functional properties [271, 131]. The knowledge elements may include such artifacts as ontologies and ontology mappings, definitions of shared vocabularies and so on.

Finally, SOSE framework is facilitated by horizontal utility services that realize a service-oriented computing environment and an ecosystem where the preceding artifacts can be shared and utilized in an applicable manner. The utility services are provided by a SOA middleware and include such services as different metainformation repositories and provide service discovery functionality, for example. The Pilarcos interoperability middleware [141, 139] provides the basis for the SOSE framework introduced below by delivering infrastructure services for sharing metainformation (business network models and service definitions), service publication and discovery, collaboration establishment, eContracting, and trust and reputation management.

The SOSE framework illustrated in Figure 3.3 comprises two engineering processes, namely a *"Service Engineering"* process and a *"Service-Based System Engineering"* (SBSE) process, each comprised of a series of engineering phases. The supporting activities are included under a notion of *"Variability Management"* facility familiar from software product line engineering (SPLE) disciplines [201].

The individual processes, and the individual engineering phases and activities included in the

SOSE framework are loosely coupled. The processes are related by a the shared SOA middleware and variability management facilities. Each engineering phase within a process is related to the previous phase through the SOA middleware, i.e. the output provided by a distinct phase is published using the utility services, after which they can be fetched from the environment when needed. The loosely coupled approach to software engineering processes, and the fundamental role of a SOA middleware as part of SOSE processes are probably the most striking differences when compared to more traditional software engineering disciplines. These preceding elements of the SOSE framework are discussed briefly in the following.

3.3.2 Service engineering process

Service engineering aims for efficient delivery of services for the purpose of enabling loosely coupled service collaborations. The service engineering process produces specifications and implementations for individual services based on the set of service requirements stemming from user demands, business process requirements and service-level innovations. A preparatory business case analysis and planning phase is presumed to happen before the initiation of a service engineering process for identifying the requirements and motivating the development of individual services. In the first phase of the software engineering process a service-oriented analysis and design (SOAD) is conducted which produces service specifications, including the corresponding business protocols complying with the given requirements. During service design such issues as service cohesion and coupling, or technology independence must be addressed and taken as guiding principles of the design activities [193, 203].

During service development the specifications provided by the SOAD phase are utilized for developing service implementations. An MDE-based [225] development process is followed during which business logic is injected to the generated software components. As the final activity of the service development phase, a technology specific *service description* characterizing the capabilities of the service implementation is made available, possibly even published using the service broker functionality of the SOA middleware.

While business protocols are taken as quite stable within a certain business domain, there will emerge situations where business protocols need to be extended or specialized. So-called *business protocol refinement* is needed in such situations to address specific requirements emerging from customer needs or business requirements, for example. Business protocol refinement is used for introducing behavioural variability in service engineering processes. Especially, such refinements must preserve business protocol compatibility, that is refinements that contradict with previously established service interoperability guarantees should not be allowed.

In service deployment a service implementation is installed to a technological platform: resources are reserved for the service and service-specific provisioning decisions are made. After the service has been deployed, a *service offer* refining the service description can be published. A service offer includes, in addition to the technical details included already in the service description, provisioning information, such as the price for service usage and terms of use. Finally, in the maintenance phase service implementations are executed and monitored.

3.3.3 Service-based system engineering process

Service-based system engineering (SBSE) process aims for agile development of business processes and produces descriptions of service choreographies and orchestrations. The SBSE process is catalyzed by a collaboration requirement stemming from a particular business context. New business opportunities enabled by technological innovations, business demands such as in-

sourcing or out-sourcing needs, as well as business process level innovations and optimization are probably the most common sources for collaboration requirements. Service-based system engineering strives for enabling successful and efficient business collaborations.

The SBSE process begins with collaboration analysis and design where business requirements are analysed and choreography descriptions are produced. Collaboration analysis comprises such activities as business case analysis and actor identification. The results of these activities are used during choreography design for identifying roles, designing the overall behaviour of the choreography and annotating the choreography elements with non-functional requirements where necessary.

Business process design produces a business process description complying with the functional and non-functional requirements of the choreography role taken as an input. During business process development this description is refined to an executable orchestration.

Two kinds of business process refinement activities are required by the SBSE process: *abstraction refinement* and *aspectual refinement*. Abstraction refinement is an essential part of any software engineering process based on the Model-Driven Engineering [225] approach. During abstraction refinement actions included in a business process are transformed to more concrete ones. Actions (and behaviour) can be expressed using three abstraction levels coinciding with the MDA [84] framework, CIM (Computation Independent Model), PIM (Platform Independent Model) and PSM (Platform Specific Model), for example.

In modern MDE-based software engineering approaches aspect oriented modelling [229], feature modelling [209] and model weaving are utilized for refining functional models with cross-cutting aspects, such as QoS dependability support [131]. Such aspects may introduce additional structure and behaviour to business processes. Behavioural refinement is used for weaving the behaviour required by a certain aspect to the "core" business process. These kinds of refinements are called from now on as aspectual refinements.

Finally during business process provisioning and deployment phases the orchestration implementation is attached with auditing, billing and management operations, for example. The business process is then deployed to a local execution environment where it is executed and monitored.

3.3.4 Variability management activities

The *Variability Management* activities provide artifacts for *1)* service and business process development purposes, *2)* reusing individual services in different contexts, and *3)* business knowledge unification. During service and business process development, model refining transformations are utilized for generating more concrete models and implementation components from models described at a higher level of abstraction. Also models describing specific platforms and runtime environments as well as model based patterns capturing the service deployment best-practices [10] can be valuable variability management artifacts for development purposes. Service reuse is facilitated by model transformations for service adaption, whereas aspect models and weaving models are utilizable for late encapsulation of services with non-functional properties. Finally mappings between ontologies and vocabularies can be published for unifying heterogeneous business knowledge.

3.4 Infrastructure services and tools for inter-enterprise computing

The federated service community approach requires a selection of infrastructure services and tools for managing the loosely coupled collaboration establishment and service-oriented software engineering processes. In the following the archetypes of such facilities are identified and their

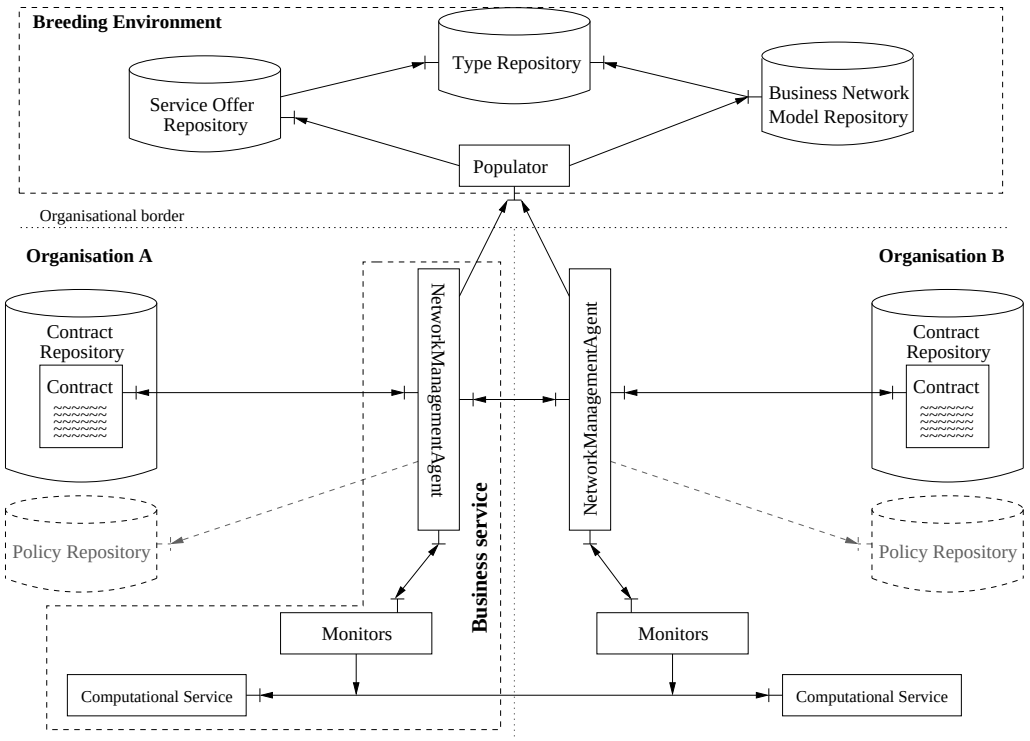

Figure 3.4: The Pilarcos interoperability middleware (slightly adapted from [140]).

properties are discussed. Section 3.4.1 introduces the infrastructure services needed while the essential tools for facilitating the SOSE processes are discussed in Section 3.4.2.

3.4.1 SOA middleware services

In the following we represent the infrastructure services provided by the Pilarcos interoperability middleware [141, 139] which represents a modern, business-oriented approach for managing the complexities of electronic business in open service markets. The Pilarcos framework [141, 139] provides an environment for establishing federated collaborations between autonomous enterprises. Interoperation is based on utilization of shared meta-models, such as Business Network Models defining the collaboration structures [141], service offers describing business service capabilities delivered by an enterprise, and service types [217].

A set of interoperability service utilities consisting of meta-information repositories, populators [139], network management agents [165] and monitors is needed for managing the life-cycle of an electronic business collaboration. The interoperability services provided by the Pilarcos middleware are illustrated in Figure 3.4. In this figure, boxes represent active services, cylinders are meta-information repositories, and arrows represent functional dependencies between the entities. The network management agent, the monitor and the computational service providing the actual business logic constitute a *business service* which is aware of its business context, that is the collaboration scenario, business rules, and organizational policies affecting its operation.

The collaboration establishment process discussed in Section 2.1 and illustrated in Figure 2.1 consist of three phases, namely service selection, collaboration population and negotiation. In the service selection phase a service discovery functionality is needed for identifying and locating potential service providers from the open service markets. Service discovery functionality is part

of a *service trading* function provided by the SOA middleware. A service trading function provides a repository for advertising and discovering business services. The trading mechanism is an implementation of a broker pattern variant [43] that is commonly utilised to decouple component interfaces in distributed systems. It has been utilized for example in the ODP standardisation [110], the related OMG CORBA middleware platform [178], and most notably in the Service-Oriented Computing paradigm [191, 231] and Web Services architecture [262].

The service trading mechanism consists of service offer repositories and type management infrastructure. Service offers (advertisements) are exported into service trading infrastructure by service providers. A *service offer repository* can be used for maintaining a consistent view on the available set of services and it provides operations for publishing new business services and querying the available business services.

Service offer repository is closely related with a *type repository* which maintains type specifications and their relationships [112]. A type specification in the context of federated service community refers to business service specifications and business document specifications, especially. A type repository provides operations for publishing and querying type specifications, asserting relationships between type specifications, and querying relationships between a given type and other types [108], for example. The primary purpose of a type repository is to maintain interoperability in an open distributed computing environment by providing consistency and conformance validation functionality for business service publication and discovery. The service offer and service type repositories are illustrated in Figure 3.4 among the other Pilarcos middleware [141, 139] services.

During collaboration population a *populator* [141, 139] is used for establishing interoperable communities. The main purpose of a populator is to construct a set of collaboration contract templates from the business network model and the set of dependable services given. The populator selects the most suitable service offers for roles described in a business network model; this is achieved by solving a constraint satisfaction problem between the prerequisites of the business network model and service offers [139]. Business network models are published in *business network model repositories*. When reflecting the features of interoperability discussed in Section 3.2, the population process addresses technological and semantic interoperability at the community level.

In the negotiation phase facilities are needed for providing the community participant candidates with mechanisms for further deciding about the form of collaboration to be. In the Pilarcos framework, a *Network Management Agent* (NMA) is used for representing a community member during the collaboration negotiation phase [141, 139]. The NMA is responsible for handling negotiations with the other potential members of a community and possible re-negotiations during if members are changed during the operation of the community. The NMAs also maintain information about the state of the community progress and determine the suitable reactions to activities taking place in the community [165, 139].

Finally, after successful negotiations between the partners an electronic collaboration contract, *eContract*, is established. The operation of the resulting business collaboration network is governed by the eContract which is distributed to all the participants of the community [165]. Monitoring infrastructure is utilized for validating the correspondences between collaboration contract, the actual behaviour of business services, and local business rules and organizational policies. Communication protocol specific interceptors are used to fetch communicated messages before they leave or enter an enterprise computing system. Outbound monitoring where the enterprise oversees its own behaviour is needed, since the underlying applications implementing the business service functionality may not be aware of changes in business rules or policies. It is part of the functionality of a NMA to validate the behaviour of a business service against the current

policies. Inbound monitoring is needed to verify that the partners comply to the corresponding eContract. Meta-information provided in the eContract is utilised by the monitoring infrastructure to validate conformance between the expected and actual behaviour of business services.

Trust and reputation management systems [213, 214] are needed in a truly open collaboration environment for further restricting the selection of services and service providers to be considered for participating communities. More over, the metainformation included in the different kinds of repositories (e.g. service offer and service type repositories) need to be trusted such that the collaboration establishment process can not be contaminated by malicious actors. In the Pilarcos middleware, trust management is provided by a dynamic combination of experience information and a subjective analysis of the situation in which trust is needed [139]. That is, the SOA middleware of the Pilarcos framework provides facilities for establishing decisions about trust relationships during community population processes. Trust decisions are made during the community negotiation phase in the Pilarcos framework [139].

3.4.2 Tools for service-oriented software engineering

A set of tools is needed for facilitating service-oriented software engineering processes and their supporting activities. Such a set of design and development tools use the SOA middleware for managing the meta-information related to the development activities, such as for discovery of service definitions, business network models and model transformations. In the following an archetypical tool-chain for service-oriented software engineering is characterized by identifying the essential elements of such framework. The exemplary SOSE process that is discussed is a bottom-up process that utilizes service types in many of its phases. Service types and service typing is discussed more thoroughly in Section 6.2; in the context of this section it is only needed to know that service types provide definitions for kinds of business services. The tool-chain described utilizes the Pilarcos middleware services discussed previously.

In the Pilarcos framework service types are used for defining services and they provide an abstraction of business services for software engineers, enterprise modelling experts and system administrators to establish interoperable collaborative systems. Service types are used for interoperability validation during the design of business services, and by code generation and conformance validation tools during the implementation phase. For business network modelling and population purposes service types provide means for interoperability verification between business services and networks. On deployment phase and operation of business networks service types are used to establish conformance validation. The tools and platforms used in different tasks utilise public infrastructure services through Pilarcos middleware interfaces.

Service types are products of the software engineering design phase and their construction is supported by business service modelling tools. The design phase captures the characteristics of the business service that are determined by application domain, intended usage patterns and optionally by pre-existing service types. This interdependence is illustrated as a two-way meta-information exchange between design-time service modelling tools and public type repositories in the Figure 3.5. Behavioural description provided by a service type can be utilised for simulation and prototyping purposes. When the business service designer is satisfied with the properties of the service type, it is published in a public type repository. Before the new service type can be published, it has to be checked against the typing rules of the targeted service type repository. This functionality is to be embedded also into the service modelling tools.

In the implementation phase published service types are used to aid business service development. Conformance and compatibility validations between a service type and the implementation interface are used to guarantee interoperability with eligible business services. Service types can

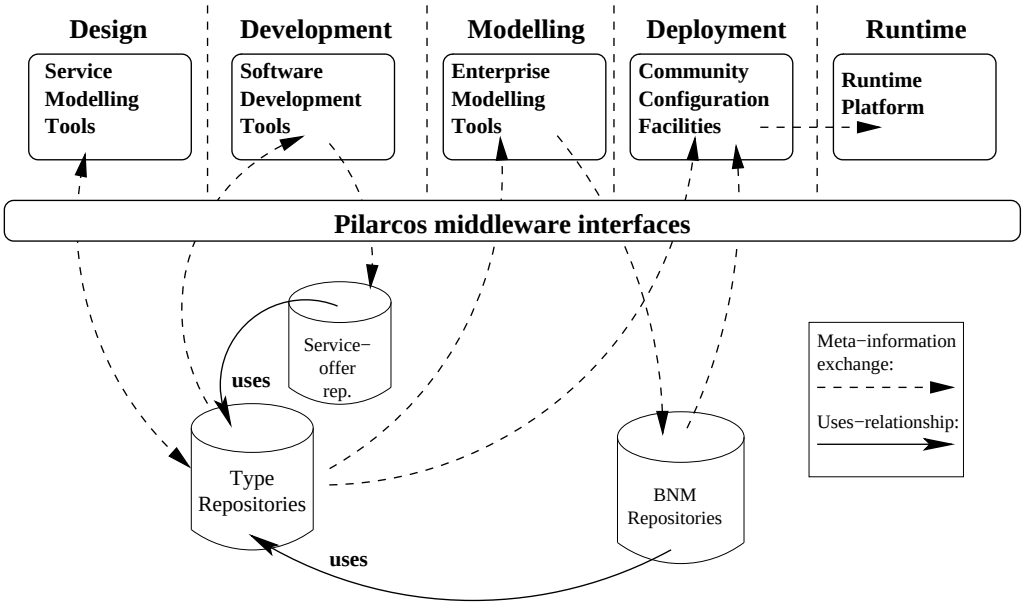

Figure 3.5: Use of service types during business service life-cycle.

be used as templates for the implementations. A suitable service type is fetched from a type repository, after which generative methods can be used to provide skeleton code for the service implementation from the service type. This usage pattern of service types is similar to the MDA [84] approach; service types can be considered as a platform independent model (PIM) of business services. Service types could also be used to generate dummy implementations for testing the implementation with respect to other business services.

When the new business service is published in a public service offer repository, its behavioural properties and especially its conformance to the claimed service type are verified. Each service offer must correspond to a service type and when a service offer is exported to the system, its conformance with respect to the claimed service type is validated. Service offer repositories use type repositories to accomplish this task. This procedure guarantees type safety in them by ensuring that no ill-typed business services are published.

Service types are re-usable design components. Their primary purpose during the business network modelling phase is to provide well-structured design elements to be used with enterprise modelling tools. The structure and properties of a business network are modeled in Business Network Models (BNM). Further details of BNMs can be found from [142]. During the modelling phase, service types are utilized for composing business roles. Business roles are attached to each other with connectors to describe the topology of the business network.

Enterprise modelling tools use type repositories for fetching interoperable service types. Type checking and matching operations provided by type repositories are exploited when verifying service type interoperability and searching for applicable service types to use in the business network model. The resulting business network model is published in a public BNM repository. During publication the corresponding business network model is verified using the validation functionalities provided by type repositories.

Two different kinds of activities can be identified from the deployment phase of federated service communities: community breeding and local computing infrastructure configuration. In the community breeding process a business network model is used as a template to be populated with

business services [140]. During service discovery appropriate business services are located from service offer repositories via their service types. However, the client is sometimes not interested in business services with a specific service type, but instead looks for business services which offer a certain kind of functionality. To serve this purpose, a service offer repository may exploit type matching functionality provided by the type repositories to return services of matching type. A matching business service provides the same functionality semantically, yet it is not exactly of the same service type.

As federated service communities are very loosely coupled systems and their collaboration is based on negotiated contracts, runtime monitoring is needed for catching any erroneous behaviour and for establishing coordination of the community. For the local computation infrastructure configuration, service types provide the means for generating the monitoring components or configuring monitors with appropriate rules.

Chapter 4

A modelling framework for service-oriented software engineering

Service-oriented computing is a model-centric paradigm of distributed computing where service descriptions and business process descriptions, among other models describing system artifacts and properties, play a fundamental role. Due to the loosely coupled nature of service ecosystems, the service-oriented software engineering processes, production artifacts used in those processes and models related to the service-based system life-cycles have to be described rigorously. In addition, harmonized and dependable knowledge about the best practices within a domain, and standardized vocabulary for enabling exchange of business knowledge is needed for facilitating loosely coupled business collaborations. Software engineering methodologies based on the MDE [225] discipline require modelling languages and tools for encouraging effective use of different sorts of models and model transformations during the corresponding development processes.

Thus, it is not a surprise that the role of models, modelling languages and tools, ontologies, and infrastructure facilities for knowledge management become invaluable for service-oriented electronic business collaborations. The knowledge elements that are required for establishing service-based business collaborations can not be pre-selected and open world assumption definitely holds for service descriptions and other kinds of meta-information required for establishing loosely coupled collaborations.

In the following we elaborate on the notion and role of model-driven engineering in the context of service-oriented computing environments. While the basic setting of model-driven engineering was already discussed in Section 2.3, the discussion that follows elaborates on the principles of model-driven engineering and points out some more specific topics relevant for managing the complexity of loosely coupled service-oriented computing environments. Especially, we discuss the role of two different sorts of models, namely system models and ontologies, and reflect upon their commonalities and differences. Finally, the implications of the preceding principles to the management of business and interoperability knowledge is discussed.

Section 4.1 elaborates the previous discussion about modelling disciplines by introducing the concepts of linguistic and ontological metamodelling. Foundational metamodelling relationships, such as model conformance, system representation and model extension [16] are also described. After that, the modelling framework is characterized in Section 4.2 as a set of inter-related metamodels. The set of metamodels includes models for describing service-oriented software engineering methodologies, domain ontologies, and knowledge management facilities required for facilitating the processes and the service ecosystem. The metamodels for describing methodologies and knowledge management artifacts are described in detail while the metamodels for so-called

domain ontologies and reference models are discussed in later Chapters.

4.1 Foundations for metamodelling practices

In service ecosystems explicit and formal models are needed on the one hand as specifications for enabling efficient and dependable software production practices, and on the other hand for describing the vocabularies and concepts used in some specific business or technology domain. Models in general can be distinguished into two different sorts based on their purpose and role with respect to the systems they represent, namely *system models* and *ontologies*.

System model is a *prescriptive model* that is used to specify and control the system under study [11]. A system model gives a specification of the system that must be conformed to by the corresponding implementations. Model-driven software engineering typically emphasizes the use of prescriptive models for facilitating development processes where abstract models of the system are first designed and then refined during the process to more concrete models and development artifacts.

An ontology is a *descriptive model* used for characterizing the existing world, the environment and the domain of the system [11]. An ontology is inherently associated with with an open-world assumption: anything that is not explicitly stated remains unknown. Especially, two different systems (models) may satisfy an ontology if they differ in areas that are not explicitly mentioned in the ontology [11]. This is in contrast with the system models that completely specify a system within the limits of the corresponding point of view and the abstraction level used.

A modelling framework for modern service ecosystems must provide mechanisms for both system modelling and ontology engineering purposes. In the following subsections we introduce the essentials of modern metamodelling disciplines. First Section 4.1.1 describes the foundational concepts for a metamodelling framework. After that in Section 4.1.2 we introduce the notions of linguistic and ontological metamodelling which provide means for unifying system models and ontologies. In Section 4.1.3 we discuss briefly some issues related to management of metamodelling information.

4.1.1 Models and metamodelling relationships

For defining the meaning of models, metamodels, ontologies and the corresponding metamodelling relationships we build the notion of a model to a classification of systems presented in [77, 78, 89]. A *system* is considered as the primary element of discourse in the context of MDE [78]. Systems can be classified to *physical systems*, *digital systems* and *abstract systems*; this classification is not complete but represents a partitioning sufficient for discussing the principles of model-driven engineering. Physical systems are *"observable elements or phenomenons pertaining to the physical world"* and represents things such as "a travel agency" [78, 89]. Digital systems are those systems that *"reside in computer memories and are processed by computers"*, thus providing a digital representation (e.g. in XML [267] format) of some system. Finally, abstract systems are *"ideas and concepts that eventually reside in human mind to be processed by human brains"* [78] and represent concepts and their relations from the biological domain, for example [89].

A system can play the role of *model* with respect to another system [77]. That is, when a system is considered as a *representation of* another system, we say that the first one is a model of the second one, the *system under study*. A model represents the system under study in a way that satisfies Liskov's principle of substitutability [16] Liskov's principle [151] states that two entities are substitutable with each other if and only if every property that can be proved about an

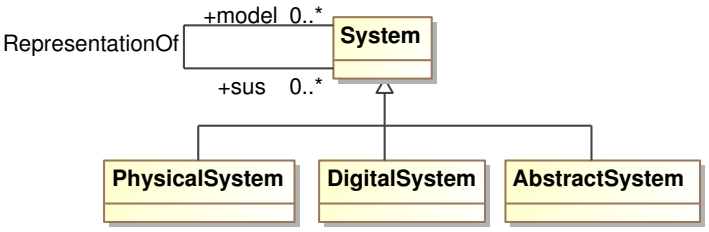

Figure 4.1: Classification of systems [77].

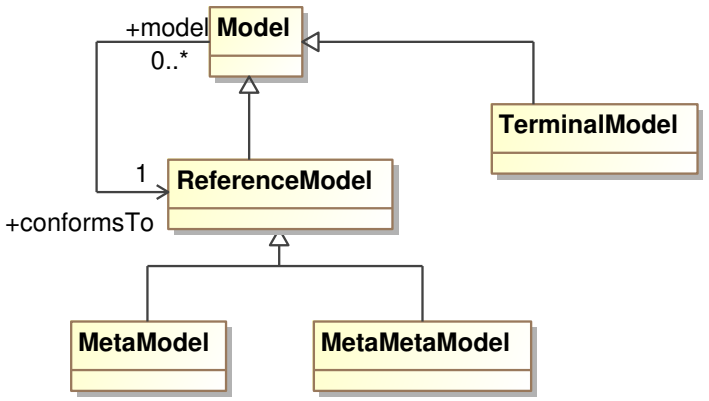

Figure 4.2: Relationship between models and reference models [121].

entity can also be proved about the other. The classification of systems and the *RepresentationOf* relationship between systems is illustrated in Figure 4.1.

From the linguistic perspective there are two different kinds of models residing at separate linguistic levels that are used for practising model-driven engineering. At the type level so-called *reference models* [121, 247] are used for defining modelling languages. Reference model defines the typing rules for models, that is the kinds of model elements and the way they can be arranged, related, and constrained [30]. A reference model thus specifies the rules and the language for describing corresponding kinds of models. At the instance level, models are constructed in conformance to a reference model. A model M is said to conform to its reference model RM if and only if each model element in M has its corresponding metaelement in RM [30]. The relationships between a model and its reference model is illustrated in Figure 4.2.

While the relationships between models and their reference models do not constraint the number linguistic levels, typically three linguistic modelling levels are used. The models residing at these linguistic levels are commonly known as *terminal models* (or just a model when the distinction is clear from the context), *metamodels* and *metametamodels* [84, 121]. As illustrated in Figure 4.2, a terminal model is a model that conforms to a metamodel. A metamodel is a model that conforms to a metametamodel, which conforms to itself. Such a three-level, closed metamodelling hierarchy is followed by the OMG's Meta-Object Facility (MOF) framework, which defines a metametamodel called the MOF model [181], for example. In the MDA framework the UML [182] modelling language is defined using a metamodel conforming to the MOF model.

In addition to the conformance and representation relationships discussed briefly above, there

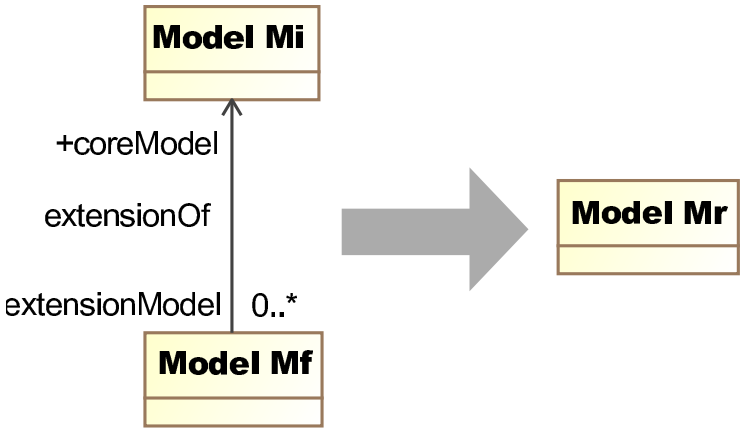

Figure 4.3: Model extension [16].

is a third relationship relating models that is applicable for facilitating model-driven engineering, namely the relationship of *model extension* [16]. Model extension is used for deriving new models from base models and it is especially useful for creating hierarchies of metamodels [16]. The model extension relationship is illustrated in Figure 4.3. In the figure, M_i is a core model representing concepts for a kind of a system. Model M_f is the extension model, which defines some new concepts not existing in M_i but making references to the existing concepts of M_i [16]. The result of the extension is a new model M_r. The extension may re-define and extend the core model elements. All the models, M_i, M_f and M_r, conform to the same reference model.

4.1.2 Linguistic and ontological metamodelling

When facilitating an MDE-based framework for service-oriented software engineering, methodologies for developing and maintaining both system models and domain ontologies must be provided. This can be accomplished by considering explicitly the linguistic and ontological nature of metamodelling [12], providing a unifying metamodelling foundation that enables a natural correspondence between these two modelling spaces, such as discussed in [11, 194], and allowing the two kinds of models to interact with each other in a controllable manner through applicable knowledge management mechanisms.

In this context metamodelling refers to the activities that are required for formalizing the structure and semantics of modeling languages and ontologies. As discussed in Section 4.1.1, a hierarchy of terminal models, metamodels, and metametamodels can be used for this purpose. In the context of system specifications, the terminal models are the system models specified using an appropriate metamodelling language, and the system models conform to the corresponding metamodel. In the context of ontology engineering or conceptual modelling, the terminal models are known as *domain ontologies* and the metamodels they conform to are usually known as *upper-level ontologies* (see for example [11]) or domain ontology metamodels.

Linguistic metamodelling is used for defining modelling languages and their primitives on the metamodel level [12, 89] and so-called *linguistic instantiation* is used for instantiating model elements from the types defined within a corresponding metamodel. That is, linguistic instantiation crosses modelling levels and forms the basis for linguistic metalevels [12] (e.g. levels comprised of metametamodels, metamodels and models). In Figure 4.4 the linguistic instantiation takes place between the elements residing at L1 and L2 (linguistic) modelling levels separated verti-

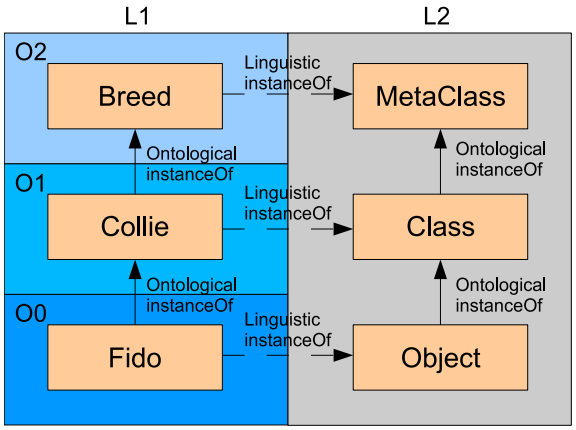

Figure 4.4: An example of ontological and linguistic instantiation relationships [12]

cally: *Breed* is a linguistic instance of *MetaClass*, and *Collie* is a linguistic instance of *Class*, for example.

Domain concepts are designed using ontological metamodelling where so-called *ontological instantiation* is used for creating domain specific artifacts using the concepts defined at the upper-level ontology, or an ontology metamodel [12, 89]. Ontological instantiation takes place within a linguistic modelling level [12] and between two models representing the concepts related by an ontological *"is-a"* relationship. In Figure 4.4 the linguistic modelling level L1 includes three ontological modelling levels O2, O1 and O0, and linguistic instantiation takes places between the concepts of *Breed*, *Collie* and *Fido* correspondingly. It should be noted that ontological instantiation is not a transitive relationships: while in the example illustrated in Figure 4.4 *Fido is-a Collie is-a Breed* does hold, it does not make sense to say that *Fido is-a Breed*.

Ontological instantiation provides support for facilitating dynamic user extensions to modelling concepts, modelling notation and the models created from them [12]. Metaconcepts such as *Breed* in Figure 4.4 allow for addition of new creature species, for example. This is especially invaluable for facilitating dynamic and open knowledge management for service-oriented collaborative computing environments.

Linguistic metamodelling is used for defining modelling languages for a specific domain of expertise. For this purpose each domain concept is given an intensional meaning via a modelling element (e.g. a class) attached with the classifying properties of the concept. Then, a model element is an instance of the concept if and only if it conforms with the classification criteria defined in the metamodel for that specific concept. That is, the linguistic instantiation depends primarily on the linguistic *conformsTo* relationship between a model and its reference model, as discussed in Section 4.1.1.

There is an essential difference between linguistic and ontological instantiation, in addition to the fact that the former is an inter-modelling level relationships while the latter is an intra-modelling level relationship. This difference is about the *intensional* and *extensional* meaning [134, 89] of concepts defined in the models. While in the example case illustrated in Figure 4.4 *Fido* is conformant with the characteristics of a *Collie*, as specified in the intentional part of the *Collie* concepts, it also belongs to the set of all *Collie*s. That is, in ontological metamodelling the instantiation relationship happens between two concepts if and only if one concept is conformant to the intentional part of the other and is *an element of* the extension of the other concept [89].

The foundational relationships for ontological metamodelling are summarized in Figure 4.5.

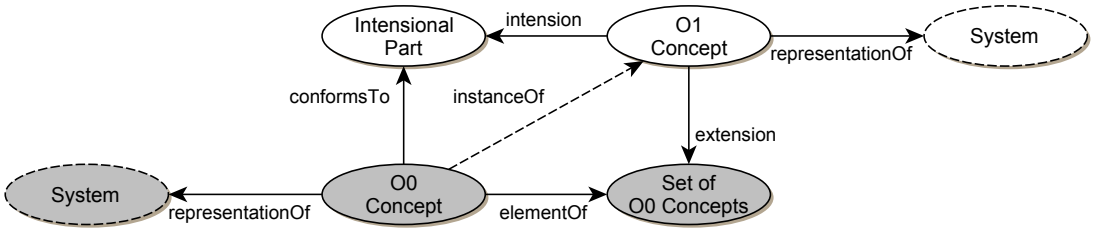

Figure 4.5: Ontological metamodelling relationships

While the ontological instantiation is probably the primary means for a user to derive ontological hierarchies, this relationship is actually a derived relationship induced by the *conformsTo* and *elementOf* relationships between a concept (i.e. "O0 Concept" in Figure 4.5, and the intensional and extensional meaning of its ontological type, the "O1 Concept" in Figure 4.5).

The extensional semantics of concepts is typically neglected in model-driven engineering approaches that concentrate on the linguistic metamodelling issues. However, in ontology description languages such as the OWL [259], concept classifiers are explicitly associated with their extensional meaning, that is, the set of their instances.

4.1.3 Metamodelling and knowledge management

The identification of and separation between linguistic and ontological metamodelling clarifies the role of models on one hand as enablers service-oriented software engineering and service-oriented computing (the linguistic dimension), and on the other hand as facilitators for knowledge management in open, evolvable environments (the ontological dimension). This separation also motivates the pragmatic need for explicit knowledge management facilities from a more theoretical perspective. The consequences of ontological and linguistic metamodelling relationships in the context of collaborative computing environments are especially interesting from the knowledge management perspective; these implications are discussed in the following.

From a knowledge management perspective a conceptual model, such as the toy-example illustrated in Figure 4.4, includes two fundamental relationships that have to be addressed. First of all, the *conformsTo* relationship is an inter-level relationship between two ontological layers and thus always involves a semantic interpretation. Thus a formulation for the semantic correspondence between a concept in lower layer and the intention of a concept in the higher layer has to exist. That is, *conformsTo* relationship is a semantic operator whose meaning has to be formally defined and validated before a correspondence between domain concepts in two separate ontological layers can be assured.

Secondly, the *elementOf* relationship between a concept and the corresponding set of concepts has to align with the conformance criterion. That is, any instance of a concept, an individual service description for example, can be considered as an element of the corresponding set of concepts if and only if it conforms with the corresponding intention defined at the upper ontological modelling level. The relationship between the activity of adding knowledge into the system (the set of concepts) and the conformance criterion is utterly important in a setting involving an open environment where users may declare new types of concepts by utilizing a multi-level ontological metamodelling framework. Such an environment necessitates the use of (model) repositories for maintaining the consistency of knowledge.

A *repository* is an infrastructure service that is responsible for maintaining linguistic and ontological consistency between domain concepts. A repository is basically a shared database of information which is used to store models of a certain (engineering) concept. It may contain information about document structures, interface definitions or ontologies [25, 190], for example. A repository offers semantically a higher level view on its information content that a plain database management system [25]. This semantically enriched view on domain information is provided by repository's *information model* which specifies the semantics and structure for repository functionality and information contents. Especially, the information model prescribes how the extensional part of a upper level concept should be managed with respect to the conformance criterion and what kind of additional relationships are to be maintained between the concepts.

Another important issue that needs to be covered when dealing with evolvable knowledge is the correspondence between a concept and its intention. In practical modelling scenarios a concept and its intention intertwine in a sense that typically only the intentional part, or the schemata for concepts, is explicitly modelled and the existence of the ontological concept is left implicit. The ontological concepts are however represented by the *names* each modelling artifact is given. Names are fundamental elements in distributed computing systems as they can be used for referencing different entities in different contexts [113] and also because of their ontological relevance. Thus names and management of naming becomes an important issue in an open environment with evolvable ontologies.

Concepts maintained in metainformation repositories are identified by names in collaborative computing systems; the naming conventions for concepts can be based on application and business domains or organisation domains, for example. When name handling is implemented in a manner that assures global consistency, names can be considered as globally unique identifiers and formulated as Uniform Resource Identifiers, or URIs [24], for example. In such a setting metainformation elements can be located, identified and referenced by their names. Namespaces can be used for structuring the universe of names and also for providing administrative domains over naming conventions. Name references are applicable for reusing names and giving aliases for same concepts in different contexts. Such name handling functionality can be part of metainformation repositories or implemented in separate name registries [109] providing name based resolution of metainformation.

4.2 Metamodels for formalizing the modelling framework

The modelling framework for service-oriented software engineering that is presented in this Thesis involves several areas of concerns whose inherent properties have to be formalized. In addition, the relationships between these concerns need to be defined unambiguously. To service these purposes, the modelling framework is defined itself as a metamodel that involves elements describing the methodological, ontological, and knowledge management perspectives on service-oriented software engineering and service-based collaborative computing. These elements are defined in separate metamodels and the relationships between the elements are provided by metamodelling relationships described in Section 4.1. The core metamodels comprise of a *1)* methodology metamodel, *2)* a knowledge management metamodel, and *3)* a domain ontology metamodel. These metamodels are then specialized and extended to form a reference model characterizing the properties of a domain targeted for service-oriented software engineering and collaborative computing. The individual metamodels and their inter-relationships are illustrated in Figure 4.6 and discussed in the following.

The *methodology metamodel* defines the elements needed for defining the actors, work prod-

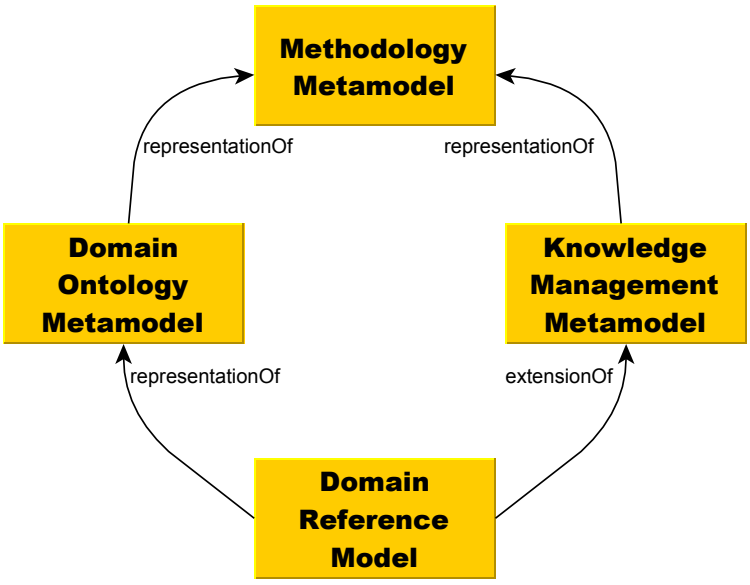

Figure 4.6: An overview of the foundational metamodels for realizing a modelling framework for service-oriented software engineering.

ucts and processes of service-oriented software engineering, for example. It is based on the OPEN (Object-oriented Process, Environment, and Notation) Process Framework (OPF), a metamodel-based method engineering framework [100]. While the discussion about the methodological metamodel and its implications on the properties of the SOSE framework is left quite small in this thesis, its has an important role in the resulting framework. First of all, the methodological metamodel provides a unifying vocabulary for describing about the engineering processes and activities needed for developing services and service-based systems. It also provides the means for describing what kind of knowledge elements, infrastructure services, and tools are needed for facilitating a service-oriented software engineering process. Methodological metamodels, such as the OPF, provide means for instrumenting method engineering, a discipline for developing and reusing best practices within the software engineering domain. Secondly, the methodological metamodel can be considered as some sort of "top-metamodel" for the whole modelling framework presented: both the knowledge management metamodel and the ontological metamodel are representations of the methodological metamodel. Especially, the work products, actors and infrastructure services required by a methodology model need to be represented by the elements of the previously mentioned metamodels.

The *knowledge management metamodel* provides modelling artifacts for defining a *megamodel* [31, 29] for service-oriented software engineering. A megamodel is a model that represents or at refers to models, and describes how models themselves are maintained and what are the global relationships (e.g. transformations) that exist between the models [31, 29]. The megamodel for SOSE presented in the following is based on the core metamodels developed for the Atlas Megamodel Management Architecture (AMMA) [29, 15, 245]. The AMMA metamodels are extended for enabling linguistic and ontological metamodelling, based on the work presented in [12, 89]. Even more importantly, the knowledge management metamodel provides extensions that explicitly define and formalize the relationships between the ontological and (linguistic) meta-

modelling *technical spaces*, following the terms used in recent discussions [194], thus providing a basis for knowledge management in service-oriented computing ecosystem.

The *domain ontology metamodel* defines the top-ontology for service-based cooperative communities. The metamodel is based on ontological metamodelling principles and defines explicitly the intensional and extensional meanings of the domain concepts. In addition the the meaning of individual concepts, the domain ontology metamodel also defines global relationships existing between the concepts. Such global relationships represent the fundamental semantic properties of the corresponding domains of expertise, or system under study.

Finally, the user-level models are based on a *domain reference model* which is based on the domain ontology and knowledge management metamodels. The domain reference model provides a representation of the domain ontology by describing the types of models that represent the domain concepts. In the following subsections we describe the previously mentioned metamodels in more detail.

4.2.1 Knowledge management metamodel

The knowledge management metamodel comprises of three individual metamodels: *1)* a systemic metamodel formalizing the foundations for linguistic and ontological metamodelling, *2)* a global model management metamodel describing the modelling artifacts and their inter-relationships, and *3)* a knowledge repository metamodel making explicit the relationships between ontological concepts, modelling artifacts and repository services.

The systemic metamodel formalizes the system-based metamodelling foundations introduced in [77, 78, 89]. As illustrated in Figure 4.7, a *System* is associated to other systems with the metamodelling relationships discussed in Section 4.1.1: a *System* may conform to (*conformsTo*) another *System*, be a representation of (*representationOf*) some other system, or be an element (*elementOf*) of a set of *System*s. Three kinds of systems are identified and differentiated by the metamodel, following the (incomplete) classification presented in [77]: *PhysicalSystem*s, *DigitalSystem*s, and *AbstractSystem*s. An *AbstractSystem* is the only one associated explicitly with intensional and extensional descriptions, where the extensional semantics of a system is provided by the concept of *Set*, following [89]. Subsequently, every ontology in this modelling framework is considered as an abstract system.

The global model management metamodel describes the modelling artifacts and their inter-relationships that can be used for managing the knowledge within some engineering domain. The metamodel is based on the AM3Core metamodel developed in the AMMA [245] project and described partially in [29, 15] [1]. The AM3Core metamodel describes the concepts needed for describing new kinds of megamodels. As illustrated in Figure 4.8, a *MegaModel* contains a set of *Elements*. Each element is a container of some *Metadata* that describes for example the contents of a model. There are two basic kinds of elements that are distinguished: identified and located elements. An *IdentifiedElement* is a modelling artifact that can be referenced even if it is not locally available [15]. A *LocatedElement* is provided with a physical identifier, a *Locator*, that can be dereferenced to an actual modelling element. A *Group* is a logical set of elements of a megamodel [15]. Modelling artifacts are represented by the *Entity* element, which is both an identified and located element. Finally, modelling relationships such as model transformations, are based on the notions of *Relationship* and *DirectedRelationship*. A composition of directed relationships is called a *Chain*.

[1] Actually the AM3Core metamodel used in this thesis is the latest version found from the AMMA AM3-project version control system [244]

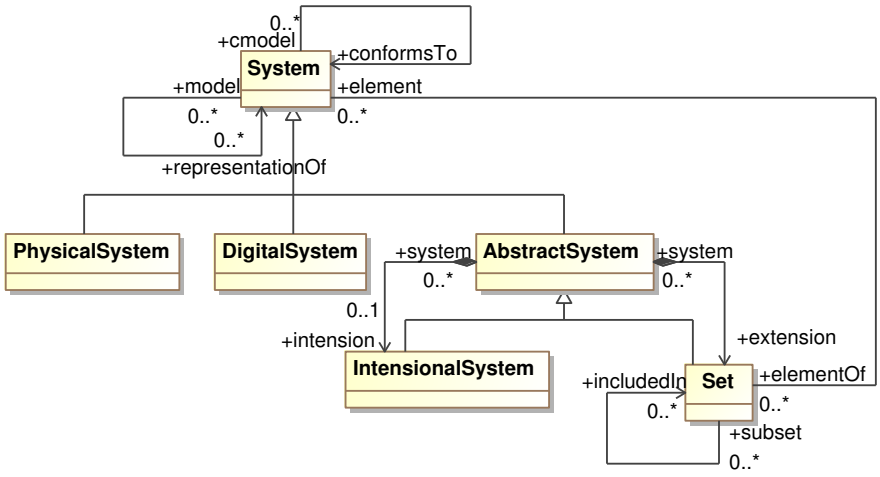

Figure 4.7: The systemic metamodel

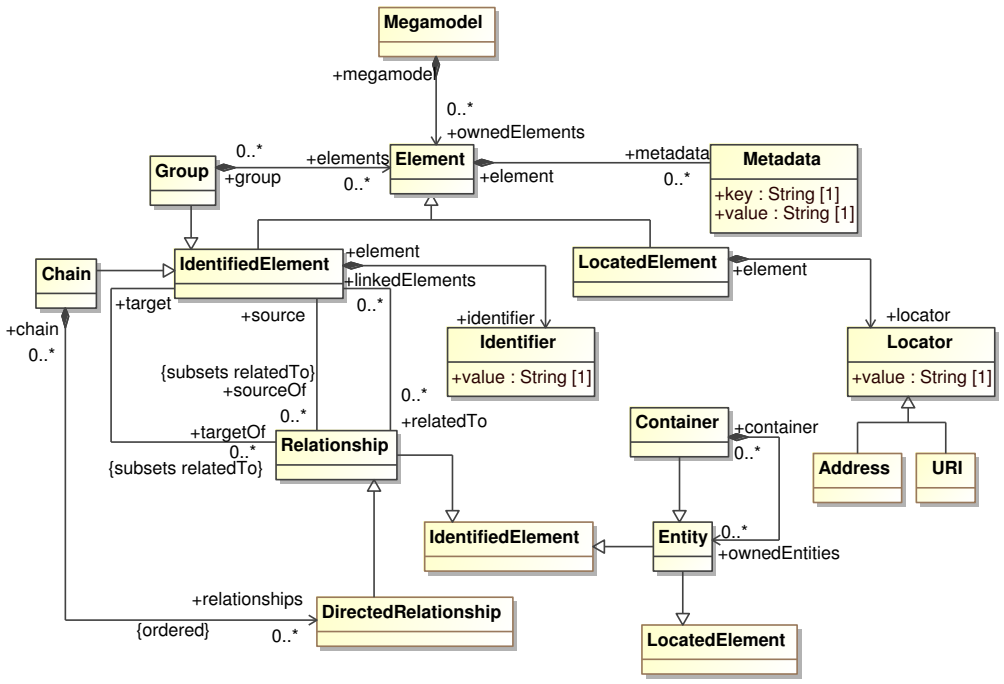

Figure 4.8: The AM3Core metamodel

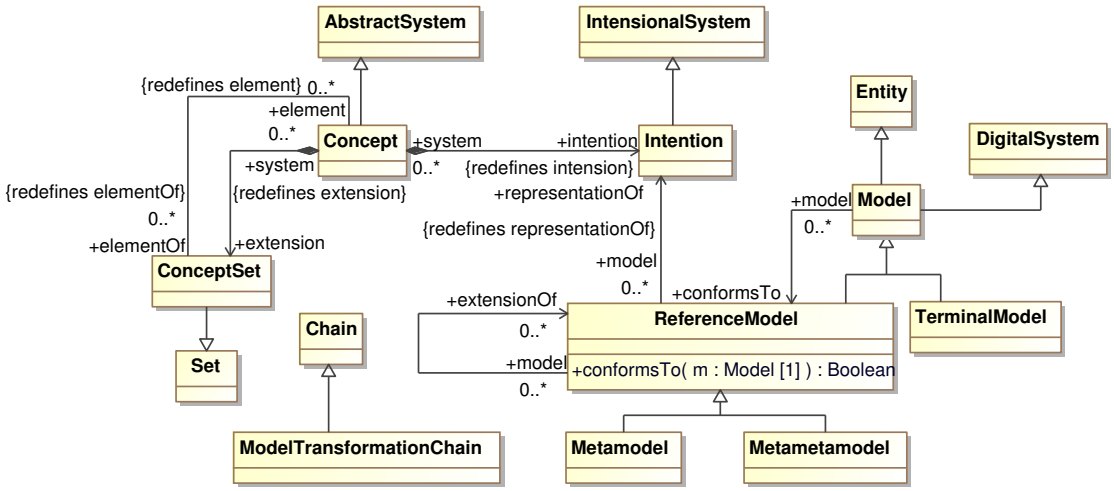

Figure 4.9: The global model management metamodel

The concepts defined by the AM3Core metamodel are used by a global model management metamodel describing the essential modelling artifacts, that is models and modelling relationships such as transformations. The global model management metamodel described in the following is an extension of the Global Model Management (GMM) metamodel developed in the ATLAS MegaModel Management (AM3) project [41, 243]. The AM3 GMM metamodel describes a model as a specialization of the AM3Core *Entity* element, as well as defines several kinds of model transformations as specializations of the *Relationship* and *DirectedRelationship* concepts illustrated in Figure 4.8. The model transformation specific elements are dismissed from the following discussion, however. The global model management metamodel developed for this Thesis has some significant changes made to the AM3 version of the GMM metamodel. These changes concern the concepts of models and reference models. The resulting metamodel is illustrated in Figure 4.9 and discussed below.

The global model management metamodel defines a *Model* as a specialization of the *Entity* concept, similarly to the AM3 GMM metamodel. In distinction to the AM3 GMM metamodel, a *Model* element is provided with additional properties. First of all, a model is considered as a kind of a *DigitalSystem*. This means that the models that are part of the model management domain are provided by a digital representation, such as XMI [184] for example.

For enabling model extension [16] mechanisms to be used for developing and reusing modelling artifacts, a *ReferenceModel* is provided with *extensionOf* association to another reference model. In addition, a reference model is associated with an explicit *conformsTo* operation, which implements the reference model specific conformance validation procedures. This function encodes the semantic properties associated to the reference model and expressed in an applicable logical framework, similarly to [247].

Most importantly, a *ReferenceModel* is considered as a representation of an intension for some ontological concept. This relationship between reference models and concepts described in a domain ontology provides the foundation managing business and engineering knowledge. Domain ontologies are in this framework considered as collections of concepts and their inter-relationships. An ontological concept is represented by the *Concept* element, which is considered as an abstract system. The intensional meaning of a concept is provided by the *Intension* element and extensional

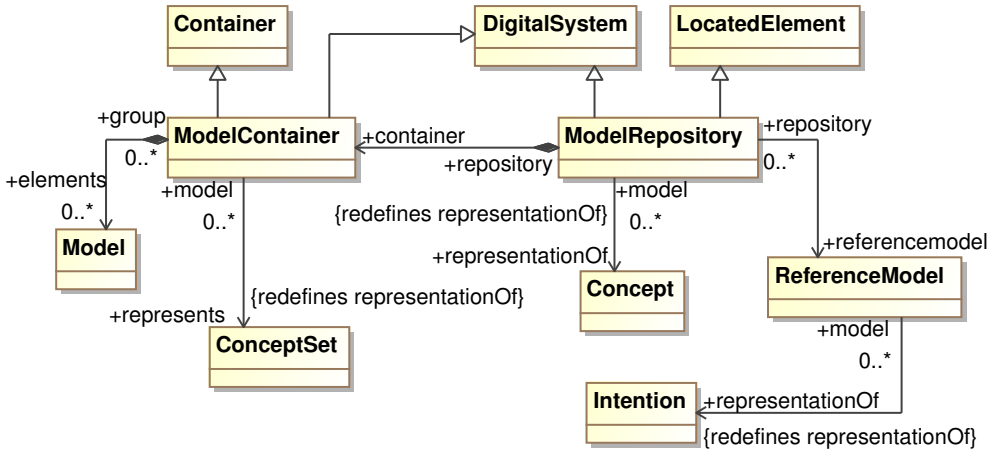

Figure 4.10: The model repository metamodel

meaning by a set of concepts (*ConceptSet* element).

Finally, the model repository metamodel illustrated in Figure 4.10 formalizes the connections between repositories, models and reference models, and ontological concepts. A *ModelRepository* is considered as a locatable modelling artifact and a *DigitalSystem* that provides a model containment service. Especially, a model repository can be considered as a technological representation of an ontological concept. The model container provided by a repository comprises a set of *Model* artifacts; the *ModelContainer* thus provides a representation of a set of *Concept*s. The models that are stored within a repository conform to the reference model associated with the corresponding kind of repository by the *modeltype* association.

4.2.2 Methodology metamodel

The methodology metamodel defines modelling element that are needed for describing the actors, work products, and processes applicable in service-oriented software engineering processes. The methodology metamodel is based on the OPEN Process Framework [100] and its illustration is given in Figure 4.11.

As illustrated in Figure 4.11, a *Methodology* is comprised of *Producers*, *WorkUnits*, *WorkProducts*, and *Artefacts*. A *Producers* represents the actors of a methodology. Producers perform *WorkUnits* for creating, iterating and maintaining *WorkProduct*s [100], possibly using some specific engineering *Tool* targeted for this purpose. A *WorkUnit* is "a functionally cohesive operation performed by a Producer" [100] where a work-unit can be further classified to *WorkActivity*, *WorkTask*, and *Technique*. Finally, a work-product is "anything of value produced during the development process" [100]. Finally, an *Artefact* represents some element of engineering that is invaluable for facilitating the methodology itself. The essential Artefacts identified in the methodology metamodel include *Tool*s, *Documentation*, and *Product*s.

Based on the basic engineering methodology concepts defined by the methodology metamodel, a metamodel for defining product life-cycles is created. As illustrated in Figure 4.12, a *Product* is defined as both an Artefact of a methodology and a work-unit, that is, an end-result usable and having value outside the methodological framework. In the context of service-oriented computing, services and service-based systems can be considered as primary products addressed

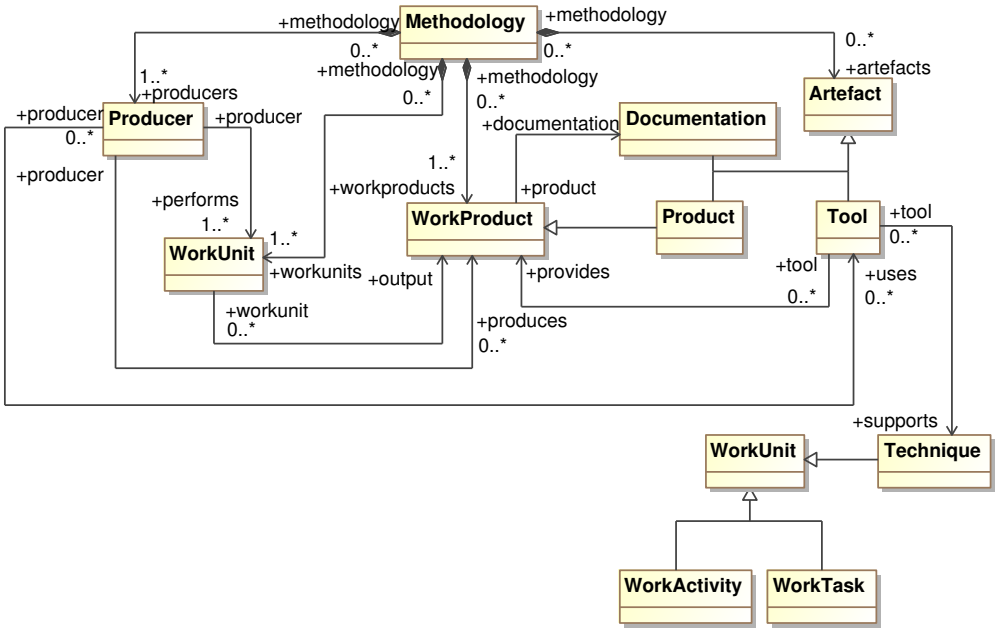

Figure 4.11: The methodology metamodel

by the corresponding engineering efforts and methodologies. A *Product* is associated with a *ProductLifeCycle* that is defined as a ordered set of *Phase*s. When considering a service-product, the corresponding life-cycle defined could comprise of design, modelling, development, deployment and maintenance phases, for example. Each *Phase* is further described by a corresponding *Process* that describes the activities to be taken for fulfilling the requirements of the corresponding life-cycle phase. The process activities are defined as *Stage*s which correspond to the work-activities, -tasks and techniques defined by the methodology metamodel described previously.

4.2.3 Domain ontology metamodel

The domain ontology metamodels defined in this Thesis are built on a top-ontology for cooperative communities. This top-ontology is defined by conceptualizing the essential elements of cooperative communities, such as cooperating entities, features of such entities, and the notion of role-based cooperation. In the rest of this Thesis, we are actually going to present a specialization hierarchy of three domain ontology metamodels; this hierarchy is illustrated in Figure 4.13 with examples of the new concepts defined by the corresponding metamodel given within the white ellipses.

All the domain ontology metamodels have some common structure: they describe the kinds of cooperation forms existing, the entities taking part in such cooperations, and the kinds of features associated with the corresponding kinds of entities. The most abstract top-ontology describes the domain of cooperative communities. Based on the cooperative community domain ontology, a top-ontology describing the domain of service-based cooperative communities prescribes specializations of the cooperation concepts and by introducing additional concepts needed for addressing the service-oriented viewpoints on the subject. Finally, a domain ontology metamodel for so-called federated service communities is provided which further specializes some of the concepts defined in the previous top-ontologies.

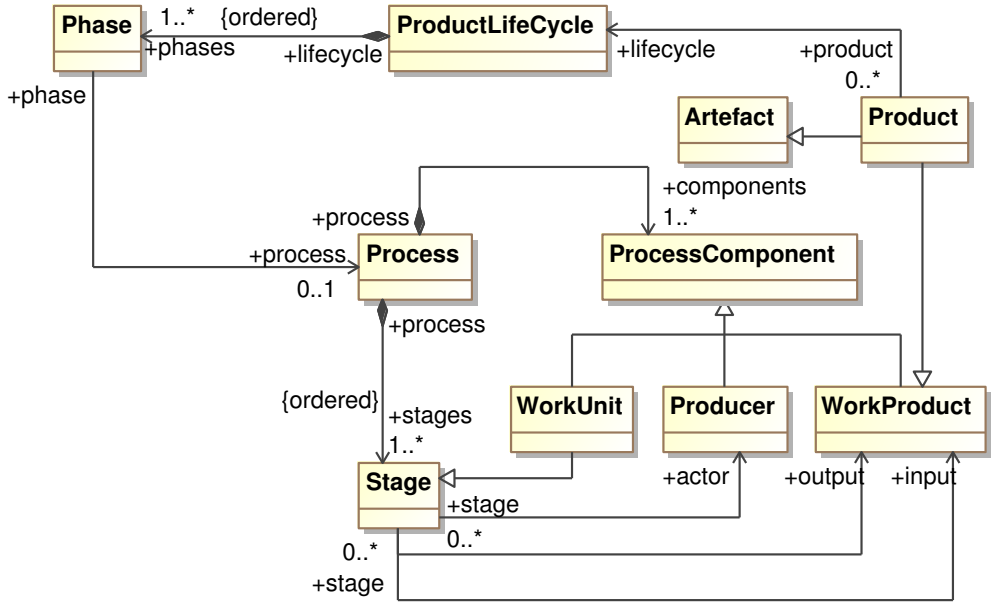

Figure 4.12: The product lifecycle metamodel

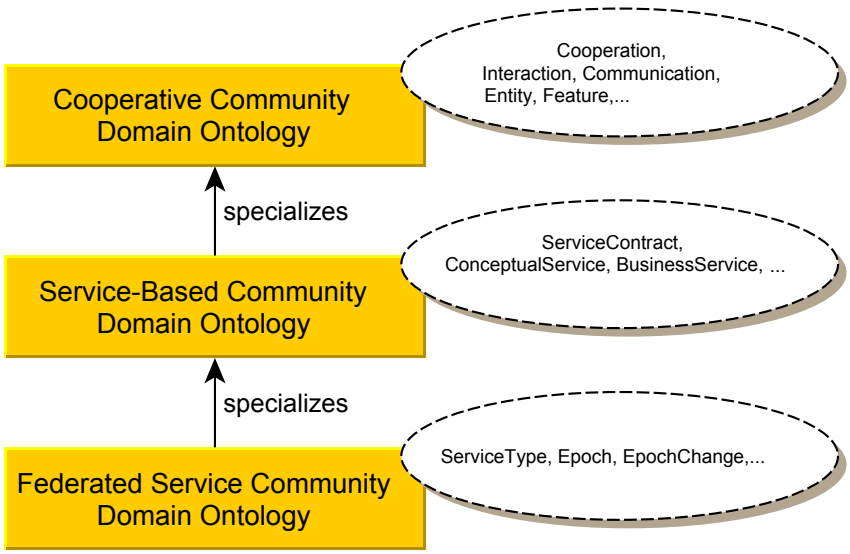

Figure 4.13: The hierarchy of domain ontology metamodels.

The metamodels defined in the following Chapters define the domain ontologies of collaborative systems and federated service communities. Each metamodel gives a description of the fundamental concepts in the corresponding domain of collaboration. A verbal description is given for all the concepts and more importantly, the concepts and their inter-relationships are described using UML class diagrams [179] and corresponding modelling notations.

4.2.4 Domain reference models

The purpose of a domain reference model is to specify the types of models needed for facilitating a domain of collaborative computing and corresponding software engineering processes. The model types are defined as specializations of the modelling Artefacts defined in a knowledge management metamodel. That is, a domain reference model describes the links between a domain ontology and a knowledge management metamodel, thus implicitly prescribing the necessary knowledge management facilities, or repositories, needed to facilitate a modelling framework.

Domain reference model is a representation of a domain ontology metamodel, that is, each concept of the domain ontology is represented by a modelling Artefact in the domain reference model. Secondly, the domain reference model is an extension of an appropriate knowledge management metamodel. Especially, the domain reference model introduces new model kinds that describe the modelling Artefacts needed represent the domain ontology concepts. In addition, the domain reference model can introduce additional modelling Artefacts that are needed to facilitate a corresponding kind of a collaborative computing and software engineering framework.

In this Thesis a domain reference model for federated service communities is described in Chapter 6. The domain reference model is based on the domain ontology metamodel for federated service communities and extends the knowledge management metamodel defined for service-based communities.

Chapter 5

Domain ontologies for service-based collaborations

For establishing loosely coupled, service-based collaborations the concepts needed and applied in such context need to be formalized. In this chapter domain ontologies describing such concepts are defined. For each of the domain ontologies, cooperative community domain ontology and service-based community domain ontology, first an overview of the top-level ontology is given which describes a selection of the core concepts in the ontology and their global ontological relationships. After that, the individual concepts of an ontology are elaborated by describing their intentional parts.

For establishing interoperable cooperative communities, the behavioural artifacts taking part in cooperations are provided with descriptions of their behavioural features. More over, global behaviour of cooperative communities themselves need to be prescribed for attaining a common comprehension of their intentions. Behaviour of a business service may comprise a set of simple document-exchange patterns, such as the message-exchange patterns provided by the forthcoming WSDL 2.0 [266] standard, or more complex conversations sequencing multiple activities, for example. When given formal semantics, the behavioural descriptions can be utilized for analyzing the properties of services and for validating service interoperability within cooperations. Different formal methods such as finite state machines [21], Petri nets [97], and process algebras [220] have been used for such purposes. Two different kinds of behavioural artifacts are used in the domain ontology of cooperative communities, namely *behavioural pattern*s and *event structures*.

The contents of this chapter are as follows. Section 5.1 first introduces some conventions used for describing the domain ontologies. A domain ontology for cooperative communities is defined in Section 5.2. The domain ontology introduces concepts for defining cooperation kinds, and the entities features and behavioural artifacts related to such cooperative collaboration forms, for example. The concepts of cooperative communities provide the foundations for defining the domain ontology for service-based communities which is then described in Section 5.3. In addition to specializing the cooperative community concepts, the domain ontology introduces new ontologies for expressing elements fundamental for establishing service-oriented computing environments. Such concepts include conceptual and business services, service contracts, and service connectors, for example.

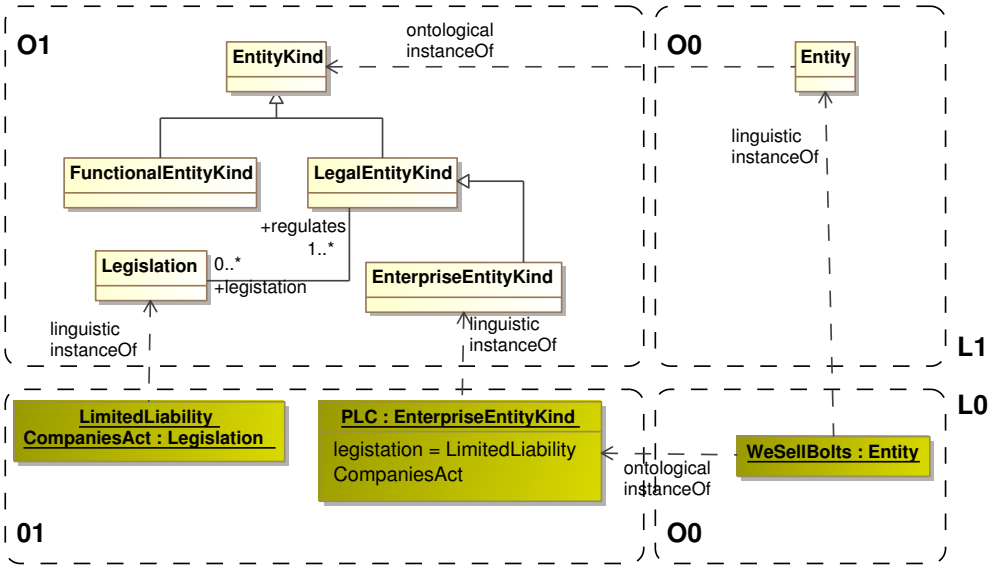

Figure 5.1: Illustrating the use of two-level ontological metamodelling concepts.

5.1 Describing the domain ontology metamodels

The intentional parts of domain ontology concepts are defined in the following as metamodels that are described using UML class-diagrams [182]. UML-specific notation is also used for defining generalization hierarchies, associations, and composition relationships between the elements related to an ontological concept. While UML was originally developed for object-oriented modelling purposes, UML [182] is also suitable for modelling different kinds of metamodels; not the least because the foundation based on the Meta Object Facility [181].

The metamodels describing the concept intentions in the following sections distinguish between the ontological types and instances of concepts. A two-level ontological metamodelling approach is used for describing the ontological metamodels. At the *O1* level (see Section 4.1.2) ontological types are defined that represent the kinds of ontological concepts that can be created and their generic properties. Instances of ontological concepts are declared at the *O0* level in conformance with the corresponding ontological types. As discussed already in Section 4.1.2, such existence of multiple ontological metamodelling levels enables extendable typing structures that are invaluable in the context of loosely coupled collaborative computing environments.

The usage of two-level ontological metamodelling framework is illustrated in Figure 5.1. The two linguistic metamodelling levels, *L1* at the top of the figure and *L0* at the bottom part, consist of metamodels and terminal models, correspondingly. The two ontological metamodelling levels comprise of ontological types at the *O1* level (left side of the figure) and their ontological instances at the *O0* level (right side of the figure). In this example a concept of *EntityKind* is classified to sub-concepts describing functional and legal entity kinds. A legal entity kind is represented by the concept of *LegalEntityKind* which is associated with a set of legislative acts that somehow address the actions of corresponding kind of legal entities. Further, the concept of *LegalEntityKind* has a subconcept of *EnterpriseEntityKind*, representing the kinds of enterprises that can take part in cooperative contexts. The ontological concepts used in this example are elaborated in the corresponding sections that follow.

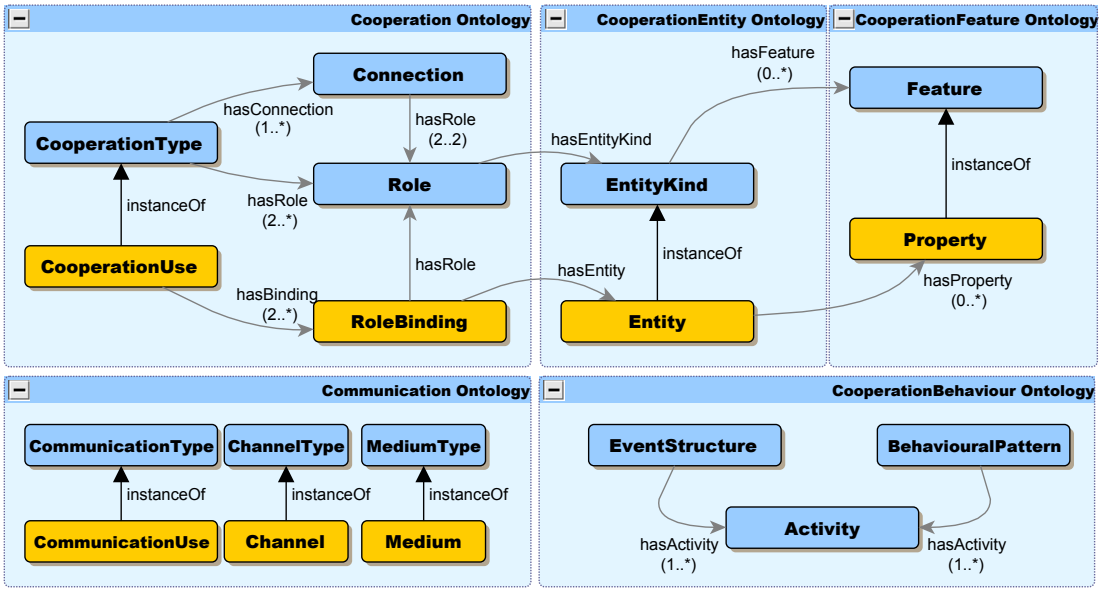

Figure 5.2: Top-level concepts for the cooperative community ontology.

Consequently, new kinds of enterprises can be added by users to the knowledge management system. In the example case illustrated in Figure 5.1, a public limited company (*PLC*) has been provided as one specific kind of an enterprise. The *PLC* concept is a linguistic instance of the *EnterpriseEntityKind* concept and it is associated with *LimitedLiabilityCompaniesAct* concept, which is a linguistic instance of the *Legislation* concept. Now enterprise willing to act in cooperative communities can publish their public information. This is done by publishing a model that is a linguistic instance of the *Entity* concept. In the case of an public limited company, the *Entity* instance must also be an ontological instance of the *PLC* concept defined at the ontological type level of the linguistic O0 level. In the example case, a company named *"WeSellBolts"* announces itself as a public limited company.

5.2 Ontology for cooperative communities

In the context of this Thesis, cooperation is considered as *an arrangement between entities taking roles corresponding to some specified context of joint operation*. Cooperation can take place between legal entities (e.g. a service cooperation (agreement) relationship between an individual and an enterprise), between interaction endpoints, behavioural entities (e.g. in case of human actors), or between information entities (e.g. federated databases), for example.

The ontology for cooperative communities is defined as a collection of inter-related sub-ontologies, or groupings of concepts. First of all, an ontology describing the meaning of cooperation itself is needed. Secondly, an ontology for describing the entities that can take part in a cooperation needs to be in place. Finally, an ontology for describing the features the entities can have and that may affect on the properties of cooperation must be provided. Such an arrangement of ontologies for declaring the ontology for cooperative communities is illustrated in Figure 5.2.

The *Cooperation Ontology* illustrated in Figure 5.2 describes the concept of cooperation by using two ontological levels. At the type level (O1), *CooperationType* concept is used to describe

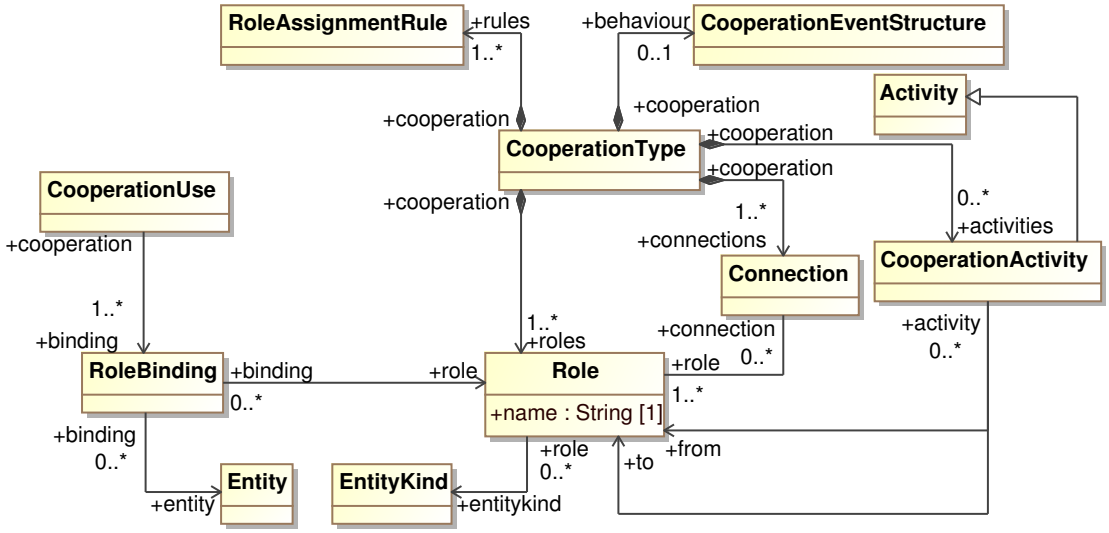

Figure 5.3: Cooperation ontology metamodel.

the roles (*Role*) and connections (*Connection*) that comprise the structure of a cooperation. At the ontological instance level (O0) the concept of *CooperationUse* is defined as a set of bindings between the roles defined at the type level, and entities chosen for the collaboration. These bindings are represented by the *RoleBinding* concept.

The *Communication Ontology* describes concepts needed for facilitating communicative cooperations. The ontology introduces the notion of communication as a specializations interactions concept defined in the *Cooperation* ontology. In addition, concepts representing communication facilities such as communication channels and media are defined.

Cooperative entities are described in conformance with the *CooperationEntity Ontology*. At the type level, the ontology defines a concept of *EntityKind* that declares the generic properties for a kind of entities (e.g. enterprises). These type level properties are called as *Features* for which a corresponding ontology, the *CooperationFeature Ontology* is provided. At the instance level, actual entities, such as a specific enterprise, are prescribed. The features defined at the type level correspond to properties (*Property*) at the instance level.

In the following subsections the ontologies describing cooperations, cooperation entities and cooperation features are discussed in more detail and the intentions of the corresponding concepts are formalized as UML class diagrams [182].

5.2.1 Cooperation ontology

The cooperation ontology metamodel describes the elements for defining different kinds of cooperation relationships. The metamodel is illustrated in Figure 5.3 and it essentially prescribes a set of roles, their inter-relationships and bindings of the roles to a set of participants. The concept of *CooperationType* defines a set of *Role* and a set of *Connection* elements. Each *CooperationType* must have at least two role definitions having at least one connection defined between them.

Roles are used to declare expected behaviour, rights, obligations, as well as position and relationships with respect to other roles for entities willing to act in a cooperative context (see [32] for the different aspects of the notion of role in social contexts). In practise, each *Role* declaration

should also declare a unique name for the role within the corresponding cooperation such that entities can be referred indirectly through the role names. A role definition includes a set of inter-role connections. The *Connection* concept relates exactly two *Role* elements with a symmetric relationship. The roles in a *Connection* must be distinct, that is degenerate connections from a role to itself are not permitted. More over, a role can be attached to a certain kind of an entity, as presented by the *entitykind* association.

Roles and their interconnections can be attached with domain specific constraints using the *RoleAssignmentRule* elements. The constraints so prescribed are evaluated during role binding and in the context of *CooperationUse* elements. As an example, an assignment rule for a cooperation involving the roles of a *Notary*, *Provider* and *Consumer* might state that the entity taking the role of *Notary* can not take part in the other roles.

The concept of *CooperationUse* is an ontological instance of the *CooperationType* concept and describes a particular application of the corresponding type of cooperation. *Cooperationuse* prescribes a set of role bindings that attach the cooperation participants to the roles defined in the corresponding kind of cooperation specification. Cooperation roles are attached to participants using the *RoleBinding* concept. The role attached to a participant constraints and regulates the behaviour and properties of the entity to suit the requirements of the corresponding form of cooperation. The structure of the cooperation metamodel is inspired by the PIM4SOA [20] and UML metamodels [179] with respect to the relationships between *CooperationType* and *CooperationUse* elements, roles and role bindings. For example the UML metamodel [179] defines the concepts of *Collaboration* and *CollaborationUse*, and role bindings in a similar fashion.

Behaviour assigned for a role by a cooperation declaration is considered as a prescription of expected behaviour and it plays an important role when considering the consistency of cooperations or business service interoperability, for example. The behavioral properties for a cooperation are defined using the concept of *CooperationEventStructure* which specifies behaviour as a partially ordered set of events labelled with the *CooperationActivity* concepts. The basic unit of behaviour is described by the concept of *Activity*, which represents *"any activity that is considered as a conceptual entity at the given level of abstraction"* [257]. An *Activity* is considered a physical or mental act of performing something that changes the state of the cooperative environment. In the case of communication behaviour, the set of activities would include send and receive activities mediating the different kinds of information entities involved between cooperation participants, for example.

For formalizing cooperation behaviour so-called *labelled prime event structures* [275] are used. Event structures provide an elegant and generic formalization of behaviour that is particularly well suited for expressing the externally observable behaviour of service-oriented business processes. Further, event structure semantics allow for abstraction refinement and aspectual refinement of business processes (see Section 3.3.3). Such refinements can not be easily incorporated to behavioural models following interleaving semantics of concurrency, such as process algebraic approaches.

While in the context of this Thesis we do not provide a concrete syntax for describing cooperative behaviour, we assume that a behavioural description attached with a cooperation model can be transformed to a format that coincides with the formal structure of labelled prime event structures. Prime event structures model distributed computation as event occurrences together with relations expressing their causal dependency and non-determinism [275]. An event in this context is considered as an instantaneous and indivisible action which happens once in a computation. Individual events will be labelled with activities for expressing recurring occurrences of similar activities.

Before giving the definition for labelled prime event structures, some notational conventions

have to be introduced. For two sets A and B, their union is denoted as $A \cup B$, intersection as $A \cap B$ and their difference as $A - B$. An empty set is denoted with $\emptyset$. Let $(X, \leq)$ be a partially ordered set (i.e. $\leq$ is a reflexive, transitive and antisymmetric relation over the set X) and $Y \subseteq X$. Then the left-closure of Y is defined as $\downarrow Y = \{x | \exists y \in Y, x \leq y\}$. When $Y = \{y\}$ we just write $\downarrow y$ instead of $\downarrow \{y\}$.

Event structures provide a *true concurrent model* or a *causal model* of concurrency, among models like Petri-nets or trace languages[275]. The definition for labelled prime event structures [275, pp.42] is given in Definition 5.2.1. Labelled prime event structure is a variant of the more generic notion of event structures [275].

Definition 5.2.1 (Prime event structures [275]) *A **(prime) event structure** is a tuple $ES = (E, \leq, \#)$ consisting of a partially ordered set of **events** $(E, \leq)$ where $\leq$ is called the **causal dependency relation**, a binary irreflexive relation $\#$ on E which is symmetric and disjoint on $\leq$ known as the **conflict relation**. We let $e, e', \ldots$ to range over events.*

*Causal dependency and conflict relations must satisfy the principles of **finite causes** ($\downarrow e$ is finite for every $e \in E$) and **conflict inheritance** ($e \# e' \leq e'' \Rightarrow e \# e''$).*

*Two events are **concurrent** ($e \, co \, e'$) if they are not in conflict or causally dependent: $e \, co \, e' \Leftrightarrow \neg((e \# e') or (e \leq e') or (e' \leq e))$.*

*A **labelled prime event structure** is a tuple (ES, λ) where ES is an event structure, and $\lambda : E \rightarrow L$ is a function called the **event labelling function** where L is a finite set of **labels**.*

From now on, (labelled) prime event structures can be referred to simply as (labelled) event structures.

A subset $X \subseteq E$ of events is *conflict free*, if for all $e, e' \in X : \neg(e \# e')$. A subset $X \subseteq E$ is a *configuration* of event structure $(E, \leq, \#)$ if $X = \downarrow X$ and X is conflict free. A configuration represents a set of events that have occurred at some point of time.

Let C_{ES} denote the set of finite configurations of an event structure ES and $X, Y, \ldots$ range over configurations. The empty set of events $\emptyset$ is a configuration, as well as is the left-closure $e \downarrow = \{e' | \exists \in E e' \leq e\}$ for all $e \in E$ [275]. We define $\#(c) = \{e' | \exists e \in c : e \# e'\}$ and call $ES \backslash c = (E', \leq', \#')$ the *substructure rooted at* c, where $E' = E - (c \cup \#()c)$, $\leq'$ is $\leq$ restricted to $E' \times E'$, and $\#'$ is $\#$ restricted to $E' \times E'$ [246]. The corresponding structure ES' is also a valid event structure. We say that two event structures ES_1 and ES_2 are *isomorphic*, denoted as $ES_1 \equiv ES_2$, if there exists a bijection $f : E_1 \rightarrow E_2$, such that $\forall e, e' \in E_1$ it holds that $e \leq_1 e'$ iff $f(e) \leq_2 f(e')$, and $e \#_1 e'$ iff $f(e) \#_2 f(e')$. In addition, for labelled event structures the labelling must be preserved by bijection f: $\lambda_2(f(e)) = \lambda_1(e)$ for every $e \in E_1$ [246].

For $e \in E$ and $X \in C_{ES}$ we say that the event e is *enabled at* X if and only if $e \notin X$ and $X \cup \{e\} \in C_{ES}$. The set of events enabled at the configuration X is denoted as $en(X)$. We say that event structure ES is *finitely enabling* if $en(X)$ is finite for every $X \in C_{ES}$. In the following, we consider only event structures that are finitely enabling. We write $X \xrightarrow{e} X'$ for representing an event transition between configurations X and X' when $e \in en(X)$ and $X' = X \cup \{e\}$. A labelled event structure is *without auto-concurrency* if and only if concurrent events in the structure do not carry the same label [257]: $\forall X \in C_{ES}, \forall d, e \in X.(d \, co \, e$ and $\ell(d) = \ell(e))$ implies $d = e$.

Event structures provide an elegant formalism for expressing externally observable behaviour in the context of cooperative communities. Especially, event structures allow behavioural refinement [257] needed for effective and agile development of business processes. This is a considerable distinction when compared to interleaving models of concurrent systems, such as process algebraic approaches, for example.

For theoretical and practical reasons we restrict the expressibility of cooperative behaviour. Whereas the event structures defined above enable expression of potentially infinitely variable

behaviour, a class of so-called *regular event structures* [246] represents potentially infinite concurrent behaviour which involves repetition of past activities.

For an event structure ES an equivalence relation can be defined over its set of configurations as $R_{ES} \subseteq C_{ES} \times C_{ES}$ such that $cR_{ES}c'$ iff $ES\backslash c \equiv ES\backslash c'$ [246]. Using this equivalence, the formalization of regular event structures is given in Definition 5.2.2.

Definition 5.2.2 (Regular event structures [246]) *Event structure ES is **regular** iff R_{ES} is of finite index (the set of equivalence classes of R_{ES} is finite) and there exists an integer k such that $|en(c)| \leq k$ for every $c \in C_{ES}$.*

That is, a regular event structure must be finitely enabling and comprise of finitely many equivalence classes of configurations.

As illustrated previously in Figure 5.3 *Cooperation* is attached with of *CooperationEventStructure* that describes the global behaviour of the cooperation. A cooperation event structure is a specialization of the event structure concept which redefines the event labeling to consider activities more suitable for expressing behaviour of role-based behaviour. Each cooperation activity is provided with associations which declare the initiator and receiver roles of the activity. A cooperation activity is thus represented as a pair of roles declaring the direction of the corresponding activity.

The formalization for cooperation event structures is given in Definition 5.2.3. It should be noted that cooperation event structures (and its derivatives) are assumed to be without auto-concurrency. This is a natural assumption when considering cooperative role-based behaviour: parallel activities between a pair of roles should always be distinguishable by their properties.

Definition 5.2.3 (Cooperation event structures) *A **cooperation event structure** is a tuple (ES, λ) where ES is a regular event structure and $\lambda : E \rightarrow (R \times L \times R)$ is a mapping called the **cooperation event labelling function** where R is a set of roles in the corresponding cooperation model and L is a finite set of labels. When $(r_1, a, r_2) \in rng(\lambda)$ then r_1 is called as the **initiator** and r_2 as the **terminator** of the activity. We assume that cooperation event structures are without auto-concurrency.*

Interaction

An interaction realizes information exchange between two or more entities with dedicated roles and is carried over an interaction medium. The semantics of interaction is prescribed on the one hand by the behaviour and characteristics implied by the roles, and on the other hand by the properties declared for the interaction endpoints taking part in it.

Interaction is a kind of cooperation that comprises interaction roles and behaviour to be used in the interaction, and takes place between interaction endpoints. The intention for the concept of interaction is defined by the metamodel illustrated in Figure 5.4. *InteractionType* prescribes the roles involved in the interaction and their inter-connections. An *InteractionRole* is a specialization of the *Role* concept that is only attachable to entities that are of kind *InteractionEndpointEntity*.

An *InteractionEndpointKind* is a subconcept of the *EndpointEntityKind* and it represents the locus of interactions, namely the local artifacts for facilitating interactions. The information provided by an interaction endpoint enables directly or indirectly (via resolution mechanisms or mediators) the interaction activities conforming to the behaviour defined in the corresponding type of interaction.

It should be noted that the *InteractionRoleBinding* concept differs in a significant manner from the corresponding concept used by the *CooperationUse* concept: an interaction role binding must also identify the connection that a interaction endpoint is bound to. This emphasizes the role of connectivity in interaction, as opposed to possibly other kinds of cooperation.

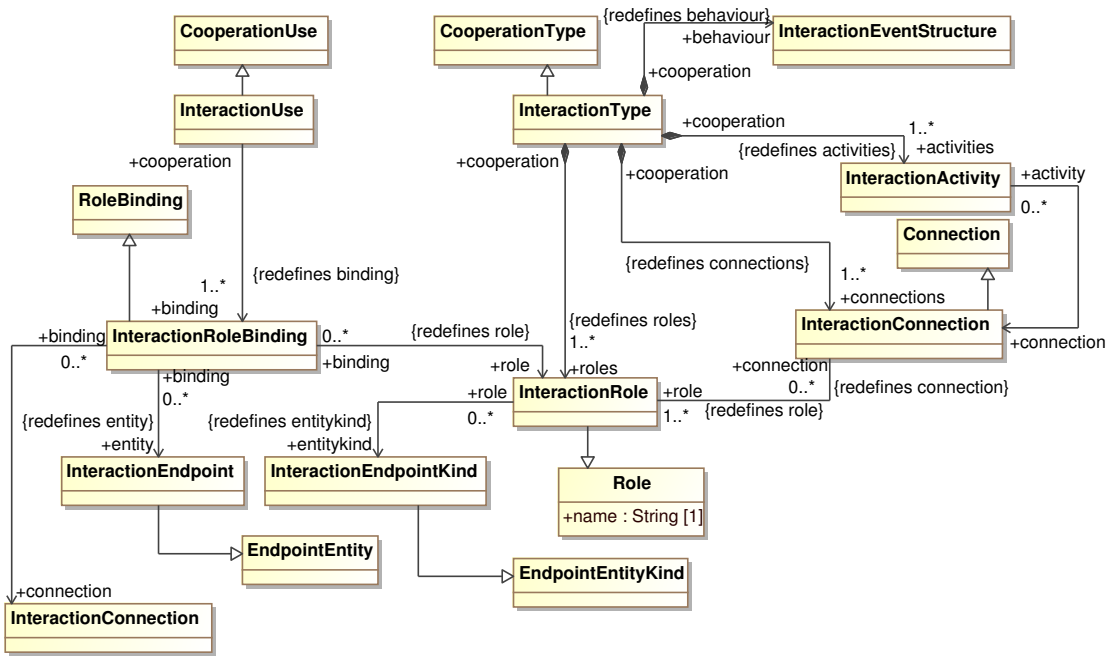

Figure 5.4: Ontology for interaction.

An *InteractionType* is associated with the kind of behaviour that is represented by the concept of *InteractionEventStructure*. Interaction behaviour is formalized by *interaction event structures* that specialize the notion of cooperation event structures to consider activities which explicitly declare the connection to be used by the activity. Formalization for interaction event structures is given in Definition 5.2.4.

Definition 5.2.4 (Interaction event structures) *An **interaction event structure** is a tuple (ES, λ) such that ES is a regular event structure and $\lambda : E \rightarrow (R \times L \times C \times R)$ is a mapping called the **interaction event labelling function**, where R is a finite set of roles, L a finite set of labels, and C is a set of connections relating the roles of the corresponding cooperation model. When $(r_1, a, c, r_2) \in rng(\lambda)$ then r_1 is called as the **actor** and r_2 as the **reactor** of the activity, and c as the **interaction medium**.*

Communication

In the context of service-oriented computing, communication can be regarded as the most important kind of interactive cooperation. Communication involves exchange of information through an explicit communication channel that provides an abstraction of "direct connectivity" between the communicating entities. Consequently, the behaviour applied for communication consists of activities for receiving and sending information over the dedicated communication channel [6].

The concept of communication is a specialization of the interaction concept with specific roles, connections and behaviour associated with it. As the specializations do not introduce any radically new elements to the structure presented in the context of interaction, the metamodel defining the concept of communication is not discussed here. The metamodel for communication follows the concepts defined in the definition of communication event structures discussed in the following.

The global behaviour of a communication-based cooperation is described by using the concept of *communication event structure* which specializes the concept of interaction event structures. Activities in communication event structures are defined as tuples $(from, C, I, to)$ where $from$ and to represent the participating roles, C is the connection to be used for communication, and I is the information element communicated. An *information element* is a kind of information entity that can be considered as an adequate and meaningful unit of information the properties of some information entity and that is provided with features that declare the structure or the semantics of an information entity, for example. The corresponding formalization for communication event structures is given in Definition 5.2.5.

Definition 5.2.5 (Communication event structures) *A **communication event structure** is a tuple (ES, λ) such that ES is a regular event structure and $\lambda : E \rightarrow (R \times C \times I \times R)$ is a mapping called the **communication event labelling function**, where R is a finite set of roles, C is a finite set of connections, and I is a finite set of information elements considered in the corresponding communication model. When $(r_1, c, i, r_2) \in rng(\lambda)$ then r_1 is called as the **sender** and r_2 as the **receiver**, c as the **interaction medium**, and i as **information contents** of the activity.*

5.2.2 Cooperation entity ontology

The kinds of entities in cooperative communities are classified by the ontology illustrated in Figure 5.5. An *EntityKind* describes a kind of entity where an entity is considered as an identifiable individual of some sort. Each *EntityKind* can be attached with a set of *Feature* concepts that describe the properties the corresponding kind of entities should have.

The entities encountered in cooperations can be distinguished to two different kinds, namely functional and legal entities. Functional entities provide the facilities for delivering cooperative activities while legal entities represent the cooperating parties that are bound by mutual agreements or contracts to deliver the required commitments. As the collaborative systems are based on the service-oriented computing paradigm [231, 191], the classification of functional entity kinds of cooperative entities are basically those identified from the context of service-oriented computing.

The *FunctionalEntityKind* concept represents entities (conceptual or concrete) that can be considered as elements that deliver the core functionality in cooperative communities by utilizing their behavioural features. Every entity of functional kind is associated with at least on functional feature that enables the corresponding kind of functionality; this is represented by the *functionalfeatures* association. The following functional entity kinds have been identified from service-oriented computing systems: 1) *behavioural entities*, 2) *information entities*, and 3) *endpoint entities*.

A *BehaviouralEntityKind* represents entities whose externally observable behaviour contributes to the realization of a cooperative community. Such an entity might be a conceptual service providing the behavioural patterns required by a cooperation, for example. A behavioural entity manifests observable behaviour; this is represented by the association with the *BehaviouralFeature* concept.

The information exchanged between the actors of a cooperative community comprises the primary motivation and justification for interactions. Information in the context of service-oriented computing is delivered using well-defined structured documents. The document definitions prescribe both syntactical structure and ontological interpretation for the information they represent. An *InformationEntityKind* represents entities that contribute to the information content of cooperation. Examples of such entities include business document definitions and their instances, knowledge bases, and messages communicated over channels. Standardised document description languages such as XML [267] and XML-Schema [263] can be used for definition of the structural

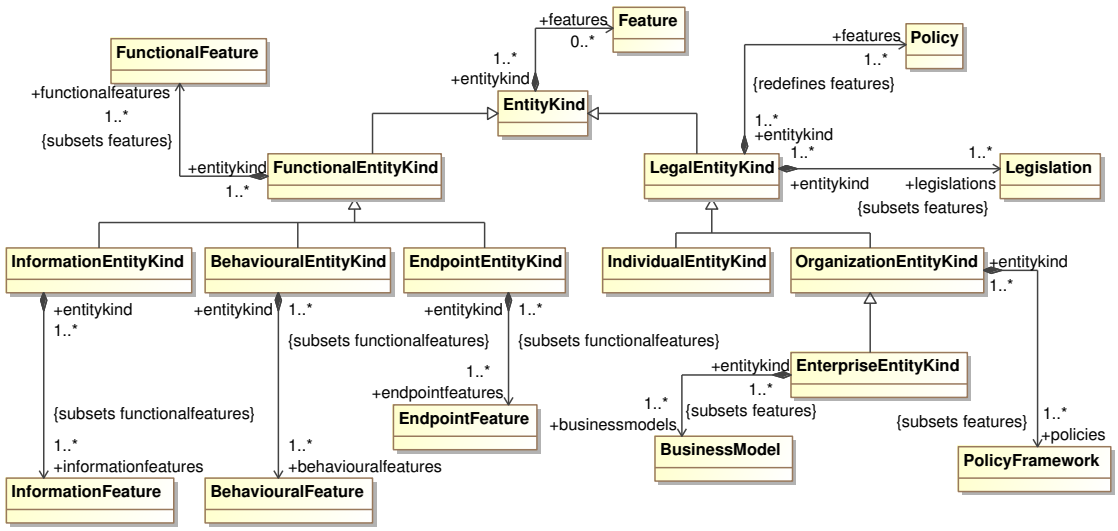

Figure 5.5: Describing the kinds of entities in cooperative communities.

features of an information entity kind. Semantic features for information entities can be prescribed using vocabularies such as ebXML [133] or RosettaNet [211], common taxonomies such as North American Industry Classification System (NAICS) [171] or generic ontology description languages, such as the OWL [259].

Service interactions are enabled by interaction endpoints that provide the necessary knowledge, such as identification and location information, for establishing bindings with the corresponding services. Interaction endpoints encountered in cooperative communities are represented by the *EndpointEntityKind* concept which manifests the local facilities reserved and provided by the cooperating parties for the purpose of delivering the required interactions. Such endpoint entities can have features such as a name, an address and properties prescribing the usage of the corresponding service, for example. The endpoint name can be used for referencing the business service indirectly; this can be utilized for example to maintain a binding between a business service and it's client during business service migration. The endpoint address provides the actual, technology specific handle to be used for interacting with the business service.

Entities that are committed to provide the functionality required for enabling cooperative activities are classified under the concept of *LegalEntityKind*. Legal entities include for example organizations and individuals. A *LegalEntityKind* represents cooperation actors that can be obligated to deliver the required functionality and behaviour through contracts and agreements. The activities of a legal entity are constrained by a set of *policies*. The concept of *Policy* describes a set of rules regulating the activities of legal entities; the concept of *Policy* is discussed in Section 5.2.3. Any legal entity is presumed to follow the legislation relevant for the corresponding context of cooperation. This is is represented by the concept of *Legislation* which is modelled as a subset of the policies.

Legal entities are classified at the top-level to organizations and individuals. An *OrganizationEntityKind* represents the kinds of organization, where an organization is *"a social arrangement which pursues collective goals, which controls its own performance, and which has a boundary separating it from its environment"* [273]. Each organization is attached with a set of organizational policies that declare organization-specific rules, such as best practices, decision making

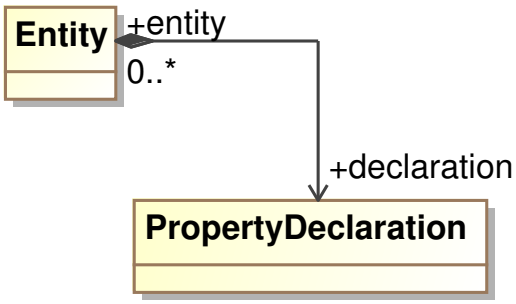

Figure 5.6: Metamodel for the concept of cooperative entity.

rules, or access policies. Such organizational policies are described under the concept of *PolicyFramework*.

An important specialization of an organization is that of an enterprise. The concept of *EnterpriseEntityKind* extends the *OrganizationEntityKind* with a collection of *BusinessModel* elements. A business model describes a set of *business rules* that are considered as declarative rules that define or constrain some aspect of business [86]. Finally, an *IndividualEntityKind* represents the kinds of natural persons, where a natural person is considered simply as *"a human being perceptible through the senses and subject to physical laws"* [272].

Ontological instances of *EntityKind* are represented by the concept of *Entity*. As illustrated in Figure 5.6 an entity provides a property declaration that defines the capabilities that are committed to and provided by the entity during cooperation. The concept of *PropertyDeclaration* is defined in the following section.

5.2.3 Cooperation feature ontology

The properties of cooperative communities and the capabilities of entities acting in the communities are prescribed by the set of features associated with the elements and entities of cooperation. A *feature* is here considered as a distinguishable capability or property of an element of a cooperative community. In distinction to traditional interpretation of features in software engineering domain (see for example [44]), here features also define the core behaviour and capabilities of functional entities instead of being just additive to the core functionality. Modelling the core functionality of entities, such as behaviour of behavioural entities, as features enables a more cohesive management of inter-relationships between different kinds of features. Features thus affect or define the properties of cooperation and may interact with each other. A *functional feature* is a feature that can be directly associated with a functional entity. A *non-functional feature* describes a feature of a legal entity, cooperation facility such as communication channels, or a service relationship.

Functional features

As discussed in Section 5.2.2, the functional entities of cooperative communities can be classified to behavioural, endpoint and information entities. Consequently the functional features in cooperative communities is classified into behavioural features, endpoint features and information features. The resulting ontology of functional features is illustrated in Figure 5.7.

In the ontology, feature interactions are described using the *interactsWith* association between *Feature* concepts. The most important kinds of features are provided below the classes of *Be-*

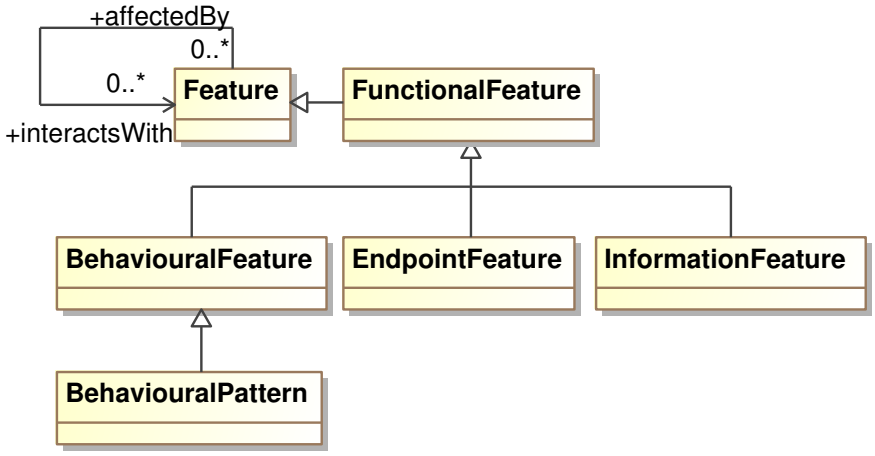

Figure 5.7: Functional features in cooperative communities.

haviouralFeature, *EndpointFeature* and *InformationFeature*. In the following we concentrate on the behavioural features. Endpoint features associated with endpoint entities and information features associated with information entities are discussed in more detail in the following sections.

A *BehaviouralPattern* describes the behaviour of a behavioural entity. The features associated with endpoint entities are classified further to transmission and messaging related features. Behavioural patterns are used for expressing sequential behaviour with finitely many distinguishable activities. The behavioural patterns are formalized using a variant of labelled transitions systems (LTS) which provide a simple and compact representation for action-based behaviour. Especially, LTS:s have well-established methods for validating interesting behavioural properties, such as liveness and safety[128] or conformance with respect to logical specifications [61]. In the context of this thesis, labelled transition systems are formulated as given in Definition 5.2.6:

Definition 5.2.6 (Labelled transition systems) *A **labelled transition system** (LTS) is a tuple (S, T, s_0, ℓ) where S is a set of **states**, $T \subseteq S \times S$ is a **transition relation**, s_0 is the **initial state**, and $\ell : T \to L$ is a **transition labelling function** where L is a finite set of **labels**.*

We let $a, b, c, \ldots$ range over labels in L and write $s \xrightarrow{a} s'$ when $t = (s, s') \in T$ and $\ell(t) = a$.

*Let $A = (S, T, s_0, \ell)$ be a labelled transition system. For a possibly empty sequence of activities $v = \alpha_1\alpha_2 \ldots \alpha_n$, such that $\alpha_1, \alpha_2, \ldots, \alpha_n \in L$ and $s, s' \in S$, we write $s \xrightarrow{v} s'$ if and only if $s = s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_n} s_k = s'$ for some $s_1, s_2, \ldots, s_k \in S$. We call the a sequence of transitions $(s_0, s_1)(s_1, s_2) \ldots (s_{n-1}, s_n)$ a **computation**. The corresponding sequence of labels $\alpha_1\alpha_2 \ldots \alpha_n$ is called a **computation trace** or simply a **trace**. The set of all traces of an LTS A is denoted as $Tr(A) = \{v | v \text{ is a trace in } A\}$.*

Before giving the definition for behavioural patterns, we have to introduce some concepts related to languages, regularity of languages, and computation traces. An *alphabet* is a finite, nonempty set of symbols [104, pp. 28]. For an alphabet $\Sigma = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ a *word* is a finite sequence of symbols chosen from the set Σ. We let $v, w, \ldots$ to range over sequences. An empty sequence ϵ is a sequence with zero occurrences of symbols and it can be chosen from any alphabet whatsoever [104]. Sequences are prefix closed: if $v = \alpha_1\alpha_2 \ldots \alpha_n$ is a sequence, the also every prefix $\alpha_1 \ldots \alpha_k, 1 \leq k \leq n$ of v is also a sequence. The set of finite sequences over Σ is denoted

as Σ^*. Concatenation between a word and a symbol is denoted as $v\alpha$ when $v \in \Sigma^*$ and $\alpha \in \Sigma$. If Σ is an alphabet and $L \subseteq \Sigma^*$, then L is a *language over* Σ [104, pp. 30].

Words can also be defined as labelling functions defined over linearly ordered sets. A *linearly ordered set* $(L, \leq)$ is a set equipped with a reflexive, transitive, anti-symmetric and total order. Sometimes we simply say that L is a linearly ordered set to mean that $(L, \leq)$ is a linearly ordered set. A *morphism* $h : (L, \leq) \to (L', \leq')$ between linearly ordered sets is a function $h : L \to L'$ which preserves the ordering: for every $l, l' \in L$ it holds that if $l \leq l'$ then $h(l) \leq' h(l')$.

Now, a *(generalized) word* $(L_u, \leq, u, A)$ consists of a linearly ordered set $(L_u, \leq)$, a non-empty set of *labels* A, and a *labelling function* $u : L_u \to A$ [36]. We use the the labelling function u to name the word, and let $(L_u, \leq)$ to denote the underlying linear order under u. We say that u is a word on A, or u is a A-labelled word, over the linear order $(L_u, \leq)$ [36]. The range of the labelling function is called as the alphabet of the word. When L_u is empty, we have the empty word which is denoted with ϵ.

Assume that u and v are words with underlying linear orders $(L_u, \leq)$ and $(L_v, \leq')$. Now morphism $h : u \to v$ is a morphism $h : (L_u, \leq) \to (L_v, \leq')$ which preserves the labelling: when $x \in L_u$ then $u(x) = v(h(x))$ [36]. Two words u and v are *isomorphic* when there are morphisms $h : u \to v$ and $g : v \to u$ such that $g(h(x)) = x$ and $h(g(y)) = y$ for every $x \in L_u$ and $y \in L_v$. We identify words that are isomorphic.

For defining the notion of regularity over languages, we need to introduce the concepts of equivalence relations and classes:

Definition 5.2.7 (Equivalence relations and classes, and quotient sets) *Let A be a set of elements. An **equivalence relation** R over set A is a binary relation $R \subseteq A \times A$ that is reflexive, symmetric and transitive. We write aRb when $(a, b) \in R$ and say that a and b are equivalent.*

*When $a \in A$ we represent its **equivalence class** by $[a] = \{b \in A | aRb\}$. Equivalence relation R over set A partitions elements of A into disjoint equivalence classes. The **quotient set** A/R of A by R is the set of all its equivalence classes: $A/R = \{[x] | x \in A\}$.*

An equivalence relation R over the alphabet Σ is *right-invariant* if for all $v, w \in \Sigma^*$, vRw if for all $z \in \Sigma^*$ $vzRwz$ [224]. Such a right-invariant equivalence relation comprises of equivalence classes where each of the classes include words that can not be *distinguished* by any suffix $z \in \Sigma^*$. We denote a right-invariant equivalence relation over language L as R_L which is defined as follows:

Definition 5.2.8 (Equivalence relation R_L over language L [224]) *Given a language L over Σ, the equivalence relation R_L is defined as follows: words $v, w \in \Sigma^*$ are equivalent, vR_Lw, iff for all $z \in \Sigma^* : vRw \Leftrightarrow vzRwz$.*

Now, regularity of language L can be characterized through R_L by the Myhill-Nerode theorem:

Theorem 5.2.1 (Myhill-Nerode theorem (as formulated in [224])) *L is a regular language iff R_L has a finite number of equivalence classes. Furthermore, if L is regular, it is the union of some of the equivalence classes of R_L.*

Now, behavioural patterns are formalized as described in Definition 5.2.9

Definition 5.2.9 (Behavioural patterns) *A **behavioural pattern** is an LTS (S, T, s_0, ℓ) that is*

1. ***finitely branching**, $\forall s \in S : \{s' | \exists s' \in S : (s, s') \in T\}$ is finite,*

2. ***without backward branching***: $\forall (s', s), (s'', s) \in T : s' = s''$, *and*

3. ***deterministic***: $\forall t = (s, s'), t' = (s, s'') \in T : \ell(t) = \ell(t') \Rightarrow s' = s''$.

*The elements in the range of function ℓ, denoted $rng(\ell)$, are called **activities**. We let $\alpha, \beta, \ldots$ to range over activities.*

*We further restrict the expressibility of behavioural patterns by taking into consideration only labelled transition systems B that express **regular behaviour** in a sense that $Tr(B) \subseteq rng(\ell)^*$ is a regular language.*

In behavioural artifacts that are derived from the formulation of the behavioural pattern the activities are structured instead of being simple labels. The specializations of behavioural pattern extend the transition labelling function to reflect the semantics of the corresponding kinds of behaviour, interaction or communication, for example. This is the reason to emphasize the distinction between the set of labels L and the set of activities in $rng(\ell)$ in the definition of the behavioural patterns (and not to call the set of labels L as activities).

An *interaction pattern* is a kind of behavioural pattern composed of activities provided by an interaction endpoint. In an interaction pattern each interaction transition can be considered as a tuple (s, a, m, s') where $s, s' \in S$ are states, $a \in L$ is an activity label, and $m \in M$ is a *modality* which defines the necessity of an activity.

Consequently, the formal semantics for interaction patterns is based on a variant of modal transition systems [145] which allow characterization of behavioural *compatibility* and *refinement* [145] in a generic, extendable and domain independent manner. Interactions pattern formalization extends the structure defined for behavioural patterns with a labelling function that considers activities having modalities. Formalization for the concept of interaction pattern is given by Definition 5.2.10.

Definition 5.2.10 (Interaction patterns) *An **interaction pattern** is a behavioural pattern (S, T, s_0, ℓ) where the transition labelling function $\ell : T \to (L \times M)$ attaches each transition with a label and a modality characterizing the interaction activity such that L is a finite set of labels and M is a finite set of **modalities**.*

*The set of modalities contains two subsets, $M_{may} \subseteq M$ the set of **may (allowable) modalities** and $M_{must} \subseteq M$ the set of **must (required) modalities**. We let $m, m', \ldots$ to range over modalities.*

*When $t = (s, s') \in T$, $\ell(t) = (a, m)$ and $m \in M_{may}$ ($m \in M_{must}$) we write $s \xrightarrow[may]{m(a)} s'$ ($s \xrightarrow[must]{m(a)} s'$) for an **allowed action** (required action).*

*We say that the set M of modalities is **consistent** if there exists a symmetric bijection $dual : M \to M$ such that for each $(m, m') \in dual$ it holds that either $m \in M_{may}$ and $m' \in M_{must}$ or $m \in M_{must}$ and $m' \in M_{may}$. When $(m, m') \in dual$ we say that m' is the* dual *(modality) of m (and vice versa). When $m \in M$ is a modality of a consistent set of modalities we denote its dual as $\overline{m}$; using this notation it holds that for all $m \in M$: $dual(m) = \overline{m}$ when M is consistent.*

Finally, communicative behaviour is specified using structures that provide communication-related specializations of the behavioural artifacts for describing interaction. Communication endpoint behaviour is defined using so-called *communication patterns* where each transition is considered as a tuple (s, a, m, i, s'), where $s, s' \in S$ are states, $a \in L$ is an activity label, and $m \in M$ is the modality of the activity, and $i \in I$ is an information element describing the properties of the information entity communicated.

Communication-endpoint activities allow two kinds of activities to be taken, having *"send"* and *"receive"* modalities, correspondingly. The formalization for the concept of communication patterns is given in Definition 5.2.11.

Definition 5.2.11 (Communication patterns) *A **communication pattern** is a behavioural pattern (S, T, s_0, ℓ) where the transition labelling function $\ell : T \rightarrow (L \times M \times I)$ labels transitions with a label, a modality and an information element representing the characteristics of the communicative activity such that L is a finite set of labels, M is a finite set of modalities, and I is a finite set of **information elements**.*

Non-functional features

Non-functional features are further classified to extra-functional and collaborative features. *Extra-functional* features address issues related to the dependability of cooperative activities and the underlying communication platform, such as messaging and transmission features of the communication medium. *Collaborative features* address issues related to the pragmatic interoperability, such as policies and business rules. The ontology for non-functional features is illustrated in Figure 5.8. The classification given is naturally incomplete and can be later provided with more top-level concepts, if necessary. The domain ontology for service-based communities introduces another class of non-functional features, namely contractual features, for example (see Section 5.3.3).

Policies regulate the activities of legal entities and are described by the *Policy* concept. A policy in the context of cooperative communities represents a set of rules that constrain, regulate or otherwise affect the behaviour or properties of the collaboration in question. Policies are attached to legal entities. *Legislation* comprises a set of legislative rules called as *Act*s. Every domain of concern may have its own regulating legislative system. In the example illustrated in Figure 5.1 enterprise entities that represent public limited companies are bound by the corresponding legislation, for example. A *PolicyFramework* describes a set of organizational policies that can be bound to organizational entities. Finally, a *BusinessModel* followed by an enterprise entity is comprised of a set of business rules.

Property declarations

A property declaration describes the set of properties that can be provided by an entity, such as an endpoint entity, or a specific cooperation facility (see Section 5.2.4), such as a communication channel, for example. Property declarations are constrained by the features supported by the corresponding kind of cooperation entity or facility. As an example, an endpoint entity kind can support several endpoint features; the corresponding endpoint entities can provide property declarations defined over properties that are ontological instances of the corresponding kinds of features. Property declarations are defined using the concept of *PropertyDeclaration* whose intention is illustrated in Figure 5.9.

A *PropertyDeclaration* can either be a single property assertion providing a value for a property, or a set of properties. The concept of *PropAssertion* associates a *Property* with a *PropValue* which provides a value from the value set of the corresponding property. A property assertion can also be defined without value; in this case the corresponding property represents a boolean option (either the property is available or not). Each *PropertySet* may comprise of a selection of subsets, and is further classified into five distinct categories. The classification provides several set constructors over property assertions and follows the requirements of the Pilarcos populator service [139]. *PropNoneOf* property set declares an exclusive selection of property declarations

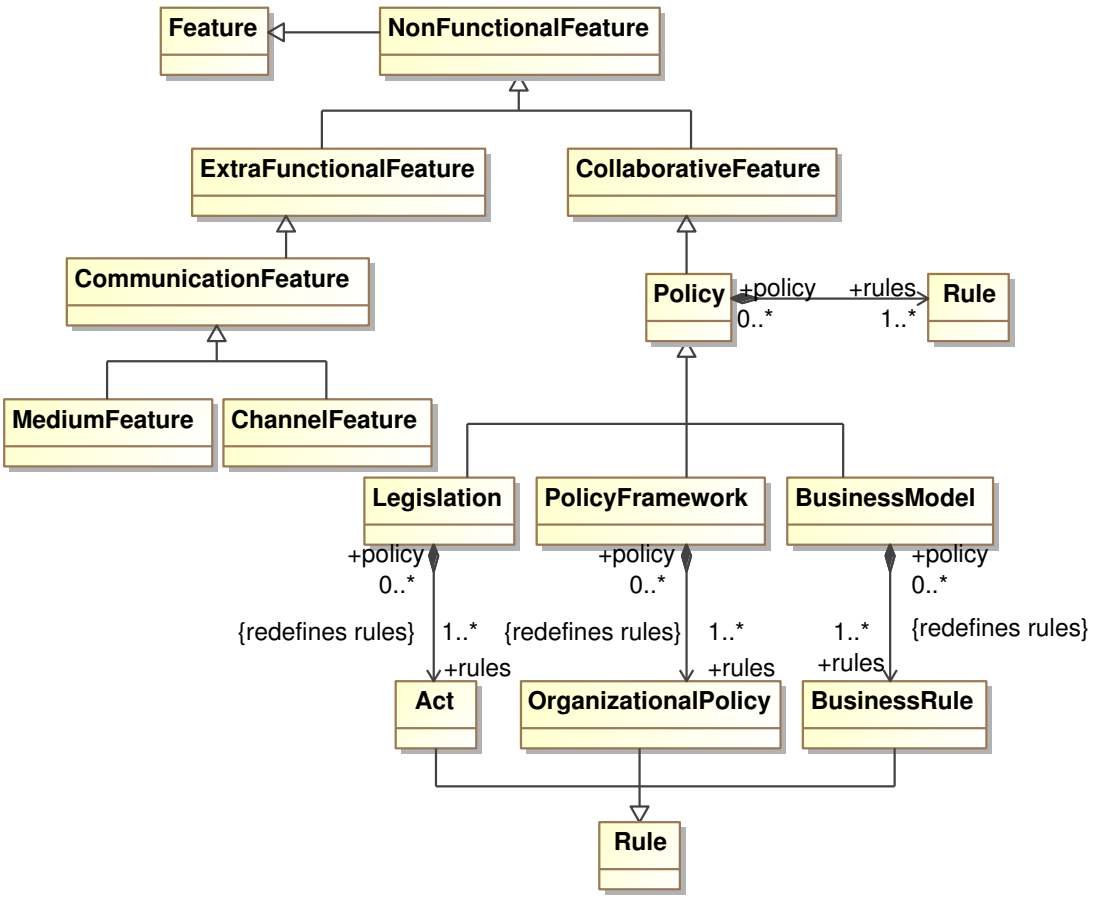

Figure 5.8: Non-functional features in cooperative communities.

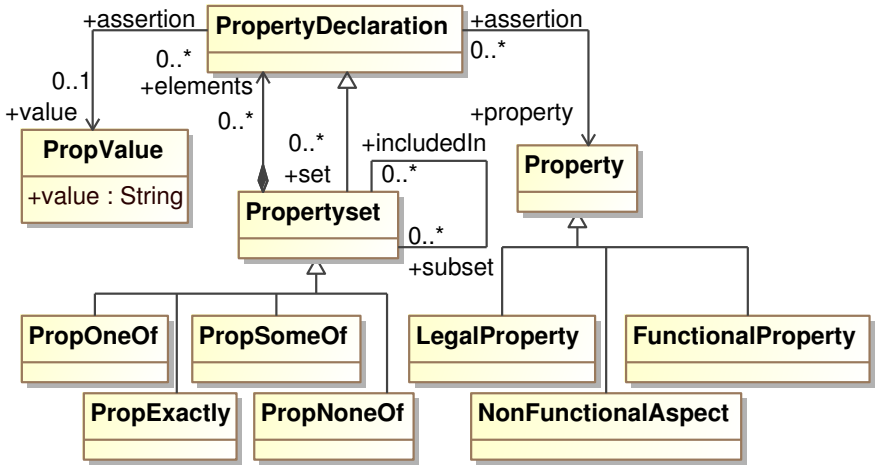

Figure 5.9: A metamodel describing the concept of property declaration.

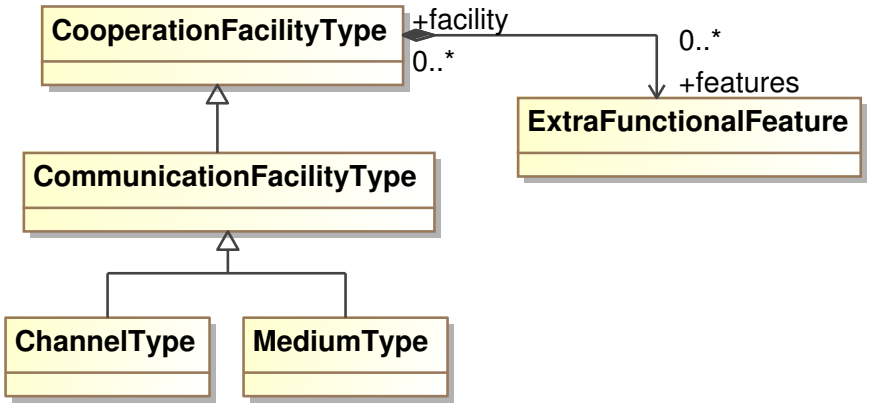

Figure 5.10: Metamodel defining the kinds of cooperation facilities.

needed for declaring restrictions over the open universe of cooperation facility properties. A property set defined with the concept of *PropExactly* declares that all the property declarations included in the set have to be available. Property sets of kind *PropSome* and *PropOneOf* declare that some of or exactly one of the included elements have to be chosen from the available property declarations, correspondingly.

5.2.4 Cooperation facility ontology

In addition to the foundational elements of cooperation introduced above, cooperation must typically be provided with secondary artifacts for facilitating interactions, for example. In the domain ontology for cooperative communities communication is considered as a kind of interaction which involves an explicit channel abstraction. Subsequently, abstractions for channels and media is needed.

Different kinds of cooperation facilities are represented by the concept of *CooperationFacilityKind*, as illustrated in Figure 5.10. Whereas functional and legal entity kinds of cooperation are associated with functional and contractual features, the cooperation facilities can be provided with features represented via the concept of *CooperationFeature*. Communication facilities are regarded as a kind of cooperation facilities. In the domain ontology for cooperative communities communication facilities are further classified to two separate concepts. The concept of *ChannelType* is used for defining different kinds of channel abstractions, and the concept of *MediumType* for defining the kinds of media available for instrumenting communication.

Cooperation facility instances are represented by the concept of *CooperationFacility*, as illustrated in Figure 5.11. Especially, whereas functional and legal actors in cooperation are associated with properties, cooperation facilities are associated with *non-functional aspects*, that is properties that are ontological instances of the *ExtraFunctionalFeature* concept. A non-functional aspect declares a capability of shared cooperation abstraction, such as communication channel or medium. Such capability may provide a specific kind of messaging scheme or encryption of messages to be used in communication, for example. Consequently, capabilities that are typically regarded as "functional" in technology-driven frameworks, such as SOAP transportation bindings in the context of WSDL [55], are considered secondary to the service functionality and behaviour in this framework and thus non-functional.

A communication channel is considered in the domain ontology as a concept that provides

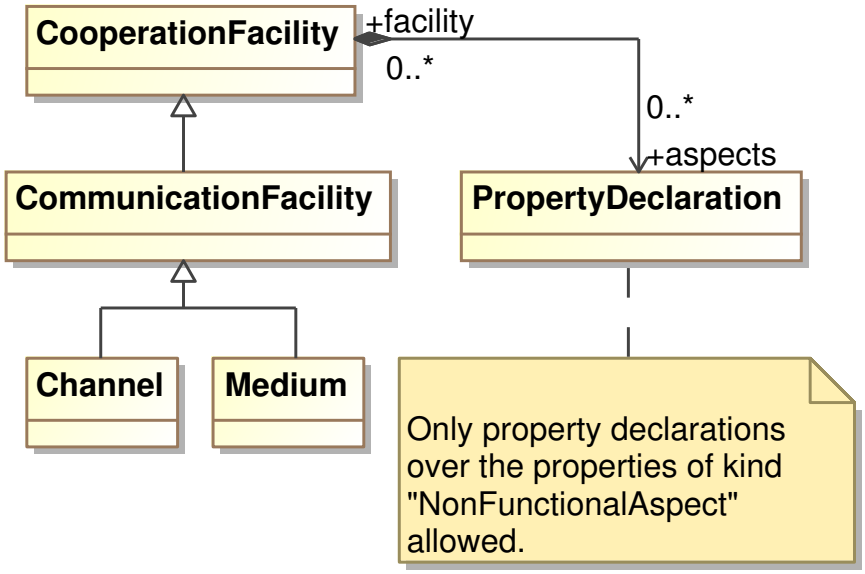

Figure 5.11: Metamodel defining the cooperation facilities.

support for communicative activities and mediates the corresponding information over dedicated communication media. From the cooperation perspective, each *CommunicationConnection* defined in a *CommunicationType* will be provided with a communication channel to instrument the cooperative activities between communication endpoints. Different channel types can be defined for supporting different kinds of communication activities. A channel type essentially defines the set of communication activity modalities supported, routing of activities to appropriate media, and a phasing of actions for realizing appropriate kind of communication.

The metamodel formalizing the concept of communication channel type is illustrated in Figure 5.12. A *ChannelType* comprises of a set of *Modality* elements, two *ChannelPortTypes*, and a set of *CommunicationPhase* elements. A modality in this context defines the necessity of communicative activities and provides the means for declaring semantics for communication. Modalities such as *"send"* and *"receive"* or *"publish"* and *"notify"* can be used for describing the kinds of activities found in typical distributed systems, for example.

The concept of *ChannelPortType* defines properties for a kind of communication ports. Each *ChannelPortType* is associated with usage instructions defining how communication activities directed to corresponding communication ports are mediated to different communication media. Communication media usage is based on partitioning of communication activities based on their modalities. A communication channel may use two communication media for separating up- and down-stream communication, for example. Communication media usage instructions are declared by a set of *CommunicationUsage* elements. The set of modalities declared by *CommunicationUsage* elements must cover all the modalities declared for the corresponding kind of *ChannelType*.

Each *ChannelPortType* is associated with a set of communication phases that describe what kind of facilitating actions are needed for realizing communication. A *CommunicationPhase* represents a phase in communication in which some processing actions related to different aspects of communication will be applied by the communication ports. Each communication phase can

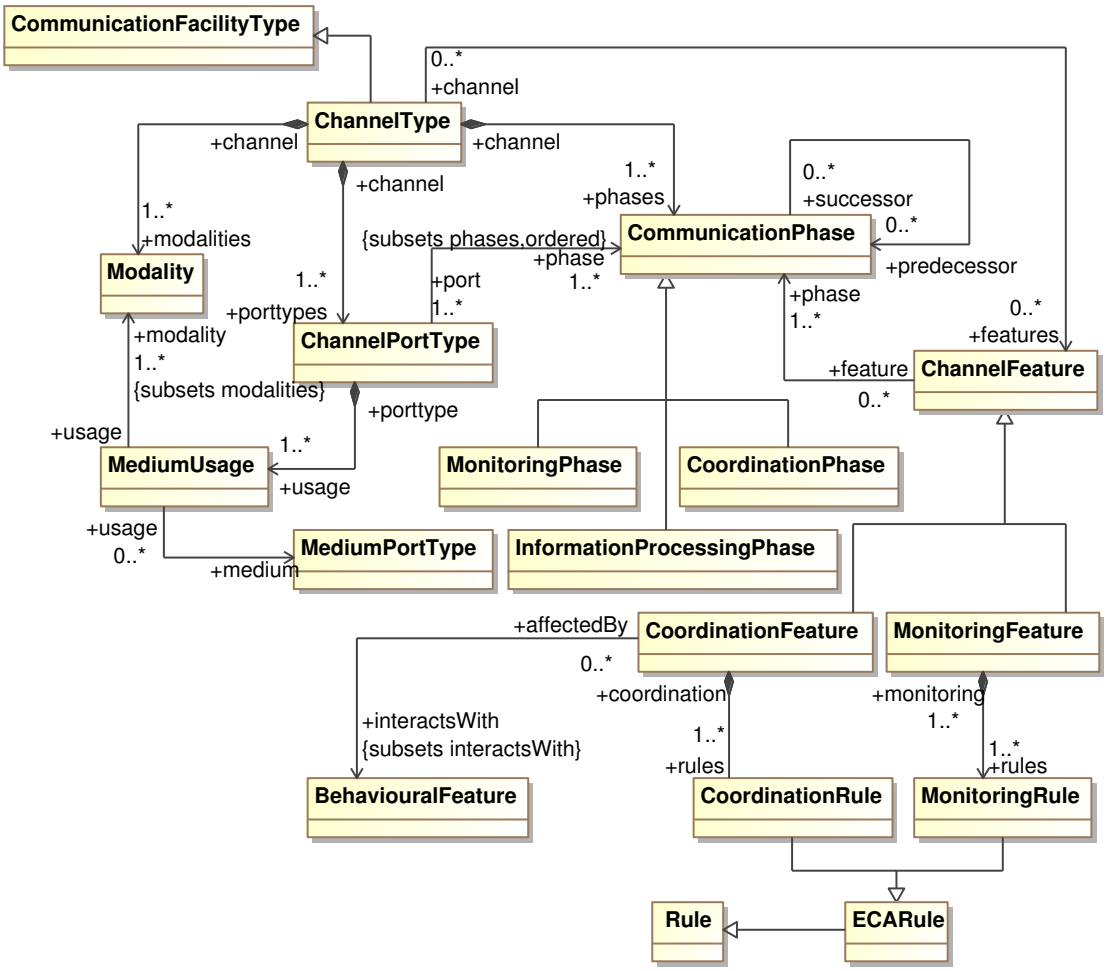

Figure 5.12: Metamodel for communication channel types.

depend on a set of other communication phases acting as necessary conditions for that phase. A certain kind of coordination phase may require that a specific kind of monitoring phase is in place, for example.

The classification of communication phases comprises monitoring, information processing and coordination. Monitoring provides the means for observing the activities mediated over a communication medium. Monitoring is here considered as a kind of passive interception of communication, in a sense that the actions taken in a monitoring phase do not affect the contents, behaviour or semantics of communication.

The information processing phase comprises of actions that affect in way or another the information contents of communication. This may involve attaching digital signatures to business documents, hiding information due to organizational privacy policies, or adding collaboration context specific additional information to the communicated messages, for example.

Coordination phase comprises of actions that regulate, alter or constrain communicative behaviour. Such coordinating actions may involve notifications of activities taken within a communication medium, adding behavioural patterns to the communicative behaviour due to enterprise business rules, or neglecting some behavioural options available due to collaboration context, for example.

Finally, each *ChannelType* can be associated with a set of channel features. The classification of the *ChannelFeature* concept involves features for monitoring and coordination. Each *ChannelFeature* must be associated with at least one communication phase. A channel feature can be reused by several distinct channel types, as long as they are used within similar communication phase (monitoring, information processing, or coordination).

A *CoordinationFeature* is used within the communication channels to coordinate the cobehaviour of individual behavioural entities. Coordination requirements are described using coordination rules represented by the *CoordinationRule* concept. A coordination rule is a specific kind of event-condition-action–rule (*ECARule*) that reacts on the behavioural events (e.g. reception of a specific message) happening in a communication channel. Channels may also involve different kinds of monitoring for the purpose of facilitating collaborations. Such elements are described using the *MonitoringFeature* concept which comprises a set of monitoring rules. A *MonitoringRule* is also a reactive ECA-rule specifying the actions to be taken when some specific event takes place in communication.

The ontological instance of a *ChannelType* is known as a *Channel* and its intention is provided by the metamodel illustrated in Figure 5.13. A *Channel* comprises two ports providing bindings between communication endpoints and media. The concept of *ChannelPort* defines such bindings by using the *EndpointBinding* concept that associates a *MediumPort* that is discussed below with a *CommunicationEndpoint*.

While communication channel concepts provide means for addressing the semantics of communication, the concepts formalizing communication media provide a technologically oriented view on communication. A communication medium is considered as an abstraction for information mediation which involves two communication ports providing features needed for realizing the communication. Reflecting this abstraction, a concept of medium type is used for describing different kinds of communication media comprising of a set of communication features and two medium port types.

The intention for the concept of medium type is defined by the metamodel illustrated in Figure 5.14. A *MediumType* comprises a set of features defining the properties and capabilities of corresponding kind of communication media. The concept of *MediumPortType* defines a kind of communication port as an ordered subset of the medium type communication features.

Medium features describe the capabilities needed for initializing a message for the actual com-

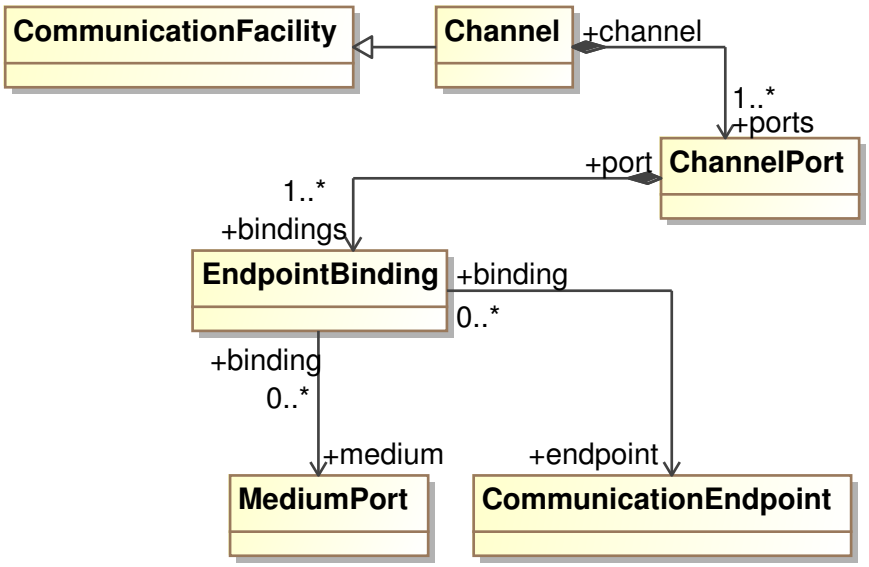

Figure 5.13: Metamodel for communication channels.

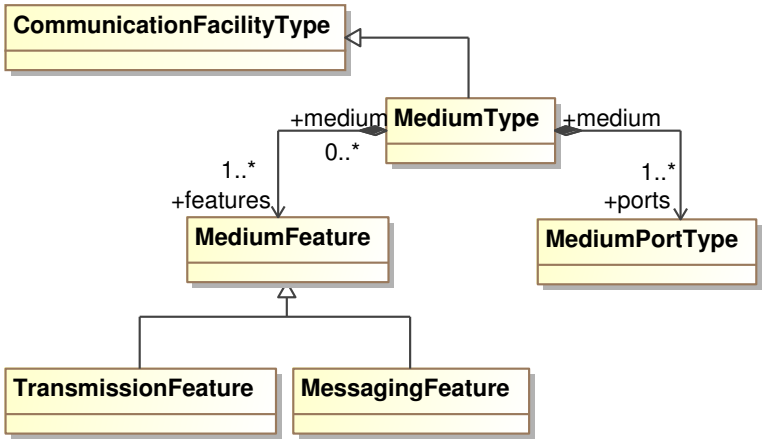

Figure 5.14: Metamodel for defining medium types.

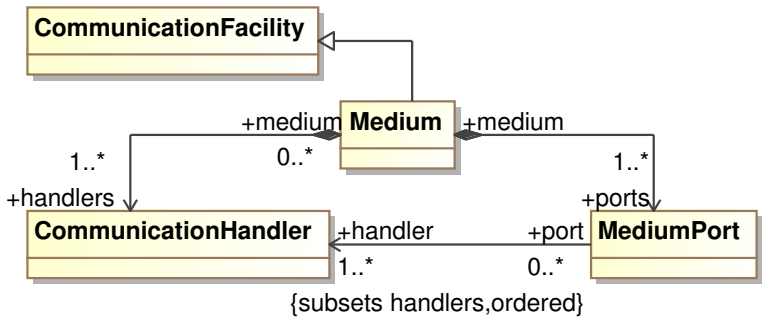

Figure 5.15: Metamodel for describing communication media.

munication. Medium features are classified to messaging features and transmission features. The concept of *MessagingFeature* is used to represent capabilities that provide serialization of information contents into an appropriate digital format, such as XML.

The concept of *TransmissionFeature* is a representation for communication medium capabilities that provide support for enveloping the message contents into protocol specific frames, or for encryption and decryption of messages, for example. A *TransmissionFeature* may declare the transmission protocol used for communication, such as HTTP [208], for example. The messaging features may declare that an endpoint entity uses SOAP [265] or REST [92] messaging for communication, for example. Information features declare the structure and semantics of information by providing structural descriptions and references common ontological concepts, for example.

It should be noted that in contrast to quite common interpretations, communication medium features are considered as non-functional features in the domain ontology for cooperative communities. This is a conscious design decision justified by the fact that communication medium represents a pure abstraction within the domain ontology which provides the means for discussing shared features of communication. That is, a communication medium is *not* a functional entity, a thing justifying its own existence, and can not thus be associated with functional features, as discussed in Section 5.2.3. More over, communication medium features can be considered as extra-functional capabilities of communication which do not change or affect the actual properties, such as information contents or behaviour, of communication.

Communication media instances are defined in accordance to the concept of *Medium*, whose intention is illustrated in Figure 5.15. A *Medium* defines a set of medium ports and a set of communication handlers. The *MediumPort* concept represents an end point of the communication medium abstraction and comprises a set of *CommunicationHandler*s implementing the features declared in the *MediumPortType* of the corresponding *MediumType*.

5.3 Ontology for service-based communities

The domain ontology for service-based communities consists of five foundational ontologies representing different concerns of service-oriented computing environments. It builds upon the ontology for cooperative communities and provides specializations for several concepts described by the cooperative communities domain ontology.

Behaviour in service-based communities is manifested by three different artifacts. The behaviour provided by conceptual services and implemented by business services are described by

service conversations and *orchestration* processes, correspondingly. Service conversations are especially used for providing the means for validation of service interoperability. Orchestration is applied by service providers for composing their computational service into business processes providing more complex behaviour conforming to requirements laid for a certain cooperation role, for example. Finally, the global behaviour of service-based communities is defined by *choreographies* which describe business document exchange taking place between service endpoints.

In addition, the domain ontology for service-based communities introduces ontologies for service grounding and contracting purposes. The top-level concepts of these ontologies are illustrated in Figure 5.16.

The *ServiceCooperation Ontology* describes the concepts of *ServiceChoreography* and *ServiceContract*, which are specializations of the *Communication* and *Cooperation* concepts, correspondingly. The *ServiceEntity* ontology provides service-based specializations of the *CooperationEntity* concepts while *ServiceBehaviour* specializes the ontology of *CooperationBehaviour* to better reflect the nature of modern electronic business networks. The *ServiceGrounding* ontology represents artifacts of service-oriented computing environments that are needed for establishing the bindings between conceptual service, business services and computational services, and contextualizing services with contractual and communication specific information. The detailed descriptions of these concepts and their intentions are provided in the following sections.

5.3.1 Service cooperation ontology

Service cooperation ontology defines concepts needed for declaring the cooperation structures of service-based collaborations. Service-based collaborations involve communication between service endpoints that are bound by the contractual relationships committed to by the legal entities providing the required business services. Following this characterization, two kinds of cooperative relationships need to be conceptualizes: service choreographies describing the communication behaviour of service-based communities, and service contracts declaring the contractual relationships between cooperation partners.

The concept of service choreography is a specialization of the communication concept described in Section 5.2.1. As illustrated in Figure 5.17 the specializations are straightforward, and structurally the *ServiceChoreographyType* concept does not add any new elements when compared to the corresponding elements of communication. The cooperation now takes place between service endpoints and the behavioural features of the cooperation is described by using the concept of *ChoreographyEventStructure* that composes *ChoreographyActivity*-elements provided by the *ServiceChoreographyType*.

A *choreography* defines a distributed business process that coordinates collaboration between a set of distributed services exchanging business documents. A choreography is declared by a set of choreography roles and the mutual activities taking place between the roles. Choreography behaviour is formalized in the domain ontology of service-based communities with *choreography event structures*. A choreography event structure is a specialization of the communication event structure concept where the concept of information elements is replaced with the more specialized notion of business documents. The formalization of choreography event structures is given in Definition 5.3.1.

Definition 5.3.1 (Choreography event structures) *A **choreography event structure** is a tuple* (ES, λ) *such that* ES *is a regular event structure and* $\lambda : E \rightarrow (R \times C \times D \times R)$ *is a mapping called the **choreography event labelling function**, where* R *is a finite set of roles,* C *is a finite set of connections, and* D *is a finite set of **business document types** considered in the service chore-*

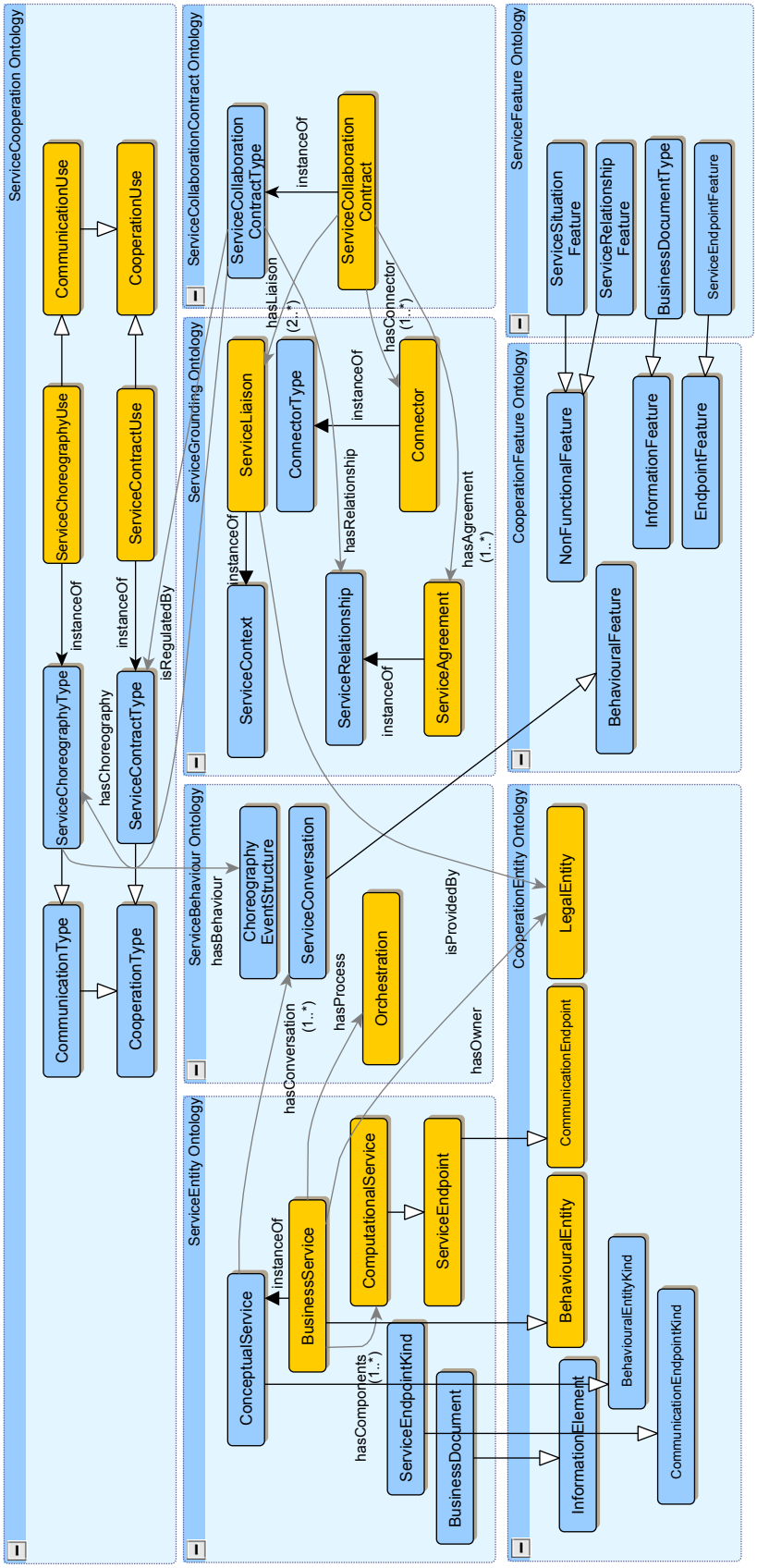

Figure 5.16: Top-level concepts for the service-based community ontology.

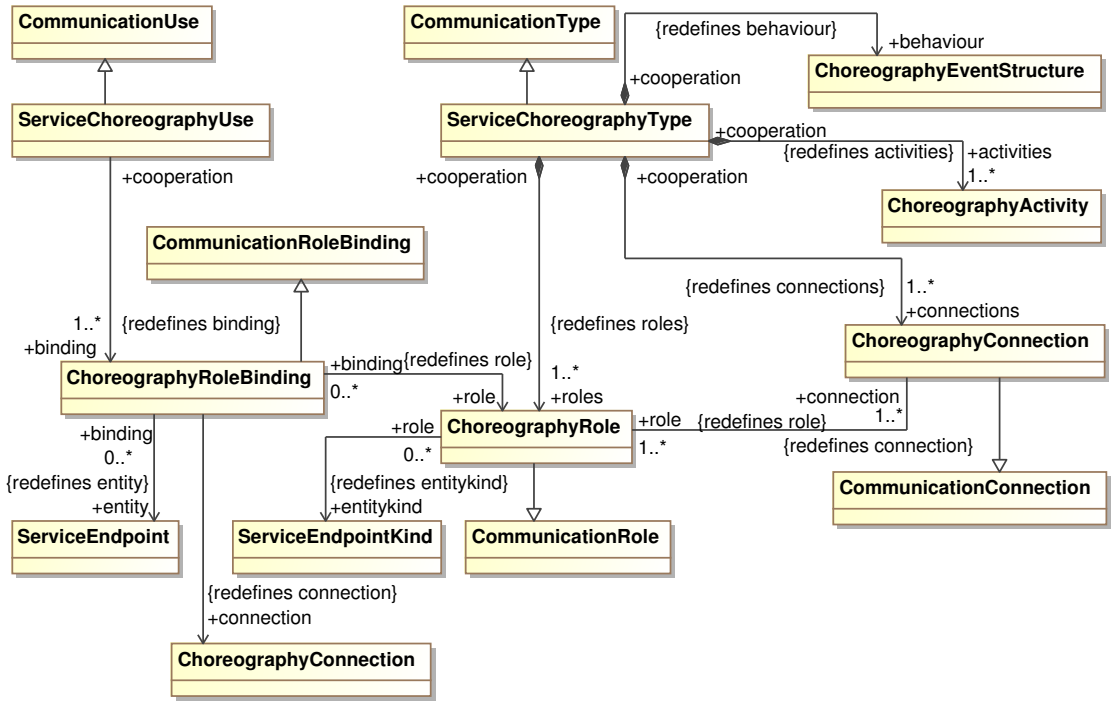

Figure 5.17: Service choreography ontology.

ography model. When $(r_1, c, d, r_2) \in rng(\lambda)$ *then* r_1 *is called as the* ***sender*** *and* r_2 *as the* ***receiver*** *of the activity,* c *as the* ***interaction medium****, and* d *as* ***information contents*** *of the activity.*

Service relationship is regarded as a cooperation between two legal entities, such as an organization and an individual. The legal entities are designated typically with a role of service provider or service consumer. The service provider is committed to deliver functionality conforming with the corresponding conceptual service. The service consumer is expected to use the service in a way that is in line with the service provisioning scenario.

As illustrated in Figure 5.18, the concept of *ServiceContractType* specializes *CooperationType* by restricting the types of roles as well as connections applicable. A service contract is considered as a cooperation relationship between legal entities, as declared by the notion of *LegalRole*, which subsumes the kind of roles needed for service relationships. The legal roles are inter-related by corresponding kinds of connections.

5.3.2 Service entity ontology

Service-based communities are populated by functional entities representing different aspects of service-oriented computing. The kinds of functional entities encountered in service-based communities are classified at the top-level to conceptual services, service endpoints, and business documents. These concepts are represented by the elements of *ConceptualService*, *ServiceEndpointKind*, and *BusinessDocument* in the metamodel illustrated in Figure 5.19.

A conceptual service determines the "idea" of a service that should be provided to clientele. This client-centric intent of a service is represented in the concept of *ConceptualService* as a set of *service conversations*, following the common terminology laid in [98, 42, 19] for example.

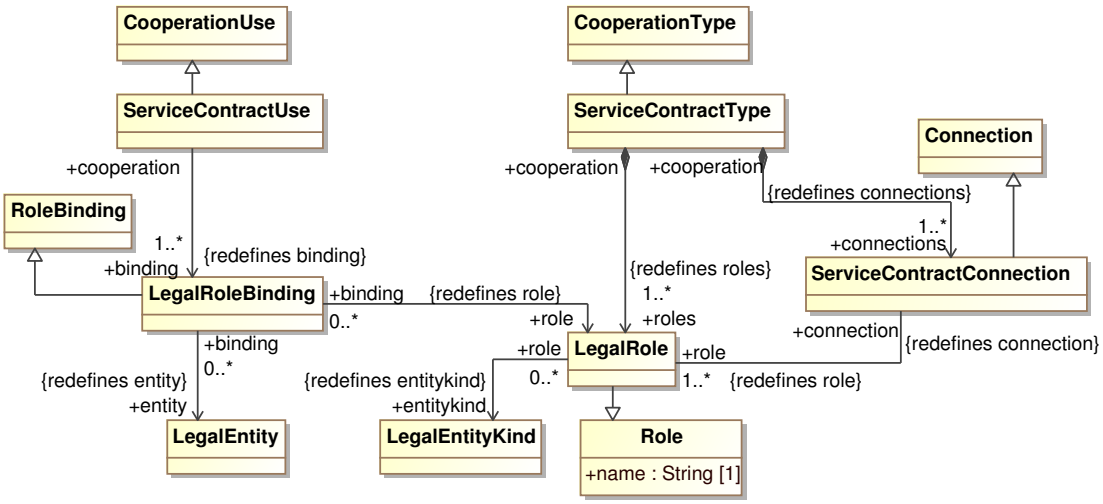

Figure 5.18: An ontology for describing service contracts.

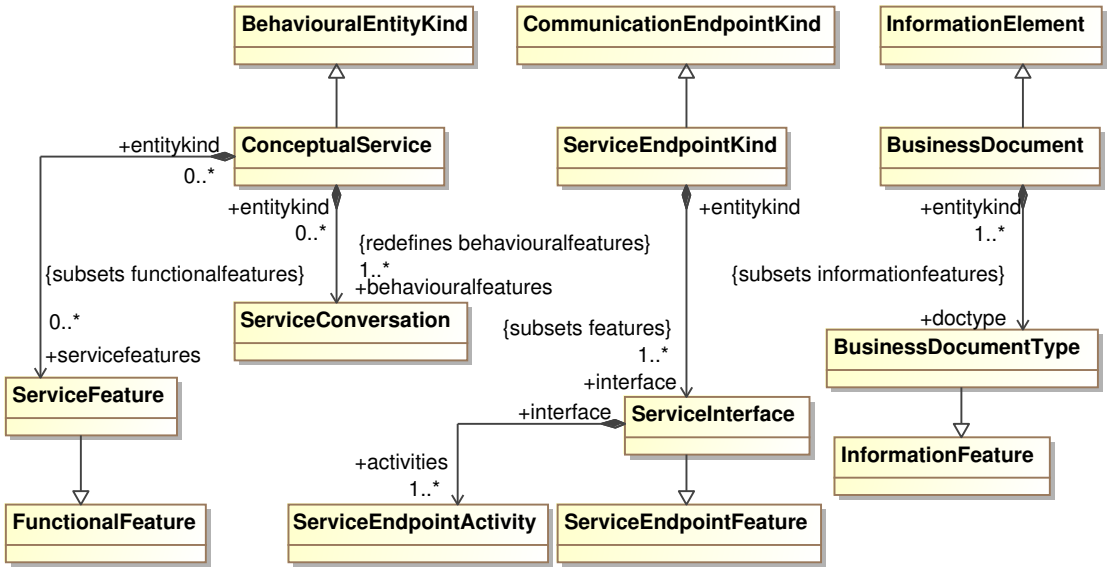

Figure 5.19: An ontology describing the functional entity kinds in service-based communities.

Each *ServiceConversation* is a specialization of the *BehaviouralPattern* concept. In addition for specializing the kind of behaviour accepted for services, the conceptual service introduces a new kind of functional feature, namely *service features*. A *ServiceFeature* represents functional features that are meaningful in the context of conceptual services. Such features are specified in domain ontologies of their own, and typically represent enumerations of selectable service properties. These kinds of functional service features together with the extra-functional properties of the non-functional feature spectrum represent static and selectable properties used during service discovery for finding, differentiating and selecting appropriate service instances.

Conceptual services are identified using domain analysis methods such as use case analysis, customer feedback or interviews. Requirements and features laid for a conceptual service are then typically formalized in service definitions. The service definitions must be formal enough to prevent unambiguous interpretations about the purpose, applicability and behaviour of the service.

The kinds of service endpoints are represented in the ontology by the concept of *ServiceEndpointKind*, which is a specialization of the *CommunicationEndpointKind*. Each *ServiceEndpointKind* is provided with an obligatory feature, namely its interface. The *ServiceInterface* describes the kinds of activities supported by the corresponding kind of interface. Activities are represented by the concept of *ServiceEndpointActivity* and they are used in service conversations as units of behaviour.

Business documents are information elements that are associated with an explicit description of their structural features. Business documents are used in service conversations and service choreographies to represent the business information to be communicated. In the metamodel, the concept of *BusinessDocument* is used for representing business documents.

At the functional entity instance level of service based communities, an instance of a conceptual service is known as a *BusinessService*. As illustrated in Figure 5.20, the intention of the *BusinessService* concept comprises a property declaration, a set of *endpoints*, and an optional *process*. The property declaration may only involve properties that are ontological instances of the *ServiceFeature* concept. Endpoints are provided by a set of *computational services*. A computational service implements the business logic by utilising the enterprise infrastructure services and enterprise applications, and provides a service-oriented interface for accessing the corresponding business functionality. A business service is owned by a legal entity.

The *ComputationalService* is a locatable service artifact and a specialization of the *ServiceEndpoint* concept that provides a set of service operations for accessing business functions. Each *ServiceOperation* associated with a *ServiceEndpoint* is an ontological instance of a *ServiceEndpointActivity* defined in the service interface of the corresponding *ServiceEndpointKind* concept.

An *orchestration* describes a locally executable business process which coordinates a set of computational services in such a way that the resulting composite service can fulfill the requirements of the targeted cooperative contexts. Typically orchestration processes are implemented *a priori* by service providers for targeting specific choreography role prerequisites. However, dynamic orchestration process establishment in an *ad hoc* manner can also be considered as a viable alternative when provided with appropriate facilities. Such *dynamic service composition* involves usually some kind of formal reasoning and synthesis of the composition logic, see for example [22] and [107].

In the domain ontology for service-based communities the notion of orchestration is considered as an instance-level concept, in contrast to service conversations and choreographies that are defined at the conceptual type level. An orchestration describes a locally executable business process that is associated with a business service. The metamodel describing the intention of the orchestration concept is illustrated in Figure 5.21.

The main element of the metamodel is *Orchestration* which is defined as a kind of *Busi-*

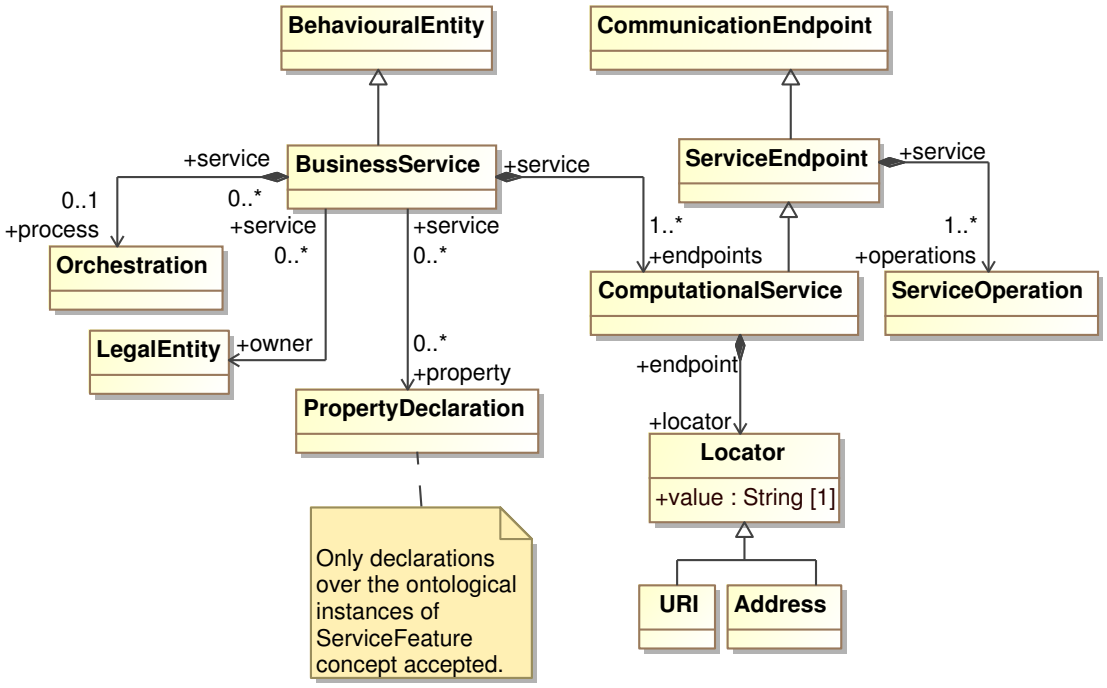

Figure 5.20: An ontology describing the functional entities in service-based communities.

nessProcess that comprises a set of message descriptions and a business process activities. While a *BusinessProcess* is considered as a sequence of business process activities, an orchestration description must not include any such activities, only another business process. The *Orchestration* element serves also a modularization mechanism for business process descriptions.

A *Message* is an ontological instance of the *BusinessDocumentType* concept and represents business data to be communicated. Business data is communicated through a *ServiceOperation* which is associated with an appropriate *Modality*, such as *SendModality* or *ReceiveModality* reflecting the modalities supported by the corresponding kind of communication patterns, for example. Service operations are provided by service endpoints, as discussed in Section 5.3.2. Other kinds of activities used regularly in business process description languages such as WS-BPEL [242] include activities for data processing, for example. Such activities are not considered by the presented top-level characterization of business processes.

The business processes that can be described are classified to two different classes: *1)* process groups, and *2)* conditional processes. Process groups represent process constructors that conjoin several business processes into a more complex business process. A *SequentialProcess* defines an ordered sequence of processes which are executed sequentially: when the execution of one business process ends, the succeeding business process is initiated. A *ParallelProcess* describes a parallel flow of processes: each process instance is executed side by side, possibly synchronizing via business process activities in between. Finally, a *BranchingProcess* represents a choice between multiple execution options. A branching process is provided with a group of processes from which one process is selected for execution based on the enabled activities the processes have, and on the domain specific activity scheduling semantics.

Conditional processes are kind of business processes that are associated with a logical condition and two business processes executed based on the condition. A *ConditionalProcess* is asso-

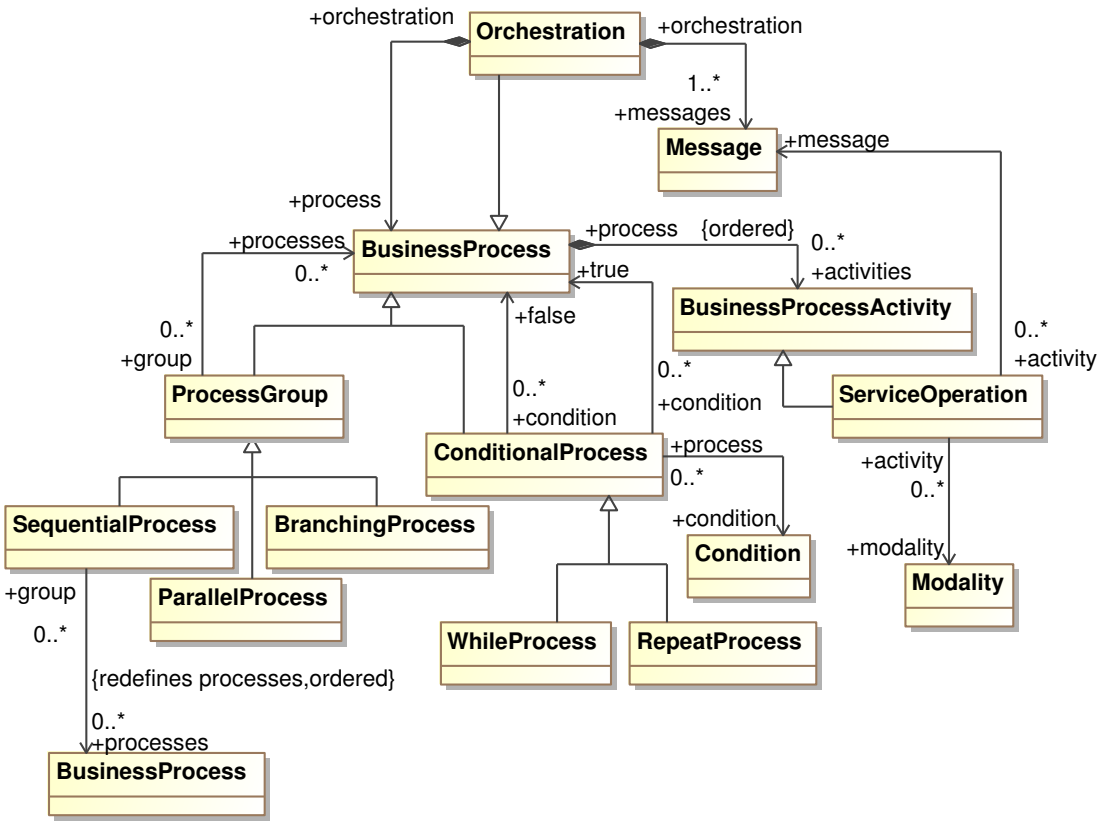

Figure 5.21: Description of the orchestration concept.

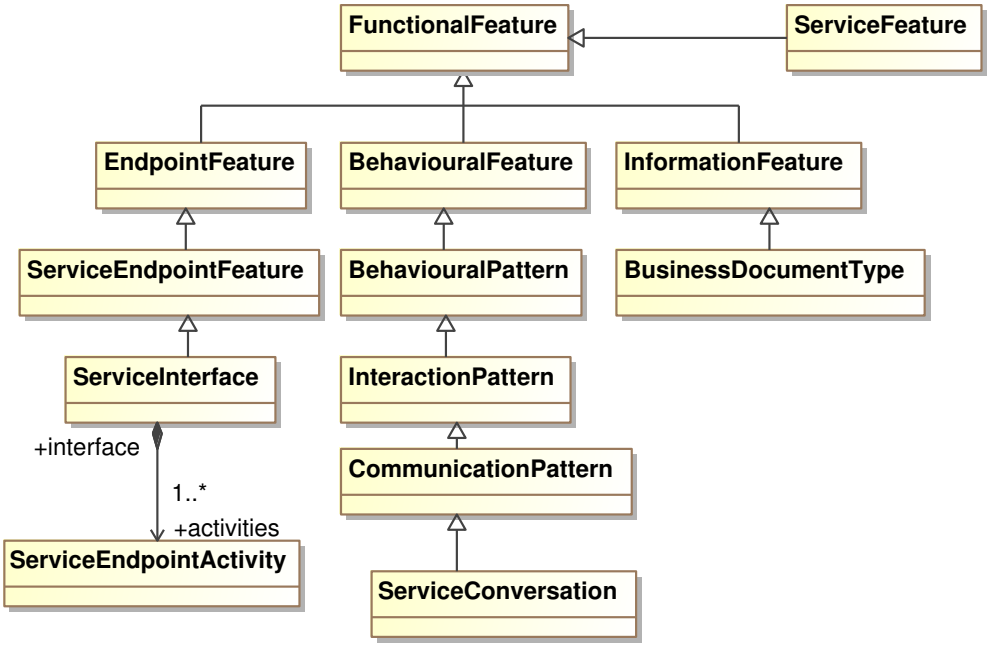

Figure 5.22: An ontology describing the functional features of service-based communities.

ciated with two business processes, a *true* and *false* process. The business process referred to by the *true*-association is executable as long as the logical condition represented by the concept of *Condition* evaluates to true. When the condition evaluates to false, the business processes referred to by the *false*-association is selected for execution. The concept of *ConditionalProcess* is further classified to *WhileProcess* and *RepeatProcess* classes, in which the former represents a process where the execution condition is evaluated before the execution of the *true*-process while in the latter case the condition is evaluated after the execution of the *true*-process.

The metamodel defined for describing the intention of the *Orchestration* concept follows the prevailing properties of process description languages used for both specifying and describing process-like behaviour. Especially, the orchestration metamodel provided is based on the constructs found on process algebras, such as the pi-calculus [166, 222], and WS-BPEL [242], and on the work done for formalizing business process description languages (see for example [155]). The formal semantics of business process orchestrations is not considered further in the context of this thesis.

5.3.3 Service feature ontology

Service concepts defined in the service-based community domain ontology are associated with specific features that are described in the following. On the functional side, the features that are specific for service-based community comprise the notions of *ServiceInterface*, *ServiceConversation* and *BusinessDocumentType*. For service endpoints the most notable feature associated is the concept of *ServiceInterface*, as illustrated in Figure 5.22. A *ServiceInterface* comprises a set of *ServiceEndpointActivity* elements which describes the functional capabilities of corresponding kind of services.

Conceptual services are associated with service conversations which are represented by the

concept of *ServiceConversation* in the ontology. A service conversation is a behavioural feature of a *ConceptualService* describing a communication-based behavioural pattern. A *service conversation* defines bilateral behaviour accepted by certain kind of services and provides a modular and platform independent characterisation of logically self-contained interactions, similarly to [230]. A service conversation is a specialization of the communication pattern concept. Service conversations are kind of communication patterns in which business documents are communicated by service endpoint activities. A business document is an information element with an explicitly defined document structure, or type. The formalization of the service conversation concept is given by Definition 5.3.2.

Definition 5.3.2 (Service conversations) *A **service conversation** is a behavioural pattern (S, T, s_0, ℓ), where the transition labelling function $\ell : T \rightarrow (L \times M \times D)$ labels transitions with a label, a modality and an information element representing the characteristics of the conversation activity such that L is a finite set of labels, M is a finite set of modalities, and D is a finite set of **business document types**.*

Business document types are defined using the *BusinessDocumentType* concept. A *BusinessDocumentType* can be either a basic type, a structured type, or a type reference. The set of basic types include integers and real numbers, strings, booleans, and finally URI:s [23]. This selection of basic types provides a sufficient support for typing the primitive elements found in typical programming and modeling languages. The inclusion of URI's at the set of basic types is motivated by the importance of Internet and Web Services in the context of modern enterprise computing.

Business document types represent information elements that have structural features similar to the XML-Schema [263], the most widely known and adopted metalanguage for describing the structure of XML documents. Business document types are represented formally as labeled and nested lists, similarly to [105, 106]. A countably infinite sets of labels, ranged over by $l, l', \ldots$ and variables, ranged over by $U, U', \ldots$, are assumed. Labels represent the element names utilized in XML-Schema definitions to describe document structures whereas the variables allow for restricted modularization of schema definitions as well as mutual recursion between business document typing structures. A set of basic types bt, such as strings and integers are assumed. In addition, a business document naming environment N has to be in place, such that for each variable U encountered there exists a mapping $N(U)$ from the variable name to the corresponding schema definition of form $type\ U = D$. The syntax for business document types is defined by the grammar given in Table 5.1.

D	::=	**(types)**		bt	::=	**(basic types)**	
		$()$	(empty sequence)			$string$	(string)
		$l[D]$	(label)			int	(integer)
		D, D	(sequence)			$real$	(real)
		$D + D$	(union)			$boolean$	(boolean)
		U	(variable)			uri	(uri)
		bt	(basic type)				

Table 5.1: Syntax for business document types.

The syntax given for business document types in Table 5.1 defines a *context free language* [104]. For practical purposes some additional constraints over the syntax have to be made. Especially, only tail recursion is allowed to keep the corresponding business document types correspond to so called regular tree languages [106], for which the language inclusion problem is decidable [104].

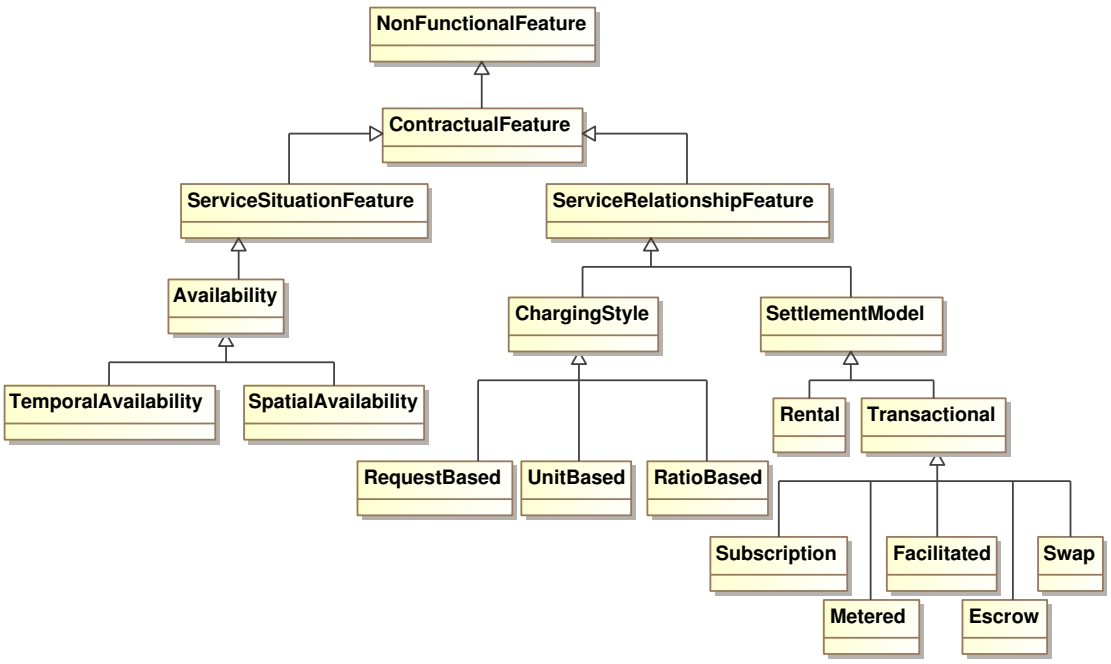

Figure 5.23: Non-functional features of service-based communities.

If unrestricted recursion was allowed, the corresponding language would be a so called context free language for which language inclusion problem is undecidable [104]. This language inclusion property is needed for enabling business document subtyping, which will be discussed in Chapter 6.

Service-based communities introduce two specific kinds of contractual features to the top-level ontology of non-functional features. These features are associated with the service situation and service relationship concepts introduced in Section 5.3.4. The service situation features describe the kinds of properties that can be placed for business services in different usage situations. Features associated with service relationships describe mutual obligations and agreements settled between the legal entities participating in a service relationship, for example. The corresponding classification is presented in Figure 5.23.

Availability is a feature which refers to either temporal or spatial constraints [187] on the usage of a business service in a certain service situation. Usage of a service can be temporally constrained to service only on business days, for example. Spatially service usage can be constrained with respect to geographic properties, for example. Concepts of *ChargingStyle* and *SettlementModel* are associated with a service relationships and they represent the contractual obligations related to the grounds of payment for service usage.

The charging styles are classified to three different styles, following [187]: *1)* request-based payment, in which client is charged per service request or delivery, *2)* unit-based payment, in which client pays by unit of measure or granularity, and *3)* ratio-based payment, where client is charged on a percentage or ratio basis of some aspect of the service. Settlement models reflect the ordering and relationship of payment and service delivery obligations between the legal entities in a service relationship. The *TransactionalModel* represents the most typical "delivery for payment"-scenario, including both one-off delivery or multiple deliveries within a long-term service relationship. The transactional settlement can be further classified to subscription, metered,

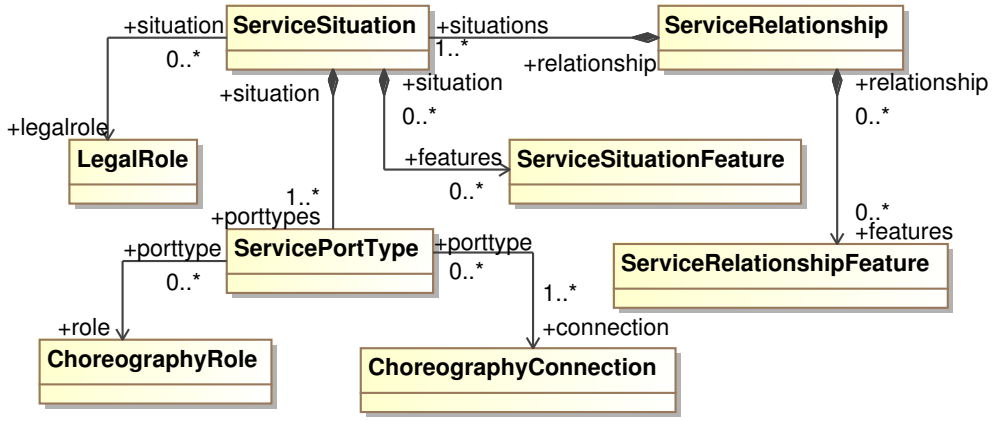

Figure 5.24: A metamodel describing the notion of service relationships.

facilitated, escrow, and swap based models [187]. The *RentalModel* represents a kind of service relationship settlement where the corresponding service is being "loaned" within a short-term or long-term relationship [187]. Typically temporal constraints are associated with the rental model.

5.3.4 Service grounding ontology

The service grounding ontology provides concepts for relating the contractual and technical aspects of service-based cooperation. First of all, concepts are provided for associating communication relationships with appropriate contracting relationships. Secondly, at the instance level concepts are provided for enterprises to offer their business services for use in specific service usage contexts. Finally, concepts are provided for associating mutual service agreements with technological artifacts facilitating the communication activities between corresponding service endpoints.

A service relationship is considered in the context of service-based communities as a contractually regulated, bilateral inter-relationship between legal entities which is realized by communicative behaviour. Service relationship provides the means for negotiating bilateral service-level agreements, for example. The concept of *ServiceRelationship* formalizes this relationship as defined by the metamodel illustrated in Figure 5.24.

A *ServiceRelationship* comprises a set of service relationship features and two *service situations*. A *ServiceSituation* describes a position or an end point of a bilateral service relationship by declaring its features, and legal and communicative obligations.

The concept of *ServiceSituation* is associated with a legal role to specify the contractual position in the service relationship and so-called *service port type*. The concept of *ServicePortType* declares a selection of choreography connections and a role shared by all the connections as a choreography-specific unit of service relationships. The notion of a port type is illustrated in Figure 5.25 where two service choreography roles named simply as *"Role A"* and *"Role B"* are provided. The roles are inter-related by three connections. Both of the roles might be further connected with other roles, but these connections or roles are not shown in the figure. Now, a service port type is provided which defines that one of the port types declared for the cooperation scenario comprises of the two upper connection ends at the *"Role A"*-side of the connections.

The corresponding service port type is illustrated in Figure 5.25 by the white rectangle with la-

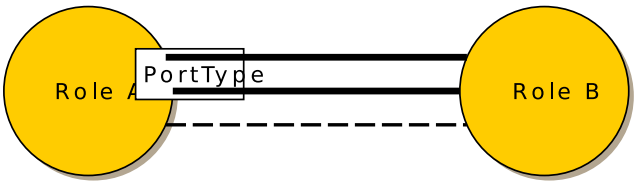

Figure 5.25: Illustrating the notion of a service port type.

bel *PortType*. Similarily, the same set of connections must be provided with a port type associated with the *"Role B"*-side of the connections. The connection that is left out of in the example must also be provided with corresponding port type declarations if it is needed in the service relationship in question. Typically all connections relating a pair of roles will be provided with corresponding service port types representing the corresponding sides the the connections. However, sometimes it can be attainable to provide a grouping of connections such that different connections can be associated with different kinds of obligations, for example.

The legal and choreography roles referenced from a service relationship must be dual in a sense that they are related by the same cooperation connection, correspondingly. That is, two roles residing in separate service contract or service choreography relationships can not be bound by the *ServiceRelationship* concept. Service relationships and service situations are associated with corresponding kinds of non-functional features, as discussed in Section 5.3.3.

The ontological instance of a service relationship is called a *service agreement*. A service agreement declares a set of service relationship properties and two service provision commitments, one for each partner needed for realizing a service relationship. The property declaration associated with a *ServiceAgreement* must be defined over properties that are ontological instances of the *ServiceRelationshipFeature* concept, as illustrated in Figure 5.26.

A *service provision commitment* comprises properties and bindings between the legal and choreography roles of the service relationship, and the legal entity taking part in the relationship and his provided computational services. The property declaration for *ServiceProvisionCommitment* is valid only over concepts that are ontological instances of the *ServiceSituationFeature* concept.

For establishing the communication relationship required for realizing a service choreography, the concept of *service connector* is used. A service connector represents a binding between the conceptual service relationships and the technological communication facilities. A connector type illustrated in Figure 5.27 is a concept for declaring different kinds of connectors. A *ConnectorType* is composed of two connector port types that represent the different sides of a service relationship. Each *ConnectorPortType* is associated with a service port type declaring the service relationship context and a set of channel bindings. The concept of *ChannelBindingType* is used for associating a channel port type with the corresponding service choreography connection. As a result of this construction, each end of a choreography connection is associated with a channel.

The concept of a connector represents the ontological instance of a connector type. Figure 5.28 illustrates the concept of *Connector* which is simply a composition of the ontological instances of the corresponding connector type elements.

Finally, the service connector concepts which are specific for a certain service relationship and communication channel usage are related with service entities. These relationships are established by service usage concepts which describe how individual services (e.g. conceptual services, busi-

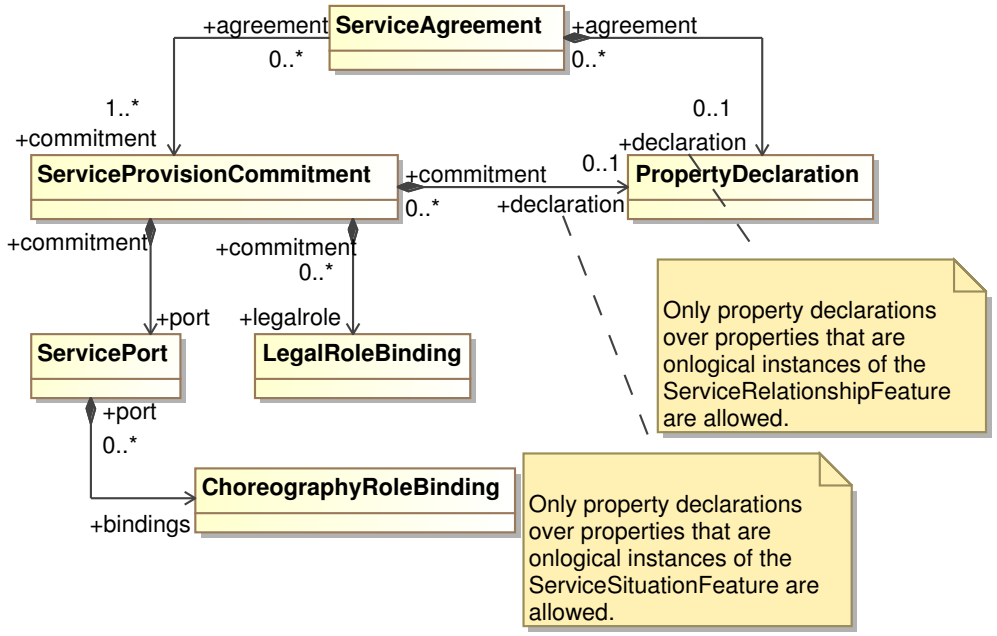

Figure 5.26: A metamodel describing the notion of service agreement.

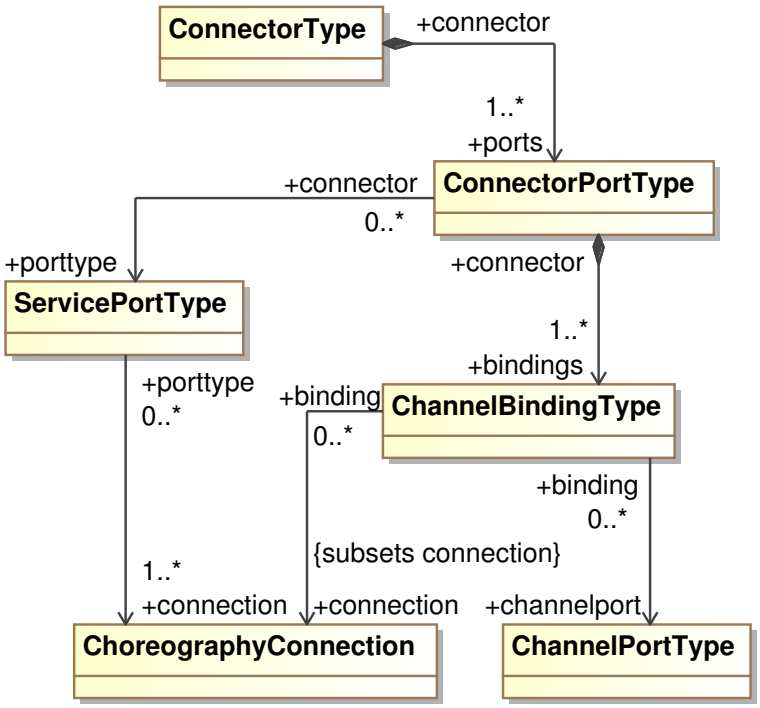

Figure 5.27: A metamodel describing the notion of connector types.

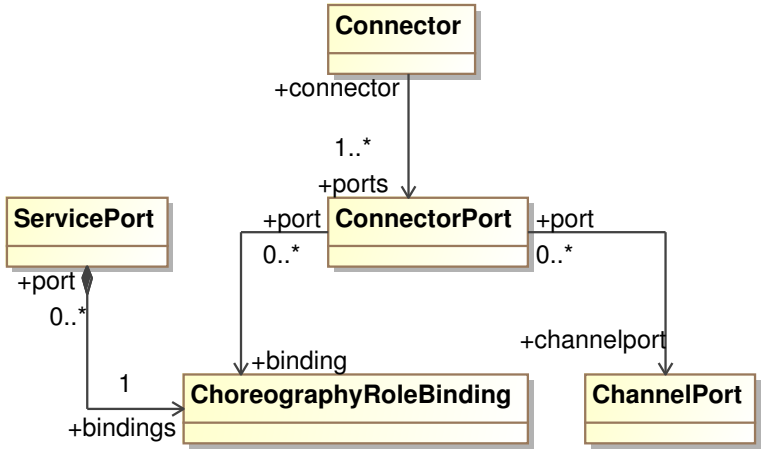

Figure 5.28: A metamodel describing for service connectors.

ness services and computational services) are related to a specific service relationship and technological grounding. The concept of *service context* defines a binding between a service connector and conceptual services. The binding is provided by associating each channel binding type defined in a service connector with a service conversation of a conceptual service, as illustrated in Figure 5.29.

Effectively, the concept of service context defines a service composition in which service conversations provided by a set of conceptual services are used for providing the behaviour to be taken over a choreography connection. Accordingly, consistency of a service context definition needs to be validated. Especially, the behaviour prescribed by service conversations need to be matched by the behaviour of corresponding choreography role. For this purpose, a correspondence needs to be established by the endpoint activities used in service conversations and cooperation activities used in choreography event structures. Such a mapping is domain specific and can be defined once for each domain. In such context, the service conversations can be considered as behavioural types used for specifying the behaviour that can be provided over the corresponding choreography connections. Behavioural types in general have been used to reason about objects which possess non-uniform behaviour [173].

The ontological instance of a service context is known in the service-based community domain ontology as a *service liaisons*. Service liaisons are used by service providers to advertise their business services. More specifically, service liaisons declare the service provision commitments and connector ports established by a service provider to publish her business services to the clientele. The intention of the service liaison concept is given by the metamodel illustrated in Figure 5.30.

The *ServiceLiaison* concept is associated with a business service provided by a legal entity. The computational services included in the business services are used within the *ServiceProvisionCommitment* elements to declare service ports that match the prerequisites of a specific service relationship. Finally, *ConnectorPort* elements are provided by the service liaison, which associate channel ports needed for establishing communication with the service.

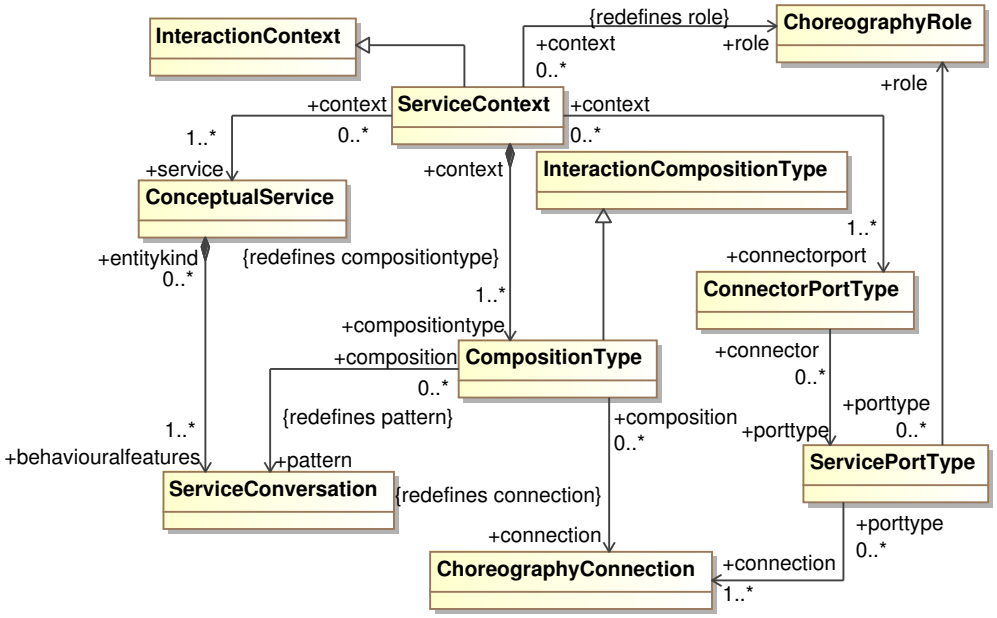

Figure 5.29: A metamodel describing service contexts.

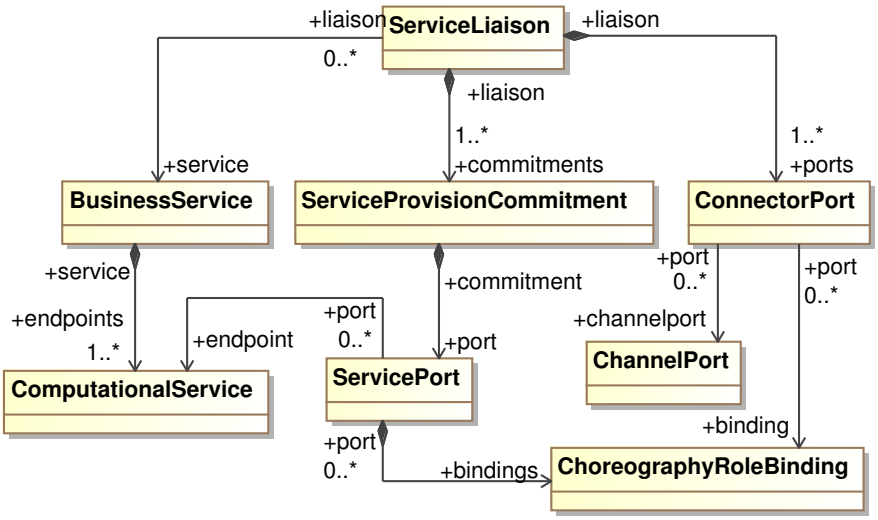

Figure 5.30: A metamodel describing service liaisons.

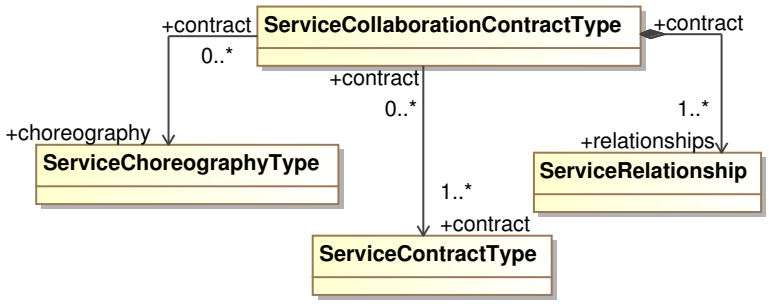

Figure 5.31: A metamodel describing service collaboration contract types.

5.3.5 Service collaboration contract ontology

Collaboration in service-based communities is regulated by explicit collaboration contracts. Collaboration contracts relate service choreographies with contractual relationships and technological elements that manifest the commitments and infrastructure facilities needed for instrumenting corresponding kinds of collaborations. Collaboration contracts are defined and maintained via the two-level ontological metamodelling approach familiar from previous sections, that is collaboration kinds are prescribed at the type level and they are instantiated to actual contracts at the instance level.

Collaboration contract types are defined by a metamodel that is illustrated in Figure 5.31. A *ServiceCollaborationContractType* is comprised of references a choreography type, and a set of service contract types. The legal roles prescribed in service contract types are bound to service relationships that are declared by the *ServiceCollaborationContractType*. Consequently, a service collaboration contract type prescribes what kinds of legal roles are associated with which kind of service choreography connections in the corresponding kinds of collaborations.

Collaboration contract instances are defined by a metamodel illustrated in Figure 5.32. A *ServiceCollaborationContract* comprises two or more service liaisons, a set of service agreements and a set of connectors. The set of service liaisons represent the individual commitments made by service providers. A service collaboration contract then defines a set of service agreements agreements describing service relationship incarnations. Finally, a service collaboration contract defines a set of service connectors which relate the individual connector port elements provided in the service liaisons. In Figure 5.32 some elements of previously described concepts, such as service liaison or connector, are unfolded to show the inter-relationships that underlie the service contract concept.

It should be noted that the concepts of *ServiceCollaborationContractType* and *ServiceCollaborationContract* are not symmetric, that is, the instance level incarnation of the collaboration contract concept is not a direct composition of the instances of the corresponding contract type elements. While *ServiceLiason* is an ontological instance of *ServiceContractType* the other two elements of service collaboration contract, *ServiceLiaison* and *Connector* are not ontological instances of either *ServiceRelationship* or *ServiceChoreographyType* found in the corresponding collaboration contract type.

There are several consequence from the formulation of the collaboration contracting concepts introduced above. First of all, the collaboration contract types, as well as service choreography and service contract types, are independent from the kinds of communication facilities and services needed for realizing such contracts. This sets the scene for reusable, platform independent

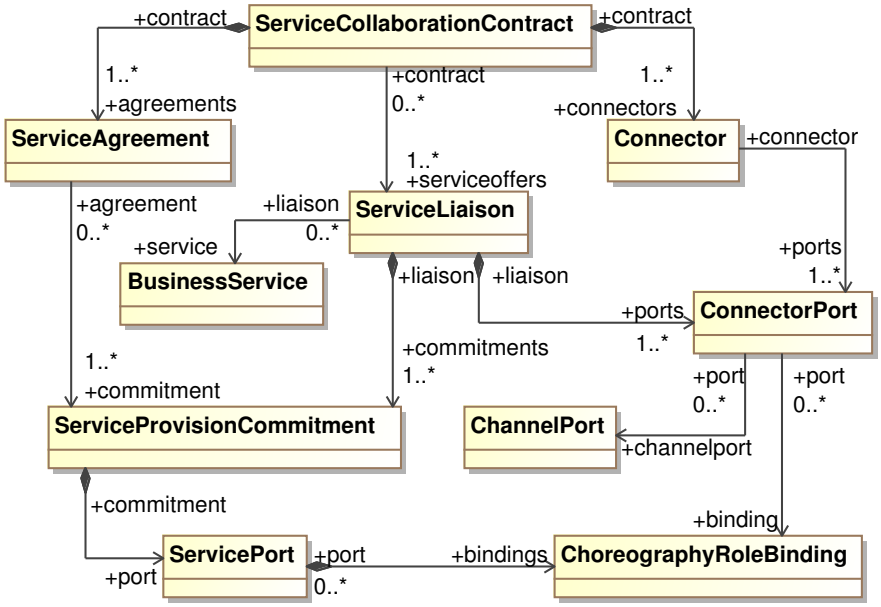

Figure 5.32: A metamodel describing service collaboration contracts.

collaboration contract types that can be designed and validated separately from the technological artifacts. Most important kinds of collaboration contracts can be even standardized, for example.

Secondly, the properties laid for collaborations in contract types propagate naturally through the abstractions of service agreements and connectors to individual business services, their providers and communication facilities. This means that the mechanisms for propagating the more abstract features all the way to technical properties need to be in place. Finally, the ontological consistency between the collaboration contract and its instances need to be rigorously defined, since a service collaboration contract is very loosely coupled knowledge artifact that includes elements from individual service providers as well as from domain specific shared cooperation practices.

Chapter 6

Towards a service-ecosystem for federated service communities

Federated service community is an approach of collaborative computing adopted in the Pilarcos framework where business services are developed independently, and the provided B2B middleware services are used to ensure that technical, semantic, and pragmatic interoperability is maintained during collaboration establishment and operation [141, 139]. For addressing the complexity of interoperability knowledge management, establishing loosely coupled business service collaborations, and instrumenting corresponding service ecosystems with utility services and engineering tools, a formalization of the federated service communities is needed. For this purpose, a reference model for federated service communities formalizing the concepts of the Pilarcos framework is defined in this chapter.

The reference model for federated service communities comprises a domain ontology and a knowledge management metamodel reflecting the concepts of the ontology. The domain ontology for federated service communities is largely based on the service-based community domain ontology defined in Chapter 5. There are however some concepts utilized by the Pilarcos framework; these concepts are formalized by the domain ontology for federated service communities described in Section 6.1. A notion of service types is used for managing business service interoperability; this concept is formalized in Section 6.2. The corresponding knowledge management metamodel for federated service communities is introduced in Section 6.3. We conclude with Section 6.4 which provides a discussion about the facilities required for instrumenting service-oriented software engineering within this framework.

6.1 Domain ontology for federated service communities

The domain ontology for federated service communities is largely based on the concepts provided already by the service-based community ontologies. There are however some additional concepts that need to be introduced to realize the Pilarcos approach to collaborative computing. These new concepts include concepts of eContracts and epochs for realizing eContracting processes and evolvable business networks as well as a specialization of the conceptual service that enables interoperability validation. The Pilarcos framework and its foundational concepts are first introduced in Section 6.1.1 and formalized by corresponding metamodels defined in Section 6.1.2.

6.1.1 Pilarcos framework

The Pilarcos project concerns development of service-oriented middleware infrastructure for open inter-enterprise computing environments [141, 139, 140, 142]. The Pilarcos framework proposes a federated model of inter-enterprise collaboration networks, or virtual enterprises, comprised of autonomic business services whose collaboration is regulated by electronic contracts. Metainformation describing the properties of business services and networks, and infrastructure services providing a community breeding and management environment are utilised for establishing the virtual enterprises. The collaboration constellations comprising of autonomous business services are called in the Pilarcos framework as *eCommunities* and they are established dynamically to serve a certain business scenario or opportunity.

The operation of an eCommunity is governed by an electronic contract, or *eContract*, which is negotiated dynamically by the participants [165]. The eContract is structured according to a *business network model* which explicates the roles of partners and the interactions between roles that are needed for reaching the objective of the eCommunity. Each eContract is further structured by *epochs*, periods of activity where the jointly provided service and the structure of the eCommunity is stable [139]. Separate epochs can be used for breach recovery or otherwise well-limited activity with different set of roles still progressing the work of the eCommunity and each epoch constitutes a collaboration itself. In the context of virtual enterprises the epochs may consist of virtual enterprise formation, operation and dissolution [175], for example.

A business service in the Pilarcos framework is comprised of a computational service, a monitor, and a Business Network Agent as illustrated in the Figure 6.1. Business context awareness is provided during the operation of a business service in the corresponding Business Network Agent (NMA) which utilises the monitor to observe and control the operation of the computational service. Local contract and policy repositories are used to store information concerning the contract information, business rules and policies effective within the virtual enterprise and the service provider organization itself.

Preliminary blanket agreements or initial trust relationships might be required before these electronic contracts can be formed. Given the necessary prerequisites, an eCommunity is established by utilising service trading and community population mechanisms provided by the Pilarcos eCommunity breeding environment, and multilateral negotiations about the properties of the business network [141]. The eCommunity breeding environment consists of public meta-information repositories and populator services, as illustrated in Figure 6.1. Service offer and type repositories provide a service trading mechanism, similarly to the ODP trading and type repository functions [111, 108]. Type repositories are persistent storages of service type information which are used as the primary means for achieving interoperation between business services [217]. Business network model repositories are used to publish and discover descriptions of eCommunity configurations, or Business Network Models (BNM) [141].

Given a business network model, the task of the breeding environment is to provide a set of potential eCommunity contracts templates to be negotiated further by the participants. The eCommunity breeding environment utilises the meta-information services to accomplish this task. There are two phases in the breeding process: population phase performed by a populator service and sub-sequential negotiations performed by the NMAs of the participants. Breeding environment services, such as populators and type repositories, are not required from all sites, but can be provided as infrastructure services as a business on its own right [141].

The populator represents a breeding process phase where appropriate business service providers are selected for eCommunity roles. The populator function takes a business network model and utilises service type and service offer repositories for fetching compatible business service

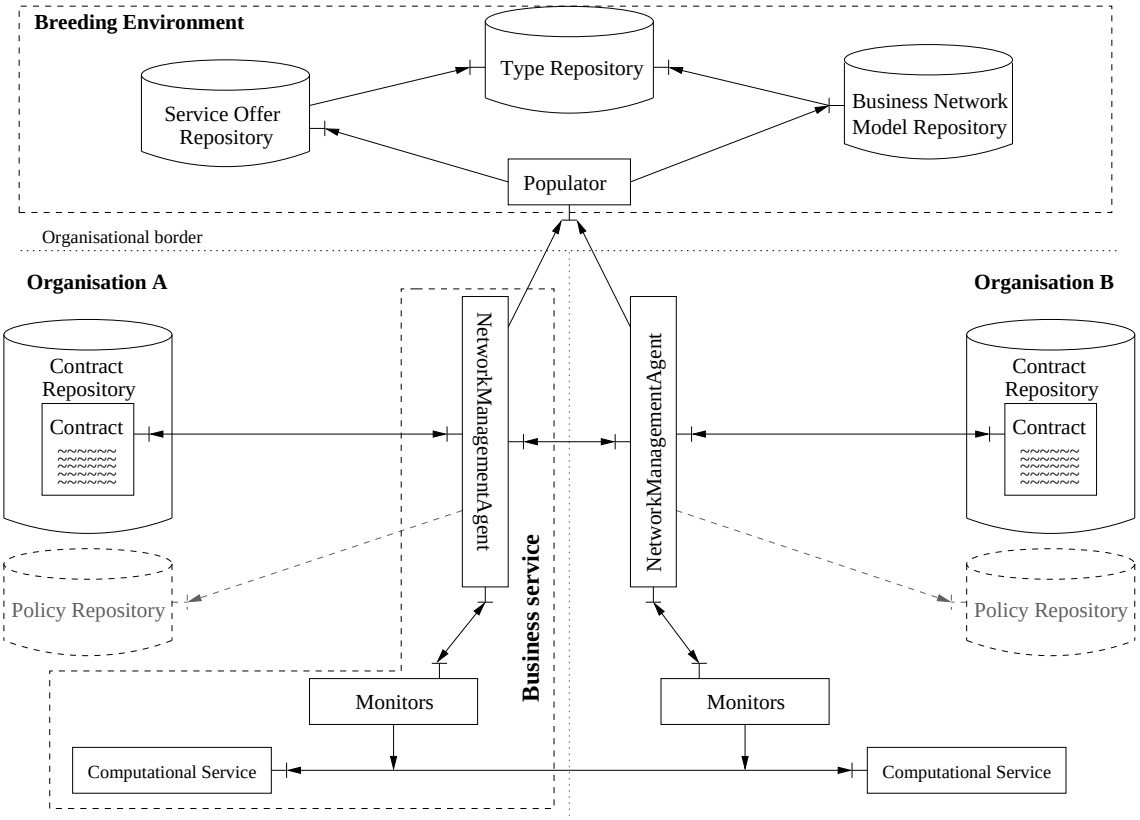

Figure 6.1: An overview of the Pilarcos architecture. Arrows represent communication relationships, boxes are active agents and cylinders are information repositories. [215]

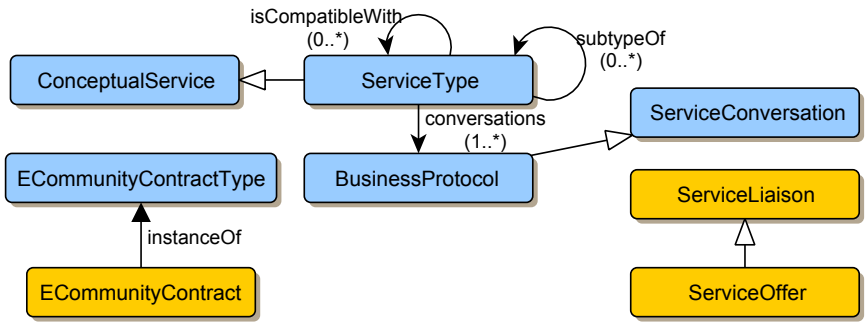

Figure 6.2: New concepts introduced by the federated service community domain ontology.

providers for each eCommunity role. The populator selects the business services to an eCommunity on a basis of a constraint satisfaction process which considers the compatibility of the business service attributes [139]. When a set of compatible service offers fulfilling the requirements of the business network roles have been found, the populator returns the description of the populated eCommunity to the initiator of the population process. Population processes are initiated by enterprises willing to establish business collaborations.

After a successful population process the corresponding eCommunity description is distributed to the participants [139]. Negotiations are held between the participants to decide about the final properties of the collaboration. For the negotiation phase, the web-Pilarcos framework provides generic negotiation interfaces and meta-level protocols. The negotiations and eCommunity management during the operation of the community are handled by the NMAs [141, 165]. The NMAs provide uniform interfaces and they act as the representatives for the autonomous business services during the breeding process and operation of collaboration networks. The collaboration management interfaces of NMAs provide functionality for example for renegotiating part of the collaboration contract, to query the status of the contract, and to control transitions between eCommunity epochs [165].

6.1.2 Concepts for describing eCommunities

The concepts specific for federated service communities include eCommunity contracts, service offers and service types. These concepts facilitating eCommunity negotiations and contract-based collaborations are illustrated in Figure 6.2. Service liaisons in the domain ontology are called as *service offers*. In the current version of the metamodel the concept of service offer does not introduce any new features with respect to service liaison but serves as a placeholder for future extensions and aligns the vocabulary with that of the Pilarcos framework.

Cooperation in eCommunities is based on eContracts declaring the life-cycle of the community and structure of its epochs. In the federated service communities domain ontology, eContracts are is defined using the concept of *ECommunityContractType* whose intention is defined by the metamodel illustrated in Figure 6.3. An *ECommunityContractType* comprises of a set of service collaboration contracts and an eCommunity life-cycle declaration. Service collaborations contracts are represented by the concept of *ServiceCollaborationContractType* defined in Section 5.3.5.

Life-cycle declaration is optional for eContracts involving only a single service collaboration contract. For communities requiring a series of epochs to attain the collaboration goals, a life-cycle declaration is obligatory. An *ECommunityLifecycle* comprises a series of transitions between the

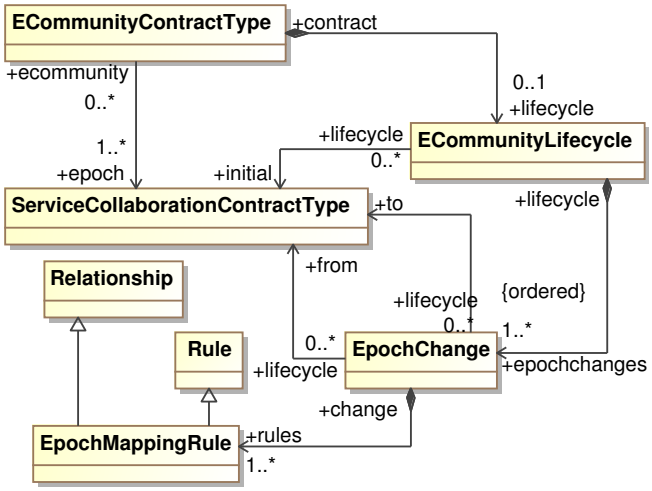

Figure 6.3: A metamodel describing the concept of eCommunity contract type.

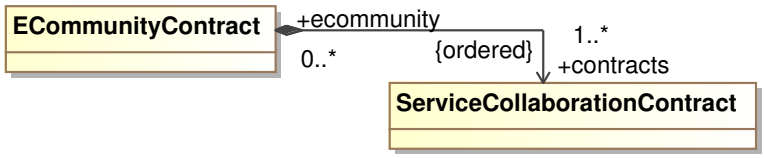

Figure 6.4: A metamodel describing the concept of eCommunity contract.

epochs of the corresponding eContract, effectively defining a process for changing epochs within an eCommunity contract type. The initial epoch of the eCommunity life-cycle is identified by the *initial*-association for clarity of presentation. Epoch transitions are represented by the concept of *EpochChange*, each declaring the source (*from*-association) and destination (*to*-association) epochs, and corresponding epoch mapping rules.

The epoch mapping rules represented by the *EpochMappingRule* concept prescribe how properties from one epoch are propagated to the other. An epoch mapping rule may declare that a participant in one role should also play some other role in another epoch, for example. Correspondingly, an epoch mapping rule is also a specialization of the *Relationship* concept. In general, the epoch mapping rules do not have to refer to subsequent epochs in the eCommunity life-cycle, but elements from any previous epoch can be mapped to a subsequent one. Epoch mapping rules could in practise be modelled using model weaving constructs [64], for example. The classification and semantics of epoch mapping rules is not discussed further in the context of this thesis.

The ontological instance of a *ECommunityContractType* is illustrated in Figure 6.4. An *ECommunityContract* is defined as a sequence of *ServiceCollaborationContract* elements. The ordering of the sequence and individual service collaboration contracts must conform with the eCommunity life-cycle defined in the corresponding eCommunity contract type.

6.2 Service types for business service interoperability

The notion of conceptual service is specialized by the domain ontology of federated service communities; the corresponding concept is called a *service type*. The notion of service type is provided with a behavioural type system based on the notion of session types [103, 90, 255] that provides a syntax-driven characterization for behavioural compatibility and refinement. More over, service types are provided with a recursion operator allowing us to express repetitive behaviour.

The typing discipline attached with service types is based on structural and behavioural typing whose fundamentals are briefly introduced in Section 6.2.1. After that, the concept of service type is formalized in Section 6.2.2. Finally, the use of services types for guaranteeing business service interoperability is defined in Section 6.2.3.

6.2.1 Structural and behavioural typing

The service typing discipline is based on formal theories such as type systems for structured document descriptions [105, 106] and behavioural typing of processes [241, 103, 90, 255]. To make the definition of the service typing discipline tractable and unambiguous, some underlying theoretical frameworks need to be introduced. This background knowledge is briefly discussed in the following.

Types are used in programming and modeling languages for denoting a set of objects that share a common structure or shape, or a set of more generic properties. Dually, a type prescribes conformance criteria for an object to be classified into a certain category in the universe of discourse. More over, type systems that define a set of typing rules and corresponding type checking procedures are used for preventing certain kinds of misbehaviour from happening during the operation of the corresponding system. In strongly typed programming languages the type system prevents assignment of invalid values into variables during the operation of the program, for example. Type systems are also usable for enforcing disciplined programming and modeling practices, documentation, and for ensuring language safety [197]. A well defined type system can be utilised as a static proof method to analyse system behaviour and to prevent incorrect usage of programming and modeling language constructs.

For the purpose of introducing type systems, a restricted version of simply typed lambda-calculus (see for example [197]) is adopted in the following. A set of (finite) types T includes functions ($T \rightarrow T$), binary products ($T \times T$) and disjoint binary unions ($T + T$). Additionally, two primitive types, namely natural numbers (Nat) and integers (Int) are included in the example type system. The grammar for the types is then given as follows:

$$T ::= T \rightarrow T \mid T \times T \mid T + T \mid Nat \mid Int$$

A type system (or type theory) is defined with a set of rules that prescribe how types are attached to terms, and in the presence of subtyping, how types are related to each other by the subtype-relationship. The rules are represented usually as inference rules, each comprising a set of premises and a conclusion. A typical inference rule found in almost every type system and logical system is the rule of assumption which states that if something is assumed then we can infer that assumption. Formally this is represented as an inference rule such as $T - ASSUMP$ described below:

$$\text{T-ASSUMP}\ \frac{x : T \in \Sigma}{\Sigma \vdash x : T}$$

The rule named $T - ASSUMP$ describes the interaction between the assumptions given in a *typing environment* Σ and type inference. Here, x denotes some term or object in the universe of discourse, the notation $x : T$ means that x is of type T. Typing environment Σ is a set of *type bindings* of form $x : T$. Now, the rule $T - ASSUMP$ says that if the type binding $x : T$ is included in the typing environment Σ, then it can be inferred that x has the type T under the typing environment Σ. The inference rules can be considered as an recipe for a type checking procedures: when read in a bottom-up manner, a typing rule gives an algorithm for checking if a term has the corresponding type. Correspondingly, in the case of the rule $T - ASSUMP$, validating that $\Sigma \vdash x : T$ involves checking if the typing environment includes the type binding $x : T$. In the context of this thesis, typing rules defining how to bind types to terms are not used since business service interoperability is addressed at the service type level using so-called *subtyping rules*.

Subtyping is an inclusion relation between types which relates two type descriptions to each other in a well-defined way. This "well-definedness" is a property of the type system and is implied by the design choices made during the development of the programming or modeling language. Subtyping is a flexible mechanism for sharing interface structures and behaviours that is typically expected to provide sufficient conditions for substitutability. This is known as the Liskov's substitutability principle: any property proved about super-type instances also holds for its subtype instances [151].

There are different kinds of subtyping mechanisms, or criteria, that have been developed for different purposes. The two of the most well known subtyping mechanisms are nominal and structural subtyping. Nominal subtyping is predominant in the conventional object-oriented programming languages utilised in the industry, such as Java [9], C++ [237] or C♯ [57]. In nominal subtyping the programmers and designers construct explicit subtyping hierarchies by annotating type declarations with subtyping relationships. In the Java programming language a class declaration *public class Foo implements Bar* defines that the class *Foo* is by definition a subtype of the predetermined class *Bar*, for example.

In structural subtyping the hierarchies are implicit as they are induced by the structural properties of type and class definitions. For example the *OCaml* functional programming language uses structural subtyping [80, 148]. Structural subtyping is harder to implement because the algorithms needed for subtyping validation or inference are complex. Structural subtyping however provides much more flexibility that nominal subtyping.

Also hybrids between the nominal and structural subtyping disciplines have been developed for addressing some particular, usually pragmatic need. In [80] so called semantic casts are used to introduce flexibility of structural subtyping into conventional languages with nominal subtyping hierarchies. Especially, the XML-Schema language [263] and the corresponding type systems (e.g. [264, 106]) utilize both nominal and structural subtyping mechanisms.

Finally, behavioural typing can be thought as providing an abstraction of the environment some process is executing on. In this context, typing of a process describes assumptions about environmental properties, such as communication channel behaviour. Behavioural typing systems are based on feasible behavioural abstractions of the underlying, more complex behavioural artifacts such as processes, and formal rules providing relationships for providing correspondences between the behaviour of abstractions and the behaviour of the processes.

In general, behavioural types are used to reason about objects which possess non-uniform behaviour [173]. Different kinds of behavioural typing rules have been used to describe or prove several kinds of properties of processes, such as deadlock freedom, absence of service denial and correct usage of communication ports, for example [129, 173, 198, 130]. Behavioural type checking can be used as a substitutive or complementary method for other formal methods, such as model checking [51, 172].

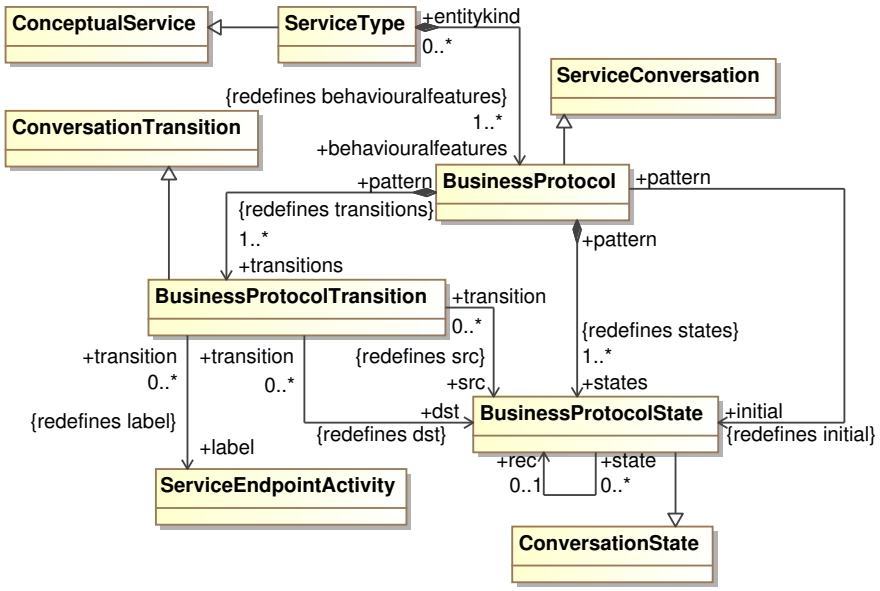

Figure 6.5: A metamodel describing the concept of service type.

6.2.2 Enabling service typing

In the domain ontology for federated service communities a *service type* defines the characteristic features for a kind of business services. Service type is a specialization of the *ConceptualService* concept introduced in the service-based community domain ontology, as illustrated Figure 6.5. Structurally the *ServiceType* concept follows the intention of conceptual service very closely (see Section 5.3.2). The most important difference is the introduction of *BusinessProtocol* concept that specializes service conversations. The states in a business protocols are provided with an optional *rec* association providing means for expressing repetitive behaviour. Especially, the concept of *BusinessProtocol* follows the characterization of session types defined in [241, 103, 90, 255].

Service types are used to constrain the behaviour of business services, to validate consistency of cooperation abstractions, and for verifying behavioural substitutability and compatibility between kinds of services. Service types are considered as behavioural types in the context of federated service community reference model. Consequently, the ontological conformance relationship between a business service and the corresponding service type becomes a behavioural typing relationship (see for example [173]). However, at this point, actual behavioural typing of processes is left out of discussion and in the following we concentrate on formalizing the behavioural compatibility and refinement relationships between service types.

As service types represent behavioural entity kinds, the semantics to be attached with the *subtypeOf* - and *compatibleWith*-associations of the *ServiceType* concept (see Figure 6.2) are supposed to induce a behavioural relationships between service kinds. Substitutability properties between service types types are particularly important in this context: a behavioural subtype should respect same constraints and have same possible executions that of its super-type [173]. Thus, a service type should always be usable in same cooperation contexts as its super-type. More pragmatically, behavioural subtyping hierarchy between service types can be used as a service classification criteria and a basis for service taxonomy for enabling more responsive service discovery, for example.

Service type concept introduces refinements over the concepts of service conversations and

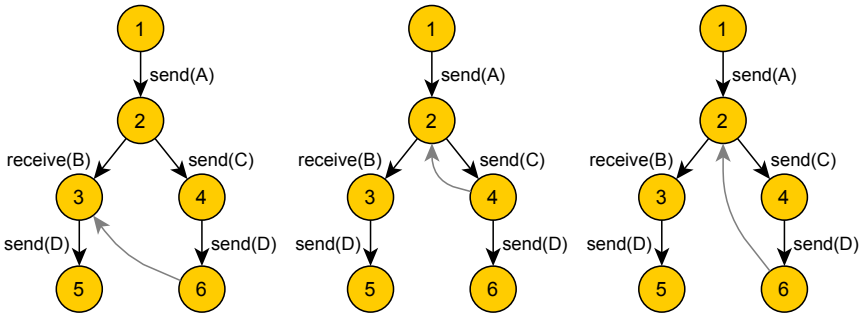

Figure 6.6: Illustrating the notion of recursion in business protocols.

business document types used for expressing conceptual services. First of all, regular (i.e. tail) recursion is used in the business protocols attached for expressing repetitive behaviour. Correspondingly, the interaction pattern induced by a finitely expressed business protocol is potentially infinite.

The notion of recursion in business protocols is illustrated in Figure 6.6. The business protocols describe communicative behaviour where documents of type *A*, *B*, *C* and *D* and communicated with *send* and *receive* activities. Recursion is illustrated with gray, unlabelled arrows between the enumerated states. The left-most business protocol is invalid, since the recursion is not regular as the destination (state 3) of the recursion arrow is not in the same backward path (6,4,2,1) as its source state (state 6). The business protocol in the middle is also invalid, since the source state (state 4) of the recursion transition comprises of another transition from state 4 to state 6, thus violating the "tail-positioning" of regular recursion. Only the right-most business protocol is valid and describes repetitive behaviour using regular recursion.

Secondly, constraints are set over the syntax of business protocols and document types for attaining a service typing discipline which is computationally attractable. The constraints laid over the syntaxes restrict the corresponding formal structures to regular and deterministic forms. Service behaviour is constrained by the federated service community domain ontology with respect to the branching constructs that can be expressed with business protocols. While a service conversation state in the service-based community ontology can be provided with any kind of outgoing transitions, in the federated service communities business protocol transitions sharing a source state must also share the same modality. Such unimodality of branching constructs is needed for attaining feasible forms of behavioural compatibility and refinement relationships. With this respect, all the business protocols illustrated in Figure 6.6 are invalid, since state 2 has two transitions, $(2, receive(B), 3)$ and $(2, send(C), 4)$, that have different modalities.

The syntactic restrictions described above can be formalized by representing an abstract syntax for business protocols that complies with the constraints. The corresponding abstract syntax is defined in Table 6.1 and comprises of constructs for defining activity sequencing, choices, recursion, and empty behaviour.

Business protocol activities are represented in the syntax as $m(D)$ where $m \in M$ is a modality from the set of modalities M, and D is a business document type following the abstract syntax defined in Table 5.1. Activity sequencing is used for defining behaviour comprising of several consecutive activities. A business protocol $send(A).receive(B)$ represents behaviour where first a business document of type A is sent and then business document of type B is received. It is easy to see, that business protocols conforming to the abstract syntax defined in Table 6.1 are also valid

B	::=	**(business protocol)**		
		$m(D).B, m \in M$	(activity sequencing)	$\Sigma_i^m D_i : B_i, m \in M$ (choice)
		$\mu t.B$	(recursive definition)	t (recursion variable)
		$\perp$	(empty behaviour)	

Table 6.1: Abstract syntax for business protocols in federated service communities.

interaction patterns.

Unimodality of branching is coerced by the definition of *choice* construct in the abstract syntax. When a choice $D_k : B_k, k \in I$, where I is an indexing set, is made the corresponding branch of behaviour B_k is selected and other choices are disregarded. Choices are expected to be deterministic with respect to the document types used in the activities; this property is formalized later when the constraints for business document types are defined.

When only modalities of *send* and *receive* are considered, the resulting abstract syntax for corresponding kinds of business protocols follows the abstract syntax of so-called *session types* [241, 103, 90, 255]. Following the terminology of session typing discipline, two kinds of choice constructs can be distinguished based on the modality of the activities in a choice

$$\Sigma^m D_i : B_i$$

: *branching* defining a choice between required activities ($m \in M_{must}$), and *selection* defining a choice between allowable activities ($m \in M_{may}$). The former can be represented as $\Sigma_i^{must} D_i : B_i$ and the latter as $\Sigma_i^{may} D_i : B_i$.

Expression of repetitive behaviour is provided by a countably infinite set of recursion variables $t, t', \ldots$. A business protocol $\mu t.send(A).receive(B).t$ specifies repetitive behaviour where first a business document of type A is sent and then a business document of type B is received, for example. Finally, a business protocol of type $\perp$ represents behaviour that has terminated successfully. Each business protocol is attached implicitly with an ending of type $\perp$, that is, $send(A).receive(B)$ is considered formally as $send(A).receive(B).\perp$.

There are two important restrictions to be made over the business document type syntax for achieving efficient subtyping. First of all, only tail recursion can be allowed to keep the subtyping decidable. By adopting this syntactic condition, the corresponding business document types correspond to so called regular tree languages [106], for which language inclusion problem is decidable [104]. If unrestricted recursion was allowed, the corresponding language would be a so called context free language for which language inclusion problem, and thus schema subtyping, is undecidable [104].

Secondly, in business document type unions only *deterministic labeling* is allowed, that is unions of form $l[D_1] + l[D_2]$ are prohibited. Such a deterministic labeling makes the business document subtyping polynomial instead of exponential [40]. These two restrictions are in line with the XML-Schema [263] standard which represents a regular tree grammar (or a single type tree grammar, to be more specific) with a deterministic content model [170]; this implies that the restrictions made are not too strict and provide descriptive power well enough suited for realistic cases.

6.2.3 Business protocol duality and subtyping

The relationships of compatibility and substitutability between business protocols provide the basic mechanisms for validating behavioural business service interoperability. Compatibility expresses a relationship of successful co-behaviour of where corresponding business protocols can

$$\overline{\overline{S}} = S \qquad \overline{end} = end \qquad \overline{!\alpha.S} = ?\alpha.\overline{S} \qquad \overline{?\alpha.S} = !\alpha.\overline{S}$$

$$\overline{?\Sigma_i \alpha_i : S_i} = !\Sigma_i \alpha_i : \overline{S_i} \qquad \overline{!\Sigma_i \alpha_i : S_i} = ?\Sigma_i \alpha_i : \overline{S_i} \qquad \overline{\mu t.S} = \mu t.\overline{S}$$

Table 6.2: The rules for session type duality relationship [255].

$$\text{SUB-EMPTY } \frac{}{() \leq ()} \qquad \text{SUB-TOP } \frac{}{D \leq \mathbf{T}} \qquad \text{SUB-SEQ } \frac{D_1 \leq D_2 \qquad D_1' \leq D_2'}{l[D_1], D_1' \leq l[D_2], D_2'}$$

$$\text{SUB-UNION1 } \frac{D \leq D_1 \quad or \quad D \leq D_2}{D \leq D_1 + D_2} \qquad \text{SUB-UNION2 } \frac{D_1 \leq D \qquad D_2 \leq D}{D_1 + D_2 \leq D}$$

$$\text{SUB-NAME1 } \frac{D \leq N(U)}{D \leq U} \qquad \text{SUB-NAME2 } \frac{N(U) \leq D}{U \leq D} \qquad \text{SUB-BASIC } \frac{bt \preceq bt'}{bt \leq bt'}$$

Table 6.3: Business document subtyping rules [40, 1].

proceed in such a way that both end successfully. Substitutability means a relationships between a pair of business protocols A and B which states that if A is substitutable with B, then B can be used in any context where A can be used. These concepts are formalized by the notions of session type duality and session subtyping [255] that are presented below.

For the definition of session type compatibility, a notion of behavioural duality is needed which expresses that two session types are counter-parts of each other. A session type S has a *dual type* $\overline{S}$ providing complementary behaviour if for each input action in α there is a complementary output action in $\overline{\alpha}$. The duality is defined formally by the rules given in Table 6.2.

Substitutability and compatibility

The subtyping relation is induced by the subtyping relationships prescribed for basic types, the sequence type, and the union type. The subtyping relation $\leq \subseteq D \times D$ is the least transitive and reflexive relation provided by rules given in Table 6.3. The rules and the notion of subtyping are similar to the subtyping presented in [1] and [40].

Business protocol subtyping provides a criterion for substitutability of business protocols. Given two business protocols S and T we write $S \leq T$ and say that S is a subtype of T if and only if they obey the inference rules given in Table 6.4. The business protocol subtyping rules are quite standard (see for example [90, 255]) the only notable differences being in rules *S-BRANCH* and *S-SELECT*. In these rules, we use the notation $|\Sigma_i^m D_i : B_i|$ for expressing the number of elements in a choice construct with modality m. The subtyping rules are inferred with respect to a typing environment Γ which consists of a finite set of inequalities $S \leq T$. Then, $\Gamma \wedge S \leq T$ means that S is a subtype of T, given the inequalities in context Γ. When $\emptyset \wedge S \leq T$ we simply write $S \leq T$. We assume that the set of basic types bt is provided with corresponding subtyping relation $\preceq$ between the basic types. Subtyping is co-variant on input and contra-variant in the output; branching and selection are both co-variant. Unwinding of recursive definitions is defined as usual [90]: $\mathbf{unwind}(\mu t.S) = S\{\mu t.S/t\}$.

Session subtyping provides a criterion for safe substitution between session types (and thus

$$\text{S-ASSUMP}\ \frac{S_1 \leq S_2 \in \Gamma}{\Gamma \vdash S_1 \leq S_2} \qquad \text{S-END}\ \frac{}{\Gamma \vdash end \leq end}$$

$$\text{S-INPUT}\ \frac{\Gamma \vdash T_1 \leq T_2 \qquad S_1 \leq S_2}{\Gamma \vdash ?T_1.S_1 \leq ?T_2.S_2} \qquad \text{S-OUTPUT}\ \frac{\Gamma \vdash T_2 \leq T_1 \qquad \Gamma \vdash S_1 \leq S_2}{\Gamma \vdash !T_1.S_1 \leq !T_2.S_2}$$

$$\text{S-BRANCH}\ \frac{\begin{array}{c}|\Sigma_i^? T_i : S_i| \leq |\Sigma_j^? T'_j : S'_j| \\ \forall k \in \{1, \ldots, |\Sigma_i^? T_i : S_i|\} : \Gamma \vdash S_k \leq S'_k \qquad T_k \leq T'_k\end{array}}{\Gamma \vdash \Sigma_i^? T_i : S_i \leq \Sigma_j^? T'_j : S'_j}$$

$$\text{S-SELECT}\ \frac{\begin{array}{c}|\Sigma_j^! T'_j : S'_j| \leq |\Sigma_i^! T_i : S_i| \\ \forall k \in \{1, \ldots, |\Sigma_i^! T'_i : S'_i|\} : \Gamma \vdash S'_k \leq S_k \qquad T'_k \leq T_k\end{array}}{\Gamma \vdash \Sigma_i^! T_i : S_i \leq \Sigma_j^! T'_i : S'_i}$$

$$\text{S-REC-L}\ \frac{\Gamma, \mu t.S_1 \leq S_2 \vdash \mathbf{unwind}(\mu t.S_1) \leq S_2}{\Gamma \vdash \mu t.S_1 \leq S_2}$$

$$\text{S-REC-R}\ \frac{\Gamma, S_1 \leq \mu t.S_2 \vdash S_1 \leq \mathbf{unwind}(\mu t.S_2)}{\Gamma \vdash S_1 \leq \mu t.S_2}$$

Table 6.4: Business protocol subtyping rules [90].

business protocols): if session type S_1 is a subtype of S_2, written as $S_1 \leq S_2$, then S_1 can be used in any context where S_2 is used [255].

Provided with the notions of session duality and subtyping, a concept of *compatibility* was introduced in [255]: a session type S_1 is compatible with S_2, if $S_1 \leq S'$ for some session type S' which is a dual of S_2. Compatibility between session types guarantees that composition between processes that share a session channel and have compatible typing for that channel can proceed without deadlocks with respect to the session. It should be noted, that in general session typing of processes does not provide a complete interoperability validation mechanisms in a sense that compatible typing would imply interoperability between business processes. This is because session types do not consider the inter-leaving of actions between different sessions or the order between initialisation of sessions as processes do.

6.3 Knowledge management for federated service communities

For establishing the necessary knowledge management repositories for federated service communities, the domain concept intentions are provided with metamodel representations. The relationships between the concepts and their prescriptive models are formalized by the federated service community knowledge management metamodel, an extension of the global model management metamodel defined in Section 4.2. While the domain concepts are used for describing an ontology of federated service communities and provide a common vocabulary to manage such constellations, the model representations of their intentions provide means for developing corresponding kinds of engineering artifacts and representing the information contents during community estab-

lishment processes.

In this Section, we describe how a reference architecture for federated service communities is constructed within the Pilarcos framework. In Section 6.3.1 we describe a conceptual approach and the technology to be used for unifying ontology engineering and model-driven engineering domains. After that, Section 6.3.2 introduces the process of knowledge artifact composition. Section 6.3.3 introduces a set of platform-independent intermediate models that are used in the generation of knowledge repositories. In Section 6.3.4 a platform-specific model for the knowledge repositories is introduced. The approach to be presented in this Section uses model transformation and model weaving for automatizing of knowledge repository generation; for this purpose the Atlas Transformation Language (ATL) and the Atlas Model Weaver (AMW) [68] technologies are utilized. Finally, some implementation issues are discussed in Section 6.3.5.

6.3.1 Unifying technical spaces

Whereas linguistic metamodelling is used for defining metamodels that prescribe the form of artifacts utilized within a domain of concern, ontological metamodelling is used for defining top-level ontologies that describe some domain. In the context of service-oriented computing and related software engineering methodologies both of these metamodelling approaches are needed to facilitate a service ecosystem for loosely coupled service collaborations. For this purpose, the corresponding technological domains of ontological engineering and model-driven engineering need to be unified at some level.

In the context of model-driven engineering such technological domains are known as *technical spaces*. A technical space is *"a working context with a set of associated concepts, body of knowledge, tools, required skills and possibilites"* [136]. All engineering domains are associated with a corresponding technical space comprising domain specific best practices and tools that are applied for the engineering activities. Model-driven engineering, ontology engineering, or database management system engineering have their own characteristic technical spaces, for example.

Different technical spaces can be bridged using transformations that arbitrate the intentions of the engineering artifacts from a domain to another. Such bridging is needed typically during system changeovers or for transitioning knowledge between administrative domains with different technical spaces. With respect to model-driven engineering, transformations between technical spaces can be used to facilitate individual spaces with new capabilities, such as model checking or model enrichment within a metamodelling technical space [194], not usually available,.

In the modelling framework presented in this Thesis, technical spaces are not bridged but instead two different spaces are used in conjunction to establish a feasible service ecosystem and a corresponding modelling framework. The metamodelling technical space consisting of model-driven engineering artifacts and the ontological technical space consisting of ontology engineering artifacts are conjoined by the knowledge management metamodel, or megamodel represented in Section 4.2. Within this conjoinment the intentions of ontological concepts residing in the ontological technical space are represented concretely as metamodels residing in the metamodelling technical space. This approach can be classified as a "hybrid approach" to technical space transformation, as discussed in [194]. However, we do not unify the technical spaces to mediate differences between artifacts residing in them, but to genuinely utilize the characteristics of the corresponding technological domains. Especially, the approach utilizes the descriptive nature of ontological engineering domain to manage the open nature of service-oriented computing environments and the prescriptiveness of model-driven engineering is utilized for facilitating service-oriented system engineering.

The domain ontologies that were represented informally in the previous Chapters as diagrams

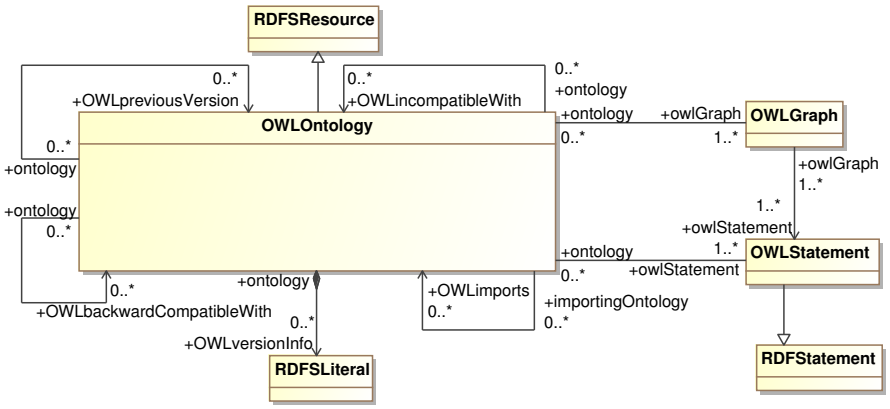

Figure 6.7: ODM Ontology metamodel.

comprising of ellipses and arrows are formalized using the Web Ontology Language (OWL) [259]. The corresponding OWL ontology represents the domain concepts and their inter-relationships, but does not dictate the structure of the concepts. The OWL specifications [259] define the syntax and semantics for an ontology description language that is built upon the simpler RDF [260] specification. The OWL language comprises three sub-languages distinguished by their expressive power, namely OWL Lite, OWL DL, and OWL Full. For expressing simple classification hierarchies and simple constraints the OWL Lite sub-language can be used which provides constructs for subclass hierarchy construction via subclasses and property restrictions. OWL DL provides a more expressive language with semantics based on description logics [174], thus retaining computational completeness and decidability. Finally, the OWL Full sub-language enables maximum expressiveness but does not provide any computational guarantees [259]. In OWL Full classes can be treated as individuals which is not allowed in OWL DL or OWL Lite, for example.

Metamodels representing the intentions of domain concepts are defined as Eclipse Ecore models. Ecore is name of the the metametamodel within the Eclipse Modeling Framework (EMF) [69], a modelling and code generation framework developed under several Eclipse [67] projects that strive for open-source model-driven engineering tools and facilities. The ECore model aligns closely with the MOF standard [181] with naming conventions being the most fundamental difference between these two metametamodels.

As the first step in the unification process, the OWL ontology is transformed into Ontology Definition Metamodel (ODM) - compliant [183] ECore representation. ODM is an OMG specification for providing a MOF metamodel to support development of ontologies using UML modelling tools and two-way model transformations between ontologies described with ontology representation languages and ontologies described using a dedicated UML profile [39]. Especially, the ODM specification provides MOF-based metamodels describing OWL ontologies.

An OWL-ontology is described by using the concept of *OWLOntology* which is illustrated in Figure 6.7. An *OWLOntology* comprises a set of OWL graphs and a set of OWL statements. An *OWLGraph* is a specialization of an *RDFGraph* which represents graphs conforming to the Resource Description Framework (RDF) [260] specification. Each *RDFGraph* comprises a set of statements, or triples represented by the concept *RDFStatement* where each triple specifies a *subject*, *predicate* and an *object*. The subset of RDF statements that are valid OWL statements are represented by the concept of *OWLStatement* [39].

While the ODM specification strives for as complete mapping between MOF and ontology de-

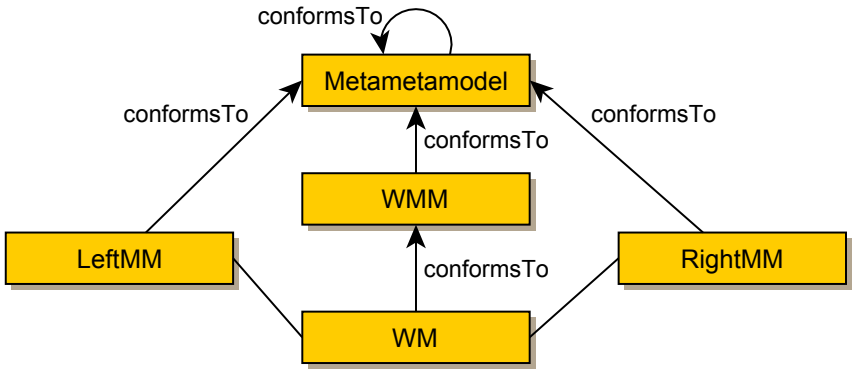

Figure 6.8: Weaving conformance relationship in a situation involving two weaved models [64].

scription languages as possible, only a subset of the ODM is needed for the purpose of the technical space unification, namely the RDFBase, RDFS, and OWLBase metamodels [183]. These metamodels provide all the necessary elements for representing different kinds of classes and properties in OWL declarations. Implementing the Ecore models on the basis of the ODM specification is straight-forward. The Jena 2 [115] semantic web framework for Java includes an ontology API that can be used for reading and writing OWL files, and even for reasoning over OWL ontologies. In this case, only the OWL import functionality of Jena 2 is needed.

Provided with an Ecore representation of the domain ontology, an *annotation model* describing the relationships between the domain concepts and the metamodels defining concept intentions can be constructed. These relationships declare if a repository should be provided for the corresponding knowledge artifact. Consequently, the annotation model for federated service communities follows the mappings illustrated previously in Table 6.5.

The annotation model is a *weaving model* [64] relating the metamodels representing the domain ontology and concept intentions. Model weaving is an operation over models in which typed links between a set of models are established [64]. Whereas model transformation metamodels have fixed semantics, weaving metamodels describe links between model elements that have user-specified semantics. Especially, model weaving can be utilized in automatic creation of model transformations based on the annotations defined within a weaving model.

The corresponding arrangement involving two metamodels is quite common in model weaving; the corresponding conformance relationships between metamodels are illustrated in Figure 6.8. In our case, the *LeftMM* corresponds to the Ecore-based ODM metamodel, and *RightMM* to an Ecore metamodel defining the concept intentions. The weaving metametamodel *WMM* defines the link types that can be used to associate concepts in the *LeftMM* with their intentions in the *RightMM*. The weaving model *WM* is a model that conforms to the weaving metamodel *WMM*.

For mapping the domain concepts with their intentions a weaving metamodel defining correspondences between the ODM-based domain ontology metamodel and the concept intention metamodel is defined. While the ODM-metamodel is good for representing ontologies within the metamodelling technical space, it is not maybe the best model to use for the mapping purposes. Thus, the ontology model actually presented for the user as an Ecore class hierarchy with ontology concepts and properties represented as classes and references.

The facilities provided by the Atlas Model Weaver (AMW) tool [68] are used for the construction of the annotation model. Weaving metamodels are constructed as model extensions [16]

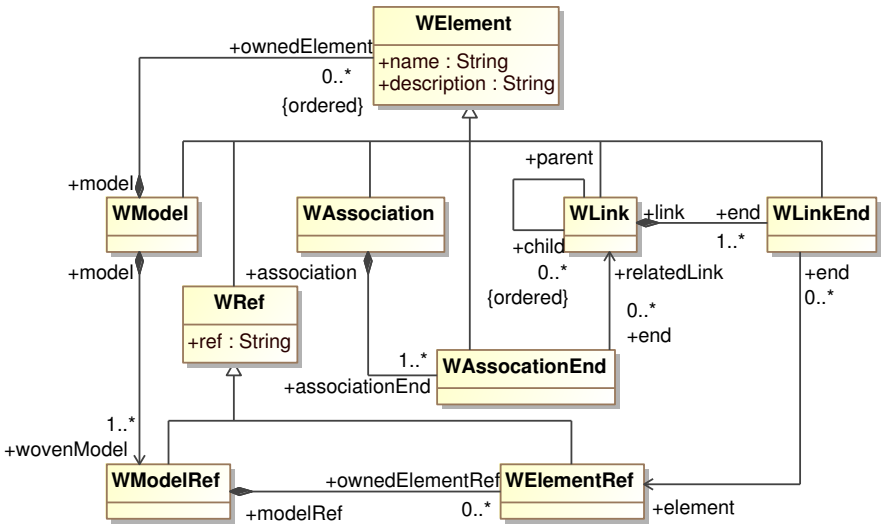

Figure 6.9: The AMW core weaving metamodel.

over the core weaving metamodel [64, 62]. The core weaving metamodel illustrated in Figure 6.9 defines the elements of *WModel*, *WModelRef*, *WElement*, *WElementRef*, *WLink*, *WLinkEnd*, *WAssociation* and *WAssociationEnd*. These are the model elements that are extended by the actual weaving metamodels to represent the corresponding model weaving semantics.

6.3.2 Composing knowledge artifacts

In a service-oriented computing environment capabilities and properties of services are described using *service declarations* that are published and discovered using shared repositories provided by the service ecosystem. In service-based communities, and thus federated service communities two kinds of service declarations are distinguished: service definitions describing conceptual services and service descriptions declaring the properties of business services. The corresponding metamodel defining service declarations in the federated service community reference model is illustrated in Figure 6.10. A *ServiceDeclaration* is considered as an abstract reference model with two specializations: *ServiceDefinition* representing the intention of the conceptual service concept, and *ServiceDescription* representing business service intentions.

While service declarations are used for publishing the different kinds of services available, service advertisements and queries are used for discovering services to be used within a specific context. As illustrated in Figure 6.11, service discovery models represent the intentional part of the service offer concept defined in the federated service community domain ontology. While a *ServiceAdvertisement* must provide a complete representation of a *ServiceOffer*, a *ServiceQuery* can be left incomplete with respect to some elements of the service offer concept and is considered as a matching template over service advertisements.

Consequently, there can be several representations for a domain ontology concept in the same reference model. In the case of *ServiceAdvertisement* and *ServiceQuery* models, the former will be provided with a specific repository to store service offers where as the latter is considered as a temporary engineering artifact that is not provided with a persistent storage.

Similarly to service declarations also other domain concepts are provided with prescriptive model representations; these knowledge mappings are summarized in Table 6.5. The left column

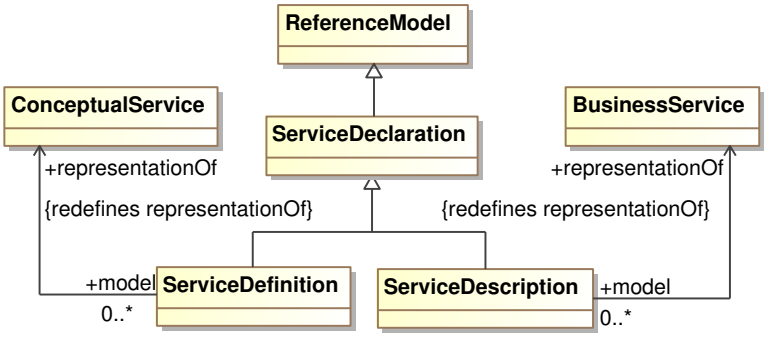

Figure 6.10: Service declarations in federated service community reference model.

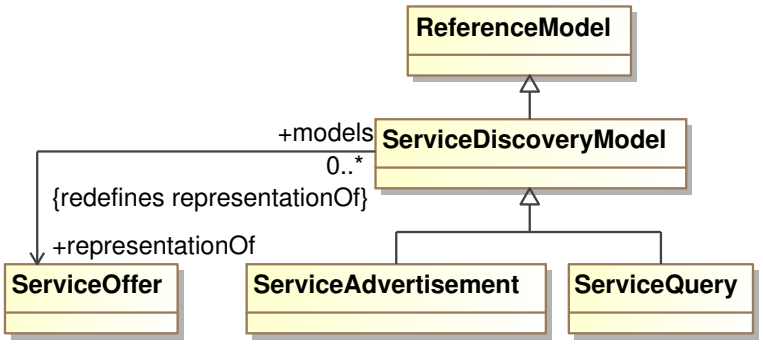

Figure 6.11: Service discovery models in federated service community reference model.

Domain concept	In repository	Intentional metamodel
Service declaration representations		
ServiceType	yes	ServiceDefinition
BusinessService	yes	ServiceDescription
Service usage representations		
ServiceContext	yes	ServiceContextModel
ServiceOffer	yes	ServiceAdvertisement
	no	ServiceQuery
Service cooperation representations		
ServiceChoreographyType	yes	BusinessNetworkModel
Service contract representations		
ServiceCollaborationContractType	yes	EpochModel
ECommunityContractType	yes	EContractModel
ECommunityContract	yes	EContract
Cooperation facility representations		
MediumType	yes	MediumModel
ChannelType	yes	ChannelModel
ConnectorType	yes	ConnectorModel
Service behaviour representations		
BusinessProtocol	yes / no	BusinessProtocol
Service feature representations		
BusinessDocumentType	yes	BusinessDocumentType

Table 6.5: Relationships between domain ontology concepts and the models representing their intentions in the federated service community reference model.

contains a name for the domain concept, middle column states if a model repository must be used for managing the corresponding kind of knowledge artifacts, and the right column states the name for the prescriptive metamodel in the model-driven engineering domain. It should be noted that table 6.5 does not include all mappings needed, but only represents the most important concepts and their corresponding representation models.

Domain concepts are associated with their intentional metamodels using a model weaving metamodel. The mappings annotated between domain concepts and their intentions must declare if a model repository should be generated, and what is the name of the model artifact that corresponds to the concept. The weaving metamodel for domain ontology concept mapping comprises of link types that represent correspondences between concepts and metamodel classes, and ontology properties and metamodel associations, correspondingly.

The foundational elements of the domain concept mapping weaving metamodel are illustrated in Figure 6.12. The metamodel defines two link types: *KnowledgeArtefact* that is used for associating a concept with its intension class thus representing a knowledge artifact, and *PropertySynonym* which equates a property of the domain ontology with an association in the domain intention metamodel. While the weaving metamodel is illustrated in Figure 6.12 as a class diagram, in practise weaving metamodels in the AMW framework are defined using the Kernel MetaMetaModel language [121] which provides a textual domain-specific language for metamodel creating.

In the weaving metamodel for domain ontology concept mappings, the *left* association of a *KnowledgeArtefact* element should refer to a class representing a named OWL-class, and the *right* association to an Ecore-class representing the intention of the concept. The *PropertySynonym* link type is used when the domain ontology and the intentional metamodel have overlapping properties and associations. Such overlapping between ontology properties and metamodel associations must be annotated explicitly to identify semantic relationships that possibly should be associated with

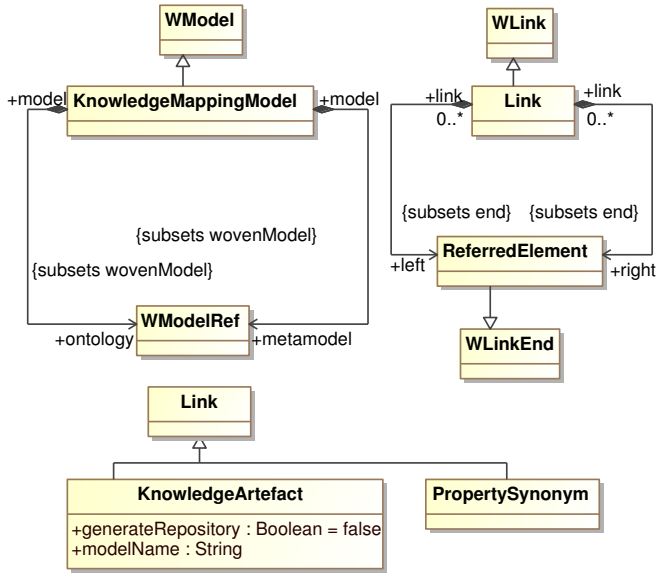

Figure 6.12: Weaving metamodel for domain ontology concept mapping.

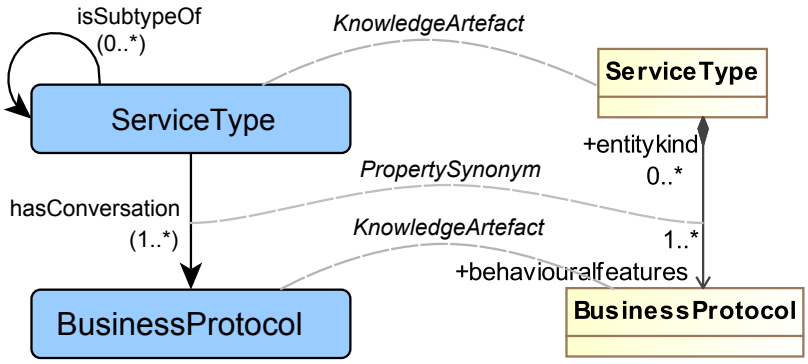

Figure 6.13: An example illustration of concept and property mapping.

semantic interpretation. Any domain ontology property that is not annotated as a synonym with an association in the corresponding intention metamodel is considered as an intrinsic semantic property of the domain, that is, a non-structural relationship relating domain concepts.

An example of a mapping between a domain concept and its intentional metamodel is given in Figure 6.13. In this example, the concept of *ServiceType* defined in the federated service community domain ontology is mapped with the metamodel representing its intention in the metamodelling technical space. As defined in the previous chapter, the intentions for the concepts of *ServiceType* and *BusinessProtocol* are named following their concepts. However, where the ontological concept of *ServiceType* is associated with a set of *BusinessProtocol* concepts, in the metamodelling technical space the association relating a service type with its business protocol representations is named as *behaviouralfeatures*, reflecting the subtyping hierarchy of behavioural entities.

Especially, the *subtypeOf* property of the *ServiceType* domain concept is *not* provided with

an equal association in the intentional metamodel. This design decision is rationalized by the fact that subtyping between service types is a semantic relationship between service type instances that should be associated with an interpretation formalized by the business protocol subtyping rules provided in Section 6.2.3. Currently, this mapping has to be provided manually by implementing the corresponding validation functionality to the knowledge artifact repository. However, also the semantics of ontological relationships, in this case the subtyping rules, should be explicitly modelled in the future to automate the generation of knowledge repository functionality as far as possible. Rule markup language such as RuleML [212] and its MOF-based metamodel [269] could be utilized for this purpose.

6.3.3 Generating metamodels for knowledge repositories

The weaving model defining the correspondences between domain ontology concepts within the ontology engineering technical space and their intentions in the metamodelling technical space facilitates generation of knowledge repositories. For each *KnowledgeArtefact* element defined in the weaving model a repository will be generated. So-called *higher-order transformations* (HOT) [28, 120], that is transformations that take transformations as input, produce transformations or both, are used in this process. The transformation discussed in the following are based on the Atlas Transformation Language (ATL) [123, 122].

The transformation process as a whole and the dependencies between different models is illustrated in Figure 6.14. The higher-order transformation *AMW2ATL_HOT* takes as input three different models: metamodel *MMDO* representing the domain ontology, metamodel *MMR* representing the domain concept intentions, and the weaving model, or annotation model*MW* provided by a user. Given these inputs the higher-order transformation produces another ATL-transformation named *KA2Repository* that is specific for the given domain ontology, intentional metamodel, and annotation model.

Finally, the *KA2Repository* transformation produced by the higher-order transformation is used for generating a knowledge repository metamodel. The *KA2Repository* transformation takes a knowledge artifact of the domain (i.e. a concept-intention pair) and produces a repository metamodel conforming to a knowledge repository metamodel.

The knowledge repository metamodel is based on the concept of model repository introduced in Section 4.2. The metamodel represents knowledge repositories in federated service communities and is illustrated in Figure 6.15; the *redefines* property declarations have been left out from the figure to increase the readability of the diagram. A *KnowledgeRepository* comprises a set of *RepositoryRelationships* and is associated with a *RepositoryItemType*. Each *RepositoryRelationship* is a relationship between two *Model* artifacts and represent a semantic relationship defined within the domain ontology. The set of *RepositoryRelationship* elements include all those inter-concept relationships that were not annotated as synonyms of metamodel associations in the weaving model.

The *RepositoryItemType* is a technical space specific representation of a concept intention. In the context of the Pilarcos framework, the Eclipse Modelling Framework (EMF) [69] and its Ecore metametamodel is used for model representations. EMF [69] provides facilities for basic model management activities, including serialization of models and code generation. For enabling model serialization, every element of an Ecore model has to be contained by some resource. More over, for alleviating modelling practises it should be possible to attach model elements with (descriptive) names that can be used for referencing the elements and clarifying the model presentation. Therefore the Ecore models are represented typically as a package with one root element that comprises a set of named elements.

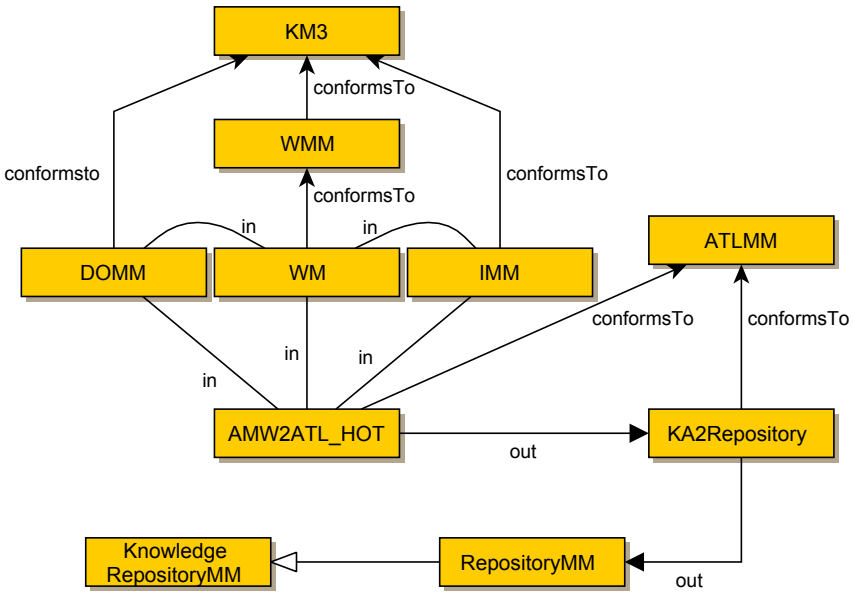

Figure 6.14: Generating metamodels for knowledge artifact repositories.

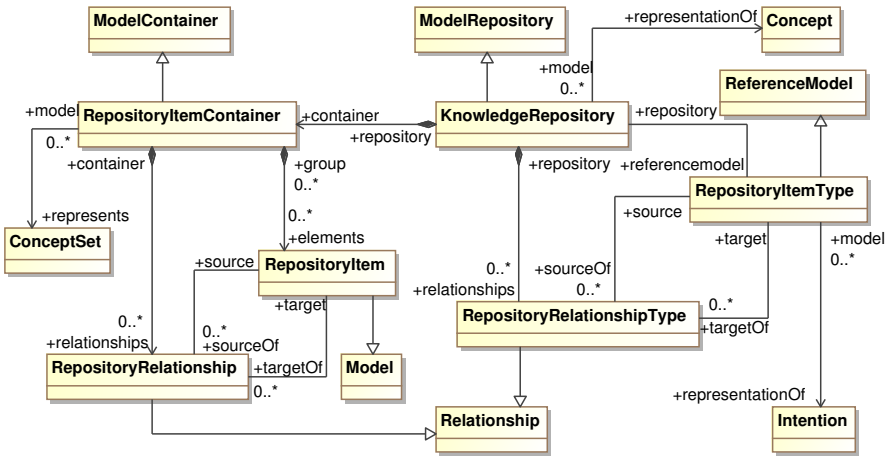

Figure 6.15: An intermediate metamodel for knowledge repositories.

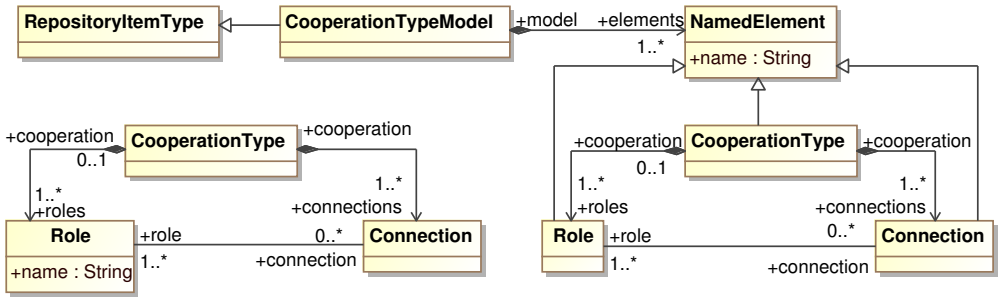

Figure 6.16: From a metamodel representing a concept intention (at the lower left corner) to a Ecore-based repository item type.

Ecore-based representations of the concept intentions can be derived automatically by using model refactoring techniques based on model transformation concepts [33]. When deriving the *RepositoryItemType* models the refactoring involves associating a *NamedElement* as a supertype for model elements appropriately, and pulling-up the *name*–attributes for preventing naming clashes. As an example, Figure 6.16 illustrates part of the *CooperationType* metamodel representing a concept intention discussed in Section 5.2, and the corresponding repository item type after refactoring. The knowledge artifact of *CooperationType* is represented by *CooperationTypeModel* that comprises a set of *NamedElement* elements. All the elements that were part of the concept intention metamodel are now subclasses of the *NamedElement*, and the *Role*-element does not contain a *name*-attribute after the refactoring.

6.3.4 Providing knowledge repository implementation artifacts

ModelBus is a conceptual architecture and a technological platform based on middleware technology that provides modelling services for model-driven engineering purposes [35]. ModelBus infrastructure was initially developed under the ModelWare-project [168] and its development is now further continued as part of the Eclipse Model Driven Development integration (MDDi) project [72] that strives for realization of modelling tool integration within an open model-driven engineering environment.

ModelBus architecture provides an abstract platform and a tool-bus that can be used for facilitating a service ecosystem with knowledge repositories. Especially, the ModelBus architecture provides a metamodel describing modelling services which can be utilized as high-level platform-specific model for the knowledge repositories. The ModelBus modelling service metamodel is illustrated in Figure 6.17.

Provided with a repository model and a ModelBus modelling service model, implementation artifacts for knowledge artifact repository can be generated automatically. The repository generation process involves three different transformations: *1)* a model-to-model transformation from *KnowledgeRepository* model to *ModelBus* model, *2)* a model-to-text transformation from the *ModelBus* model to a modelling service WSDL, and *3)* a text-to-text transformation from WSDL to Java classes. These transformations comprise a repository generation process which is illustrated in Figure 6.18.

The *KR2ModelBus* transformation takes a model conforming to the *KnowledgeRepository* metamodel and transforms it to a model conforming to the ModelBus modelling service model. The *KnowledgeRepository* element is transformed to a ModelBus *ModelingInterface* element. The

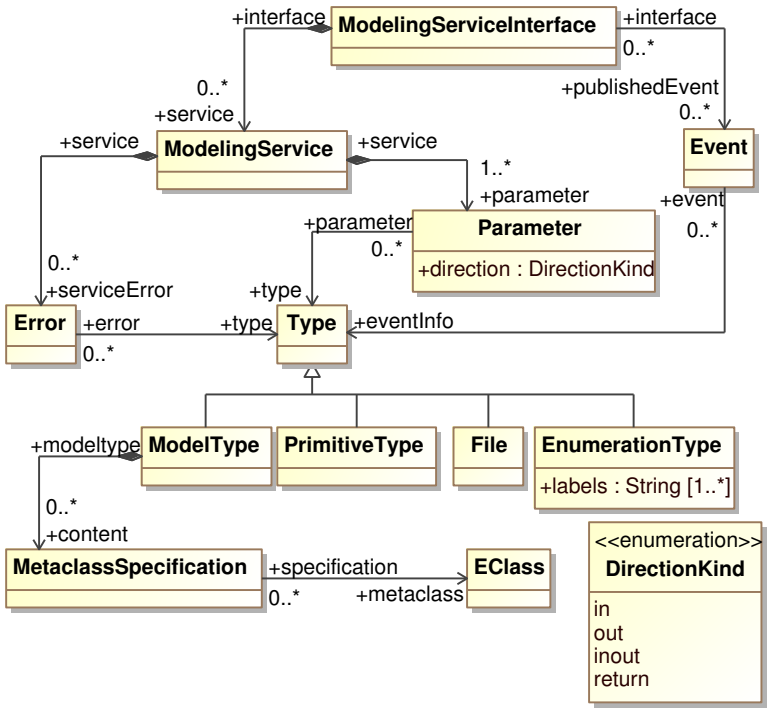

Figure 6.17: Simplified illustration of the ModelBus modelling service metamodel.

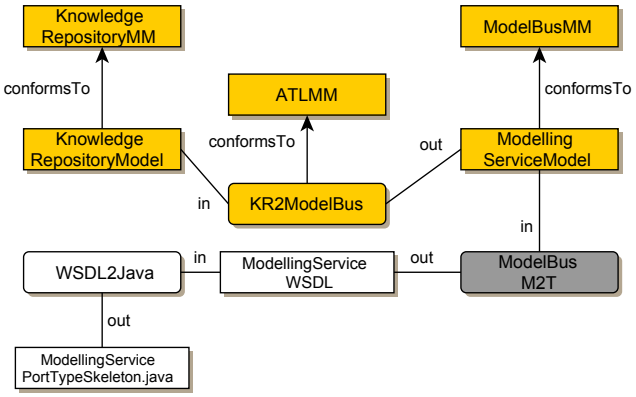

Figure 6.18: Generating knowledge repository implementation artifacts.

generated modelling service model describes a CRUD (for Create, Read, Update and Delete) interface over the knowledge repository models and relationships. The model maintenance modeling service takes a parameter whose type is defined by the *RepositoryItemType* of the repository; correspondingly, the *MetaclassSpecification* associated with the modelling service type references to the Ecore class representing the corresponding model element (e.g. the *CooperationTypeModel* in Figure 6.16). Similarly for each *RepositoryRelationship* element in the knowledge repository model a *ModellingService* element is created in the ModelBus modelling service description model.

Model-to-text (M2T) transformations are used in the knowledge repository generation process for delivering a WSDL-description of the corresponding modelling service interface. The ModelBus implementation that is provided as part of the MDDi [72] project is accompanied with code generation facilities that can be utilized for implementing the modelling services. Especially, the ModelBus code generation tools provide model-to-text (M2T) transformations that generate WSDL [55] interfaces for the modelling services described. The model-to-text transformation is based on the Java Emitter Templates (JET) which is a code generation framework implemented by the Eclipse M2T project [71].

Finally the Web Service implementation artifacts, such as Java classes and deployment descriptors, can be generated from the modelling service WSDL-description with Apache Axis2 [8] tools.

6.3.5 Implementation issues

The knowledge repository implementations are generated from the modelling artifacts as defined in previous sections. In the following we discuss briefly the technical issues related to these implementations and consider especially the type repository implementation. The knowledge repositories provide services for storage and retrieval of models, and maintenance of extendable ontologies and vocabularies of models such as service types or business network models. For decoupling the different functionalities needed for implementing a knowledge repository, the functionality is modularized in four separate main modules. A knowledge repository comprises four functional modules illustrated in Figure 6.19: *1)* Web Service interface, *2)* model repository handler, *3)* model validation, and *4)* model persistency modules.

The main artefacts of the Web Services module are provided by the Axis2 [8] *WSDL2Java* code generator. The implementation provided by the Axis2 framework handles SOAP messaging issues, such as marshaling of SOAP envelopes etc. Especially, the code generator provides Java bean classes that represent the different messages exchanged. These bean classes are then used in the stubs and skeletons to represent the information contents of SOAP messages. The Java classes generated by the Axis2 for service skeleton classes have to be provided with the actual repository functionality providing validation and persistency services. For this purpose the skeleton class is instrumented with knowledge repository specific Java code that converts Axis2 Object Model (AXIOM)-based representations of the SOAP messages to an Eclipse EMF-based [69] Java bean representation of the model. After that the Web Services module calls the model repository handler code with appropriate parameters.

Model repository handler module provides the business logic needed for repository functionality. This module implements the CRUD operations over the repository items and relationships. Implementation of the model repository handler functionality can be largely automated. The model handler core classes can be generated from the intentional metamodel using the EMF code generator [69] and a JET [70] model-to-text transformation taking the knowledge repository model as an input provides all the necessary information for generating at least the basic CRUD operations.

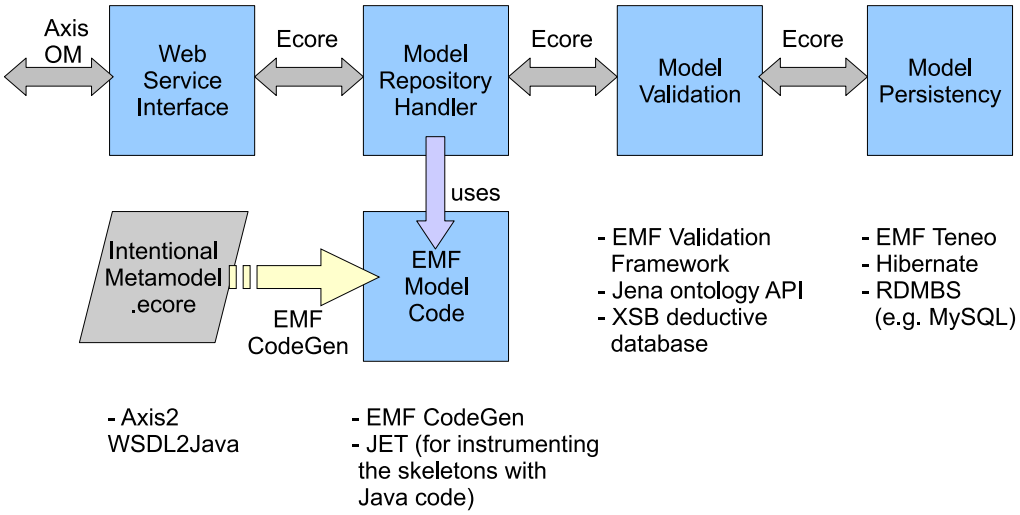

Figure 6.19: Knowledge repository implementation modules and related technological artifacts.

The implementation utilizes the functionality provided by model validation and model persistency modules.

Model validation module provides the essential functionality for maintaining the consistency of knowledge artefacts with respect to the underlying domain ontology and intentional metamodels. Especially, the semantics of the ontological relationships, including the ontological and linguistic conformance relationships, is implemented in this module. Currently the validation module skeleton needs to be manually provided with the appropriate validation functionality. In the future we wish to automate the implementation of the validation procedures as far as possible by utilizing the OWL semantics of the domain ontology and OCL annotations in the intentional metamodels. The Eclipse EMF Validation Framework [69] provides a potential implementation framework for attaching such domain specific validation functionality to the knowledge repository in a generic manner.

The validation module implementation is based on the EMF and its Validation framework [69] for handling the models and their linguistic validation, Jena ontology API [115] for managing ontological metamodelling issues, and XSB deductive database and tabled logic system [219] for validating that the semantics of the ontological relationships (e.g. service subtyping) are preserved. The knowledge repositories have actually quite natural connection with so called deductive databases [87]. Deductive databases extend conventional relational databases with mechanisms that permit automated logical deduction over the asserted facts based on theories and integrity constraints defined for the corresponding information [87]. For example, a service type repository can be considered as a kind of deductive database where the theory contains the rules of the corresponding type system and integrity rules define additional consistency rules (such as transitivity of subtyping relation) to be maintained over the typing information.

Finally, the model persistency module provides basic support for storing and retrieving models and model relationships. For this purpose the Eclipse EMF Teneo [69] which provides a database persistency solution for EMF models can be utilized. The Teneo framework utilizes the EMF framework and Hibernate [102] for generating model specific object-relational–mappings.

6.4 Providing service-oriented software engineering facilities

Provided with a domain-ontology of federated service communities and the corresponding knowledge management infrastructure, the ecosystem is still lacking one fundamental ingredient: the facilities for instrumenting service-oriented software engineering. The SOSE facilities include design and development tools for both functional and non-functional service artefacts, as well as methodologies that utilize these tools and the global knowledge management infrastructure. The tools and methodologies are aligned with the underlying domain ontology.

6.4.1 Tools for creating the engineering artefacts

The engineering artefacts in the federated service community ecosystem include service definitions and descriptions, and business network models, for example. All these artefacts must be provided with corresponding modelling tools that are used by the service designers. In practise, we have two choices for implementing the modelling tools: creating them from a scratch, or taking advantage of the UML extension mechanism. Both of these approaches have their weaknesses and strengths.

When creating the modelling tools from a scratch we are not constrained by any modelling notation or diagram layout. On the one hand, this gives us freedom to use notations that best suite the corresponding modelling task in hand. On the other hand, use of heterogeneous and varying modelling notations across the ecosystem may also become a hinder for exchanging modelling knowledge and conventions between partners within a "global software engineering" (see for example [59]) community.

UML profiling mechanism [182] provides means for extending UML diagrams with domain specific vocabulary and constraints. The most remarkable benefit of this approach is that modelling tools already available and familiar to modelling professionals can be used for creating the domain ontology models. On the downside, the UML profiling mechanism does not provide same level of flexibility as the previous approach for expressing the domain concepts. UML profiling mechanism can be used within the federated service community for modelling the behaviour of services and service choreographies using UML activity and sequence diagrams [182], for example. UML profiles have been used for implementing domain specific modelling languages for service-oriented architectures [154, 20] and their non-functional aspects [268], for example.

6.4.2 Management of non-functional features in service ecosystems

In service ecosystems comprised of business services, collaboration infrastructure services, and their production facilities non-functional features are used *a)* as selection criteria during service discovery, *b)* as parts of collaboration contracts and service-level agreements, and *c)* as artefacts for enabling efficient service engineering practices. Being such a pervasive element of service ecosystems, non-functional features induce several kinds of issues during the life-cycle of service-oriented systems and business services. First of all, as non-functional features are used for selecting and ranking services during service discovery, mechanism that allow identification and comparison between them must be provided. Secondly, dynamically negotiatable collaboration contracts and service-level agreements require mechanisms that allow validating compatibility between individual non-functional features, and consistency of compositions between a set of non-functional features and corresponding functional elements. Finally, service-oriented software engineering must be provided with appropriate mechanisms for non-functional feature introduction to enable efficient service production.

The issues related to non-functional features in business service ecosystems are in general related to their identification, selection, and introduction during business service development and collaboration establishment. For facilitating feature identification and selection, an ontology providing means for their classification is needed. Introduction of non-functional properties have two separate meanings in this context: dynamic binding of non-functional features during business service binding, and static binding during service engineering artifact production. In both cases, the most relevant issues are related to the compatibility of non-functional features with each other, and consistency of compositions between a set of non-functional features and corresponding functional elements.

While the set of possible non-functional features is open and can not be predetermined or enumerated due to their context dependency and evolution of systems, we believe that their usage can be disciplined by deliberate management facilities. The previous chapters have laid a foundation for such facilities by elaborating the nature of non-functional features within the domain ontologies and metamodels. We believe that in a context with the service engineering perspective and openness of the knowledge environment being such fundamental elements, non-functional features have to be provided also with "translational semantics" defining how non-functional features affect the functional elements of the abstract platform (defined by the domain ontology and corresponding metamodels). Towards this end, different kinds of non-functional features shall be attached with their characterizing weaving metamodels [64] that declare their intentions as generic model transformations. Corresponding weaving models are then used for producing the actual model transformations that implement the non-functional properties at the level of individual models and the abstract platform.

Chapter 7

Discussion

This Thesis has laid foundations for instrumenting open service ecosystems with infrastructure services and service-oriented software engineering facilities. This foundation was realized by a modelling framework defined as a series of domain ontologies describing the essential entities, features and cooperation forms found in collaborative computing environments. These basic concepts and their inter-relationships were extracted by a deliberate analysis of modern collaborative computing especially in the context of inter-enterprise collaborations.

In Chapter 2 we identified the essential concepts and enablers of modern collaborative computing environments. We analysed the properties of collaboration agents and environments, and different mechanisms for achieving interoperability within electronic collaborations. Based on this analysis we identified some characteristic attributes of electronic collaboration environments and provided a comparison between a selection of modern collaboration platforms. We found out that current state of the art in collaborative computing does not provide the necessities for truly loosely coupled collaborations in open service ecosystems. We concluded Chapter 2 by introducing two significant engineering disciplines for enabling service-based collaborative computing, namely service-oriented computing [191] and model-driven engineering [225].

Chapter 3 concentrated on the issues related to collaborative computing in the context of inter-enterprise cooperation. The discussion addressed the concepts of inter-enterprise collaboration, interoperability issues within such environments, as well as provided a characterization of a service-oriented software engineering framework and its supporting infrastructure. We first identified the essential concepts of inter-enterprise collaborative computing, including business services, business protocols and processes, and business collaboration networks. After that we elaborated on the management of interoperability by introducing a classification of interoperability with respect to the different aspects of inter-enterprise computing, and provided a brief discussion about different kinds of interoperability dependencies (i.e. horizontal and vertical interoperability) between these aspects. Section 3.3 introduced a proposal for service-oriented software engineering framework comprising of two primary engineering processes, variability management activities and infrastructure services for enabling global software engineering practices within the framework. The infrastructure services and tool-chain required for instrumenting this vision were then discussed in Section 3.4.

The core for the contribution of this thesis was given in Chapter 4 which defined the foundational metamodels, or domain ontologies needed for establishing a modelling framework for service-oriented software engineering. The modelling framework is founded on modern modelling disciplines and model-driven engineering practices. Especially, the provided modelling framework addresses both linguistic and ontological metamodelling activities by providing metamodels that unify the corresponding technical spaces and engineering domains. We believe that such a uni-

fication is a prerequisite for managing the different model artifacts within the open knowledge landscape required for providing loosely coupled engineering disciplines.

Actual domain ontologies describing collaborative computing environments of varying abstraction levels were defined in Chapter 5. Starting from a generic domain ontology describing cooperative communities we ended up to the domain ontology of service-based communities. Metamodel extension and specialization were utilized for refine the domain ontologies to more detailed ones. The concepts of the domain ontologies reflected the analysis of collaborative and inter-enterprise computing environments done in the previous chapters.

Finally Chapter 6 provided the first steps towards establishing a service ecosystem for federated service communities based on the principles and domain ontologies laid in the preceding chapters. First a domain ontology for federated service communities was described in Section 6.1. The domain ontology makes explicit the concepts used in the Pilarcos framework [141, 139] which introduces concepts such as service types, service offers and eContracts. The concept of service type is of uttermost importance when considering interoperability management in federated service communities; for this reason, it was discussed separately in Section 6.2. Basically, a service type represents a conceptual service and formalizes interoperability between services using a session typing discipline [103, 90, 255].

The engineering knowledge artifacts fundamental in the federated service community ecosystem were discussed in Section 6.3 which introduced the knowledge management metamodel. In this section we also described how in practise the ontological and linguistic metamodelling artifacts will be unified by utilization of model weaving [64] and how knowledge repositories will be semi-automatically generated for each selected concept.

The knowledge artifact repositories share similarities with the type repository function described as part of the ODP standardization [108]. An ODP type repository provides functions for management of typing information, naming of types and interworking and federation of different type repositories [108, 169, 137]. Type repository is thus basically a persistent storage of meta-information consisting type descriptions and relationships between them, and operations for publishing, retrieving, querying and validating typing information. The type repository function defined as part of the ODP standardization [108] is based on the OMG's Meta-Object Facility (MOF) standard [181]. In this context, the MOF provides for a standardized manner of establishing repositories of type information for arbitrary type systems. Each knowledge artifact repository in the Pilarcos framework can be considered as kind of an ODP type repository with the ontological and linguistic conformance relationships dictating the corresponding typing rules and a modelling service interface providing operations for managing the typing information. The knowledge artifact repositories of the Pilarcos framework are based on the EMF platform [69] and the Ecore metamodelling language.

We concluded Chapter 6 with a discussion about the implementation choices for the required modelling tools as well as gave a preliminary vision about the management of non-functional features in service-based ecosystems. These two elements need to be provided for establishing an appropriate software-engineering environment targeted for global, loosely coupled engineering of service-oriented systems.

This thesis contributes especially to the disciplines of service-oriented software engineering and model-driven engineering. Establishment of a service-oriented software engineering (SOSE) framework necessitates four key ingredients: *1)* infrastructure services for providing a service-oriented computing environment, *2)* a tool-chain for service and business process development, *3)* a conceptual framework in form of metamodels conjoining the platforms, tools, development processes and relevant actors, and finally *4)* formal methods for providing consistency and correctness of service and business process development activities.

In our previous work we have addressed the necessary infrastructure services in the context of Pilarcos interoperability middleware [141, 139] and have given a preliminary characterization for the kinds of tools needed in a SOSE tool-chain [217]. In this thesis we have provided the conceptual framework in form of domain ontologies that describe the concepts and inter-relationships between service-engineering artifacts, such as service definitions and service descriptions, and the abstract service-oriented computing platform. This conceptual framework will be evaluated by using the software constructs, that is knowledge repositories, modelling tools, and infrastructure services, derived from these domain ontologies.

From the model-driven engineering perspective, this thesis contributes to the ongoing work on unifying ontological and linguistic metamodelling practices as well as coupling domain specific formalisms to model management infrastructure. Attaching formal semantics to MOF-based modelling framework has been addressed by various researchers. In [114] the authors introduce a method for attaching formal semantics based on Horn logic to domain specific modeling languages. A generic method for anchoring formal semantics in domain specific modeling languages has been introduced in [52]. In [199, 200] the authors apply Constructive Type Theory (CTT) for formalizing the MOF metamodelling approach. The higher-order nature of CTT allows to uniformly treat the semantics of models, metamodels, and the MOF model itself [200]. The benefit of this formalization is that correct typing corresponds to provable correct metamodels and models [199].

This thesis takes however another approach for anchoring formal semantics to a modelling domain. The approach is based on unification between ontological and linguistic metamodelling, similarly to[194]. Formal domain specific semantics is attached to the domain ontologies and their concepts, while the metamodelling technical space considers only the semantics related to linguistic metamodelling constructs, such as linguistic conformance between a terminal model and its reference model. The ontological and metamodelling disciplines are then unified by constructing a megamodel that describes the relationships between domain ontologies and their modelling space counterparts. These relationships are then realized by model repositories that provide facilities for managing the ontological and linguistic metamodelling relationships between modelling elements as well as the domain ontology relationships. The domain ontology relationships can be provided with domain specific semantics. The repository is responsible for managing the model information and keeping it consistent with respect to all the relationships.

By following this approach, we can anchor formal semantics to a modelling domain in a way that does not disrupt the current practices and theories of model-driven engineering. We can use already available model-driven engineering tools, such as diagram editors, transformation engines and so on, to implement linguistic metamodelling related activities. On the other hand, the semantics of domain concepts, such as business services, are defined solely in the ontological metamodelling space. While the linguistic metamodelling technical space is inherently attached with semantics based on graph-theory, the ontological technical space is typically associated with more expressive logical frameworks, such as first-order logic, description logic, or frame-based logics.

We claim, that in distinction to such work as [194] which use transformations or annotations to unify ontological and linguistic technical spaces, the approach used in the thesis provides a more natural unification the descriptive ontologies and descriptive models. In the future, a modelling methodology that addresses both the ontological and linguistic modelling viewpoints should be given for fully taking advantage of this approach. Until then, tools and methodologies can be used separately in their respective modelling domains (e.g. OWL for ontological modelling and UML for linguistic modelling) to provide the modelling framework with domain ontology concepts and their intentions.

The work presented is part of a constructive research about collaborative computing environments and service-oriented software engineering frameworks. The constructs to be validated include the domain ontologies metamodels presented in this thesis, and the knowledge repository generation and implementation approach. As the domain ontology as quite large and involves ingredients from various engineering disciplines, such as service-oriented computing, model-driven engineering and even method engineering (see for example [101]), the validation of this work can not be provided completely within the limits of academic thesis. Instead, a "sufficiently complete" subset of the concepts in the domain ontology will be taken through a thorough validation procedure by implementing the corresponding knowledge repositories and modelling tools. This subset includes at least service types and offers, business services, and business network models. Selection of this subset of concepts for further studies implies that also a service typing discipline associating service types with business services must be provided; this involves development of a typing discipline for the cooperation event structures. Concepts related to the service-oriented software engineering methodology and management of non-functional features using weaving models will be left for future research and out of rigorous validation procedure, for example.

The work presented here induces several other research tracks and questions to be considered later. Service-oriented software engineering methodologies provide a whole another discipline which should be studied for providing a complete service ecosystem, for example. Such work involves development of software-engineering processes suitable for service-engineering and service-based system engineering. From a global software engineering point-of-view, such processes should also be provided with explicit models. Such software engineering processes are envisioned to be composed of modelling workflows that utilize the knowledge repositories and different stake-holders involved in the engineering process. Consequently, the knowledge repositories must be provided with facilities to support such modelling workflows with transaction and notification support, for example.

Bibliography

[1] ACCIAI, L., AND BOREALE, M. XPi: a typed process calculus for XML messaging. In *7th IFIP International Conference on Formal Methods for Object-Based Distributed Systems (FMOODS)* (2005), vol. 3535 of *Lecture Notes in Computer Science*, Springer-Verlag.

[2] AFSAMANESH, H., GARITA, C., HERTZBERGER, B., AND SANTOS SILVA, V. Management of distributed information in virtual enterprises - the PRODNET approach. In *ICE'97 - International Conference on Concurrent Enterprising* (1997).

[3] ALLEN, R., AND GARLAN, D. Formalizing architectural connection. In *ICSE '94* (Los Alamitos, CA, USA, 1994), IEEE Computer Society Press, pp. 71–80.

[4] ALONSO, G., AGRAWAL, D., ABBADI, A. E., KAMATH, M., GUNTHOR, R., AND MOHAN, C. Advanced Transaction Models in Workflow Contexts. In *Twelfth International Conference on Data Engineering (ICDE '96)* (Washington, DC, USA, 1996), IEEE Computer Society, pp. 574–581.

[5] AMADIO, R. M., AND CARDELLI, L. Subtyping Recursive Types. *ACM Transactions on Programming Languages and Systems 15*, 4 (Sept. 1993), 575–631.

[6] ANDREWS, G. R. Paradigms for process interaction in distributed programs. *ACM Comput. Surv. 23*, 1 (1991), 49–90.

[7] ANTONIOU, G., AND ARIEF, M. Executable declarative business rules and their use in electronic commerce. In *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing* (New York, NY, USA, 2002), ACM Press, pp. 6–10.

[8] APACHE. *Apache Axis2 framework*, 2008. http://ws.apache.org/axis2 (04.11.2008).

[9] ARNOLD, K., AND GOSLING, J. *The Java Programming Language*. Addison-Wesley, 1996.

[10] ARNOLD, W., EILAM, T., KALANTAR, M., KONSTANTINOU, A. V., AND TOTOK, A. A. Pattern Based SOA Deployment. In *Service-Oriented Computing – ICSOC 2007* (2007), vol. 4749 of *Lecture Notes in Computer Science*, Springer, pp. 1–12.

[11] ASSMANN, U., ZSCHALER, S., AND WAGNER, G. Ontologies, Meta-models, and the Model-Driven Paradigm. In *Ontologies for Software Engineering and Software Technology*, C. Calero, F. Ruiz, and M. Piattini, Eds. Springer Berlin / Heidelberg, 2006, pp. 249–273.

[12] ATKINSON, C., AND KÜHNE, T. Model-driven development: A metamodeling foundation. *IEEE Softw. 20*, 5 (2003), 36–41.

[13] BALDONI, M., BAROGLIO, C., MARTELLI, A., AND PATTI, V. A priori conformance verification for guaranteeing interoperability in open environments. In *International Conference on Service-Oriented Computing - ICSOC 2006* (2006), vol. 4294 of *Lecture Notes in Computer Science*, Springer, pp. 339–351.

[14] BALDONI, R., CONTENTI, M., AND VIRGILLITO, A. The Evolution of Publish/Subscribe Communication Systems. In *Future Directions in Distributed Computing: Reserach and Position Papers* (2002), vol. 2584 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 137–141.

[15] BARBERO, M., JOUAULT, F., , AND BÉZIVIN, J. Model Driven Management of Complex Systems: Implementing the Macroscope's vision. In *15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2008)* (2008), IEEE, pp. 277–286.

[16] BARBERO, M., JOUAULT, F., GRAY, J., AND BÉZIVIN, J. A Practical Approach to Model Extension. In *ECMDA-FA 2007* (2007), vol. 4530 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 32–42.

[17] BARBUCEANU, M., AND LO, W.-K. A multi-attribute utility theoretic negotiation architecture for electronic commerce. In *Proceedings of the fourth international conference on Autonomous agents* (2000), ACM Press, pp. 239–246.

[18] BEITZ, A., AND BEARMAN, M. An ODP Trading Service for DCE. In *First International Workshop on Services in Distributed and Networked Environments* (1994), IEEE, pp. 42–49.

[19] BENATALLAH, B., CASATI, F., AND TOUMANI, F. Web service conversation modeling: A cornerstone for e-business automation. *IEEE Internet Computing 8*, 1 (2004), 46–54.

[20] BENGUARIA, G., LARRUCEA, X., ELVESAETER, B., NEPLE, T., BEARDSMORE, A., AND FRIESS, M. A Platform Independent Model for Service Oriented Architectures. In *Enterprise Interoperability: New Challenges and Approaches* (Apr. 2007), G. Doumeingts, J. MÃijller, G. Morel, and B. Vallespir, Eds., Springer, pp. 23–32.

[21] BERARDI, D., CALVANESE, D., GIACOMO, G. D., LENZERINI, M., AND MECELLA, M. Automatic Composition of E-services That Export Their Behavior. In *Service-Oriented Computing - ICSOC 2003* (2003), vol. 2910 of *LNCS*, Springer-Verlag GmbH, pp. 43–58.

[22] BERARDI, D., GIACOMO, G. D., LENZERINI, M., MECELLA, M., AND CALVANESE, D. Synthesis of underspecified composite e-services based on automated reasoning. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing* (New York, NY, USA, 2004), ACM Press, pp. 105–114.

[23] BERNERS-LEE, T., FIELDING, R., AND MASINTER, L. Uniform Resource Identifiers (URI): Generic Syntax, 1998. RFC 2396.

[24] BERNERS-LEE, T., FIELDING, R., AND MASINTER, L. Uniform Resource Identifiers (URI): Generic Syntax, 1998. RFC 2396.

[25] BERNSTEIN, P. A. Repositories and object oriented databases. *SIGMOD Rec. 27*, 1 (1998), 88–96.

[26] BÉZIVIN, J. On the unification power of models. *Software and Systems Modeling 4*, 2 (May 2005), 171–188.

[27] BÉZIVIN, J., BARBERO, M., AND JOUAULT, F. On the applicability scope of model driven engineering. In *Model-Based Methodologies for Pervasive and Embedded Software (MOMPES'07)* (2007), IEEE, pp. 3–7.

[28] BÉZIVIN, J., BÜTTNER, F., GOGOLLA, M., JOUAULT, F., KURTEV, I., AND LINDOW, A. Model Transformations? Transformation Models! In *Model Driven Engineering Languages and Systems* (2006), vol. 4199 of *Lecture Notes in Computer Science*, Springer, pp. 440–453.

[29] BÉZIVIN, J., JOUAULT, F., ROSENTHAL, P., AND VALDURIEZ, P. Modeling in the Large and Modeling in the Small. In *Model Driven Architecture* (2005), vol. 3599 of *Lecture Notes in Computer Science*, Springer, pp. 33–46.

[30] BÉZIVIN, J., JOUAULT, F., AND TOUZET, D. Principles, standards and tools for model engineering. In *ICECCS '05: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 28–29.

[31] BÉZIVIN, J., JOUAULT, F., AND VALDURIEZ, P. On the Need for Megamodels. In *OOPSLA workshop on Best Practices for Model-Driven Software Development* (2004).

[32] BIDDLE, B. J., AND THOMAS, E. J., Eds. *Role Theory: Concepts and Research*. John Wiley & Sons, 1966.

[33] BIERMANN, E., EHRIG, K., KÖHLER, C., KUHNS, G., TAENTZER, G., AND WEISS, E. EMF Model Refactoring based on Graph Transformation Concepts. *Electronic Communications of the EASST 3* (2006). ISSN 1863-2122. http://eceasst.cs.tu-berlin.de/index.php/eceasst/issue/view/3.

[34] BIRRELL, A. D., AND NELSON, B. J. Implementing remote procedure calls. *ACM Trans. Comput. Syst. 2*, 1 (1984), 39–59.

[35] BLANC, X., GERVAIS, M.-P., AND SRIPLAKICH, P. Model Bus: Towards the Interoperability of Modelling Tools. In *Model Driven Architecture* (2005), vol. 3599 of *Lecture Notes in Computer Science*, Springer, pp. 17–32.

[36] BLOOM, S. L., AND ÉSIK, Z. The equational theory of regular words. *Information and Computation 197* (2005), 55–89.

[37] BOCKISCH, C., HAUPT, M., MEZINI, M., AND OSTERMANN, K. Virtual machine support for dynamic join points. In *Proceedings of the 3rd international conference on Aspect-oriented software development* (2004), ACM Press, pp. 83–92.

[38] BRANT, J., FOOTE, B., JOHNSON, R. E., AND ROBERTS, D. Wrappers to the rescue. In *ECOOP'98* (1998), vol. 1445 of *LNCS*, Springer-Verlag, pp. 396–417.

[39] BROCKMANS, S., COLOMB, R. M., HAASE, P., KENDALL, E. F., WALLACE, E. K., WELTY, C., AND XIE, G. T. A Model Driven Approach for Building OWL DL and OWL Full Ontologies. In *International Conference on the Semantic Web (ISWC)* (2006), vol. 4273 of *Lecture Notes in Computer Science*, Springer, pp. 187–200.

[40] BROWN, A., LANEVE, C., AND MEREDITH, G. PiDuce: A process calculus with native XML datatypes. In *2nd International Workshop on Web Services and Formal Methods (WS-FM 2005)* (Versailles, France, Sept. 2005). Proceedings to be published in the Springer LNCS series.

[41] BRUNELIÉRE, H., ALLILAIRE, F., BÉZIVIN, J., AND JOUAULT, F. Global model management in eclipse gmt/am3. In *Eclipse Technology eXchange workshop (eTX) at the ECOOP 2006 Conference* (2006).

[42] BULTAN, T., FU, X., HULL, R., AND SU, J. Conversation specification: a new approach to design and analysis of e-service composition. In *Proceedings of the twelfth international conference on World Wide Web* (2003), ACM Press, pp. 403–410.

[43] BUSCHMANN, F., MEUNIER, R., ROHNERT, H., SOMMERLAD, P., AND STAL, M. *Pattern-Oriented Software Architecture: A System of Patters*, vol. 1. Wiley, July 2001.

[44] CALDER, M., KOLBERG, M., MAGILL, E. H., AND REIFF-MARGANIEC, S. Feature interaction: a critical review and considered forecast. *Computer Networks 41*, 1 (Jan. 2003), 115–141.

[45] CAMARINHA-MATOS, L. M., AND AFSARMANESH, H. Service federation in virtual organisations. In *PROLAMAT'01* (Budabest, Hungary, Nov. 2001).

[46] CAMARINHA-MATOS, L. M., AND AFSARMANESH, H. Collaborative networks: Value creation in knowledge society. In *PROLAMAT 2006, Knowledge Enterprise: Intelligent Strategies in Product Design, Manufacturing, and Management* (2006), vol. 207, pp. 26–40.

[47] CAMARINHA-MATOS, L. M., AND AFSARMANESH, H. Creation of virtual organizations in a breeding environment. In *INCOM 2006, 12th IFAC Symposium on Information Control Problems Manufacturing* (2006), Elsevier Science. To appear. http://www.uninova.pt/~cam/ev/INCOM06.PDF (29.09.2006).

[48] CANAL, C., FUENTES, L., PIMENTEL, E., TROYA, J. M., AND VALLECILLO, A. Extending CORBA Interfaces with Protocols. *The Computer Journal 44*, 5 (Oct. 2001), 448–462.

[49] CANAL, C., PIMENTEL, E., AND TROYA, J. M. Compatibility and inheritance in software architectures. *Science of Computer Programming 41*, 2 (Oct. 2001), 105–138.

[50] CARDOSO, J., BOSTROM, R. P., AND SHETH, A. Workflow Management Systems and ERP Systems: Differences, Commonalities, and Applications. *Inf. Tech. and Management 5*, 3-4 (2004), 319–338.

[51] CHAKI, S., RAJAMANI, S. K., AND REHOF, J. Types as models: model checking message-passing programs. In *POPL '02: Proceedings of the 29th symposium on Principles of programming languages* (New York, NY, USA, 2002), ACM Press, pp. 45–57.

[52] CHEN, K., SZTIPANOVITS, J., AND NEEMA, S. Toward a semantic anchoring infrastructure for domain-specific modeling languages. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software* (New York, NY, USA, 2005), ACM Press, pp. 35–43.

[53] CHINNICI, R., MOREAU, J.-J., RYMAN, A., AND WEERAWARANA, S. *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Langauge*, 2.0 ed. W3C, Aug. 2005. W3C Wordking Draft.

[54] CHOPRA, A. K., AND SINGH, M. P. Producing Compliant Interactions: Conformance, Coverage, and Interoperability. In *Declarative Agent Languages and Technologies IV* (2006), vol. 4327 of *Lecture Notes in Computer Science*, Springer, pp. 1–15.

[55] CHRISTENSEN, E., CURBERA, F., MEREDITH, G., AND WEERAWARANA, S. *Web Services Description Language (WSDL) 1.1*, 1.1 ed. W3C, Mar. 2001.

[56] CLARKE, E. M., AND WING, J. M. Formal methods: state of the art and future directions. *ACM Comput. Surv. 28*, 4 (1996), 626–643.

[57] The C♯ Language, 2005. http://msdn.microsoft.com/vcsharp/programming/language/.

[58] CZARNECKI, K., AND HELSEN, S. Classification of Model Transformation Approaches. In *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture* (2003).

[59] DAMIAN, D., AND MOITRA, D. Guest editors' introduction: Global software development: How far have we come? *IEEE Softw. 23*, 5 (2006), 17–19.

[60] DAMODARAN, S. B2B integration over the Internet with XML: RosettaNet successes and challenges. In *13th international World Wide Web conference on Alternate track papers & posters* (New York, NY, USA, 2004), ACM, pp. 188–195.

[61] DE NICOLA, R., AND VAANDRAGER, F. Action versus state based logics for transition systems. In *Proceedings of the LITP spring school on theoretical computer science on Semantics of systems of concurrent processes* (1990), Springer-Verlag New York, Inc., pp. 407–419.

[62] DEL FABRO, M., BÉZIVIN, J., AND VALDURIEZ, P. Weaving Models with the Eclipse AMW plugin. In *Eclipse Modeling Symposium, Eclipse Summit Europe* (2006). http://www.eclipsecon.org/summiteurope2006/presentations/ESE2006-EclipseModelingSymposium2_WeavingModels.pdf.

[63] DEREMER, F., AND KRON, H. Programming-in-the large versus programming-in-the-small. In *Proceedings of the international conference on Reliable software* (1975), pp. 114–121.

[64] DIDONET DEL FABRO, M., AND JOUAULT, F. Model Transformation and Weaving in the AMMA Platform. In *Workshop on Generative and Transformational Techniques in Software Engineering (GTTSE)* (2005), pp. 71–77. Retrievable from http://www.lina.sciences.univ-nantes.fr/Publications/2005/DJ05 (30.01.2007).

[65] DORAN, J. E., FRANKLIN, S., JENNINGS, N. R., AND NORMAN, T. J. On cooperation in multi-agent systems. *Knowl. Eng. Rev. 12*, 3 (1997), 309–314.

[66] DURR, P., STAIJEN, T., BERGMANS, L., AND AKSIT, M. Reasoning about semantic conflicts between aspects. *2nd European Interactive Workshop on Aspects in Software (EIWAS2̌01905), Sept* (2005).

[67] Eclipse - an open development platform. http://www.eclipse.org/, 2008.

[68] Atlas Model Weaver (AMW) website. http://www.eclipse.org/gmt/amw/, June 2008.

[69] Eclipse Modeling Framework website. http://www.eclipse.org/modeling/emf/, 2008.

[70] Eclipse Model To Text (M2T) project. http://www.eclipse.org/modeling/m2t, Oct. 2008.

[71] Model To Text (M2T). http://www.eclipse.org/modeling/m2t, June 2008.

[72] Eclipse Model Driven Development integration (MDDi) website. http://www.eclipse.org/mddi/, 2008.

[73] New issues in object interoperability. In *Object-Oriented Technology: ECOOP 2000 Workshops, Panels, and Posters* (2000), A. M. J. Malenfant, S. Moisan, Ed., vol. 1964 of *LNCS*, Springer-Verlag GmbH.

[74] ERASALA, N., YEN, D. C., AND RAJKUMAR, T. M. Enterprise application integration in the electronic commerce world. *Comput. Stand. Interfaces 25*, 2 (2003), 69–82.

[75] FANG, J., HU, S., AND HAN, Y. A service interoperability assessment model for service composition. In *IEEE International Conference on Services Computing (SCC 2004)* (2004), IEEE, pp. 153–158.

[76] FATOOHI, R., GUNWANI, V., WANG, Q., AND ZHENG, C. Performance evaluation of middleware bridging technologies. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2000), IEEE, pp. 34–39.

[77] FAVRE, J. Towards a Basic Theory to Model Model Driven Engineering. In *3rd Workshop in Software Model Engineering in conjunction with UML2004, WiSME* (2004).

[78] FAVRE, J.-M. Foundations of Model (Driven) (Reverse) Engineering : Models - Episode I: Stories of The Fidus Papyrus and of The Solarus. In *Language Engineering for Model-Driven Software Development* (2004).

[79] FEKETE, A., GREENFIELD, P., KUO, D., AND JANG, J. Transactions in loosely coupled distributed systems. In *CRPITS'17: Proceedings of the Fourteenth Australasian database conference on Database technologies 2003* (Darlinghurst, Australia, Australia, 2003), Australian Computer Society, Inc., pp. 7–12.

[80] FINDLER, R. B., FLATT, M., AND FELLEISEN, M. Semantic Casts: Contracts and Structural Subtyping in a Nominal World. In *ECOOP* (2004), pp. 364–388.

[81] FOSTER, I., KESSELMAN, C., NICK, J., AND TUECKE, S. Grid services for distributed system integration. *Computer 35*, 6 (jun 2002), 37–46.

[82] FRANCE, R., RAY, I., GEORG, G., AND GHOSH, S. An Aspect-Oriented Approach to Early Design Modeling. *IEE Proceedings-Software 151*, 4 (2004), 173–185.

[83] FRANCE, R. B., KIM, D.-K., GHOSH, S., AND SONG, E. A UML-Based Pattern Specification Technique. *IEEE Trans. Softw. Eng. 30*, 3 (2004), 193–206.

[84] FRANKEL, D. S. *Model Driven Architecture: Applying MDA to Enterprise Computing*. OMG Press, 2003.

[85] FRANKEL, D. S. An MDA Manifesto. *MDA Journal* (May 2004). http://www.bptrends.com (01.09.2006).

[86] FU, G., SHAO, J., EMBURY, S., GRAY, W., AND LIU, X. A framework for business rule presentation. In *12th International Workshop on Database and Expert Systems Applications* (2001), pp. 922–926.

[87] GALLAIRE, H., MINKER, J., AND NICOLAS, J.-M. Logic and databases: A deductive approach. *ACM Comput. Surv. 16*, 2 (1984), 153–185.

[88] GARLAN, D., ALLEN, R., AND OCKERBLOOM, J. Architectural mismatch or why it's hard to build systems out of existing parts. In *ICSE '95* (New York, NY, USA, 1995), ACM Press, pp. 179–185.

[89] GASEVIC, D., KAVIANI, N., AND HATALA, M. On Metamodeling in Megamodels. In *Model Driven Engineering Languages and Systems* (2007), vol. 4735 of *Lecture Notes in Computer Science*, Springer, pp. 91–105.

[90] GAY, S., AND HOLE, M. Types and Subtypes for Client-Server Interactions. *Lecture Notes in Computer Science 1576* (1999), 74–90.

[91] GEORGAKOPOULOS, D., HORNICK, M., AND SHETH, A. An overview of workflow management: from process modeling to workflow automation infrastructure. *Distrib. Parallel Databases 3*, 2 (1995), 119–153.

[92] GOTH, G. Critics Say Web Services Need a REST. *IEEE Distributed Systems Online 5*, 12 (2004), 1.

[93] GRAY, J., BAPTY, T., NEEMA, S., SCHMIDT, D. C., GOKHALE, A., AND NATARAJAN, B. An Approach for Supporting Aspect-Oriented Domain Modeling. In *Generative Programming and Component Engineering* (2003), vol. 2830 of *Lecture Notes in Computer Science*, Springer-Verlag.

[94] GREFEN, P., ABERER, K., HOFFNER, Y., AND LUDWIG, H. CrossFlow: Cross-Organizational Workflow Management in Dynamic Virtual Enterprises. *International Journal of Computer Systmes Sciences and Engineering 15*, 5 (2000), 277–290.

[95] GROSOF, B. N., LABROU, Y., AND CHAN, H. Y. A declarative approach to business rules in contracts: courteous logic programs in XML. In *EC '99: Proceedings of the 1st ACM conference on Electronic commerce* (New York, NY, USA, 1999), ACM Press, pp. 68–77.

[96] GRUDIN, J. Computer-Supported Cooperative Work: History and Focus. *Computer 27*, 5 (1994), 19–26.

[97] HAMADI, R., AND BENATALLAH, B. A Petri net-based model for web service composition. In *CRPITS'17: Proceedings of the Fourteenth Australasian database conference on Database technologies 2003* (Darlinghurst, Australia, Australia, 2003), Australian Computer Society, Inc., pp. 191–200.

[98] HANSON, J. E., NANDI, P., AND KUMARAN, S. Conversation Support for Business Process Integration. In *EDOC '02: Proceedings of the Sixth International ENTERPRISE DISTRIBUTED OBJECT COMPUTING Conference (EDOC'02)* (Washington, DC, USA, 2002), IEEE Computer Society, p. 65.

[99] HAY, J. D., AND ATLEE, J. M. Composing features and resolving interactions. In *SIGSOFT '00/FSE-8: Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering* (New York, NY, USA, 2000), ACM, pp. 110–119.

[100] HENDERSON-SELLERS, B. Method engineering for OO systems development. *Commun. ACM 46*, 10 (2003), 73–78.

[101] HENDERSON-SELLERS, B., FRANCE, R., GEORG, G., AND REDDY, R. A method engineering approach to developing aspect-oriented modelling processes based on the OPEN process framework. *Inf. Softw. Technol. 49*, 7 (2007), 761–773.

[102] Hibernate – Relational Persistence for Java and .NET. http://www.hibernate.org/, Oct. 2008.

[103] HONDA, K., VASCONCELOS, V. T., AND KUBO, M. Language primitives and type discipline for structured communication-based programming. In *Proceedings of the 7th European Symposium on Programming* (1998), Springer-Verlag, pp. 122–138.

[104] HOPCROFT, J. E., MOTWANI, R., AND ULLMAN, J. D. *Introduction to Automata Theory, Languages, and Computation*, 2 ed. Addison Wesley, 2001.

[105] HOSOYA, H., AND PIERCE, B. C. XDuce: A statically typed XML processing language. *ACM Trans. Inter. Tech. 3*, 2 (2003), 117–148.

[106] HOSOYA, H., VOUILLON, J., AND PIERCE, B. C. Regular expression types for XML. *ACM Trans. Program. Lang. Syst. 27*, 1 (2005), 46–90.

[107] HULL, R., AND SU, J. Tools for composite web services: a short overview. *SIGMOD Rec. 34*, 2 (2005), 86–95.

[108] ISO. *ISO/IEC FDIS 14769: Type Repository Function (draft)*. ISO/IEC, 1999.

[109] ISO/IEC JTC1. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing. ODP Naming framework*, 1996. Committee Draft.

[110] ISO/IEC JTC1. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing. Reference Model of Open Distributed Processi ng. Part 3: Architecture*, 1996. IS10746-3.

[111] ISO/IEC JTC1. *ISO/IEC 13235: Information Technology – Open Distributed Processing. ODP Trading function.*, 1997.

[112] ISO/IEC JTC1/SC7. *ISO/IEC 10746-3: Information technology – Open Distributed Processing – Reference model: Architect ure*, Sept. 1996.

[113] ISO/IEC JTC1/SC7. *ISO/IEC 10746-1: Information technology – Open Distributed Processing – Reference model: Overview*, Dec. 1998.

[114] JACKSON, E. K., AND SZTIPANOVITS, J. Towards a formal foundation for domain specific modeling languages. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software* (New York, NY, USA, 2006), ACM Press, pp. 53–62.

[115] Jena: A Semantic Web Framework for Java. http://jena.sourceforge.net, June 2008.

[116] JENNINGS, N. R. An agent-based approach for building complex software systems. *Commun. ACM 44*, 4 (2001), 35–41.

[117] JENNINGS, N. R., FARATIN, P., NORMAN, T. J., O'BRIEN, P., AND ODGERS, B. Autonomous Agents for Business Process Management. *Int. Journal of Applied Artificial Intelligence 14*, 2 (2000), 145–189.

[118] JENNINGS, N. R., NORMAN, T. J., AND FARATIN, P. ADEPT: an agent-based approach to business process management. *SIGMOD Rec. 27*, 4 (1998), 32–39.

[119] JHA, S., PALSBERG, J., AND ZHAO, T. Efficient Type Matching. *Lecture Notes in Computer Science 2303* (2002), 187–206.

[120] JOSSIC, A., DIDONET DEL FABRO, M., LERAT, J., BÉZIVIN, J., AND JOUAULT, F. Model Integration with Model Weaving: a Case Study in System Architecture. In *International Conference on Systems Engineering and Modeling (ICSEM'07)* (Mar. 2007), IEEE, pp. 79–84.

[121] JOUAULT, F., AND BÉZIVIN, J. KM3: A DSL for Metamodel Specification. In *Formal Methods for Open Object-Based Distributed Systems* (2006), vol. 4037 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 171–185.

[122] JOUAULT, F., AND KURTEV, I. On the architectural alignment of ATL and QVT. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing* (New York, NY, USA, 2006), ACM Press, pp. 1188–1195.

[123] JOUAULT, F., AND KURTEV, I. Transforming Models with ATL. In *Satellite Events at the MoDELS 2005 Conference* (2006), vol. 3844 of *Lecture Notes in Computer Science*, Springer, pp. 128–138.

[124] KAGAL, L., AND FININ, T. Modeling Communicative Behavior Using Permissions and Obligations. In *Agent Communication: International Workshop on Agent Communication, AC 2004* (2005), no. 3396 in LNAI, Springer-Verlag Berlin Heidelberg, pp. 120–133.

[125] KAVANTZAS, N., BURDETT, D., RITZINGER, G., FLETCHER, T., LAFON, Y., AND BARRETO, C. *Web Services Choreography Description language*. W3C, Nov. 2005. http://www.w3.org/TR/ws-cdl-10/, W3C Candidate Recommendation.

[126] KELLER, A., KAR, G., LUDWIG, H., DAN, A., AND HELLERSTEIN, J. Managing dynamic services: a contract based approach to a conceptual architecture. In *Network Operations and Management Symposium* (2002), IFIP, IEEE, pp. 513–528.

[127] KERSCHBERG, L. The role of loose coupling in expert database system architectures. In *5th International Conference on Data Engineering* (Feb. 1989), IEEE, pp. 255–256.

[128] KINDLER, E. Safety and liveness properties: A survey. *Bulletin of the European Association for Theoretical Computer Science 53* (1994), 268–272.

[129] KOBAYASHI, N. A partially deadlock-free typed process calculus. *ACM Trans. Program. Lang. Syst. 20*, 2 (1998), 436–482.

[130] KOBAYASHI, N., PIERCE, B. C., AND TURNER, D. N. Linearity and the pi-calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1996), ACM Press, pp. 358–371.

[131] KÖLLMANN, C., KUTVONEN, L., LININGTON, P., AND SOLBERG, A. An Aspect-Oriented Approach to Manage QoS Dependability Dimensions in Model Driven Development. In *The 3rd International Workshop on Model-Driven Enterprise Information Systems (MDEIS 2007)* (2007).

[132] KONSTANTAS, D. Object oriented interoperability. In *ECOOP '93 - Object-Oriented Programming: 7th European Conference* (1993), vol. 707 of *LNCS*, Springer-Verlag GmbH, pp. 80–102.

[133] KOTOK, A., AND WEBBER, D. R. R. *ebXML: The New Global Standard for Doing Business Over the Internet*. New Riders, Boston, 2001.

[134] KÜHNE, T. Matters of (Meta-) Modeling. *Software and Systems Modeling (SoSyM) 5*, 17 (Dec. 2006), 369–385.

[135] KUMAR, V. Algorithms for constraint-satisfaction problems: a survey. *AI Mag. 13*, 1 (1992), 32–44.

[136] KURTEV, I., BÉZIVIN, J., AND AKSIT, M. Technological Spaces: An Initial Appraisal, 2002. http://www.scientificcommons.org/27172017 (05.06.2008).

[137] KUTVONEN, L. *Trading services in open distributed environments*. PhD thesis, University of Helsinki, 1998.

[138] KUTVONEN, L. TUBE - trust based on evidence, Apr. 2004. http://www.cs.helsinki.fi/Lea.Kutvonen/tube/.

[139] KUTVONEN, L., METSO, J., AND RUOHOMAA, S. From trading to eCommunity management: Responding to social and contractual challenges. *Information Systems Frontiers (ISF) - Special Issue on Enterprise Services Computing: Evolution and Challenges 9*, 2–3 (July 2007), 181–194.

[140] KUTVONEN, L., METSO, J., AND RUOKOLAINEN, T. Inter-enterprise collaboration management in dynamic business networks. In *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences, CoopIS, DOA, and ODBASE* (Agia Napa, Cyprus, Nov. 2005), vol. 3760 of *Lecture Notes in Computer Science*.

[141] KUTVONEN, L., RUOKOLAINEN, T., AND METSO, J. Interoperability middleware for federated business services in web-Pilarcos. *International Journal in Enterprise Information Systems 3*, 1 (Jan. 2007), 1–21.

[142] KUTVONEN, L., RUOKOLAINEN, T., METSO, J., AND HAATAJA, J. Interoperability middleware for federated enterprise applications in web-Pilarcos. In *Interoperability of Enterprise Software and Applications* (Dec. 2005), Springer-Verlag. ISBN: 1-84628-151-2.

[143] KUTVONEN, L., RUOKOLAINEN, T., RUOHOMAA, S., AND METSO, J. Service-Oriented Middleware for Managing Inter-Enterprise Collaborations. In *Global Implications of Modern Enterprise Information Systems: Technologies and Applications*, A. Gunasekaran, Ed. IGI Global, Oct. 2008, pp. 208–241.

[144] LAMB, D. A. Idl: sharing intermediate representations. *ACM Trans. Program. Lang. Syst. 9*, 3 (1987), 297–318.

[145] LARSEN, K., AND THOMSEN, B. A modal process logic. In *Third Annual Symposium on Logic in Computer Science* (1988), IEEE, pp. 203–210.

[146] LAZCANO, A., ALONSO, G., SCHULDT, H., AND SCHULER, C. The WISE approach to Electronic Commerce. *Computer Systems Science and Engineering 15*, 5 (2000), 345–357.

[147] LEE, J., SIAU, K., AND HONG, S. Enterprise integration with erp and eai. *Commun. ACM 46*, 2 (2003), 54–60.

[148] LEROY, X. *The Objective Caml system release 3.08*. INRIA. http://caml.inria.fr/distrib/ocaml-3.08/ocaml-3.08-refman.pdf.

[149] LI, J., ZHU, H., AND PU, G. Conformance Validation between Choreography and Orchestration. In *TASE '07: Proceedings of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 473–482.

[150] LINTHICUM, D. S. *B2B Application Integration: e-Business–Enable Your Enterprise*. Addison-Wesley, 2001.

[151] LISKOV, B. H., AND WING, J. M. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst. 16*, 6 (1994), 1811–1841.

[152] LIU, Y., YU, F., SU, S. Y. W., AND LAM, H. A cost-benefit evaluation server for decision support in e-business. *Decision Support Systems 36*, 1 (2003), 81–97.

[153] LOMUSCIO, A. R., WOOLDRIDGE, M., AND JENNINGS, N. R. A classification scheme for negotiation in electronic commerce. *Lecture Notes in Computer Science 1991* (Jan. 2001), 19–??

[154] LÓPEZ-SANZ, M., ACUÒA, C. J., CUESTA, C. E., AND MARCOS, E. Modelling of Service-Oriented Architectures with UML. *Electron. Notes Theor. Comput. Sci. 194*, 4 (2008), 23–37.

[155] LUCCHI, R., AND MAZZARA, M. A pi-calculus based semantics for WS-BPEL. *Journal of Logic and Algebraic Programming (JLAP)*, 1 (Jan. 2007), 96–118.

[156] LUCKHAM, D., AND VERA, J. An event-based architecture definition language. *IEEE Transactions on Software Engineering 21*, 9 (sep 1995), 717–734.

[157] LUDWIG, H., KELLER, A., DAN, A., KING, R., AND FRANCK, R. A service level agreement language for dynamic electronic services. *Electronic Commerce Research 3*, 1-2 (2003), 43–59.

[158] MAEDCHE, A. Ontology Learning for the Semantic Web. *Intelligent Systems 16*, 2 (Mar. 2001), 72–79.

[159] MAES, P. Concepts and experiments in computational reflection. In *Conference proceedings on Object-oriented programming systems, languages and applications* (1987), ACM Press, pp. 147–155.

[160] MAGEE, J., DULAY, N., AND KRAMER, J. Structuring parallel and distributed programs. *Software Engineering 8*, 2 (Mar. 1993), 73–82.

[161] MCILRAITH, S., SON, T., AND ZENG, H. Semantic Web services. *Intelligent Systems 16*, 2 (2001), 46–53.

[162] MECELLA, M., AND PERNICI, B. Designing wrapper components for e-services in integrating heterogeneous systems. *The VLDB Journal 10*, 1 (2001), 2–15.

[163] MEDJAHED, B., BENATALLAH, B., BOUGUETTAYA, A., NGU, A. H. H., AND ELMAGARMID, A. K. Business-to-business interactions: issues and enabling technologies. *The VLDB Journal 12*, 1 (2003), 59–85.

[164] MEJÍA, R., LÓPEZ, A., AND MOLINA, A. Experiences in developing collaborative engineering environments: An action research approach. *Comput. Ind. 58*, 4 (2007), 329–346.

[165] METSO, J., AND KUTVONEN, L. Managing Virtual Organizations with Contracts. In *Workshop on Contract Architectures and Languages (CoALa2005)* (Enschede, The Netherlands, Sept. 2005). To be published.

[166] MILNER, R., PARROW, J., AND WALKER, D. A Calculus of Mobile Processes, Part I/II. *Journal of Information and Computation 100* (Sept. 1992), 1–77.

[167] MILOSEVIC, Z., J"SANG, A., DIMITRAKOS, T., AND PATTON, M. A. Discretionary enforcement of electronic contracts. In *EDOC '02: Proceedings of the Sixth International Enterprise Distributed Object Computing Conference (EDOC'02)* (Washington, DC, USA, 2002), IEEE Computer Society, p. 39.

[168] MODELling solution for softWARE systems (MODELWARE). http://www.modelware-ist.org, 2006. IST Project 511731 (completed).

[169] MUENKE, M., LAMERSDORF, W., CHRISTIANSEN, B. O., AND MUELLER-JONES, K. Type management: A key to software reuse in open distributed systems. In *1st International Enterprise Distributed Object Computing Conference (EDOC '97)* (1997), pp. 78–89.

[170] MURATA, M., LEE, D., MANI, M., AND KAWAGUCHI, K. Taxonomy of XML schema languages using formal language theory. *ACM Trans. Inter. Tech. 5*, 4 (2005), 660–704.

[171] NAICS. *North American Industry Classification System (NAICS)*. http://www.naics.com.

[172] NAIK, M., AND PALSBERG, J. A Type System Equivalent to a Model Checker. In *Programming Languages and Systems: 14th European Symposium on Programming (ESOP)* (Berlin / Heidelberg, 2005), vol. 3444 of *Lecture Notes in Computer Science*, Springer, pp. 374–388.

[173] NAJM, E., NIMOUR, A., AND STEFANI, J.-B. *Formal methods for distributed processing: a survey of object-oriented approaches*. Cambridge University Press, 2001, ch. Behavioural typing for objects and process calculi, pp. 281–301.

[174] NARDI, D., AND BRACHMAN, R. *Description Logic Handbook*. Cambridge University Press, 2002, ch. An Introduction to Description Logics, pp. 5–44.

[175] NAYAK, N., BHASKARAN, K., AND DAS, R. Virtual enterprises: building blocks for dynamic e-business. In *Proceedings of the workshop on Information technology for virtual enterprises* (2001), IEEE Computer Society, pp. 80–87.

[176] OBJECT MANAGEMENT GROUP. *CORBA Trading Object Service Specification*, May 2000. http://cgi.omg.org/docs/formal/00-06-27.pdf.

[177] OBJECT MANAGEMENT GROUP. *CORBA 3.0 - OMG IDL Syntax and Semantics chapter*, jul 2002.

[178] OBJECT MANAGEMENT GROUP. *Common Object Request Broker Architecture (CORBA) v3.0.3*, Mar. 2004. OMG Document formal/04-03-01.

[179] OBJECT MANAGEMENT GROUP. *Unified Modeling Language: Superstructure*, 2 ed., Aug. 2005.

[180] OBJECT MANAGEMENT GROUP. *Business Process Modeling Notation (BPMN) Specification*, 1.0 ed., Feb. 2006. Final Adopted Specification; dtc/06-02-01.

[181] OBJECT MANAGEMENT GROUP. *Meta Object Facility (MOF) Core Specification*, 2.0 ed., Jan. 2006. OMG Available Specification – formal/06-01-01.

[182] OBJECT MANAGEMENT GROUP. *Unified Modeling Language: Infrastructure*, 2 ed., Mar. 2006.

[183] OBJECT MANAGEMENT GROUP. *Ontology Definition Metamodel*, Sept. 2007. OMG Adopted Specification (ptc/2007-09-09).

[184] OMG. *MOF 2.0 / XMI Mapping Specification, v2.1.1*, 2007.

[185] ORFALI, R., HARKEY, D., AND EDWARDS, J. *Instant CORBA*. John Wiley & Sons, Inc., 1997.

[186] OSMAN, N., ROBERTSON, D., AND WALTON, C. Run-time model checking of interaction and deontic models for multi-agent systems. In *5th International joint conference on Autonomous Agents and Multiagent Systems (AAMAS'06)* (New York, NY, USA, 2006), ACM Press, pp. 238–240.

[187] O'SULLIVAN, J., EDMOND, D., AND TER HOFSTEDE, A. What's in a Service? Towards Accurate Description of Non-Functional Service Properties. *Distributed and Parallel Databases 12* (2002), 117–133.

[188] OWL-S COALITION. *OWL-S 1.1 Release*, Nov. 2004.

[189] PALSBERG, J., AND ZHAO, T. Efficient and flexible matching of recursive types. In *Logic in Computer Science* (2000), pp. 388–398.

[190] PAN, J., CRANEFIELD, S., AND CARTER, D. A lightweight ontology repository. In *AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems* (2003), ACM Press, pp. 632–638.

[191] PAPAZOGLOU, M. P., AND GEORGAKOPOULOS, D. Service-oriented computing. *Commun. ACM 46*, 10 (2003), 24–28.

[192] PAPAZOGLOU, M. P., AND HEUVEL, W.-J. Service oriented architectures: approaches, technologies and research issues. *The VLDB Journal 16*, 3 (2007), 389–415.

[193] PAPAZOGLOU, M. P., AND VAN DEN HEUVEL, W.-J. Business process development life cycle methodology. *Commun. ACM 50*, 10 (2007), 79–85.

[194] PARREIRAS, F. S., STAAB, S., AND WINTER, A. On marrying ontological and metamodeling technical spaces. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering* (New York, NY, USA, 2007), ACM, pp. 439–448.

[195] PEER, J. Bringing together semantic web and web services. In *The Semantic Web - ISWC 2002: First International Semantic Web Conference* (2002), vol. 2342 of *Lecture Notes in Computer Science*, Springer-Verlag Heidelberg, pp. 279–291.

[196] PETERSON, J. L. Petri Nets. *ACM Comput. Surv. 9*, 3 (1977), 223–252.

[197] PIERCE, B. C. *Types and Programming Languages*. The MIT Press, Cambridge, MA, 2002.

[198] PIERCE, B. C., AND SANGIORGI, D. Typing and Subtyping for Mobile Processes. In *Proceedings 8th IEEE Logics in Computer Science* (Montreal, Canada, June 1993), pp. 376–385.

[199] POERNOMO, I. A Type Theoretic Framework for Formal Metamodelling. In *Architecting Systems with Trustworthy Components* (2006), R. Reaussner, Ed., Lecture Notes in Computer Science, Springer Berlin / Heidelberg, pp. 262–298.

[200] POERNOMO, I. The meta-object facility typed. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing* (New York, NY, USA, 2006), ACM Press, pp. 1845–1849.

[201] POHL, K., AND METZGER, A. Variability management in software product line engineering. In *ICSE '06: Proceeding of the 28th international conference on Software engineering* (New York, NY, USA, 2006), ACM, pp. 1049–1050.

[202] POPOVICI, A., GROSS, T., AND ALONSO, G. Dynamic weaving for aspect-oriented programming. In *Proceedings of the 1st international conference on Aspect-oriented software development* (2002), ACM Press, pp. 141–147.

[203] QUARTEL, D., DIJKMAN, R., AND VAN SINDEREN, M. Methodological support for service-oriented design with isdl. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing* (New York, NY, USA, 2004), ACM Press, pp. 1–10.

[204] RABELO, R. J., CAMARINHA-MATOS, L. M., AND VALLEJOS, R. V. Agent-based brokerage for virtual enterprise creation in the moulds industry. In *Proceedings of the IFIP TC5/WG5.3 Second IFIP Working Conference on Infrastructures for Virtual Organizations: Managing Cooperation in Virtual Organiz ations and Electronic Busimess towards Smart Organizations* (Deventer, The Netherlands, 2000), Kluwer, B.V., pp. 281–290.

[205] RAHM, E., AND BERNSTEIN, P. A. A survey of approaches to automatic schema matching. *The VLDB Journal 10*, 4 (2001), 334–350.

[206] RAHM, E., DO, H.-H., AND MAÃ§MANN, S. Matching large XML schemas. *SIGMOD Rec. 33*, 4 (2004), 26–31.

[207] RAMA, J., AND BISHOP, J. A survey and comparison of CSCW groupware applications. In *SAICSIT '06: Proceedings of the 2006 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing couuntries* (New York, NY, USA, 2006), ACM Press, pp. 198–205.

[208] Hypertext Transfer Protocol – HTTP/1.1. Tech. rep., The Internet Society, June 1999. RFC2616.

[209] RIEBISCH, M. Towards a More Precise Definition of Feature Models. In *Modelling Variability for Object-Oriented Product Lines ECOOP Workshop* (2003), pp. 64–76.

[210] RINE, D., NADA, N., AND JABER, K. Using adapters to reduce interaction complexity in reusable component-based software development. In *SSR '99: Symposium on Software Reusability* (New York, NY, USA, 1999), ACM Press, pp. 37–43.

[211] ROSETTANET CONSORTIUM. Rosettanet implementation framework: Core specification v02.00.00, 2004. http://www.rosettanet.org/.

[212] The Rule Markup Initiative. http://www.ruleml.org/, oct 2008.

[213] RUOHOMAA, S., AND KUTVONEN, L. Trust management survey. In *Proceedings of the iTrust 3rd International Conference on Trust Management, 23–26, May, 2005, Rocquencourt, France* (May 2005), Springer-Verlag, LNCS 3477/2005, pp. 77–92.

[214] RUOHOMAA, S., KUTVONEN, L., AND KOUTROULI, E. Reputation management survey. In *Proceedings of the 2nd International Conference on Availability, Reliability and Security (ARES 2007)* (Vienna, Austria, Apr. 2007), IEEE Computer Society, pp. 103–111.

[215] RUOKOLAINEN, T., AND KUTVONEN, L. Addressing Autonomy and Interoperability in Breeding Environments. In *Network-Centric Collaboration and Supporting Frameworks* (Boston, Sept. 2006), L. Camarinho-Matos, H. Afsarmanesh, and M. Ollus, Eds., vol. 224 of *IFIP International Federation for Information Processing*, Springer, pp. 481–488.

[216] RUOKOLAINEN, T., AND KUTVONEN, L. Interoperability in Service-Based Communities. In *Business Process Management Workshops: BPM 2005 International Workshops, BPI, BPD, ENEI, BPRM, WSCOBPM, BPS* (2006), C. Bussler and A. Haller, Eds., vol. 3812 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 317–328.

[217] RUOKOLAINEN, T., AND KUTVONEN, L. Service Typing in Collaborative Systems. In *Enterprise Interoperability: New Challenges and Approaches* (Apr. 2007), G. Doumeingts, J. Müller, G. Morel, and B. Vallespir, Eds., Springer, pp. 343–354.

[218] RUSSELL, S. J., AND NORVIG, P. *Artificial Intelligence: A Modern Approach*, 2nd ed. Prentice-Hall, 2002.

[219] SAGONAS, K., SWIFT, T., AND WARREN, D. S. XSB as an efficient deductive database engine. In *SIGMOD '94: International conference on Management of data* (New York, NY, USA, 1994), ACM Press, pp. 442–453.

[220] SALAN, G., BORDEAUX, L., AND SCHAERF, M. Describing and Reasoning on Web Services using Process Algebra. In *ICWS '04: Proceedings of the IEEE International Conference on Web Services (ICWS'04)* (Washington, DC, USA, 2004), IEEE Computer Society, p. 43.

[221] SANEN, F., TRUYEN, E., AND JOOSEN, W. Managing Concern Interactions in Middleware. In *Distributed Applications and Interoperable Systems (DAIS)* (2007), vol. 4531 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 267–283.

[222] SANGIORGI, D., AND WALKER, D. *The Pi-Calculus — A Theory of Mobile Processes*. Cambridge University Press, 2001.

[223] SAP-ERP Software Web pages.

[224] SAVAGE, J. E. *Models of Computation: Exploring the Power of Computing*. Addison-Wesley, 1998.

[225] SCHMIDT, D. C. Model-Driven Engineering. *Computer 39*, 2 (Feb. 2006), 25–31.

[226] SCHMIDT, M.-T., HUTCHISON, B., LAMBROS, P., AND PHIPPEN, R. The enterprise service bus: making service-oriented architecture real. *IBM Syst. J. 44*, 4 (2005), 781–797.

[227] SHOHAM, Y. Agent-oriented programming. *Artif. Intell. 60*, 1 (1993), 51–92.

[228] SIMÉON, J., AND WADLER, P. The essence of XML. *SIGPLAN Not. 38*, 1 (2003), 1–13.

[229] SIMMONDS, D., SOLBERG, A., REDDY, R., FRANCE, R., AND GHOSH, S. An Aspect Oriented Model Driven Framework. In *Ninth IEEE International EDOC Enterprise Computing Conference (EDOC'05)* (2005), IEEE Computer Society.

[230] SINGH, M. P., CHOPRA, A. K., DESAI, N., AND MALLYA, A. U. Protocols for processes: programming in the large for open systems. *SIGPLAN Not. 39*, 12 (2004), 73–83.

[231] SINGH, M. P., AND HUHNS, M. N. *Service-Oriented Computing: Semantic, Processes, Agents*. John Wiley & Sons, Ltd., West Sussex, England, 2005.

[232] SKENE, J., LAMANNA, D. D., AND EMMERICH, W. Precise Service Level Agreements. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 179–188.

[233] SKENE, J., SKENE, A., CRAMPTON, J., AND EMMERICH, W. The monitorability of service-level agreements for application-service provision. In *WOSP '07: Proceedings of the 6th international workshop on Software and performance* (New York, NY, USA, 2007), ACM Press, pp. 3–14.

[234] STEEN, M., AND DERRICK, J. Formalising ODP enterprise policies. In *EDOC'99* (1999), IEEE, pp. 84–93.

[235] STEINMETZ, R., AND WEHRLE, K. What Is This "Peer-to-Peer" About? In *Peer-to-Peer Systems and Applications*, vol. 3485 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005, pp. 9–16. http://dx.doi.org/10.1007/11530657_2.

[236] STOJANOVIC, Z., AND DAHANAYAKE, A., Eds. *Service-Oriented Software System Engineering: Challenges and Practices*. Idea Group Publishing, 2005.

[237] STROUSTRUP, B. *The C++ Programming Language*, 2nd ed. Addison-Wesley, 1991.

[238] SU, S. Y., HUANG, C., HAMMER, J., AND ET AL. An internet-based negotiation server for e-commerce. *The International Journal on Very Large Data Bases 10*, 1 (Aug. 2001), 72–90.

[239] SUN MICROSYSTEMS. *Java 2 Platform, Enterprise Edition (J2EE), 1.4 Specification*, 2002.

[240] TAGG, R. Workflow in different styles of virtual enterprise. In *ITVE '01: Proceedings of the workshop on Information technology for virtual enterprises* (Washington, DC, USA, 2001), IEEE Computer Society, pp. 21–28.

[241] TAKEUCHI, K., HONDA, K., AND KUBO, M. An interaction-based language and its typing system. In *6th European Conference on Parallel Architectures and Languages* (1994), pp. 398–413.

[242] THATTE, S. *Specification: Business Process Execution Language for Web Services Version 1.1*, May 2003.

[243] THE ECLIPSE FOUNDATION. Am3. The Eclipsepedia – the Eclipse.org Wiki, 2008.

[244] THE ECLIPSE FOUNDATION. Am3/how install am3 from cvs. The Eclipsepedia – the Eclipse.org Wiki, 2008.

[245] THE ECLIPSE FOUNDATION. Amma. The Eclipsepedia – the Eclipse.org Wiki, 2008.

[246] THIAGARAJAN, P. Regular Event Structures and Finite Petri Nets: A Conjecture. In *Formal and Natural Computing* (2002), vol. 2300 of *Lecture Notes in Computer Science*, Springer, pp. 244–253.

[247] THIRIOUX, X., COMBEMALE, B., CRÃL'GUT, X., AND GAROCHE, P.-L. A Framework to Formalize the MDE Foundations. In *Workshop on Towers of Models co-located with TOOLS Europe* (June 2007), R. F. Paige and J. BÃl'zivin, Eds., pp. 14–30.

[248] TSAI, W. T. Service-Oriented System Engineering: A New Paradigm. In *SOSE '05: Proceedings of the IEEE International Workshop* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 3–8.

[249] TSALGATIDOU, A., AND PILIOURA, T. An overview of standards and related technology in web services. *Distrib. Parallel Databases 12*, 2-3 (2002), 135–162.

[250] UDDI. *Universal Description, Discovery, and Integration of Business for the Web*, Oct. 2001.

[251] Uniform Electronic Transactions Act. http://www.law.upenn.edu/bll/ulc/fnact99/1990s/ueta.htm, July 1999. 2 (16).

[252] VALATKAITE, I., AND VASILECAS, O. A conceptual graphs approach for business rules modeling. In *Advances in Databases and Information Systems* (Sept. 2003), vol. 2798 of *LNCS*, pp. 178–189.

[253] VALLECILLO, A., HERNÁNDEZ, J., AND TROYA, J. M. Object Interoperability. In *Object-Oriented Technology. ECOOP'99 Workshop* (1999), vol. 1743 of *LNCS*, Springer-Verlag Heidelberg, pp. 1–21.

[254] VALLECILLO, A., HERNANDEZ, J., AND TROYA, J. M. Component Interoperability. Tech. Rep. ITI-2000-37, University of Malaga, July 2000.

[255] VALLECILLO, A., VASCONCELOS, V. T., AND RAVARA, A. Typing the Behavior of Objects and Components using Session Types. *Electronic Notes in Theoretical Computer Science 68*, 3 (2003). Presented at FOCLASA'02.

[256] VAN DER AALST, W. Loosely coupled interorganizational workflows: modeling and analyzing workflows crossing organizational boundaries. *Information & Management 37* (2000), 67–75.

[257] VAN GLABBEEK, R., AND GOLTZ, U. Refinement of actions and equivalence notions for concurrent systems. *Acta Inf. 37*, 4-5 (2000), 229–327.

[258] W3C. *XSL Transformations*, Nov. 1999. W3C Recommendation 16 November 1999.

[259] W3C. *OWL Web Ontology Language Guide*, Feb. 2004. W3C Recommendation 10 February 2004.

[260] W3C. *RDF Primer*, Feb. 2004. W3C Recommendation 10 February 2004.

[261] W3C. *RDF Vocabulary Description Language 1.0: RDF Schema*, Feb. 2004. W3C Recommendation 10 February 2004.

[262] W3C. *Web Services Architecture*, Feb. 2004. W3C Working Group Note 11.

[263] W3C. *XML Schema Documentation; Part 1:Structures, Part 2: Datatypes*, 2nd ed., Oct. 2004. W3C Recommendation, http://www.w3.org/XML/Schema.

[264] W3C. *XQuery 1.0 and XPath 2.0 Formal Semantics*, 2005. W3C Candidate Recommendation 3 November 2005.

[265] W3C. *SOAP Version 1.2 Part 1: Messaging Framework*, 2 ed., Apr. 2007. W3C Recommendation.

[266] W3C. *Web Services Description Language (WSDL) Version 2.0 Part 2: Adjuncts*, Mar. 2007.

[267] W3C WORKING GROUPS. *Extensible Markup Language (XML)*. W3C, 2005. http://www.w3.org/XML/, valid 5th October 2005.

[268] WADA, H., SUZUKI, J., AND OBA, K. Modeling non-functional aspects in service oriented architecture. In *SCC '06: Proceedings of the IEEE International Conference on Services Computing* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 222–229.

[269] WAGNER, G., TABET, S., AND BOLEY, H. *MOF-RuleML: The Abstract Syntax of RuleML as a MOF Model*, oct 2003. http://www.omg.org/docs/br/03-10-02.pdf (31.10.2008).

[270] WEGNER, P. Interoperability. *ACM Computing Surveys (CSUR) 28*, 1 (1996), 285–287.

[271] WHITTLE, J., ARAÚJO, J., AND MOREIRA, A. Composing aspect models with graph transformations. In *EA '06: Proceedings of the 2006 international workshop on Early aspects at ICSE* (New York, NY, USA, 2006), ACM Press, pp. 59–65.

[272] Wikipedia, the free encyclopedia: Natural person. http://en.wikipedia.org/wiki/Natural_person, May 2007.

[273] Wikipedia, the free encyclopedia: Organization. http://en.wikipedia.org/wiki/Organization, May 2007.

[274] WINANS, T. Object technology in the extended enterprise. In *2nd International Enterprise Distributed Object Computing Workshop* (1998), IEEE, pp. 378–389.

[275] WINSKEL, G., AND NIELSEN, M. Models for concurrency. In *Semantic Modelling*, S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, Eds., vol. 4 of *Handbook of Logic in Computer Science*. Clarendon Press, 1995.

[276] WOOLRIDGE, M. Agent-based software engineering. *IEE Proceedings, Software Engineering 144*, 1 (Feb. 1997), 26–37.

[277] YELLIN, D. M., AND STROM, R. E. Protocol specifications and component adaptors. *ACM Trans. Program. Lang. Syst. 19*, 2 (1997), 292–333.

[278] ZAMBONELLI, F., AND OMICINI, A. Challenges and research directions in agent-oriented software engineering. *Autonomous Agents and Multi-Agent Systems 9*, 3 (2004), 253–283.

[279] ZAREMSKI, A. M., AND WING, J. M. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology (TOSEM) 6*, 4 (1997), 333–369.