KATHOLIEKE UNIVERSITEIT
LEUVEN

Arenberg Doctoral School of Science, Engineering & Technology
Faculty of Engineering
Department of Computer Science

Macodo: Architecture-Centric Support for Dynamic Service Collaborations

Robrecht HAESEVOETS

Dissertation presented in partial
fulfillment of the requirements for
the degree of Doctor
in Engineering

February 2012

Macodo: Architecture-Centric Support for Dynamic Service Collaborations

Robrecht HAESEVOETS

Jury:
Prof. dr. ir. Paul Sas, chair
Prof. dr. Tom Holvoet, promotor
Prof. dr. Danny Weyns, co-promotor
(Linnaeus University)
Prof. dr. ir. Wouter Joosen
Prof. dr. ir. Eric Steegmans
Prof. dr. ir. Frank Piessens

Prof. dr. Virginia Dignum
(Delft University of Technology)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering

February 2012

D/2012/7515/15
ISBN 978-94-6018-476-5

Abstract

Flexible integration and collaboration of information systems, both within and across company borders, has become essential to success in current business environments. In the domain of supply chain management, for example, the arrival of a new order can no longer be handled by a single system. Instead, it requires complex collaborations, between multiple systems and services across companies, that constantly have to be adapted to changing market needs. The primary goal of information technology in such domains is to support flexible integration and collaboration between these systems. Without proper integration and collaboration, systems can easily become islands of information, resulting in inefficient and inflexible solutions. Realizing collaborations and building the supporting infrastructures, however, poses huge engineering challenges, from architectural design to actual implementation. To address these challenges, current state of practice relies on middleware, workflow management, service-oriented architecture, and Web services.

In this thesis, we argue that several engineering challenges are insufficiently addressed by current state of practice, such as modularization of collaborations, managing complexity, and separation of concerns. This often leads to faults, complex solutions with little reuse, and reduced productivity. Most of these problems relate to the lack of proper collaboration abstractions and the missing reification of these abstractions throughout the development process. To address a number of these problems, we present Macodo, an approach that consists of three complementary parts: (1) a set of abstractions for dynamic service collaborations, (2) a set of architectural views that reify these abstractions at architectural level, allowing to build and document service collaborations in terms of software elements, and (3) a middleware infrastructure that supports the collaboration abstractions at implementation level. Macodo focusses on service collaborations that take place in a restricted collaboration environment, managed by a trusted party. To validate the contributions of this thesis, we apply them in a supply chain management case and evaluate them in a controlled experiment. Results show that the use of Macodo, compared to state of practice, can provide an improvement in terms of fault density, design complexity, level of reuse, and productivity.

Samenvatting

Flexibele integratie en samenwerking van informatiesystemen, zowel binnen als tussen verschillende bedrijven, worden alsmaar belangrijker om succesvol te zijn in allerlei sectoren. Een voorbeeld is het domein van *supply chain management*. Elk bedrijf in een *supply chain* heeft een waaier aan systemen, vaak verspreid over verschillende afdelingen en locaties. Een eenvoudige bestelling kan niet langer afgehandeld worden door een geïsoleerd systeem, maar vereist een complexe samenwerking tussen allerlei systemen en services van verschillende bedrijven. Deze samenwerkingen moeten bovendien voortdurend aangepast worden aan de steeds evoluerende markt. Zo kunnen leveranciers en klanten van vandaag morgen reeds veranderen. Zonder flexibele integratie en samenwerking worden systemen al snel informatie-eilanden, wat leidt tot inefficiënte en niet-flexibele oplossingen. De belangrijkste rol van informatietechnologie in dergelijke sectoren is dan ook het ondersteunen van flexibele integratie en samenwerking tussen informatiesystemen. Het realiseren van zulke samenwerkingen en het bouwen van de ondersteunende infrastructuur stelt echter enorme uitdagingen op het vlak van ontwikkeling, architecturaal ontwerp en implementatie. Om deze uitdagingen aan te pakken berust men momenteel op *middleware*, *workflow management*, *service-oriented architecture* en *Web services*.

In deze thesis stellen we dat verschillende problemen onvoldoende aangepakt worden door de huidige oplossingen. Belangrijke probleemgebieden zijn de modularisatie van samenwerkingen, het beheren van complexiteit en het scheiden van *concerns*. Dit leidt vaak tot fouten, complexe oplossingen, weinig herbruik en in het algemeen een lagere ontwikkelingsproductiviteit. De meeste van deze problemen zijn het gevolg van het gebrek aan de juiste samenwerkingsabstracties en het gebrek aan ondersteuning van deze abstracties doorheen het software-ontwikkelingsproces. Om deze problemen aan te pakken, presenteert deze thesis Macodo. Macodo bestaat uit drie complementaire contributies:

- **Een conceptueel model voor dynamische samenwerkingen [207, 101, 103].** Dit model beschrijft een set van samenwerkingsabstracties en definieert de 'taal' voor Macodo. Het model consolideert eerdere onderzoeksresultaten en is gebaseerd op rol-gebaseerde modelleringstechnieken uit verschillende domeinen. De abstracties laten toe om complexe

samenwerkingen te modulariseren en om interacties, gedrag en het beheer van samenwerkingen voor te stellen als aparte *concerns* [99, 102, 206].

- **Een set van architecturale views voor het ontwerpen en documenteren van samenwerkingen in termen van software elementen [208].** De architecturale views bieden modelleringsconcepten aan die de samenwerkingsabstracties voorstellen op architecturaal niveau. Dit laat toe om samenwerkingen te ontwerpen en te documenteren, maar ook om te redeneren over samenwerkingskwaliteiten in termen van software-elementen.

- **Een *proof of concept* middleware infrastructuur die de samenwerkingsabstracties ondersteunt als programmeerabstracties [208, 104, 100, 98].** Deze middleware infrastructuur biedt een concreet platform aan om samenwerkingen, ontworpen in de architecturale views, effectief te ontwikkelen en te implementeren. De middleware mapt de samenwerkingsabstracties op bestaande *Web service* technologie en ondersteunt ze als programmeerabstracties zonder de nood aan nieuwe standaarden.

Om de contributies van deze thesis te valideren, passen we ze toe in een *supply chain management* case en evalueren we het conceptueel model en de architecturale views in een gecontroleerd experiment. Resultaten van het experiment tonen aan dat het gebruik van Macodo, vergeleken met de huidige oplossingen, effectief kan leiden tot minder fouten, minder complexe oplossingen, meer herbruik en een hogere ontwikkelingsproductiviteit.

Dankwoord

Verschillende personen hebben een belangrijke rol gespeeld in het realiseren van dit onderzoek. Eerst en vooral zijn er mijn promotoren Danny Weyns en Tom Holvoet. Danny is altijd een drijvende, inspirerende en kritische kracht geweest achter dit werk. Tom bood alle vertrouwen en vrijheid om aan interessant onderzoek te doen, maar op de juiste momenten ook de nodige kritische reflectie. Zowel Tom als Danny hebben er bovendien voor gezorgd dat dit onderzoek effectief kon gevalideerd worden met studenten. Deze validatie vormt een belangrijk luik van dit onderzoek. Ook juryleden Virginia Dignum, Wouter Joosen, Frank Piessens en Eric Steegmans hebben verschillende suggesties gegeven die in de finale versie van deze tekst zijn verwerkt. Daarnaast hebben de talloze discussies en samenwerkingen met collega's en ex-collega's van AgentWise en DistriNet zeker hun impact gehad.

Tot slot, aan alle vrienden en familie, we hebben het misschien niet vaak over onderzoek gehad, jullie hebben in elke geval voor de nodige afwisseling en steun gezorgd.

Robrecht
Januari 2012

Contents

Chapter 1

Introduction

In today's volatile business environments, flexible integration and collaboration of information systems, both within and across company borders, is essential to success. Take the domain of supply chain management. Each company in the supply chain relies on a multitude of systems, ranging from ERP systems[1] to procurement and transport tracking systems, often spread across departments and locations. The arrival of a new order can no longer be handled by a single system, but requires complex collaborations among multiple systems and services across companies. Without the proper integration and collaboration, these systems become islands of information, resulting in inefficiency, inflexibility, and poor visibility in supply chains [182]. Furthermore, the dynamic and unpredictable market is causing a constant change in the supply chain network. Today's suppliers and customers might change tomorrow. This requires collaborations to be easily adapted to current market needs. The primary goal of information technology is to support flexible integration and collaboration among these systems, both within and across companies borders [182, 121].

Realizing collaborations that can easily be adapted and building the supporting information systems poses huge engineering challenges, from architectural design to actual implementation. To address these challenges, current state of practice relies on middleware, workflow management, service-oriented architecture (SOA) and Web services. Several challenges, however, are still insufficiently addressed. Key problem areas are modularization of collaborations, managing complexity, and separation of concerns. This can lead to faults, complex solutions with little reuse, and reduced productivity. Most of these problems relate to the lack of proper collaboration abstractions, and the missing support for these abstractions throughout the development cycle (from architecture to actual implementation). The need for abstractions has been recognized in general software engineering [11, 17, 51], business process modeling [50] and SOA [186, 122].

[1] ERP: Enterprise Resource Planning.

In this thesis we present Macodo to address a number of these problems. Macodo consists of three complementary parts: (1) a set of abstractions for dynamic collaborations, (2) a set of architectural views, based on these abstractions, to build and document collaborations in terms of software elements, and (3) a middleware infrastructure that supports the collaboration abstractions at implementation level. Macodo focusses on service collaborations that take place in a restricted collaboration environment, managed by a trusted party. We apply the main contributions in a supply chain management case and evaluate them in a controlled experiment.

Overview. Section 1.1 starts by elaborating the challenges in engineering dynamic collaborations and discusses the available solutions based on middleware, workflow management, SOA and Web services. Next, Sect. 1.2 defines the scope of this thesis. Within this scope, Sect. 1.3 formulates a set of concrete problem statements, related to engineering collaborations, which are insufficiently addressed by current solutions. Finally, Section 1.4 lists the contributions of this thesis, and Sect. 1.5 gives an overview of the following chapters.

1.1 Context

A collaboration can be defined as the process of entities working together to achieve a set of goals. Collaborations are typically realized using a form of coordination, which organizes entities to act in a coherent and structured way [150]. Coordination relies on direct or indirect communication between the entities [143] and requires a shared 'language'. Engineering collaborations among distributed software entities is a complex task and poses huge engineering challenges.

These challenges can be divided into business challenges, technological challenges and design challenges. The main business challenges are establishing trust, maintaining communities of possible collaboration partners, selecting partners, and setting up collaboration agreements [8, 146]. Technology challenges range from integrating and inter-operating distributed and heterogeneous information systems and realizing coordination and mutual understanding among participating systems [143], to the actual deployment and operation of required software in each participant. Apart from these challenges, engineering dynamic collaborations leads to another set of problems. How do we manage the design complexity? How do we express the main decisions on how the system achieves the required functions and realizes the required qualities? How do we communicate these decisions to other stakeholders, such as developers, clients and maintainers, to achieve mutual understanding? Without proper support, this leads to complex and inflexible systems, in which it is hard to react to dynamics, both on short-term (quickly adjusting collaborations) and mid-term (adapting software or re-engineering collaborations).

Collaborations can take place in several types of environments, differing in terms of openness, stability, flexibility, and trustfulness [52]. The type of environment is a major influence on the engineering efforts required to realize successful collaborations. Open environments, such as the Internet, put no restrictions on participants. This leads to high flexibility, but makes it hard to predict the outcome of interactions or to establish any mutual trust. Closed environments (e.g., within one company) are able to provide stability and trust with smaller engineering efforts, but do not allow external participants. To build realistic collaborations, developers require semi-open or semi-closed environments[2]. Such environments, which are gaining prominence [165, 47, 16, 164], take a more pragmatic approach. The maintainer of the software, deployed in such an environment, puts restrictions on the participants. This allows to achieve the necessary stability and trust [140], while still allowing selected participants to join. In the supply chain domain, trusted third-parties are providing such environments. Examples are 3PLs and 4PLs (third-party and fourth-party logistics providers), or companies like SupplyOn[3] and GXS[4]. By acting as trusted integrators, these companies are able to realize controlled collaborations among multiple supply chain partners [177], without the need of complex peer-to-peer protocols.

The rest of this section discusses the most prominent solutions to engineer dynamic collaborations. Middleware (Sect. 1.1.1) provides a conventional solution to address the challenges of integrating and inter-operating distributed and heterogeneous information systems. Enterprise Application Integration (EAI) (Sect. 1.1.1) extends the capabilities of middleware to integrate heterogeneous applications at intra-enterprise level. EAI is typically used in combination with workflow management (WfM) and business process management (BPM). WfM and BPM (Sect. 1.1.2) allow to express and realize the actual integration and collaboration logic. Service-oriented architecture (SOA) and Web services (WS) (Sect. 1.1.3), two more recent developments, address challenges related to integration and collaboration in distributed, but also open, environments. Web services have revived the use of BPM and WfM and can be seen as another extension to middleware and EAI to go from intra-enterprise to inter-enterprise integration.

1.1.1 Middleware and Enterprise Application Integration

Middleware is commonly defined as the software layer that sits between the operating system and the applications on each site of a distributed system [137]. In a broader sense, it can be seen as the set of software services that enables the interoperation of distributed software components running on different machines and operating systems. Middleware offers a set of programming abstractions to

[2] Semi-open environments allow the controlled joining of selected participants. In semi-closed environments, new participants do not really join, but selected participants can get a trusted representative in the collaboration.

[3] www.supplyon.com

[4] www.gxs.com

facilitate the development of complex distributed systems. Good abstractions are key to successful software engineering [86, 210, 138]. Abstractions can hide low level details of hardware, networks and distribution, and provide developers access to functionality that they otherwise would have to implement from scratch. But middleware is also infrastructure. For abstractions to be useful, they need good supporting infrastructure, providing a comprehensive platform for developing and running complex distributed systems [12]. The main types of conventional middleware are remote-procedure calls (RPC), transaction processing (TP) monitors, object request brokers (ORB) and message-oriented middleware (MOM). When systems become too heterogeneous in nature and functionality (e.g., different interfaces, data formats, and interaction protocols), conventional middleware can lead to complex solutions that are hard to realize and maintain.

Enterprise application integration (EAI) can be seen as the next step in the evolution of middleware [178]. While conventional middleware focusses on integrating systems or sub-systems that are physically close, EAI extends these features to integrate systems that are complete applications on their own [12]. EAI addresses the two main concerns of intra-enterprise integration: (1) dealing with heterogeneity, by providing a uniform view to the applications that integrate the systems, and (2) defining, enacting, and managing the actual application integration and composition logic. To deal with heterogeneity, EAI relies on the use of message brokers and adapters. Message brokers, an extension of the conventional MOM, allow to flexibly route information between system entities, creating a 'hub and spoke' communication structure. Adapters map heterogeneous data formats, interfaces, and protocols to a common model and format. A modern example of such an infrastructure is the enterprise service bus (ESB) [161]. The actual integration within EAI is realized using workflow management systems (WfMS), where workflows describe the different execution steps of the integration process.

Web services can be seen as another extension to conventional middleware and EAI. They provide the technology that allows companies to go from intra-enterprise integration to inter-enterprise integration [12].

1.1.2 Business Process and Workflow Management

Business process management (BPM) and workflow management (WfM) are often used to express and realize the actual integration or collaboration logic (e.g., in EAI). BPM and WfM have a long history and are closely related [202, 160]. BPM is often seen as a process-oriented management discipline, in which WfM can be used as a concrete flow management technology [109]. Others see WfM as a subset of BPM in terms of features [136].

The Workflow Management Coalition defines a workflow as "The automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules." [1]. Others have defined a workflow as "a coherent set of

activities carried out by a collaborating group to achieve a goal" [160]. The first workflows consisted of policies and procedure manuals, describing how things were done inside a company or organization. In the 90s, workflow management systems (WfMS) emerged, providing active support for simple business processes. Once created, however, the processes in these systems were hard to change. With the introduction of business process management systems (BPMS), the process has become a first-class object, which can be defined, enacted, managed, and easily changed [160].

In the context of EAI, WfM has caused a shift of programming in the small, to assembling or programming in the large, avoiding the hard-coding of business processes into the application [202, 12]. More recently, Web service composition has revived the use of WfM and BPM [12]. New composition languages, such as WS-BPEL[5] [5], allow to orchestrate Web services using workflow concepts. Hill et al. [110] and Ko et al. [136], however, make an explicit distinction between processes in service-oriented architecture (SOA) and business processes. Ko et al. [136] see BPM as a process-oriented management discipline aided by IT, while SOA is an IT architectural paradigm. Hill et al. [110] say BPM "organizes people for greater agility" while SOA "organizes technology for greater agility".

1.1.3 Service-Oriented Architecture and Web Services

Service-oriented architecture (SOA) and Web services (WS) address several challenges related to integration and collaboration in distributed, but also open, environments. SOA offers a set of design principles and architectural patterns to develop and integrate distributed systems. The main characteristics of SOA are loose coupling, service abstraction, service autonomy, discoverability, but also reuse and service composability [68, 161]. SOA typically implies the use of Web services, the most prominent technology stack to realize SOA. The term Web service is also used to define a piece of software that exposes some functionality (or service) and makes it available through standard Web technologies [12]. The main logical elements of SOA are service providers, service clients, and service registries. Service providers register their Web service in service registries, where they can be discovered by clients. Clients can then invoke the actual Web services using XML-based messages via standard Internet protocols.

A key principle of SOA is service orchestration [68]. Service orchestration allows to combine, or orchestrate, the functionality of multiple services. Orchestration is typically described as a workflow, defining the order of service invocations, which can be executed using an orchestration engine. The most prominent language for orchestration is WS-BPEL [5]. The two main uses of orchestration are service composition and application integration. Service composition uses orchestration to encapsulate the functionality of multiple services into a new composite services, hiding the details of the underlying services. Clients of composite services no

[5] WS-BPEL: Web Services Business Process Execution Language

longer know whether the are using a 'basic' or composite service. The other use of orchestration is to integrate applications, often in the context of EAI, but also for inter-enterprise collaborations.

The use of orchestration typically leads to a layered or hierarchical service architecture [68, 161]. Each layer represents a different level of service abstraction. At the lowest levels, application services provide access to basic application logic and functionalities. In the middle levels, business services compose application and other business services into reusable blocks of business logic. At the highest levels, process services orchestrate business services to realize the actual integration, collaboration, and business processes of enterprise systems.

In the context of inter-enterprise integration, orchestration is often seen as a private process, because it has to be executed in a central place, by a single party. In such a setting, choreography can be used to describe a public coordination or integration protocol between services of multiple enterprises. Choreography defines the externally visible (public) behavior of a service, while an orchestration defines the internal (private) behavior of a service. Choreographies are described similarly to orchestrations using workflow-based languages, such as WS-CDL [128]. A choreography, however, is not executable, but has to be translated to local (private) orchestrations.

There are two main perspectives on the real value of SOA and Web services [12]. One is to see Web services as a revolutionary technology that will change the way we think about middleware, application integration, and how we use the Internet. This would lead to dynamic interactions and seamless integration of IT infrastructures in completely open communities of businesses. The more pragmatic view is to see Web services as just another step in the evolution of middleware and EAI. In this view, Web services are no more than an additional layer on top of existing middleware and EAI platforms, and provide a set of simple, lowest common denominator interfaces for interactions across the Internet. This view is in line with the use of trusted third-parties, providing a restricted collaboration environment, by hosting the shared 'middleware' or executing a central orchestration. To this day, Web services are still mostly used for conventional EAI and inter-enterprise integrations in restricted settings [12]. The main reason for this, is that many complexities of inter-enterprise integration in open environments, such as trust, legal issues, mutual semantics, and complexities related to decentralization and peer-to-peer interaction, are not solved by current Web services.

1.2 Scope of this Thesis

Engineering collaborations is a very broad domain. To realize any useful contribution, it is important to take a proper scope. In this thesis, we focus on service collaborations that can be defined as follows:

> *The controlled interchange of information between a set of distributed entities (e.g., Web services) and the controlled execution of related tasks by these entities in order to achieve a set of goals.*

In addition, the scope of this thesis is limited to collaborations that take place in a restricted environment, characterized by the following properties:

- there is a central owner or maintainer (e.g., a trusted third-party), responsible for managing the collaborations;
- participation is controlled, it can be considered a semi-open or semi-closed collaboration environment;
- participants have a pre-established trust in the environment and its maintainer;
- there is a common ontology or mutual understanding between the participants;
- participants communicate using standard Web service technology.
- collaboration dynamics are restricted to:
  - creating new and terminating ongoing collaborations of predefined types in a controlled way;
  - dynamically adding and removing collaboration participants in a consistent manner.

Examples of such restricted environments are intra-company collaborations, EAI platforms, integration services provided by trusted 3rd-parties (e.g., 4PLs in the supply chain domain), and inter-company collaborations between strategic partners.

1.3 Problem Statement

Although state of practice addresses several engineering challenges, some key challenges remain unanswered. Research on middleware, BPM, and SOA has mainly focussed on individual service interactions, isolated processes and low-level infrastructure, neglecting the problems of how services collaborate [186, 122, 166]. This is partly caused by the focus of many approaches on decentralized or peer-to-peer coordination protocols [12], and the functional decomposition perspective taken by most service composition approaches.

When engineering dynamic collaborations, however, we are also confronted with other (often less technical) problems. How do we manage the design complexity? How do we express the main decisions on how the system achieves the required functions and realizes the required qualities? How do we communicate these

decisions to other stakeholders, such as developers, clients and maintainers, to achieve mutual understanding?

Software architecture provides an answer to these questions. The architecture of a software system defines its essential structures, which comprise software elements, the externally visible properties of those elements and the relationships between them [17], and with the environment [4]. Software architecture is a way to deal with complexity and serves as a vehicle of communication with stakeholders for mutual understanding and negotiation. Software architecture manifests the earliest set of design decisions and provides the main structures to realize both the required functionalities and quality attributes.

In this section we formulate a set of key problems in the domain of engineering dynamic collaborations that should be addressed at conceptual, architectural, and implementation level. We look at the lack of proper mechanisms to decompose and modularize complex collaborations, the focus of current solutions on functional decomposition, and the missing reification of collaboration abstractions throughout the development cycle. Based on these problem statements, we can define the main research questions of this thesis. The problem statements are further illustrated in Section 2.7.3 using a supply chain management case.

1.3.1 Lack of Proper Decomposition Mechanisms

Current languages for business process modeling, service orchestration, and service choreography, such as WS-CDL[6] [128] and WS-BPEL[7] [5], provide limited decomposition mechanisms. In addition, there are currently no standard mechanisms or high-level views, that allow to reason about collaboration management on a higher level of abstraction or at architectural level [158, 147]. Several process modeling languages even lack support for standard interaction patterns related to dynamics [202, 136].

Realizing collaborations, or even a single business process [48], however, involves the integration of many concerns such as interaction functionality, participant responsibilities, management, robustness, and variability. Due to the lack of proper decomposition mechanisms, models, and abstractions, conventional methods address most, if not all, of these concerns at implementation level [158, 147] and often in a single model or process description [122, 194]. This results in monolithic models and processes, for which development and maintenance complexity rapidly increases, and productivity decreases as the number of involved systems and services grows. Working with monolithic models also increases fault probability and limits possible reuse.

Some approaches try to address this problem by externalizing part of the behavior or management of processes, using business rule management systems [79, 176].

[6] WS-CDL: Web Services Choreography Description Language
[7] WS-BPEL: Web Services Business Process Execution Language

While the process describes the main functionality, external business rules describe variable business and management rules. Such approaches, however, lack proper abstraction and result in a complex set of 'if-then' rules, which do not represent the collaboration structure and are hard to represent at architectural level.

Key Problems

- The lack of proper decomposition mechanisms leads to monolithic processes and models. This increases complexity and fault probability, and limits reuse and productivity.
- A single model or process addresses multiple concerns (e.g., interaction functionality, participant responsibilities, and management), affecting the separation of concerns.

Key Challenges

Identify and describe concepts that:

- provide more natural units of decomposition for collaborations to promote reuse and improve understandability;
- allow to separate different concerns, such as interactions, participant behavior and responsibilities, and collaboration management.

1.3.2 Focus on Functional Decomposition

BPM promotes reuse through sub-processes (when supported[8]). SOA promotes modularization by encapsulating orchestration and composition of services as a reusable service (often called composite services). The use of sub-processes and composite services, however, leads to a functional decomposition of processes and services [68]. Such a functional decomposition does not preserve the underlying collaboration structure and makes it hard to capture interactions and represent the responsibilities of each participant. Interaction logic, participant behavior, and management are easily scattered across multiple processes and services. Developers cannot express or reason about collaborations in terms of relevant collaboration concepts. This can lead to increased complexity and faults, while making reuse in

[8]Current standards for orchestration and choreography, such as WS-CDL [128] and WS-BPEL [5], do not support the notion of sub-process. BPMN (Business Process Model and Notation) [6], a graphical notation for orchestration and choreography, does support sub-processes, but this is not reified by execution languages such as WS-BPEL.

terms of collaborations harder. The use of composite services further aggravates this problem by hiding the underlying services or participants[9].

Key Problems

- Functional decomposition of collaborations does not preserve the underlying collaboration structure, affecting the understandability of systems.
- Collaboration management is easily scattered across system components or is not expressed in terms of underlying collaboration structures and relevant collaboration concepts. This reduces the understandability and adaptability of systems.

Key Challenges

- Support decompositions that preserve the underlying collaboration structures and that allow to explicitly represent participants, behavior, and interactions as separate concepts.

1.3.3 Missing Reification of Collaboration Abstractions Throughout the Development Cycle

Several domains do provide useful abstractions to represent collaborations. Examples are role-based abstractions in conceptual and object-oriented modeling [190, 95], agent organizations [73, 213, 114], and role-based techniques for BPM [48, 160, 26]. Conventional middleware and current Web service infrastructures, however, do not reify these abstractions in software architecture, design, or implementation. As a result the problems identified by the previous two problem statements are not addressed throughout the development cycle. Furthermore, the introduction of new abstractions into the technology stack often requires new standards and languages. The current technology stack, however, is already getting confusing, and contains an abundance of languages and models [199, 136].

Key Problems

- Lack of modeling and programming abstractions that reify relevant collaboration concepts. This retains the problems of the first two problem statements and limits developers and other stockholders in expressing and reasoning about concerns and quality attributes of collaborations.

[9] A related trend can be observed in process-oriented views for services [186], in which a service is not only represented by its functional interface, but also exposes part of its internal process structure.

Key Challenges

- Reify collaboration concepts at architectural level and implementation level.
- Represent and document collaborations in terms of software elements.
- Integrate programming abstractions in current technology stacks without the need for new standards.

1.3.4 Main Research Questions

The three problem statements lead to the following research questions:

(1) *What are good abstractions to support and promote modularization of service collaborations, in order to reduce the number of faults, better manage complexity, and improve reuse and productivity?*

(2) *How can these abstractions be reified and supported throughout the development cycle (i.e., architecture, design, and implementation)?*

1.4 Contributions

To address the research questions, this thesis provides three main contributions:

1. **A conceptual model for dynamic collaborations [207, 101, 103].** The conceptual model describes a set of collaboration abstractions that define the vocabulary for Macodo. The model consolidates earlier research results and is based on role-based modeling techniques from several domains. The abstractions support a more natural decomposition and modularization of collaborations that maintains the underlying collaboration structures. They also allow to model interactions, individual responsibilities, behavior of participants, and management of collaborations as separate concerns [99, 102, 206].
2. **A set of architectural views to design and document collaborations in terms of software elements [208].** The architectural views provide architectural modeling abstractions that reify the collaboration abstractions of the conceptual model at architectural level. They allow to design, document, and reason about collaborations and their qualities in terms of software elements.
3. **A proof of concept middleware infrastructure supporting collaboration abstractions as programming abstractions [208, 104, 100, 98].**

The proof of concept middleware infrastructure provides a concrete platform to develop and implement collaborations that are designed in the architectural views. The middleware maps collaboration abstractions to concrete Web service technology (e.g., WSDL and WS-BPEL) and supports them as programming abstractions without the need for new standards.

To validate these contributions, we apply them in a supply chain management case and evaluate the Macodo conceptual model and architectural views in a controlled experiment. Results of the experiment show that the use of Macodo, compared to state of practice, can provide an improvement in terms of fault density, design complexity, level of reuse, and productivity.

1.5 Overview of this thesis

We conclude this chapter with an overview of this thesis.

Chapter 2 gives a concise overview of the most important domains in which the work of this thesis is situated: role-based modeling, software architecture, Web service technology, and supply chain management. We give the necessary background information to understand the following chapters, and illustrate the problem statements in a supply chain management case.

Chapter 3 presents the Macodo model, a conceptual model for dynamic collaborations. The model specifies a set of collaboration abstractions, independent of design and implementation concerns. It defines the vocabulary for Macodo and serves as a guide for developers and readers of this thesis. The model also provides the foundation for the subsequent chapters.

Chapter 4 introduces the architectural views for Macodo. These views map the collaboration abstractions of the conceptual model to architectural modeling abstractions and allow to design and document collaborations in terms of software elements.

Chapter 5 presents a proof of concept middleware infrastructure that maps the software elements of the architectural views to Web service technology. The middleware supports the collaboration abstractions at implementation level.

Chapter 6 evaluates two main contributions of this thesis, the conceptual model and the architectural views, using a controlled experiment. In the experiment, we compare Macodo with a reference approach in the context of designing systems that support centrally managed collaborations among a set of Web services.

Chapter 7 positions the contributions of this thesis with respect to related work. We focus on existing organization models and infrastructures, techniques to deal with variation in processes and collaborations, and decomposition mechanisms for business processes and collaborations.

Chapter 8 draws conclusions, summarizes the main contributions, and discusses possible opportunities for future work.

Chapter 2

Background

The work presented in this thesis is situated in several domains. This chapter discusses the most important domains and gives a concise overview of background information required to understand the following chapters.

2.1 Introduction

Four key domains form the basis for Macodo: role-based modeling, software architecture, Web services, and supply chain management (Fig. 2.1).

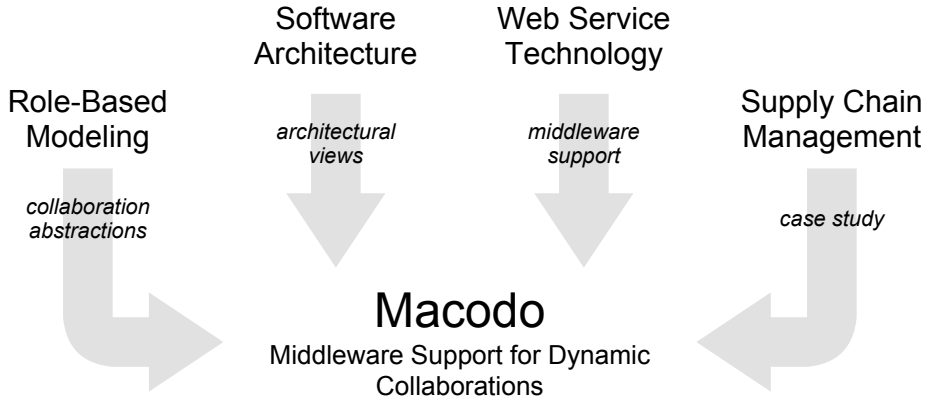


Figure 2.1: The four most important background domains of Macodo.

Role-based modeling provides the conceptual basis for the Macodo collaboration abstractions on which the work of this thesis is founded. Software architecture

provides mechanisms, such as architectural views, to deal with complexity and communicate with stakeholders. This thesis uses architectural views to describe and document collaborations in terms of software elements. To implement architectures described in these views, we use the Web service technology stack to create a proof of concept middleware infrastructure. To illustrate the problem statements of this thesis, and the different concepts throughout the following chapters, we use a supply chain management case. Supply chain management is a domain where collaborations play a primary role. This domain has been one of the key drivers for research tracks such as virtual organizations.

Overview. Section 2.2 starts by discussing role-based modeling techniques in the fields of business process modeling, and object-oriented and conceptual modeling. Section 2.3 gives an overview of roles and organization in the domain of multi-agent systems. Next, Sect. 2.4 discusses software architecture views, and Sect. 2.5 gives a concise overview of the most important Web service technologies. Finally, Sect. 2.6 and Sect. 2.7 discuss the domains of virtual organizations and supply chain management. Section 2.7 also introduces a running example, and illustrates the problem statements of this thesis in the running example.

2.2 Role-Based Modeling

Roles and organizations have a rich history in social science [162], organization theory [167, 35], and multi-agent systems [54, 85, 152, 61]. They are accepted as valuable abstractions to structure and manage both human and artificial societies [52], and to design interactions in multi-agent systems [124, 191, 85]. The concept of role is also recognized as an important modeling concept in object-oriented and conceptual modeling [190, 95], and business process modeling [48, 160, 26]. Even tough, the role concept has not received the attention it deserves and there is no consensus on the definition of roles or how they should be integrated in established modeling frameworks and mainstream programming languages [190, 95, 26, 108].

We first give a brief historical overview of roles (Sect. 2.2.1), and then discuss roles in conceptual and object-oriented modeling (Sect. 2.2.2), and business process modeling (Sect. 2.2.3). Section 2.3 provides an overview of roles and organizations in multi-agent systems.

2.2.1 History of Role-Based Modeling

Interest in roles started in the late 1920s when social scientists started to use roles to study patterns of human behavior and social structure. Roles were used to represent social positions and associated behavior. This idea was inspired by actors

taking up a role in a play. In computer science, roles first made their entrance in data models, when people observed that most conventional file records were role-oriented [14]. For example, files typically dealt with employees, customers, patients or students. The influence of roles on today's data models, however, is rather modest [190].

Later on, roles were further defined by making a distinction between 'natural types' and 'role types' [188]. Natural types relate to the essence of an entity, while a role type depends on an accidental relationship to some other entity. In this context, role types were often seen as subtypes of natural types, for example, the role types employee and student are subtypes of the natural type person. This idea has further developed into an ontological distinction between role types and natural types [92]. To be considered a role, a concept has to be founded (to belong to a role requires relationships to others) and has to lack semantic rigidity (entities can enter and leave a role without losing their identity). A natural type is not founded (to belong to a natural type does not require relationships to others) and has semantic rigidity (entities cannot drop their natural type without losing their identity). For example, an employee is a role, but person is a natural type. To be an employee requires a relation to a company (founded) and persons can enter and leave the role of employee without losing their identity (lacks semantic rigidity). Some concepts are neither, to be a teenager, does not require a relation to others (lacks foundation), and persons can stop being a teenager without losing their identity (lacks semantic rigidity). Such concepts are typically states or phases of an individual.

2.2.2 Roles in Object-Oriented and Conceptual Modeling

In object-oriented and conceptual modeling, roles are often seen as a natural complement to objects and relations, but there is an ongoing discussion on how to integrate roles into contemporary object-oriented and conceptual modeling languages. Steimann [190] identifies three common ways to see roles. Each way has its benefits and drawbacks. The first and most simple way, is to see roles as named places in a relationship. For example, the unified modeling language (UML) [193] uses roles to name association ends. Using roles in this way, shows that roles exist in the context of a relationship.

A second way is to see roles as subtypes (e.g., of natural types). Although this is often used, it leads to conceptual problems. For example, the roles customer and supplier can be seen as subtypes of the natural type person. But they can also be seen as subtypes of the natural type company. This means, that customers and suppliers are a subset of the intersection of persons and companies, which is either very small or empty. Similar problems are encountered when modeling roles as super-types or generalizations (e.g., of natural types). Guizzardi [95], tries to avoid this problem by introducing an ontological design pattern for role modeling, based on generalization and specialization.

The third way is to see roles as independent types of which the instances represent role specific state and behavior, but not identity. Roles are related to objects by a played-by relation. Picking up a role creates a new instance of the role type. Dropping a role destroys it. When playing a role, an object and the role become indivisible from the outside. In RM-ODP [197], for example, a role identifies, in a template for a composite object, a behavior to be associated with one of the component objects. Despite being useful metaphor, it requires serious changes to most conceptual modeling languages [95].

Although many researchers have argued the advantages of roles in programming languages, the concept of role has not found its way into mainstream programming languages [108]. Some languages do support roles, such as PowerJava [15], ObjectTeams [108], and Perl 6[1]. In these languages, roles allow objects to evolve over time. Each role is associated with an interface. Objects change their roles depending on the context and can play multiple roles at a time.

2.2.3 Roles in Business Process Modeling

In business process modeling, most prominent approaches rely on a procedural or data-oriented view of a process. Processes are described as activities and the data flows between them. This leads to a functional decomposition, in which activities carried out by individual systems or people are scattered throughout the model. This makes it hard to abstract away from the details of the process, or to capture the interactions between the systems or people who carry out the activities [168, 26, 131]. Ould [160] calls for collaboration-centric BPM. The essence of a process is a collaboration between a set of people or systems. In collaboration-centric BPM, collaboration is a primitive to model processes. A BPM system would support roles and mediate their interactions and make the intended collaboration happen.

Several researchers have proposed role-based BPM techniques [48, 160, 26]. Two prominent examples are Role Interaction Nets [183] (RIN) and Role Activity Diagrams (RAD) [113, 159]. Both languages have a formal underpinning (such as Petri nets [163]) and are based on organizational role theory. Key concepts in RIN and RAD are roles and interactions. In RIN, a 'role interaction network' is composed of a set of roles. The behavior of each role is described as a set of interactions with itself (solitary actions) and with other roles. In RAD, a process is a coherent set of activities, carried out by a collaborating set of roles to achieve a goal (i.e., a set of desired states). A role type defines a responsibility within a process. It is described as a set of actions (or activities) and interactions with other roles. A role instance is an instance of a role type that operates independently and concurrently. Role instances can be associated with an actor, that carries out the role, but role instances can also exist without an actor. In RAD, role instances can create other role instances, but this does not say anything about the actors

[1] http://dev.perl.org/perl6/

playing the instance, instantiating a role only creates the responsibility within a process. Interactions allow role instances to coordinate their activities, by aligning the states of the interacting activities. Riva [160] presents a BPM method based on RAD. The overall process design consists of case processes (handling units of work), case management processes (managing case processes) and case strategy processes (changing case processes and case management process in the long-term). Each type of process is described using RAD concepts.

In current mainstream business process modeling, roles only play a minor part. In WS-BPEL [5] roles are only used to distinguish the interfaces (called portTypes) defined in the partnerLinkTypes. In BPMN [6], a 'PartnerRole' can be used to represent a participant and its multiplicity in a collaboration (i.e., a message exchange between two or more processes) like roles in UML interaction diagrams. Lanes (or swim-lanes) can also be used to represent roles, but BPMN does not define their semantics, so they can be used at the designers will [6]. In WS-CDL [128], role types define the observable behavior of a party within a collaboration. Each behavior is defined as an interface. The choreography specifies the actual interaction. In most of these approaches roles are only used to name the endpoint of binary relationships and to identify participants in an interaction.

2.3 Roles and Organizations in Multi-Agent Systems

Organizations and roles have also been extensively studied and used in the domain of multi-agent systems [73, 213, 114, 61]. A multi-agent system (MAS) is a system composed of multiple interacting computing elements, known as agents. These agents have two important capabilities: (1) they are capable of autonomous action (to some extent); and (2) they can interact with other agents [212].

Organizations and roles are used to design and engineer MAS, to constrain and control behavior in agent societies, and to study and analyze properties of MAS. In its most general form, an organization can be seen as a cooperation pattern, or a process, that constraints the actions and interaction of agents towards some purpose [36]. In a more specific form, an organization can also refer to a collective entity with an explicit identity [180].

The rest of this section discusses how organizations can be used to both model MAS (Sect. 2.3.1) and implement MAS (Sect. 2.3.2). The chapter on related work (Sect. 7.1) gives a more elaborate discussion of existing organization approaches for MAS.

2.3.1 Organization-Oriented Modeling of Multi-Agent Systems

Within MAS, a distinction can be made between agent-oriented MAS, or agent-centered MAS (ACMAS), and organization-oriented MAS, or organization-centered

MAS (OCMAS) [74, 21, 62].

An agent-oriented MAS is designed in terms of agents and their mental states, taking an individual perspective on the system. This is how many traditional multi-agent systems are designed, which is reflected in classic agent development frameworks such as JADE [20] and FIPA-OS [169]. An example is an ant colony [66], where the 'organization' only exists as an observable emergent phenomenon. Inside the ants, there is no notion of organization.

Purely relying on the agent's internal architecture, however, is often insufficient to deal with more complex interactions in a MAS [74]. To address this problem, several agent methodologies started to take an organization-oriented perspective to design MAS [213, 73, 215, 129]. Organization-oriented MAS are explicitly designed in terms of organizations and roles.

Generic Meta-Models and Methodologies

Prominent agent methodologies have adopted organizations [213, 215, 54] and several researchers have proposed generic meta-models to model MAS using organizations.

AGR (Agent/Group/Role) [73, 75], for example, is a generic meta-model for MAS in which agents, playing roles, are organized into groups. Groups define a set of possible roles (or functional positions), an interaction graph (specifying the valid interactions between roles), and an interaction language. An organization consists of a set of groups and an organization structure that relates the roles in different groups to each other.

Odell et al. [152] also propose a meta-model based on the concepts of agent, group, and role. This model, inspired by human-based organization techniques, tries to clarify what the concept of role means in the context of agent-based systems. These ideas and concepts are then formalized as an extension to the UML superstructure [151].

In more recent work, PIM4Agents (Platform-Independent Metamodel for Agent systems) [105] proposes a model to standardize agent-oriented methodologies and meta-models. It defines an abstract syntax for MAS and relies on model-driven development to transform PIM4Agent models into executable code for agent platforms such as JACK [115] and JADE [20].

A&A (Agent & Artifact) [155] is another meta-model that tries to offer a generic meta-model for MAS. It is based on ideas behind TuCSoN (Tuple Centres Spread over the Network) [156], and relies on two main concepts: agents and artifacts. Agents are proactive entities in charge of the goals and tasks. Artifacts are non-autonomous function-oriented entities providing services and functions to agents (e.g., blackboards, knowledge bases, shared task schedulers). Although A&A

does not provide any direct support for organizations, artifacts can encapsulate organizations, called organizational artifacts [117].

Using Multiple Dimensions to Describe Agent Organizations

Several approaches for agent organizations rely on the use of multiple dimensions or structures to describe organizations [215, 70, 118, 60, 76]. Three common structures are:

- An organization or social structure that defines the organization in terms of roles and how roles relate (e.g., hierarchies and power relations).
- A functional or interaction structure that describes the functional aspects of an organization (e.g., in terms of tasks, goals, or interactions). This structure describes how an organization can, or should, achieve its goals. It often defines reusable patterns of interactions and task divisions. Two common ways to describes this structure are the use of scenes and goal decomposition trees.
- A normative structure or a set of normative rules. This structure defines the rights and obligations of agents in the organization. Such rights and obligations can relate to both the organization structure (e.g., which roles to play), and the functional or interaction structure (e.g., what tasks or interactions to execute, or which goals to achieve).

MASQ (Multi-Agent System based on Quadrants) [76] tries to generalize this idea, by proposing a meta-model that extends the AGR (Agent/Group/Role) model [73] and aims to offer abstractions for all aspects of an organization-centered MAS. MASQ relies on a four-quadrant framework, where the analysis and design of a system is performed along two axes: an interior/exterior dimension and an individual/collective dimension.

Norms and Normative Rules

Several organization models for MAS rely on the notion of norms or normative rules to constrain and structure behavior within agent organizations [69, 118, 60]. Norms are social conventions on how agents should behave and interact with each other [58]. A norm can be an obligation, a permission, or a prohibition, and can be modeled using deontic logic [205], a field of logic concerned with obligation, permission, and related concepts. To constrain the behavior of agents within organizations, norms can be coupled to roles, interactions, and goals [69, 118, 65]. A norm can read like *"when an agent A (1) is committed to a mission M that (2) includes a goal G, and (3) the mission's scheme is well-formed, and (4) the goal is feasible, then agent A is obliged to achieve the goal G before its deadline D"* [116].

Deontic logic, however, does not imply why norms exist or why agents should adhere to then. It is only a language that allows to describe how agents can adopt norms, violate norms, or adhere to them [58]. An important aspect within organizations, therefor, is the enforcement of norms, or making sure that agents adhere to the norms [83, 91, 80]. Norms are typically enforced in two ways, using norm regimentation or using norm enforcement.

Norm regimentation is a pro-active technique. It provides mechanisms to prevent agents from violating norms. Several agent researchers, however, argue that the violation of norms can be functional to a society as a whole, requiring mechanisms in which agents can violate norms [91]. Such mechanisms are called norm enforcement. Norm enforcement is a reactive mechanism, in which a sanction is applied to the agent after it violates a norm.

Normative rules offer an expressive and fine-grained mechanism to control the behavior of agents. From an engineering perspective, however, designing norms, and managing the vast amount of rules that are required to define a complex systems, poses some serious challenges.

Scene-Based and Goal-Based Approaches

Two common ways to describe the functional aspects of an organization in MAS are scenes [69, 60, 18] and goal-decomposition trees [118, 171].

A scene describes a possible interaction between the roles in an organization. Scenes can be combined in a more complex structures that define how an organization achieves it goals. Such a structure consists of a set of scenes and relations between these scenes. Relations between scenes allow to define the order of scenes, synchronization, and parallelism. In addition to these relations, the scene structure can also define how an actor can go from a role in one scene to a role in another scene, and whether the actor is free to choose the next scene. The actual functionality of an organization is realized by agents executing scenes. When comparing scene-based approaches to workflows, the overall scene-structure can be seen as a global workflow and individual scenes as sub-workflows.

When using goal-decomposition trees, the focus is on the division of tasks. The functionality of an organization is decomposed in a set of goals, plans to achieve goals, and sub-goals that make up these plans. To realize the actual functionality, goals have to be assigned to specific agents, roles, or groups. In such approaches, the dominant decomposition is a hierarchy of goals, plans, or tasks, and reuse is in the form of goals, plans, or tasks. Interactions are not modeled explicitly, but are the results of one or multiple goals or tasks that make agents interact.

Scene-based and goal-based approaches are often combined with norms. Norms are used to define which scenes agents can or should execute [69, 60], or which goals agents can or should realize [118]. The chapter on related work (Sect. 7.1) provides a more elaborate discussion of specific scene-based and goal-based approaches.

2.3.2 Organization-Oriented Implementation of Multi-Agent Systems

This difference in agent-oriented and organization-oriented MAS is not only present at design-level, but also at implementation-level [21]. In an agent-oriented implementation of an organization, the focus is on how to develop reasoning mechanisms that allow agents to reason about organization structures, making them 'organization-aware' [74, 64]. This type of implementation is often used in combination with norm enforcement, where 'intelligent' agents have to be able to reason about the norms in a system and decide whether to violate them or not [64].

In an organization-oriented implementation (also called system-centered or institution-centered implementation), the main concern is how to develop the infrastructure that enforces the organization constraints [71, 120]. Such a type of implementation is often used in combination with norm regimentation, where the organization infrastructure makes sure that agents cannot violate any norms of the organization [71, 120].

Organization infrastructures are typically conceived as a three-layered middleware architecture (Fig. 2.2) [21, 71, 120, 209]. The bottom layer is an agent or communication middleware that provides a basic infrastructure for agent communication, perception, and action. The middle layer is the organization infrastructure, providing a set of organization services[2]. The top layer is a domain-specific agent layer that uses the organization services provided by the organization infrastructure.

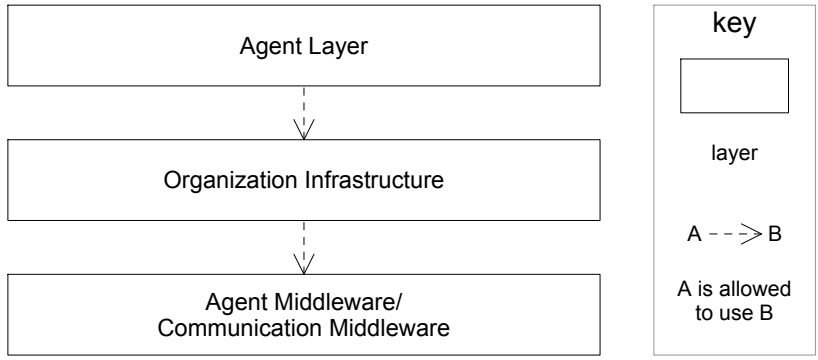


Figure 2.2: A common three-layered middleware used for organization infrastructures.

Other approaches, such as MadKit [96] (based on AGR [73]) and ORA4MAS [117] (based on Moise+ [120]) take a more hybrid approach by combining agent-oriented

[2]Many organization infrastructures also provide an organizational proxy. Such a proxy acts as a mediator between an infrastructure or middleware-managed organization, and an agent.

and organization-oriented implementation techniques. ORA4MAS uses CArtAgO (Common "Artifacts for Agent" Open framework)[3] [174, 173] a framework for implementing artifact-based MAS. Instead of having a traditional organization middleware, organizational artifacts are responsible for norm regimentation, and the detection of norm violations, while organizational agents are responsible for the evaluation, and judgment of norm violations.

2.4 Software Architecture

Software architecture plays an important role in the development of complex software systems. The architecture of a software system can be defined as the essential structures, which comprise software elements, the externally visible properties of those elements, and the relationships between them [17], and the relationships with the environment [4]. Software architecture is a way to deal with complexity and serves as a vehicle of communication with stakeholders for mutual understanding and negotiation. Software architecture manifests the earliest set of design decisions and provides the main structures to realize both the required functionalities and quality attributes.

The documentation of a software architecture plays an important role throughout the lifecycle of a software system. This documentation can range from informal sketches to formal notations, and is is typically structured as a set of *views*. A view is a representation of a set of system elements and the relationships associated with them [17, 4]. It represents a specific perspective on the system with respect to particular concerns. Examples are the layered view, the deployment view, and the 4+1 views from Kruchten [139]. Documenting an architecture is a matter of documenting the relevant views and then adding documentation that applies to more than one view [17]. Although no fixed set of views is appropriate for every system, there are three common views which allow architects to think about software in three different ways [17]:

- Component-and-connector (C&C) Views: allow to structure the system as a set of software elements that have runtime behavior and interactions.
- Module Views: allow to structure the system as a set of implementation units.
- Allocation Views: allow to relate the system to elements in its environment.

Each type of view introduces a set of architectural concepts. We briefly discuss the concepts that are used in the following chapters.

[3] http://cartago.sourceforge.net/

2.4.1 Component & Connector Views

The elements of a Component & Connector (C&C) View are *components* (instances of *component types*) and *connectors* (instances of *connector types*). Components (Fig. 2.3) are the principal computation elements and data stores that execute in a system. They have a set of *ports* (the interfaces of a component) through which they interact with other components via connectors. The component type defines the number and type of ports. Connectors (Fig. 2.3) represent runtime pathways of interaction between two or more components. They embody a protocol of interaction and have a set of *connector roles* (the interfaces of a connector). A connector role defines how a component can use the connector. The connector type defines the number and type of connector roles.

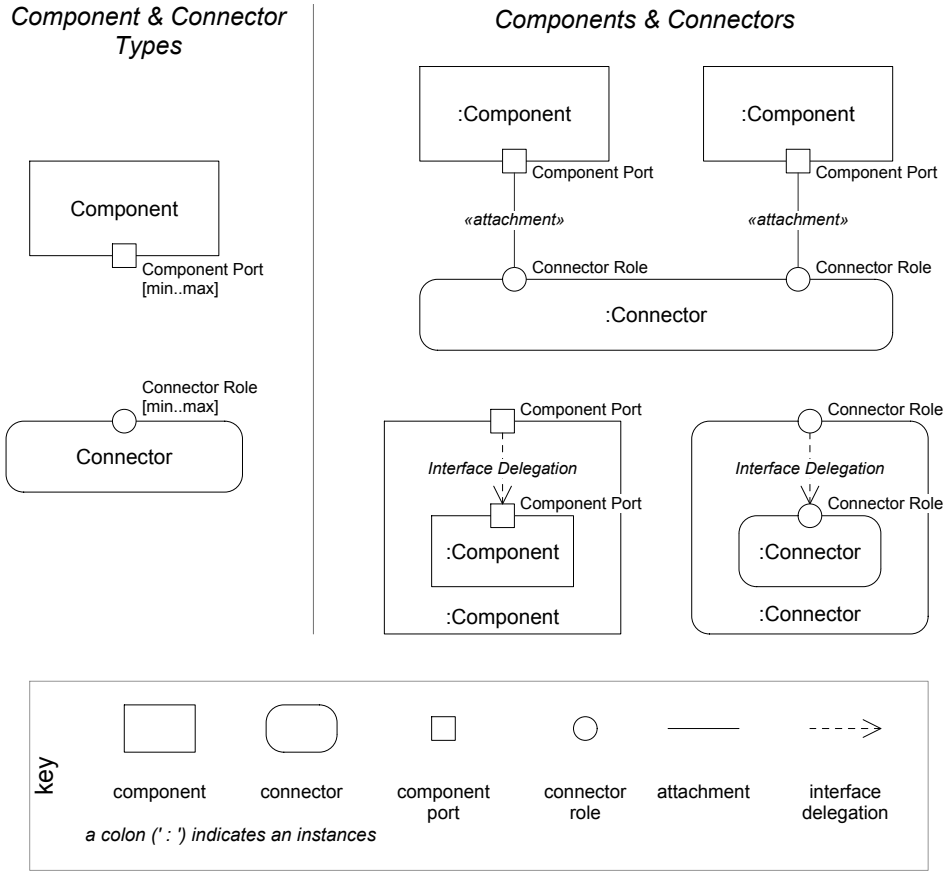


Figure 2.3: The main architectural concepts of the Component & Connector View.

There are two relations in the C&C view: *attachments* and *interface delegations*

(Fig. 2.3). An attachment associates a component port with a connector role, which results in a graph of components and connectors. An interface delegation associates a component port or connector role with a component port or connector role of the 'internal' sub-architecture of a component or connector. This allows to further decompose components and connectors.

2.4.2 Module Views

The main elements of a module view are *modules*. A module is an implementation unit of software that provides a coherent set of responsibilities. There are three types of possible relations between modules: *'is part of'* (defining a part/whole relationship), *'depends on'* (defining a dependency between modules), and *'is a'* (defining a generalization/specialization).

There is not always a one-to-one mapping of component and connector types to modules. Component and connector types represent pre-composed run-time elements that are ready to be deployed. Their implementation can be spread over multiple modules, or a single module can also translate to a set of components and connectors at runtime.

2.4.3 From Abstract Concepts to Domain-Specific Building Blocks

Components and modules can be seen as building blocks to create software architectures. Their true meaning and semantics, however, is only given when they are applied to specific domains. The simplest example of modules and components are classes and objects. More complex examples are JAR (Java ARchive) files and Java beans, and .NET DLL (Dynamic-Link Library) files and .NET components. The semantics of these modules and components is given by the underlying programming frameworks and component models.

2.5 Web Service Technologies and Standards

Web services are the most prominent technology stack to realize service-oriented architecture (SOA). Technologies that belong to this stack are often labeled 'WS-*'. In this section, we discuss some prominent technologies that are used in the following chapters: Web services, the Web Service Description Language (WSDL), SOAP, the Business Process Execution Language for Web Services (WS-BPEL), and the Business Process Model and Notation (BPMN).

2.5.1 Web Service

The term Web service can be used in two ways. One way is to refer to the Web service technology stack in general. The second way is to refer to a piece of software that exposes some functionality, or service, and makes this functionality available through standard Web technologies [12]. A more concrete definition is given by the Web service glossary [97]. According to this glossary, a Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL [43]), and other systems interact with the Web service as prescribed by its definition using SOAP-messages [93]. These SOAP-messages are typically exchanged using HTTP with an XML serialization in conjunction with other Web-related standards. Today, Web services can be implemented and exposed using many standard programming languages, but specialized languages, such as WS-BPEL, also exist.

2.5.2 WSDL

The Web Services Description Language (WSDL) [43] is an XML-based language that can be used to describe the functional aspects of a Web service. A WSDL description, or WSDL definition, has an abstract part and an optional concrete part. The abstract part describes the interface definitions of one or more Web services. This description is similar to a method signature in a programming language. The concrete, optional, part defines a binding of one or more Web services to a set of concrete endpoints, allowing to make actual calls to the Web services. To describe the abstract part, WSDL provides the following concepts:

- **PortType:** Defines a Web service as a collection of provided operations.
- **Operation:** Defines a single action of a Web service, similar to a method or function call in a programming language. An operation is defined as an input, output and fault message. The output and fault message are optional, depending on the type of operation.
- **Message:** Contains the information needed to perform operations. Messages consists of multiple parts of different data types.
- **Type:** Describes a data type used in a message.

A portType describes the interface of a single Web service. Many interactions between systems, however, are bidirectional or have a conversational nature. This means that each interaction partner implements or realizes a Web service and uses the Web service of its partner. To describe such a conversational relation, an extension to WSDL can be used to describe partnerLinkTypes [5]. A partnerLinkType defines up to two roles, each linked to a specific portType. A

conversation participant then implements one portType and uses the other (Fig. 2.4, p. 30, left).

To describe the concrete part of a WSDL definition, the following concepts are available:

- **Binding:** Specifies a concrete protocol and data format for the operations and messages of a particular portType. Examples of bindings are SOAP over HTTP (HyperText Transfer Protocol) and JMS (Java Message Service). A portType can have multiple bindings, based on different protocols and data formats.
- **Port:** Defines a communication endpoint for a specific binding. This endpoint consists of a concrete address or Url where the Web service can be reached.
- **Service:** Defines a concrete service as a set of related ports.

By separating the abstract part from the concrete part, interface definitions can be reused with different types of bindings. This allows to define a Web service independent of its eventual use. A WSDL definition, therefore, has multiple purposes. The abstract part can be used to find Web services that provide certain operations, or to implement a Web service according to the interface specifications. The concrete part of a Web service is used to make an actual call to a Web service. In combination with a service registry, a service consumer that knows the abstract part of the WSDL can use the registry to find a Web service that implements it.

It is important to note that a WSDL definition only describes the different types of operations and related messages, but not the actual business protocol, such as the order of messages. For example, a Web service that exposes operations to place an order, expects messages to arrive in a certain order. To define this business protocol, or the order of messages between two partners, a service choreography can be used.

2.5.3 SOAP

SOAP (originally called Simple Object Access Protocol) is a lightweight protocol for exchanging structured information in a decentralized and distributed environment [93]. It defines an extensible messaging framework, using XML technologies, to provide a message construct that can be exchanged over a variety of underlying protocols, such as Hypertext Transfer Protocol (HTTP) and Simple Mail Transfer Protocol (SMTP). SOAP is independent of any particular programming model, and focuses on simplicity and extensibility. Notable extensions are WS-Security [3], WS-ReliableMessaging [53], and WS-Addressing [94], adding security, reliability, and correlation features to SOAP. SOAP is often used as the foundation layer for the Web services protocol stack.

2.5.4 WS-BPEL

The Business Process Execution Language for Web Services (WS-BPEL) [5], is an XML-based workflow language for Web services. It is one of the most prominent languages to both define and execute service orchestrations and business processes involving Web services. WS-BPEL can also be seen as a specialized programming language to implement Web services. WS-BPEL allows to define two types of processes: abstract processes and executable processes. An abstract processes is partially specified and is not intended to be executed. A typical use of an abstract process is to describe a choreography. Executable processes are fully specified and can be executed on a compatible BPEL engine (e.g., Apache ODE[4]).

PartnerLinks

A BPEL process can both use Web services and expose Web services. The external entities with which a BPEL process interacts are called partners. The concrete interaction points with partners are called partnerLinks (Fig. 2.4). Partners can be service consumers, service providers, or both (e.g., in a conversational interaction). A partnerLink corresponds to a specific partnerLinkType, defined in a WSDL definition. It assigns the roles defined by the partnerLinkType to the BPEL process ('myRole') or to a partner ('partnerRole') (Fig. 2.4, left). The 'myRole' defines a Web service exposed by the BPEL process. The 'partnerRole' represents an external Web service, used by the BPEL process. A partnerLinkType can also define just one role, in which case the corresponding partnerLink represents a non-conversational interaction point in one direction.

Activities and Control Flow.

A BPEL process defines a sequence or workflow of actions that interact with the different partnerLinks, by sending data to partners or receiving data from partners (Fig. 2.4, right). This workflow is defined using BPEL activities. There are two types of activities: basic activities and structured activities. The main basic activities are the following:

- **invoke:** Invokes the Web service of a partner.
- **receive:** Waits for a partner to invoke an operation on a Web service exposed by the process.
- **reply:** Generates a response for a synchronous operation.
- **assign:** Manipulates internal data.

[4] http://ode.apache.org/

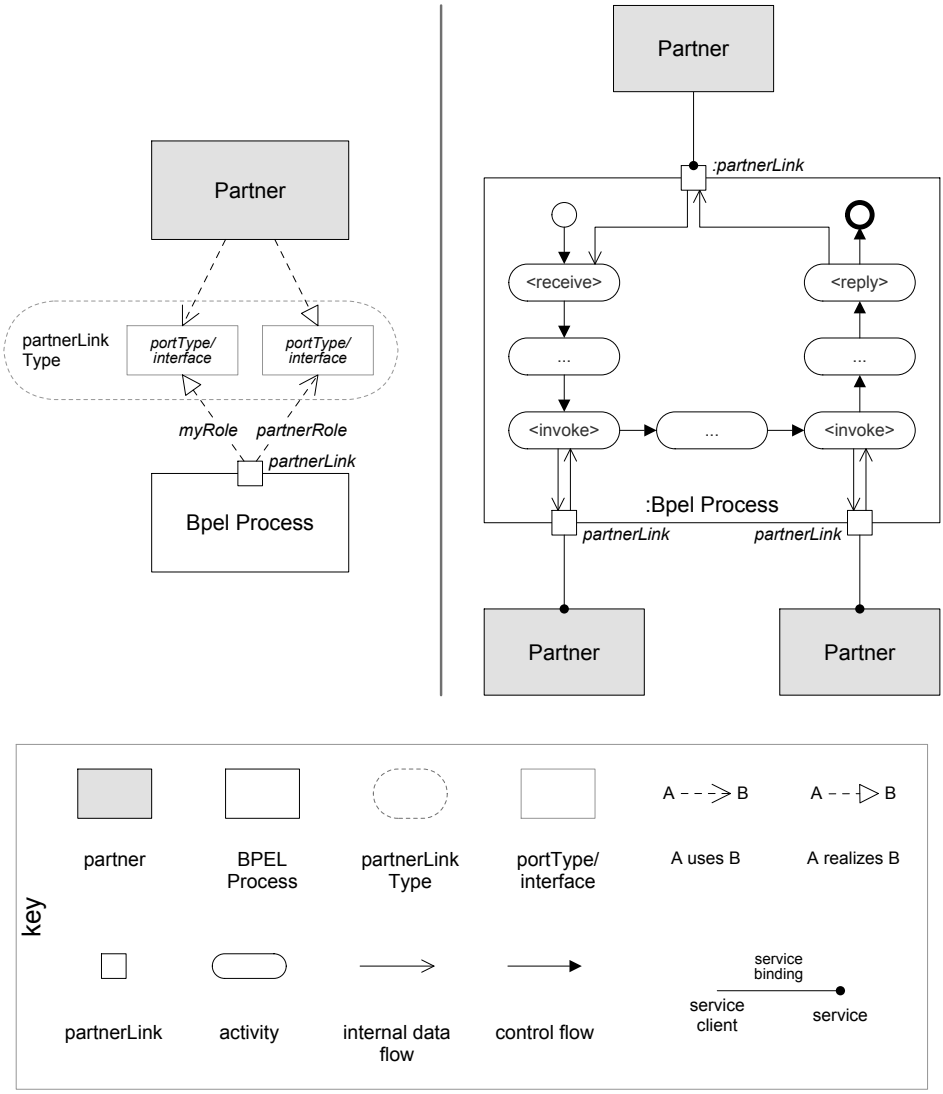


Figure 2.4: *Left:* A partnerLink of a BPEL process implementing the 'myRole' and using the 'partnerRole'. The 'myRole' is used by a partner acting as a service consumer. The 'partnerRole' is implemented by a partner acting as a service provider. *Right:* A BPEL process is specified as a set of activities that define a workflow between the different partnerLinks.

- **throw:** Raises a fault or exception.
- **wait:** Waits for some time.
- **terminate:** Terminates the process.

Structured activities allow to define the control flow of the process. The main structured activities are the following:

- **sequence:** Structures activities to be invoked in an ordered sequence.
- **if:** Defines an if-then-else structure.
- **while:** Defines a conditional loop.
- **flow:** Enables parallel flows in the process.
- **pick:** Selects a path based on the occurrence of a specific event.

Message Correlation

When a BPEL process is deployed on a BPEL engine, there can be multiple process instances that execute in parallel. For example, for an order process that handles incoming orders, there can be one process instance for each order that is received. To relate incoming message to the correct process, BPEL relies on message correlation. A BPEL process can define a correlation set which allows to correlate message to the right process instance based on the content of the message. For example, the order process can correlate the messages to the correct process instance based on the order ID.

Binding PartnerLinks to Concrete Partners

The partnerLinks of a BPEL process have to be bound to concrete partners. This binding can happen both external to the BPEL process (e.g., in the concrete part of a WSDL or by the execution platform), as well as in the process itself. The latter is called a dynamic partnerLink, which allows a process to change it's own partnerLinks. This can be done by retrieving a concrete partner from a service repository or an incoming message.

2.5.5 BPMN

The Business Process Model and Notation (BPMN) [6] is a graphical language, maintained by the Object Management Group (OMG), for specifying business

processes. Its main goal is to provide a notation for business processes that is readily understandable by business users.

BPMN provides four main graphical elements: flow objects, connecting objects, swimlanes, and artifacts. Flow objects (e.g., events, activities, and gateways) allow to define the behavior of a process, and connecting objects (e.g., sequence flow, message flow, and association) allow to connect flow objects to each other and other information. Swimlanes (e.g., pools and lanes) can be used to group elements and to represent participants in a process. Artifacts (e.g., data objects and text annotations) can be used to provide additional information about elements.

BPMN 2.0 supports three main diagrams: process diagrams, collaboration diagrams, and choreographies. A process diagram describes a sequence or flow of activities. It can be used to define both executable and non-executable private processes, and non-executable public processes. A collaboration diagram depicts the interactions between two or more business entities. Each participant is represented as a 'pool'. The interaction is represented as a set of message flows between these pools. Collaboration diagrams are often combined with process diagrams. Each pool in the collaboration diagram is refined as a process (Fig. 2.5). A choreography can be used to define the expected behavior between a set of interacting participants. It is in fact a special type of process diagram. While a normal process exists within a pool, a choreography exists between pools.

BPMN supports decomposition of processes and choreographies in the form of sub-processes and sub-choreographies. Sub-processes and sub-choreographies can be used as a visual aid to manage complexity (i.e., collapsing certain parts of a diagram) or to define a reusable process or choreography definition.

Although the primary goal of BPMN is to provide a graphical presentation of business processes, BPMN 2.0 also defines formal execution semantics. Tools are starting to emerge that provide execution support for BPMN 2.0 definitions (e.g., jBPM[5] and Bonita Open Solution[6]). Partial compliance of BPMN 2.0 with WS-BPEL 2.0 allows to translate a subset of BPMN 2.0 into executable BPEL code.

2.6 Virtual Organizations and Enterprises

The term Collaborative Networked Organization (CNO) is used to represent emerging collaborations between a set of organizations [28]. Well-known examples are Virtual Organizations (VO) and Virtual Enterprises (VE), which are temporary alliances of organizations or enterprises (i.e., commercial organizations) that share resources and skills to achieve a goal. The actual collaboration is realized using computer networks. VOs and VEs are called dynamic or instant if they can

[5] http://www.jboss.org/jbpm
[6] http://www.bonitasoft.com/

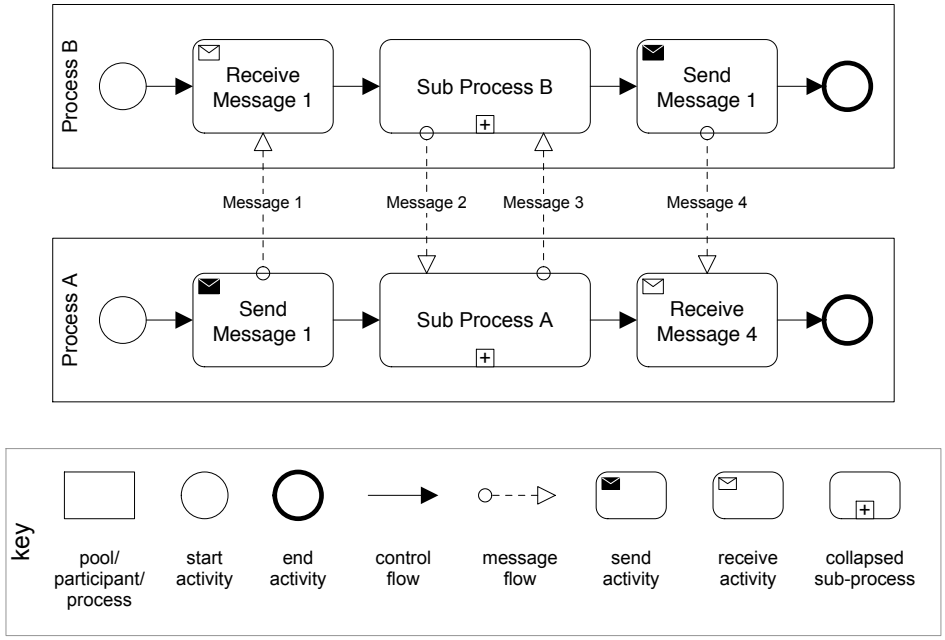


Figure 2.5: An simple example of a collaboration diagram in BPMN, in which each pool is refined as a process.

be established in a short time to quickly respond to changing collaboration opportunities. A special case of a virtual enterprise is the extended enterprise, an organization in which the dominant enterprise extends its boundaries to all or some of its partners. Other examples are Virtual Organization Breeding Environments (VBE), Professional Virtual Communities and Virtual Laboratories [8]. An overview is of types and formation techniques is given in [27] and [29].

Virtual organizations and enterprises cover a very broad domain. It focusses on the business aspects of collaborations as well as the technical aspects of collaborations. PRO-VE[7], for example, a working conference on virtual enterprises, covers topics ranging from self-organizing systems and organizational models, to value creation and asset management in collaborative networks. The concept of virtual organization is often applied to the domain of supply chain management [90], but also to the domain of Grid computing [81]. Agents have been used to both setup and manage virtual organizations [153, 149, 30].

There have been several research projects aimed at the effective formation of dynamic virtual organizations and enterprises. One of the more recent projects is CrossWork [90, 146]. CrossWork has a very broad scope, it aims at supporting

[7] http://www.pro-ve.org/

the entire life cycle of a virtual enterprise, from formation and all related business aspects, to actual operation, evolution and dissolution of a virtual enterprise. CrossWork builds on two previous projects: CrossFlow [89] and MaBE (Multi-agent Business Environment) [125]. CrossFlow focused on concepts and technology for workflow support in dynamic virtual enterprises. MaBE (Multi-agent Business Environment) set out to deliver an agent-based technology infrastructure suitable for implementing business support systems.

Another, but older, project is WISE (Workflow based Internet SErvices) [13]. WISE focusses on providing a platform for process-based B2B e-commerce. In WISE, services offered by participants are considered black-box. They are linked at design time into a workflow process, which is executed by a central workflow engine that controls the cross-organization processes (called virtual business processes).

2.7 Supply Chain Management

In this thesis, we use supply chain management as an application domain to illustrate the main research results. Our focus on supply chain management comes from a larger research project DiCoMas[8]. In this project, together with several academic and industrial partners, we have studied the use of software agents to improve integration and collaboration among supply chain partners.

In this section we first provide some background on the supply chain management domain and introduce a concrete supply chain management case, which we use as a running example throughout the following chapters. In the next section, we will use the supply chain management case to illustrate the main problem statements of the thesis.

2.7.1 Background

A *supply chain* can be defined as "all parties involved, directly or indirectly, in fulfilling a customer request; this includes not only the manufacturer and suppliers, but also transporters, warehouses, retailers, and even customers themselves" [42]. An example is a supply chain network in the food sector (Fig. 2.6). A food supplier (vendor) provides products for a local supermarket chain (customer), which consists of retailers (retailer) and a headquarters (retailer HQ). To provide more efficient replenishment, an inventory is kept in a central warehouse, close to the different retail outlets. The actual supply chain network consists of both product flows and information flows between the different supply chain partners.

Supply chain *management* can be defined as "the coordination of production, inventory, location, and transportation among the participants in a supply chain

[8] http://distrinet.cs.kuleuven.be/projects/dicomas/index.html

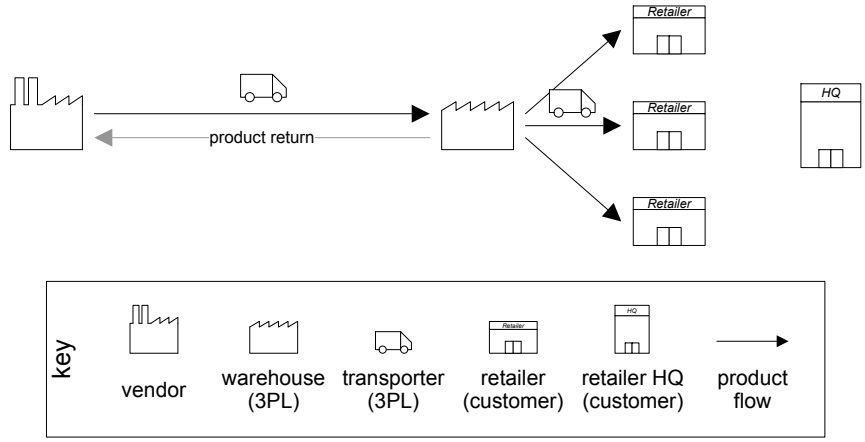


Figure 2.6: An example of a supply chain network in the food sector.

to achieve the best mix of responsiveness and efficiency for the market being served" [121]. Supply chain management relies on several models to realize the actual product flows and information flows in the supply chain network. We briefly discuss some prominent models that are used in the case study and introduce the notion of third-party and fourth party logistics providers (3PLs and 4PLs).

Inventory and Replenishment Models

Two common replenishment models are customer managed inventory (CMI) and vendor managed inventory (VMI). The CMI model represents the classic vendor-customer relation. The customer issues a purchase order to the vendor and the vendor realizes this order. The VMI model, on the other hand, omits the normal order procedure from customer to vendor. Instead, the vendor automatically replenishes the customer's inventory when needed, based on various sources of information, such as inventory levels at the customer's warehouse or point of sales (POS) data [179]. A famous example of VMI is the collaborations between Procter & Gamble and Wal-Mart in the USA, based on monitoring the actual store shelves. VMI can be extended to CPFR (collaborative planning, forecasting and replenishment) [2, 187], which is more comprehensive than VMI and allows to better deal with changing demand patterns (e.g., future promotions). CMI and VMI can be used together with consignment or charge-on-delivery (i.e., non-consignment) models. With consignment, inventory remains property of the vendor, even if it is located at the customer's location, until it is actually consumed by the customer (e.g., the actual sale of products in a supermarket). With charge-on-delivery, the products become property of the customer upon delivery.

Third-Party and Fourth-Party Logistics Providers

The realization of CMI, VMI, or CPFR models in modern supply chains is becoming increasingly complex, requiring extensive support in terms of business, technology and infrastructure [177]. Most supply chain companies lack the knowledge, resources and capabilities to realize such networks on their own, and have to rely on third-party and fourth-party logistics providers (3PLs and 4PLs). A 3PL provides vertical supply chain solutions, that include warehousing, transportation and all other logistics activities. A 4PL, on the other hand, operates horizontally across the whole supply chain, acting as an integrator that assembles the resources, capabilities and services of different supply chain partners and 3PLs, to provide an end-to-end solution for the customer [44]. One of the main roles of a 4PL is to integrate the different information systems of each party involved in the supply chain network. In contrast to a 3PL, which owns actual trucks and warehouses, a 4PL only owns IT systems and intellectual capital.

2.7.2 Running Example: A Supply Chain Management Case

The running example revolves around a 4PL that provides end-to-end integration services for supply chain networks in the food sector (e.g., Fig. 2.6). To restrict the scope of our case study, we make the following assumptions:

- there are collaboration agreements between the different vendors, customers and 3PLs to set up a specific supply chain network;
- vendors, customers and 3PLs have a contract with the 4PL to support these collaborations;
- supply chain partners have a common ontology or mutual understanding (e.g., determined in the collaboration agreement);
- supply chain partners communicate with the 4PL using standard Web service technology.

Even in this restricted setting, there can be a lot of variability in the supply chain network and the services provided by each party. For example, the number of retailers can change, a different replenishment model can be used (CMI or VMI with consignment or charge-on-delivery), or the location of the inventory can vary (central warehouse versus local inventory at retailer's location). In terms of services, one warehouse can offer loading and unloading services, while another does not, or warehouses can offer a different software interface to access their services. Each type of variability leads to a different type of collaboration.

The role of the 4PL is to support and manage the collaboration between the different supply chain partners. The 4PL may have to support multiple supply

chain networks at the same time, each based on a different type of collaboration. A collaboration goes beyond an individual interaction. For example, a collaboration between a vendor, customer and 3PL, includes all the forecasting, replenishment and call-off interactions that take place in the supply chain network. Management of a collaboration includes different activities:

- **Collaboration setup.** The 4PL determines who will collaborate with whom and the type of collaboration.
- **Correct execution of interactions.** After the setup, the 4PL ensures the correct execution of the required interactions.
- **Changing the collaboration.** Once active, a collaboration is likely to change. Participants can change, for example, a 3PL is switched or a retailer is added. Services of participants can change, for example, a vendor updates a service or one of its services becomes unavailable. A collaboration itself can also change, for example, CMI is switched to VMI or the 4PL finds a bug in one of the collaborations and issues an update. Many of these changes can imply a complex change scenario.
- **Terminating the collaboration.** When a supply chain network is dissolved, the corresponding collaboration has to be terminated. This can imply a more complex termination scenario. For example, when termination a supply chain network with consignment, the remaining inventory in the warehouse has to be returned to the vendor.

The 4PL also has a number of additional requirements:

- **Decomposition and modularity.** Collaborations can become complex, and due to variability, several parts of a collaboration will be the same, while others differ. The 4PL wants to be able to compose collaborations from reusable modules.
- **Separation of concerns.** The 4PL wants to be able to introduce new types of collaborations or change existing types (e.g., fixing a bug, changing the responsibility of a participants). To ease this process, different concerns should be separated (e.g., management versus collaboration functionality, responsibilities versus interactions) and changes should not propagate.

Concrete Scenarios

We focus on two concrete scenarios in this supply chain case (Fig. 2.7). Each scenario involves a supermarket chain, a vendor, and some 3PLs providing warehousing and transportation services. The first scenario (*Collaboration A*) uses a VMI model

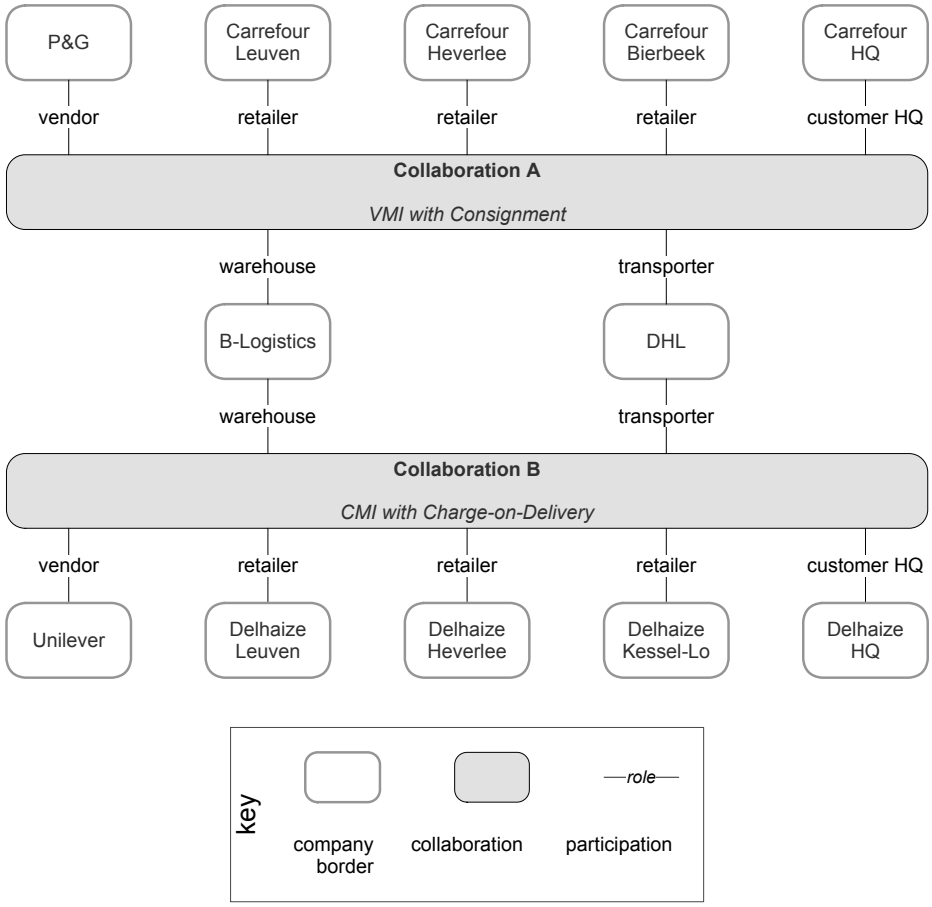


Figure 2.7: Two collaboration scenarios involving two warehouse chains.

together with consignment. The second scenario (*Collaboration B*) uses a CMI model with charge-on-delivery.

Collaboration A is specified as follows (Fig. 2.8):

1. **Forecast.** Customer (retailer HQ) and vendor negotiate on sales and order forecasts, based on historical data and future promotions.
2. **Inventory Report.** The warehouse reports the current inventory status to the vendor on a regular basis.
3. **Replenishment Order.** Based on the inventory status and the sales and order forecast, the vendor issues a replenishment order.

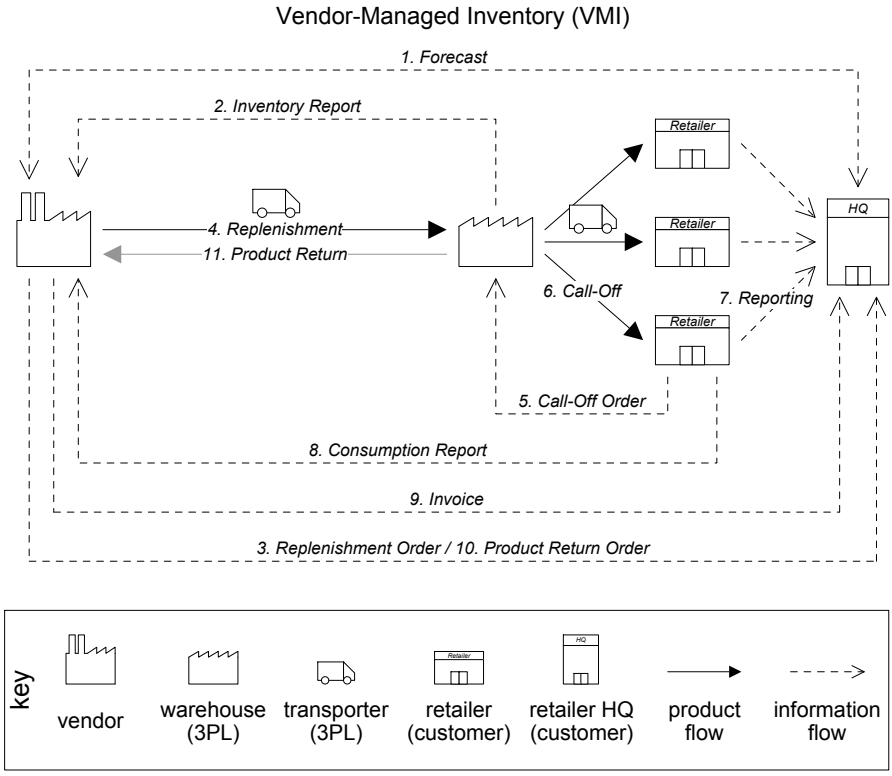


Figure 2.8: Details of the supply chain collaboration based on VMI with consignment.

4. **Replenishment.** The vendor executes the replenishment order, which results in a product flow from the vendor to the warehouse. The transport is realized by a 3PL. The products in the warehouse are still owned by the vendor (consignment).

5. **Call-Off Order.** A retailer is almost out of a certain product and calls off some products from the warehouse.

6. **Call-Off.** The warehouse executes the call-off order, resulting in a product flow from the warehouse to the retailer. The transport is realized by a 3PL. The products are now 'consumed' by the retailer and no longer property of the vendor.

7. **Reporting.** The retailer reports the call-off to the headquarters.

8. **Consumption Report.** The retailer reports the consumption to the vendor.

9. **Invoice.** The vendor generates an invoice which is send to the retailer HQ.

10. **Product Return Order.** Based on the inventory status and the sales and order forecast, the vendor decides to retrieve some products from the warehouse (because they will not be consumed) and generates a product return order.

11. **Product Return.** The product return is executed. The transport is realized by a 3PL.

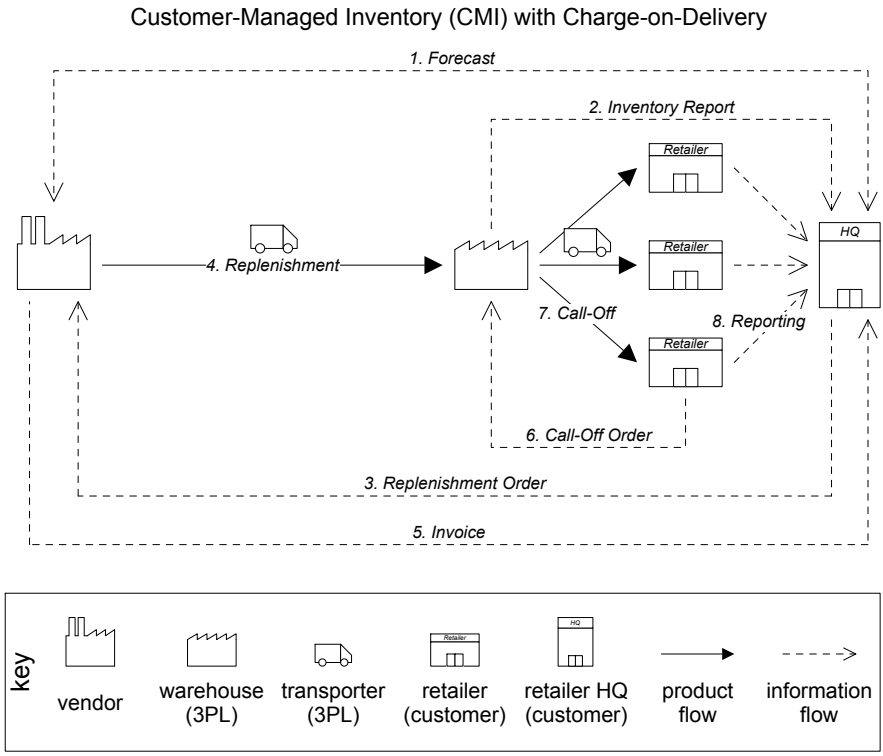


Figure 2.9: Details of the supply chain collaboration based on CMI with charge-on-delivery.

Collaboration B is specified in a similar way (Fig. 2.9):

1. **Forecast.** Retailer HQ and vendor negotiate on sales and order forecasts, based on historical data and future promotions.

2. **Inventory Report.** The warehouse reports the current inventory status to the retailers on a regular basis.

3. **Replenishment Order.** Based on the inventory status and the sales and order forecast, the retailer HQ issues a replenishment order.

4. **Replenishment.** The vendor executes the replenishment order, which results in a product flow from the vendor to the warehouse. The transport is realized by a 3PL. The products in the warehouse are now property of the customer (non-consignment).
5. **Invoice.** The vendor issues an invoice to the customer (charge-on-delivery).
6. **Call-Off Order.** A retailer is almost out of a certain product and calls off some products from the warehouse.
7. **Call-Off.** The warehouse executes the call-off order, resulting in a product flow from the warehouse to the retailer. The transport is realized by a 3PL.
8. **Reporting.** The retailer reports the call-off to the headquarters.

In reality, the different activities (flows) take place in parallel, and each flow of information can be a complex protocol, involving additional parties.

2.7.3 Illustration of Problem Statements in the Supply Chain Management Case

In this section, we look at how our 4PL can use choreography and orchestration to support the supply chain collaborations. We use these examples to illustrate the problems statements of this thesis (Sect. 1.3).

Choreography-Based Collaborations

When using choreography, the 4PL defines and publishes a set of 'public' coordination processes, visible to all supply chain partners. These processes describe the collaboration and corresponding interactions between the different supply chain partners. Since current choreography standards do not provide any decomposition mechanisms[9], the 4PL has to use a single monolithic process or a set of separate processes. This would, for example, result in separate processes for forecasting, replenishment, call-off, and product return. Given the public coordination process, each supply chain partner can translate this process into a private process that implements a part of the public process (Fig. 2.10). Considering the variability in the supply chain networks, the 4PL may publish different variations of each public process (e.g., VMI and CMI, with consignment or charge-on-delivery), which results in different versions of the private processes, called process variability[10].

Using choreography, the 4PL cannot execute any direct control on the supply chain partners. The actual collaboration is the result of each partner executing their

[9]BPMN does provide some limited decomposition concepts (e.g., choreography task and sub-choreography), but it is not defined how this translates or maps to reusable private processes.
[10]Existing techniques to handle process variation are discussed in Sect. 7.2.

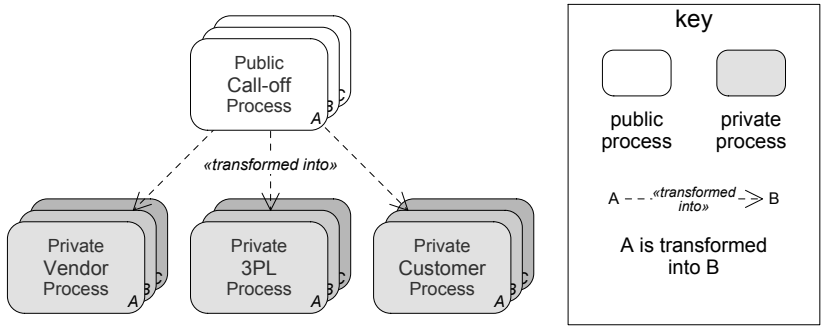


Figure 2.10: Different variations of a public coordination process for call-off transformed into a set of private processes.

private process. This leads to a decentralized setup (Fig. 2.11). The role of the 4PL is reduced to defining and publishing the public process, assisting partners in implementing the private processes, and providing service registries (containing the services exposed by private processes) to set up the collaboration.

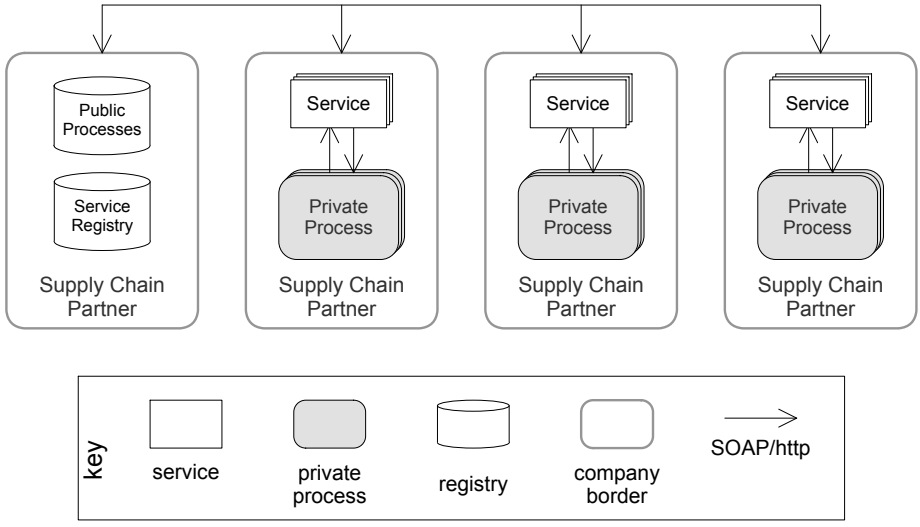


Figure 2.11: Decentralized setup when using choreography-based collaborations.

Since the actual collaboration is executed in a decentralized manner, the private processes can become a lot more complex than the original public process. This can lead to complex peer-to-peer protocols embedded in each private process. A number of researchers are looking into the automatic translation of the public process into

a set of compatible private processes [131]. This would reduce complexity, but such methods are not yet available in current practice.

Another consequence of decentralization is that each supply chain partner is now responsible for the management of the collaboration. Every aspect of collaboration management has to be realized in a decentralized way, requiring additional coordination processes. For example, replacing a 3PL in a supply chain network, with another 3PL, that provides a slightly different service, requires a reconfiguration or even redeployment of private processes in all other supply chain partners.

Key Problems. The key problems with choreography-based collaborations can be summarized as follows:

- Lack of proper decomposition mechanisms leads to monolithic processes.
- A single model addresses all concerns, such as functionality, management, and responsibilities.
- Choreographies typically represent a single interaction. Composing choreographies is not easily supported. There is no standard way to support management beyond a single choreography.
- Decentralization and the lack of central control require complex peer-to-peer protocols.

Orchestration-Based Collaborations

Similar to choreography, the 4PL can use orchestration to define a set of collaboration processes. However, instead of having each supply chain partner implement part of this process, the 4PL can execute and manage these processes itself, and keep control over the correct execution of each process (Fig. 2.12). The local private processes of each supply chain partner become less complex and focus on realizing specific Web services. This setup is similar to the one used by the CrossWorks approach to realize virtual enterprises [90, 146].

Instead of using a monolithic process, that implements the complete process, the 4PL can use sub-processes and composite services to decompose complex supply chain collaborations, again with the necessary variations (Fig. 2.13). At the highest level of this decomposition, there is a global orchestration process, in between a set of reusable business processes, and at the lowest level the actual services provided by the different supply chain partners.

The use of sub-processes and composite services typically leads to a functional decomposition. Such a decomposition does not represent the underlying

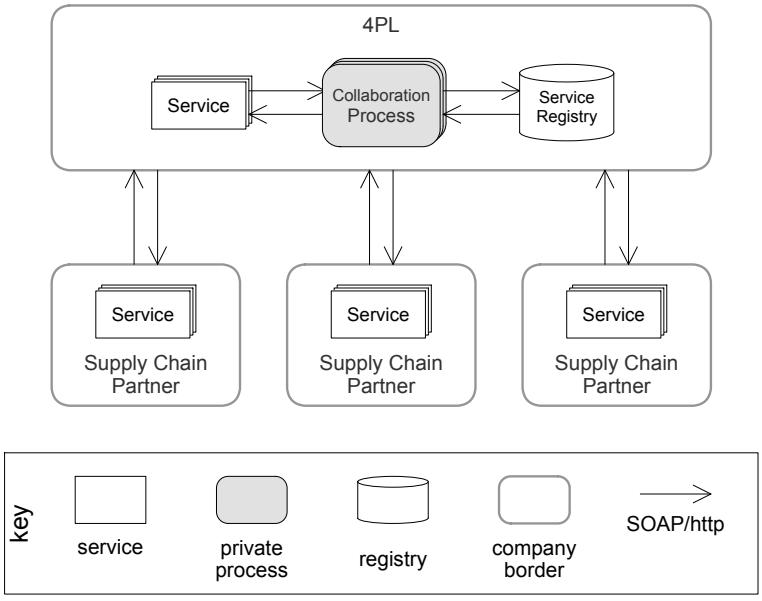


Figure 2.12: Centralized setup when using orchestration-based collaborations.

collaboration structure. As a result, responsibilities and behavior of individual participants are easily scattered over multiple processes and services (Fig. 2.14).

When using orchestration, several management tasks – setup, change and termination – translate to the problem of correctly selecting, binding and invoking sub-processes and participant services. For example, when a retailer (*Carrefour Express*) in *Collaboration A* (Fig. 2.7), wants to do a call-off, the 4PL should start a specific variation of the call-off process that supports VMI, invoke a call-off reporting process that supports consignment, and use the order services of *B-Logistics* (the warehouse in this supply chain network).

The realization of this type of management can be done in several ways [79]. A first option is to embed the management in the process itself. For example, the call-off process defines that in case of CMI, it should use *Call-Off Ordering Process A*, while in case of VMI, it should use *Call-Off Ordering Process B*. Similarly, the process can define, that if *Carrefour* is the customer, it should use the services of vendor *P&G*. Embedding the management in the process itself, however, leads to complex code, which is hard to maintain.

A second option is to put management in external rules and policies, to increase reuse, maintainability and separation of concerns. This can be done using a business rules management system (BRMS), which can be exposed as a service to the processes [176]. For example, when the ordering process needs a concrete

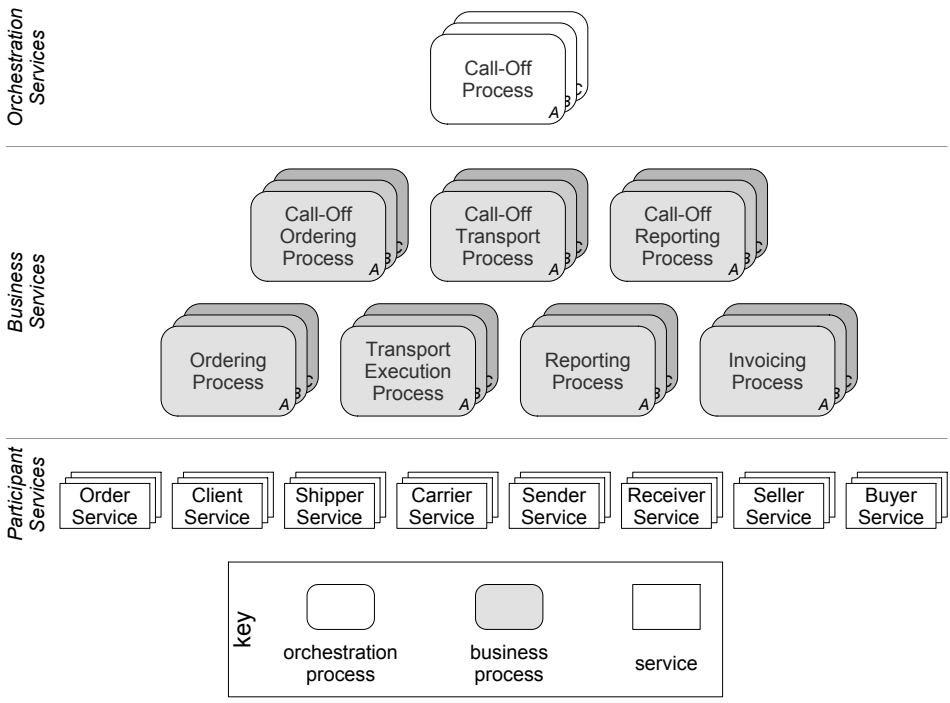


Figure 2.13: Functional decomposition of a collaboration and possible variability.

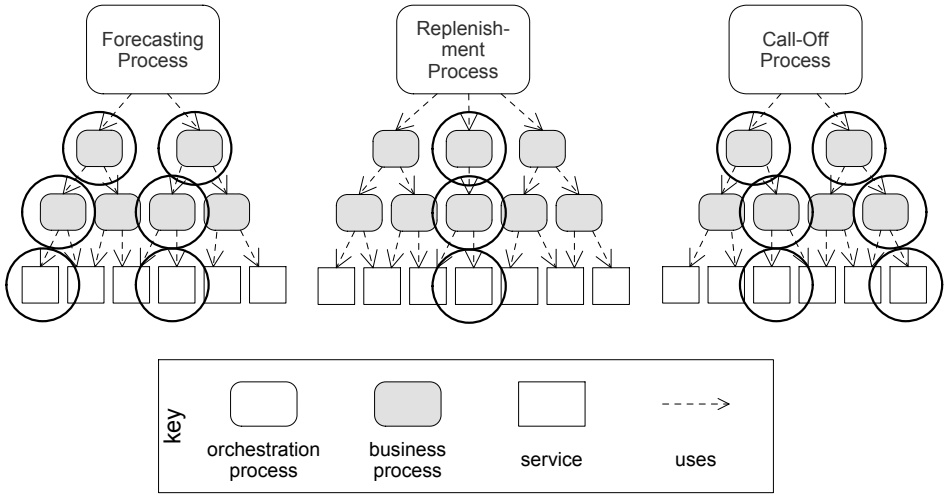


Figure 2.14: An example of responsibilities and behavior of a participant (black circles) being scattered over multiple processes and services.

participant service, it queries the policy service to find the correct endpoint reference. Exposing policies as a service to processes, however, does not make management transparent to the processes. A more recent solution to this problem is the use of a router or a dynamic process assembler together with a BRMS [148, 144]. Instead of querying the policy service, a process just sends all its invocations to the router. The router then determines which endpoints are to be used at runtime based on the incoming request, its context and the current rules and policies (Fig. 2.15).

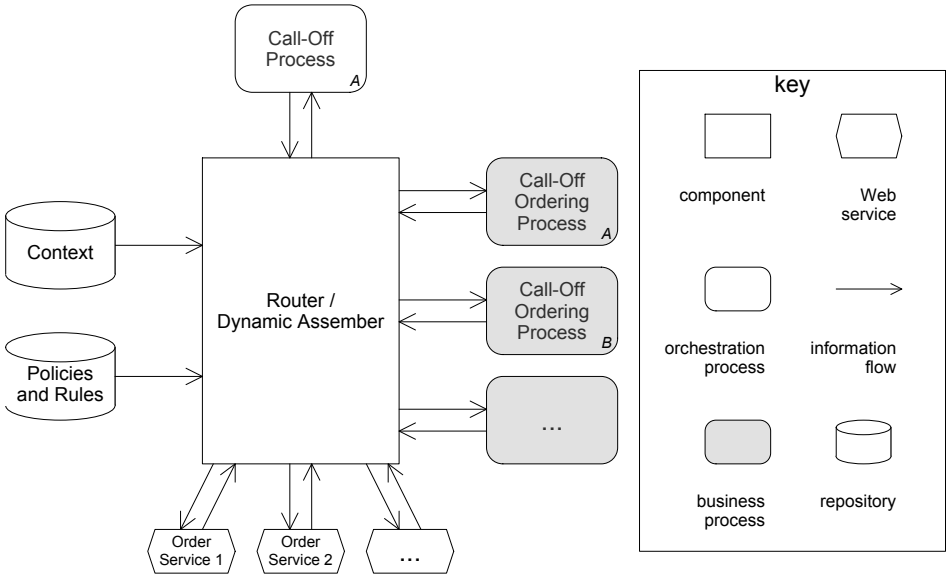


Figure 2.15: Dynamic router in orchestration.

When using policies as a service, routers, or dynamic process assemblers, we are confronted with another problem. Policies and rules are expressed as traditional rules in business rules management systems, which are basically 'if-then' statements. Such rules are useful for typical business policies (e.g., using a different process for 'regular' and 'privileged' users), but make it hard to express the semantics of a collaboration. This results in a complex set of 'if-then' statements, which do not represent any of the collaboration structure. An example is shown below:

```
rule "VMI with Warehouse"
  if
    inventoryLocation == "warehouse" &&
    replenishmentModel == "VMI"
  then
    use "Call-Off Process A"
end
```

```
rule "CMI with warehouse"
  if
    inventoryLocation == "warehouse" &&
    replenishmentModel == "CMI"
  then
    use "Call-Off Process B"
end

rule "VMI with Consignment"
  if
    replenishmentModel == "VMI" &&
    inventoryOwnerShip == "Consignment"
  then
    use "Call-Off Reporting Process A"
end

rule "CMI with Charge-On-Delivery"
  if
    replenishmentModel == "CMI" &&
    inventoryOwnerShip == "Charge-on-Delivery"
  then
    use "Call-Off Reporting Process C"
end

...
```

Key Problems. The key problems with orchestration-based collaborations can be summarized as follows:

- Functional decomposition does not preserve the underlying collaboration structure.
- Participant behavior and individual responsibilities are easily scattered.
- Adaptability (management, dynamics) is addressed in a single model, or when separated, hard to express in terms of collaboration concepts and structures.

Chapter 3

The Macodo Model: A Conceptual Model for Dynamic Collaborations

This chapter presents the Macodo model, a conceptual model that describes abstractions for dynamic collaborations. The model defines the vocabulary for Macodo and serves as a guide for developers and readers of this thesis. The model also provides the foundation for the following chapters. Chapter 4 introduces a set of architectural views that map the collaboration abstractions to architectural modeling abstractions, allowing to design and document collaborations in terms of software elements. Chapter 5 presents a proof of concept middleware infrastructure that maps these software elements to Web service technology and supports the collaboration abstractions at implementation level. The model presented in this chapter is not intended to be used directly by developers. Actual modeling is done using the Macodo architectural views.

3.1 Introduction

Macodo[1] supports the development of dynamic collaborations by providing a set of collaboration abstractions and supporting these abstractions at architectural level and implementation level. An important step in providing such support is to define the meaning and scope of these collaboration abstractions. The Macodo model defines the key abstractions independent of design and implementation concerns. The type of collaborations the conceptual model intends to conceptualize are

[1] Macodo is an abbreviation for Middleware Architecture for COntext-driven Dynamic Organizations.

defined by the scope of this thesis. These collaborations are defined as follows: *"the controlled interchange of information between a set of distributed entities (e.g., Web services) and the controlled execution of related tasks by these entities in order to achieve a set of goals"*. Furthermore, the Macodo model is restricted to collaborations that take place in a restricted environment (see Sect. 1.2).

Most existing organization models [59, 73, 18] start from a top-down approach, trying to model how entities collaborate. Macodo uses a bottom-up perspective. Starting from current state of practice on middleware, BPM, and SOA, it introduces role-based abstractions to improve modularization of service collaborations. To do so, Macodo consolidates earlier research results [207, 101, 102] and borrows concepts from several domains, such as object-oriented modeling [190, 95], BPM [183, 160, 26], and agent organizations [59, 73, 18]. The different perspective of Macodo, however, does not allow to take any existing models 'as-is'.

The model presented in this chapter is the result of several iterations. There are three main drivers behind these iterations:

- The conceptual challenges identified by the problem statements of this thesis:
  - Providing more natural units of decomposition for collaborations that preserve the underlying collaboration structures.
  - Providing better concepts for collaboration management to improve adaptability.
  - Improving the separation of concerns, such as collaboration functionality from collaboration management, and participant responsibilities from interactions.
- The application of the model in an in-depth supply chain management case study.
- Extensive feedback from the initial pilot study of Macodo. In this study, subjects used the Macodo architectural views to design and document collaborations in a software system[2].

Overview. The rest of this chapter introduces the conceptual model in detail. Section 3.2 defines the core abstractions of Macodo. Section 3.3 introduces a number of additional abstractions to better represent the modularization and adaptability of collaborations. After introducing each concept, we give examples using our running example in the supply chain management case.

[2]This study is described in Chapter 6.

3.2 Macodo Core Abstractions

The core abstractions of Macodo are *organization*, *actor*, *role*, *conversation*, and *behavior* (Fig. 3.1). Organizations define reusable collaborations as a set of roles and conversations between these roles. Actors represent entities in the environment capable of playing roles. Each role defines a coherent set of rights and responsibilities in the organization. These rights and responsibilities can range from executing behaviors to participating in conversations with other roles.

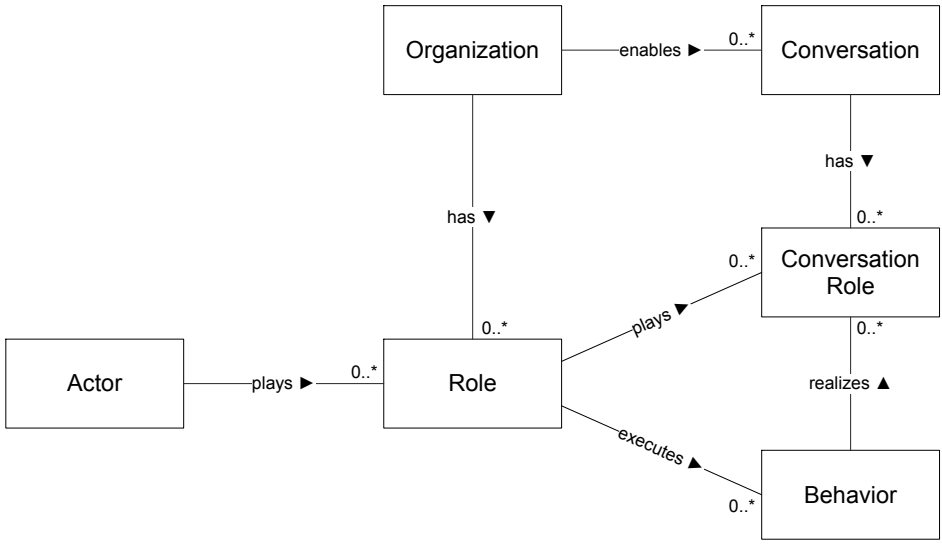


Figure 3.1: The core Macodo abstractions and their relations.

3.2.1 Organization

An organization represents a collaboration between a set of actors. It is defined as a set of roles, played by actors, and the conversations between these roles. Roles define the rights and responsibilities in the collaborations, and conversations represent the interactions between these roles. Organizations can be created and destroyed and there can be multiple organizations of the same type. By allowing the existence of multiple concurrent organizations, unlike many existing organization models, Macodo allows to represent multiple concurrent collaborations taking place within a single system. Organization provides a concept to represent collaborations at a higher level of abstraction. Hierarchies of organizations are not explicitly supported, but can be realized in an application-specific manner.

Example. The supply chain case introduced two types of supply chain collaborations: a VMI-based supply chain (Fig. 2.8, p. 39) and a CMI-based supply chain (Fig. 2.9, p. 40). Both collaboration types can be defined as an organization: the *Vmi Organization* and *Cmi Organization*.

The running example (Fig. 2.7, p. 38) can then be modeled as two concrete organizations: *orgA* and *orgB* (Fig. 3.2).

To support other supply chains, the 4PL can create additional organizations. When supply chains no longer need to be supported, the 4PL can destroy the corresponding organizations.

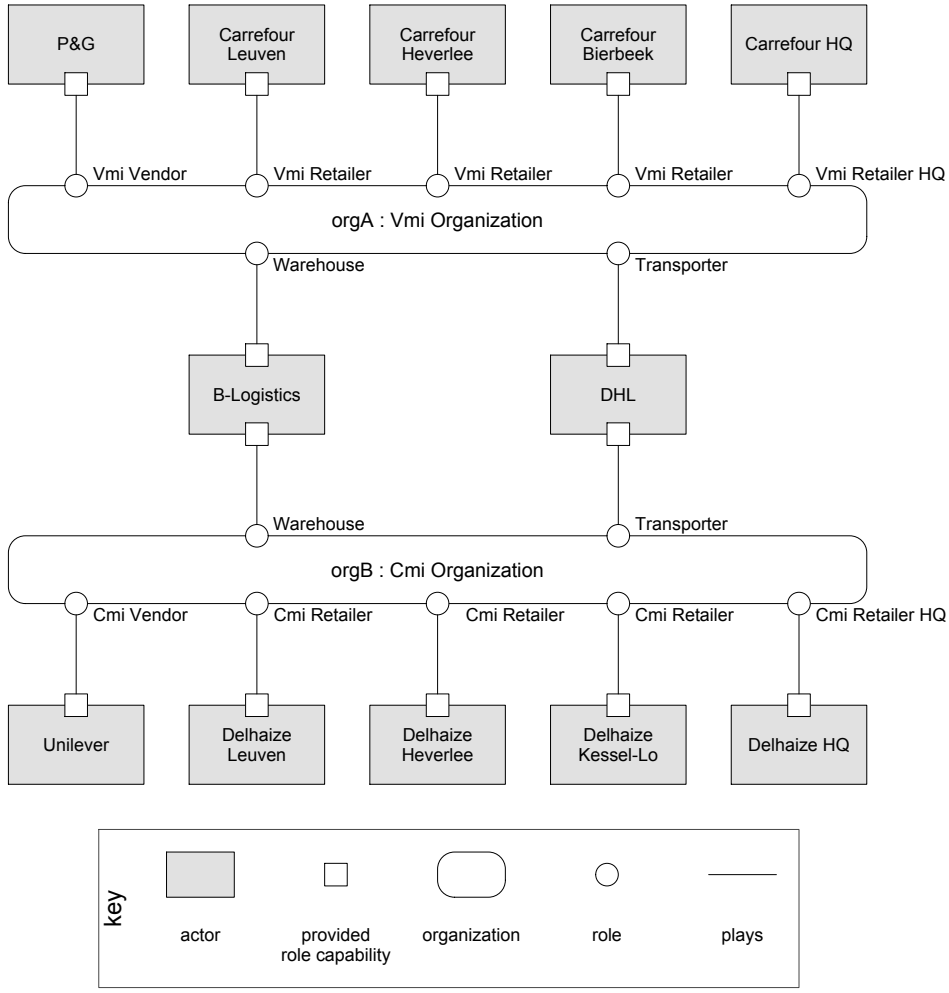


Figure 3.2: Mapping the collaborations of the supply chain example (Fig. 2.7, p. 38) to organizations (capabilities are explained in Sect. 3.3.3).

3.2.2 Actor

An actor represents an entity in the environment capable of participating in a collaboration, by playing a role in an organization. In a concrete system, actors can be computer systems, business entities, software agents, services, or even people. The concept of actor allows to make abstraction of collaboration participants and focus on the collaboration itself.

Example. The supply chain partners and their systems in the running example (e.g., *P&G* and *Carrefour Leuven*) can be represented as actors that play roles in one or more organizations (Fig. 3.2).

3.2.3 Role

A role represents a coherent set of rights and responsibilities inside an organization. These rights and responsibilities are defined in terms of conversation roles the role can play, and behaviors the role can execute. Conversation roles allow the role to interact with other roles. Behaviors describe additional role functionality, such as the execution of tasks, or how to realize specific conversation roles.

Roles are played by actors, and can be reused across organizations. To join an organization, an actor starts playing a role. To leave the organization, the actor stops playing the role. Roles can have a cardinality greater than one. This means that there can be multiple role instances of the same role type in a single organization. Each role instance has a unique identity. This allows multiple actors to play the same role, while still being able to address each role instance individually.

The Macodo model makes an explicit distinction between roles, conversation roles, and behaviors. An alternative would be to unify the concepts of role and conversation role, or even behavior. Roles in Macodo, however, encapsulate responsibilities and functionalities that go beyond individual conversation roles and behaviors. They allow to combine multiple conversation roles and behaviors as a reusable unit, improving modularization and reuse in complex collaborations.

Example. The *Vmi Organization* (Fig. 3.2) defines five types of roles. The roles of *Warehouse* and *Transporter* are used in both the *Vmi Organization* and *Cmi Organization*.

In the running example, each supply chain partner plays a role in the corresponding organization (Fig. 3.2). The role of *Vmi Retailer*, for example, is played by multiple actors in *orgA*.

The roles in an organization are not static. The 4PL can create and destroy roles as supply chain partners join and leave supply chain collaborations. At *T1*, for example, there are two retailers in *OrgA* (Fig. 3.3). At *T2*, *Carrefour Bierbeek* has joined as retailer, and a new *Vmi Retailer* role is created. At *T3*, *Carrefour Heverlee* is no longer a retailer, and the corresponding role is destroyed.

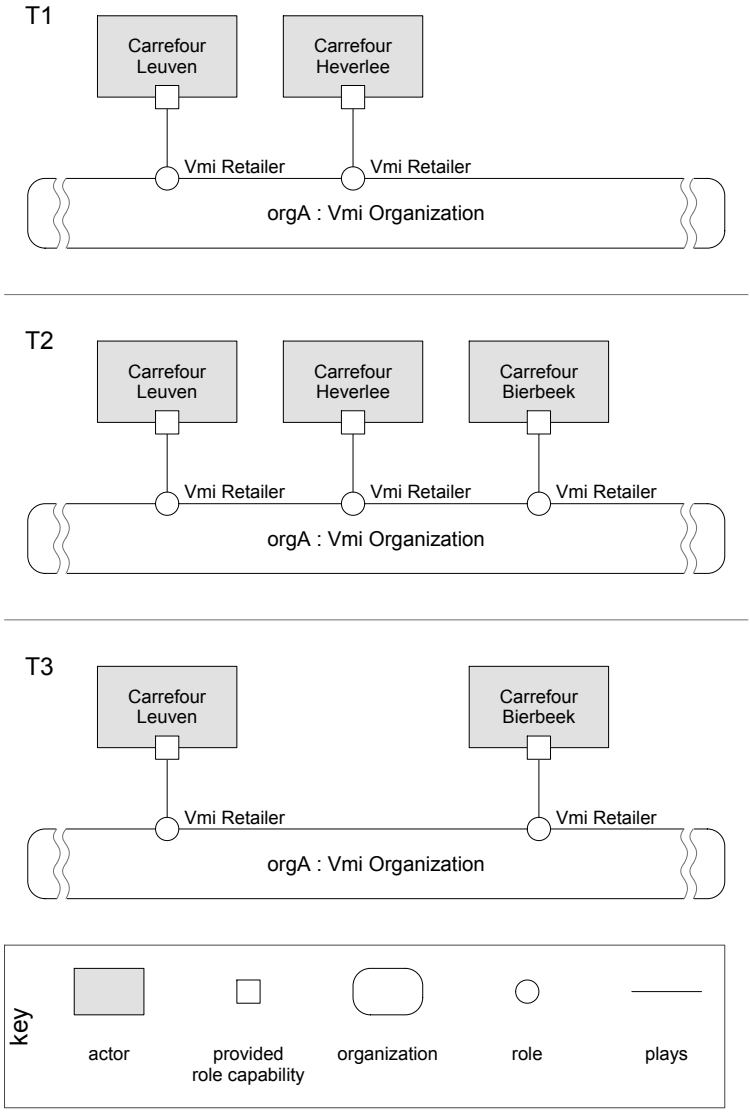


Figure 3.3: The roles in an organization can change over time. The 4PL can add and remove retailers from the supply chain organizations.

3.2.4 Conversation

A conversation represents a reusable interaction inside an organization. It is defined as an interaction protocol and a set of conversation roles. The interaction protocol is application-specific and can, for example, specify the type and order of messages exchanged between the conversation roles, or possible interactions states. The conversation roles are played by roles of an organization and represent specific responsibilities in the interaction. A conversation role can be an initiator or participant of the interaction, and can have a cardinality greater than one.

Example. To support the interactions in the *Vmi Organization* and *Cmi Organizations*, we define a set of conversations, such as *Inventory Reporting Conversation*, *Call-Off Conversation*, and *Transport Conversation*. Each type of conversation defines a set of conversation roles and an interaction protocol. The *Inventory Reporting Conversation* and *Call-Off Conversation* (Fig. 3.4), for example, each define two conversation roles. In the *Inventory Reporting Conversation*, *Inventory* is the initiator, with a cardinality of one, and *Client* is participant with an unlimited cardinality.

The interaction protocol of the *Call-Off Conversation* defines that the caller first sends a call-off order to the stock, which can be accepted or rejected. If accepted, the stock sends a confirmation and the delivery info. The caller sends a delivery report after the call-off is completed.

The possible conversations in an organization are defined by the organization and not by the roles. Roles only define which conversation roles they can play. This keeps roles independent of each other and allows both roles and conversations to be reused across organizations. Together with roles, conversations provide a natural decomposition mechanism for collaborations.

Example. The *Vmi Organization* enables multiple conversations between its roles (Fig. 3.5). To realize inventory reporting, for example, the *Warehouse* role can initiate the *Inventory Reporting Conversation* with the *Vmi Vendor* and *Vmi Customer HQ* role (both roles play the *Client* conversation role).

The *Cmi Organization* also uses the *Inventory Reporting Conversation* (Fig. 3.6), but enables it between the *Warehouse* role and the *Cmi Customer HQ* role.

Within the *Vmi Organization*, the *Transport Conversation* can be used to realize call-off, replenishment, and product return.

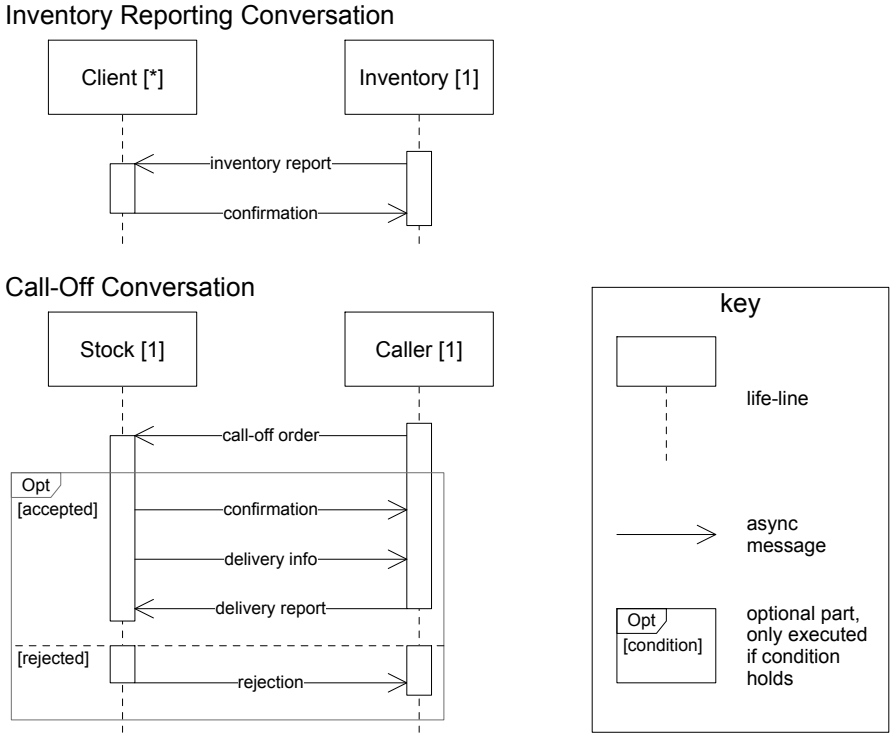


Figure 3.4: The *Inventory Reporting* and *Call-Off* conversation are defined as a set of conversation roles and an interaction protocol.

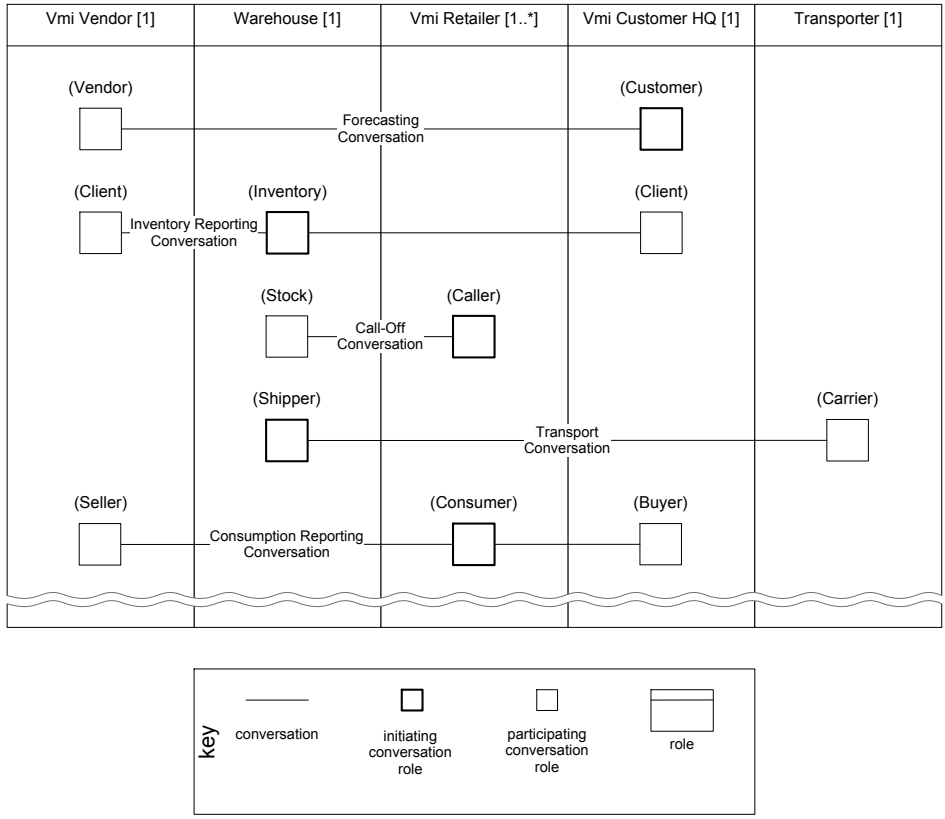


Figure 3.5: A subset of the possible conversations in the *Vmi Organization* to enable forecasting, inventory reporting, and product call-offs.

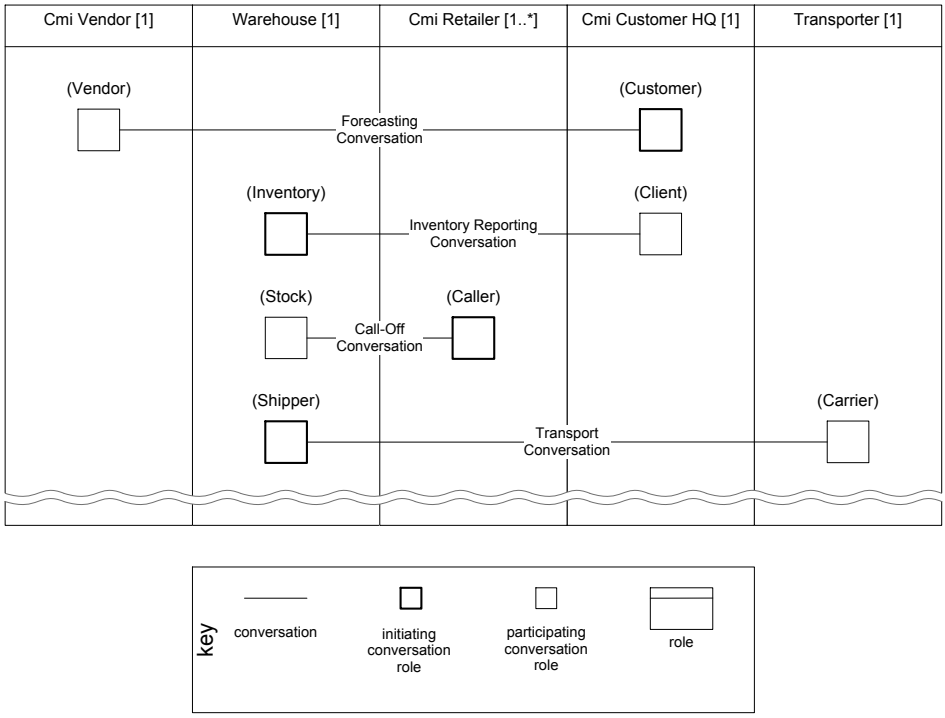


Figure 3.6: A subset of the possible conversations in the *Cmi Organization* to enable forecasting, inventory reporting, and product call-offs.

3.2.5 Behavior

A behavior represents a coherent part of reusable role functionality. Behaviors describe how a role should mediate the actions of its actor with the rest of the organization. These actions can range from executing particular tasks, to realizing specific conversation roles. Behaviors are executed by roles. The execution of a behavior can be initiated by the actor of a role, a conversation, or the role itself. Like a conversation, the internal realization of a behavior is application-specific. A behavior can, for example, be realized as a business process or BPEL process that orchestrates the actions of the actor with different conversations in which the role participates.

Behavior provides a concept to decompose role functionality in addition to individual conversation roles. Conversations and conversation roles encapsulate concerns related to specific interactions. Behaviors encapsulate additional concerns, such as the execution of tasks, or concerns that cross-cut multiple conversation. This allows to reuse behaviors and conversations across roles and organizations.

Example. The *Warehouse* role defines two behaviors: the *Inventory Reporting Behavior* and the *Call-Off Fulfillment Behavior* (Fig. 3.7). The *Inventory Reporting Behavior* is initiated by the actor. This behavior can be seen as an adapter for the conversation role *Stock*. It defines how to collect the required information on inventory levels and how to pass it to the conversation role. By separating the behavior from the conversation role, we separate two concerns: the collection of inventory data, and the distribution of inventory data among clients. This allows to change the way data is collected, independent of the distribution, and vice versa. It also allows to reuse the conversation with warehouses that collect data in a different manner.

The *Call-Off Fulfillment Behavior* is initiated by a *Call-Off Conversation* and can initiate a *Transport Conversation*. This behavior encapsulates a responsibility beyond an individual interaction: the responsibility of the *Warehouse* Role to initiate a transport interaction to fulfill the call-off. A representation of this behavior is given in Fig. 3.8. The vertical lanes represent the actor of the role and the conversations in which the behavior participates (as seen from the perspective of the behavior). The behavior starts by receiving a call-off order from from the call-off conversation. Next, it sends this call-off order to the actor of the role, who responds with a confirmation. This confirmation is passed back to the call-off conversation, and so on.

Other behaviors encapsulate similar responsibilities. The *Call-Off Behavior* of the retailer role, for example, initiates a *Consumption Reporting Conversation* after a successful call-off.

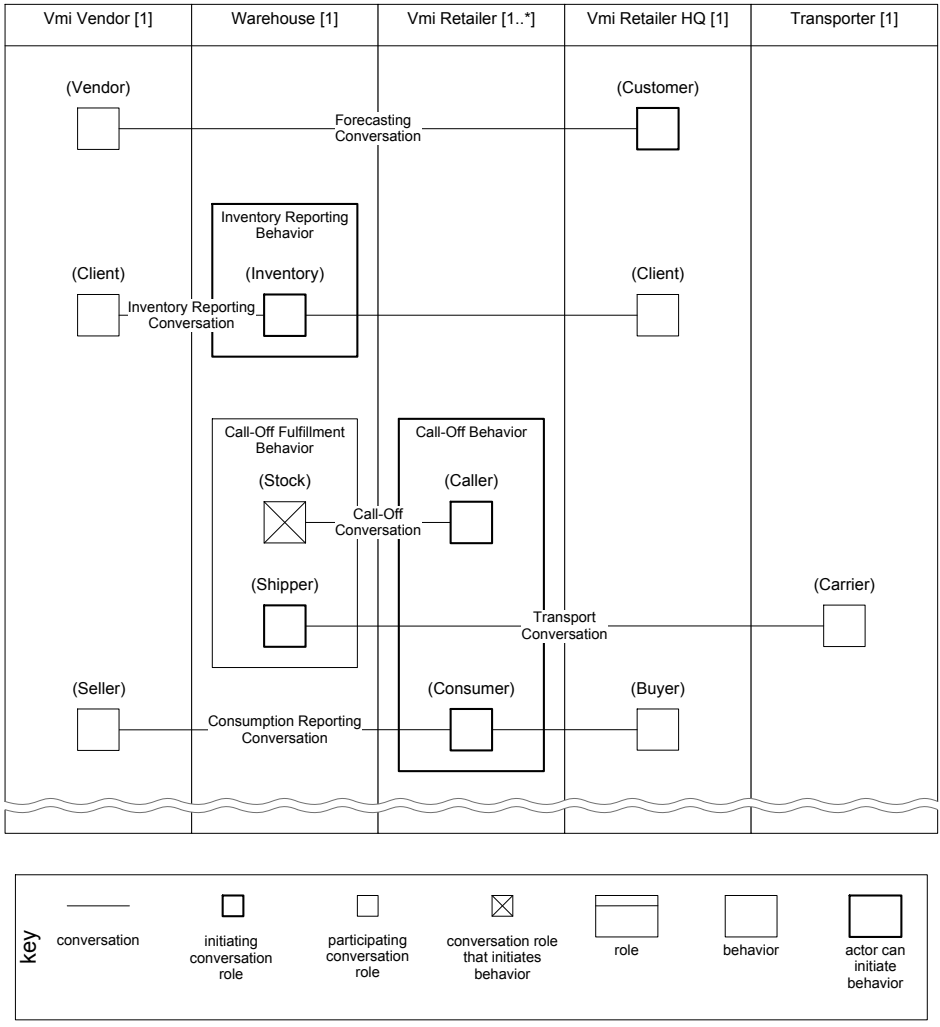


Figure 3.7: Roles, behaviors, and conversations in the *Vmi Organization*. Roles can play conversation roles and execute behaviors. Behaviors can also realize conversation roles.

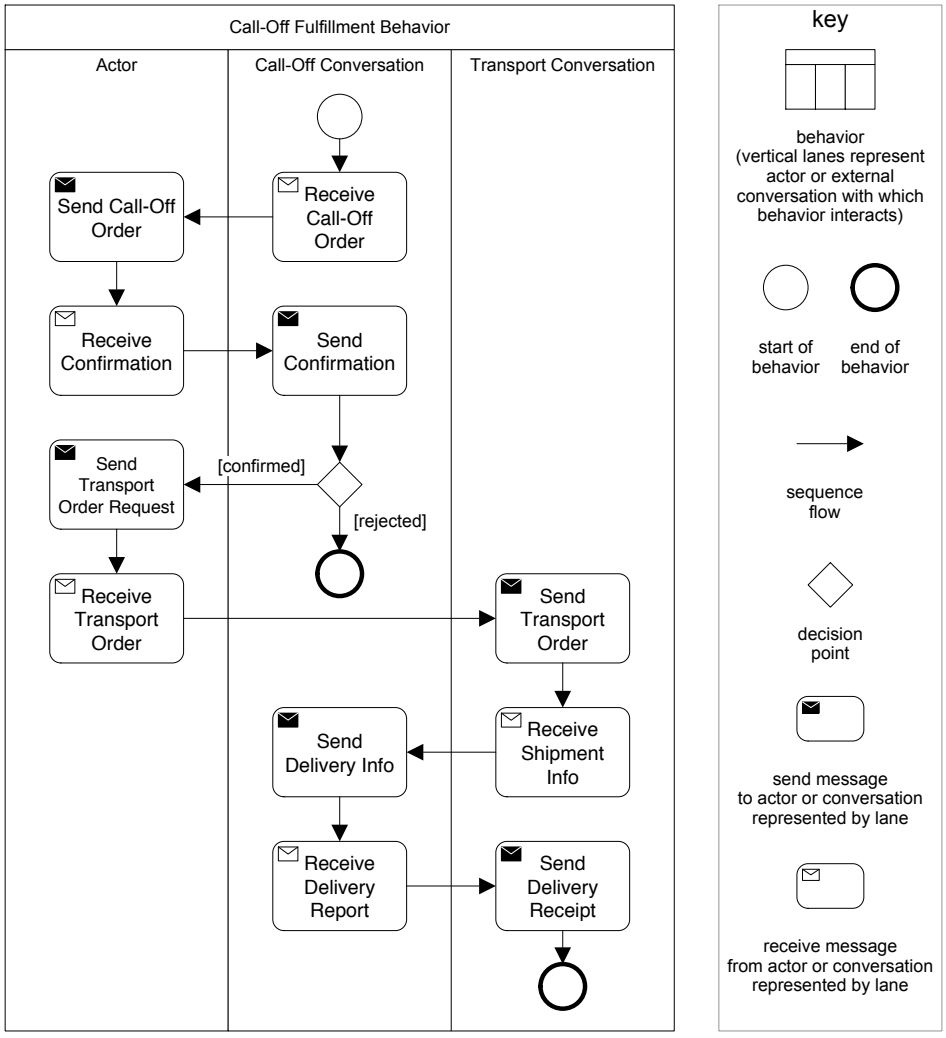


Figure 3.8: A representation of the call-off fulfillment behavior. The vertical lanes represent the actor of the role and the conversations in which the behavior participates (as seen from the perspective of the behavior).

3.3 Additional Abstractions

To better represent the modularity and adaptability of collaborations, we introduce a set of additional concepts and abstractions. The abstraction of *role state* (Sect. 3.3.1) allows to describe the adaptability of individual roles. Organization Dynamics (Sect. 3.3.2) describes how role states together with the other abstractions can be used to realize the adaptability of an entire collaboration. Finally, the abstraction of *capability* (Sect. 3.3.3) provides a way to decouple actors, roles, behaviors, and conversations to improve reuse and modularity.

3.3.1 Role State

The concepts of organization and role allow to create and terminate collaborations and to add and remove participants. These concepts, however, are insufficient to terminate collaborations or remove participants in a controlled manner. Terminating organizations or removing participants in the presence of active conversations and executing behaviors can lead to inconsistencies within a collaboration.

To address this problem, the model introduces the abstraction of *role state* (Fig. 3.9). Every role is always in a specific state, that can change over time. This state constrains the conversation roles a role can play and the behaviors a role can execute. The concept of role state allows to introduce role life-cycles that put roles in a safe state, before terminating a collaboration or removing a specific role.

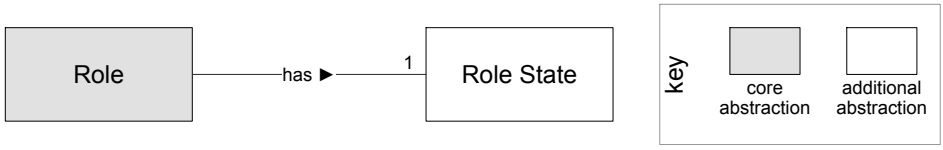


Figure 3.9: A role is always in a specific role state.

Example. To support the addition and removal of a *Vmi Retailer* role, we introduce three possible role states (Fig. 3.10, top): *Active*, *Deactivating*, and *Inactive*. Each state defines which conversation roles and behaviors can be initiated and how many concurrent conversation roles and behaviors there can be.

In the *Active* state, all conversation roles can be played and all behaviors can be executed. In the *Deactivating state*, no new conversation roles or behaviors can be started, but there can still be active conversation roles and behaviors. In the *Inactive* state there can be no active conversation roles or behaviors.

Using these three role states, a possible life cycle of the *Vmi Retailer* role can be as follows (Fig. 3.10): creating a retailer role puts it in the *Inactive* state, activating the role puts it in the *Active* state, initiating the deactivation behavior puts it in *Deactivating* state, and after all interactions and behaviors have terminated, the role returns to the *Inactive* state.

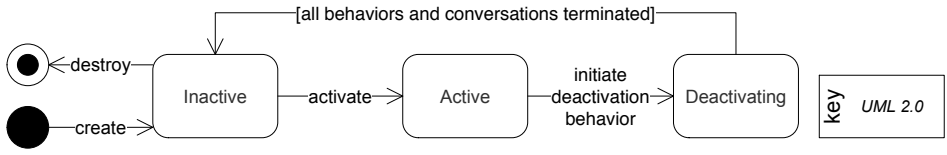


Figure 3.10: A possible life cycle for the *Vmi Retailer* role based on three role states.

The Macodo model does provide any concepts for conversation states and behavior states. The motivation for this, is that while roles are specific to the Macodo model, conversations and behaviors, used in real systems, are often application specific and can take many different forms. As a result, introducing a general concept of state for conversations and behaviors would limit the applicability of the Macodo model. When concretely modeling conversations and behaviors, using the architectural views, or implementing organizations using supporting middleware, application-specific notions of state can be introduced.

3.3.2 Organization Dynamics

Collaborations are typically not static. Collaborations can be set up and destroyed, and participants can join and leave, execute new behaviors, and start new conversations. To support this type of adaptability, the Macodo model defines a set of basic organization dynamics:

- **Creating and destroying organizations.** Organization can be created and destroyed. Destroying an organization destroys all roles, behaviors and conversations within the organization.
- **Creating and destroying roles.** Roles can be created and destroyed. Creating a role assigns the rights and responsibilities to a specific actor in a particular organization. The actor and organization of a role cannot change during the lifetime of the role. Destroying a role terminates all behaviors of the role.
- **Evolving role states.** The state of a roles can evolve. Once changed, the behavior and conversation constraints defined by the role state apply. A role is always in one state.
- **Initiating and Terminating Conversations.** Conversations can be initiated and terminated. Roles can only play conversation roles that are available in their current state.
- **Initiating and Terminating Behaviors.** Behaviors can be initiated and terminated. Roles can only execute behavior available in their current state.

3.3.3 Capability

Actors can play roles, roles can play conversation roles and execute behavior, and behaviors can also realize conversation roles. Defining roles and behaviors with explicit references to each other or to conversations creates dependencies. These dependencies can can limit reuse and modularity. To address this problem, the model introduces the abstraction of *capability*.

A capability describes the requirements to play a role, play a conversation role, or execute a behavior. In a concrete system, these requirements could translate to a set of required and provided interfaces, and a corresponding usage protocol. Capabilities can be used as an intermediate to describe the dependencies between the different collaboration concepts. There are three types of capabilities (Fig. 3.11).

Conversation roles require a conversation capability to be played. Behaviors require a behavior capability to be executed. Behaviors, however, can also provide one or more conversation capabilities. This allows behaviors to realize conversation roles. A behavior providing a certain conversation capability can thus play any conversation role that requires this capability. This allows a behavior to be reused with different types of conversations.

A role requires a role capability. This capability specifies the requirements to play a role and is defined as a set of conversation capabilities (one for each conversation role the role can play that is not realized by a behavior) and a set of behavior capabilities (one for each behavior the role can execute). Actors can provide such a role capability. This means that they are capable of playing the conversation

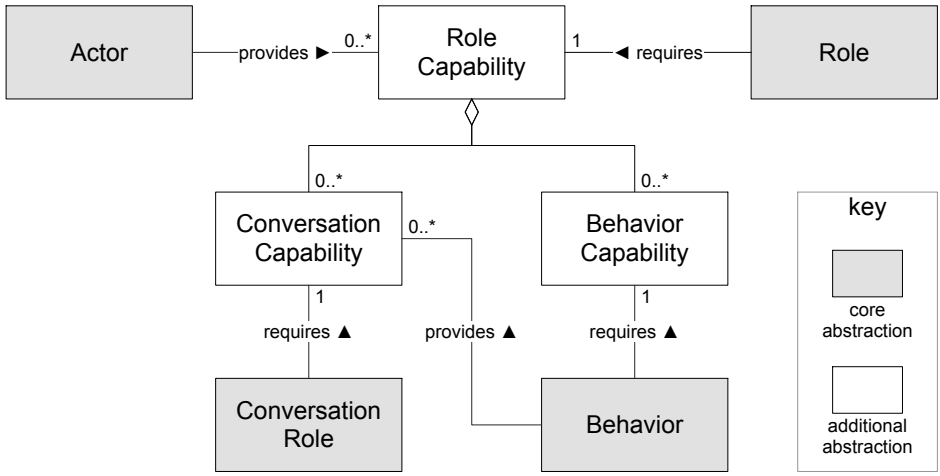


Figure 3.11: Conversation roles require a conversation capability to be played. Behaviors require a behavior capability to be executed, but can also provide conversation capabilities to realize certain conversation roles. A role requires a role capability, which can be provided by an actor. This role capability consists of the conversation capability for each conversation role the role can play that is not realized by a behavior, and the behavior capability for each behavior the role can execute.

roles of the role, and executing the behaviors of the role. Actors providing a role capability can play any role requiring this capability. This avoids any direct dependency between actors and roles.

Example. The *Vmi Retailer* role requires a role capability, which consists of a conversation capability and a behavior capability (Fig. 3.12). This capability is provided by three actors, allowing them to play the *Vmi Retailer* role.

The different conversation roles require a conversation capability. The *Client* conversation role of the *Inventory Reporting Conversation*, for example, requires the *Client Capability*.

The *Call-Off Behavior* requires one behavior capability and provides two conversation capabilities. The *Caller Capability* allows the *Call-Off Behavior* to play the *Caller* conversation role of the *Call-Off Conversation*. The *Consumer Capability* allows the *Call-Off Behavior* to play the *Consumer* conversation role of the *Consumption Reporting Conversation*. Note that the *Call-Off Behavior* can be reused with any conversation that has conversation roles requiring the provided conversation capabilities.

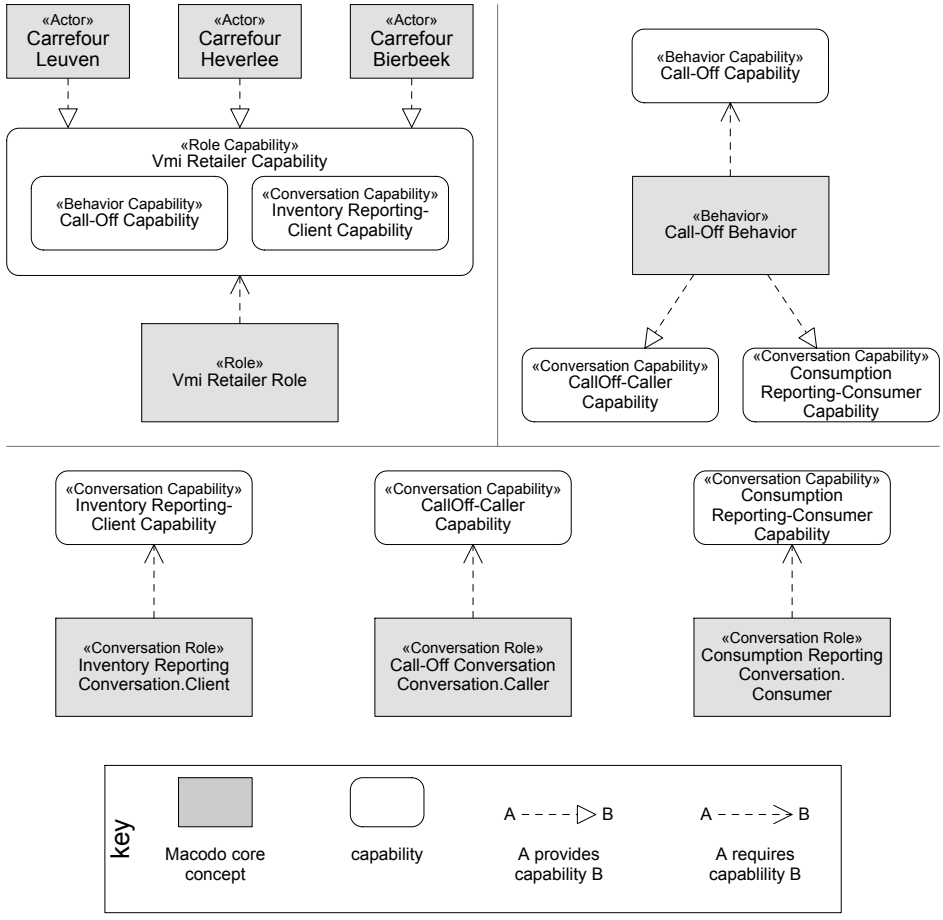


Figure 3.12: An example of actors, roles, behaviors, and conversation roles requiring and providing capabilities.

3.4 Conclusions

This chapter presented a conceptual model for dynamic collaborations. It introduced a set of collaboration abstractions that define the vocabulary for Macodo and provide a foundation for the subsequent chapters. The model, however, is not intended to be used directly by developers. Actual modeling is done using the Macodo architectural views.

The core abstractions of the model are *organization*, *actor*, *role*, *conversation*, and *behavior*. Organizations define reusable collaborations as a set of roles and conversations between these roles. Actors represent entities in the environment capable of playing roles. Each role defines a coherent set of rights and responsibilities in the organization. These rights and responsibilities can range from executing behaviors to participating in conversations with other roles.

Apart from the core abstractions, the model introduced three additional concepts. The concept of *role state* and *organization dynamics* allow to better describe the adaptability of individual roles and collaborations. The concept of *capability* allows to decouple actors, roles, behaviors, and conversations to improve reuse and modularity of collaborations.

Chapter 4

Macodo Architectural Views

Chapter 3 introduced a conceptual model for dynamic collaborations and defined a set of collaboration abstractions independent of design and implementation concerns. This chapter presents a set of architectural views that allow to use these abstractions to design, document, and reason about collaborations in terms of software elements. Each architectural view introduces a set of architectural modeling abstractions that reify the collaboration abstractions at architectural level. The next chapter (Chapter 5) presents a middleware infrastructure that provides a platform to implement service collaborations, designed in the architectural views, using standard Web service technology.

4.1 Introduction

Software architecture plays an important role in the development process of software systems [17]. It provides a way to deal with complexity and it serves as a vehicle of communication with stakeholders for mutual understanding and negotiation. Software architecture manifests the earliest set of design decisions and provides the main structures to realize both the required functionalities and quality attributes.

The documentation of software architecture is a key element in the successful development of complex software systems. This documentation can range from informal sketches to formal notations, and is typically structured as a set of architectural *views* [139, 4, 84]. A view is a representation of a set of system elements and the relationships associated with them. Each view takes a specific perspective on the system with respect to a particular set of concerns. It only shows those parts of the system that are relevant to addressing these concerns. This allows to manage complexity and provides a way to communicate specific architectural decisions to relevant stakeholders.

This chapter introduces a set of architectural views that allow to design and document collaborations in terms of software elements. Each architectural view introduces a set of architectural modeling abstractions that reify the collaboration abstractions of the Macodo model at architectural level. The motivation, or architectural drivers, for these views are derived from the problem statements of this thesis:

- *Functionality.* One of the primary drivers is the ability to express and reason about collaboration functionality.
- *Managing Complexity and Separation of Concerns.* When designing collaborations, several concerns need to be documented. Each view should provide a different perspective on collaborations and focus on specific concerns.
- *Understandability and Expressibility.* By providing modeling abstractions for collaborations, views can allow architects and other stakeholders to express and reason about collaboration qualities and concerns in terms of collaboration concepts (e.g., representing availability in terms of roles or relating performance to specific actors).
- *Reuse.* To promote modular design, views should provide mechanisms to decompose complex collaborations into reusable building blocks.
- *Modifiability and Runtime Adaptation.* Collaborations have to be easy to modify, both at design time and runtime. Collaborations can be created and destroyed and participants can join and leave. Views should provide ways to promote modifiability and document runtime adaptation.
- *Conceptual Integrity.* Conceptual integrity is about achieving a unified theme or vision in the design of a system at all levels [17], and has been considered one of the most important considerations in the design of a system [22]. Views should allow to document collaborations in a uniform way at different levels of abstraction.

Based on these drivers we present three architectural views for Macodo:

- **Organization Module View (Sect. 4.2).** The Organization Module View is a Module View that describes the collaborations in a system in terms of implementation units. It promotes modularity, reuse, and modifiability, by decomposing collaborations into reusable organization modules, and by expressing commonalities and variations among these modules.
- **Organization & Actor View (Sect. 4.3).** The Organization & Actor View is a Component & Connector (C&C) View that identifies and describes the main actors in a system and the organizations between them. Actors are represented as components and organizations as connectors encapsulating complex collaborations. The Organization & Actor View allows to express

and reason about runtime qualities of collaborations at the highest level of abstraction.

- **Role & Conversation View (Sect. 4.4).** The Role & Conversation View is a C&C View that identifies the main roles inside an organization, their behaviors, and the conversations between the roles. Roles are represented as components, conversations as connectors encapsulating reusable interactions, and behaviors as sub-components of role components. The Role & Conversation View can be used to refine the internal architecture of an organization connector. It provides abstractions to model behaviors and interactions as separate concerns, and allows to describe runtime qualities and adaptation of collaborations in detail.

To design and document the architecture of a complete system, developers will need to combine the Macodo views with other existing architectural views. The deployment of collaborations, for example, can be described using normal allocation views [84].

To define each type of Macodo view, this chapter follows the style guide proposed by Bass et al. [84]. A style guide provides a manual on how to use a type of view. It starts by introducing and describing the possible elements and relations that can occur in a view. Elements are the architectural building blocks that can be used in the view. Relations are the possible associations between elements that allow them to work together to realize actual functionality and quality of a system. Next, the style guide defines a set of constraints and rules that apply to the elements and relations of a view in order to create a valid instance of a view. Finally, the guide describes what the view is for, what type of notation can be used, and how it relates to other types of views. Each view is also illustrated with concrete examples from the supply chain management case. An extended example of a view documentation can be found in Appendix A.

Overview. Section 4.2, 4.3, and 4.4 describe the style guide for each Macodo view. Section 4.5 discusses how to document an actual instance of a Macodo view.

4.2 Organization Module View

The Organization Module View is a Module View that defines the principle implementation units of the collaborations in a system and the relations between them. Types of organizations, roles, conversations, and behaviors are represented as modules (Fig. 4.1). Capabilities of roles, conversations, and behaviors are mapped to interfaces that are required and provided by these modules.

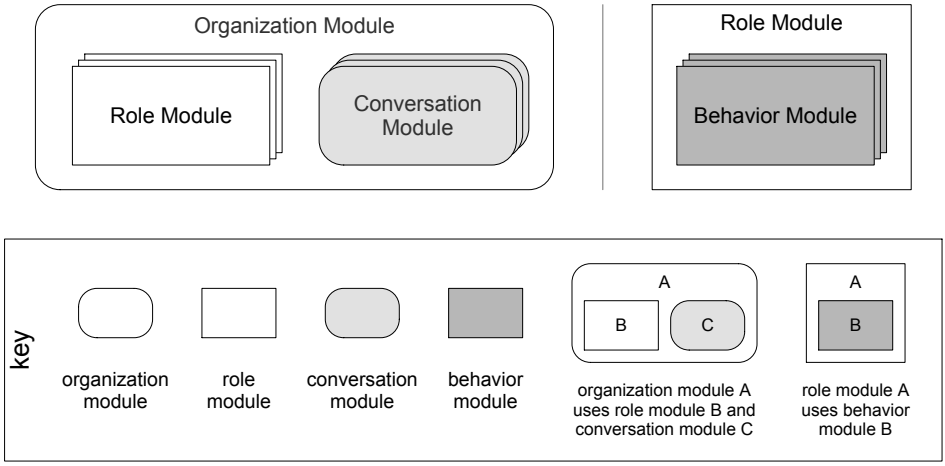


Figure 4.1: Types of organizations, roles, conversations, and behaviors can be mapped to modules. *Left:* An organization module uses a set of role modules and conversation modules. *Right:* A role module uses a set of behavior modules.

4.2.1 Elements, Relations and Their Properties

Elements

Organization Module. An organization module represents the implementation unit of a specific collaboration type. It defines a set of possible roles and conversations, and depends on a set of role modules and conversation modules (Fig. 4.1, left). These dependencies are represented by the 'uses'-relation, which is explained below.

Role Module. A role module models the implementation unit of specific role type. It specifies a coherent set of responsibilities within an organization and depends on a set of behavior modules, which represent the behaviors of the role (Fig. 4.1, right).

Conversation Module. A conversation module represents the implementation unit of a specific conversation type. It defines a reusable type of interaction, which can be used by one or more organization modules.

Behavior Module. A behavior module models the implementation unit of a specific behavior type. It specifies a reusable unit of role functionality that can be used by roles modules.

Interfaces of Elements

At architectural level, the concept of capability can be used to group the interfaces of the different modules. Each conversation and behavior capability defines a pair of interfaces. A role capability becomes a set of these interfaces (Fig. 4.2). The documentation of such interfaces can include a business protocol that defines how to use the interfaces.

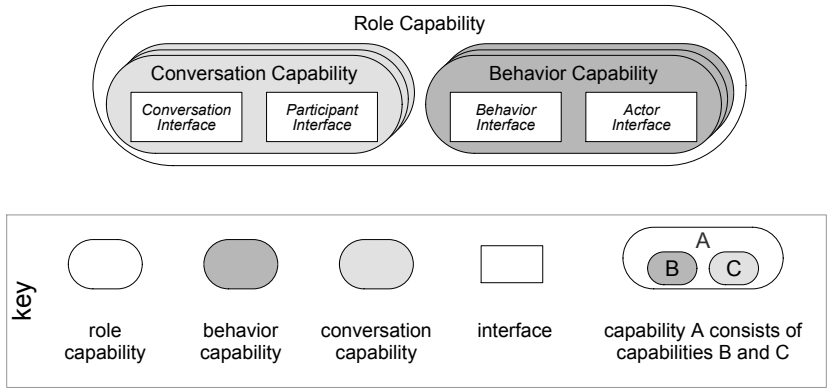


Figure 4.2: Capabilities can be used to group the interfaces of different modules.

A conversation capability defines a conversation interface and a participant interface. Requiring a conversation capability translates to using the participant interface and realizing the conversation interface. Providing a conversation capability means to realize the participant interface and to use the conversation interface. Similarly, a behavior capability defines a behavior interface and an actor interface. Using this concept of capability we can define the interfaces of each module as a set of required and provided capabilities. Organization modules, however, have no explicit interfaces.

The interfaces of a role module can be modeled as a required role capability and a set of provided conversation capabilities (Fig. 4.3, left). This means, the role module realizes a set of conversation and behavior interfaces, which can be used

by the actor of the role, and uses a set of participant and actor interface, which can be realized by the actor of the role (Fig. 4.3, right). To play conversation roles, the role module also realizes a set of participant interfaces, and uses a set of conversation interfaces.

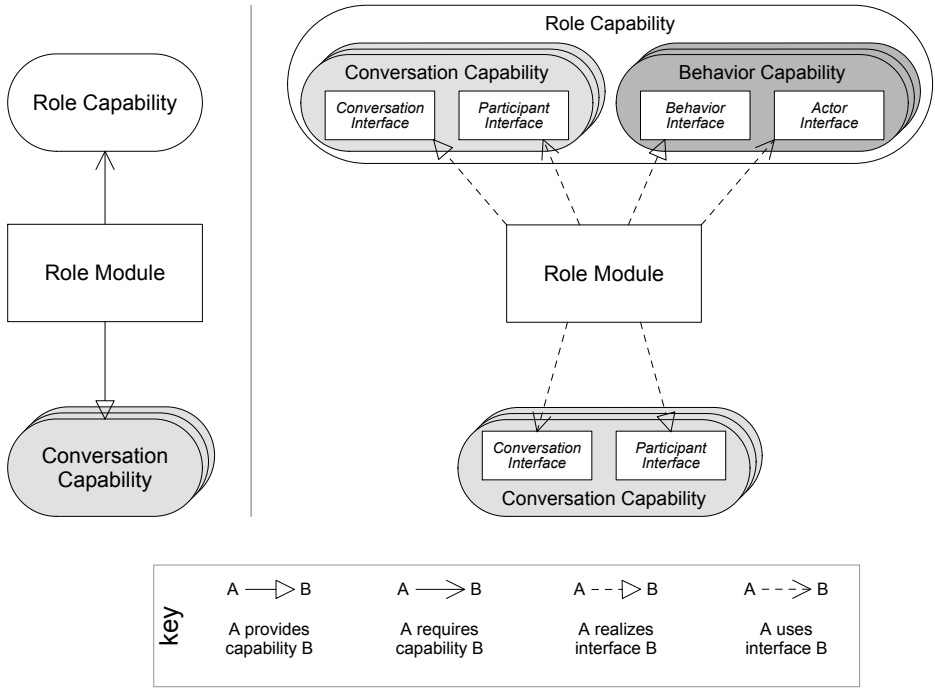


Figure 4.3: The interfaces of a role module can be modeled as a required role capability and a set of provided conversation capabilities.

The interfaces of a conversation module can be defined as a set of required conversation capabilities (Fig. 4.4). These capabilities correspond to the conversation roles of the conversation.

The interfaces of a behavior module can be modeled as a required behavior capability, and a set of provided conversation capabilities (Fig. 4.5). The provided conversation capabilities allow the behavior to play one or more conversation roles.

Relations

Uses. The uses relation is a form of the depends-on relation [17]. Modules can use other modules. Module A uses module B, if A depends on the presence of a correctly functioning B to satisfy its own requirements.

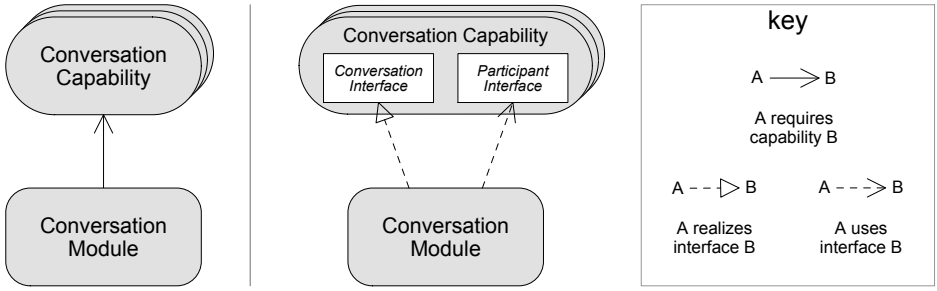


Figure 4.4: The interfaces of a conversation module can be modeled as a set of required conversation capabilities.

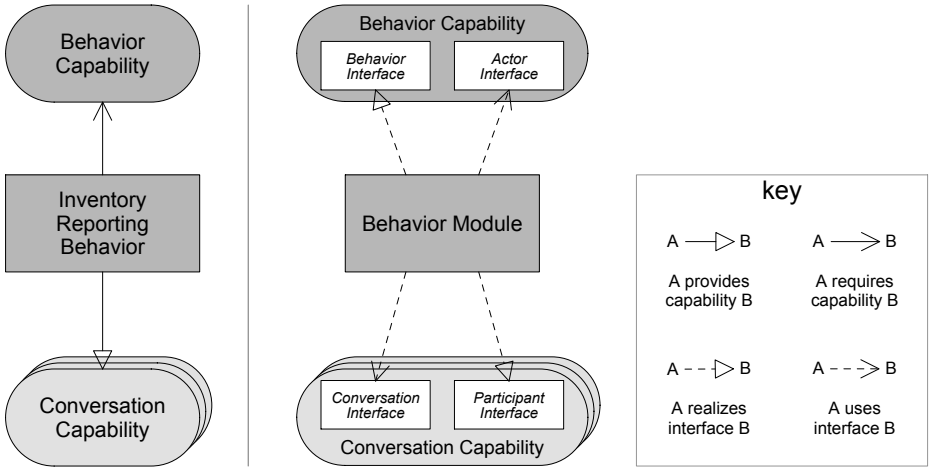


Figure 4.5: The interfaces of a behavior module can be modeled as a required behavior capability and a set of provided conversation capabilities.

Uses Interface. The uses interface relation is a form of the uses relation for interfaces. Modules can use specific interfaces. Module A uses interface C, if A depends on the presence of a correctly functioning realization of interface C to satisfy its own requirements.

Interface Realization. The interface realization relation is a form of the generalization relation. Module A realizes interface B, if it correctly implements interface B.

Capability Provision. The capability provision relation is a combination of the uses interface relation and the interface realization relation. Module A provides behavior capability B, if A realizes the actor interface and uses the behavior interface of capability B. Module A provides conversation capability C, if A realizes the participant interface and uses the conversation interface of C.

Capability Requirement. The capability requirement relation is also a combination of the uses interface relation and the interface realization relation. Module A requires behavior capability B, if A uses the actor interfaces and realizes the behavior interface of capability B. Module A requires conversation capability C, if A uses the participant interface and realizes the conversation interface of C.

4.2.2 Constraints

The following constraints apply to the elements and relations of the Organization Module View:

- An organization module can use a set of role modules and conversation modules. It cannot be involved in any interface realization, capability provision, or capability requirement relation.
- A role modules can use a set of behavior modules. It always requires one role capability, consisting of a set of behavior and conversation capabilities, and it can provide a set of conversation capabilities.
- A conversation module can require a set of conversation capabilities.
- A behavior module always requires one behavior capability and can provide a set of conversation capabilities.

4.2.3 What the Organization Module View Is For

The Organization Module View shows how the collaborations in a system are structured in terms implementation units. Architects can use the Organization

Module View to identify the responsibilities of different modules and to document and organize the interfaces of these modules in terms of provided and required capabilities. Such information can be used by developer of organizations and developers of actors, but also by managers, testers, and maintainers who want to understand the system.

In addition, the Organization Module View allows to express and reason about particular collaboration qualities:

- **Modularity.** Architects can decompose complex collaborations in a set of organization modules, role modules, conversation modules, and behavior modules, each with their specific responsibilities. This allows architects, developers, and managers to better grasp the system and can reduce design and implementation faults.
- **Reusability.** Organization modules, role modules, conversation modules and behavior modules can be defined as reusable building blocks. To further promote reuse, architects can express commonalities and variations among these modules. This information is useful to designers and implementers. Reuse of existing modules and code can increase productivity.

4.2.4 Notation

The Organization Module View can be described using a graphical notation (Fig. 4.6). The graphical notation is mainly used for the primary presentation. It gives a visual representation of the different modules inside a system.

4.2.5 Relation to Other Views

Organization modules can be mapped to organization connectors in the Organization & Actor View (Sect. 4.3). Role modules, conversation modules, and behavior modules can be mapped to role components, conversation connectors and behavior components in the Role & Conversation View (Sect. 4.4).

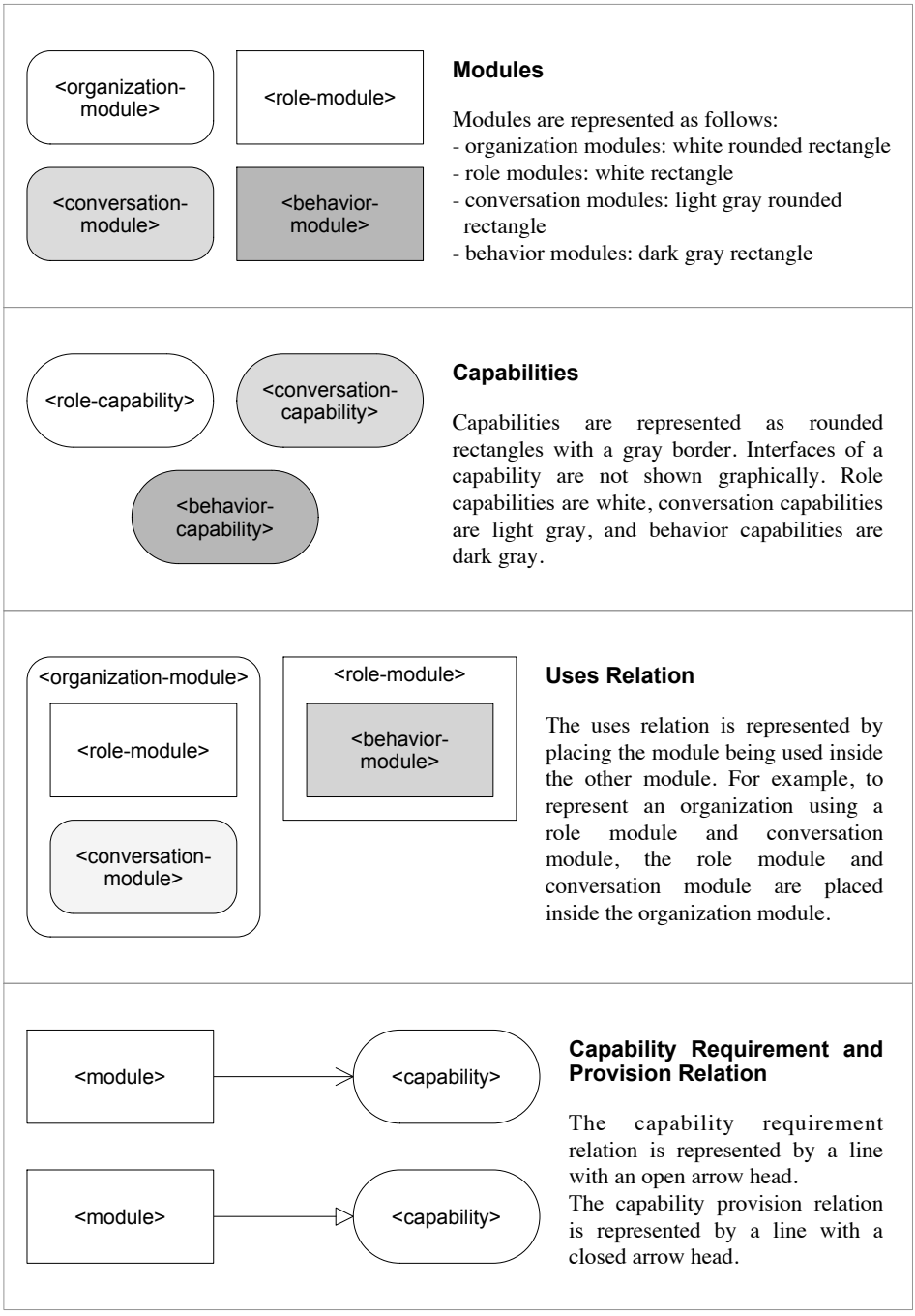


Figure 4.6: The elements of the graphical notation for the Organization Module View.

4.2.6 Examples

The supply chain case introduced two types of supply chain collaborations: a VMI-based supply chain (Fig. 2.8, p. 39) and a CMI-based supply chain (Fig. 2.9, p. 40). Both collaboration types can be modeled as an organization module: the *Vmi Organization* and *Cmi Organization* (Fig. 4.7). Each organization module uses a set of role modules and conversation modules. The role modules *Warehouse Role* and *Transporter Role*, and the conversation modules *Forecasting Conversation*, *Inventory Reporting Conversation*, *Call-Off Conversation*, and *Transport Conversation* are used by both organization modules.

Conversation module *Inventory Reporting Conversation* requires two conversation capabilities, *Inventory Capability* and *Client Capability*, one for each of its conversation roles (Fig. 4.8).

The *Warehouse Role* module requires the *Warehouse Role Capability* and provides a set of *Conversation Capabilities* such as *Inventory Capability* (Fig. 4.9). This allows the role to realize the *Inventory* role of the *Inventory Reporting Conversation*. The *Warehouse Role* module also uses a set of behavior modules, such as the *Inventory Reporting Behavior* and the *Call-Off Fulfillment Behavior*.

The *Inventory Reporting Behavior* module requires the *Inventory Reporting Behavior Capability* and provides the *Inventory Capability*, allowing the behavior to realize conversation roles that require this capability (Fig. 4.10).

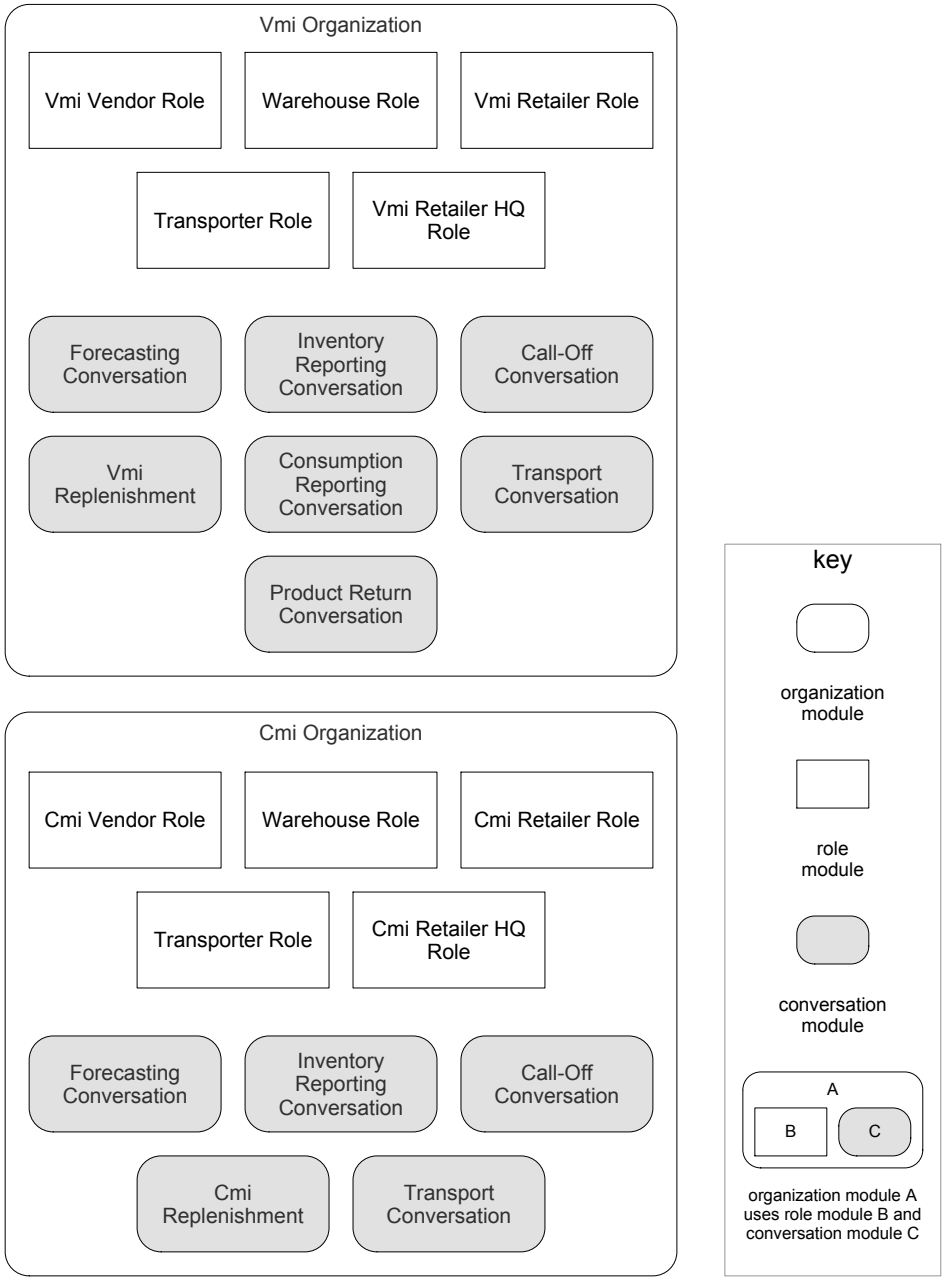


Figure 4.7: Two organization modules supporting two types of supply chain collaborations.

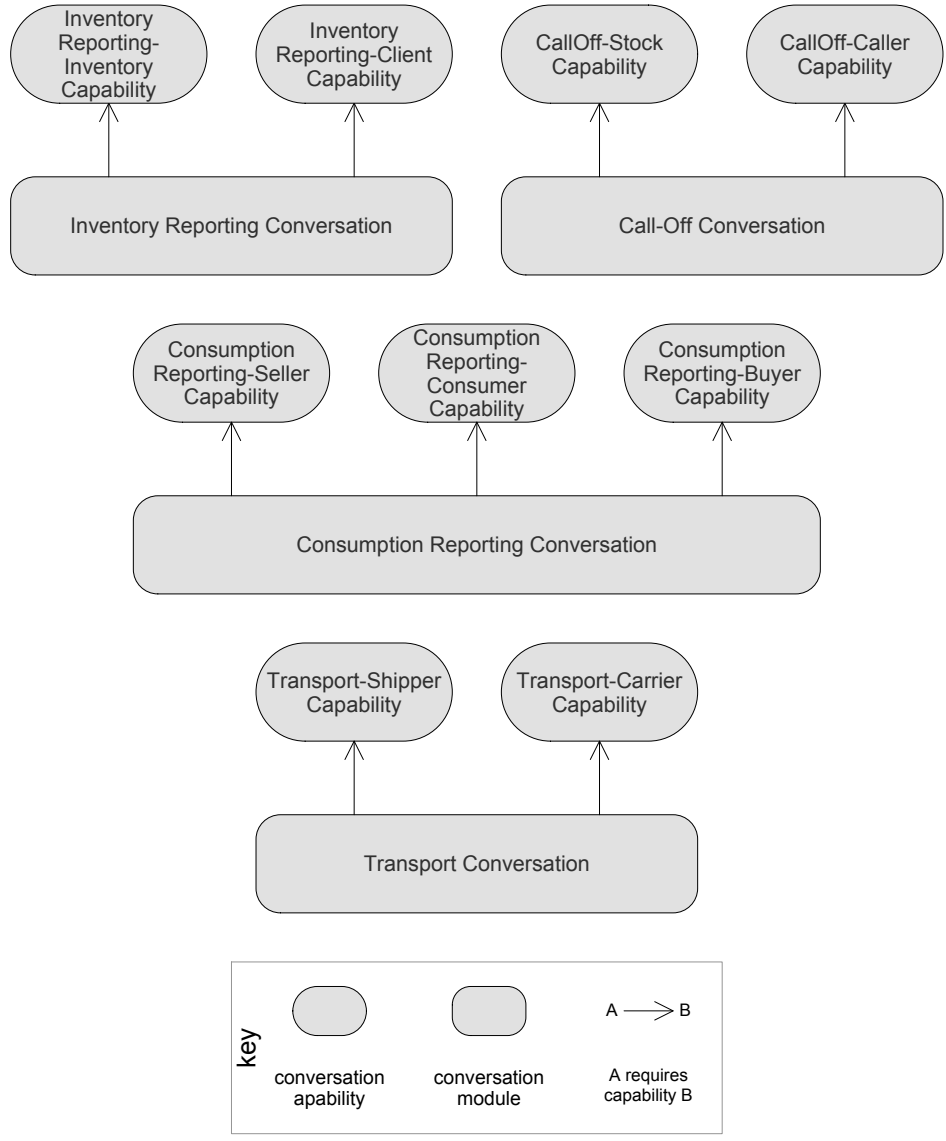


Figure 4.8: A subset of the conversation modules and their required conversation capabilities.

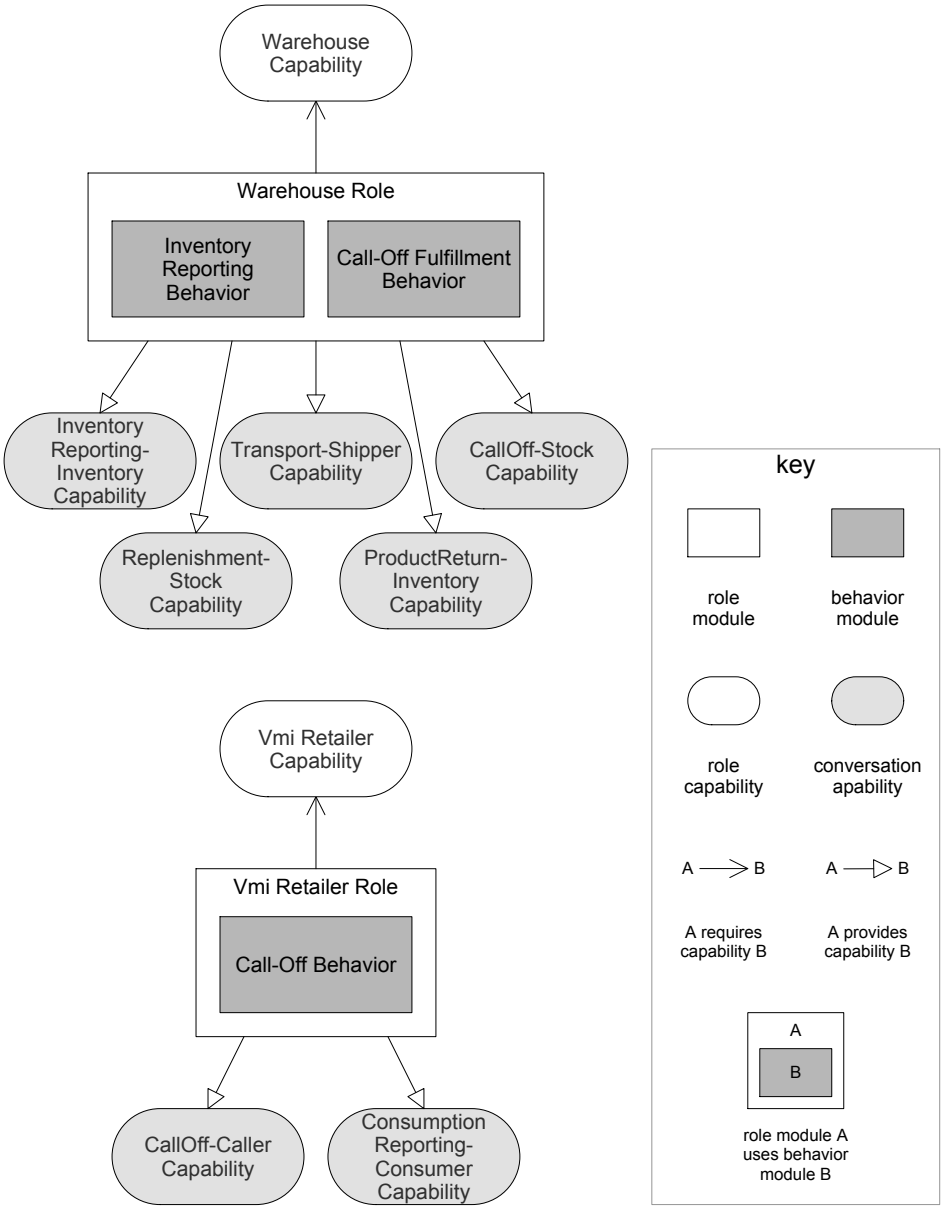


Figure 4.9: A subset of the role modules. Each module only shows a subset of the behavior modules and provided conversation capabilities.

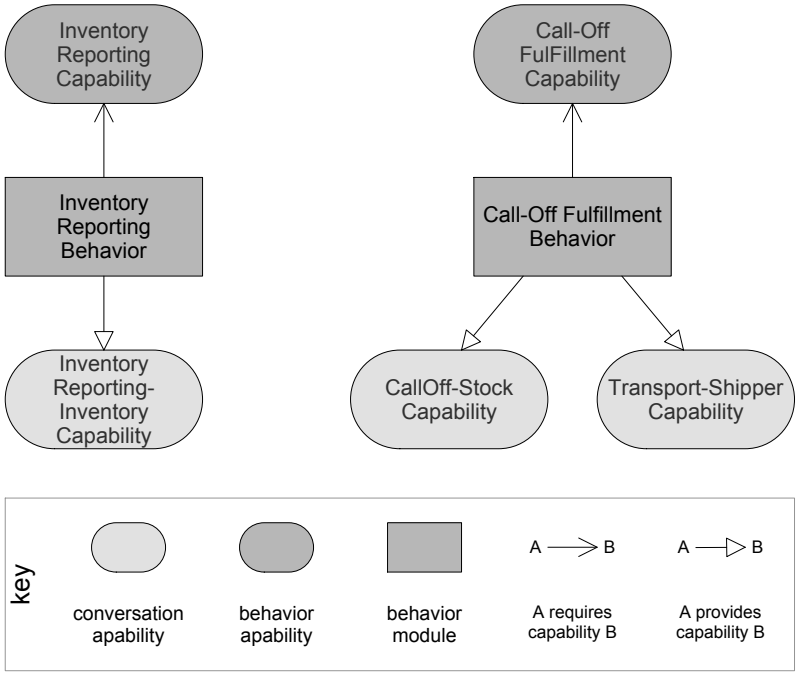


Figure 4.10: A subset of the behavior modules used by the different role modules.

4.3 Organization & Actor View

The Organization & Actor View is a Component & Connector View that identifies the main actors in a running system and the organizations between them. Actors are represented as components and organizations as connectors that encapsulate complex collaborations (Fig. 4.11).

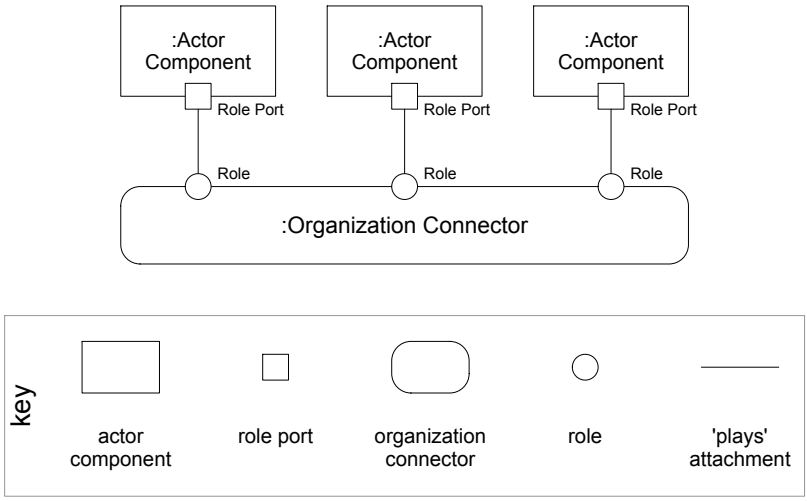


Figure 4.11: Organization instances are mapped to connectors encapsulating complex collaborations. Role instances are mapped to connector roles. Actors are mapped to components that can be attached to these roles.

4.3.1 Element Types, Relation Types, and Properties

Elements

Actor Components. Actor components represent the actors in the system that are able to participate in collaborations. Actor components have a set of role ports (Fig. 4.11). These roles ports define points of potential interaction with other actor components through an organization connector. Actor components can have multiple role ports of the same type (e.g., when it can play the same role multiple times).

Organization Connectors. An organization connector represents a specific organization instance in the system. It encapsulates a complex collaboration between a set of actor components. The roles of an organization are mapped to

connector roles (Fig. 4.11). These roles specify how the organization connector can be used by actor components to participate in the encapsulated collaboration. An organization connector can have multiple roles of the same type (e.g., when a role can be played multiple times in parallel).

Interfaces of Elements

Roles define the runtime interfaces of organization connectors. These interfaces correspond to a required role capability. Role ports define the runtime interfaces of actor components. The interfaces defined by a role port correspond to a provided role capability. All interfaces of elements should be properly documented in the element catalog (see Sect. 4.5).

Reusable Element Types

To reuse organization connectors, an explicit organization connector type can be defined. An organization connector type defines a reusable type of organization connector, with its possible roles and their multiplicity, and can be mapped to a specific organization module. A running system can have multiple organization connector instances of the same type.

Relations

Attachment. An attachment represents the 'plays' relation between an actor and a role. It connects a role port of an actor component to a role of an organization connector (Fig. 4.11). An attachment relation indicates the participation of an actor component in the collaboration encapsulated by an organization connector.

4.3.2 Constraints

The following constraints apply to the elements and relations of the Organization & Actor View:

- Actor components can only be attached to organization connectors, not to other actor components.
- Organization connectors can only be attached to actor components, not to other organization connectors.
- An attachment can only be made between a compatible role port and organization role.

- A role port is compatible with an organization role, if the role port provides the role capability required by the organization role.

4.3.3 Documenting Dynamics and Runtime Adaptation

At architectural level, the organization dynamics defined by the Macodo model (see Sect. 3.3.2) can be mapped to architectural variation points that are exercised at runtime. A variation point is a place in an architecture where a specific type of flexibility or variability has been built in [17]. Variability is the ability to quickly achieve change in a preplanned way through some sort of variation mechanism. When variation is performed at runtime, the architecture is called a dynamic architecture.

Variability is documented in the variability guide of the architectural views (see Sect. 4.5) in which the variability applies. The Organization & Actor View can be used to document two types of organization dynamics: creating and destroying organizations, and creating and destroying roles.

Creating and Destroying Organizations

Mapping to Architectural Variability. Organizations are created by instantiating an organization connector of a specific organization connector type. Organizations are destroyed by destroying the corresponding organization connector.

Documenting the Variability. The documentation should include a list of possible organization connector types in the system. Each type should be properly described in the Organization & Actor View.

Creating and Destroying Roles.

Mapping to Architectural Variability. A role is created by instantiating a connector role on an organization connector and attaching it to the role port of the corresponding actor component. A role is destroyed by destroying the corresponding connector role and removing the attachment.

Documenting the Variability. The documentation should include a list of possible connector roles for each organization connector type, and the minimum and maximum number of concurrent instances. To document this variability, the graphical notation for an organization connector type can be used (see Sect. 4.3.5, p. 89), or a simple textual table. This table has the following form:

<organization-connector-type>
- *<role-component-type>* [<min-occur>..<max-occur>]

Each organization connector type has a set of role types and their minimum and maximum occurrences.

4.3.4 What the Organization & Actor View Is For

The Organization & Actor View identifies and describes the main actors in a system and the organizations between them. Architects can use the Organization & Actor View to describe running systems in terms of concrete collaborations, and to assign roles to specific actors, while making abstraction of organization details. This enables managers, analysts, and users of organizations to reason about the collaborations in the running system and the assignment of responsibilities.

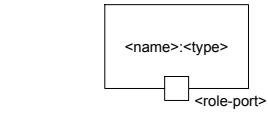
In addition, the Organization & Actor View allows to express and reason about a set of runtime qualities at a high level of abstraction:

- **Collaboration qualities.** Architects can express runtime qualities of collaborations in terms of roles and actor components. Examples are expressing availability in terms of roles or relating performance to specific actor components. This allows developers of actor components, managers, and analysts to reason about these qualities using collaboration concepts.
- **Reuse.** Architects can describe how organization connector types are reused throughout a running system. This information is useful for managers, analysts, and testers to assess the scope and impact of runtime elements.
- **Runtime Adaptation.** Architects can use the Organization & Actor View to express organization dynamics or runtime adaptation at a high level of abstraction, in terms of creating and destroying organization connectors and attaching roles to actor components. Information on runtime adaptation is useful for developers, managers, analysts, and users who want to understand the working of the system.

4.3.5 Notation

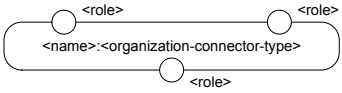
The Organization & Actor View can be described using a graphical notation (Fig. 4.12). The graphical notation is mainly used for the primary presentation. It gives a visual overview of the actor components in the system and the organization connectors between them.

A colon (':') in the name of an element indicates that the element is an instance of a certain component or connector type. The part before the colon represents the name of the instance. The part after the colon represents the type.



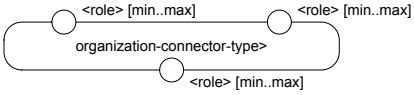
Actor Component

An actor component is represented as a rectangle. Role ports are shown as little squares (with a name) on the border of the actor component.



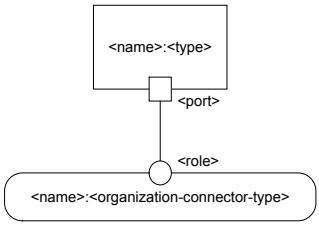
Organization Connector

An organization connector is represented as a rounded rectangle. Roles are shown as little circles (with a name) on the border of the organization connector.



Organization Connector Type

The notation can also be used to represent organization connector types. A type has no instance name, and for each role there can be a range indicator to define the minimum and maximum number of occurences.



'Plays' Attachment

The attachment of an organization role to a role port is represented by a line that connects the organization role with the role port.

Figure 4.12: The elements of the graphical notation for the Organization & Actor View.

4.3.6 Relation to Other Views

The Organization Module View (Sect. 4.2) shows the implementation units for an organization connector. The Role & Conversation View (Sect. 4.4) can be used to refine the internal architecture of an organization connector. Every role of an organization connector is refined as a role component.

4.3.7 Examples

The supply chain case described a set of concrete collaborations (Fig. 2.7, p. 38). To support these collaborations, we can structure the architecture of our system using actor components and organization connectors (Fig. 4.13). The different supply chain partners, participating in the collaborations, are represented as actor components. The collaborations between the supply chain partners are realized by a set of organizations connectors. The actor component of each participating supply chain partner is attached to a corresponding organization role.

Since there can be multiple organization connectors of the same type, we define a set of organization connector types (Fig. 4.14). Each organization connector type encapsulates a specific type of collaboration. Multiple organization connectors can be instantiated of the same type.

The organization connector types can be easily mapped to the organization modules we defined before (see Fig. 4.7, p. 82). This type of information should be documented in the mapping between views:

Element in Organization & Actor View	**Element in Organization Module View**
Vmi Organization	*Vmi Organization Module*
Cmi Organization	*Cmi Organization Module*

Runtime Adaptation. The 4PL in our supply chain case has the ability to dynamically create and destroy organizations and roles. This type of dynamics is documented in the variability guide of a view. It contains a list of possible organization connector types in the system, and defines the possible roles for each type of organization connector. So far there are two possible organization connector types:

- *Vmi Organization*
- *Cmi Organization*

Each organization connector type has a set of possible roles:

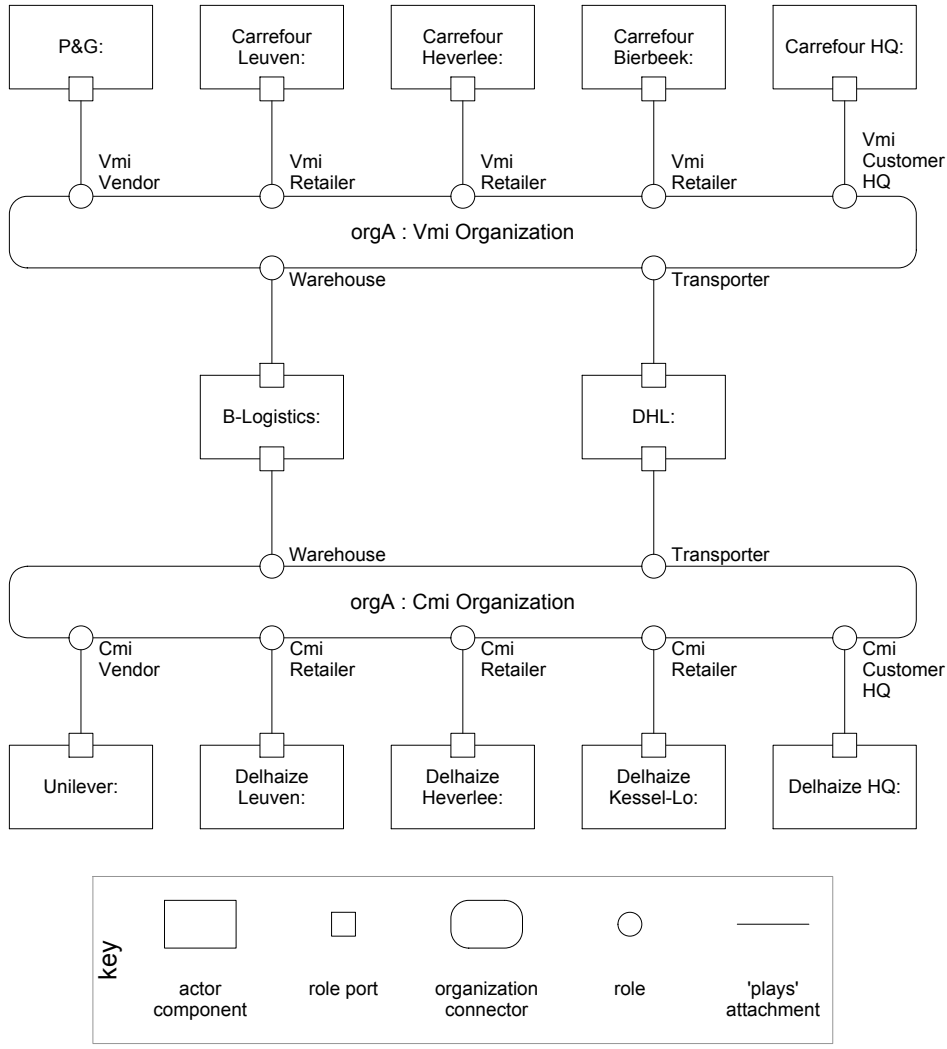


Figure 4.13: An example of an Organization & Actor View showing two organization connectors attached to a set of actor components.

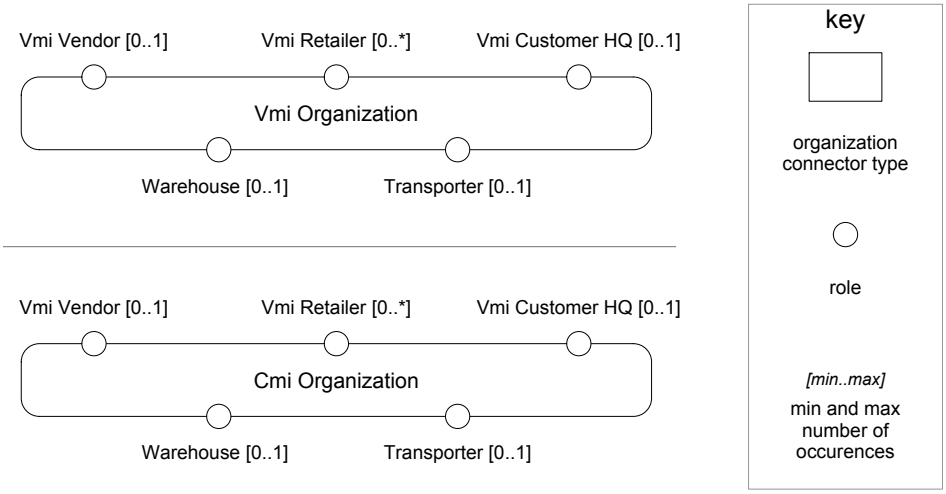


Figure 4.14: An example of organization connector types for two types of supply chain collaborations. Each organization connector type defines a set of organization roles and their multiplicity.

Vmi Organization
- *Vmi Vendor* [0..1]
- *Warehouse* [0..1]
- *Vmi Retailer* [0..*]
- *Vmi Retailer HQ* [0..1]
- *Transporter* [0..1]

Cmi Organization
- *Cmi Vendor* [0..1]
- *Warehouse* [0..1]
- *Vmi Retailer* [0..*]
- *Vmi Retailer HQ* [0..1]
- *Vmi Retailer HQ* [0..1]

The same information was already defined using the graphical notation in Fig. 4.14, p. 93.

Using the possible variation points, the 4PL can define an application-specific organization management logic that creates and destroys organization connectors and roles (Fig. 4.15).

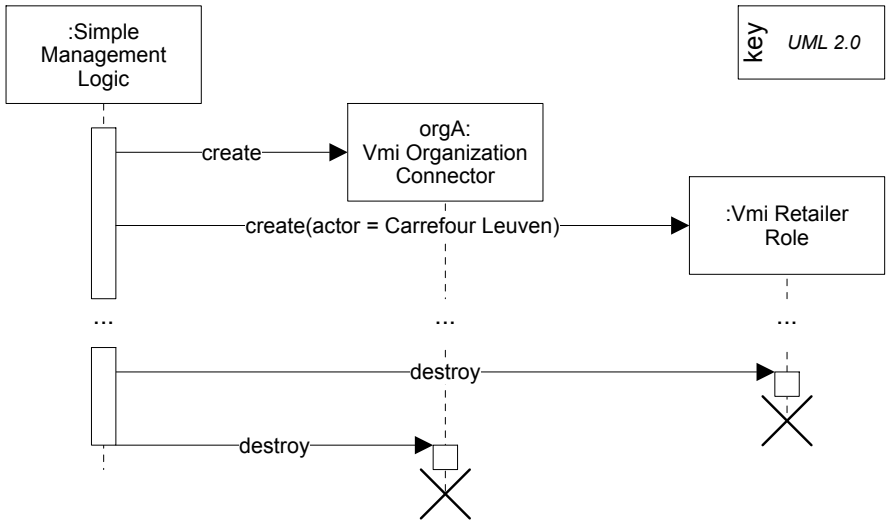


Figure 4.15: A simple management logic creates and destroys an organization connector and role.

4.4 Role & Conversation View

The Role & Conversation View is a Component & Connector View that refines the internal architecture of an organization connector in terms of roles, behaviors, and conversations between roles. Roles are mapped to components, behaviors to sub-components of role components, and conversations to connectors between role components (Fig. 4.16). The notation in Fig. 4.16 is only used to illustrate the mapping of Macodo concepts to architectural abstractions. An alternative and recommended notation for the Role & Conversation View is presented in Sect. 4.4.5.

4.4.1 Elements, Relations, and Their Properties

Elements

Role Components. A role component represents a specific role instance inside an organization connector. It mediates the actions of a particular actor component with those of the rest of the organizations. Role components have two types of ports: an actor port and a set of conversation ports (Fig. 4.16). The actor port allows the role component to interact with the actor component that 'plays' the role. Conversation ports allow the role component to interact with other role components through conversation connectors. Role components always have one actor port and can have multiple conversation ports of the same type, allowing a role component to have parallel conversations of the same type.

Conversation Connectors. A conversation connector represents a specific conversation in an organization connector. It encapsulates an interaction between a set of role components. The conversation roles of a conversation are mapped to connector roles (Fig. 4.16). These connector roles specify how the conversation connector can be used by role components to participate in the encapsulated interaction. An conversation connector can have multiple conversation roles of the same type (e.g., when a conversation role can be played multiple times in parallel).

Behavior Components. Behavior components represent a behavior that is executed by a role component. It encapsulates the execution of a specific role functionality. A behavior component has two types of ports: an actor port and a set of conversation ports (Fig. 4.16). The actor port allows the behavior component to interact with the actor component of the encapsulating role component. Conversation ports allow a behavior component to realize some of the conversation ports of the encapsulating role component. Actor components always have one actor port and can have multiple conversation ports of the same type.

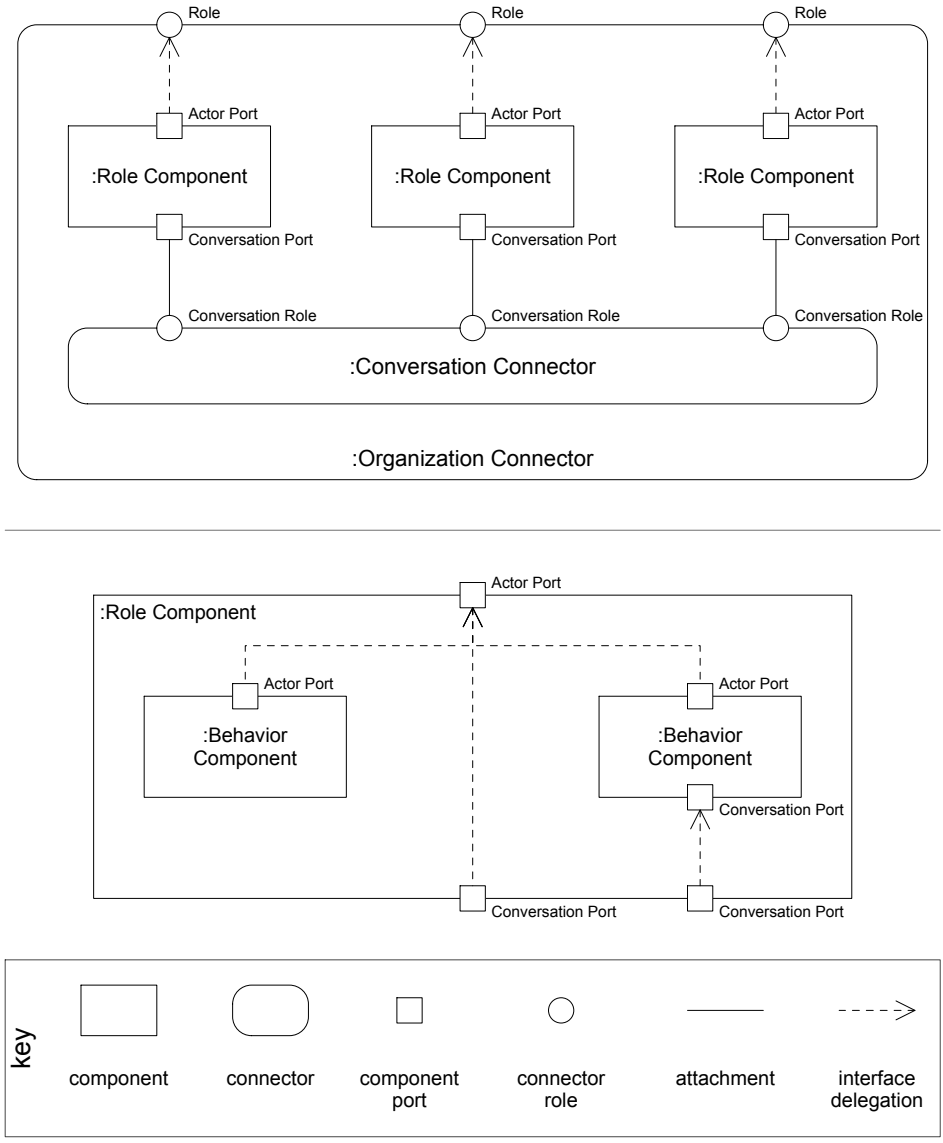


Figure 4.16: Roles are mapped to components, conversations to connectors that encapsulate reusable interactions, and behaviors to sub-components of role components. The notation in this diagram is only used to illustrate the mapping of Macodo concepts.

Interfaces of Elements

Conversation roles define the runtime interfaces of conversation connectors. These interfaces correspond to a required conversation capability. Actor ports and conversation ports define the runtime interfaces of role components and behavior components. The interfaces defined by an actor port of a role component correspond to a required role capability. The interfaces defined by an actor port of a behavior component correspond to a required behavior capability. Interfaces of conversation ports correspond to a provided conversation capability. All interfaces of elements should be properly documented in the element catalog (see Sect. 4.5).

Properties of Elements

Role States. Role components are always in a specific role state. This state is documented as an additional property of a role component. The state of a role component constraints the number and type of conversation ports and behavior components. Role states are documented in the variability guide (see Sect. 4.4.3).

Initiating and Participating Conversation Roles. Conversation roles have a property indicating whether they are an initiator or participant of the interaction encapsulated by the conversation connector.

Behavior Component Creation. Behavior components have a property that defines how they are created. There are three options: by the actor of the role, through a conversation, or by the role itself. This is also documented in the variability guide.

Reusable Element Types

To reuse role components, conversation connectors, and behavior components, explicit component and connector types can be defined. Such types define the number and type of component and connector ports, and can be mapped to specific role modules, conversation modules, and behavior modules. A running system can have multiple component and connector instances of each type.

Relations

Attachments. An attachment represents the 'plays' relation between an organization role and a conversation role. It connects a conversation port of a role component to a conversation role of a conversation connector. An attachment relation

indicates the participation of a role component in the interaction encapsulated by a conversation connector.

Interface Delegations An interface delegation delegates the interfaces of one port to the interfaces of an other port. This allows to refine the 'internal' architecture of a component. In the Role & Conversation View, interface delegation is used for delegating conversation ports of role components and actor ports of behavior components.

- **Delegating Conversation Ports of Role Components.** From an external view, the conversation port of a role component provides a conversation capability. Internally, however, we need to define how to realize these capabilities. This can be done in two ways. A first option is to delegate the conversation port to the actor port of the role component. This delegates the act of providing the conversation capability to the actor of the role. To be valid a valid delegation, the conversation capability should be part of the role capability. A second option is to delegate the conversation port to the conversation port of a behavior component. This makes the behavior component responsible for providing the actual conversation capability.
- **Delegating Actor Ports of Behavior Components.** The actor port of a behavior component requires a behavior capability. To provide this capability, we use the interface delegation relation to delegate the actor port of the behavior component to the actor port of the encapsulating role component. This means that the actual behavior capability has to be provided by the actor of the role. To be a valid delegation, the behavior capability should be part of the role capability.

4.4.2 Constraints

The following constraints apply to the elements and relations of the Role & Conversation View:

- Role components can only be attached to conversation connectors, not to other role components.
- A role component is always in a specific role state.
- Conversation connectors can only be attached to role components, not to other conversation connectors, and cannot appear in isolation.
- An attachment can only be made between a compatible conversation port and conversation role.
- A conversation port is compatible with a conversation role, if the actor port provides the conversation capability required by the conversation role.

- An interface delegation can only be defined only between two compatible component ports. Ports are compatible if the capabilities of the delegated port is a subset of the capabilities of the target port.
- A conversation port of a role component can only be delegated to the actor port of the role component, or to a conversation port of an encapsulated behavior component.
- The actor port of a behavior component can only be delegated to the actor port of an encapsulating role component.
- Actor ports of role components, and actor ports and conversation ports of behavior components cannot be attached to any connector in this view.

4.4.3 Documenting Dynamics and Runtime Adaptation

Organization dynamics and runtime adaptation can be documented in the variability guide of a view (as discussed in Sect. 4.3.3). The Role & Conversation View can be used to document three types of organization dynamics: evolving role states, initiating and terminating conversations, and initiating and terminating behaviors.

Evolving Role States

Mapping to Architectural Variability. The state of a role is changed by changing the state of the corresponding role component.

Documenting the Variability. The documentation includes the following elements:

- A list of possible role states for each type of role component.
- A specification of each role state in terms of conversation and behavior constraints. For each type of conversation port and behavior component, these constraints define whether new instances can be created and the maximum number of parallel occurrences. For behaviors, the state also indicates how the behavior can be created (i.e., by the actor of the role, through a conversation, or by the role itself).
- Additional constraints on how the state of a role component can change.

Individual role states can be defined using a textual table. The possible role states and the constraints on how this state can change can be documented using a UML state diagram [193].

Initiating and Terminating Conversations

Mapping to Architectural Variability. A conversation is initiated by creating a new conversation connector between a set of role components. This also creates a set of conversation ports on the role components and attachments to the conversation roles of the conversation connector. A conversation is terminated by terminating the corresponding conversation connector and removing possible attachments.

Documenting the Variability. The documentation includes the following elements:

- A list of possible conversations between the different types of role components. A possible conversation is defined by a conversation connector type and a mapping of possible conversation roles to conversation ports of role component types.
- Dependencies on other variation points. The creation of conversation connectors and the attachment to role components depends on the current state of these components. These states define the type of maximum number of parallel conversation ports.

The documentation of this variability can be done using a textual notation or using the graphical notation of the Role & Conversation View (see Sect. 4.4.5, p. 101). In the textual notation, possible conversations are specified by a conversation connector type and a mapping of conversation roles to conversation ports of role component types as follows:

<conversation-connector-type>
- *<conversation-role>* → *<role-component-type>.<conversation-port>*

Each conversation connector type has a set of conversation roles that are mapped (→) to the conversation port of a specific role component type.

Initiating and Terminating Behaviors

Mapping to Architectural Variability. A behavior is initiated by instantiating a new behavior component in a role component. Only behavior components available in the current state of the role component can be instantiated. A behavior is terminated by terminating the corresponding behavior component.

Documenting the Variability. This type of variability is documented in the possible role states of a role component (see Sect. 4.4.3).

4.4.4 What the Role & Conversation View Is For

The Role & Conversation View identifies the main roles inside an organization, their behaviors and responsibilities, and the conversation between these roles. Architects can use the Role & Conversation View to refine the internal architecture of organization connectors. This information is useful for developers of organization connectors, but also for developers of actor components, managers, maintainers, and testers who need to understand how organization connectors work.

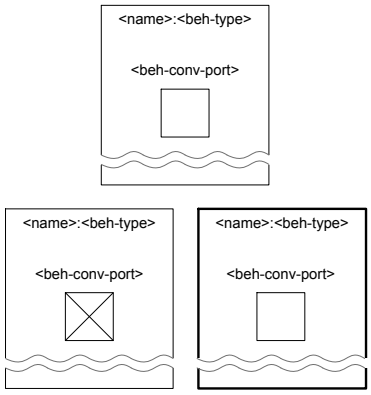
In addition, the Role & Conversation View allows to express and reason about runtime qualities of collaborations in more detail:

- **Collaboration Qualities.** Architects can express runtime qualities of collaborations in detail, using role components, conversation connectors and behavior components. Examples are expressing robustness in terms of behaviors, or describing throughput in terms of conversations. This allows developers of actor components, managers, and analysts to reason about these qualities using detailed collaboration concepts.
- **Reuse.** Architects can describe how role component types, conversation connector types, and behavior component types are reused within organization connectors. This information is useful for managers, analysts, and testers to assess the scope and impact of runtime elements.
- **Runtime Adaptation.** Architects can use the Role & Conversation View to express organization dynamics or runtime adaptation in detail, by showing how the state of individual roles can evolve, and which conversation connectors and behavior components can be instantiated. Information on runtime adaptation is useful for developers, managers, analysts, and users who want to understand the inner workings of an organization connector.

4.4.5 Notation

The notation used to introduce the elements of this view (Fig. 4.16) easily leads to cluttered diagrams. To describe more complex organization connectors, an alternative graphical notation can be used (Fig. 4.17 and Fig. 4.18). This notation is mainly used for the primary representation. It gives a visual representation of how an organization connector is structured as a set of role components, conversation connectors, and behavior connectors.

A colon (':') in the name of an element indicates that the element is an instance of a certain component/connector type. The part before the colon represents the name of the instance. The part after the colon represents the component/connector type.

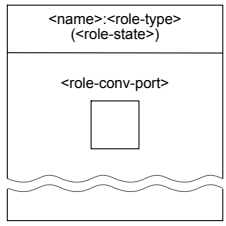


Behavior Component

A behavior component is represented as a white rectangle with a name and a type. Conversation ports are shown as little squares (with a name) inside this rectangle. The actor port is not shown.

Horizontal waves (‘≈’) can be used to indicate that certain parts of the behavior component are omitted.

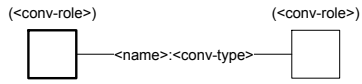
The notation can also be used to indicate how a behavior component is created. A wide border indicates the behavior component is created by the actor of the role. A conversation port with a cross (‘x’) indicates the behavior component is created by an incoming interaction. A plain border and no marked conversation port indicates the behavior is created by the role itself.



Role Component

A role component is represented as a vertical lane with a name, a type, and a state. Conversation ports are shown as little squares (with a name) inside this rectangle. The actor port is not shown.

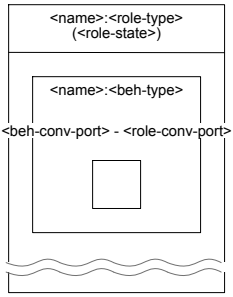
Horizontal waves (‘≈’) can be used to indicate that certain parts of the role component are omitted.



Conversation Connector

A conversation connector is represented as a horizontal line with a name and a type. Conversation roles are shown as squares on this line with a name between round brackets. Initiator roles are indicated by a wide border.

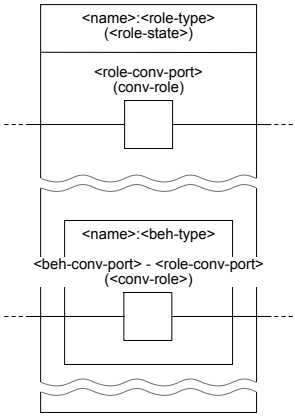
Figure 4.17: The elements of the graphical notation for the Role & Conversation View (part 1).



Interface Delegation

The delegation of an actor port of a behavior component to and actor port of a role component is shown by placing the behavior component inside the role component.

The delegation of a conversation port of a role component to a conversation port of a behavior component is shown by placing the conversation port of the behavior component over the conversation port of the role component. The names of both conversation ports are separated by a hyphen (‘-’).



Attachment

The attachment of a conversation port to a conversation role is represented by placing the conversation role of a conversation connector over the conversation port of a role component.

Figure 4.18: The elements of the graphical notation for the Role & Conversation View (part 2).

Actor ports of role components and actor ports of behavior components are omitted, since their visual representation does not contribute to the content of this view. Despite this simplification, all elements and their ports have to be properly defined in the element catalog of a view.

4.4.6 Relation to Other Views

The Role & Conversation View can be used to define the inner-architecture of an organization connector defined in the Organization & Actor View (Sect. 4.3). Every role component corresponds to an organization role of the organization connector. Creating an organization connector role in the Organization & Actor View (Sect. 4.3) corresponds to the creation of a role component in the Role & Conversation View. Destroying an organization connector role in the Organization & Actor View corresponds to destroying a role component in the Role & Conversation View, which also destroys all encapsulated behavior components. The Organization Module View (Sect. 4.2) shows the implementation units for role components, conversation connectors, and behavior components.

4.4.7 Examples

The organization connectors defined in the Organization & Actor View (see Fig. 4.13, p. 92) can be further refined. The organization connector *orgA:VmiSupplyChain*, for example, can be refined as a set of role components and a set of conversation connectors between them (Fig. 4.19)[1]. Each role component can also be refined with a set of behavior components.

Since there can be multiple role components, conversation connectors, and behavior components of the same type, we can define a set of component and connector types. These types can be easily mapped to the role modules, conversation modules, and behavior modules defined in the Organization Module View (see Fig. 4.8, Fig. 4.9, and Fig. 4.10, p. 85). This type of information is typically documented in the mapping between views:

Element in Role & Conversation View	**Element in Organization Module View**
Vmi Vendor	*Vmi Vendor Role Module*
Warehouse	*Warehouse Role Module*
...	...
Inventory Reporting Behavior	*Inventory Reporting Behavior Module*
...	...
Inventory Reporting Conversation	*Inventory Reporting Conversation Module*
...	...

[1] The example only shows one *Vmi Retailer* role component.

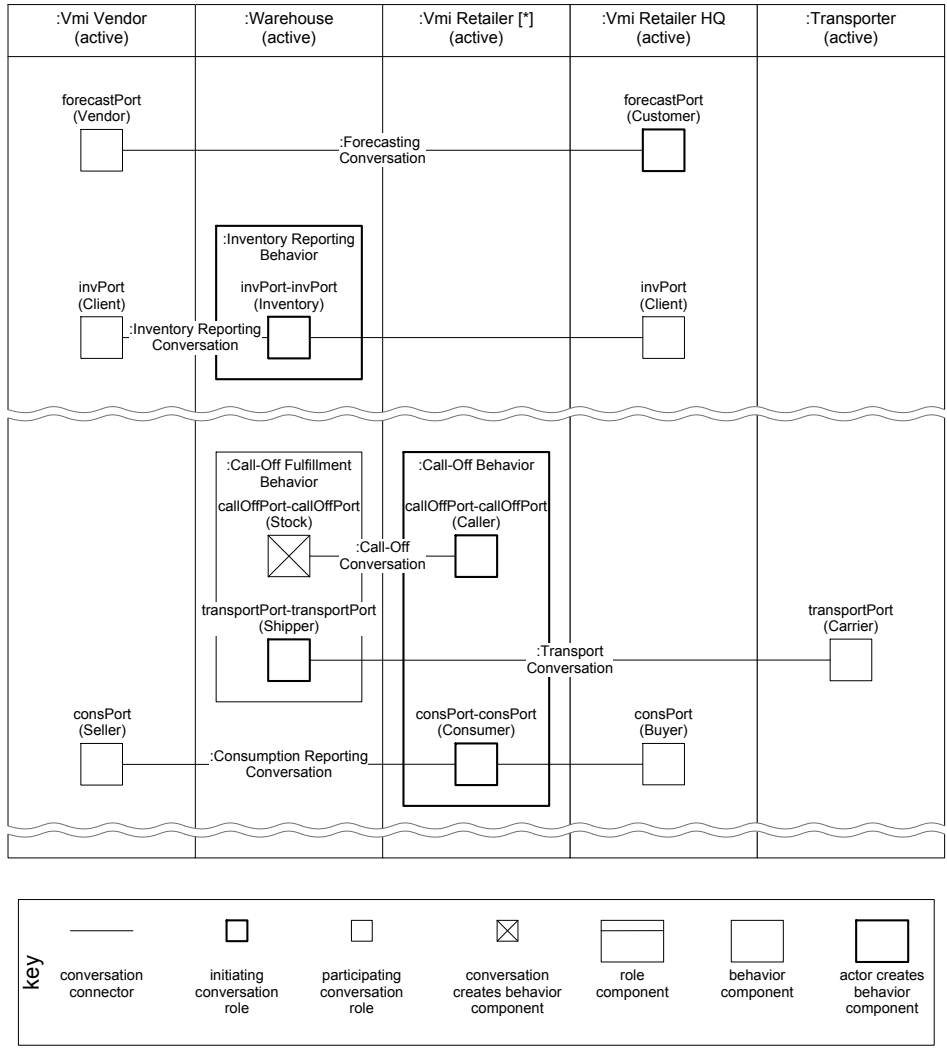


Figure 4.19: An example of a Role & Conversation View showing the refinement of an organization connector (*orgA:VmiSupplyChain*). The organization connector is realized by five role components and a set of conversation connectors between them. (Note: the example only shows one *Vmi Retailer* role component.)

The actual behavior of a conversation connector or behavior component can be documented using BPMN diagrams (Fig. 4.20 and 4.21). In this notation, we use vertical lanes to represent connector roles, and component ports. This allows us to document the behavior of the conversation connector or behavior component in terms of messages send to or received from these connector roles and component ports.

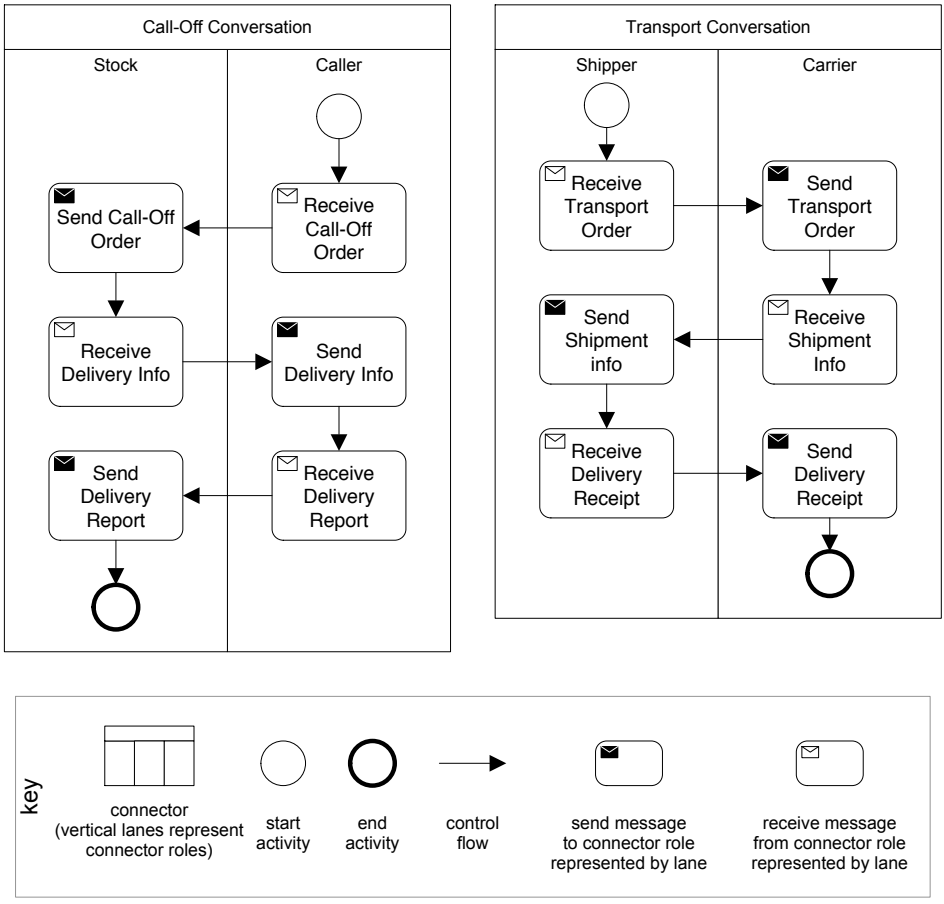


Figure 4.20: The documentation of the behavior of two conversation connectors using BPMN. Vertical lanes represent connector roles.

Runtime Adaptation. The roles of the *VmiSupplyChain* and *CmiSupplyChain* organization connector types have a set of possible states. For the *Vmi Retailer* role component, the possible states are defined in a table (Fig. 4.22, top). How these states can change is documented in a UML state diagram (Fig. 4.22, bottom).

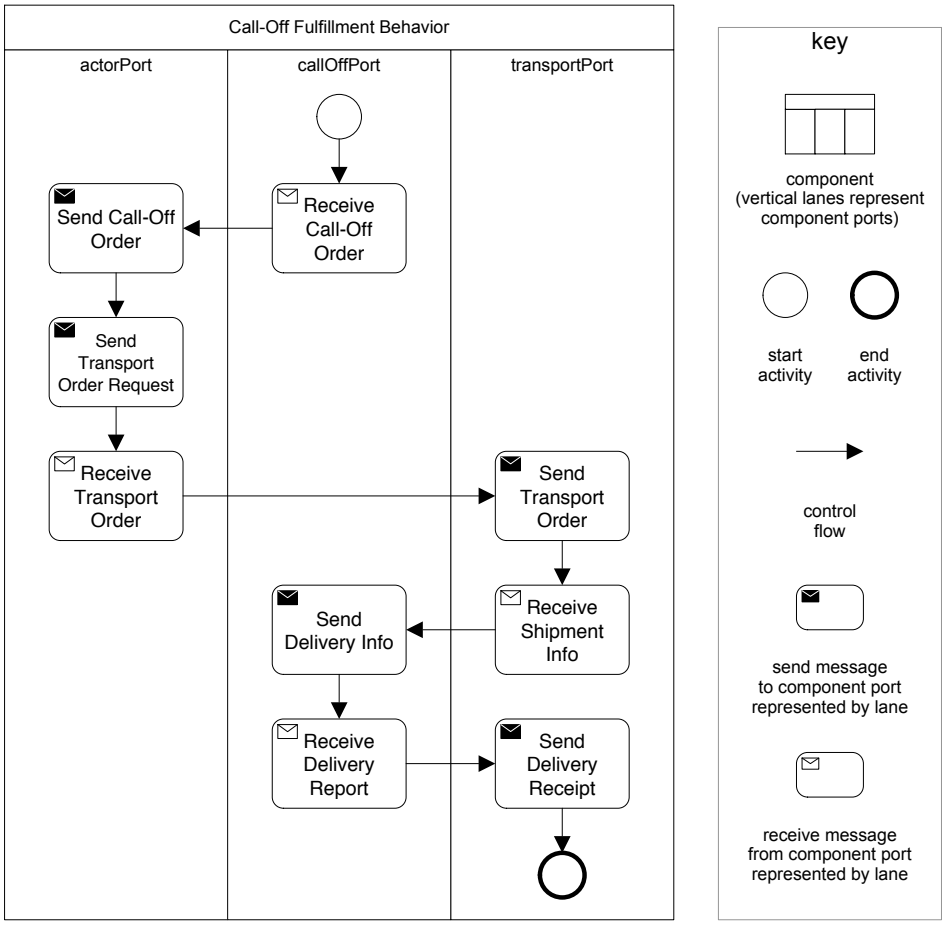


Figure 4.21: The documentation of the behavior of a behavior component using a BPMN diagram. Vertical lanes represent component ports.

		role state					
		Active		Deactivating		Inactive	
		creation	max occur.	creation	max occur.	creation	max occur.
conversation ports	callOffPort	yes	*	no	*	no	0
	consPort	yes	*	no	*	no	0
behaviors	Call-Off Behavior	actor	*	no	*	no	0

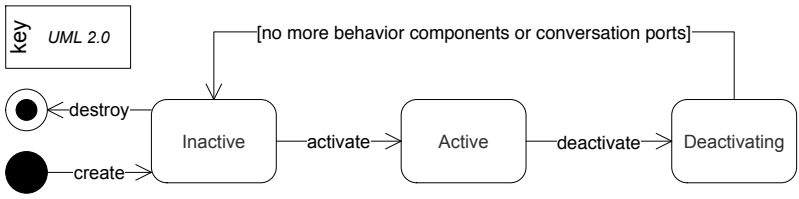


Figure 4.22: *Top:* The possible role states of the *Vmi Retailer* role component. *Bottom:* A state diagram defining how these states can change.

Each organization connector type enables a set of possible conversations between the different role component types. The *Vmi Organization* enables several conversations. The *Transport Conversation* connector is used between both the *Warehouse* and *Transporter* role, and the *Vmi Vendor* and *Transporter* role (see Appendix A).:

Forecasting Conversation
- *Vendor* → *VmiVendor.forecastPort*
- *Customer* → *VmiRetailerHQ.forecastPort*

Inventory Reporting Conversation
- *Inventory* → *Warehouse.invPort*
- *Client* → *VmiVendor.invPort*
- *Client* → *VmiRetailerHQ.invPort*

Call-Off Conversation
- *Caller* → *VmiRetailer.callOffPort*
- *Stock* → *Warehouse.callOffPort*

Transport Conversation
- *Shipper* → *Warehouse.transportPort*
- *Carrier* → *Transporter.transportPort*

Consumption Reporting Conversation
- *Consumer* → *VmiRetailer.consPort*
- *Buyer* → *VmiRetailerHQ.consPort*
- *Seller* → *VmiVendor.consPort*

Transport Conversation
- *Shipper* → *VmiVendor.tranportPort*
- *Carrier* → *Transporter.transportPort*

...

The conversations in this table can also be represented using the graphical notation of the Role & Conversation View (see Fig. 4.19, p. 105).

Using the possible states of the *Vmi Retailer* role component, the management logic of the 4PL (see Fig. 4.15, p. 94) can be further refined (Fig. 4.23).

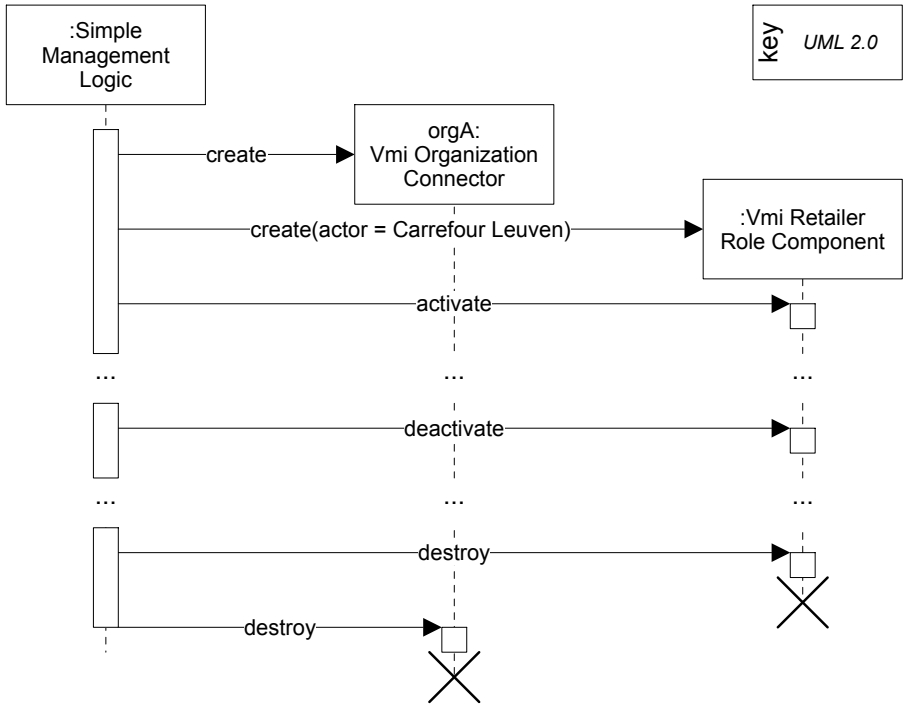


Figure 4.23: A simple management logic creates and destroys an organization connector and role component, and evolves the state of the role component.

4.5 Using Macodo Views

Using the style guide of a Macodo view, concrete instances of this view type can be created to document the actual architecture of a system. An example of such a view documentation can be found in Appendix A. The documentation of a view instance has five sections:

- A *Primary Presentation* shows the elements and relations in the view, typically represented in a graphical way. It is the starting point for documenting a view. The primary presentation may consists of more than one diagram. For example, a diagram for each type of organization in the system. This chapter provided a graphical notation for each type of Macodo view.
- An *Element Catalog* details at least those elements depicted in the primary presentation. It should include a description of all elements and relations in the view, and a specification of the element interfaces and behaviors. Interfaces of Macodo architectural elements can be documented like standard interfaces. Interfaces only need to be documented once. For example, to document the interfaces of actor ports and conversation ports, the element catalog can refer to interfaces defined in an Organization Module View. To document the behavior of Macodo architectural elements, semi-formal notations such as UML sequence diagrams and activity diagrams [193] and BPMN [154], or formal notations such as Z [189] and Alloy [123] can be used.
- A *Context Diagram* shows how the system or portion of the system depicted in the view relates to its environment.
- A *Variability Guide* shows how to exercise any variation points that are part of the architecture shown in the view. This section can be used to document both design time variability as well as runtime dynamics and adaptation. This chapter discussed how the variability guide of the Organization & Actor View and the Role & Conversation View can be used to document runtime adaptation of organization connectors, role components, conversation connectors, and behavior components.
- A *Rationale* section should explain the main architectural decisions that are made in the view.

In addition to the documentation of each view, the complete documentation of an architecture also contains elements such as an overview, a mapping between views, and general rationale. The mapping between views is of particular interest to document how the different elements of each view are related to each other.

4.6 Conclusions

In this chapter we presented a set of architectural views to design, model, and document collaborations in terms of software elements. Each view introduces a set of architectural modeling abstractions that reify some of the collaboration abstractions introduced in Chapter 3. The different views address the main architectural drivers derived from the problem statements of this thesis:

- *Functionality.* The three Macodo views allow to document collaboration functionality in terms of collaboration abstractions such as role components and conversation connectors.
- *Managing Complexity and Separation of Concerns.* The different Macodo views provide modeling abstractions to model participants responsibilities, individual behavior, and interactions as separate concerns.
- *Understandability and Expressibility.* Each Macodo view allows architects and other stakeholders to express and reason about particular collaboration qualities and concerns in terms of collaboration concepts.
- *Reuse.* The Organization Module View allows to decompose collaborations in terms of modules and to express commonalities and variations among these modules. The Organization & Actor View and Role & Conversation View can be used to express reuse of different connector and component types.
- *Modifiability and Runtime Adaptation.* The Organization & Actor View and Role & Conversation View can be used to focus on different aspects of runtime adaptation. The Organization & Actor View allows to express the creation and termination of collaborations, and the addition and removal of participants. The Role & Conversation View allows to reason about the concurrency of interactions within a collaboration, and to express the dynamics of individual roles in terms of evolving role states.
- *Conceptual Integrity.* Each Macodo view reifies a set of collaboration abstractions. Clear mappings between the elements of different Macodo views allow to describe a system in a uniform way.

Chapter 5

Proof of Concept Middleware Infrastructure

In this chapter we present a proof of concept middleware infrastructure for Macodo. The scope of this middleware is defined by two main goals. First, we want to provide a proof of concept platform to implement collaborations that are designed using the Macodo architectural views, presented in Chapter 4. Second, we want to show how Macodo can be integrated in the current technology stack without the need for new standards. This is done by mapping the Macodo collaboration abstractions to concrete Web service technology and supporting them as programming abstractions.

This chapter does not introduce any new concepts to Macodo. To fully understand the technical exposition in this chapter, a basic knowledge about Web service technologies, introduced in Chapter 2, is required.

5.1 Introduction

Middleware is commonly defined as the software layer that sits between the operating system and the applications on each site of a distributed system [137]. In a broader sense, it can be seen as the set of software services that enables the interoperation of distributed software components running on different machines and operating systems. Middleware typically offers a set of programming abstractions to facilitate the development of complex distributed systems. Good abstractions are key to successful software engineering [86, 210, 138]. Abstractions can hide low level details of hardware, networks and distribution, and provide developers access to functionality that otherwise would have to be implemented from scratch. But middleware is also infrastructure. For abstractions to be useful, they need good

supporting infrastructure, providing a comprehensive platform for developing and running complex distributed systems [12].

In this chapter we present the Macodo middleware, a middleware that supports the Macodo abstractions at implementation level. The middleware is based on a mapping of the Macodo architectural modeling abstractions to concrete Web service technologies. A discussion of these technologies can be found in Sect. 2.5, p. 26.

Middleware Mapping in a Nutshell

An informal overview of the middleware mapping is shown in Fig. 5.1. Conversation and behavior capabilities are mapped to partnerLinkTypes and can be defined using the Web Services Description Language (WSDL). A role capability becomes a set of these partnerLinkTypes. Providing and requiring a capability translates to the ability to expose and use a set of Web services.

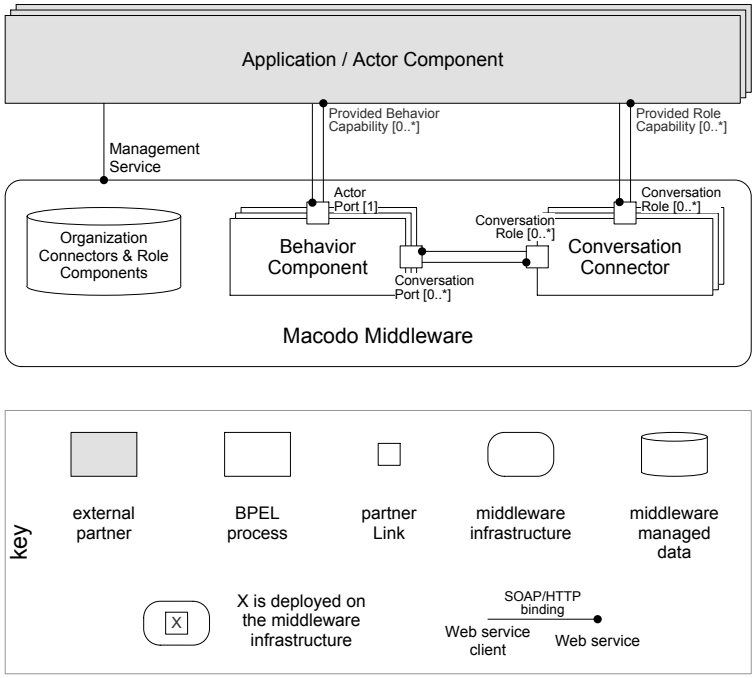


Figure 5.1: An informal overview of the mapping of Macodo architectural modeling abstractions to Web service technology.

Actor components can then be mapped to components or systems that interact with the conversation connectors, behavior components, and the Macodo middleware using SOAP over HTTP. The role port of an actor component translates to a set of

provided behavior capabilities and conversation capabilities. Actor components can also use a management service, exposed as a Web service by the Macodo middleware, which allows to register actor components in the Macodo middleware and manage the life-cycle of organization connectors and role components.

Conversation connectors and behavior components are defined and executed as BPEL (Business Process Execution Language) processes. Conversation roles of conversation connectors, and actor ports and conversation ports of behavior components become partnerLinks of these BPEL processes (Fig. 5.1). Concrete instances of conversation connectors and behavior components become process instances that execute on a BPEL engine.

Organization connectors and role components do not have a direct mapping to Web service technology. They become data structures maintained by the middleware. The Macodo middleware uses these data structures to mediate the interactions between actor components, conversation connectors, and behavior components according to the current organizations and roles.

Overview. Section 5.2 discusses how to implement organizations using the Macodo middleware. Next, Sect. 5.3 covers how to concretely deploy, manage and use organization connectors and role components. Sect. 5.4 gives a high-level overview of a proof of concept middleware architecture and discusses a prototype implementation. Section 5.5 concludes and discusses possible extensions to the proof of concept middleware and the current prototype implementation.

5.2 Implementing Organizations

In terms of implementation units, the Macodo middleware can be seen as a software layer that is used by a set of application layers (Fig. 5.2). The application layers can be divided in an actor layer and an organization layer. The actor layer is allowed to use the organization layer and consists of the actors in a system and other application specific modules. The organization layer contains a set of application specific organization modules, role modules, conversation modules, and behavior modules, and is allowed to use the Macodo middleware layer. The architectural design of the organization layer can be described using the Organization Module View (Sect. 4.2).

Using a bottom-up approach, this section discusses how each module of the organization layer can be implemented in the Macodo middleware using XML (Extensible Markup Language), WSDL, and WS-BPEL. This results in a set of implementation packages. An overview of these packages is shown in Fig. 5.3. The complete XML schema definitions for the different modules can be found in Appendix C.1. The implementation of the actor layer is outside the scope of this thesis.

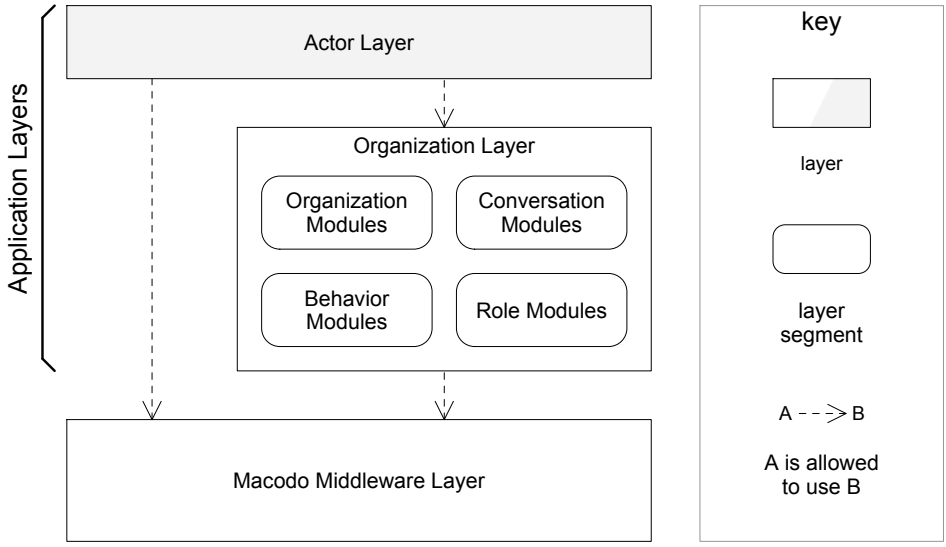


Figure 5.2: The Macodo middleware can be seen as a software layer that is used by an application specific organization layer and actor layer.

5.2.1 Specifying Capabilities

Conversation and behavior capabilities can be specified using XML and WSDL. A capability is defined in an XML file, which contains a name, a reference to a WSDL definition, and a mapping of the capability interfaces to portTypes defined in the WSDL definition. An example of a behavior capability for the *Inventory Reporting Behavior* and a conversation capability for the *Inventory* conversation role is given below.

```
1  <behaviorCapability name="InventoryReportingCapability"
2                      interfaceSpecification="InvRepBehCap.wsdl"
3                      actorPortType="actorPortType"
4                      behaviorPortType="behaviorPortType"/>
5
6  <conversationCapability name="InventoryReportingInventoryCapability"
7                          interfaceSpecification="InvCap.wsdl"
8                          participantPortType="participantPortType"
9                          conversationPortType="conversationPortType"/>
```

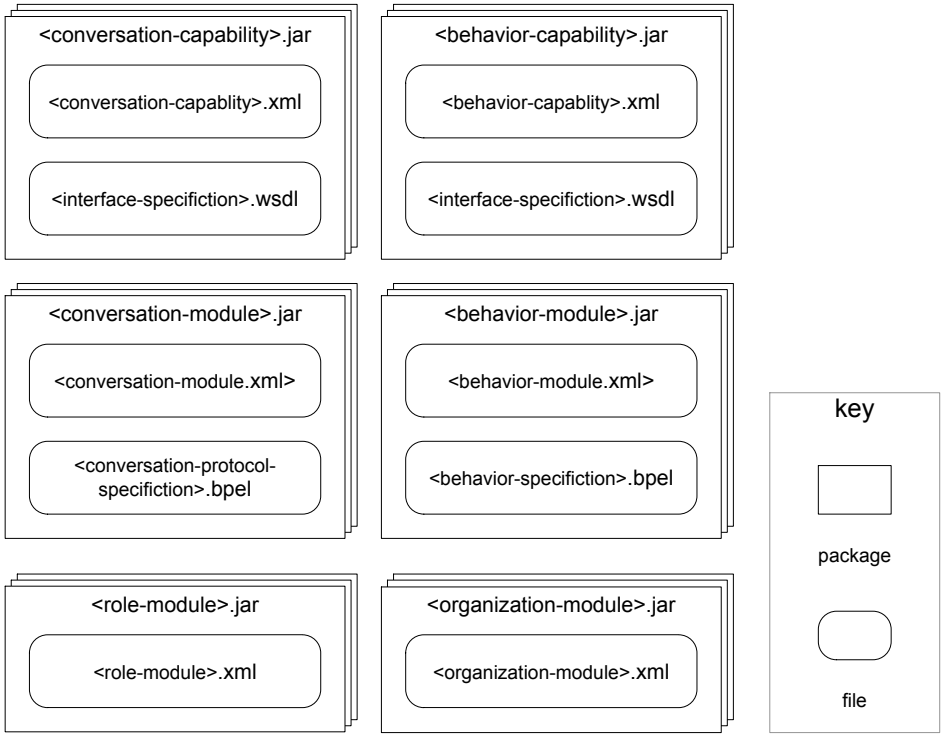


Figure 5.3: The organization layer can be specified in the Macodo middleware using XML, WSDL, and WS-BPEL.

5.2.2 Implementing Conversation Modules

A conversation module defines a specific conversation connector type. It consists of an XML file, which defines the key properties of the conversation connector type, and a BPEL definition, which specifies the conversation protocol as a BPEL process. The XML file has the following elements:

- The name of the conversation connector type.
- A reference to the BPEL definition that specifies the conversation protocol.
- A set of possible conversation roles. Each conversation role has a name, a required conversation capability, a minimum and maximum number of parallel occurrences, and a reference to a partnerLink of the BPEL definition. The latter maps the conversation role to a concrete partnerLink of the BPEL definition.

An example of a conversation module is given below, showing the specification of a conversation connector type for the *Inventory Reporting Conversation*. The conversation has two possible conversation roles, *Inventory* and *Client*:

```
<conversationConnectorType name="InventoryReportingConversation"
                           protocolSpecification="InvRepConversation.bpel">
  <conversationRole name="Inventory"
                    conversationCapability="InventoryReportingInventoryCapability"
                    minOccur="1" maxOccur="1" initiator="true"
                    partnerLink="Inventory"/>
  <conversationRole name="Client"
                    conversationCapability="InventoryReportingClientCapability"
                    minOccur="1" maxOccur="unbounded" initiator="false"
                    partnerLink="Client"/>
</conversationConnectorType>
```

The BPEL definition specifies the conversation protocol. It has a partnerLink for each conversation role. This partnerLink implements the conversation interface defined in the partnerLinkType of the corresponding conversation capability, and uses the participant interface. A graphical overview (using BPMN) of such a BPEL definition is given in Fig. 5.4.

5.2.3 Implementing Behavior Modules

A behavior module defines a specific behavior component type. It consists of an XML file, defining the key properties of the behavior component type, and a BPEL definition, specifying the actual behavior as a BPEL process. The XML file has the following elements:

- The name of the behavior component type.
- A reference to the BPEL definition that specifies the actual behavior.
- An actor port. The actor port has a required behavior capability and a reference to a partnerLink of the BPEL definition.
- A set of possible conversation ports. Each conversation port has a name, a provided conversation capability, and a reference to a partnerLink of the BPEL definition.

These elements allow to map the actor port and conversation ports of the behavior component to concrete partnerLinks of the BPEL definition. An example of a behavior module is given below, specifying a behavior component type for the *Inventory Reporting Behavior*. The behavior has an actor port and one conversation port, allowing the behavior to play the *Inventory* conversation role.

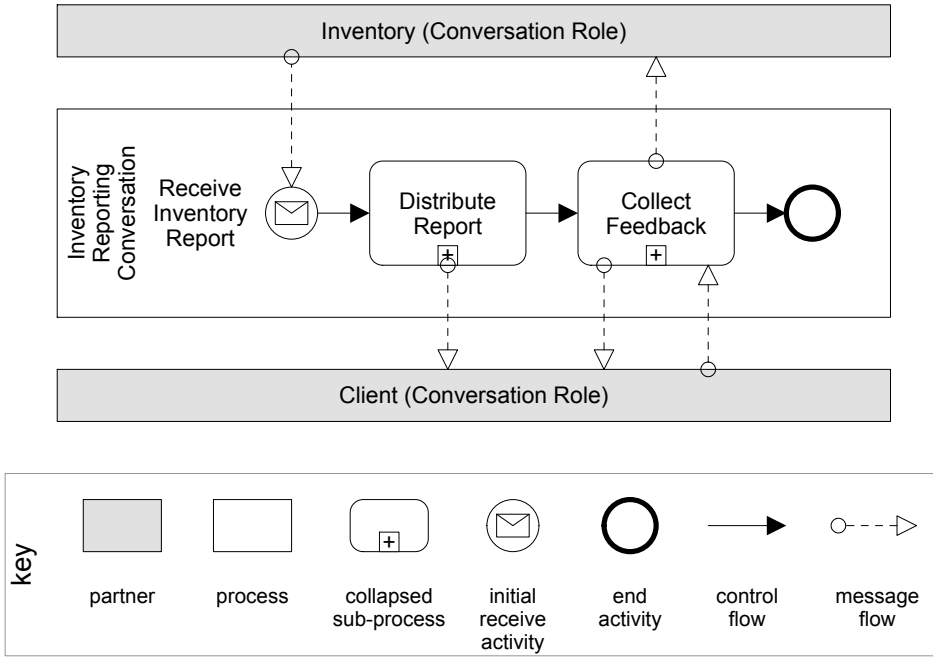


Figure 5.4: The BPEL definition for the inventory reporting conversation, represented in BPMN.

```
<behaviorComponentType name="InventoryReportingBehavior"
                       behaviorSpecification="InvRepBehavior.bpel">
  <actorPort behaviorCapability="InventoryReportingCapability"
             partnerLink="ActorPort"/>
  <conversationPort name="InventoryReportingPort"
                    conversationCapability="InventoryReportingInventoryCapability"
                    partnerLink="InventoryReportingPort"/>
</behaviorComponentType>
```

The referenced BPEL definition specifies the actual behavior. It has one partnerLink for the actor port and one for each conversation port. The partnerLink for the actor port implements the behavior interface defined in the partnerLinkType of the corresponding behavior capability, and uses the actor interface. The partnerLinks for each conversation port implement the participant interface and use the conversation interface of the corresponding conversation capabilities. A graphical overview (using BPMN) of such a BPEL definition is given in Fig. 5.5.

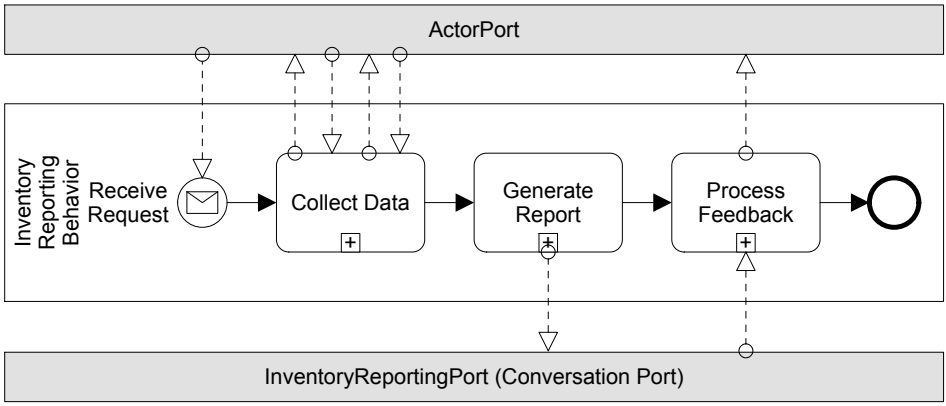


Figure 5.5: The BPEL definition for the inventory reporting behavior, represented in BPMN (key can be found in Fig. 5.4).

5.2.4 Implementing Role Modules

A role module defines a specific role component type. A role module is specified in an XML file containing the following elements:

- The name of the role component type.
- A set of possible conversation ports. Each conversation port has a name and a provided conversation capability.
- A set of possible behavior types. Each behavior type has a name and a behavior component type, defined in a behavior module.
- A set of interface delegations. Each interface delegation delegates a conversation port of the role component to a conversation port of one of the behavior types. Actor ports are automatically delegated. These delegations do not have to be specified.
- A set of possible role states. Each role state defines a set of conversation constraints and behavior constraints. A conversation constraint defines whether a specific type of conversation port can be created and the maximum number of concurrent instances. A behavior constraint defines whether a behavior component of a specific behavior type can be created and the maximum number of concurrent instances.

There can be multiple behavior types for the same behavior component type. This allows to reuse behavior component types (e.g., to apply different behavior constraints to each behavior type). Actor ports of role components do not need

to be specified, since they can be derived automatically as the aggregation of all actor ports of behavior types and all conversation ports of the role component that are not delegated to behavior types. An example of a role module for the *Warehouse* role is given below (only some of the interface delegations and role states are shown). The role module defines a role component type with three types of conversation ports and two behavior types:

```
<roleComponentType name="Warehouse">
  <conversationPort name="InventoryPort"
                   conversationCapability="InventoryReportingInventoryCapability"/
  <conversationPort name="StockPort"
                   conversationCapability="CallOffStockCapability"/>
  <conversationPort name="ShipperPort"
                   conversationCapability="TransportShipperCapability"/>

  <behaviorType name="InventoryReportingBehavior"
                behaviorComponentType="InventoryReportingBehavior"/>
  <behaviorType name="CallOffFulfillmentBehavior"
                behaviorComponentType="CallOffFulfillmentBehavior"/>

  <interfaceDelegation behaviorType="InventoryReportingBehavior"
                       behaviorConversationPort="InventoryPort"
                       roleConversationPort="InventoryPort"/>
  ...

  <roleState name="Inactive">
    <conversationConstraint conversationPort="InventoryPort"
                            creation="false" maxOccur="0"/>
    <conversationConstraint conversationPort="StockPort"
                            creation="false" maxOccur="0"/>
    <conversationConstraint conversationPort="ShipperPort"
                            creation="false" maxOccur="0"/>
    <behaviorConstraint behaviorType="InventoryReportingBehavior"
                        creation="false" maxOccur="0"/>
    <behaviorConstraint behaviorType="CallOffFulfillmentBehavior"
                        creation="false" maxOccur="0"/>
  </roleState>
  ...
</roleComponentType>
```

5.2.5 Implementing Organization Modules

An organization module defines a specific organization connector type. An organization module is specified in an XML file (Fig. 5.3) containing the following elements:

- The name of the organization connector type.

- A set of possible role types. Each role type has a name, a role component type (defined in an external role module), and the minimum and maximum number of parallel occurrences.
- A set of possible conversation types. A conversation type specifies a particular type of conversation between specific role types of the organization. Each conversation type has a name, a conversation connector type (defined in an external conversation module), and a set of role mappings. A role mapping maps a conversation role of the conversation connector type to a conversation port of a specific role type of the organization connector.

There can be multiple role types for the same role component type, and multiple conversation types for the same conversation connector type. This allows to reuse role component types and conversation connector types within an organization connector type. An example of an organization module for the *Vmi Organization* is given below (only some of the conversation types are shown). The module defines an organization connector type with six role types and a number of conversation types:

```
<organizationConnectorType name="VmiOrganization">
  <roleType name="Vendor" roleComponentType="VmiVendor"
            minOccur="0" maxOccur="1"/>
  <roleType name="Warehouse" roleComponentType="Warehouse"
            minOccur="0" maxOccur="1"/>
  <roleType name="Retailer" roleComponentType="VmiRetailer"
            minOccur="0" maxOccur="1"/>
  <roleType name="RetailerHQ" roleComponentType="VmiRetailerHQ"
            minOccur="0" maxOccur="1"/>
  <roleType name="Transporter" roleComponentType="Transporter"
            minOccur="0" maxOccur="1"/>

  <conversationType name="CallOffTransportConversation"
                    conversationConnectorType="TransportConversation">
    <roleMapping conversationRole="Shipper"
                 roleType="Warehouse"
                 conversationPort="ShipperPort"/>
    <roleMapping conversationRole="Carrier"
                 roleType="Transporter"
                 conversationPort="CarrierPort"/>
  </conversationType>
  <conversationType name="ReplenishmentTransportConversation"
                    conversationConnectorType="TransportConversation">
    <roleMapping conversationRole="Shipper"
                 roleType="Vendor"
                 conversationPort="ShipperPort"/>
    <roleMapping conversationRole="Carrier"
                 roleType="Transporter"
                 conversationPort="CarrierPort"/>
  </conversationType>
  ...
</organizationConnectorType>
```

In the above example, the *Transport Conversation* connector type is used twice. Once between the *Warehouse* role and the *Transporter* role, and once between the *Vendor* role and the *Transporter* role. This means, that in an actual organization connector of the *Vmi Organization* type, roles of type *Warehouse* and *Vendor* can initiate a *Transport Conversation* with the *Transporter* role.

5.3 Deploying and Using Organizations

Once defined, the modules of the organization layer, can be loaded in the Macodo middleware. The management service of the Macodo middleware can then be used to register actors and to manage the life-cycle of concrete organization connectors and role components. Once a role has been assigned to an actor, the actor can use the role. The management service of the Macodo middleware is exposed as a Web service, but illustrated as a Java interface in this section. A complete specification of the *Management Service* can be found in Appendix D.

5.3.1 Registering Actors

In order to assign actors to roles in organizations, actors first needs to be registered in the middleware together with their provided capabilities. The management service provides a set of operations that allow to register actors and provided capabilities of actors:

```
1  ActorID registerActor(String name);
2
3  void registerProvidedCapability(ActorID actorComponentID,
4                                  ProvidedCapability providedCapability)
5    throws NoSuchActorException, InvalidCapabilityException;
```

A provided capability is defined as a reference to a conversation or behavior capability, and a concrete endpoint for the participant or actor interface of the capability. An example is given below:

```
1  <providedCapability ID="pc35" capability="InventoryCapability">
2      <endpointReference address="http://acme.com:8080/capabilities"
3                         portType="InventoryCapabilityPort"
4                         serviceName="InventoryCapabilityService"/>
5  </providedCapability>
```

By registering provided capabilities of actors in the Macodo middleware, the middleware knows how and where to reach the actor for each capability.

5.3.2 Managing the Life-Cycle of Organization Connectors and Role Components

Managing Organization Connectors

The life-cycle of organization connectors is managed using the management service of the Macodo middleware. There are two main life-cycle operations: creating organization connectors and destroying organization connectors:

```
1  OrganizationConnectorID createOrganizationConnector(
2                      OrganizationConnectorType organizationConnectorType,
3                      String name)
4    throws InvalidOrganizationConnectorTypeException;
5
6  void destroyOrganizationConnector(
7                      OrganizationConnectorID organizationConnectorID)
8    throws NoSuchOrganizationConnectorException;
```

Organization connectors can only be created of organization connector types that are loaded in the Macodo middleware. Destroying an organization connector destroys all encapsulated role components, conversation connectors, and behavior components.

Managing Role Components

The life-cycle of role components is also managed using the Macodo management service. There are three main life-cycle operations: creating role components, destroying role components, and changing the state of role components:

```
1  RoleComponentID createRoleComponent(
2                      OrganizationConnectorID organizationConnectorID,
3                      RoleType roleType,
4                      ActorID actorID)
5    throws NoSuchOrganizationConnectorException, IllegalRoleTypeException,
6           RoleMultiplicityException, NoSuchActorException,
7           InsufficientCapabilitiesException;
8
9  void destroyRoleComponent(RoleComponentID roleComponentID)
10   throws NoSuchRoleComponentException;
11
12 void setRoleState(RoleComponentID roleComponentID,
13                    RoleState roleState)
14   throws NoSuchRoleComponentException, IllegalRoleStateException;
```

Role components are always created within a specific organization connector, for a specific role type supported by this organization connector, and for a registered

actor that provides sufficient capabilities. Destroying a role component destroys all encapsulated behavior components.

The state of a role component can only be changed to a state defined by the role component type. The Macodo middleware does not enforce any state machine, this is an application specific concern.

Once a role component is created, the actor of the role needs to be able to use the role component. To do so, the actor can retrieve a role endpoint and the current state of a role component:

```
RoleEndpoint getRoleEndpoint(RoleComponentID roleComponentID)
  throws NoSuchRoleComponentException

Rolestate getRoleState(RoleComponentID roleComponentID)
  throws NoSuchRoleComponentException;
```

A role endpoint contains all information for the actor to actually use the role, such as, the endpoints of each behavior type and conversation type. The role state tells the actor which conversations and behavior are available. An example of a role endpoint is given below:

```
<roleEndpoint roleComponentID="rc543"
              roleType="Warehouse">
  <behavior behaviorType="InventoryReportingBehavior">
    <endpointReference address="http://acme.com:8080/behaviors"
                       portType="InventoryReportingBehaviorPort"
                       serviceName="InventoryReportingBehaviorService"/>
  </behavior>
  <behavior behaviorType="CallOffFulfillmentBehavior">
    <endpointReference address="http://acme.com:8080/behaviors"
                       portType="CallOffFulfillmentBehaviorPort"
                       serviceName="CallOffFulfillmentBehaviorService"/>
  </behavior>
</roleEndpoint>
```

Example

Fig. 5.6 gives an example of an application component that uses the Macodo management service to perform a sequence of operations: (1) register an actor; (2) register a capability of this actor; (3) create an organization connector; (4) create a role component; (5) retrieve the endpoint of this role component. The role endpoint is then passed to an actor component (6), which can now 'play the role'.

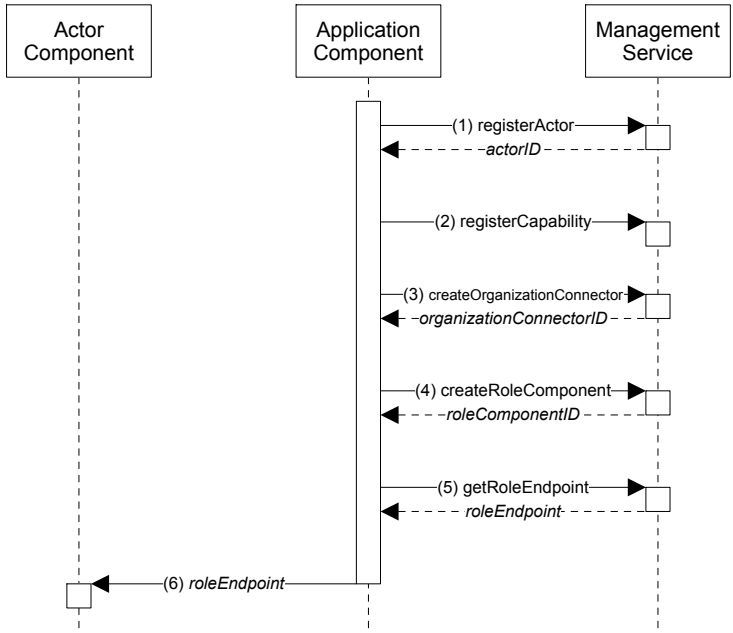


Figure 5.6: An example of an application component using the Macodo management service to manage the life-cycle of organization connectors and role components.

5.3.3 Using Role Components

Once an actor is registered in the Macodo middleware and assigned a role component, it can 'play' the role using the role endpoint. Playing a role means the actor participates in conversations and executes behaviors that are available in the current role state. To participate in a conversation, an actor interacts with a conversation connector. To execute a behavior, an actor interacts with a behavior component. Behavior components can also interact with conversation connectors, when they realize a conversation role. These interactions can be initiated by an actor, a conversation connector, or a behavior component.

The interactions between actor components, conversation connectors, and role components are mediated by the Macodo middleware. All messages pass through the Macodo middleware, which routes these messages to the correct conversation connector, behavior component, or actor component.

In order for the Macodo middleware and for actor components to determine the role component in whose context messages are send, all messages between the Macodo middleware and the actor components contain additional Macodo data. In its most simple form, this data consists of a role component ID, which uniquely identifies the role component to which a message belongs. Macodo data can be

added as a custom SOAP header or as an additional message part[1].

Example

Figure 5.7 shows an example of an actor component executing a behavior. The actor initiates the behavior by sending the initial message to the middleware. The middleware then creates a new behavior component and routes all messages between the new behavior component and the actor component. Messages between the middleware and the actor component contain additional Macodo data.

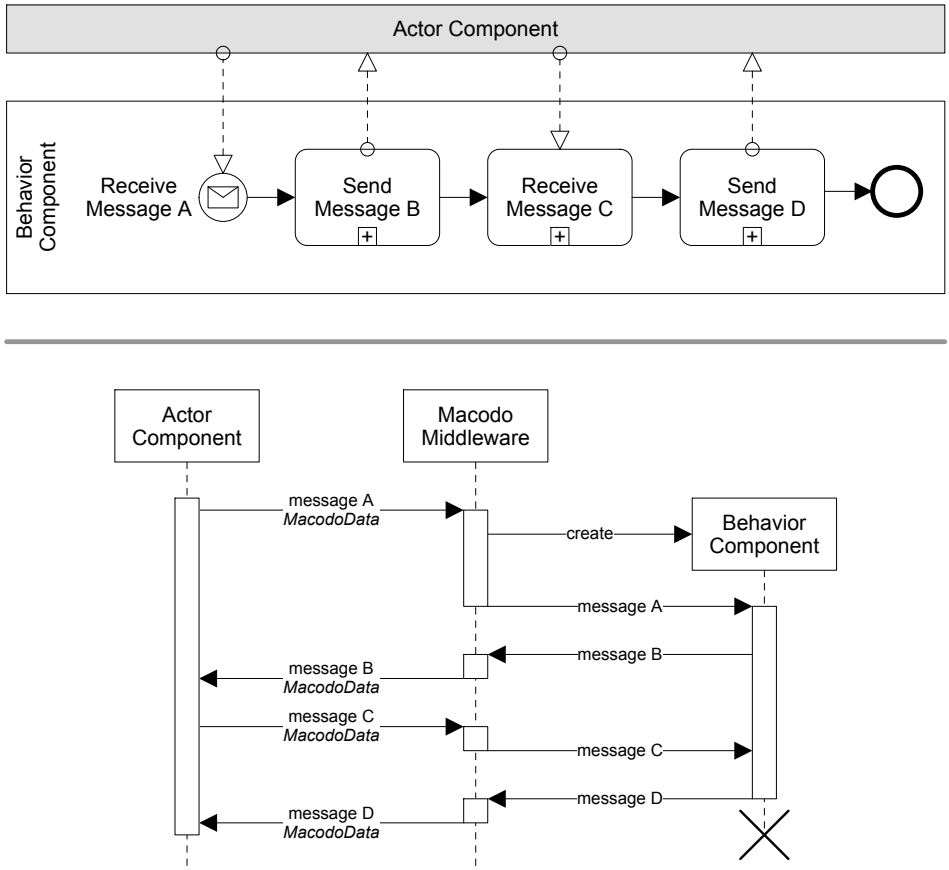


Figure 5.7: An example of an actor component executing behavior. *Top:* The specification of a simple behavior component (key can be found in Fig. 5.4, p. 119). *Bottom:* The middleware creates the behavior component and mediates the messages.

[1]This option is only available when using the document style binding.

5.4 Proof of Concept Middleware Architecture

In this section we present a proof of concept architecture for the Macodo middleware. We give a high-level overview of the architecture (Sect. 5.4.1) and briefly discuss a prototype implementation based on instrumentation (Sect. 5.4.2).

5.4.1 High-Level Component & Connector View

The core architecture consists of four component types (*Conversation Container*, *Behavior Container*, *Organization Manager* and *Organization Mediator*) and a set of shared data repositories (Fig. 5.8). The shared data repositories are used to persist organization state and to synchronize the actions of different middleware components.

Conversation containers and behavior containers host the conversation connectors and behavior components. They are deployed on a separate BPEL engine and have three main responsibilities: configuring conversation connectors and behavior components; managing the life-cycle of conversation connectors and behaviors components; and adding correct Macodo data to all messages send by conversation connectors and behavior components. Configuring conversation connectors and behavior components consists of binding each partnerLink (i.e., conversation roles, actor ports, and conversation ports) to concrete Web services of participants and actors. Managing the life-cycle of conversation connectors and behaviors components includes checking whether they can be created given the current role state of participants and actors. To realize these responsibilities, conversation containers and behavior containers have access to an organization mediator service, exposed by the organization mediator components. These organization mediator components are responsible for mediating the interactions between actors, conversation connectors, and behavior components, according to the current organization connectors and role components.

The organization manager components are responsible for the management of organization connectors and role components. They provide a management service (discussed in Sect. 5.3) which can be used by the actor components to register actors, and manage the life-cycle of role components and organization connectors. Which actors or application components have access to this management service is an application-specific concern.

To serve multiple clients in parallel, the organization manager and organization mediator components run as stateless session beans on a Java EE server. Each client is served by a separate bean instance. Interaction between actor components and the different middleware components is based on SOAP over HTTP. Every component exposes and uses a set of Web services.

The organization manager and organization mediator components use a set of shared data repositories to store data, communicate, and synchronize their actions.

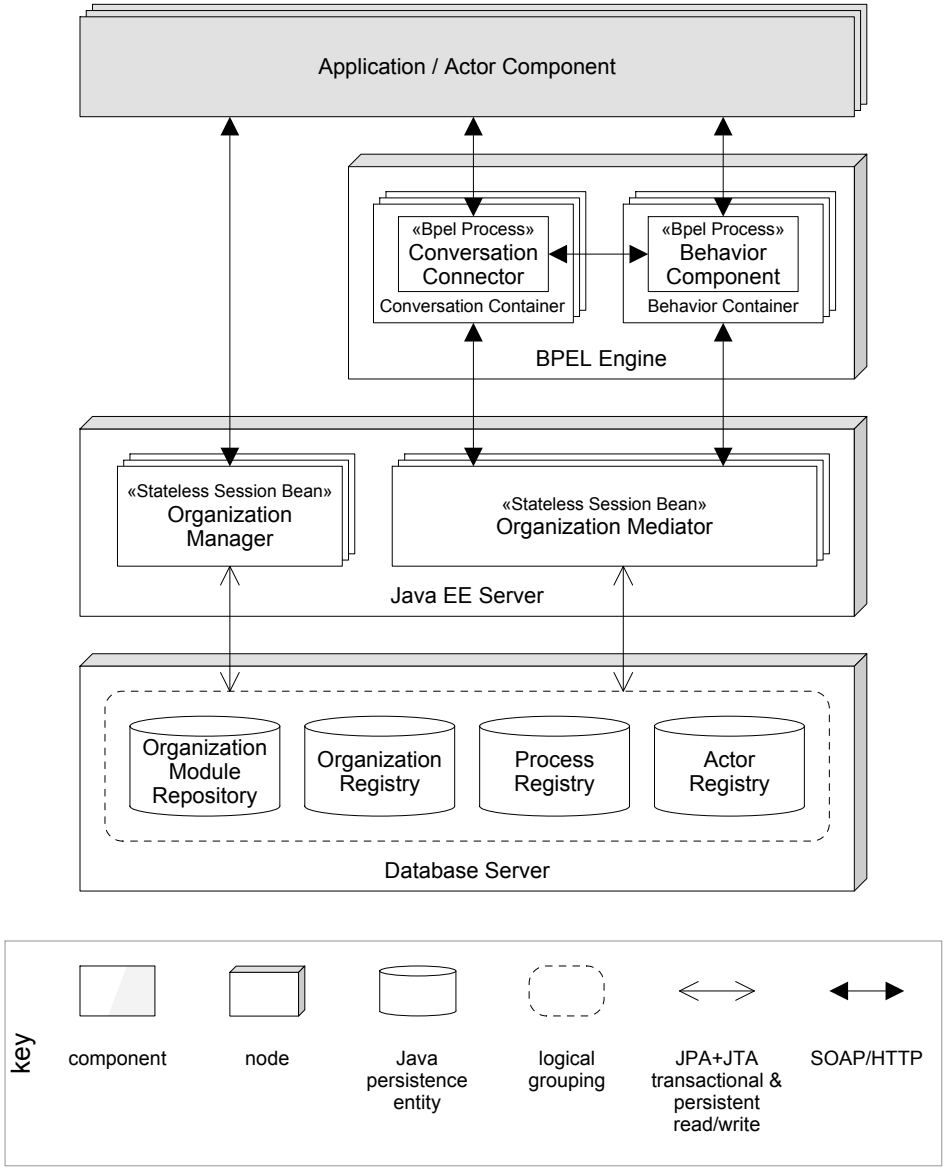


Figure 5.8: A high-level overview of the Macodo middleware architecture.

Interaction with these repositories is done using the Java Persistence API (JPA) and the Java Transaction API (JTA) to ensure persistence and data consistency. The data repositories can be divided in four repositories that each store a specific type of data:

- The *Organization Module Repository* maintains the specification of the all available organization connector types, role component types, conversation connector types, and behavior component types that are loaded in the Macodo middleware.
- The *Organization Registry* maintains the current organization connectors and their role components. This registry also maintains the current state of role components and references to active conversation connectors and behavior components.
- The *Process Registry* maintains all data (i.e., WSDL definitions and service endpoints) to use currently deployed conversation connector types and behavior component types.
- The *Actor Registry* maintains all data on actors and their provided capabilities that are registered in the Macodo middleware.

Rationale. Using a shared data style between the organization manager and organization mediator components simplifies synchronization, and allows to keep instances of middleware services stateless and short-lived. These stateless and short-lived middleware services can be replicated and deployed on multiple servers to improve performance and increase availability. Furthermore, the shared data style decouples data producers and data consumers, and allows to modify existing middleware services, or add new middleware services, without affecting the way data is stored and maintained. By dividing the repositories in four sub-repositories, each repository can focus on a specific type of data, to further increase separation of concerns.

Using SOAP over HTTP for interactions between the different middleware components and actors enables interoperability. It allows to distribute middleware components and actors, and to deploy them on different types of platforms. For example, conversation connectors and behavior components are deployed on a BPEL engine, middleware services run on a Java EE server, and actors can be deployed on any type of platform that supports SOAP over HTTP.

5.4.2 Prototype Implementation

A prototype implementation was built using Java EE and Open ESB[2]. The implementation relies on the instrumentation of the BPEL definitions of

[2] http://openesb-community.org/

conversation connectors and behavior components. Instrumentation means that certain activities, such as service calls and variable assignments are automatically added or weaved into an existing BPEL definition. Instead of explicitly hosting conversation connectors and behavior components in a container, container logic is instrumented in the original BPEL definitions (Fig. 5.9). The resulting instrumented BPEL definitions are directly deployed on a BPEL engine.

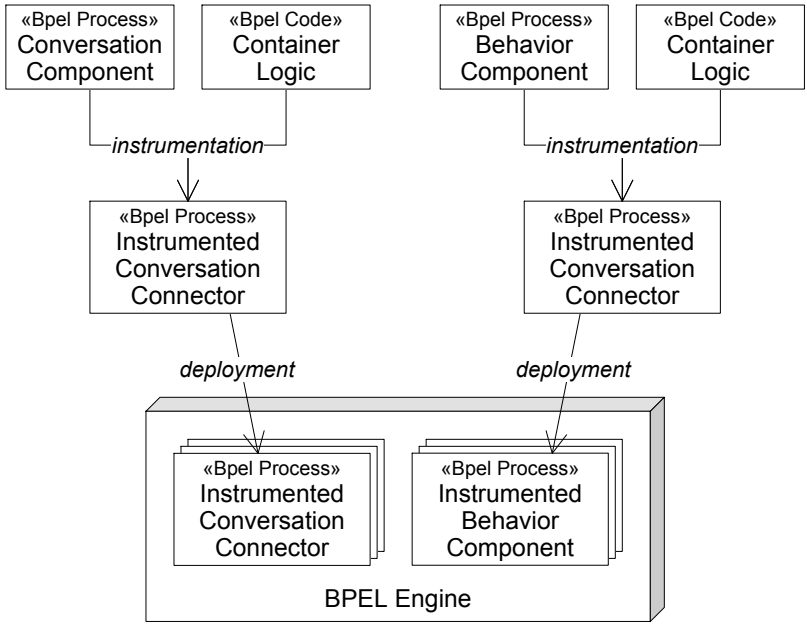


Figure 5.9: A conceptual representation of the process to instrument conversation connectors and behavior components with container logic.

Instances of conversation connectors and behavior components are now created like normal process instances. That is, by sending messages. More specifically, when sending a message to a conversation connector or behavior component, the message is not send directly to the corresponding BPEL process. It is send to the BPEL engine on which the corresponding BPEL definition is deployed. The BPEL engine then determines whether a new process instance needs to be created, or whether the messages needs to be send to an existing process instance. This decision is made using message correlation (see Sect. 2.5.4). The correlation of messages in Macodo is similar to normal correlation and is typically application-specific.

The instrumentation of conversation connectors and behavior components adds three types of container logic: life-cycle management logic, configuration logic, and Macodo data logic. Life-cycle management logic can be divided into registration logic and termination logic. Registration logic is added right after the first activity

of the BPEL definition (Fig. 5.10, bottom). It registers the conversation connector or behavior component using the organization mediator service, which checks whether the conversation connector or behavior component can be created. If it cannot be created, the organization mediator service generates a fault, which is passed on to the actor. If registration is successful, it returns all required information to configure the conversation connector or behavior component.

Configuration and Macodo data logic is added right after the registration logic (Fig. 5.10, bottom). This logic uses data retrieved from the organization mediator service to configure all partnerLinks of the BPEL definition and to add Macodo data to all outgoing messages. The termination logic is added after the final activity of the BPEL definition (Fig. 5.10, bottom). Termination uses the organization mediator service to signal that the conversation connector or behavior component has terminated.

Example

Figure 5.11 shows a sequence diagram of the initiation of a behavior component defined in Fig. 5.10. The actor initiates the behavior by sending the first message of the behavior (*message A*). This message is intercepted by the BPEL engine, which based on existing correlation data decides to create a new process (or behavior component) instance. The message is then passed to the newly created behavior component.

The container logic in the behavior component uses the Macodo data attached to the first message to register the behavior component with the organization mediator service. If the actor component has the correct role in the correct state, the organization mediator service returns all required confirmation data for the behavior component. If the behavior component cannot be created, the mediator service returns a fault, which is passed on to the actor.

If registration is successful, the configuration and Macodo data logic in the behavior component use the returned configuration data to fully configure the behavior component. The behavior component can now interact with the actor component. After the final message of the behavior component, the termination logic of the behavior component signals the termination using the organization mediator service.

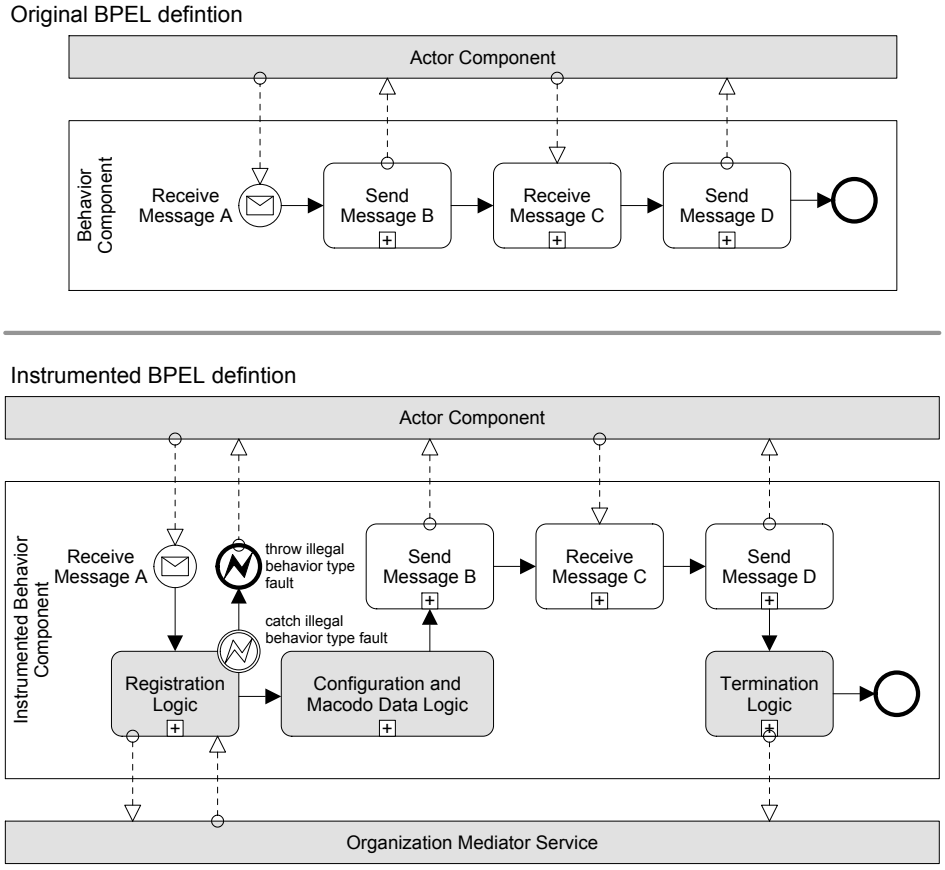


Figure 5.10: An example of a BPEL definition for a behavior component instrumented with container logic.

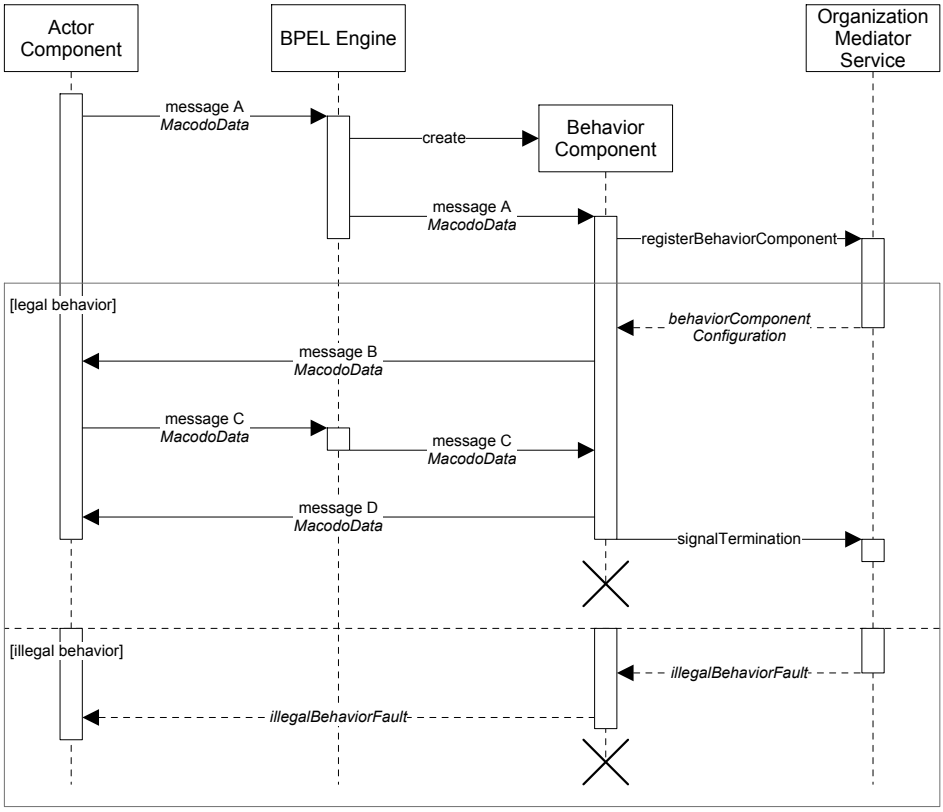


Figure 5.11: An example of an actor component initiating an instrumented behavior component.

5.5 Conclusions

This chapter presented a proof of concept middleware for Macodo. The middleware is based on a mapping of Macodo architectural abstractions to concrete Web service technology. Each type of Macodo module can be implemented using XML, WSDL, and WS-BPEL. A management service provided by the Macodo middleware allows applications to register actors and to manage the life-cycle of organization connectors and role components. This chapter also presented a proof of concept architecture for the Macodo middleware and discussed a prototype implementation based on process instrumentation.

Extensions to the Proof of Concept Middleware

The main purpose of the middleware presented in this chapter is to provide a proof of concept for integrating the Macodo abstractions as programming abstractions in the current Web service technology stack without the need for new standards. There are some interesting directions to extend the current middleware architecture and prototype implementation. Key extensions to the middleware architecture are:

- Supporting conversation connectors and behavior components that are not BPEL-based. Interesting options for conversation connectors are shared-data repositories and publish-subscribe mechanisms. Alternative behavior components could be based on Java EE components or executable BPMN definitions.
- Supporting the dynamic addition of Macodo modules at runtime.
- Supporting conversations in which participants are determined dynamically at runtime. In the current middleware architecture, participants of a conversation connector are fully determined when a conversation connector is created and cannot change during the execution of the encapsulated interaction. This extension, however, will make the specification and execution of conversation connectors more complex.
- Allowing actors to explicitly select the other participants of a conversation. In the current middleware architecture, the mapping of conversation roles to organization roles is defined by the organization connector type at design time. Actor components playing a role in an organization, can only initiate predefined conversations.

Interesting extensions to the current prototype implementation include:

- Automating the instrumentation of conversation connectors and behavior components. This can be done in a straight-forward manner. The current

prototype relies on manual instrumentation of BPEL definitions with container logic.

- Supporting conversation connectors with conversation roles of a higher multiplicity. Due to our choice of Open ESB, which currently does not provide a full implementation of the WS-BPEL 2.0 standard, conversation role multiplicity is currently limited to one.

Chapter 6

Evaluation: A Controlled Experiment

In this chapter, we evaluate two main contributions of this thesis: the Macodo model and the Macodo architectural views. The evaluation consists of a study that compares Macodo with a reference approach in the context of designing systems that support centrally managed collaborations between a set of Web services. The study is performed with students of a Master in Software Engineering program from universities in Sweden and Ukraine.

6.1 Introduction

The goal of the evaluation is to address the following research question:

> Do the modeling abstractions provided by Macodo:
>
> 1. reduce fault density,
> 2. reduce design complexity,
> 3. increase the level of reuse,
> 4. increase productivity,
>
> compared to standard approaches for designing centrally managed collaborations between Web services.

The objects of the study are Macodo and a reference approach. Macodo is an approach that provides modeling abstractions for service collaborations and supports these abstractions throughout the development cycle. The reference

approach, a representative for the current state of practice, is defined as the following set of techniques and technologies:

- basic service-oriented architecture (SOA) design principles and architectures,
- standard Web services (i.e., WSDL and SOAP over HTTP),
- service orchestration (i.e., WS-BPEL),
- Web service repositories,
- standard process notations (i.e., WS-BPEL and BPMN).

The study takes place in the context of a master course on Web services and is based on an elaborate pilot study. The subjects of the study are 67 students of a Master in Software Engineering program from a university in Sweden, and two universities in Ukraine. The experiment itself is conducted as a block subject-object quasi-experiment. It consists of two experiment sessions, attended by all subjects. In each experiment session, subjects have to create a design of a system that supports a number of centrally managed collaborations between a set of Web services, using a specific approach.

All materials used in the study can be found in an online **experiment package**[1], including raw data and results.

Overview. The rest of the chapter discusses the study in detail, roughly following the outline proposed in [181] to report on empirical studies. We start by describing the experiment planning (Sect. 6.2). Next, we discuss the experiment design (Sect. 6.2.5) and execution (Sect. 6.3). Finally, we analyze the results (Sect. 6.4) and discuss our finding (Sect. 6.5), including potential threats to validity.

6.2 Experiment Planning

The goal of the experiment is defined in the introduction. The other three main tasks of planning the experiment are: selecting the subjects for the experiment (Sect. 6.2.2), creating the experimental materials (Sect. 6.2.3), and formulating the hypotheses for the experiment (Sect. 6.2.4). The last task includes defining the independent and dependent variables, and selecting proper metrics to measure the dependent variables. We start by discussing the lessons we learned from an exploratory pilot study.

[1] http://people.cs.kuleuven.be/~robrecht.haesevoets/macodo/

6.2.1 Pilot Study

The study presented in this chapter is based on an elaborate pilot study. The pilot study had two main goals: get feedback from developers that used Macodo, and explore opportunities for a controlled experiment.

The pilot study was performed with a group of 11 computer science master students, and took place as part of an advanced course on software architectures for distributed systems. During this course, students had to make two projects in teams of two, in which the focus was on documenting a software architecture using a set of architectural views. In one project, the architecture was created and documented using the reference approach and standard architectural views. In the other project the architecture was created and documented using the Macodo architectural views.

During the course and the study, we gained a lot of experience and received valuable feedback from the students. We learned the following main lessons:

- **Provide an explicit set of building blocks that the subjects can use.** In the pilot study, students came up with very diverse solutions and relied on different technologies that were hard to compare. As a result, it was unclear to what we were comparing (Macodo versus any other solution?). To avoid this problem, it is important to fully constrain the solution and design space and make choices or diversions in technology impossible.
- **Have students perform the experiment in a controlled environment (e.g., a supervised experiment session).** In the pilot study, students created their projects as a type of home assignment. To gather additional data, we asked students to keep a log book. Students, however, were sometimes careless in keeping the requested log book, making such data unreliable. Furthermore, there was unavoidable communication between the different teams related to the projects.
- **Make the assignments concrete and enable uniform measures.** The projects of the pilot study consisted of a set of functional and quality requirements for a particular system. Even with concrete requirements, however, the problem was still too open, resulting in students focussing on different aspects of the system. This made it hard to assess the correctness of solutions and define reliable, objective, and uniform measures.

6.2.2 Subjects

Due to limited resources[2], we use students of a Master in Software Engineering program as subjects for our study. Although these students do not represent expert software engineers, they are the next generation of software professionals and are relatively close to the population of interest [134].

[2]We rely on convenience sampling. That is, the nearest and most convenient persons are selected as subjects.

More specifically, the subjects are master students that follow a course on Web services. The course takes place in the Linnaeus University in Sweden, but students from Ukraine can also take this course. They follow the lectures using a live video stream. In total, the subjects consists of 37 students from the Linnaeus University in Sweden[3], and 30 students from 2 universities in Ukraine[4][5]. Since all students have a similar background and follow the same course, we consider them as a uniform group throughout the rest of the experiment.

6.2.3 Experimental Materials

For the experiment, we use two types of resources: an assignment and a debriefing questionnaire. All materials can be found in the online experiment package.

Assignments

An assignment is a small project in which subjects are asked to create a design for a system that supports a number of centrally managed collaborations between a set of external entities. Communication between the entities and the system is done using predefined Web services. The system can be seen as a platform that realizes service orchestrations according to a set of predefined collaboration types. Collaborations can be dynamically instantiated between different entities.

There are two different assignments (A and B). Having multiple assignments, allows to have subjects solve a different assignment in multiple experiment sessions (discussed in Sect. 6.2.5). Assignment A is based on an eHealth case in which different health care providers have to collaborate on different types of hospital floors. Assignment B involves a set of automated production lines in a manufacturing company, where different resources have to collaborate to manufacture a specific product type. Each type of hospital floor, and each type of production line, is a collaboration type to be supported.

Both assignments have a requirements document, which has the following elements: (1) a short problem description; (2) a set of functional scenarios to be supported, in terms of interactions between the external Web services; and (3) a set of predefined Web services (parnterLinkTypes) that allow the external entities to interact with the system and vice versa.

The assignments ask for two deliverables: (1) an architecture of the system in terms of modules, and (2) a detailed design of each module using a standard process notation. All deliverables have to be written down using pen and paper on seven provided answering sheets.

[3]Department of Computer Science - Linnaeus University (Campus Växjö), Sweden

[4]System Programming Department (Institute of Computer Systems) - Odessa National Polytechnic University, Ukraine

[5]Software technology department - Kharkiv National University of Radioelectronics, Ukraine

	Scenario 1	Scenario 2	Scenario 3
Interaction a	12	12	
Interaction b			5
Interaction c	12		
Interaction d		10	10
Total FP	24	22	15

Table 6.1: The adjusted function point count (FP) for assignment A and assignment B.

Assignment A and B have a similar level of complexity and require an equal amount of functionality. To measure the amount of functionality in each assignment we use the Albrecht's approach to calculate function points [9]. Function points are intended to measure the amount of functionality in a system described by its specification. It can also be used as a measure for software size [72]. The calculation of function points is based on the number of inputs and outputs of a system and additional complexity. A detailed discussion of how function points are calculated, can be found in Appendix D.1. The requirements of both assignments can be divided in three main scenarios (table 6.1). Each scenario consists of two interactions that have to be supported. Multiple scenarios can require the same interaction. In total, each assignment has 61 function points to be supported.

To constrain the assignments and their solution space, subjects are allowed to make a number of assumptions about the assignments and use simplified notations to create their deliverables.

Assumptions about the Assignments. To deal with the dynamic instantiation of predefined collaboration types, subjects can make the following assumptions specific to each treatment:

Reference Approach	Macodo
A predefined repository is available that contains all required data on collaborations (types, specifications, service endpoints). A predefined Web service can be used by the system to query these repositories.	Subjects only need to specify the different types of organizations. The instantiation of concrete organizations is outside the scope of the assignment.
External entities are assumed to have correct service endpoints for the system.	External entities are assumed to have correct service endpoints for behavior components and conversation connectors.

Simplified Notation. There are three notations used for the assignments. For the architecture of the system in terms of modules, specific notations are provided for the reference approach and Macodo. For the detailed design, the same simplified business process notation, based on BPMN, is used for both the reference approach and Macodo. This simplified process notation makes abstraction of specific correlation mechanisms and trivial 'assign' activities can be omitted. Examples of all notations can be found in Appendix D.2. A full specification of the notations can be found in the experiment package.

Debriefing Questionnaire

In addition to the assignments, we also use a questionnaire. The purpose of the questionnaire is to provide an alternative view on the results from the perspective of the subjects. The questionnaire is an online form, filled out by the subjects after each experiment session. The form consists of questions on the experience of subjects with the experiment and the treatment that was used. These questions cover the difficulty of using an approach, the confidence subjects have in the correctness of their design, and the intuitivity of the approach. All questions have a 5-level response scale. The complete questionnaires can be found in the experiment package.

6.2.4 Hypotheses and Variables

Based on the goals of the experiment, we can formulate a set of hypotheses. Each goal is mapped to a null hypothesis to be tested, and an alternative hypothesis to be accepted if the null hypothesis is rejected. Every hypothesis requires the definition of a set of independent and dependent variables, and a selection of proper metrics to measure the dependent variables. We also discuss how to handle faulty designs.

Hypotheses Formulation

We formulate four null hypotheses (H_0) and four alternative hypotheses (H_a):

- H_{01}: There is no difference in fault density between a design created using the reference approach and a design created using Macodo.

$$H_{01} : \mu_{fault.density_{Ref}} = \mu_{fault.density_{Mac}} \tag{6.1}$$

$$H_{a1} : \mu_{fault.density_{Ref}} > \mu_{fault.density_{Mac}} \tag{6.2}$$

- H_{02}: There is no difference in complexity between a design created using the reference approach and a design created using Macodo.

$$H_{02} : \mu_{complexity_{Ref}} = \mu_{complexity_{Mac}} \tag{6.3}$$

$$H_{a2} : \mu_{complexity_{Ref}} > \mu_{complexity_{Mac}} \tag{6.4}$$

- H_{03}: There is no difference in the level of reuse between a design created using the reference approach and a design created using Macodo.

$$H_{03} : \mu_{reuse_{Ref}} = \mu_{reuse_{Mac}} \tag{6.5}$$

$$H_{a3} : \mu_{reuse_{Ref}} < \mu_{reuse_{Mac}} \tag{6.6}$$

- H_{04}: There is no difference in design productivity between using the reference approach or using Macodo.

$$H_{04} : \mu_{productivity_{Ref}} = \mu_{productivity_{Mac}} \tag{6.7}$$

$$H_{a4} : \mu_{productivity_{Ref}} < \mu_{productivity_{Mac}} \tag{6.8}$$

Independent Variables

Independent variables are variables in the experiment process that can be manipulated and controlled. In our experiment, there are three independent variables:

- **Approach:** The approach used by a subject to solve an assignment. This variable is the factor of the experiment. That is, the independent variable that is changed to see the effect on the independent variables. This factor has two possible values, called treatments: the reference approach and Macodo. A specific value of a factor is also called a 'treatment'.
- **Subject:** The student solving the assignment.
- **Assignment:** The problem to be solved by the subject (assignment A or assignment B). Since both assignments have a similar level of complexity and require the same amount of functionality, the assignment is not considered as a factor but as a fixed variable. This decision is further supported by additional statistical tests (see Sect. 6.4.3).

Dependent Variables

Dependent variables are the variables that we want to study to see effect of different treatments (the reference approach or Macodo). For each hypothesis, we define the corresponding independent variable and select a proper metric to measure the effect.

Fault Density. Fault density is commonly defined as follows [72]:

$$fault\ density = \frac{\#\ known\ faults}{product\ size} \tag{6.9}$$

A fault is a mistake in a software product as the result of a human error[6]. A fault is only seen by the developer. Simply counting the number of faults, however, does not measure the impact of faults. A more precise metric is to measure the amount of change that is required to make the design work [72].

In our experiment, we measure faults in the detailed design. In the detailed design, subjects use the same notation for both treatments, allowing to define a uniform measure for changes. This is not possible for the architecture of the system. A detailed discussion of how changes are measured and the cost of changes can be found in Appendix D.3.

There are three common ways to measure the size of a software product: length, functionality, and complexity. These measures can be applied to specification, design, and code [72]. We measure size in terms of functionality. Every requirement (i.e., functional scenario) is divided into a set of function points (see Sect. 6.2.3). Size is then measured as the total number of function points that are supported by the design. This also allows us to use incomplete designs that do not cover all functionality. We can then calculate fault density as follows:

$$fault\ density = \frac{\#\ changes}{\#\ supported\ function\ points} \tag{6.10}$$

Design Complexity. We measure complexity in the detailed design, where modules are defined as processes. Two representative measures are: (1) activity complexity (AC) per function point, and (2) average control flow complexity (CFC) per module. Activity complexity (AC) is measured by counting the number activities in a module or process [33]. AC per function point measures the amount of 'code' needed to support a certain amount of functionality. Control flow complexity (CFC) also takes splits, joins, loops, and ending and starting points into account. We use an existing CFC metric that has some empirical validation [33, 34]. CFC per module measures the average complexity of a module.

$$AC\ per\ function\ point = \frac{\sum_{m\ \epsilon\ modules} AC(m)}{\#\ supported\ function\ points} \tag{6.11}$$

$$CFC\ per\ module = \frac{\sum_{m\ \epsilon\ modules} CFC(m)}{\#\ modules} \tag{6.12}$$

[6] A fault is different from a failure. A failure is the departure of a system from its required behavior. Faults can lead to failures. A failure reflects the user's view of the system.

Level of Reuse. The level of reuse in a software system is commonly measured as follows [82]:

$$level\ of\ reuse = \frac{lines\ of\ reused\ code\ in\ system}{total\ lines\ of\ code\ in\ system} \tag{6.13}$$

Instead of counting individual lines, reuse is typically measured per procedure or module. A procedure or module is considered reused only if it is used by more than one other procedure or module [82]. Given the context of our experiment, lines of code are mapped to a representative concept in the detailed design: the activity complexity (AC) per module [33]. The level of reuse can then be defined as:

$$level\ of\ reuse = \frac{\sum_{m\ \epsilon\ reused\ modules} AC(m)}{\sum_{m\ \epsilon\ modules} AC(m)} \tag{6.14}$$

To determine which modules are reused, we rely on the architecture of the system in terms of modules, which is part of the design.

Productivity. The productivity of a programmer or designer is typically measured as follows [72]:

$$productivity = \frac{\#\ implemented\ function\ points}{person\ months} \tag{6.15}$$

Since subjects create a design, we measure the function points supported by the design. We can then measure design productivity during the experiment as follows:

$$productivity = \frac{\#\ supported\ function\ points}{time\ spend\ on\ design} \tag{6.16}$$

Measuring in the Presence of Design Faults

Designs are likely to contain faults, which can make it hard to decide whether a function point is supported or not (e.g., forgetting one activity can break down an entire scenario). As a result, faults do no only effect fault density, but also complexity, reuse, and productivity. In addition, designs can contain modules or parts that are not used, or do not provide any required functionality (e.g., since assignments are created using pen and paper, subjects can include process definitions or left-overs that have no functionality). Taking these parts of a design into account when measuring will skew results for all types of metrics.

To avoid these problems, we use two counter-measures: (1) we trim designs of modules and parts that do not contribute to any required function point (these

parts are not used in any measurement), and (2) we correct each module in the trimmed designs before measuring complexity, reuse, and productivity. The concrete trimming and correction procedure is described in Sect. 6.4.1. The time required to correct a design is not considered in the productivity measure, due to the lack of a proper way to estimate this time.

To decide which function points are supported, it is sufficient for the reference approach, to look at the detailed design. For Macodo, however, the architecture contains essential elements to make the system work. Since faults are only measured in the detailed design (due to the lack of a uniform measurement for the architecture), we cannot just correct faults in the architecture of Macodo. Instead, when deciding which function points are supported, we take the architecture for Macodo 'as is'. More concretely, if a mistake is made in the architecture of Macodo, this part of the functionality is not supported, independent of the detailed design.

6.2.5 Experiment Design

There are two important aspects to the design of our experiment: the context of the experiment (i.e., external constraints), and the design of the experiment itself. We also provide a motivation for our design.

Context of the Experiment

The experiment takes place as part of a 9-week master course on Web services. The 9-week course is split into 2 parts of 5 and 4 weeks (Fig. 6.1). In part I, students are educated on the current state of practice for Web services, which fully covers the reference approach. In part II, students are educated on advanced techniques for Web services, which includes Macodo. Every part consists of one or two weeks of lectures, two weeks of home study, and one week for evaluation. In each part, students have to hand in an obligatory home assignment similar to the assignments used for the actual experiment. They also receive feedback and a model solution on these home assignments before the experiment sessions takes place. The home assignment is the same for both parts, but students have to use the reference approach in part I and Macodo in part II. In part I, students also have to do a practical assignment on WS-BPEL. The experiment itself takes place in the last week of each part. During these weeks, there is a 3 hour time slot, in which students first receive a short test on the course material (20-30 minutes) and then the assignment for the experiment. In the context of the course, students are graded on all tests and experiments. All course material can be found in the experiment package.

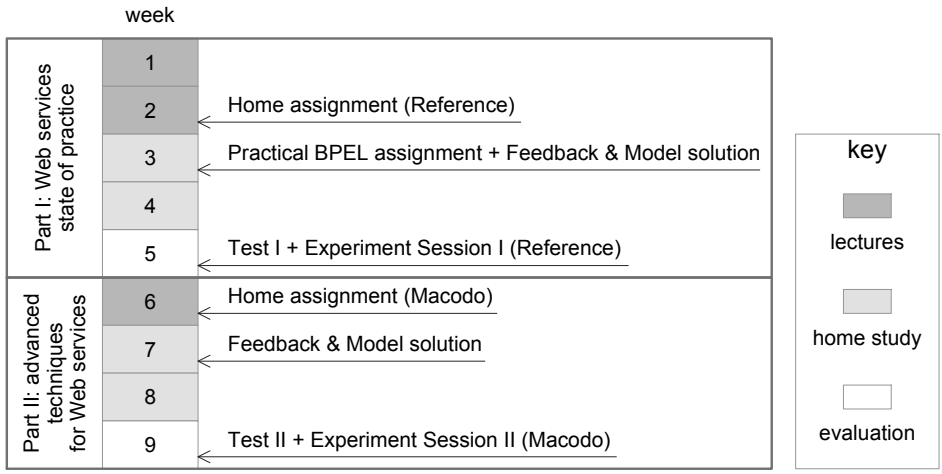


Figure 6.1: Overview of the course in which the experiment takes place.

Design of the Experiment

The experiment is conducted as a blocked subject-object quasi-experiment. Blocked subject-object means that each subject receives both treatments (the reference approach and Macodo). This allows paired comparison of samples. The experiment is a quasi-experiment [31] because it is performed on a single group and there is no randomization of the order in which the treatments are applied to the subjects. That is, all subjects first learn the reference approach and use it in experiment session I, and then learn Macodo and use it in experiment session II. This experiment design can be summarized as follows:

$$X_1\ O_1\ X_2\ O_2 \tag{6.17}$$

With X representing the exposure of the group to a treatment (learning the reference approach (X_1), or Macodo (X_2)), and O an observation (experiment session I (O_1), and experiment session II (O_2)).

The experiment uses a set of two assignments (see Sect. 6.2.3). In experiment session I, half of the subjects receives assignment A, the other half receives assignment B. This is done in a randomized fashion. In experiment session II, the assignments are switched: subjects who received assignment A in experiment session I now receive assignment B, and vice versa.

There are some additional aspects relevant to the experiment design:

- Students receive the same home assignment before each experiment session. This assignment is similar to the assignments used for the experiment sessions, and serves as a 'dry run'.
- Students are graded on both experiment sessions (with an equal portion of points), and are aware of this.
- Students are not aware of the experiment until after experiment session II.
- Students do not receive any feedback on the experiment sessions (including their grades) until after experiment session II.

Motivation.

Using the same order of treatments for all subjects introduces a number of potential threats to the internal validity of the experiment (discussed in Sect. 6.5.2). There are, however, a number of practical constraints to motivate this decision:

- The experiment takes place in the context of a course. Students have to learn both approaches (the reference approach and Macodo).
- Students follow lectures as a single group. It is practically infeasible to give some students different lectures than others.
- Macodo is an approach that builds on top of the reference approach. Knowing the reference approach is a prerequisite to use Macodo.
- In reality, developers would always learn and use the reference approach first and then Macodo.

Given these constraints, we can look at possible alternative experiment designs:

- **Completely Randomized Design.** Each subject is randomly assigned a single treatment.
- **Paired Comparison Design.** Each subject receives both treatments but in a random order.
- **4-Period Non-Randomized Crossover Design.** Similar to our design, but after experiment session II, there are two additional sessions, in which subjects first use the reference approach, and then again Macodo.

The completely randomized design and the paired comparison design, both yield a 'true' experiment. Both experiments, however, have to be conducted at the end of the 9-week course. This is because students cannot learn the reference approach or Macodo in separate groups. Conducting the experiment at the end of the course introduces two main problems:

- Subjects know Macodo when using the reference approach. This can pollute samples for the reference approach, when subjects try to emulate Macodo using the reference approach.
- Students are educated on the reference approach first, then on Macodo. This would favor Macodo, since it is the last approach they have learned.

For the completely randomized design, students would only be evaluated on one approach, which can be seen as unfair. The 4-period non-randomized crossover design would allow to reduce some threats to internal validity. This experiment, however, requires four experiment sessions, which is practically infeasible.

6.3 Execution

The experiment sessions take place during a normal course moment which all students attend. This is done in a classroom of the university, and under the supervision of one or more instructors. Subjects have to perform the experiment individually, using pen and paper, according to the following procedure:

1. Subject receives an assignment (A or B) and a corresponding requirements document. The assignment includes 7 blank pages to write down the design.
2. Upon receiving the assignment, the subject writes down the current time (starting time of assignment).
3. Subject creates and writes down a design for the assignment.
4. Before handing in the assignment, the subject writes down the current time (finishing time of assignment).
5. Subject hands in the assignment and requirements document.
6. Within 24 hours after the experiment, the subject fills out the online debriefing questionnaire.

In addition to this procedure, there are a number of constraints that apply during the experiment:

- Subjects have at most three hours to complete the assignment.
- Subjects work individually and cannot communicate with other subjects.
- Subjects have no access to the internet, course material or course notes.

6.4 Analysis

This section summarizes the collected data and describes results of the experiment devoid of any interpretation. The analysis consists of five parts: the collection of data (Sect. 6.4.1), the preparation of the data set (Sect. 6.4.2), the selection of proper statistical tests (Sect. 6.4.3), descriptive statistics for each dependent variable, and testing the corresponding hypotheses (Sect. 6.4.4 - 6.4.7). Section 6.4.3 also provides support for our decision to treat the assignment as a fixed variable. In Sect. 6.4.8, we briefly describe the responses to the questionnaires after each experiment session.

6.4.1 Data Collection

Every design is processed according to a predefined procedure (Fig. 6.2). The first step is to correct the design. This step is performed independently by two instructors[7]. The first step consists of three tasks: (a) determine the supported function points; (b) trim the design of any excessive parts that do not cover the supported function points; (c) correct the trimmed design.

Supported function points are based on the existing modules in the design (i.e., no additional modules have to be created to support the function point) and whether the solution is correctable to actually support the function point. Counting the number of supported function points is done per interaction that is supported (see Sect. 6.2.3), that is, the complete interaction is supported or not. Trimming the design excludes parts of the design that do not contribute to the actual functionality.

In step 2, the corrected designs are compared and merged. In step 3, the metrics for each dependent variable are applied to the corrected design.

6.4.2 Data Set Preparation

In total, 67 subjects participated in the first experiment session, and 62 subjects in the second experiment session (5 subjects did not show up). In addition, 2 subjects did not agree to have their data used, and 7 subjects provided unusable data in 1 or 2 experiment sessions (i.e., missing designs or designs that did not cover any functionality). Ten subjects did not provide usable time data and are excluded for the productivity measure. In total, we have the following samples, usable for paired comparison:

[7] One instructor is the author of this thesis. The other instructor is a teaching assistant of the course on Web services, who has experience in Web services, but had no knowledge of Macodo prior to the course.

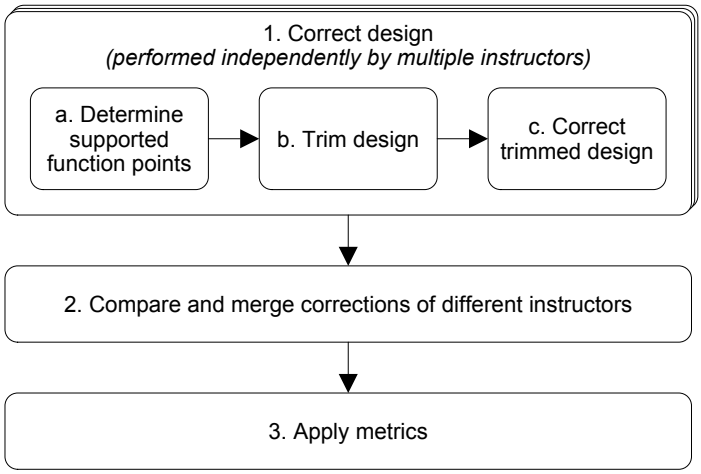


Figure 6.2: An overview of the data collection procedure.

	Experiment Session I	Experiment Session II
Assignment A	30 subjects	23 subjects
Assignment B	23 subjects	30 subjects

For the questionnaires, 54 subjects submitted the online form after each experiment sessions and agreed to have their data used.

6.4.3 Selection of Statistical Tests

For each dependent variable we do a descriptive analysis of the measurements for both treatments and the paired difference between the treatments. The paired difference is defined as follows:

$$Z_i = M_i - R_i \quad for\ i = 1, ..., n \tag{6.18}$$

M_i and R_i are the measurements of subject i for respectively Macodo and the reference approach; n is the number of subjects that produced usable data for both treatments.

To select a proper statistical test, we compare the distribution of Z_i for each dependent variable with the standard normal distribution, using the Anderson-Darling test [87]. This results in the following p-values:

	Anderson-Darling test (p-value)
Fault density	5.29×10^{-7}
AC per function point	0.0713
CFC per module	4.87×10^{-9}
Level of reuse	1.53×10^{-9}
productivity	0.431

With a significance level (α) of 0.05, we assume Z_i to be normally distributed only for the dependent variables 'AC per function point', and 'productivity'. Based on this assumption, we use the paired t-test [211] to test our hypotheses for 'AC per function point', and 'productivity', and the Wilcoxon signed-rank test [112] to test our hypotheses for the other dependent variables.

In Sect. 6.2.4, we decided to use the assignment (A or B) as a fixed variable. To support this decision, we compare the mean and distribution of each dependent variable for both assignments using the Wilcoxon signed-rank test [112] and the Kolmogorov-Smirnov Test [87]. The p-values for both tests are given below:

	Wilcoxon (p-value) (means)		Kolm.-Smir. (p-value) (distribution)	
	Reference	**Macodo**	**Reference**	**Macodo**
Fault density	0.700	0.899	0.719	0.596
AC per function point	0.907	0.632	0.519	0.710
CFC per module	0.328	0.700	0.462	0.999
Level of reuse	0.892	0.985	1.00	0.826
productivity	0.590	0.276	0.770	0.499

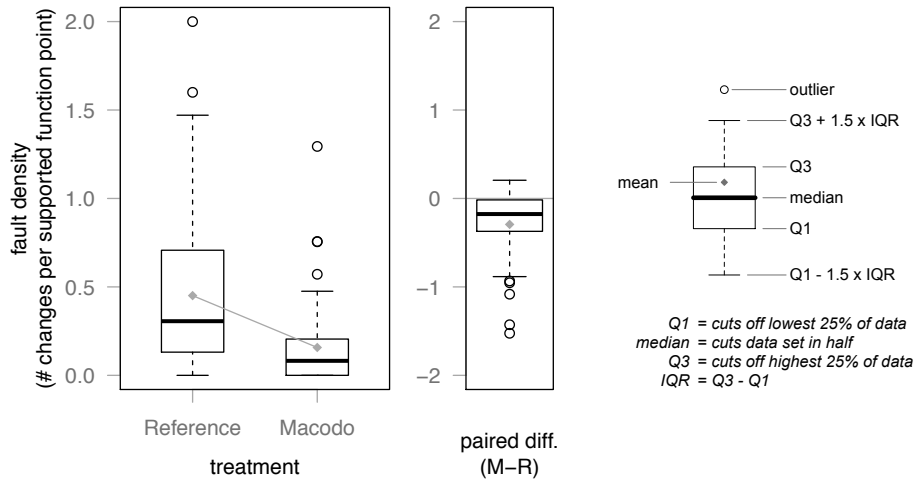
With a significance level (α) of 0.05, we assume that there is no difference in the mean and distribution of each dependent variable between both assignments.

6.4.4 Fault Density

Descriptive Analysis

The following table and boxplot (Fig. 6.3(b)) describe the measurements of fault density for both treatments, and the paired difference between the treatments. The table also provides the number of subjects that performed better, equal, or worse for Macodo:

	Fault density *(number of changes per supported function point)*					
	mean (μ)	median	st.dev	better	equal	worse
Reference	0.451	0.306	0.459			
Macodo	0.157	0.082	0.237	42	4	7
Paired Diff ($M_i - R_i$)	-0.293	-0.176	0.369			



(a) Fault density.

(b) Key for box plot.

Figure 6.3: Box plots for the fault density measurements.

Hypothesis Testing

We test the following H_0 with alternative H_a:

$$H_{01} : \mu_{fault.density_{Ref}} = \mu_{fault.density_{Mac}}$$

$$H_{a1} : \mu_{fault.density_{Ref}} > \mu_{fault.density_{Mac}}$$

The paired Wilcoxon signed-rank test results in the following p-value:

$$p\text{-}value = 2.80 \times 10^{-08} \tag{6.19}$$

With a significance (α) of 0.05, the null hypothesis is rejected.

6.4.5 Design Complexity

Descriptive Analysis

The following tables and box plots (Fig. 6.4) describe the measurements for the activity complexity (AC) per function point, and the average control flow complexity (CFC) per module:

	Activity complexity (AC) per function point					
	mean (μ)	median	st.dev	better	equal	worse
Reference	1.120	1.070	0.372			
Macodo	0.588	0.508	0.153	51	1	1
Paired Diff ($M_i - R_i$)	-0.531	-0.465	0.353			

	Average control flow complexity (CFC) per module					
	mean (μ)	median	st.dev	better	equal	worse
Reference	83.300	56.000	79.600			
Macodo	7.690	8.000	0.878	51	1	1
Paired Diff ($M_i - R_i$)	-75.600	-48.000	79.600			

Hypothesis Testing

We test the following H_0 with alternative H_a:

$$H_{02} : \mu_{complexity_{Ref}} = \mu_{complexity_{Mac}}$$

$$H_{a2} : \mu_{complexity_{Ref}} > \mu_{complexity_{Mac}}$$

The paired t-test for 'AC per function point' results in the following p-value:

$$p\text{-}value = 2.04 \times 10^{-15} \tag{6.20}$$

The paired Wilcoxon signed-rank test for 'CFC per module' results in the following p-value:

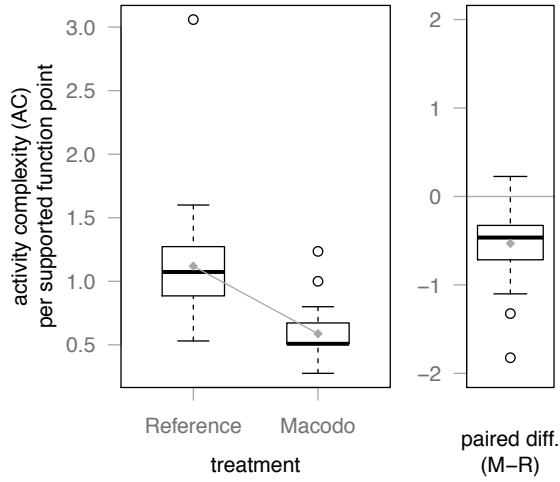
$$p\text{-}value = 1.97 \times 10^{-10} \tag{6.21}$$

With a significance (α) of 0.05, the null hypothesis is rejected for both types of complexity.

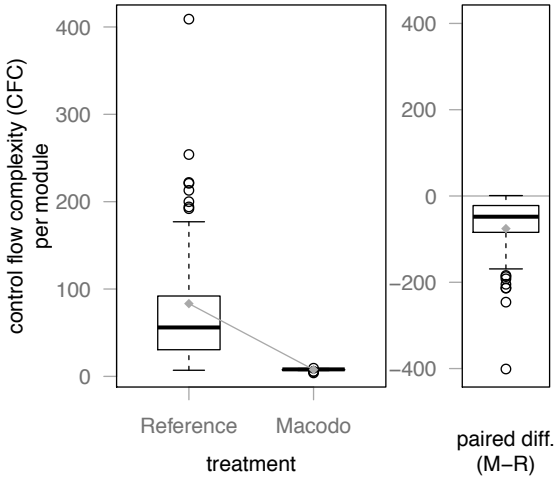
6.4.6 Level of Reuse

Descriptive Analysis

The following table and box plot (Fig. 6.5) describe the measurements for the level or reuse:



(a) Activity complexity (AC) per function point.



(b) Average control flow complexity (CFC) per module.

Figure 6.4: Box plots for the complexity measurements.

				Level of reuse (percentage)		
	mean (μ)	median	st.dev	better	equal	worse
Reference	1.84	0.00	9.77			
Macodo	68.90	87.10	27.20	51	2	0
Paired Diff ($M_i - R_i$)	67.00	87.10	27.60			

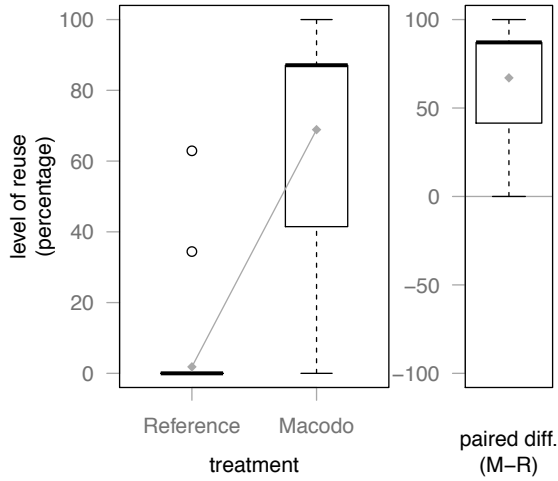


Figure 6.5: Box plots for the level of reuse measurements.

Hypothesis Testing

We test the following H_0 with alternative H_a:

$$H_{03} : \mu_{reuse_{Ref}} = \mu_{reuse_{Mac}}$$

$$H_{a3} : \mu_{reuse_{Ref}} < \mu_{reuse_{Mac}}$$

The paired Wilcoxon signed-rank test results in the following p-value:

$$p\text{-}value = 1.76 \times 10^{-10} \tag{6.22}$$

With a significance (α) of 0.05, the null hypothesis is rejected.

6.4.7 Productivity

Descriptive Analysis

The following table and box plot (Fig. 6.6) describe the measurements for productivity:

Productivity (number of supported function points per time unit)						
	mean (μ)	median	st.dev	better	equal	worse
Reference	0.289	0.300	0.134			
Macodo	0.542	0.530	0.229	39	0	4
Paired Diff ($M_i - R_i$)	0.253	0.236	0.184			

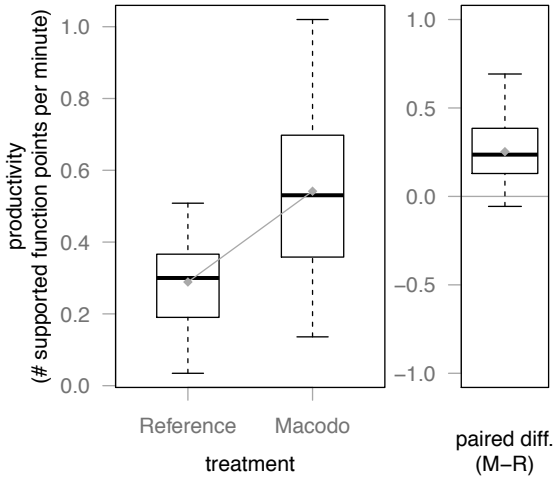


Figure 6.6: Box plots for the productivity measurements.

Hypothesis Testing

We test the following H_0 with alternative H_a:

$$H_{04} : \mu_{productivity_{Ref}} = \mu_{productivity_{Mac}}$$

$$H_{a4} : \mu_{productivity_{Ref}} < \mu_{productivity_{Mac}}$$

The paired t-test results in the following p-value:

$$p\text{-}value = 1.16 \times 10^{-11} \tag{6.23}$$

With a significance (α) of 0.05, the null hypothesis is rejected.

6.4.8 Debriefing Questionnaire

We discuss the results of four questions. Open questions and two questions with dubious formulation are not discussed[8]. Each question has a 5-level response scale:

Question 1: "How would you rate the difficulty to create a design for the exercise test using (SOA, Web services, and BPEL)/(Macodo)?"
Response Scale: very hard, hard, average, easy, very easy.

Question 2: "How would you rate your confidence in the correctness of your design for the exercise test?"
Response Scale: very poor, poor, average, good, very good.

Question 3: "Did you ever have the feeling that you did not know the real consequences of your design decisions?"
Response Scale: very often, often, sometimes, rarely, never.

Question 4: "How would you rate the overall intuitivity of (SOA, Web services and BPEL)/(Macodo) to model and design collaborations?"
Response Scale: very poor, poor, average, good, very good.

Subjects answered these questions after each experiment session. Figure 6.7 shows the improvement for Macodo on the 5-level response scale, when comparing the answers of subjects after the two sessions. For example, for 'design difficulty', 26 subjects gave an equal score for the reference approach and Macodo, 17 subjects gave a score that was 1 level higher, 6 subjects a score that was 2 levels higher, and 1 subject a score that was 3 levels higher for Macodo. Four students gave a score that was one level lower for Macodo.

6.5 Discussion

In this section, we interpret and discuss the findings of our analysis. We start by interpreting the results (Sect. 6.5.1). Next, we identify potential threats to validity (Sect. 6.5.2), and try to generalize our interpretations given these threats (Sect. 6.5.3). Finally, we discuss some lessons learned (Sect. 6.5.4).

6.5.1 Interpretation of Results

Based on the descriptive analysis, we can conclude there is a clear improvement for each dependent variable between experiment session I, in which subjects used the reference approach, and experiment session II, in which subjects used Macodo. This is confirmed by the statistical tests, which reject all four null hypotheses with a significance level (α) of 0.05.

[8] All questions and responses can be found in the experiment package.

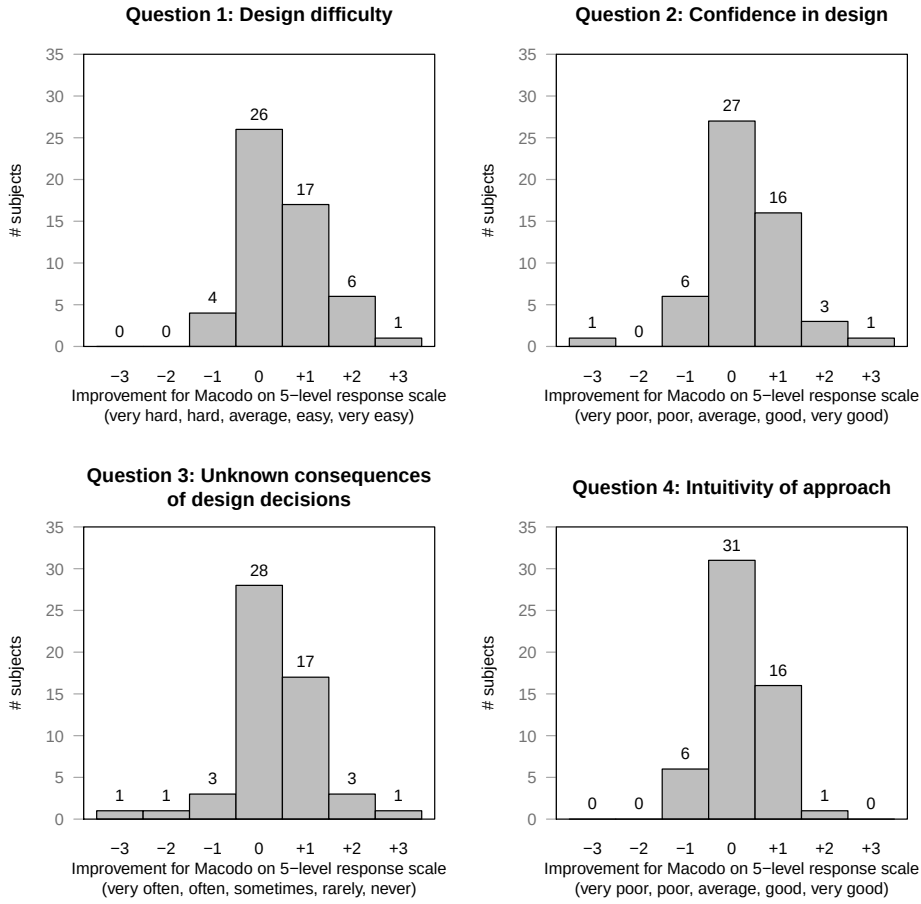


Figure 6.7: Improvement for Macodo on the 5-level response scale of each question.

Fault Density.

The average fault density (changes per supported function point) for the reference approach is 0.451, for Macodo it is 0.157, an improvement of 65%. Out of 53 subjects, 42 scored better for Macodo, and only 7 scored worse. This is reflected in a low p-value (2.80×10^{-08}).

Two possible explanations for this improvement are: (1) a better modularization of designs, and (2) reduced design complexity. On average, there are almost double the amount of modules per supported function point for Macodo (0.0812), compared to the reference approach (0.0476). Activity complexity per supported function point for Macodo is reduced by 50% compared to the reference approach, and control flow complexity per module is reduced by 91%. A better modularization and a reduction in complexity are likely to lead to less code and a smaller chance on faults.

The reduced fault density for Macodo is also reflected in the experience reported by subjects. Questionnaire responses show that 24 out of 54 subjects, find it easier to create a design with Macodo compared to reference approach, and only 4 find it harder. Similarly, 20 out of 54 have more confidence in the correctness of their design, compared to 7 which have less confidence.

Design Complexity.

The average activity complexity (AC) per function point is 1.120 for the reference approach, and 0.588 for Macodo (AC counts the number of activities). This is a reduction of almost 50%. For control flow complexity (CFC) per module there is a reduction of 91% (CFC takes splits, joins, and loops into account). For both types of complexity, 51 out of 53 subjects scored better with Macodo, and only 1 scored worse. Again, this is reflected in low p-values (2.04×10^{-15} and 1.97×10^{-10}).

Possible explanations for this improvement are: (1) a better modularization, (2) a better separation of concerns, and (3) the use of higher level abstractions. As mentioned before, the number of modules per supported function point is almost double for Macodo. In the reference approach, subjects have to deal with the management of collaborations at the lowest level (i.e., everything is expressed in a single process model). In contrast, Macodo provides abstractions that hide a number of implementation details, and allow to express the design in a more compact way.

It is also interesting to look at design complexity from the perspective of the subjects. Questionnaire responses show that 17 out of 54 subjects rated the Macodo approach more intuitive than the reference approach, and only 6 rated Macodo less intuitive than the reference approach.

Level of Reuse.

The average level of reuse for the reference approach is only 1.84%, while for Macodo this is 68.90%. In fact, for the reference approach, only two subjects have any form of reuse, while for Macodo only two subjects have no reuse. This is supported by a low p-value (1.76×10^{-10}). A possible explanation is that the reference approach does not stimulate reuse of sub-processes in typical service orchestrations. Macodo, however, provides abstractions that allow to modularize designs in terms of organizations, roles, conversations, and behaviors. Using the concept of capability, conversation connectors and behaviors components can easily be reused in different organization and role types.

Productivity.

The average productivity (supported function points per minute) for the reference approach is 0.289, for Macodo this is 0.542, an increase of 88%. 39 out of 43 subjects[9] increased their productivity with Macodo, only 4 had a lower productivity. This is reflected in a low p-value (1.16×10^{-11}).

Three possible explanations for this improvement are: (1) improved reuse, (2) reduced design complexity, and (3) the use of higher level abstractions. For the reference approach, there is almost no reuse, while for Macodo, the average reuse level is almost 70%. More reuse can easily lead to a higher productivity. A reduction in design complexity and the availability of higher level abstractions can further improve productivity, by allowing subjects to reason about the system and its functionality at a higher level of abstraction.

The higher productivity for Macodo is also confirmed by the questionnaire responses. Macodo scores better than the reference approach in terms of design difficulty, confidence in correctness, and intuitivity.

6.5.2 Threats to Validity

Given the design of the experiment (Sect. 6.2.5), a number of potential threats to validity [45, 211] can identified.

Threats to Construct Validity

Construct validity is the degree to which the operationalization of measures in the study actually represent the constructs in the real world.

[9] 10 subjects provided unreliable time data.

Evaluation Apprehension. Students might try to perform better with Macodo to impress, or favor the course holders. To reduce this threat, the experiment sessions are executed by external instructors. Students are not aware of the experiment, and Macodo is presented as an external approach. In addition, students are graded (and aware of this) on both experiment sessions for an equal amount of credits.

Mono-Operation Bias. The experiment relies on a set of two assignments. These assignments might not fully represent the type of problem on which we want to test both treatments. This threat could be reduced by using more than two assignments.

Mono-Method Bias. The measurements for fault density and complexity only consider faults and complexity in the design of each individual module. Faults and complexity at higher levels of complexity (i.e., the architecture) are not accounted for.

Experimenter Expectancies. The experimenter can transfer expectations to the participants in a way that affects performance in favor of Macodo. To reduce this threat, the main author of Macodo never has any direct contact with the subjects, and the actual experiment sessions are executed by external instructors.

Threats to Internal Validity

Internal validity is the extent to which independent variables (e.g., treatments) are actually responsible for the effects seen to the dependent variables.

History. The two experiment sessions take place on two different days, this difference can be a confounding factor in the results. To reduce this threat, both experiment sessions take place during a regular course moment on a monday morning.

Maturation, Learning Effects, and Testing. For all subjects, the effect of the first treatment (reference approach) is observed in experiment session I (week 5), and the effect of the second treatment (Macodo) is observed in experiment session II (week 9). This introduces a number of potential threats:

- Subjects can mature between the two observations. For example, subjects can take the course and the assignments more seriously and better prepare themselves.
- Subjects are tested twice, allowing them to learn the type of questioning and assignment, and the way the experiment is executed. They also have performed the home assignment twice.
- A subject's understanding of certain concepts can increase between the first and the second observation. In specific, subjects can become better at using the process notation (used for both treatments), and at reasoning in terms of processes.

To reduce these threats, a number of counter-measures have been taken:

- For each part of the course, we use the same home assignment, which is similar to the actual experiment. This serves as a 'dry run' and allows subjects to get acquainted with the way the assignments work. By using the same home assignment twice, subjects are less likely to get any additional insights related to the assignments between the two observations.
- Subjects have five weeks to prepare for the first experiment session and are aware that they will be graded on this.
- In the second part of the course, the focus is no longer on the content of the first part (e.g., process notation).

Despite these counter-measures there is still a potential impact on the actual results. More specifically, maturation, learning, and testing will likely have a positive effect on the reduction of fault density, the increase in productivity, and the reduction of complexity.

These threats could be reduced (or at least evened-out over both treatments) by randomizing the order of the treatments. As discussed in Sect. 6.2.5, this was practically infeasible and would have introduced other problems.

Threats to External Validity and Conclusion Validity

External validity is the degree to which findings of the study can be generalized to other participant populations and settings. Conclusion validity concerns generalizing the result of the experiment to the concept or theory behind the experiment.

Interaction of Selection and Treatment. Due to limited resources, we use students of a Master in Software Engineering program as subjects for our study. Although these students do not represent expert software engineers, they are the next generation of software professionals and are relatively close to the population of interest [134]. To further engage students in the experiment, students were graded on each experiment sessions.

Interaction of Setting and Treatment. The experiment relies on a set of two assignments. These assignments are created by the author of Macodo. As a result, there is a threat that the assignments are no realistic representation of the underlying problem domain, and favor Macodo over the reference approach. This threat could be reduced by introducing multiple assignments, prepared by external domain experts. Due to time and resource constraints, however, this was not possible for the experiment.

To scope the experiment, we also made a number of decisions out of practical necessity that should be taken into account when generalizing our findings:

- the use of pen and paper to write down designs;
- allowing subjects to make a number of assumptions about the systems, considered in the assignments, and their context;
- using a simplified notation for process definitions (the same notation is used for both treatments);
- the assignments use a set of predefined Web services to define the external entities.

Reliability of Measures. Measures are applied to corrected and trimmed designs. To increase the reliability and objectiveness of measures, designs are corrected and analyzed independently by two instructors. The corrected designs are then merged.

6.5.3 Inferences

In our interpretation of the results, we concluded that there is a clear improvement for each dependent variable over the two observations and we can reject each null hypothesis. We attributed these improvements and rejections mainly to the use of Macodo, compared to the use of the reference approach. Given the potential threats to validity, however, this can be a distorted view. We cannot rule out that maturation, testing, and learning between the two observations has a positive effect on the reduction of fault density, the increase in productivity, and the reduction of complexity. Nevertheless, given the explicit differences between both observations (i.e., an average improvement of 50% or more) and the counter-measures taken for these threats, it is unlikely that the measured effect is only caused by maturation, testing and learning.

We can also compare the effort required to learn and use Macodo to the effort required to learn and use the reference approach. Subjects had five weeks to learn the reference approach and four weeks to learn Macodo from scratch. Given the results, it is clear that Macodo can be learned in a reasonable amount of time, leading to a clear improvement in developer productivity. The questionnaire responses also show that the majority of the subjects (48 out of 54) find Macodo as intuitive or more intuitive than the reference approach.

In terms of generalizing our findings, it is important to take the potential threats to construct and external validity into account. We conclude that the results give a strong indication that Macodo, within the restrictions of the experiment, provides an improvement over the reference approach in terms of fault density, design complexity, level of reuse, and productivity.

6.5.4 Lessons Learned

Setting up and executing an experiment takes a lot of time and many things can go wrong. Given the practical constraints, we were not able to set up a 'true'

experiment. Nevertheless, performing an experiment is not only useful to do measurements and test hypotheses, it is also an excellent opportunity to focus research and get valuable feedback from reality.

Both the pilot study and the actual experiment, forced to make everything concrete and clear. Actual people had to use Macodo. The execution of the experiment provided additional feedback:

- People are creative. Even if a concept is well-defined, developers will make assumptions they cannot make. This was already clear in the pilot study, but still present, though less explicit, in the experiment. An interesting way to counter this, could be to provide tools that constrain developers and allow them to validate their designs.
- A number of students complaint that it was not always clear how to exactly use the abstractions provided by Macodo. More specifically, the concepts of conversation connector and behavior component allow to decompose collaborations in different ways. From the pilot study, we already noticed that students sometimes 'over-decomposed'. For example, some conversation connectors consisted of single messages. By providing additional guidelines in the documentation of Macodo, this improved in the controlled experiment. Additional methodological guidelines, however, could further improve this issue.
- Similarly, a minority of students did not reuse certain behavior components, although the behaviors were identical. This appears to be caused by a misunderstanding of the capability concept, which could be improved by additional guidelines.

6.6 Conclusions

The evaluation presented in this chapter addressed the following research questions: *"Do the modeling abstractions provided by Macodo: (1) reduce fault density; (2) reduce design complexity; (3) increase the level of reuse; and (4) increase productivity, compared to standard approaches for designing centrally managed collaborations between Web services?"*.

These research questions were translated to a set of four hypotheses and corresponding dependent variables. Given the practical constraints, the experiment was designed as a quasi-experiment. That is, all subjects first learned the reference approach and used it in a first experiment session and then learned Macodo and used it in a second experiment session.

From the descriptive analysis, we can conclude that there is a clear improvement for each dependent variable between experiment session I, in which subjects used the reference approach, and experiment session II, in which subjects used Macodo. In addition, the statistical tests show that each null hypothesis can be rejected.

Considering the potential threats to validity, however, we cannot fully attribute these improvements and rejections only to the difference in treatment. We cannot rule out that maturation, testing, and learning between the two observations has a positive effect on the reduction of fault density, the increase in productivity, and the reduction of complexity. Given the explicit differences between both observations, however, it is unlikely that the measured effect can only be attributed to maturation, testing, and learning.

We conclude that the results give a strong indication that Macodo, within the restrictions of the experiment, provides an improvement over the reference approach in terms of fault density, design complexity, level of reuse, and productivity.

Chapter 7

Related Work

Chapter 2 situated Macodo in several background domains. This chapter discusses the contributions of this thesis with respect to related work. The discussion focusses on the main topics of this thesis and its evaluation: the decomposition and modularization of complex collaborations, and the separation of concerns. More specifically this chapter focusses on existing organization models and infrastructures (Sect. 7.1), techniques to deal with variation in processes and collaborations (Sect. 7.2), and decomposition mechanisms for business processes and collaborations (Sect. 7.3).

7.1 Existing Organization Models and Infrastructures

Although the work presented in this thesis does not directly target multi-agent systems (MAS), Macodo relies on several related concepts. Therefore, it is interesting to compare (be it at a conceptual level) existing organization models to the Macodo model presented in Chapter 3. We also compare existing organization infrastructures to the proof of concept middleware presented in Chapter 5. We limit our discussion to a number of representative approaches.

7.1.1 Electronic Institutions

Electronic institutions (EIs) [70] are inspired by the notion of institutions in the human world, which set and enforce laws, monitor them, and respond to violations. EIs try to guarantee the overall behavior of open MAS to exhibit desired properties without compromising the agents' autonomy. They are an example of an organization-oriented MAS that uses regimentation to enforce norms and scenes to represent the functional aspects of an organization (see Sect. 2.3). Formally, an EI

can be defined as a set of roles (played by agents), a performative structure (defining the possible interactions), and a set of normative rules (defining the obligations of agents as a result of their actions). In addition, agents in an institution have to share a dialogic framework, which defines a common language and ontology.

The performative structure is defined as a set of scenes and the relations between these scenes. A scene (or scene protocol) defines a specific interaction protocol as a graph. Each node represents a state of the interaction protocol and arcs define transitions, triggered by speech acts of agents. Relations between scenes can define causal dependencies, synchronization, parallelism, or choice points. Role flow policies define how roles can go from one scene to another.

Normative rules can constrain the behavior of agents at two levels: intra-scene, or how to behave within a scene, and inter-scene, or how to go from one scene to another. Normative rules allow to define obligations across scenes (e.g., if an agent wins an auction in one scene, it is obliged to pay in another).

Two prominent works on electronic institutions are ISLANDER [69] and Ameli [71]. ISLANDER defines a specification language and editor for EIs, allowing to model and verify EIs. Ameli [71] provides middleware to execute EIs. Ameli is an example of a three-layered middleware for institution-centered MAS (see Sect. 2.3.2). It consists of an agent layer, a social or Ameli layer, and a communication layer. The social layer is agent-based, meaning that it is realized by four types of dedicated agents: institution manager (create EI), transition managers (control movement across scenes), scene managers (govern scene execution), and governors (mediate access between agent and infrastructure).

Electronic institutions have also been used in the domain of virtual enterprises, where the institution is used as a normative framework to validate and enforce virtual enterprise contracts, which define the cooperation agreements [32, 141]. The institution provides specific services such as brokering, reputation, negotiation mediation, and contract related services.

There are a number of similarities between Macodo and electronic institutions. Both take an organization-centered perspective on the system (at design level, as well as at implementation level), provide support using a three-layered middleware architecture, and rely on regimentation to constrain the behavior of agents[1]. One of the main difference between Macodo and electronic institutions lies in the scene-based interaction structure and the use of explicit normative rules. Scenes provide an abstraction for reusable interaction, but individual behavior is not well encapsulated and harder to reuse. Behavior of agents is partly defined by scenes, partly by scene transitions, and partly by normative rules. In addition, Macodo provides architectural views that allow to specify collaborations in terms of software elements, and reason about software qualities in terms of collaboration abstractions.

[1]Macodo does not rely on explicit norms, but required behavior is enforced (or regimented) by role components.

7.1.2 OperA

OperA (Organizations per Agents) [203, 60, 64] is a model and methodology for organizational interaction, and is supported by a modeling tool, called OperettA [63]. It allows to describe multi-agent systems at a conceptual level using three types of interrelated models: an organization model, a social model, and an interaction model.

The organization model describes a social system from the perspective of the organization. It defines an intended organizational structure and the laws that govern interaction among the agents. The social model describes a specific society, in terms role-enacting agents. A role-enacting agent has a social contract to play a specific role in a society. The interaction model describes the concrete interactions within a society.

OperA relies on four types of structures to define the organization model: social, interaction, normative, and communicative. The social structure describes the roles in a society, their objectives, and the dependencies between roles. The interaction structure describes the activities that are necessary to realize the overall objective of a society. The normative structure defines a set of norms that constrain the behavior of actors within a society. These norms can be used within the social and interaction structure. The communicative structure describes the communication primitives in terms of ontologies and communicative acts.

The interaction structure is similar to the performative structure in electronic institutions. It is defined as a set of scenes and relations between scenes. Compared to electronic institutions, where scenes represent formal protocols, scenes in OperA do not completely fix the protocol in advance, and only define a type of 'interaction skeleton' using the concept of landmarks. These 'skeletons' only specify some key parts ('landmarks') that need to be achieved in an interaction. Different concrete protocols can exist that realize a scene. This allows to maintain 'collaboration autonomy' [62]. Which concrete protocol is used, is modeled in the interaction model of OperA. Agents have to interpret the scene scripts and negotiate, to determine the concrete interaction protocols to use. To which interaction protocol they commit is then fixed in interaction contracts. In the normative structure, OperA allows to define three types of norms: role norms that describe expected behavior independent of what is expected in interactions, scene norms that describe expected behavior within a scene, and transition norms that constrain the flow between scenes.

In the context of the ALIVE research project[2], OperA has also been used in the domain of service-oriented architectures [59, 10]. The ALIVE approach combines model-driven development with coordination and organization mechanisms and aims to support reorganization and adaptation of services at design time and runtime. The approach relies on three levels of design and management: organization level, coordination level, and service level. The organization level provides a

[2] http://www.ist-alive.eu/

social context for service interactions through an explicit representation of an organizational structure (based on OperA). The coordination level defines possible interactions by specifying high-level interaction patterns between services. This level relies on 'agentified' services that are organization-aware and can reason about system objectives and generate and change workflows at runtime. The service level describes how interaction patterns can be realized using available services, and supports the semantic description and selection of services. The three levels of design and management are bound using model-driven engineering, providing automatic transformation of models from one level to another.

Unlike Macodo, the ALIVE approach relies on 'intelligent agent' that can reason about workflows and adapt them at runtime. Its focus on dynamic selection and discovery of services and the automatic generation of plans and workflows at runtime is outside the scope of Macodo.

7.1.3 Moise

Moise/Moise+ (Model of Organization for multI-agent SystEms) [107, 118, 116] is an organizational modeling language that decomposes the specification of an organization in three dimensions: a structural specification, a functional specification, and a set of norms.

The structural specification defines roles, groups of roles, and relations between roles. The functional specification defines how global organization goals should be achieved. This is done by specifying a set of schemes and a set of missions. Schemes are basically decomposition trees of goals. Global goals are decomposed in plans (to achieve these goals), which in turn are decomposed subgoals and so on. A mission is a set of coherent goals (taken from a scheme) which can then be assigned to concrete roles. But roles are not directly linked to missions. Moise uses norms to define the permissions and obligations of roles to execute specific missions.

A concrete instance of an organization is called an organizational entity. It corresponds to an organization specification and has a set of agents, groups and scheme instances. A scheme instance represents the commitment of agents to specific missions, and a set of achievements that map agents to the goals they have reached. In newer versions, Moise relies on a normative programming language (NOPL) as the basis for semantics and implementation [116]. Organization specifications and organizational entities are translated to this language and loaded into an organization platform, such as S-Moise+ [120].

Similar to Macodo, Moise describes organization dynamics as a set of possible organizational actions. There are four types of organizational actions: role adaption/leaving, groups becoming responsible for schemes, agents committing to or leaving missions, and agent achieving goals. Unlike Macodo, Moise uses the concept of goal as unit of decomposition. Global goals are decomposed in plans

and sub-goals, which are then assigned in the form of missions to agents, using the concept of norm. Moise does not provide any concepts to explicitly model behavior or interaction of roles as reusable units. Concrete behavior and interactions are the result of a set of complex norms that define which goals each role has to achieve. A norm can read like *'when an agent A: (1) is committed to a mission M that (2) includes a goal G, and (3) the mission's scheme is well-formed, and (4) the goal is feasible, then agent A is obliged to achieve the goal G before its deadline D'* [116].

S-Moise/S-Moise+[3] [119, 120] is a MAS implementation framework that follows the Moise model. It is also called the S-Moise+ organizational middleware. It has a three-layered middleware architecture consisting of an agent middleware (e.g., JASON), an organization layer, and an agent layer. The organization layer consists of an 'OrgManager' and one 'OrgBox' per agent. The OrgBox mediates the access of an agent to the organization layer and allows the agents to perform organizational actions (such as picking up a role, or committing to a mission). The OrgManager is a special agent that maintains the current state of the organizational entity.

7.1.4 TeamCore

TeamCore [172, 171] provides a platform for 'team-oriented programming'. Team-oriented programming creates a 'team program', which consists of three main artifacts: an organization hierarchy (defining a set of roles), a hierarchy of team plans, and the assignment of roles to team plans. Roles are described as a set of capabilities, similar to an interface description, describing the tasks an actor of a role can perform. A team plan expresses a joint activity as a set of 'team-level' actions. These actions often defined as abstract goals, assuming the agent or team playing a role know how to achieve this goal. The order (or sequence) of plans is defined by the hierarchy of plans.

Once a team program is defined, the specification can be loaded in the KARMA component of TeamCore. This component derives the requirements for roles (based on the capabilities), locates matching agents (e.g., using middle-agents), and assist developers in assigning agents (or groups of agents) to roles. When roles have been assigned, a set of TeamCore proxies is launched. These proxies interact with the agents, assigned to roles, and synchronize with each other using the TeamCore broadcast net and the KQML agent communication language. TeamCore proxies execute the team plans and perform actions required by the plans on their agents.

The dominant decomposition of TeamCore is a hierarchy of plans and tasks. As a result, TeamCore mainly provides abstractions to reason about a collaboration in terms of such plans and tasks. Interaction protocols are not explicitly modeled. A required interaction, becomes and abstract tasks to be realized by some role. In contrast, Macodo provides abstractions to reason about collaborations in terms

[3] http://moise.sourceforge.net/

of behaviors and interactions. Reuse within Macodo is supported at the level of roles, behaviors and conversations, while in TeamCore, it is only supported in terms of plans. In addition, Macodo uses the concept of capability, to support the encapsulation of agents, roles, behavior, and conversations. In TeamCore, plans are not well encapsulated, and capabilities are only used to represent the interfaces provides by agents and teams.

7.1.5 ROPE

ROPE (ROle-based Programming Environment) [18] is a role-based programming environment and architecture for developing MAS. It is based on the concepts of cooperation process and role. A cooperation process models an interaction as a set of phases (or states) and phase transitions, similar to scene-based approaches (Sect. 2.3.1). Cooperation processes are defined using a Petri net based language and can be executed on a ROPE engine. Each phase of a cooperation process defines a set of roles, and each role defines a small behavior to be performed in this phase. This behavior is defined in Java and resembles a method which can use and manipulate data available in the cooperation process (e.g., to trigger transitions), and use services provided by the agent playing the role. Phase transitions allow to move from one phase to another and define how agents assigned to a role in one phase have to be assigned to the roles of the new phase.

Compared to Macodo, ROPE decomposes a collaboration in terms of phases and roles in these phases. Although roles can be reused, they do not provide the same level of abstraction as roles, conversations, and behaviors in Macodo. A role in ROPE represents a small piece of behavior in a specific phase of an interaction, a role in Macodo defines a set of coherent complex behaviors, and conversations encapsulate reusable interactions. In addition, ROPE does not make a clear separation between the individual behavior of a role and the interactions between roles. Both have to be defined within role definitions, and the specification of an actual interaction is likely scattered over multiple role definitions.

7.1.6 BRAIN

BRAIN [25] is a role-based framework to develop interactions in MAS. It consists of XRole [24], an XML-based notation for roles, and RoleX [24, 23], an agent framework (or add-on to existing agent frameworks) that supports the execution of roles specified in XRole.

A role in XRole is defined by a name, a high-level description, a set of keywords (allowing to search for roles), a set of actions (or methods) available to the agent playing the role, and a set of events that the agent is expected to accept. The RoleX framework consists of a role descriptor repository, where agents can look for specific roles based on keywords, and a role implementation repository, which can

be used to find concrete role implementations. The RoleX framework also supports SOAP-based messaging between roles.

The definition of roles in XRole is very similar to a WSDL definition [43] and the delta with WSDL is not entirely clear. Compared to Macodo, the definition of behaviors in XRole is limited to a set of operations, and interactions are modeled in a rather indirect way (an interaction between roles occurs when one role performs an action and this action translates to an event that is accepted by an other role). Although the RoleX framework tries to be platform independent, it is conceived as a Java-based agent framework. Agents have to run java classes that implement roles, which can be dynamically loaded into the agent using a 'class-loader'.

7.2 Dealing with Process Variation

As illustrated in Sect. 2.7.3, an important problem in building and managing collaborations is to deal with variation. Variation is intrinsic to many application domains such as supply chain management, where a 4PL, for example, has to deal with different variations in supply chain networks, each requiring a similar, yet different solution. These solutions often implement a common reference process [77, 78] or well-known workflow patterns [201], but each implementation is slightly different due to the concrete context [77]. Such variation leads to multiple versions of business processes and workflows, which have to be properly designed and managed [106].

Traditional approaches and most BPM (Business Process Modeling) tools consider variants as separate process models, or variation is hidden or hard-wired in the control flow of business processes. This leads to redundancy, and as the number of variants grows, maintenance becomes a time-consuming and error-prone task [88].

To address this problem, several authors have proposed solutions where variants and adaptations to reference processes, are treated as first-class objects. These approaches define explicit change operations and adjustment points to configure process variants as a configurable workflow model [88]. A configurable workflow model adds a configuration layer on top of the workflow model. This configuration layer allows to alter predefined routes through the workflow during a configuration phase, and thus allows to instantiate specific variations of a workflow or business process. The configuration layer can be realized by creating different views on the process models [19], by adding and removing elements from processes [49, 170, 175, 88], or by parameterizing processes [126, 127]. The Provop approach [106], for example, configures processes by deleting an inserting process fragments, or moving specific activities.

Compared to Macodo, the above approaches do not focus on providing high-level abstractions to increase reuse. Instead they focus on making the difference and variation between processes first-class objects. In fact, most of these approaches

can be considered orthogonal to the work presented in this thesis. They provide generic mechanisms to handle variation, which could also be applied to create variants of behavior components and conversation connectors in Macodo.

7.3 Decomposition and Modularization of Business Processes and Workflows

A key feature of Macodo are abstractions that provide more natural decomposition units for dynamic collaborations. Several authors have proposed other mechanisms to decompose and modularize business processes, workflows, and business protocols. We focus our discussion on first-class support for sub-processes and sub-workflows (Sect. 7.3.1), the use of aspects (Sect. 7.3.2) and views (Sect. 7.3.3), and commitment-based approaches (Sect. 7.3.4).

7.3.1 First-Class Support for Sub-Processes and Sub-Workflows

The concept of sub-processes and sub-workflows can be used to decompose and modularize business processes and workflows. It can also be used to reuse specific fragments. In current Web service standards the concept of sub-process and sub-workflow has limited to no support. WS-BPEL [5] and WS-CDL [128] do not provide any support for decomposition. BPMN [6] supports decomposition of processes and choreographies in the form of sub-processes and sub-choreographies. Sub-processes and sub-choreographies can be used as a visual aid to manage complexity (i.e., collapsing certain parts of a diagram) or to define a reusable process or choreography definition.

There are several ongoing research efforts to support sub-processes and sub-workflows as first class-concepts, both at design, as well as at runtime. We look at techniques that rely on fragmentation of processes and workflows, and efforts to extend WS-BPEL with support for sub-processes.

Fragmentation of Processes and Workflows

Several authors present the idea of process or workflow 'fragmentation'. A fragment is a piece of process or workflow code similar to the concept of 'sub-process'. Fragments can be created from scratch or extracted from existing processes, and can be used to compose processes and reuse specific parts. Fragmentation can be done by hand or in a semi-automatic manner, in which a modeler is supported by a set of algorithms. A number of definitions for process fragments have been proposed.

Ma and Leymann [142] propose a formal definition of BPEL fragments. Like normal BPEL processes, fragments are specified in XML, using a new construct, named $< fragment >$, which inherits from $< process >$. The fragment construct allows to define behavioral context on variables, partnerLinks, and message exchanges. Fragments can be created from scratch or extracted from existing processes.

Eberle et al. [67] introduce a fragment definition to represent partial process knowledge. Based on the premise that no single person can have the entire process knowledge, they use fragments to define process building blocks. Fragments can be modeled by different persons, and are kept in a fragment repository. A fragment composer tool can implement a set of composition operations that allows fragments to be 'glued' together. This composition can be done at design time or at runtime. Actual tool support for these approaches is currently lacking.

Similar to the notion of process fragment, Adams et al. [7] present the concept of 'worklet' for workflow languages. A worklet is a small, self-contained, sub-workflow that handles a specific task or action in a larger, composite workflow. A top-level or parent workflow model captures the entire workflow at a macro level. Concrete worklets are dynamically selected from a 'repertoire' at run-time, based on the current context and a set of selection rules. Worklets have been implemented in YAWL (Yet Another Workflow Language) [198] as a custom service. The YAWL engine [200] (similar to a BPEL engine) uses this custom service to retrieve actual worklets when executing the parent process.

Compared to Macodo, the above approaches focus on providing generic mechanisms to decompose processes and workflows. The decomposition and modularization focusses on functionality, and does not provide support for underlying collaboration structure. They could, however, be combined with Macodo to enable the decomposition and modularization of more complex behavior components and conversation connectors.

Process fragmentation has also been used to execute processes in a decentralized manner [131, 130, 142]. Khalaf [131, 130] proposes a method for 'role-based' decomposition of business processes using BPEL. The main motivation for their approach is 'business process outsourcing' and 'mobile workforces'. In business process outsourcing, some parts of a private workflow are given to external parties for execution. In a mobile workforce, mobile entities can 'check out' part of a workflow and execute it. The proposed method allows to partition an existing BPEL process into several compliant BPEL processes or fragments, which can be enacted by each participant. The approach is based on modeling a process using swim-lanes. Each swim-lane represents a partner (e.g., an external party or a mobile entity) to which a fragment of the process is 'outsourced'. The authors propose an algorithm to translate a process with a swim-lane-based decomposition into a set of separate processes that communicate and synchronize using a set of 'wiring' messages.

Although the authors present their method as a 'role-based approach', the notion

of role is limited to a swim-lane (similar to BPMN). In addition, they are mainly concerned with the partitioning of existing processes, not with decomposition or reuse. In the end, the original specification is still a single processes, which is semi-automatically fragmented to enable distributed execution.

Efforts to Extend WS-BPEL

Reuse of fragments has also been studied in the context of WS-BPEL. There are a some proposals to extend WS-BPEL with native support for sub-processes [135, 196]. Kloppmann et al. [135] investigate possible options to extend WS-BPEL to support sub-processes. A sub-process is a fragment of BPEL code that can be reused within a process or across multiple processes. Two main options are considered: standalone sub-processes and inline sub-processes. Standalone sub-processes are sub-processes defined as normal BPEL processes. Interaction with these processes is like interacting with any other process. Such processes do not require any adaptation of the WS-BPEL standard or execution platforms. But they do not fully support the idea of a sub-processes. Standalone sub-processes can be compared to the way behavior components and conversation connectors are mapped to BPEL processes (Sect. 5.1).

Inline sub-processes can be defined as part of the definition of another process. Inline sub-processes can access data from its parent process directly (standalone sub-processes have to use normal service invocations). Inline sub-processes therefor provide a more natural realization of a true sub-process. Supporting inline sub-processes, however, requires adaptation to both the WS-BPEL standard and the execution platforms. Such extensions have not yet made their way into these infrastructures or the WS-BPEL standard.

7.3.2 Aspect-Based Approaches

Aspect-oriented programming (AOP) is a well known approach to support the separation of crosscutting concerns [133, 132]. Cross-cutting concerns are concerns that affect the implementation of multiple modules in a system. In AOP, cross-cutting concerns are typically modularized in aspect modules. These aspect modules can be weaved into existing code at specific 'join points'. The inserted code is called the 'advice'. A join point is a well-defined point in the execution of a program (e.g., the call of a method). Pointcuts can be used to define sets of joint points across different modules (i.e., cross-cutting) in a query-like manner. Several authors advocate an aspect-oriented approach for Web service composition [37, 46]. In the context of service composition, AOP can be used to extend process-oriented composition languages with aspect-oriented modularity mechanisms.

One of the most prominent aspect-based approach for service composition is AO4BPEL [37, 38]. AO4BPEL is an aspect-oriented workflow language for Web

services composition in which each BPEL activity is a possible join point. Pointcuts in AO4BPEL are used to refer to a set of join points, across several business processes, at which crosscutting functionality should be executed. Attributes of business processes and activities can be used as predicates to choose and define relevant join points. Similar to AspectJ [132], A04BPEL supports 'before', 'after', and 'around' advice.

Supporting AO4BPEL at runtime requires the extension of existing BPEL engines with an 'aspect runtime component' to make them 'aspect-aware'. A high-level architecture for such an extension is discussed in [38]. The authors of AO4BPEL have also proposed an aspect-oriented extension to the Business Process Modeling Notation (BPMN) in line with AO4BPEL [39].

Similar to AspectJ, AO4BPEL focusses on modularization at implementation level and does not provide high-level abstractions to structure the architecture of a system. AO4BPEL, however, provides an interesting alternative to the instrumentation used in the current prototype implementation of the Macodo middleware (Sect. 5.4.2). Container logic could be weaved into behavior components and conversation connectors as a set of aspects, instead of relying on instrumentation.

7.3.3 View-Based Approaches

View-based approaches rely on the concept of a process 'view' to provide a better decomposition and modularization of business processes. These approaches can focus on a single business processes or on cross-organizational collaborative workflows.

Tran et al. [194] propose a view-based and model-driven approach for developing service-oriented architectures, called the View-based Modeling Framework (VbMF). They rely on a set of extensible views to describe business processes. Three common views are *Flow View*, *Collaboration View* and *Information View*. The Flow View (or Orchestration View) describes a process at a high-level, making abstractions of process details (e.g., data exchange, service communication). The Collaboration View represents the interactions with other processes and services (i.e., the actual service operations). The details of these service operations are specified in the Information View, which also defines data types and messages. In addition to these views, other views can be added, such as human interaction, data access and integration [145], or traceability.

The model-driven part of VbMF consists of three main steps: view development, view integration and code generation. In view development, stakeholders create different views to describe specific business processes. View integration, which can be partially automated [195], uses model transformations to produce richer views (i.e., combination of views). These transformations can also be used to generate actual executable code (e.g., BPEL code).

Compared to Macodo, the VbMF does not directly decompose processes or collaborations. Instead, it decomposes the description of processes in a set of different views, each focussing on a specific concern. Although the original work [194] does not explicitly claim to improve reuse, more recent work does [195]. Reuse within VbMF is based on views making abstraction of certain concerns, resulting in generic and thus more reusable descriptions. As a result, reuse is mainly shown at the level of individual process elements (e.g., abstract activities) [195]. Macodo focuses on reuse of entire components (behaviors and conversations), which are well encapsulated.

Although VbMF explicitly provides a 'Collaboration View', this view is limited to showing individual interactions between different partners, by specifying the partnerLinkType and interaction protocol for each partnerLink of the process. Macodo, however, provides abstractions to decompose collaborations in terms of roles, conversations, and behaviors.

View-based techniques have also been used in the context of cross-organizational workflows [111, 41, 40]. Most of these approaches focus on integrating existing workflows of different organizations or enterprises. To do so, organizations expose a view on their local or private workflow. This allows authorized external parties to access and make use of only the related and relevant parts of the workflow. These workflow views can then be used to interconnect the different workflows and create a global cross-organizational workflow. A workflow view can thus be regarded as either a structurally correct subset of a workflow definition or a structurally correct composition of workflow definitions [111]. Compared to these approaches, Macodo focusses on decomposing collaborations from the start, instead of integrating existing workflows.

7.3.4 Commitment-Based Approaches

Commitment-based approaches [204, 214, 185] provide an alternative approach to traditional business process modeling. The main idea is to treat interactions at the level of 'business meaning', in contrast to existing approaches such as workflow management, which often focus at the level of messaging. This is done by capturing the business meaning of interactions among autonomous parties with commitments.

A commitment is a reification of a directed obligation [184] and can be compared to a social norm used in multi-agent systems [58]. An example of a commitment is $C(companyA, companyB, invoicePayed)$, meaning, company A is committed to company B for paying the invoice. Commitments can be manipulated, using operations such as create, release, assign, and delegate. Rules can describe how commitments are discharged. The theory of commitments is based in formal semantics [55].

Commitments can be used to specify business protocols [56]. A business protocol specifies an interaction between two or more roles as a set of message declarations

and logical axioms. These axioms define the effect of messages on commitments, but also constraints on data flow and event ordering. Messages are given business meaning by specifying how they affect commitments. The specification of a business protocol can then be translated into a transition system (similar to a state machine). Each state represents a specific state of the business protocol and corresponds to a set of commitments that hold in this state. Transitions between states are triggered by message exchanges and can manipulate commitments. Commitment-based business protocols can also be used to verify compliance of participants [204]. That is, check whether the behavior of an agent complies with the commitment protocol.

Amoeba [57] describes a methodology for using commitment-based protocols as building blocks to compose more complex business processes. Compositions are specified using a set of composition axioms. These axioms map roles of the composed protocol to roles of the individual protocols, specify how messages of one protocol effect the commitments in another protocol, and constrain the order of messages between different protocols.

Telang and Singh [192] also propose a meta-model to model business relations in terms of commitments. Business partners are represented as agents, and Business relations as a set of roles, played by agents. Each role corresponds to a set of commitments, expected of the agents who play the role, that refer to tasks or business activities. These tasks correspond to goals which are desired by agents. Using this simple model, a set of verification algorithms can be used to check the completeness and correctness of business models. Correctness means that no commitments are violated and completeness means that every task that is a goal of an agent is executed.

Commitment-based approaches are mainly concerned with the specification and validation of cross-organizational business processes. In contrast to Macodo, the goal is not to provide any direct runtime support, but rather specify an business protocol similar to service coordination and choreography. Although commitment-based approaches do bring more business meaning to the process definition, it still lacks higher-level abstractions. Instead of composing at the level of individual messages, composition is done at the level of individual commitments. Meta-models based on commitments [192] do introduce additional concepts, such as tasks and goals, but they still represent low-level concepts like individual business activities. While composition in Macodo is done using high-level building blocks, composition using commitment is done using low-level composition axioms. Commitments, however, do provide an interesting option to both formalize and validate the composition of role components, conversation connectors, and behavior components.

7.4 Conclusions

This chapter discussed the contributions of this thesis with respect to related work. When comparing Macodo to existing organization models and organization infrastructures, there are a number of similarities and differences. Similar to Macodo, several organization infrastructures follow a layered approach [209] (e.g., Ameli [71], S-Moise+ [120]). Ameli, S-Moise+, and TeamCore, also rely on the notion of an organization proxy, which enforces specific organizational constraints, and mediates the access of agents to the organization infrastructure (e.g., OrgBox in S-Moise+, governors in Ameli, and TeamCore proxies). These proxies can be compared to the organization manager and mediator components of the Macodo middleware architecture.

There are two main differences between Macodo and existing organization models. The first difference is that Macodo relies on a set of architectural views to describe organizations. These views directly represent collaborations in terms of software elements and allow to reason about software qualities in terms of collaboration abstractions. Many existing organization models describe organizations using multiple 'dimensions' or structures. These dimensions, however, often remain at a conceptual level, while it is unclear how they can be mapped to software. A second difference is the way functional aspects of an organization are represented and decomposed. Macodo relies on well-encapsulated concepts such as role components, conversation connectors, and behavior components. Most existing organization models follow a scene-based or goal-based approach. When using a goal-based approach, the dominant decomposition is a hierarchy of goals or the division of tasks. Interactions are not modeled explicitly, but are the results of one or multiple goals or tasks that make agents interact. Scene-based approaches can be compared to workflows, where the overall scene-structure is similar to a global workflow and individual scenes can be seen as sub-workflows that represent specific interactions. Behavior in scene-based approaches, however, is not well encapsulated.

Related work dealing with process variation (Sect. 7.2), and related work on decomposition and modularization of workflows and cross-organizational business processes (Sect. 7.3), mainly focusses on providing generic mechanisms, and typically do not provide explicit concepts to represent underlying collaboration structures. Since they focus on providing generic solutions, and are often orthogonal to the work presented in this thesis, they provide interesting opportunities to be combined with Macodo.

Work on dealing with process variation provides no high-level abstractions to increase reuse, but focusses on making variation itself a first-class object. This could be combined with Macodo to handle variation in conversation connectors and behavior components.

Approaches to support sub-processes and fragments as first-class concepts aim for a functional decomposition, but do not provide any reification of underlying collaboration structures. They could be used in Macodo to further decompose

complex conversation connectors and behavior components.

Aspect-based approaches for business processes mainly target 'implementation-level' modularity, but provide no high-level abstractions to structure the architecture of a system. These approaches do, however, provide an interesting alternative to instrumentation used in the current prototype implementation of the Macodo middleware.

View-Based approaches improve reuse by keeping concrete details in separate views, and thus making more abstract views more reusable, while Macodo focusses on providing abstractions that fully encapsulate modular parts of collaborations.

Commitment-based approaches are mainly concerned with the specification and validation of cross-organizational collaborations and business processes. Supporting collaborations at runtime or reifying them in terms of software elements is outside the scope of these works. Commitment-based approaches do, however, provide an interesting option to both formalize and validate the composition of role components, conversation connectors, and behavior components.

Chapter 8

Conclusions

Flexible integration and collaboration of information systems, both within and across company borders, has become essential to success in current business environments. Realizing collaborations and building the supporting information systems, however, poses huge engineering challenges, from architectural design to actual implementation. Current state of practice relies on middleware, workflow management, service-oriented architecture (SOA), and Web services to address these challenges.

In this thesis, we argued that several engineering challenges are still insufficiently addressed. We defined three problem statements and illustrated them in a supply chain management case:

- the lack of proper decomposition mechanisms;
- the focus of current solutions on functional decomposition;
- the missing reification of collaboration abstractions throughout the development cycle.

These problem statements lead to the following research questions:

> (1) *What are good abstractions to support and promote modularization of service collaborations, in order to reduce the number of faults, better manage complexity, and improve reuse and productivity?*
>
> (2) *How can these abstractions be reified and supported throughout the development cycle (i.e., architecture, design, and implementation)?*

To address these research questions, we presented Macodo, an approach that consists of three complementary parts: (1) a set of abstractions for dynamic service

collaborations, (2) a set of architectural views that reify these abstractions, allowing to build and document collaborations in terms of software elements, and (3) a middleware infrastructure that supports these abstractions at implementation level.

Macodo focusses on service collaborations that can be defined as follows: *"The controlled interchange of information between a set of distributed entities (e.g., Web services) and the controlled execution of related tasks by these entities in order to achieve a set of goals."*. In addition, these collaborations take place in a restricted environment, which implies that there is a central owner or maintainer, participation is controlled, and participants have a pre-established trust in the environment and its maintainer.

8.1 Contributions

This thesis presented three main contributions:

- **A conceptual model for dynamic collaborations [207, 101, 103].** The conceptual model describes a set of collaboration abstractions that define the vocabulary for Macodo. The model consolidates earlier research results and is based on role-based modeling techniques from several domains. The core abstractions (organization, role, conversation, and behavior) allow to modularize complex collaborations and represent interactions, individual responsibilities, behavior of participants, and management of collaborations as separate concerns. A set of additional concepts (role state, organization dynamics, and capability) allow to describe the adaptability of collaborations, and to improve their reuse and modularity [99, 102, 206].
- **A set of architectural views to build and document collaborations in terms of software elements [208].** The architectural views introduce architectural modeling abstractions that reify the collaboration abstractions of the conceptual model at architectural level. Each view addresses some of the main architectural drivers derived from the problem statements of this thesis:
  - express and reason about collaboration functionality and qualities in terms of collaboration abstractions;
  - separate concerns such as participant responsibilities, individual behavior, and interactions;
  - decompose collaborations to improve reuse and modularity;
  - express and reason about modifiability and runtime adaptation of collaboration;
  - describe collaborations at different levels of abstraction, while maintaining conceptual integrity.

- **A proof of concept middleware infrastructure supporting collaboration abstractions at implementation level [208, 104, 100, 98].** The proof of concept middleware infrastructure provides a concrete platform to develop and implement collaborations that are designed in the architectural views. Each type of Macodo module is mapped to concrete Web service technology, and can be implemented using XML, WSDL and WS-BPEL. A management service provided by the Macodo middleware allows applications to register actors and to manage the life-cycle of organization connectors and role components. A proof of concept middleware architecture and prototype implementation show that Macodo can be integrated in the current technology stack, without the need for new standards.

The first two contributions, the conceptual model and architectural views, were evaluated in a controlled experiment, performed with computer science master students. In this experiment, we provided an answer to four research questions: *"Do the modeling abstractions provided by Macodo: (1) reduce fault density; (2) reduce design complexity; (3) increase the level of reuse; and (4) increase productivity, compared to standard approaches for designing centrally managed collaborations between Web services?"*.

Given the practical constraints, the experiment was designed as a quasi-experiment. Taking potential threats to validity into account, the results give a strong indication that, within the restrictions of the experiment, Macodo provides an improvement over the reference approach in terms of fault density, design complexity, level of reuse, and productivity.

8.2 Future Work

In this section, we discuss some interesting opportunities for future work.

Formalization of the Conceptual Model

We have formalized previous versions of the Macodo model [207, 103, 101] using the Z notation [189] and Alloy [123]. Such formalization allows to precisely define the scope and meaning of concepts. It can also be used as a basis for future implementations and verification. An interesting option could be to combine Macodo with existing work on formal commitments in the context of business protocols [204, 214, 185]. This could provide a formal basis to validate the composition of role components, conversation connectors, and behavior components.

A Domain-Specific Extension for Macodo in an Existing ADL

A promising direction is to support the Macodo architectural views as a domain-specific extension to an existing architecture description language (ADL). An ADL is a language to specify an architecture in a formal way that is both human and machine readable. Two prominent examples of formal ADLs are ACME[1] and Pi-ADL [157]. Integrating Macodo with an existing ADL can provide a basis for graphical tool support, automated analysis and verification, and automatic code generation. Graphical tools can support the graphical notation presented in this thesis. Automated analysis and verification can be used to check compositions of conversation connectors and behavior components, and the compatibility of capabilities. Automatic code generation can translate Macodo architectural view descriptions to Macodo modules that can be directly loaded into the Macodo middleware infrastructure.

Extending Macodo with Complementary Decomposition Mechanisms.

Macodo provides organization and role abstractions to support the decomposition of complex collaborations. There are a number of promising decomposition mechanisms for workflows and business processes that can be considered orthogonal to Macodo, and which could provide a valuable addition. Two interesting tracks are dealing with variation in Macodo modules, and supporting sub-conversations and sub-behaviors to further decompose complex conversations and behaviors. To handle variation in Macodo modules, and to increase reuse of Macodo modules in different contexts, Macodo could be extended with mechanisms that support parameterization of workflows and business processes [126, 127], or dynamic configuration at runtime [49, 88, 106]. To support sub-conversations and sub-behaviors, possible options are using BPMN's support for sub-processes [6], or work on process fragments [7, 67].

Building a Mature Middleware Infrastructure

The middleware infrastructure presented in this thesis is intended as a proof of concept for Macodo. To further valorize Macodo, an essential step is to build a mature middleware infrastructure that fully supports Macodo and its features. In addition, there are a number of interesting directions to explore:

- Supporting conversation connectors and behavior components that are not BPEL-based. Examples are conversation connectors that work as shared data repositories, or Java-based behaviors.

[1] http://www.cs.cmu.edu/~acme/

- Supporting the dynamic addition of new Macodo modules at runtime. This allows introduce new types of collaborations at runtime, which can be essential in certain domains.
- Exploring alternative middleware architectures and implementations. Interesting options are the use of service routers [148, 144] instead of instrumentation, using aspect-based techniques (e.g., AO4BPEL [37]) to weave in container logic, or using a framework that supports the execution of native BPMN 2.0 definitions, such as jBPM[2] or Bonita Open Solution[3].

8.3 Closing Reflection

The underlying theme of this thesis is that abstraction is essential to software engineering. That is, abstraction in the sense of concepts, but also in the way a system is built and documented.

Many domains have developed useful abstractions and ways to cultivate abstractions. Other domains provide useful technologies, but have a need for better abstractions or are missing the reification of abstractions throughout the development cycle. Cross-fertilization between these domains can lead to new insights and progress. Macodo can be seen as such an attempt.

Macodo is the result of a long journey, in which many choices were made. Choices that were essential to scope and move forward. No research track is the result of a single person, and Macodo is no exception. It is influenced by many people, projects, and valuable collaborations with researchers and students.

Another major influence on Macodo has been the empirical evaluation. Performing such an evaluation is not only a great way to evaluate, it also provides invaluable feedback, and forces to scope. Empirical evaluation within software engineering is gaining prominence, and this can only be applauded.

[2] http://www.jboss.org/jbpm
[3] http://www.bonitasoft.com/

Appendix A

Macodo View Documentation Example

This appendix provides an extended example of a Macodo view documentation. The complete documentation of the architecture used by the 4PL could include the following views:

- Organization Module Views
  - Vmi Organization Module View
  - Cmi Organization Module View
- Organization & Actor Views
  - Carrefour Organization & Actor View
  - Delhaize Organization & Actor View
  - Acme Organization & Actor View
- Role & Conversation Views
  - Vmi Role & Conversation View
  - Cmi Role & Conversation View
- Allocation Views
  - Deployment View
  - Install View
  - Implementation View

For each organization type there is a specific Organization Module View and Role & Conversation View. For each specific supply chain network that the 4PL has to support, there is an additional Organization & Actor View. A set of allocation views (described using standard views) documents the deployment, installation, and implementation plan of the architecture.

The rest of this appendix focusses on a specific view: the Vmi Role & Conversation View. This view documents the runtime architecture of the *Vmi Organization* Connector. To document this view, we follow the view template proposed by [84]. Some parts of the documentation have been simplified or omitted.

A.1 Primary Presentation

The primary presentation of the Vmi Role & Conversation View is split in two parts (Fig. A.2 and A.2).

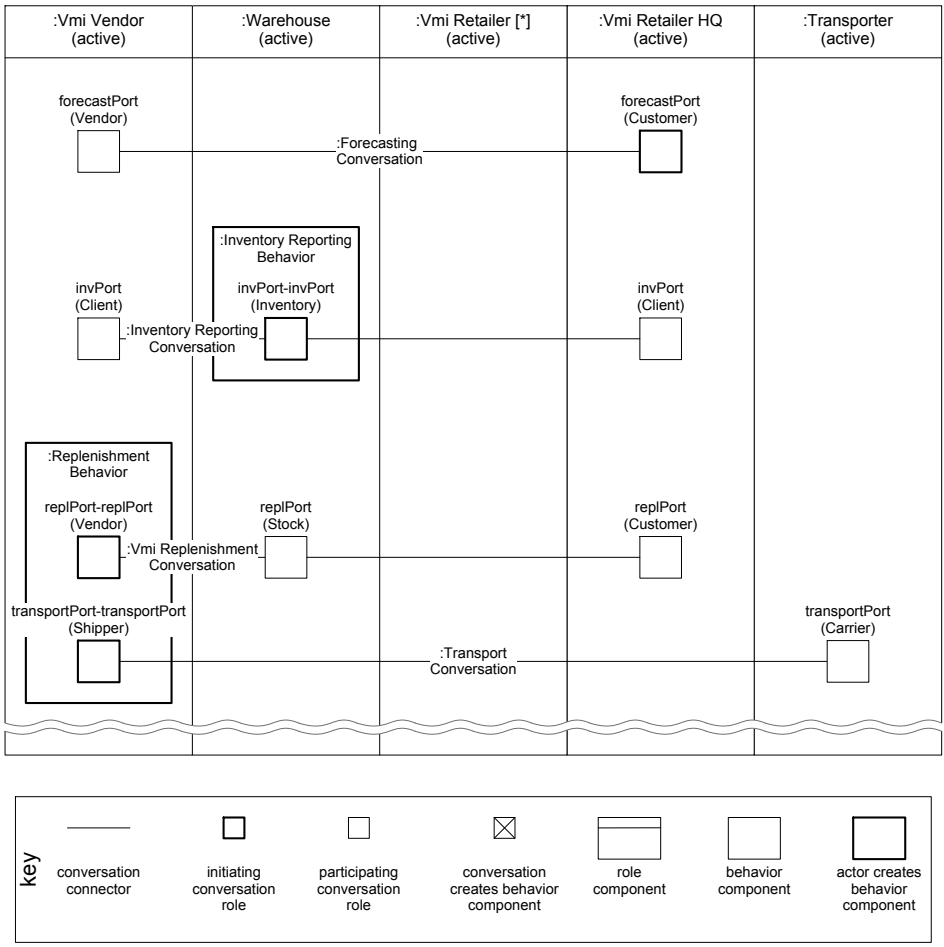


Figure A.1: Primary presentation of the Vmi Role & Conversation View part I.

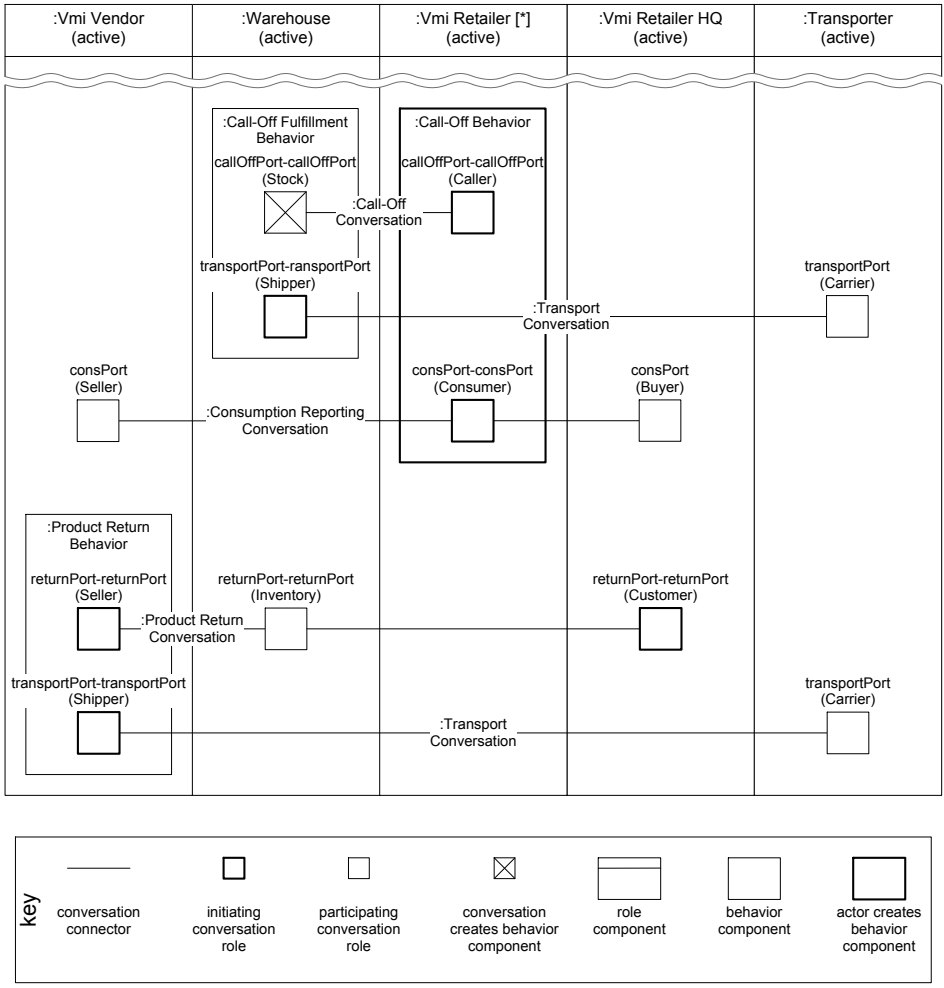


Figure A.2: Primary presentation of the Vmi Role & Conversation View part II.

A.2 Element Catalog

This section describes the different elements depicted in the primary view.

A.2.1 Role Components

Vmi Vendor Role

This role component represents the vendor role of the Vmi Organization. It encapsulates the responsibility of the vendor related to forecasting, inventory reporting, replenishment, consumption reporting, and product return. This component has the following ports:

- Actor Port: requires *Vmi Vendor Capability*
- Conversation Ports:
  - forecastPort: provides *Forecasting-Vendor Capability*
  - inventoryPort: provides *InventoryReporting-Client Capability*
  - replPort: provides *Replenishment-Vendor Capability*
  - transportPort: provides *Transport-Shipper Capability*
  - consPort: provides *ConsumptionReporting-Seller Capability*
  - returnPort: provides *ProductReturn-Seller Capability*

Delegation of conversation ports to the actor port and behavior components is defined in the primary view.

Warehouse Role

omitted from example

Vmi Retailer Role

omitted from example

Vmi Retailer HQ Role

omitted from example

Transporter Role

omitted from example

A.2.2 Conversation Connectors

Forecasting Conversation

omitted from example

Inventory Reporting Conversation

omitted from example

Replenishment Conversation

omitted from example

Call-Off Conversation

This conversation connector encapsulates the call-off interaction. It allows a caller to place a call-off order with a stock. The stock, in turn, will provide the necessary shipment info. The conversation connector is executed as a persistent BPEL process, and has two connector roles:

- Caller: requires *CallOff-Caller Capability*
- Stock: requires *CallOff-Stock Capability*

All conversation capabilities have asynchronous SOAP-based interfaces. The conversation connector is created by the Caller conversation role.

Transport Conversation

This conversation connector encapsulates the transport ordering interaction. It allows a shipper to place a transport order with a carrier. This conversation handles both the exchange of shipment info and delivery receipts. The conversation connector is executed as a persistent BPEL process, and has two connector roles:

- Shipper: requires *Transport-Shipper Capability*

- Carrier: requires *Transport-Carrier Capability*

All conversation capabilities have asynchronous SOAP-based interfaces. The conversation connector is created by the Shipper conversation role.

Consumption Reporting Conversation

omitted from example

Product Return Conversation

omitted from example

A.2.3 Behavior Components

Inventory Reporting Behavior

omitted from example

Replenishment Behavior

omitted from example

Call-Off Behavior

omitted from example

Call-Off Fulfillment Behavior

This behavior component encapsulates the role functionality to fulfill a call-off order. It is responsible for initiating a transport conversation after receiving a call-off order. The behavior component is executed as a persistent BPEL process, and has the following ports:

- Actor Port: requires *Call-Off Fulfillment Capability*
- Conversation Ports:
  - callOffPort: provides *CallOff-Stock Capability*
  - transportPort: provides *Transport-Shipper Capability*

Both the behavior capability (required by the actor port) and the conversation capabilities (provided by the conversation ports) have asynchronous SOAP-based interfaces. The behavior component is instantiated by a conversation connector, through the *callOffPort*.

Product Return Behavior

omitted from example

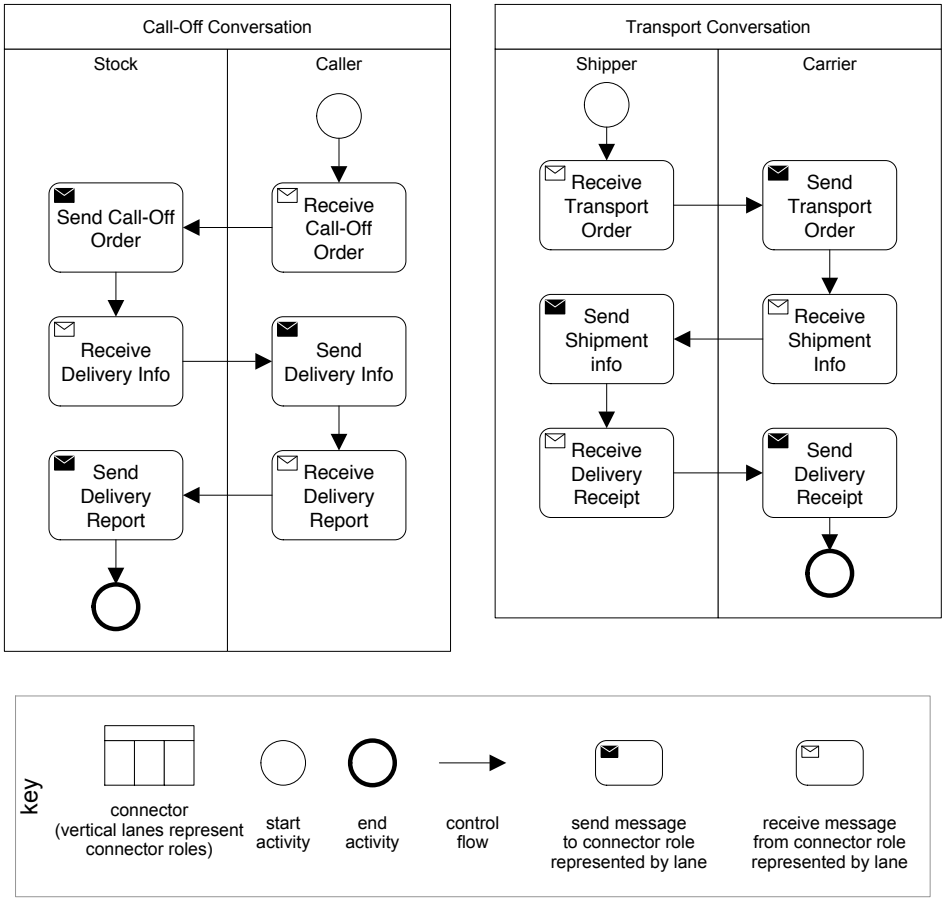
A.2.4 Capabilities and Element Interfaces

Capabilities and interfaces are described in Appendix B.

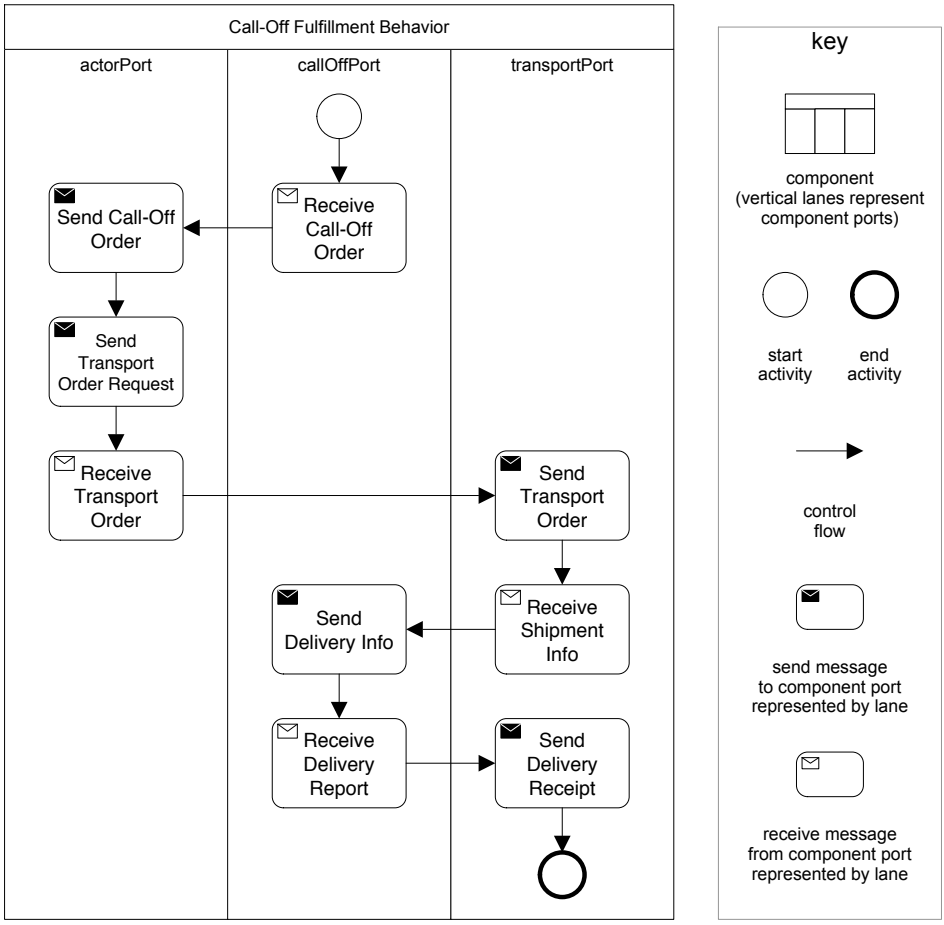
A.2.5 Element Behavior

the example only shows the behaviors of some conversation connectors and behavior components

Call-Off Conversation and Transport Conversation



Call-Off Fulfillment Behavior



A.3 Context Diagram

N/A

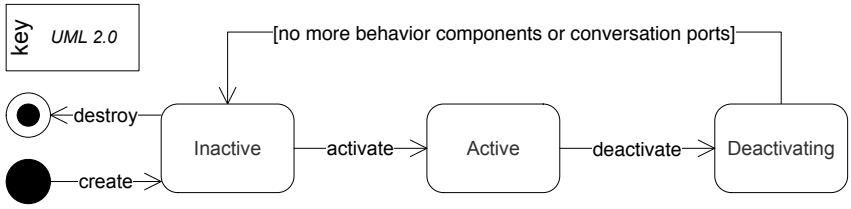
A.4 Variability Guide

A.4.1 Possible Role States

the example is limited to the state of two role components

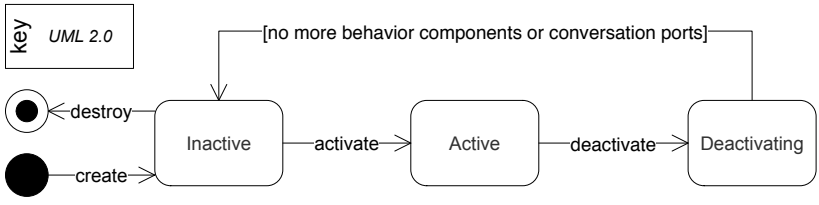
Warehouse Role

		role state					
		Active		Deactivating		Inactive	
		creation	max occur.	creation	max occur.	creation	max occur.
conversation ports	invPort	yes	*	no	*	no	0
	repPort	yes	*	no	*	no	0
	callOffPort	yes	*	no	*	no	0
	transportPort	yes	*	no	*	no	0
	returnPort	yes	*	no	*	no	0
behaviors	Inventory Reporting Behavior	actor	*	no	*	no	0
	Call-Off Fulfillment Behavior	conversation	*	no	*	no	0



Vmi Retailer Role

		role state					
		Active		Deactivating		Inactive	
		creation	max occur.	creation	max occur.	creation	max occur.
conversation ports	callOffPort	yes	*	no	*	no	0
	consPort	yes	*	no	*	no	0
behaviors	Call-Off Behavior	actor	*	no	*	no	0



A.4.2 Possible Conversation Connectors

The following table lists the possible conversation connectors in the Vmi Organization.

Forecasting Conversation
- *Vendor → VmiVendor.forecastPort*
- *Customer → VmiRetailerHQ.forecastPort*

Inventory Reporting Conversation
- *Inventory → Warehouse.invPort*
- *Client → VmiVendor.invPort*
- *Client → VmiRetailerHQ.invPort*

Replenishment Conversation
- *Vendor → VmiVendor.replPort*
- *Stock → Warehouse.replPort*
- *Customer → VmiRetailerHQ.replPort*

Transport Conversation
- *Shipper → VmiVendor.tranportPort*
- *Carrier → Transporter.transportPort*

Call-Off Conversation
- *Caller → VmiRetailer.callOffPort*
- *Stock → Warehouse.callOffPort*

Transport Conversation
- *Shipper → Warehouse.transportPort*
- *Carrier → Transporter.transportPort*

Consumption Reporting Conversation
- *Consumer → VmiRetailer.consPort*
- *Buyer → VmiRetailerHQ.consPort*
- *Seller → VmiVendor.consPort*

Product Return Conversation
- *Seller → VmiVendor.returnPort*
- *Inventory → Warehouse.returnPort*
- *Customer → VmiRetailerHQ.returnPort*

A.5 Rationale

- Using asynchronous interfaces in conversation capabilities allows to support long-running interactions between supply chain companies.
- Using asynchronous interfaces in behavior capabilities allows to support long-running behaviors.
- By using SOAP-based interfaces for capabilities, capabilities can be required and provided using normal Web services. This greatly improves the interoperability of both role components and actor components, as

well as internal implementations of conversation connectors and behavior components.

- Executing conversation connectors and behavior components as persistent BPEL processes allows to automatically recover and resume conversaion connectors and behavior components in case an application server goes down (e.g., because of hardware failure, power outages, or maintenance restarts).
- Active role states enable concurrent occurences of conversation connectors and behavior components. This allows to handle multiple forecastings, call-off orders, transports, etc. in parallel.
- By introducing a life-cycle for the different roles, we can safely add and remove role components from the organization.
- By instantiating the Call-Off Fulfillment Behavior through the Call-Off Conversation, the Warehouse role is obliged to organize a transport for each call-off. Without encapsulating this responsibility in this behavior, the responsibility would be with the actor of the Warehouse role itself.

additional rationale has been omitted

Appendix B

Macodo View Documentation Example: Capabilities and Interfaces

This appendix gives an example of the documentation of capabilities and their interfaces. We partially follow the template to document interfaces proposed by [84].

B.1 Vmi Vendor Capability

The Vmi Vendor Capability defines the capability required to play the Vmi Vendor Role. It consists of a set of conversation and behavior capabilities. These capabilities are defined in separate sections.

B.1.1 Conversation Capabilities

- Forecasting-Vendor Capability
- InventoryReporting-Client Capability
- ConsumptionReporting-Seller Capability

B.1.2 Behavior Capabilities

- Replenishment Capability

- Product Return Capability

B.2 CallOff-Caller Capability

The CallOff-Caller Capability is the conversation capability required to play the Caller Conversation Role of the Call-Off Conversation. The capability defines a conversation and participant interface.

B.2.1 Conversation Interface

Interface Identity

This is a SOAP Web service interface available to the provider of the CallOff-Caller Capability. This is typically the participant playing the corresponding conversation role. The main purpose of this interface is to allow the participant to place a call-off order.

Resources

```
void sendCallOffOrder(CallOffOrder order);
```

Send the initial call-off order.

Pre-Conditions

- The CallOffOrder argument must not be null.
- The following components of the CallOffOrder argument must not be null: orderID, deliveryDeadline, productType, quantity.

Post-Conditions

- A successful call to this interface will pass the call-off order to the conversation.

Error Handling

- **InvalidOrderException.** The operation throws this exception if the order or one of the required order components it null.

```
void sendDeliveryReport(DeliveryReport report);
```

Send a delivery report after receiving a specific order.

Pre-Conditions

- The DeliveryReport argument must not be null.
- The following components of the DeliveryReport argument must not be null: orderID, productType, quantity, condition.
- This operation can only be invoked after invoking the sendCallOffOrder operation for the corresponding order, receiving the receiveDeliveryInfo operation on the participant interface, and physically receiving the order.

Post-Conditions

- A successful call to this interface will pass the delivery report to the conversation.

Error Handling

- **InvalidReportException.** The operation throws this exception if the report or one of the required report components it null.

Data Types and Constants

- **CallOffOrder** This type is used as an argument for the sendCallOffOrder operation. It is a data structure that contains all necessary information to place an order. The attributes of a CallOffOrder object are listed below:
  - String orderID
  - DateTime orderDate
  - DateTime deliveryDeadline
  - ProductType productType
  - Quantity quantity
  - Location deliveryLocation
  - Contactinfo contactPerson
- **DeliveryReport** This type is used as an argument for the sendDeliveryReport operation. It is a data structure that contains all necessary information to submit a delivery report. The attributes of a DeliveryReport object are listed below:
  - String orderID
  - DateTime deliveryData
  - ProductType productType
  - Quantity quantity
  - ProductCondition condition

some data types have been omitted from the example

B.2.2 Participant Interface

Interface Identity

This is a SOAP Web service interface available to the requirer of the CallOff-Caller Capability. This is typically the conversation (or corresponding conversation role). The main purpose of this interface is to allow the conversation to pass required information back to the participant that places a call-off order.

Resources

```
void receiveDeliveryInfo(DeliveryInfo info);
```

Receive the delivery info for a specific call-off order.

Pre-Conditions

- The DeliveryInfo argument must not be null.
- The following components of the DeliveryInfo argument must not be null: orderID, deliveryDeadline, productType, quantity.
- This operation can only be invoked after receiving the sendCallOffOrder operation for the corresponding order on the conversation interface.

Post-Conditions

- A successful call to this interface will pass the delivery info to the participant.

Error Handling

- **InvalidInfoException.** The operation throws this exception if the info or one of the required order components it null.

Data Types and Constants

- **DeliveryInfo** This type is used as an argument for the receiveDeliveryInfo operation. It is a data structure that contains all necessary information to notify a participant of the particular delivery details. The attributes of a CallOffOrder object are listed below:
  - String orderID
  - DateTime deliveryData
  - ProductType productType

- Quantity quantity
- Location deliveryLocation
- Contactinfo contactPerson

some data types have been omitted from the example

Error Handling

All operations in this interface can raise the following exception, in addition to operation specific exceptions:

- **RemoteException.** The caller receives a RemoteException when there is a communication problem with the service provider implementing this interface.

Variatbility

N/A

Appendix C

Middleware Appendix

C.1 XML Schemas

The following listing gives a complete specification of the XML schemas used to specify the different Macodo modules:

```
1  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
2      targetNamespace="http://macodo.com/schema/specs"
3      xmlns:tns="http://macodo.com/schema/specs"
4      elementFormDefault="qualified">
5
6      <!-- CAPABILITIES -->
7
8      <xsd:complexType name="behaviorCapability">
9          <xsd:attribute name="name"
10                        type="xsd:string"/>
11         <xsd:attribute name="interfaceSpecification"
12                        type="xsd:string"/>
13         <xsd:attribute name="actorPorthType"
14                        type="xsd:string"/>
15         <xsd:attribute name="behaviorPortType"
16                        type="xsd:string"/>
17     </xsd:complexType>
18
19     <xsd:complexType name="conversationCapability">
20         <xsd:attribute name="name"
21                        type="xsd:string"/>
22         <xsd:attribute name="interfaceSpecification"
23                        type="xsd:string"/>
24         <xsd:attribute name="participantPortType"
25                        type="xsd:string"/>
26         <xsd:attribute name="conversationPortType"
27                        type="xsd:string"/>
28     </xsd:complexType>
```

```
29
30      <!-- BEHAVIOR MODULES -->
31
32      <xsd:complexType name="behaviorComponentType">
33          <xsd:sequence>
34              <xsd:element name="actorPort">
35                  <xsd:complexType>
36                      <xsd:attribute name="behaviorCapability"
37                                     type="xsd:string"/>
38                      <xsd:attribute name="partnerLink" type="xsd:string"/>
39                  </xsd:complexType>
40              </xsd:element>
41              <xsd:element name="conversationPort" minOccurs="0"
42                           maxOccurs="unbounded" type="tns:conversationPort"/>
43          </xsd:sequence>
44          <xsd:attribute name="name" type="xsd:string"/>
45          <xsd:attribute name="behaviorSpecification" type="xsd:string"/>
46      </xsd:complexType>
47
48      <xsd:complexType name="conversationPort">
49          <xsd:attribute name="name" type="xsd:string"/>
50          <xsd:attribute name="conversationCapability" type="xsd:string"/>
51          <xsd:attribute name="partnerLink" type="xsd:string"/>
52      </xsd:complexType>
53
54      <!-- CONVERSATION MODULES -->
55
56      <xsd:complexType name="conversationConnectorType">
57          <xsd:sequence>
58              <xsd:element name="conversationRole" minOccurs="0"
59                           maxOccurs="unbounded" type="tns:conversationRole"/>
60          </xsd:sequence>
61          <xsd:attribute name="name" type="xsd:string"/>
62          <xsd:attribute name="protocolSpecification" type="xsd:string"/>
63      </xsd:complexType>
64
65      <xsd:complexType name="conversationRole">
66          <xsd:attribute name="name" type="xsd:string"/>
67          <xsd:attribute name="conversationCapability" type="xsd:string"/>
68          <xsd:attribute name="minOccur" type="xsd:int"/>
69          <xsd:attribute name="maxOccur" type="xsd:int"/>
70          <xsd:attribute name="initiator" type="xsd:boolean"/>
71          <xsd:attribute name="partnerLink" type="xsd:string"/>
72      </xsd:complexType>
73
74      <!-- ROLE MODULES -->
75
76      <xsd:complexType name="RoleComponentType">
77          <xsd:sequence>
78              <xsd:element name="conversationPort" minOccurs="0"
79                           maxOccurs="unbounded" type="tns:conversationPort"/>
80              <xsd:element name="behaviorType" minOccurs="0"
81                           maxOccurs="unbounded" type="tns:behaviorType"/>
82              <xsd:element name="interfaceDelegation" minOccurs="0"
83                           maxOccurs="unbounded" type="tns:interfaceDelegation"/>
84              <xsd:element name="roleState" minOccurs="0"
85                           maxOccurs="unbounded" type="tns:roleState"/>
```

```
86          </xsd:sequence>
87          <xsd:attribute name="name" type="xsd:string"/>
88      </xsd:complexType>
89
90      <xsd:complexType name="interfaceDelegation">
91          <xsd:attribute name="behaviorType" type="xsd:string"/>
92          <xsd:attribute name="behaviorConversationPort" type="xsd:string"/>
93          <xsd:attribute name="roleConversationPort" type="xsd:string"/>
94      </xsd:complexType>
95
96      <xsd:complexType name="behaviorType">
97          <xsd:attribute name="name" type="xsd:string"/>
98          <xsd:attribute name="behaviorComponentType" type="xsd:string"/>
99      </xsd:complexType>
100
101     <xsd:complexType name="roleState">
102         <xsd:sequence>
103             <xsd:element name="conversationConstraint" minOccurs="0"
104                          maxOccurs="unbounded"
105                          type="tns:conversationConstraint"/>
106             <xsd:element name="behaviorConstraint" minOccurs="0"
107                          maxOccurs="unbounded"
108                          type="tns:behaviorConstraint"/>
109         </xsd:sequence>
110         <xsd:attribute name="name" type="xsd:string"/>
111     </xsd:complexType>
112
113     <xsd:complexType name="conversationConstraint">
114         <xsd:attribute name="conversationPort" type="xsd:string"/>
115         <xsd:attribute name="creation" type="xsd:boolean"/>
116         <xsd:attribute name="maxOccur" type="xsd:int"/>
117     </xsd:complexType>
118
119     <xsd:complexType name="behaviorConstraint">
120         <xsd:attribute name="behaviorType" type="xsd:string"/>
121         <xsd:attribute name="creation" type="xsd:boolean"/>
122         <xsd:attribute name="maxOccur" type="xsd:int"/>
123     </xsd:complexType>
124
125     <!-- ORGANIZATION MODULES -->
126
127     <xsd:complexType name="organizationConnectorType">
128         <xsd:sequence>
129             <xsd:element name="roleType" minOccurs="0"
130                          maxOccurs="unbounded" type="tns:roleType"/>
131             <xsd:element name="conversationType" minOccurs="0"
132                          maxOccurs="unbounded"
133                          type="tns:conversationType"/>
134         </xsd:sequence>
135         <xsd:attribute name="name" type="xsd:string"/>
136     </xsd:complexType>
137
138     <xsd:complexType name="roleType">
139         <xsd:attribute name="name" type="xsd:string"/>
140         <xsd:attribute name="roleComponentType" type="xsd:string"/>
141         <xsd:attribute name="minOccur" type="xsd:int"/>
142         <xsd:attribute name="maxOccur" type="xsd:int"/>
```

```
143     </xsd:complexType>
144
145     <xsd:complexType name="conversationType">
146         <xsd:sequence>
147             <xsd:element name="roleMapping" minOccurs="0"
148                          maxOccurs="unbounded" type="tns:roleMapping"/>
149         </xsd:sequence>
150         <xsd:attribute name="name" type="xsd:string"/>
151         <xsd:attribute name="conversationConnectorType" type="xsd:string"/>
152     </xsd:complexType>
153
154     <xsd:complexType name="roleMapping">
155         <xsd:attribute name="conversationRole" type="xsd:string"/>
156         <xsd:attribute name="roleType" type="xsd:string"/>
157         <xsd:attribute name="conversationPort" type="xsd:string"/>
158     </xsd:complexType>
159
160 </xsd:schema>
```

C.2 Macodo Management Service

The following listing gives a complete specification of the management interface of the Macodo middleware:

```
1  public interface ManagementInterface{
2
3    //ACTOR COMPONENTS
4
5    ActorID registerActor(String name);
6
7    void unregisterActor(ActorID actorID)
8      throws NoSuchActorException;
9
10   void registerProvidedCapability(ActorID actorComponentID,
11                                   ProvidedCapability providedCapability)
12     throws NoSuchActorException, InvalidCapabilityException;
13
14   boolean unregisterProvidedCapability(
15                              ActorID actorID,
16                              ProvidedCapabilityID providedCapabilityID)
17     throws NoSuchActorException, NoSuchCapabilityException;
18
19   Set<CapabilityID> getProvidedCapabilities(ActorID actorID)
20     throws NoSuchActorException;
21
22   //ORGANIZATION CONNECTORS
23
24   OrganizationConnectorID createOrganizationConnector(
25                       OrganizationConnectorType organizationConnectorType,
26                       String name)
27     throws InvalidOrganizationConnectorTypeException;
28
```

```
29    void destroyOrganizationConnector(
30                        OrganizationConnectorID organizationConnectorID)
31      throws NoSuchOrganizationConnectorException;
32
33    Set<OrganizationConnectorID> getOrganizationConnectors();
34
35    //ROLE COMPONENTS
36
37    RoleComponentID createRoleComponent(
38                        OrganizationConnectorID organizationConnectorID,
39                        RoleType roleType,
40                        ActorID actorID)
41      throws NoSuchOrganizationConnectorException,
42             IllegalRoleComponentTypeException,
43             RoleMultiplicityException, NoSuchActorException,
44             InsufficientCapabilitiesException;
45
46    RoleEndpoint getRoleEndpoint(RoleComponentID roleComponentID)
47      throws NoSuchRoleComponentException;
48
49    Set<RequiredCapability> getRequiredCapabilities(RoleType roleType)
50      throws IllegalRoleComponentTypeException;
51
52    set<RoleComponentID> getRoleComponents(
53                    OrganizationConnectorID organizationConnectorID)
54      throws NoSuchOrganizationConnectorException;
55
56    void destroyRoleComponent(RoleComponentID roleComponentID)
57      throws NoSuchRoleComponentException;
58
59    void setRoleState(RoleComponentID roleComponentID,
60                      RoleState roleState)
61      throws NoSuchRoleComponentException, IllegalRoleStateException;
62
63    RoleState getRoleState(RoleComponentID roleComponentID)
64      throws NoSuchRoleComponentException;
65
66    ActorID getActor(RoleComponentID roleComponentID)
67      throws NoSuchRoleComponentException;
68
69    Set<BehaviorComponentID> getBehaviorComponents(
70                                  RoleComponentID roleComponentID)
71      throws NoSuchRoleComponentException;
72
73    Set<ConversationRoleID> getConversationRoles(
74                                  RoleComponentID roleComponentID)
75      throws NoSuchRoleComponentException;
76  }
```

Appendix D

Evaluation Appendix

D.1 Calculating Function Points

As a size measurement, we use the Albrecht's approach to calculate function points [9, 72]. Function points are intended to measure the amount of functionality in a system described by its specification. To compute the number of function points, we first have to compute the unadjusted function point count (UFC). To do so, the software is represented in terms of external inputs, external outputs, external inquiries, external files, and internal files. We only consider external inputs and outputs. Next, we have to assign each item a subjective "complexity": 'simple', 'average', or 'complex'. The corresponding weights are given below [72]:

	Weighting Factor		
Item	**Simple**	**Average**	**Complex**
External inputs	3	4	6
External outputs	4	5	7

The actual measure used for size is the adjusted function point count (FP). The FP is calculated by multiplying the UFC with a technical complexity factor (TFC). The TFC is calculated by selecting those factors (out of 14) that contribute to the technical complexity of the system (the complete list can be found in [72]) with a score of 0 (irrelevant), 3 (average), or 5 (essential). We consider two technical complexity factors: F_5 (heavily used configuration) with a rating 5 and F_{11} (reusability) with a rating 3. Our technical complexity factor (TFC) results in:

$$TCF = 0.65 + 0.01 \times (5 + 3) = 0.73 \tag{D.1}$$

The requirements of both assignments can be divided in three main scenarios. Each scenario requires two interactions and multiple scenarios can require the same interaction. An overview of the four possible interactions in assignment A (eHealth case) is given in Fig. D.1 (interactions for assignment B are similar). The corresponding unadjusted function point count (FPC) and adjusted function point count (FP) for the scenarios are given below (calculations are the same for both assignments):

Unadjusted Function Point Count (UFC)

	Scenario 1	**Scenario 2**	**Scenario 3**
Interaction a	17	17	
Interaction b			7
Interaction c	17		
Interaction d		14	14
Total UCF	34	31	21

Adjusted Function Point Count (FP)

	Scenario 1	**Scenario 2**	**Scenario 3**
Interaction a	12	12	
Interaction b			5
Interaction c	12		
Interaction d		10	10
Total FC	24	22	15

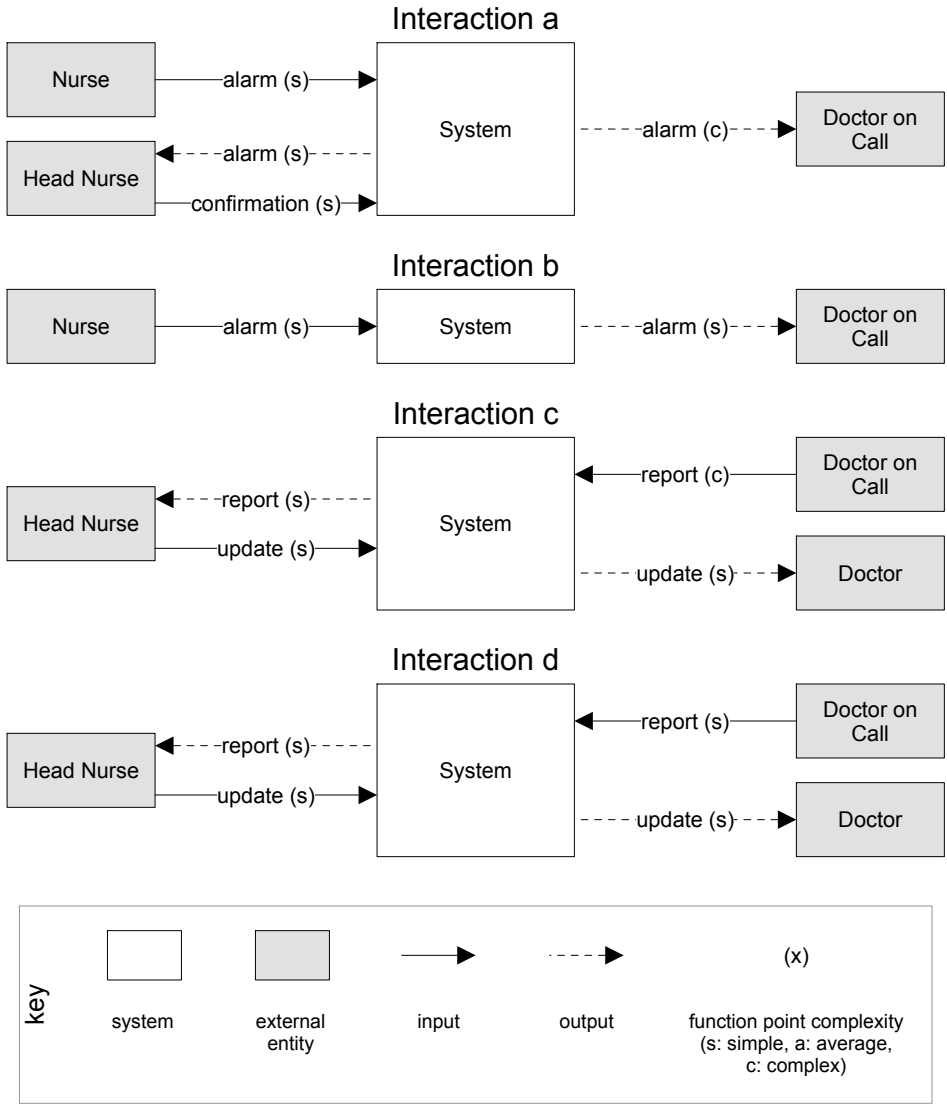


Figure D.1: An overview of the interactions for assignment A (eHealth case). The alarm output in interaction a has a higher complexity because it is a conditional output. The report input of interaction c and d have a different complexity based on when they can occur.

D.2 Notations Used in the Experiment

Figure D.2, D.3, D.4, D.5, D.6, and D.7 give an overview and some examples of the notation used for Macodo and the reference approach during the evaluation. A complete discussion of the notation can be found online[1]

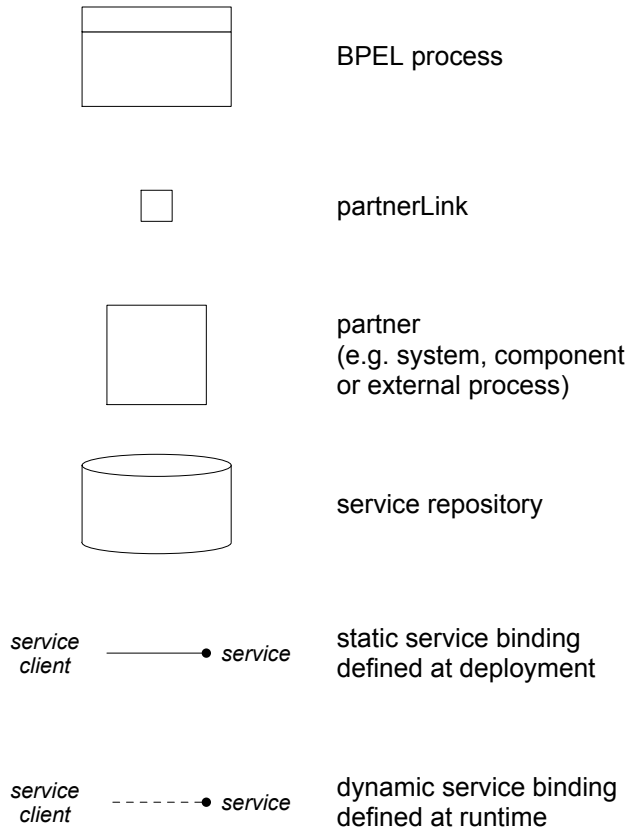


Figure D.2: Key to document the architecture of the system in the reference approach.

[1] http://people.cs.kuleuven.be/~robrecht.haesevoets/macodo/

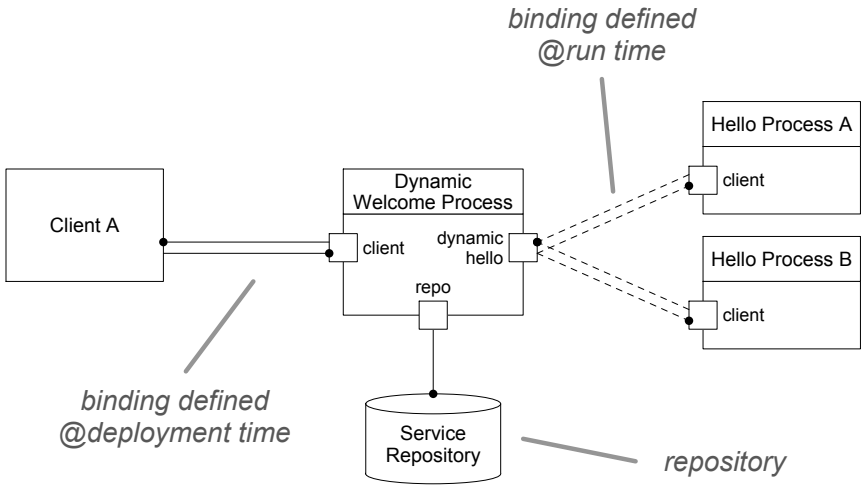


Figure D.3: An example of the notation used to document the architecture in the reference approach.

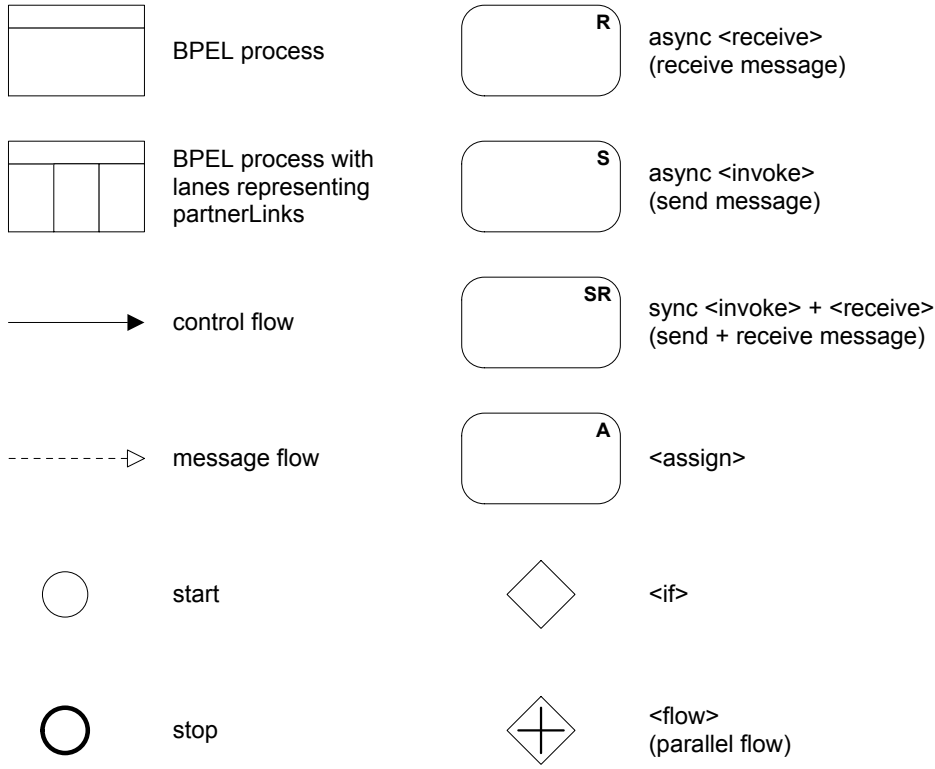


Figure D.4: Key for the simplified process notation used in the detailed design for both the reference approach and Macodo.

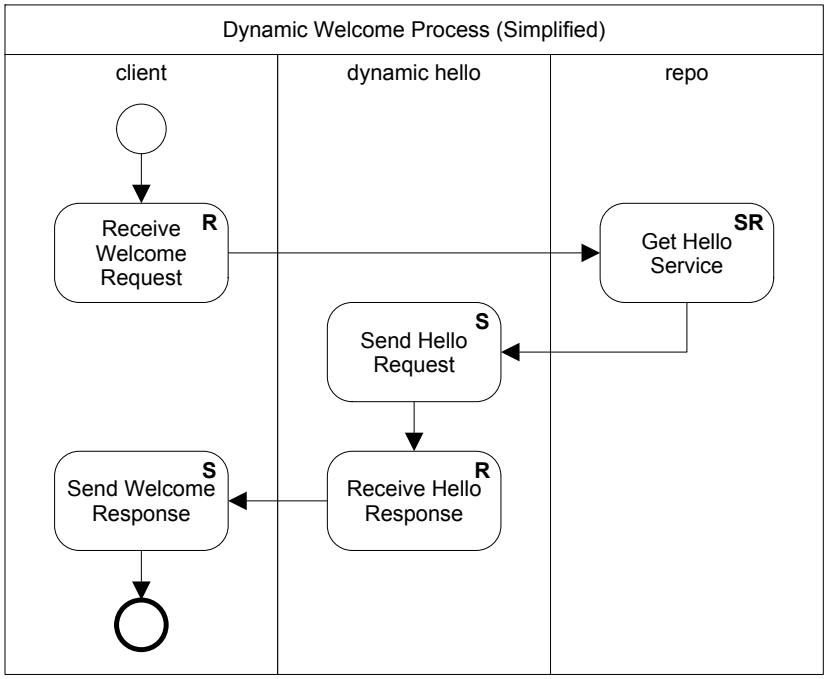


Figure D.5: An example of the simplified process notation.

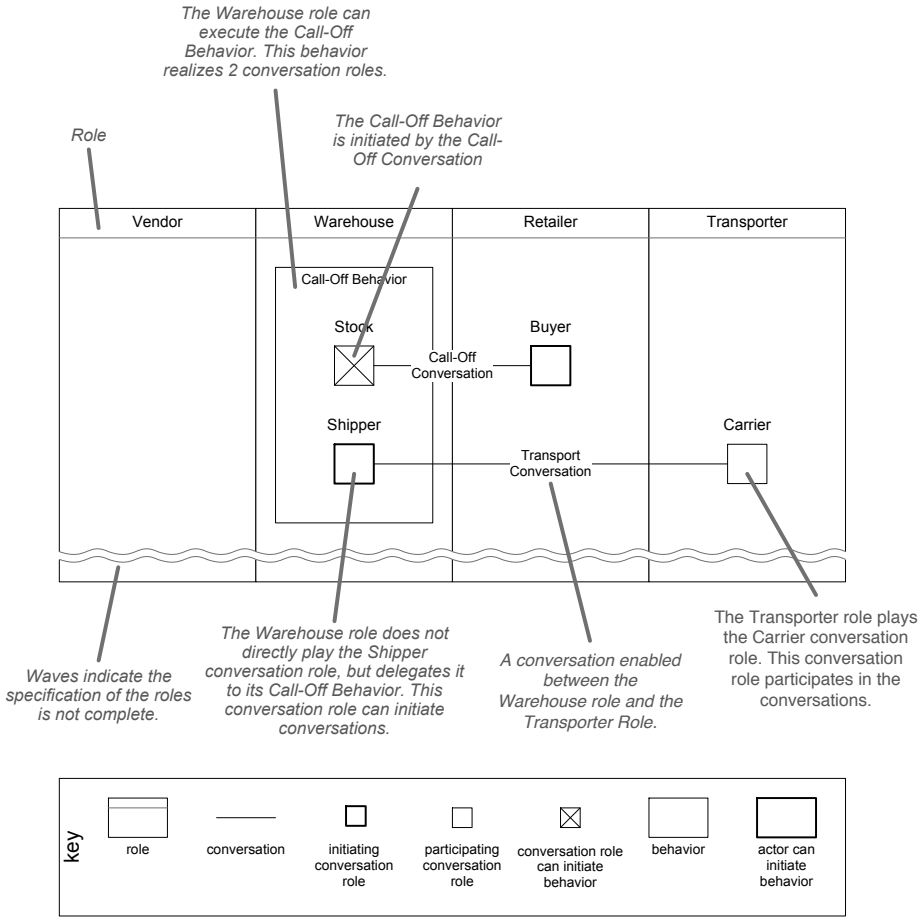


Figure D.6: The notation used to document the architecture with Macodo.

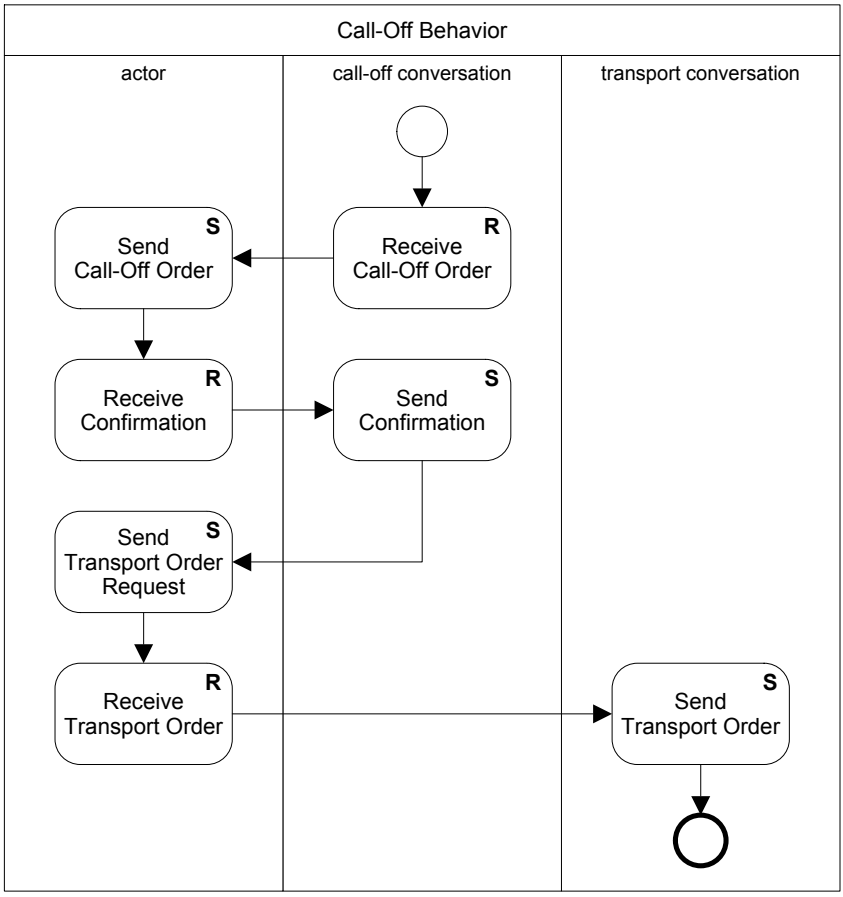


Figure D.7: An example of the simplified process notation used to document a behavior in Macodo.

D.3 Measuring Changes

The design of a solution consists of an architecture of the system in terms of modules, and a detailed design of each module. In our experiment, we only measure changes required in the detailed design. In the detailed design, subjects use the same notation for both treatments, allowing to define a uniform measure for changes. This is, however, not the case for the architecture of the system.

Below we give a table of the possible changes (in the context of the simplified notation) and their cost:

Type of Change	**Cost**
add/replace <if> activity	3
move/remove <if> activity	1
add conditions <if> activity	1
add/replace async <invoke>/<receive> activity	2
add/replace sync <invoke> activity	3
move/remove async <invoke>/<receive>	1
move/remove sync <invoke> activity	1
inverting <send>/<receive> label of activity	1
add/remove start/end point	1
add/replace lane (i.e., partnerLink)	3
remove unused lane (i.e., partnerLink)	1

The cost of changes takes the simplifications of the process notation into account. More specifically, students could omit trivial assign activities. As a result the cost of an async <invoke>/<receive> is 2, assuming we require one <assign> activity before the <invoke> activity, and one <assign> after the <receive> activity. The cost of a sync <invoke> activity is 3, assuming we require one <assign> activity before and after this activity.

Bibliography

[1] Workflow Management Coalition Terminology & Glossary. Technical Report 3, Workflow Management Coalition, 1999.

[2] Collaborative Planning, Forecasting and Replenishment (CPFR) Version 2.0. Technical report, Voluntary Interindustry Commerce Standards (VICS), 2002.

[3] Web Services Security: SOAP Message Security 1.1 (WS-Security 2004). Technical report, OASIS Open, 2006.

[4] ISO/IEC 42010 systems and software engineering – architectural description. Technical report, ISO, 2007.

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

[6] Business Process Model and Notation (BPMN) version 2.0. Technical report, OMG, 2011.

[7] M. Adams, A. Hofstede, D. Edmond, and W. van der Aalst. Worklets: A service-oriented implementation of dynamic flexibility in workflows. *On the Move to Meaningful Internet Systems CoopIS DOA GADA and ODBASE*, 4275:291–308, 2006.

[8] H. Afsarmanesh and L. M. Camarinha-Matos. A framework for management of virtual organization breeding environments. *Collaborative networks and their breeding environments*, 23:35–48, 2005.

[9] A. J. Albrecht and J. E. Gaffney. Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation. *IEEE Transactions on Software Engineering*, SE-9(6):639–648, 1983.

[10] H. Aldewereld, J. Padget, W. Vasconcelos, J. Vazquez-Salceda, P. Sergeant, and A. Staikopoulos. Adaptable, organization-aware, service-oriented computing. *IEEE Intelligent Systems*, 25(4):0–4, 2010.

[11] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.

[12] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architectures and Applications*. Springer, 2004.

[13] G. Alonso, U. Fiedler, C. Hagen, A. Lazcano, H. Schuldt, and N. Weiler. WISE: business to business e-commerce. In *Proceedings Ninth International Workshop on Research Issues on Data Engineering Information Technology for Virtual Enterprises RIDEVE99*, pages 132–139. IEEE Computer Society, 1999.

[14] C. W. Bachman and M. Daya. The Role Concept in Data Models. In *Proceedings of the 3rd International Conference on Very Large Date Bases VLDB*, pages 464–476. IEEE Computer Society, 1977.

[15] M. Baldoni, D. Informatica, and L. V. D. Torre. powerJava : Ontologically Founded Roles in Object Oriented Programming Languages. *Power*, pages 1414–1418, 2006.

[16] A. P. Barros and M. Dumas. The Rise of Web Service Ecosystems. *It Professional*, 8(5):31–37, 2006.

[17] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley Professional, 2 edition, Apr. 2003.

[18] M. Becht, T. Gurzki, J. Klarmann, and M. Muscholl. ROPE: role oriented programming environment for multiagent systems. *Proceedings Fourth IFCIS International Conference on Cooperative Information Systems. CoopIS 99 (Cat. No.PR00384)*, pages 325–333, 1999.

[19] J. Becker, P. Delfmann, A. Dreiling, R. Knackstedt, and D. Kuropka. Configurative Process Modeling - Outlining an Approach to Increased Business Process Model Usability. In M. Khosrowpour, editor, *Proceedings of the 15th IRMA International Conference*, pages 615–619. IRM Press, 2004.

[20] F. Bellifemine, A. Poggi, and G. Rimassa. Jade, A FIPA-compliant Agent Framework. In *4th International Conference on Practical Application of Intelligent Agents and Multi-Agent Technology*, London, UK, 1999.

[21] O. Boissier, J. F. Hübner, and J. S. Sichman. Organization Oriented Programming From Closed to Open Organizations. *Engineering Societies in the Agents World VII ESAW 06*, 4457(29):86–105, 2007.

[22] F. P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*, volume 7. Addison-Wesley, 1995.

[23] G. Cabri, L. Leonardi, and F. Zambonelli. Separation of concerns in agent applications by roles. In *ICDCSW '02: Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 430–438, Washington DC, USA, 2002. IEEE Computer Society.

[24] G. Cabri, L. Leonardi, and F. Zambonelli. XRole: XML roles for agent interaction. In *Proceedings of the 3rd International Symposium "From Agent Theory to Agent Implementation"*, 2002.

[25] G. Cabri, L. Leonardi, and F. Zambonelli. BRAIN: a framework for flexible role-based interactions in multiagent systems. In *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*, pages 145–161. Springer, 2003.

[26] A. Caetano, M. Zacarias, A. R. Silva, and J. Tribolet. A Role-Based Framework for Business Process Modeling. In *Proceedings of the 38th Annual Hawaii International Conference on System Sciences*. IEEE, 2005.

[27] L. Camarinha-Matos and H. Afsarmanesh. Elements of a base VE infrastructure. *Computers in Industry*, 51(2):139–163, 2003.

[28] L. Camarinha-Matos and H. Afsarmanesh. *Collaborative Networked Organizations: A research agenda for emerging business models*. Springer, 2004.

[29] L. Camarinha-Matos and H. Afsarmanesh. The emerging discipline of collaborative networks. *Virtual enterprises and collaborative networks*, pages 3–16, 2004.

[30] L. Camarinha-Matos and H. Afsarmanesh. Virtual enterprise modeling and support infrastructures: applying multi-agent system approaches. *Multi-Agent Systems and Applications*, pages 335–364, 2006.

[31] D. T. Campbell and J. C. Stanley. *Experimental and quasi-experimental designs for research*, volume 20. Rand McNally, 1963.

[32] H. L. Cardoso and E. Oliveira. Virtual Enterprise Normative Framework Within Electronic Institutions. In *Engineering societies in the agents world V*, pages 14–32. Springer, 2005.

[33] J. Cardoso. Control-flow Complexity Measurement of Processes and Weyuker ' s Properties. *Engineering and Technology*, 8(October):213–218, 2005.

[34] J. Cardoso. Process control-flow complexity metric: An empirical validation. In *International Conference on Services Computing*, pages 167–173. IEEE Computer Society, 2006.

[35] M. C. Carley and L. Gasser. Computational Organization Theory. chapter Social Dil, pages 201–253. Routledge, 1995.

[36] C. Castelfranchi. Modelling social action for AI agents. *Artificial Intelligence*, 103(1-2):157–182, 1998.

[37] A. Charfi and M. Mezini. Aspect-oriented web service composition with AO4BPEL. *Web Services*, 3250:168–182, 2004.

[38] A. Charfi and M. Mezini. AO4BPEL: An Aspect-oriented Extension to BPEL. *World Wide Web Internet And Web Information Systems*, 10(3):309–344, 2007.

[39] A. Charfi and H. Müller. Aspect-Oriented Business Process Modeling with AO4BPMN. *ECMFA*, pages 48–61, 2010.

[40] I. Chebbi, S. Dustar, and S. Tata. The view-based approach to dynamic inter-organizational workflow cooperation. *Data & Knowledge Engineering*, 56(2):139–173, 2006.

[41] D. Chiu, K. Karlapalem, Q. Li, and E. Kafeza. Workflow View Based E-Contracts in a Cross-Organizational E-Services Environment. *Distributed and Parallel Databases*, 12(2-3):193–216, 2002.

[42] S. Chopra and P. Meindl. *Supply Chain Management: Strategy, Planning and Operation*. Pearson Prentice Hall, 2007.

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

[44] M. Christopher. *Logistics and supply chain management: creating value-added networks*. Financial Times Series. Pearson Education, 2005.

[45] T. Cook and J. Stanley. *Quasi-experimentation: Design & analysis issues for field settings*. Houghton Mifflin Company, 1979.

[46] C. Courbis and A. Finkelstein. Towards an aspect weaving BPEL engine. *The Third AOSD Workshop on Aspects Components and Patterns for Infrastructure Software ACP4IS Lancaster UK*, (March), 2004.

[47] M. Cunha. Environments for Virtual Enterprise Integration. *International Journal of Enterprise Information Systems*, 5(4):71–87, 2009.

[48] B. Curtis, M. I. Kellner, and J. Over. Process modeling. *Communications of the ACM*, 35(9):75–90, 1992.

[49] K. Czarnecki and M. Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. *Generative Programming and Component Engineering*, 3676:422–437, 2005.

[50] N. P. Dalal, M. Kamath, W. J. Kolarik, and E. Sivaraman. Toward an integrated framework for modeling enterprise processes. *Communications of the ACM*, 47(3):83–87, 2004.

[51] T. H. Davenport. Putting the enterprise into the enterprise system. *Harvard Business Review*, 76(4):121–131, 1998.

[52] P. Davidsson. Categories of Artificial Societies. In *Engineering Societies in the Agents World II (ESAW)*, volume 2203 of *Lecture Notes in Artificial Intelligence*, pages 1 – 9. Springer-Verlag, 2001.

[53] D. Davis. Web Services Reliable Messaging. *Proceedings IEEE International Conference on Web Services 2004*, (February):1–62, 2009.

[54] Y. Demazeau and A. C. R. Costa. Populations and organizations in open multi-agent systems. In *Proceedings of the 1st National Symposium on Parallel and Distributed AI*, pages 1–13, 1996.

[55] N. Desai, a.K. Mallya, a.K. Chopra, and M. Singh. Interaction protocols as design abstractions for business processes. *IEEE Transactions on Software Engineering*, 31(12):1015–1027, Dec. 2005.

[56] N. Desai, A. K. Chopra, and M. P. Singh. Representing and Reasoning About Commitments in Business Processes. *Artificial Intelligence*, 22(2):1328–1333, 2007.

[57] N. Desai, A. K. Chopra, and M. P. Singh. Amoeba: A methodology for modeling and evolving cross-organizational business processes. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 19(2):1–45, 2008.

[58] F. Dignum. Autonomous agents with norms. *Artificial Intelligence and Law*, 7(1):69–79, 1999.

[59] F. Dignum, V. Dignum, J. Padget, and J. Vázquez-Salceda. Organizing web services to develop dynamic, flexible, distributed systems. *Proceedings of the 11th International Conference on Information Integration and Web-based Applications & Services - iiWAS '09*, page 225, 2009.

[60] V. Dignum. A model for organizational interaction: based on agents, founded in logic, 2004.

[61] V. Dignum. *Handbook of research on multi-agent systems: semantics and dynamics of organizational models*. Information Science Reference, Hershey, New York, USA, 2009.

[62] V. Dignum. The role of organization in agent systems. In V. Dignum, editor, *Handbook of research on multi-agent systems: semantics and dynamics of organizational models*, chapter 1, pages 1–16. Information Science Reference, Hershey, New York, USA, 2009.

[63] V. Dignum and H. Aldewereld. OperettA: Organization-Oriented Development Environment. *Proceedings of the 3rd International workshop on Languages Methodologies and Development Tools for Multiagent Systems LADS2010 Mallow*, 2010.

[64] V. Dignum, F. Dignum, and J.-J. Meyer. An agent-mediated approach to the support of knowledge sharing in organizations. *The Knowledge Engineering Review*, 19(02):147–174, 2005.

[65] V. Dignum, J. Vázquez-Salceda, and F. Dignum. Omni: Introducing social structure, norms and ontologies into agent organizations. *Programming MultiAgent Systems*, pages 181–198, 2005.

[66] A. Drogoul, B. Corbara, and S. Lalande. MANTA : New Experimental Results on the Emergence of (Artificial) Ant Societies 2 . The MANTA Agents Model of Behaviour. *Behaviour*, pages 190–211, 1995.

[67] H. Eberle, T. Unger, and F. Leymann. Process Fragments. In R. Meersman, T. Dillon, and P. Herrero, editors, *On the Move to Meaningful Internet Systems OTM 2009 Part I*, volume 5870 of *Lecture Notes in Computer Science*, pages 398–405. Springer, 2009.

[68] T. Erl. *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*. Prentice Hall, 2005.

[69] M. Esteva, D. De La Cruz, and C. Sierra. ISLANDER: an electronic institutions editor. *Proceedings of the First International Joint Conference on Auton omous Agents and Multiagent Systems*, 3:1045–1052, 2002.

[70] M. Esteva and J.-A. Rodríguez-Aguilar. On the Formal Specification of Electronic Institutions. *Agent mediated electronic commerce*, 1991:126–147, 2001.

[71] M. Esteva, B. Rosell, J. A. Rodriguez-Aguilar, and J. L. Arcos. Ameli: An agent-based middleware for electronic institutions. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 1*, pages 236–243. IEEE Computer Society Washington, DC, USA, 2004.

[72] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach, Revised*. Course Technology, 1998.

[73] J. Ferber and O. Gutknecht. A meta-model for the analysis and design of organizations in multi-agent systems. In *Proceedings International Conference on Multi Agent Systems*, pages 128–135. IEEE, 1998.

[74] J. Ferber, O. Gutknecht, and F. Michel. From Agents to Organizations : an Organizational View of Multi-Agent Systems. In *Agent-Oriented Software Engineering IV*, pages 214–230. Springer, 2003.

[75] J. Ferber, F. Michel, and J. Baez. AGRE: Integrating environments with organizations. In *First International Workshop on Environments for Multi-Agent Systems*, volume 3374 of *Lecture Notes in Computer Science*, pages 48–56, New York, NY, USA, 2005. Springer-Verlag.

[76] J. Ferber, T. Stratulat, and J. Tranier. Towards an integral approach of organizations in multi-agent systems: the MASQ approach. *Multiagent Systems Semantics and Dynamics of Organizational Models Virginia Dignum eds IGI*, (March):1–23, 2009.

[77] P. Fettke and P. Loos. Classification of reference models: a methodology and its application. *Information Systems and eBusiness Management*, 1(1):35–53, 2003.

[78] P. Fettke, P. Loos, and J. Zwicker. Business Process Reference Models: Survey and Classification. *Business Process Management Workshops*, 3812(Bprm):469–483, 2006.

[79] M. Fiammante. *Dynamic SOA and BPM: best practices for business process management and SOA agility*. IBM Press, 2009.

[80] N. Fornara and M. Colombetti. Specifying and enforcing norms in artificial institutions. *Declarative Agent Languages and Technologies VI*, (2204):1–17, 2009.

[81] I. Foster. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of High Performance Computing Applications*, 15(3):200–222, Aug. 2001.

[82] W. Frakes and C. Terry. Software reuse: metrics and models. *ACM Computing Surveys*, 28(2):415–435, 1996.

[83] A. Garcia-Camino, P. Noriega, and J. Rodriguez-Aguilar. Implementing norms in electronic institutions. *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems - AAMAS '05*, page 667, 2005.

[84] D. Garlan, F. Bachmann, J. Ivers, J. Stafford, L. Bass, P. Clements, and P. Merson. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, 2 edition, Oct. 2010.

[85] L. Gasser. Perspectives on organizations in multi-agent systems. In *Mutli-agents systems and applications*, pages 1–16, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[86] C. Ghezzi, M. Jazayeri, and D. Mandrioli. Fundamentals of software engineering. Jan. 1991.

[87] J. D. Gibbons and D. A. Wolfe. Nonparametric Statistical Inference. *Technometrics*, pages 185–194, 2003.

[88] F. Gottschalk, W. van der Aalst, M. Jansen-Vullers, and M. La Rosa. Configurable Workflow Models. *International Journal of Cooperative Information Systems*, 17(2):177–221, 2008.

[89] P. Grefen, K. Aberer, Y. Hoffner, and H. Ludwig. CrossFlow: Cross-organizational workflow management in dynamic virtual enterprises. *Computer Systems Science and Engineering*, 15(5):277, 2000.

[90] P. Grefen, N. Mehandjiev, G. Kouvas, G. Weichhart, and R. Eshuis. Dynamic business network process management in instant virtual enterprises. *Computers in Industry*, 60(2):86–103, Feb. 2009.

[91] D. Grossi, H. Aldewereld, and F. Dignum. Ubi Lex, Ibi Poena: Designing Norm Enforcement in E-Institutions. *Coordination Organizations Institutions and Norms in Agent Systems II*, 4386:101–114, 2007.

[92] N. Guarino. Concepts, Attributes and arbitrary relations. *Data and Knowledge Engineering*, 8(1992):249–261, 1992.

[93] M. Gudgin. SOAP Version 1.2, 2003.

[94] M. Gudgin, M. Hadley, and T. Rogers. Web Services Addressing 1.0 - Core, 2006.

[95] G. Guizzardi. *Ontological foundations for structural conceptual models.* PhD thesis, University of Twente, 2005.

[96] O. Gutknecht, J. Ferber, and F. Michel. Integrating tools and infrastructures for generic multi-agent systems. *Proceedings of the fifth international conference on Autonomous agents*, pages 441–448, 2001.

[97] H. Haas. Web Services Glossary, 2004.

[98] R. Haesevoets, B. V. Eylen, D. Weyns, A. Helleboogh, T. Holvoet, and W. Joosen. Managing Agent Interactions with Context-Driven Dynamic Organizations. In *Engineering Environment-Mediated Multi-Agent Systems*, volume 5049 of *Lecture Notes in Computer Science*, pages 166–186. Springer-Verlag, 2008.

[99] R. Haesevoets, E. Truyen, T. Holvoet, and W. Joosen. Weaving the Fabric of the Control Loop through Aspects. In *SOAR: Self-Organizing Architectures*, volume 6090 of *Lecture Notes in Computer Science*, pages 38–65, 2010.

[100] R. Haesevoets, B. Van Eylen, D. Weyns, A. Helleboogh, T. Holvoet, and W. Joosen. Context-driven dynamic organizations applied to coordinated monitoring of traffic jams. In D. Weyns, S. Brueckner, and Y. Demazeau, editors, *Engineering Environment-Mediated Multiagent Systems*, pages 126–143, Dresden, Germany, 2007.

[101] R. Haesevoets, D. Weyns, M. H. Cruz Torres, A. Helleboogh, T. Holvoet, and W. Joosen. A middleware model in Alloy for supply chain-wide agent interactions. In *Agent Oriented Software Engineering (AOSE)*, volume 6788 of *Lecture Notes in Computer Science*, Toronto, Canada, 2010. Springer.

[102] R. Haesevoets, D. Weyns, and T. Holvoet. A formal specification of an organization model and management model for context-driven dynamic organizations. CW Reports, CW535. Technical report, Department of Computer Science, K.U.Leuven, Heverlee, Belgium, 2009.

[103] R. Haesevoets, D. Weyns, T. Holvoet, and W. Joosen. A formal model for self-adaptive and self-healing organizations. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 116–125. IEEE, 2009.

[104] R. Haesevoets, D. Weyns, T. Holvoet, W. Joosen, and P. Valckenaers. Hierarchical organizations and a supporting software architecture for floating car data. In *Environment-Mediated Coordination in Self-Organizing and Self-Adaptive Systems*, pages 31–36. IEEE, 2008.

[105] C. Hahn, C. Madrigal-Mora, and K. Fischer. A platform-independent metamodel for multiagent systems. *Autonomous Agents and Multi-Agent Systems*, 18(2):239–266, Apr. 2008.

[106] A. Hallerbach, T. Bauer, and M. Reichert. Capturing variability in business process models: the Provop approach. *Journal of Software Maintenance and Evolution Research and Practice*, 22(6-7):519–546, 2009.

[107] M. Hannoun, O. Boissier, J. S. a. Sichman, and C. Sayettat. MOISE: An organizational model for multi-agent systems. In *IBERAMIASBIA*, volume 1952 of *LNAI*, pages 156–165. Springer-Verlag, 2000.

[108] S. Herrmann. A precise model for contextual roles: The programming language ObjectTeams/Java. *Applied Ontology*, 2(2):181–207, 2007.

[109] J. Hill, M. Pezzini, and Y. Natis. Findings: confusion remains regarding BPM terminologies. *Gartner Research*, 501(G0015581), 2008.

[110] J. Hill, J. Sinur, D. Flint, and M. J. Melenovsky. Gartner 's Position on Business Process Management , 2006. *Gartner Research*, (February), 2006.

[111] A. Hofstede, M. Orlowska, and J. Rajapakse. Verification Problems in Conceptual Workflow Specifications. In *15th International Conference on Conceptual Modeling*, volume 1157 of *Lecture Notes in Computer Science*, pages 73–88. Springer, 1996.

[112] M. Hollander and D. A. Wolfe. *Nonparametric statistical methods*, volume 2. Wiley-Interscience, 1999.

[113] A. Holt, H. Ramsey, and J. Grimes. Coordination system technology as the basis for a programming environment. *Electrical Communication*, 57(4):307–314, 1983.

[114] B. Horling and V. Lesser. A survey of multi-agent organizational paradigms. *The Knowledge Engineering Review*, 19(04):281, Nov. 2005.

[115] N. Howden, R. Rönnquist, A. Hodgson, and A. Lucas. JACK intelligent agents-summary of an agent infrastructure. In *5th International Conference on Autonomous Agents*. Citeseer, Citeseer, 2001.

[116] J. Hübner. Moise specifications - draft. Technical report, 2010.

[117] J. Hübner, O. Boissier, R. Kitio, and A. Ricci. Instrumenting multi-agent organisations with organisational artifacts and agents. *Autonomous Agents and Multi-Agent Systems*, 20(3):369–400, Apr. 2009.

[118] J. Hübner, J. Sichman, and O. Boissier. MOISE+: towards a structural, functional, and deontic model for MAS organization. *Proceedings of the first international joint conference on Autonomous agents and multiagent systems part 1*, pages 501–502, 2002.

[119] J. Hübner, J. Sichman, and O. Boissier. Using the MOISE+ for a Cooperative Framework of MAS Reorganisation. In *SBIA*, volume 3171 of *LNAI*, pages 506–515. Springer, 2004.

[120] J. Hübner, J. S. a. Sichman, and O. Boissier. S-Moise+ : A middleware for developing organised multi-agent systems. *Coordination Organizations Institutions and Norms in MultiAgent Systems*, 3913:64–78, 2006.

[121] M. Hugos. *Essentials of supply chain management*. John Wiley & Sons, Inc., Hoboken, New Jersey, 2003.

[122] M. Huhns, M. Singh, M. Burstein, K. Decker, E. Durfee, T. Finin, L. Gasser, H. Goradia, N. R. Jennings, K. Lakartaju, H. Nakashima, V. Parunak, J. Rosenschein, A. Ruvinsky, G. Sukthankar, S. Swarup, K. Sycara, M. Tambe, T. Wagner, and L. Zavala. Research directions for service-oriented multiagent systems. *IEEE Internet Computing*, 9(December):65–70, 2005.

[123] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*, volume 19. The MIT Press, 2006.

[124] N. Jennings. On agent-based software engineering. *Artificial Intelligence*, 177(2):277–296, 2000.

[125] A. Karageorgos. Agent-based optimisation of logistics and production planning. *Engineering Applications of Artificial Intelligence*, 16(4):335–348, 2003.

[126] D. Karastoyanova, F. Leymann, and A. P. Buchmann. An Approach to Parameterizing Web Service Flows. Technical report, 2005.

[127] D. Karastoyanova, F. Leymann, J. Nitzsche, B. Wetzstein, and D. Wutke. Parameterized BPEL Processes: Concepts and Implementation. In *Business Process Management*, volume 4102 of *Lecture Notes in Computer Science*, pages 471–476. Springer, 2006.

[128] N. O. Kavantzas, D. C. O. Burdett, G. N. Ritzinger, T. C. Fletcher, Y. W. Lafon, and C. A. S. I. Barreto. Web Services Choreography Description Language Version 1.0. Technical report, W3C, 2005.

[129] E. Kendall. Role Modeling for Agent System Analysis, Design, and Implementation. *IEEE Concurrency*, 8(2):34–41, 2000.

[130] R. Khalaf. Supporting business process fragmentation while maintaining operational semantics: a BPEL perspective, 2008.

[131] R. Khalaf and F. Leymann. E Role-based Decomposition of Business Processes using BPEL. *International Conference on Web Services ICWS 2006*, pages 770–780, 2006.

[132] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. *Main*, 2072(4):327–353, 2001.

[133] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, Vol. 1241, Berlin, Heidelberg, New York, 1997. Springer-Verlag.

[134] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734, 2002.

[135] M. Kloppmann, D. Koenig, F. Leymann, G. Pfau, A. Rickayzen, C. Von Riegen, and I. Trickovic. WS-BPEL Extension for Sub-processes – BPEL-SPE. *Joint white paper IBM and SAP*, 2006(September):1–17, 2005.

[136] R. Ko, S. Lee, and E. Lee. Business process management (BPM) standards: a survey. *Business Process Management Journal*, 15(5):744–791, 2009.

[137] S. Krakowiak. What is middleware, 2003. URL: http://middleware.objectweb.org/. Date retrieved: July 1, 2011.

[138] J. Kramer. Is abstraction the key to computing? *Communications of the ACM*, 50(4):36–42, Apr. 2007.

[139] P. Kruchten. The 4+1 View Model of Architecture. *IEEE Software*, 12(6):42–50, 1995.

[140] V. R. Lesser. Reflections on the Nature of Multi-Agent Coordination and Its Implications for an Agent Architecture. *Autonomous Agents and MultiAgent Systems*, 1(1):89–111, 1998.

[141] H. Lopes Cardoso and E. Oliveira. Electronic institutions for B2B: dynamic normative environments. *Artificial Intelligence and Law*, 16(1):107–128, 2007.

[142] Z. Ma and F. Leymann. BPEL Fragments for Modularized Reuse in Modeling BPEL Processes. *2009 Fifth International Conference on Networking and Services*, pages 63–68, 2009.

[143] T. W. Malone and K. Crowston. What is coordination theory and how can it help design cooperative work systems? In *Conf on ComputerSupported Cooperative Work CSCW*, number October, pages 357–370. ACM Press, 1990.

[144] A. Martin. Dynamic business process management (BPM) from IBM. Technical report, IBM, 2009.

[145] C. Mayr, U. Zdun, and S. Dustdar. View-based model-driven architecture for enhancing maintainability of data access services. *Data & Knowledge Engineering*, 70(9):794–819, 2011.

[146] N. Mehandjiev and P. Grefen. *Dynamic business process formation for instant virtual enterprises*. Springer-Verlag, New York, 1 edition, 2010.

[147] A. Michlmayr, F. Rosenberg, C. Platzer, M. Treiber, and S. Dustdar. Towards recovering the broken SOA triangle: a software engineering perspective. In *2nd international workshop on Service oriented software engineering*, volume Dubrovnik, of *978-1-59593-723-0*, pages 22–28. ACM, 2007.

[148] B. Naveen, A. N. Chandramohan, A. Chaubal, M. Keen, D. K. Nadgir, M. Sharma, M. Steele, and A. Tost. *Getting Started with IBM WebSphere Business Services Fabric V6.1*. IBM Redbooks, 2008.

[149] T. J. Norman, A. Preece, S. Chalmers, N. Jennings, M. Luck, V. D. Dang, T. D. Nguyen, V. Deora, J. Shao, A. Gray, and N. Fiddian. Agent-based formation of virtual organisations. *Knowledge-Based Systems*, 17(2-4):103–111, 2004.

[150] H. S. Nwana, L. C. Lee, and N. R. Jennings. Coordination in Software Agent Systems. *BT Technology Journal*, 14(4):79–88, 1996.

[151] J. Odell, M. Nodine, and R. Levy. A metamodel for agents, roles, and groups. *Agent-Oriented Software Engineering (AOSE) V*, pages 78–92, 2005.

[152] J. Odell, H. Parunak, and M. Fleischer. The Role of Roles in Designing Effective Agent Organizations. In *Software Engineering for Large-Scale Multi-Agent Systems*, Lecture Notes in Computer Science, Vol. 2603, pages 27–38. Springer, 2003.

[153] D. E. O'Leary, D. Kuokka, and R. Plant. Artificial Intelligence and Virtual Organizations. *Communications of the ACM*, 40(1):52–59, 1997.

[154] O.M.G. The Unified Modeling Language http://www.uml.org/.

[155] O.M.G. Business Process Model and Notation (BPMN). 2010.

[156] A. Omicini, A. Ricci, and M. Viroli. Artifacts in the A&A meta-model for multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 17(3):432–456, May 2008.

[157] A. Omicini and F. Zambonelli. Coordination for Internet Application Development. *Autonomous Agents and Multi-agent systems*, 2(3):251–269, 1999.

[158] F. Oquendo. Pi-ADL: An Architecture Description Language based on the Higher-Order Typed π-Calculus for Specifying Dynamic and Mobile Software Architectures. *ACM SIGSOFT Software Engineering Notes*, 29(3):1–14, 2004.

[159] B. Orriëns, J. Yang, and M. P. Papazoglou. Model driven service composition. *In ICSOC 2003*, pages 75–90, 2003.

[160] M. Ould. *Business processes: modelling and analysis for re-engineering and improvement*. John Wiley & Sons, 1995.

[161] M. Ould. *Business Process Management: A Rigorous Approach*. Meghan-Kiffer Press, 2005.

[162] M. P. Papazoglou. *Web Services: Principles and Technology*. Pearson education. Pearson Education, 2008.

[163] T. Parsons. Suggestions for a Sociological Approach to the Theory of Organizations-I. *Administrative Science Quarterly*, 1(1):63–85, 1956.

[164] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*, volume 24. Prentice-Hall, 1981.

[165] C. Petrie and C. Bussler. The Myth of Open Web Services: The Rise of the Service Parks. *IEEE Internet Computing*, 12(3):96–95, 2008.

[166] M. Pezzini and B. Lheureux. Integration platform as a service: moving integration to the cloud. Technical report, Gartner, Inc., Stamford, CT, 2011.

[167] K. Pfadenhauer, S. Dustdar, and B. Kittl. Challenges and Solutions for Model Driven Web Service Composition. *Information Systems Journal*, pages 126–134, 2005.

[168] J. Pfeffer. *New directions for organization theory: Problems and prospects*. Oxford University Press, USA, 1997.

[169] K. T. Phalp, P. Henderson, R. J. Walters, and G. A. Abeysinghe. RolEnact: role-based enactable models of business processes. *Information and Software Technology*, 40(3):123–133, 1998.

[170] S. Poslad, P. Buckle, and R. Hadingham. The FIPA-OS agent platform: open source for open standards. In *Proceedings of the 5th International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agents*, volume 355, 2000.

[171] F. Puhlmann, A. Schnieders, J. Weiland, and M. Weske. Variability Mechanisms for Process Models. Technical Report 17/2005, DaimlerChrysler Research and Technology, Hasso-Plattner-Institut, 2005.

[172] D. V. Pynadath and M. Tambe. An automated teamwork infrastructure for heterogeneous software agents and humans. *Autonomous Agents and Multi-Agent Systems*, 7(1):71–100, 2003.

[173] D. V. Pynadath, M. Tambe, N. Chauvat, and L. Cavedon. Toward team-oriented programming. *Lecture Notes in Computer Science*, 1757/2000:233–247, 2000.

[174] A. Ricci, M. Piunti, M. Viroli, and A. Omicini. Environment Programming in CArtAgO. *Communication*, pages 259–288, 2009.

[175] A. Ricci, M. Viroli, and A. Omicini. CArtAgO : A Framework for Prototyping Artifact-Based Environments in MAS. In D. Weyns, H. V. D. Parunak, and F. Michel, editors, *Environments for MultiAgent Systems III*, volume 4389 of *LNAI*, pages 67–86. Springer, 2007.

[176] M. Rosemann and W. van der Aalst. A configurable reference modelling language. *Information Systems Journal*, 32(1):1–23, 2007.

[177] F. Rosenberg and S. Dustdar. Business Rules Integration in BPEL — A Service-Oriented Approach. *Seventh IEEE International Conference on ECommerce Technology CEC05*, pages 476–479, 2005.

[178] A. Rushton and S. Walker. *International logistics and supply chain outsourcing: from local to global*. Kogan Page, London, UK, 1 edition, 2007.

[179] D. S. Linthicum. *Enterprise application integration*. Addison-Wesley Longman Ltd., Essex, UK, 2000.

[180] E. Sandberg. *The role of top management in supply chain management practices*. PhD thesis, Linköping University Institute of Technology, Linköping, Sweden, 2007.

[181] W. R. Scott. *Organizations: Rational, Natural, and Open Systems*, volume 8. Prentice Hall, 2003.

[182] F. Shull, J. Singer, and D. I. K. Sjoberg. *Guide to Advanced Empirical Software Engineering*, volume 6. Springer London, 2008.

[183] D. Simchi-Levi, P. Kaminsky, and E. Simchi-Levi. *Designing and managing the supply chain: concepts, strategies, and case studies*, volume 1 of *Irwin/McGraw-Hill series in operations and decision sciences*. Irwin/McGraw-Hill, 2003.

[184] B. Singh and G. L. Rein. Role Interaction Nets (RIN): A process description formalism, 1992.

[185] M. P. Singh. An ontology for commitments in multiagent systems. *Artificial Intelligence and Law*, 7(1):97–113, 1999.

[186] M. P. Singh, A. K. Chopra, and N. Desai. Commitment-Based Service-Oriented Architecture. *Computer*, 42(11):72–79, 2009.

[187] M. P. Singh and M. N. Huhns. *Service-oriented computing: semantics, processes, agents*. John Wiley & Sons, 2005.

[188] T. Skjoett-Larsen, C. Thernoe, and C. Andresen. Supply chain collaboration: Theoretical perspectives and empirical evidence. *International Journal of Physical Distribution & Logistics Management*, 33(6):531–549, 2003.

[189] J. F. Sowa. *Conceptual Structures: Information Processing in Mind and Machine*, volume 33 of *Systems Programming Series*. Addison-Wesley, 1984.

[190] J. M. Spivey. *The Z notation: A reference manual J.M. Spivey. Prentice Hall International, Hemel Hempstead, United Kingdom, 1989, Price*, volume 15 of *International Series in Computer Science*. Prentice-Hall, 1992.

[191] F. Steimann. On the representation of roles in object-oriented and conceptual modelling. *Data & Knowledge Engineering*, 35(1):83–106, 2000.

[192] M. Tambe, D. V. Pynadath, and N. Chauvat. Building Dynamic Agent Organizations in Cyberspace. *IEEE Internet Computing*, 4(2):65–73, 2000.

[193] P. R. Telang and M. P. Singh. Business Modeling via Commitments. *Service-Oriented Computing Agents Semantics and Engineering*, pages 111–125, 2009.

[194] H. Tran, U. Zdun, and S. Dustdar. View-based and Model-driven Approach for Reducing the Development Complexity in Process-Driven SOA. *Intl Working Conf on Business Process and*, pages 105–124, 2007.

[195] H. Tran, U. Zdun, and S. Dustdar. Name-based view integration for enhancing the reusability in process-driven SOAs. *Int J Business Process Integration and Management*, 5(3):229–239, 2011.

[196] I. Trickovic. Modularization and reuse in ws-bpel. Technical report, SAP Developer Network, 2005.

[197] A. Vallecillo. RM-ODP : The ISO Reference Model for Open Distributed Processing. *Management*, 27(8):69–99, 1999.

[198] W. van de Aalst. YAWL: yet another workflow language. *Information Systems Journal*, 30(4):245–275, 2005.

[199] W. van der Aalst. Don't go with the flow: Web services composition standards exposed. *IEEE Intelligent Systems*, 18(1):72–76, 2003.

[200] W. van der Aalst, L. Aldred, M. Dumas, and A. H. M. Ter Hofstede. Design and Implementation of the YAWL System. *Proceedings of the 16th International Conference on Advanced Information Systems Engineering CAiSE04*, 3084:142–159, 2004.

[201] W. van der Aalst, A. Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[202] W. van der Aalst, A. Hofstede, and M. Weske. Business Process Management : A Survey. *Business*, 2678(1):1–12, 2003.

[203] J. Vázquez-Salceda, V. Dignum, and F. Dignum. Organizing Multiagent Systems. *Autonomous Agents and MultiAgent Systems*, 11(3):307–360, 2005.

[204] M. Venkatraman and M. P. Singh. Verifying Compliance with Commitment Protocols. *Autonomous Agents and MultiAgent Systems*, 2(3):217–236, 1999.

[205] G. H. Von Wright. Deontic Logic. *Mind*, 60(237):1–15, 1951.

[206] D. Weyns, R. Haesevoets, B. V. Eylen, A. Helleboogh, T. Holvoet, and W. Joosen. Endogenous versus exogenous self-management. *Proceedings of the 2008 international workshop on Software engineering for adaptive and selfmanaging systems (SEAMS)*, pages 41–49, 2008.

[207] D. Weyns, R. Haesevoets, and A. Helleboogh. The MACODO organization model for context-driven dynamic agent organizations. *ACM Transactions on Autonomous and Adaptive Systems*, 5(4):16:1–16:29, 2010.

[208] D. Weyns, R. Haesevoets, A. Helleboogh, T. Holvoet, and W. Joosen. The MACODO middleware for context-driven dynamic agent organizations. *ACM Transactions on Autonomous and Adaptive Systems*, 5(1):3:1–3:29, Feb. 2010.

[209] D. Weyns, A. Helleboogh, T. Holvoet, and M. Schumacher. The agent environment in multi-agent systems: A middleware perspective. *Multiagent and Grid Systems*, 5(3):1–20, 2009.

[210] J. M. Wing. Computational thinking. *Communications of the ACM*, 49(3), 2006.

[211] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering: An Introduction*, volume 15 of *Software Engineering*. Kluwer Academic Publishers, 2000.

[212] M. Wooldridge. *An introduction to multiagent systems*. JOHN WILEY & SONS, LTD, Chichester, West Sussex, England, 2002.

[213] M. Wooldridge, N. Jennings, and D. Kinny. The Gaia Methodology for Agent-Oriented Analysis and Design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000.

[214] P. Yolum and M. P. Singh. Commitment Machines. *Intelligent Agents VIII*, 9624425:235–247, 2002.

[215] F. Zambonelli, N. Jennings, and M. Wooldridge. Organisational abstractions for the analysis and design of multi-agent systems. In *Agent-Oriented Software Engineering*, pages 407–422. Springer, 2001.

List of Publications

Articles in Internationally Reviewed Scientific Journals

- D. Weyns, R. Haesevoets, and A. Helleboogh. The MACODO organization model for context-driven dynamic agent organizations. *ACM Transactions on Autonomous and Adaptive Systems*, 5(4):16:1–16:29, 2010

- D. Weyns, R. Haesevoets, A. Helleboogh, T. Holvoet, and W. Joosen. The MACODO middleware for context-driven dynamic agent organizations. *ACM Transactions on Autonomous and Adaptive Systems*, 5(1):3:1–3:29, Feb. 2010.

Papers at International Conferences and Symposia, Published in Full in Proceedings

- R. Haesevoets, D. Weyns, M. H. Cruz Torres, A. Helleboogh, T. Holvoet, and W. Joosen. A middleware model in Alloy for supply chain-wide agent interactions. In *Agent Oriented Software Engineering (AOSE)*, volume 6788 of *Lecture Notes in Computer Science*, Toronto, Canada, 2010. Springer.

- R. Haesevoets, D. Weyns, T. Holvoet, and W. Joosen. A formal model for self-adaptive and self-healing organizations. In *Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 116–125. IEEE, 2009.

- R. Haesevoets, E. Truyen, T. Holvoet, and W. Joosen. Weaving the Fabric of the Control Loop through Aspects. In *SOAR: Self-Organizing Architectures*, volume 6090 of *Lecture Notes in Computer Science*, pages 38–65, 2010.

- R. Haesevoets, B. V. Eylen, D. Weyns, A. Helleboogh, T. Holvoet, and W. Joosen. Managing Agent Interactions with Context-Driven Dynamic Organizations. In *Engineering Environment-Mediated Multi-Agent Systems*, volume 5049 of *Lecture Notes in Computer Science*, pages 166–186. Springer-Verlag, 2008.

- D. Weyns, R. Haesevoets, B. V. Eylen, A. Helleboogh, T. Holvoet, and W. Joosen. Endogenous versus exogenous self-management. *Software*

Engineering for Adaptive and Self-Managing Systems (SEAMS), pages 41–49, 2008.

- R. Haesevoets, D. Weyns, T. Holvoet, W. Joosen, and P. Valckenaers. Hierarchical organizations and a supporting software architecture for floating car data. In *Environment-Mediated Coordination in Self-Organizing and Self-Adaptive Systems*, pages 31–36. IEEE, 2008.
- R. Haesevoets, B. Van Eylen, D. Weyns, A. Helleboogh, T. Holvoet, and W. Joosen. Context-driven dynamic organizations applied to coordinated monitoring of traffic jams. In *Engineering Environment-Mediated Multiagent Systems*, pages 126–143, Dresden, Germany, 2007.

Technical Reports

- R. Haesevoets, D. Weyns, and T. Holvoet. A formal specification of an organization model and management model for context-driven dynamic organizations. CW Reports, CW535. Technical report, Department of Computer Science, K.U.Leuven, Heverlee, Belgium, 2009.

Arenberg Doctoral School of Science, Engineering & Technology
Faculty of Engineering
Department of Computer Science
DistriNet Research Group
Celestijnenlaan 200A
B-3001 Heverlee