Architecture-Centric Support for Adaptive Service Collaborations

ROBRECHT HAESEVOETS, IBBT-Distrinet, Katholieke Universiteit Leuven, Belgium
DANNY WEYNS, Linnaeus University, Sweden
TOM HOLVOET, IBBT-Distrinet, Katholieke Universiteit Leuven, Belgium

In today's volatile business environments, collaboration between information systems, both within and across company borders, has become essential to success. An efficient supply chain, for example, requires the collaboration of distributed and heterogeneous systems of multiple companies. Developing such collaborative applications and building the supporting information systems poses several engineering challenges. A key challenge is to manage the ever growing design complexity. In this article, we argue that software architecture should play a more prominent role in the development of collaborative applications. This can help to better manage design complexity by modularizing collaborations and separating concerns. State of the art solutions, however, often lack proper abstractions for modeling collaborations at architectural level or do not reify these abstractions at detailed design and implementation level. Developers, on the other hand, rely on middleware, business process management, and Web services, techniques that mainly focus on low-level infrastructure.

To address the problem of managing the design complexity of collaborative applications, we present Macodo. Macodo consists of three complementary parts: (1) a set of abstractions for modeling adaptive collaborations, (2) a set of architectural views, the main contribution of this article, that reify these abstractions at architectural level, and (3) a proof of concept middleware infrastructure that supports the architectural abstractions at design and implementation level. We evaluate the architectural views in a controlled experiment. Results show that the use of Macodo can reduce fault density and design complexity, and improve reuse and productivity. The main contributions of this article are illustrated in a supply chain management case.

1. INTRODUCTION

Adaptive collaborations between information systems have become an essential element in today's business environments. In modern supply chains, for example, companies rely on a multitude of systems. The arrival of a new order can no longer be handled by a single system, but requires complex collaborations among distributed and heterogeneous systems. In addition, the dynamic and unpredictable market is causing a constant change in supply chain networks, requiring collaborations to be easily adapted to current market needs. One of the primary goals of information technology is to support the collaboration of information systems both within and across company borders [Hugos 2011; Simchi-Levi 2008].

Collaborations can take place in different types of environments [Davidsson 2001]. Open environments, such as the Internet, provide high flexibility, but make it hard to predict the outcome of interactions or to establish mutual trust. Restricted environments take a more pragmatic approach [Petrie and Bussler 2008; Pezzini and Lheureux 2011]. The maintainer of the software deployed in such an environment puts restrictions on the participants. This allows to achieve the necessary stability and trust [Lesser 1998; Cunha 2009], while still allowing selected participants to join. In the domain of supply chain management, third-party companies are providing such environments, by acting as trusted integrators [Rushton and Walker 2007]. Examples are 3PLs and 4PLs (third-party and fourth-party logistics providers), or companies like SupplyOn (www.supplyon.com) and GXS (www.gxs.com).

Developing collaborative applications, even in a restricted environment, and building the supporting information systems, is a complex task that poses several engineering challenges. These challenges include integrating and coordinating distributed and heterogeneous information systems. Although state of the art addresses many challenges, several problems remain open. A key problem is to manage the ever growing design complexity of collaborative applications.

In this article, we argue that software architecture should play a more prominent role in the development of collaborative applications. This can help to better manage design complexity by modularizing complex collaborations and separating concerns. State of the art solutions, however, often lack proper abstractions for modeling collaborations at architectural level or do not reify these abstractions at design and implementation level. Research on middleware, business process management (BPM), and service-oriented architecture (SOA), for example, has mainly focused on individual service interactions, isolated processes, and low-level infrastructure, while paying less attention to the problems of how services collaborate [Singh and Huhns 2005; Pfadenhauer et al. 2005; Ma and Leymann 2009]. Research on role-based techniques, on the other hand, has primarily focussed on the conceptual level, leaving the link between conceptual modeling, software architecture, and implementation implicit [Ferber et al. 2009; Dignum et al. 2009]. We can identify the following problems with state of the art approaches:

(1) *Lack of proper decomposition mechanisms.* State of the art approaches for business process modeling provide limited or no decomposition mechanisms [Erl 2005; Papazoglou 2008], which easily results in monolithic processes that address multiple concerns in a single model [Huhns et al. 2005; Tran et al. 2012]. In particular, most of the existing modeling languages address collaboration management at implementation level and do not provide good mechanisms to reason about collaboration management at a higher level of abstraction [Orriëns et al. 2003; Michlmayr et al. 2007]. Some languages for business process modeling, service orchestration, and choreography [Ma and Leymann 2009; Charfi and Müller 2010] do provide mecha-

nisms to decompose service collaborations, but they are not integrated in the current technology stack.

(2) *Focus on functional decomposition.* State of the art modeling techniques that provide decomposition mechanisms typically rely on the concept of sub-process or composite service [Ma and Leymann 2009; Schumm et al. 2011; OMG 2011]. The use of sub-processes and composite services leads to a functional decomposition [Erl 2005; Papazoglou 2008], which does not reify the underlying collaboration structures. In such a functional decomposition, interaction logic, participant behavior, and management are easily scattered across multiple processes and services. This hampers reasoning about collaborations and their qualities, such as reuse, modifiability, and timing constraints.

(3) *Missing reification of relevant abstractions in design and implementation.* State of the art role-based approaches in object-oriented modeling [Herrmann 2007], agent organizations [Dignum et al. 2009; Dignum and Dignum 2011; Hübner et al. 2011], and BPM [Ould 2005; Caetano et al. 2005] provide useful abstractions to model and modularize collaborations. These approaches, however, primarily focus on the conceptual level and are often hard to map to software architecture, design, or implementation. Most of these techniques require specialized tools and frameworks that are not easily integrated in mainstream software engineering.

To address these problems, we present Macodo[1,2]. Macodo provides architecture-centric support for developing adaptive service collaborations, focussing on service collaborations that take place in a restricted collaboration environment. Macodo consists of three complementary parts:

— *Abstractions to model adaptive collaborations.* The abstractions allow to modularize complex service collaborations and describe collaboration management, interactions, and participant behavior as separate concerns.

— *Architectural views.* The Macodo architectural views, the core contribution of this article, reify the collaboration abstractions as architectural modeling elements. Architects can use the Macodo views to design, document, and reason about collaborations and their qualities in terms of common software elements[3]. Features include modularizing complex service collaborations, while preserving the underlying collaboration structures, and describing collaboration management, interactions, and participant behavior as separate concerns. By mapping collaboration abstractions to common software elements, Macodo provides a close integration with the current technology stack and mainstream software engineering.

— *Proof of concept middleware support.* The middleware provides a concrete platform to develop and implement collaborations that are designed in the architectural views. 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.

---

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

[2] Macodo was initially introduced in [Weyns et al. 2010a; Weyns et al. 2010b]. New contributions of this article, with respect to [Weyns et al. 2010a; Weyns et al. 2010b], are a major revision of the underlying abstractions, the introduction of architectural views for Macodo, the use of Macodo to modularize service collaborations, the integration of Macodo in the Web service technology stack, and the evaluation in an empirical study. An earlier version of the work presented in this article was part of the PhD research of the first author [Haesevoets 2012].

[3] With common software elements, we refer to software modules, components, interfaces, etc. We explain this in detail in section 5.

The main contributions of this article, the architectural views, are evaluated in an empirical study. In this study, performed with a group of 67 final year students of a Master in Software Engineering program, we compare Macodo with a reference approach to design service collaborations taking place in restricted environments. Results show that the use of Macodo can reduce fault density and design complexity, and improve reuse and productivity.

1.1. Overview

Section 2 provides required background information on middleware, BPM, SOA, and software architecture and introduces a supply chain case that is used as running example throughout this article. In Section 3, we discuss state of the art in service collaborations and pinpoint shortcomings of current approaches. Section 4 introduces the collaboration abstractions that provide the foundation for the Macodo architectural views. Section 5 presents the Macodo architectural views. Section 6 discusses the proof of concept middleware for Macodo and a prototype implementation. In Section 7, we report the empirical study of Macodo. Section 8 concludes with a reflection on the main contributions of this article and discusses opportunities for future work.

2. BACKGROUND

To understand the contributions of this paper, some background on middleware, BPM, SOA, and software architecture is required. We start with a brief overview of middleware, BPM, and SOA and highlight some key Web service technologies. Next, we discuss what software architecture is and how it can be documented using architectural views. Finally, we introduce a supply chain management case that we use to illustrate the concepts and contributions of this article.

2.1. Middleware, BPM, and SOA

Middleware offers programming abstractions with supporting infrastructure to facilitate the development of complex distributed systems [Linthicum 2000; Alonso et al. 2004]. These abstractions can hide low-level details of hardware, networks and distribution, and provide developers access to functionality that they would otherwise have to implement from scratch. Enterprise application integration (EAI) extends conventional middleware and addresses the two main concerns of intra-enterprise integration: (1) dealing with heterogeneity, and (2) defining, enacting, and managing the actual application integration and composition logic [Linthicum 2000]. Business process management (BPM) and workflow management (WfM) provide powerful mechanisms to define and execute integration and collaboration logic (e.g., in EAI) [van der Aalst et al. 2003; Ould 2005].

Service-oriented architecture (SOA) addresses several challenges related to integration and collaboration in distributed, but also open, environments [Erl 2005; Papazoglou 2008]. Key principles of SOA are service orchestration and choreography [Erl 2005]. Orchestration allows to combine or compose the functionality of multiple services. Choreography defines a 'public' coordination or integration protocol between services of multiple enterprises. Orchestration and choreography are typically described using workflow-based languages and have revived the use of WfM and BPM. Web services can be seen as another extension to conventional middleware and EAI, providing the technology that allows companies to go from intra-enterprise integration to inter-enterprise integration [Alonso et al. 2004].

2.2. Web Service Technologies

Web services are the most prominent technology stack to realize service-oriented architecture (SOA). Technologies that belong to this stack are often labeled 'WS-*'. We highlight some prominent technologies that are used in the following sections.

Web Service. A Web service is a piece of software that exposes some functionality, or service, and makes this functionality available through standard Web technologies [Alonso et al. 2004]. It has an interface described in a machine-processable format and other systems interact with the Web service as prescribed by its definition using SOAP-messages [Gudgin 2003].

WSDL. WSDL (Web Services Description Language) [Christensen et al. 2001] is an XML-based language to describe the interfaces of a Web service. The interface definition of a single Web service is called a portType. Many interactions, 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 is used, called partnerLinkType [OASIS 2007]. A partnerLinkType defines up to two roles, each linked to a specific portType. An interaction participant then implements one portType and uses the other.

WS-BPEL. Web services can be implemented and exposed using many standard programming languages, but specialized languages, such as WS-BPEL, also exist. WS-BPEL (Business Process Execution Language for Web Services) [OASIS 2007], is an XML-based workflow language for Web services. It is one of the most prominent languages to define and execute service orchestrations and business processes involving Web services. A BPEL process is defined as a sequence of activities (the workflow) which can both use and expose Web services. The concrete interaction points with external entities (such as Web services) are called partnerLinks. A partnerLink corresponds to a specific partnerLinkType, defined in a WSDL definition. Once specified, BPEL processes can be deployed and executed on a BPEL engine (e.g., Apache ODE[4]).

2.3. Software Architecture

Software architecture and its documentation play an important role throughout the lifecycle 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, the relationships between them [Bass et al. 2003], and the relationships with the environment [ISO/IEC 2007]. Software architecture is concerned with managing complexity through abstraction. Software architecture manifests the earliest set of design decisions and provides the main structures to realize both the required functionalities and quality attributes. Architecture documentation serves as a vehicle of communication for stakeholders to get mutual understanding about a system of their interest.

The documentation of software architecture can range from informal sketches to formal notations and 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 [Bass et al. 2003; ISO/IEC 2007]. A view 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 [Kruchten 1995]. Documenting an architecture is a matter of documenting the relevant views and then adding documentation that applies to more than one view [Clements et al. 2010]. Although no fixed set of views is appropriate

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

for every system, there are three common views which allow architects to think about software systems in three different ways [Clements et al. 2010]:

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

Each type of view introduces a set of architectural elements and relations between the elements. We briefly discuss the elements and relations of the Module View and Component & Connector View that are used in the subsequent sections.

The main elements of a module view are *modules*. A module represents an implementation unit of software that provides a coherent set of responsibilities. The actual semantics of modules is only given when they are applied to a specific domain. Concrete examples of modules are a JAR (Java ARchive) file, an OSGi bundle file, a .NET DLL (Dynamic-Link Library file), or an XML specification of an implementation. Three typical types of relations between modules are *'is part of'* (defining a part/whole relationship), *'depends on'* (defining a dependency between modules), and *'is a'* (defining a generalization/specialization).

The elements of a Component & Connector (C&C) View are *components* and *connectors*. Components are the principal computation elements and data stores that execute in a system. Connectors represent runtime pathways of interaction between two or more components. Components and connectors can be composed of interconnected components themselves. Like modules, components and connectors get their semantics when applied to a concrete domain. Examples of components are an executing JavaBean, a running OSGi bundle, a running .NET component, or a BPEL process instance. Examples of common connectors are a persistent data structure, a message queue, a publish-subscribe infrastructure, a message bus, etc. Components have a set of *ports* (the interfaces of a component) through which they interact with other components via connectors. Connectors 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. Two typical relations in the C&C view are *attachments* and *interface delegations*. 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 refine the internal architecture of components and connectors.

One way to look at modules and components/connectors is to see modules as types and components and connectors as instances of these types. However, there is not always a one-to-one mapping of modules to components and connectors. Components and connectors represent run-time elements. Their implementation may be spread over multiple modules, or a single module can also translate to a set of components and connectors at runtime.

2.4. Supply Chain Management: A Running Example

A supply chain consists of a network of companies, such as manufacturers, transporters, warehouses, and retailers, that collaborate to create a product flow from initial supplier to final customer [Chopra and Meindl 2007]. Managing a supply chain includes coordinating production, inventory, and transportation among supply chain partners [Hugos 2011]. Most supply chain companies lack the required knowledge, resources, and capabilities to manage a supply chain on their own. They have to rely on third-party and fourth-party logistics providers (3PLs and 4PLs) [Rushton and Walker 2007]. 3PLs provide vertical supply chain solutions, including warehousing, trans-

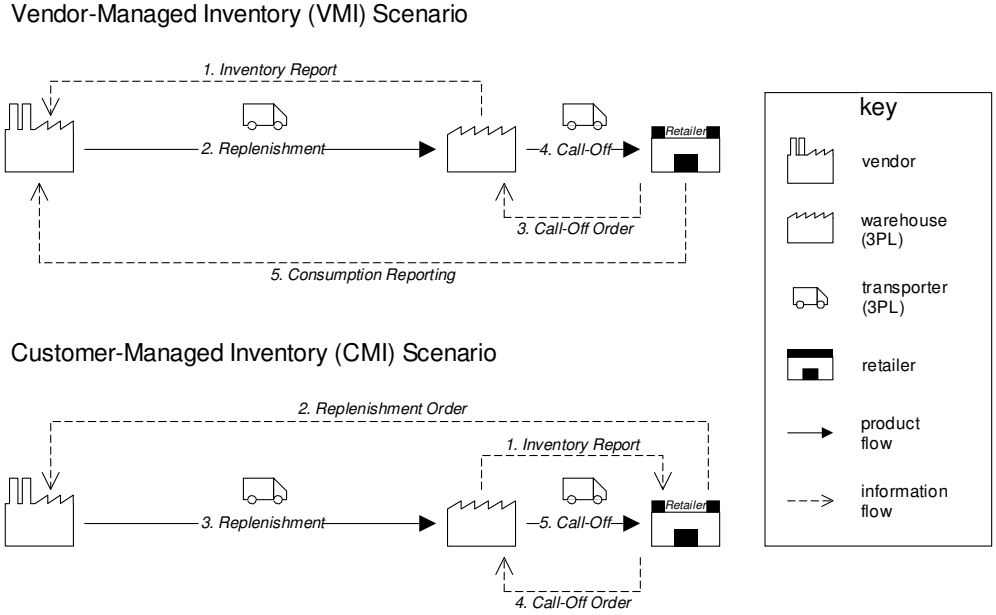


Fig. 1. Two types of supply chain networks to be supported by the 4PL

portation, and other logistics activities. 4PLs operate horizontally across the supply chain, acting as integrators that assemble resources, capabilities and services of different supply chain partners and 3PLs [Christopher 2005]. A key role of a 4PL is to provide a managed collaboration environment that allows to integrate the information systems of all companies involved in the supply chain.

As a running example, we use a 4PL that has to support two types of supply chain scenarios, as shown in Fig. 1. Each scenario involves four companies (vendor, warehouse, retailer, and transporter) that have to collaborate. All communication between the companies passes through the 4PL using standard Web services. The first scenario uses a vendor-managed inventory (VMI), where the vendor is responsible for managing the inventory. Products in the inventory, kept in an intermediate warehouse, remain property of the vendor until consumed, or called-off, by the retailer. The warehouse regularly reports inventory levels to the vendor (flow 1). Based on these inventory levels, the vendor replenishes the warehouse (flow 2). The retailer can call-off products from the warehouse (flow 3-4), after which it reports the consumption to the vendor (flow 5). The second scenario uses a customer-managed inventory (CMI), where the customer manages its own inventory. Products in the inventory, kept in an intermediate warehouse, are already property of the retailer. The warehouse regularly reports inventory levels to the retailer (flow 1). The retailer can request a replenishment from the vendor (flow 2-3), and can call-off products from the warehouse (flow 4-5).

3. STATE OF THE ART

The central contribution of this paper is a set of architectural views that provide better mechanisms to decompose and modularize complex service collaborations. Therefore, we focus our discussion of state of the art on approaches for decomposing and modularizing service collaborations. We can observe two main directions: process-based approaches and role-based approaches. Process-based approaches rely on middleware, BPM, and SOA to develop service collaborations. Role-based approaches use alterna-

tive abstractions, such as roles and organizations. For both directions, we discuss the state of the art and use the running example to pinpoint shortcomings of existing approaches.

3.1. Process-Based Approaches

Business processes and workflows play a key role in engineering collaborative applications and are the foundation of several state of the art techniques. In this section we discuss the most prominent research efforts based on business processes, such as sub-processes, aspect-based approaches, view-based approaches, and commitment-based approaches. The selection of research work is based on their relevance to our work (i.e., the ability to engineer complex service collaborations) and the prominence within their research domains. Our particular focus is on how existing approaches allow to model, decompose, and modularize complex collaborations.

3.1.1. Sub-Processes and Sub-Workflows. Sub-processes and sub-workflows can be used to decompose business processes and workflows. A common approach is workflow or process 'fragmentation' [Adams et al. 2006; Ma and Leymann 2009; Eberle et al. 2009]. A fragment is a reusable piece of process or workflow code, created from scratch or extracted from existing processes, that can be used to compose processes or workflows. Fragmentation can be done by hand or in a semi-automatic manner, in which a modeler is supported by a set of algorithms. Fragments can also be stored in shared libraries to be used for service composition [Schumm et al. 2011]. In current Web service standards these concepts have limited support. WS-BPEL [OASIS 2007] and WS-CDL [Kavantzas et al. 2005] do not provide any support for decomposition. BPMN [OMG 2011], a graphical language for business processes, supports sub-processes and sub-choreographies, which can be used as a visual aid to manage complexity (i.e., collapsing parts of a diagram) or to define reusable process or choreography definitions. Sub-processes have been studied in the context of WS-BPEL [Kloppmann et al. 2005; Trickovic 2005], but have not yet found their way into the BPEL standard or execution platforms.

Sub-processes and sub-workflows provide first-class concepts to decompose processes and workflows in terms of functionality. They do not, however, provide any concepts that reify the underlying collaboration structure. When applying these approaches to our running example, the 4PL can use these techniques to decompose each supply chain collaboration into a set of (sub-)processes. Because of the functional decomposition and the lack of collaboration structure in the decomposition, responsibilities of participants, and even interactions, are easily scattered over multiple processes and services. This makes it hard to maintain or adapt such software. What we need are additional modeling concepts that allow to decompose collaborations while maintaining the underlying collaboration structure.

3.1.2. Aspect-Based Approaches. Aspect-oriented programming (AOP) is a well known technique to support the separation of crosscutting concerns, by modularizing them in aspect modules [Kiczales et al. 1997]. These aspect modules can then be woven into existing code at specific 'join points'. Several authors advocate an aspect-oriented approach for Web service composition [Charfi and Mezini 2004; Courbis and Finkelstein 2004]. A prominent example is AO4BPEL [Charfi and Mezini 2007; Charfi and Müller 2010], an aspect-oriented workflow language in which each BPEL activity is a possible join point. Supporting AO4BPEL at runtime requires the extension of existing BPEL engines. Like AspectJ, however, AO4BPEL focusses on modularization at implementation level and lacks high-level abstractions to structure the architecture of a system.

Similar to sub-processes, applying aspect-based approaches to our running example results in a set of fragments or aspects that encapsulate functional parts of the

collaboration. For example, we could have a base process for a call-off from the warehouse and an aspect encapsulating the functionality for consumption reporting. The aspect would then be triggered after a successful call-off. Although more modular then a monolithic process, such a decomposition fails to represent the underlying collaboration structures.

3.1.3. 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 allow focusing on a single business process or on cross-organizational collaborative workflows. In the context of cross-organizational workflows, views have been used to integrate existing workflows of different organizations, where organizations expose a view on their local or private workflow [Chiu et al. 2002; Chebbi et al. 2006]. [Tran et al. 2007; Tran et al. 2012] 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. Example are the orchestration view, focusing on the internal structures of a process, the collaboration view, focussing on the external services with which a process interacts, and the information view, focussing on the data exchanged by a process. Using model transformations, they can combine views and generate executable BPEL code.

Most view-based approaches do not directly decompose processes or collaborations. Instead, the description of processes is decomposed in different views, focussing on different aspects such as structure, external services, and data, making the documentation of a process more manageable.

3.1.4. Commitment-Based Approaches. Commitment-based approaches [Singh et al. 2009] provide an alternative to traditional process modeling, by treating interactions at the level of 'business semantics' instead of messaging, using commitments. A commitment is a reification of a directed obligation [Singh 1999] and can be compared to a social norm used in multi-agent systems [Dignum 1999]. 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 used to specify business protocols [Desai et al. 2007; Telang and Singh 2012]. Messages are given business meaning by specifying how they affect commitments. Amoeba [Desai et al. 2008] describes a methodology for using commitment-based protocols as building blocks to compose more complex business processes. Amoeba allows to specify how messages of one protocol effect the commitments in another protocol and constrain the order of messages between different protocols.

Applying a commitment-based approach to our running example would result in a specification in terms of commitments of each supply chain partner and how each action affects these commitments. This specification, similar to a service choreography, can then be used by each supply chain partner to implement their local business processes, compliant with this specification. As illustrated by this example, commitment-based approaches are mainly concerned with the specification and validation of cross-organizational business processes. Although commitment-based approaches do bring more business meaning to the process definition, they still lack 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 [Telang and Singh 2012] do introduce additional concepts, such as tasks and goals, but they still represent low-level concepts like individual business activities. What we need are high-level, but reusable, building blocks, such as behaviors and interactions, that can be composed into collaborations. These collaborations should easily translate in to both

implementation units (modules) and runtime software elements (components and connectors).

3.2. Role-Based Approaches

Role-based abstractions provide an interesting alternative to develop collaborative applications. Roles have a rich history in social science [Parsons 1956], organization theory [Carley and Gasser 1995; Pfeffer 1997], and multi-agent systems [Demazeau and Costa 1996; Jennings 2000; Dignum 2009]. Roles are also recognized as an important modeling concept in object-oriented and conceptual modeling [Steimann 2000; Guizzardi 2005] and business process modeling [Ould 2005; Caetano et al. 2005]. Nevertheless, the concept of role has not received the attention it deserves. Additionally, there is no consensus on the definition of roles or how they should be integrated in established modeling frameworks and mainstream programming languages [Steimann 2000; Guizzardi 2005; Herrmann 2007]. This section discusses a selection of the most relevant and prominent role-based techniques in BPM and multi-agent systems.

3.2.1. Roles in Business Process Management. Most BPM 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 and often hierarchical decomposition, in which activities carried out by individual systems or people are scattered throughout one or more models. This makes it hard to abstract away from the details of the process, or to capture the interactions between systems or people who carry out the activities [Phalp et al. 1998; Caetano et al. 2005]. [Ould 2005] calls for collaboration-centric BPM in which a collaboration is a primitive to model processes. BPM systems should support roles and mediate their interactions to make the intended collaboration happen.

A number of researchers have proposed role-based BPM techniques. Prominent examples are Role Interaction Nets [Singh and Rein 1992] (RIN), Role Activity Diagrams (RAD) [Ould 1995], and Riva [Ould 2005], a BPM method based on RAD. RIN and RAD are modeling languages that have a formal underpinning 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 and other roles. In RAD, a process is a coherent set of activities, carried out by a collaborating set of roles to achieve a goal. Roles define a responsibility within a process and are described as a set of actions and interactions with other roles. RAD is used to model an individual process. To combine processes, Riva relies on the notion of process interaction. When processes interact, at least one role is shared. Multiple processes are synchronized by shared states of the role.

When applying RAD or Riva to our running example, we can model the different parts of our scenarios (e.g., call-off and consumption reporting) as individual processes in terms of roles and interactions between these roles. To relate these processes, we can use process interactions. Riva, however, tries to avoid encapsulation or the concept of subprocess. Even with interacting processes, Riva assumes you are modeling the same process but just not showing everything, for example, only the shared roles and states. As a result, there is little focus on reuse. Another challenge is to translate RAD models to a concrete software architecture and implementation. Since the focus of RAD and Riva is on modeling, this translation is left to the developer.

Although roles provide an interesting concept to model processes, in current mainstream BPM, roles only play a minor part. In WS-BPEL [OASIS 2007] roles are used to distinguish the interfaces (or portTypes) defined in partnerLinkTypes. In BPMN [OMG 2011], a 'PartnerRole' can be used to represent a participant in a col-

laboration (i.e., a message exchange between two or more processes) similar to 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 [OMG 2011]. In WS-CDL [Kavantzas et al. 2005], 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.

3.2.2. Roles in Multi-Agent Systems. Organizations and roles are often used to study, design, and engineer multi-agent systems (MASs). In its most general form, an organization can be seen as a cooperation pattern, or process, that constraints the actions and interactions of agents towards some purpose [Castelfranchi 1998]. In a more specific form, an organization can also refer to a collective entity with an explicit identity [Scott 2003]. Prominent agent methodologies have adopted organizations [Demazeau and Costa 1996; Zambonelli et al. 2003], logics and languages exist to describe agent organizations [Dignum and Dignum 2011; Hübner et al. 2011], and several researchers have proposed meta-models to model MAS using organizations. Prominent examples are AGR (Agent/Group/Role) [Ferber and Gutknecht 1998], Electronic Institutions (EIs) [Esteva and Rodríguez-Aguilar 2001; Solaz et al. 2011], Moise [Hannoun et al. 2000; Hübner 2010], and OperA [Dignum et al. 2005].

Many of these approaches rely on multiple dimensions or structures to describe organizations [Ferber et al. 2009]. Three common structures are:

— An organization or social structure that defines the organization in terms of roles and how roles relate.
— A functional or interaction structure that describes the functional aspects of an organization (e.g., in terms of tasks, goals, or interactions).
— A normative structure that defines the rights and obligations of agents in the organization.

These dimensions, however, often remain at a conceptual level, while it is unclear how they can be mapped to common software elements (e.g., modules, components, connectors). Some techniques provide runtime support [Ricci et al. 2009; Hübner et al. 2009; Schmitt et al. 2011], but mainly through specialized frameworks and tools.

To decompose the functional aspects of an organization, most organization models use scenes (e.g., Electronic Institutions and OperA) or goal-decomposition trees (e.g., Moise). Little-Jil [Wise et al. 2000], a language for coordinating agents, uses a technique similar to goal-decomposition trees. Scenes and goal-decomposition trees are often combined with norms, which are social conventions on how agents should behave and interact with each other [Dignum 1999]. Scenes describe possible interactions between roles of an organization. They can be combined in a more complex structure that defines how an organization achieves it goals. This structure can be compared to a global workflow where individual scenes are sub-workflows that describe individual interactions. Norms are used to define which scenes agents can or should execute [Esteva and Rodríguez-Aguilar 2001; Dignum et al. 2005]. The actual functionality of an organization is realized by agents executing scenes.

Goal-decomposition trees focus on the division of tasks, in terms of goals and plans that are assigned to agents, roles, or groups. Interactions are not modeled explicitly, but are the results of one or multiple goals or tasks that make agents interact. Norms are used to define which goals agents can or should realize. 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' [Hübner 2010].

When applying a scene-based approach to our running example, we can model the different parts of our scenarios (e.g., call-off and consumption reporting) as different scenes. Using norms, we can then, for example, define that consumption reporting should take place after a call-off. Applying goal-decomposition trees to our running example requires our scenarios to be translated into a set of goals and norms that define relations between these goals. For example, one goal can be to perform a call-off, which in turn obliges the agent to the goal of reporting the consumption.

Both scenes and goal-decomposition trees, however, do not provide support to properly encapsulate the behaviors and interactions within an organization. Scenes do provide an abstraction for reusable interaction, but do not support encapsulation of individual behaviors, making them hard to reuse. Behavior of agents is partly defined by scenes, partly by scene transitions, and partly by normative rules. Goal-decomposition trees lack proper concepts to model more complex behaviors as well as interactions 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. Although normative rules provide an expressive and fine-grained mechanism to control the behavior of agents [Hübner et al. 2011], from an engineering perspective, designing norms and managing the vast amount of rules that are required to define a complex systems poses some serious challenges.

4. ABSTRACTIONS FOR ADAPTIVE COLLABORATIONS

To support the development of collaborative applications at architectural level, we first need good modeling abstractions. This section presents the Macodo model, a set of role-based abstractions to model collaborative applications. The model focusses on collaborations that take place in a restricted collaboration environment, managed by a trusted party, such as a 4PL. The Macodo model builds upon earlier research results [Weyns et al. 2010a; Haesevoets et al. 2010] and borrows concepts from object-oriented modeling [Steimann 2000; Guizzardi 2005], BPM [Ould 2005; Caetano et al. 2005], and agent organizations [Dignum 2009]. An overview of the Macodo model is shown in Fig. 2. This section describes the key concepts of the Macodo model and illustrates them in the running example.

4.1. Actor

An actor is an entity that has access to the collaboration environment and is capable of participating in collaborations by playing roles. In a concrete system, actors can be business entities, software agents, services, or even people. For example, the concrete supply chain partners are actors in the running example.

4.2. Collaboration

A collaboration is a controlled process, taking place in the collaboration environment, of a group of actors working together towards a set of goals. For example, each supply chain network in the running example can be modelled as a collaboration (*VmiCollaboration* and *CmiCollaboration*). A collaboration consists of a set of roles, representing the different actors and their responsibilities in the collaboration, and a set interactions between the actors of these roles. Collaborations are reusable and can be created and destroyed by the manager of the collaboration. For example, as the manager, a 4PL can create or destroy collaborations in the running example.

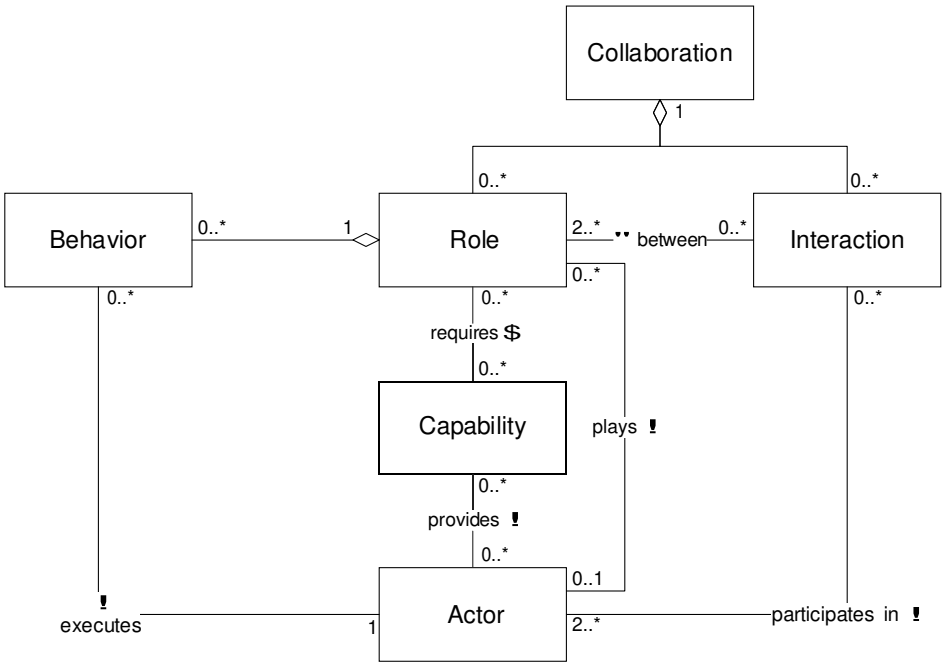


Fig. 2. The key concepts of the Macodo model and their relations.

4.3. Role

A role is the embodiment of the participation of an actor in a collaboration that defines the actor's responsibilities in that participation. When an actor enters a collaboration, a new role is created. When the actor leaves the collaboration, the corresponding role is destroyed. Within the context of a role, an actor can execute behaviors and participate in interactions with other actors in the collaboration. For example, to define the responsibilities in our *Vmi Collaboration* and *Cmi Collaboration*, we can specify a set of roles, such as *Vendor*, *Warehouse*, *Retailer*, and *Transporter*.

4.4. Behavior

A behavior is a coherent unit of reusable functionality that is executed in the context of a role. A behavior is typically application-specific and can encapsulate the execution of a task or the participation in an interaction. The latter is useful when the participation in an interaction is a complex task (e.g., following a complex protocol), or when a task involves participation and synchronization over multiple interactions (e.g., get data from role *X* using interaction *A* and pass it to role *Y* using interaction *B*). In our running example, the roles of our collaborations can be refined in terms of behaviors. For example, we can define a behavior *Inventory Reporting Behavior* for the *Warehouse* role to collect inventory levels and pass it to another role using an interaction.

4.5. Interaction

An interaction is a controlled exchange of information between the actors of a set of roles in a collaboration. An interaction can have an application-specific protocol. For example, to realize the required functionality in our supply chain collaborations, we can define interactions such as the *Inventory Reporting Interaction* (to send inventory

levels to interested parties), the *Call-Off Interaction* (to call-off products from a warehouse), and the *Transport Interaction* (to request a transport from a transporter).

4.6. Capability

A capability is the ability to correctly realize an application-specific functionality. Capabilities are provided by actors. Capabilities are used to describe the requirements to play a role, participate in an interaction, or execute a behavior. A role, for example, requires a specific set of capabilities to be played. Actors can only play those roles for which they provide the required capabilities. In our running example, a capability can be the ability to perform or realize a call-off, to organize a transport, etc.

5. MACODO ARCHITECTURAL VIEWS

The Macodo model provides abstractions to model collaborative applications. At architectural level, however, architects reason in terms of software elements. When reasoning about implementation units in a system, architects use modules, when reasoning about the runtime properties of a system, architects use components and connectors. To use the abstractions of the Macodo model at architectural level, we introduce the Macodo views. The Macodo views are domain-specific architectural views that map the abstractions from the Macodo model to common architectural elements and element relationships. These views allow to design, document, and reason about collaborations and their qualities in terms of software elements, while preserving the underlying collaboration structures.

In the Macodo views we use a one-on-one mapping of modules to components and connectors. Modules can then be seen as reusable types and components and connectors as instances of these types. There are three Macodo views, each taking a different perspective on the collaborations in a system:

— The *Collaboration View* models reusable types of collaborations and their structure. The structure shows how collaborations are modularized and decomposed into reusable units (e.g., roles, behaviors, and interactions).
— The *Collaboration & Actor View* models how collaborations are concretely used, that is, which concrete instances of collaborations are active in the system and which actor is playing which role.
— The *Role & Interaction View* models how a collaboration works in detail (i.e., its internal architecture).

Since Macodo views are specializations of standard views (i.e., Module View, Component & Connector View), they provide a close integration with existing architectural modeling techniques. In fact, architects need to combine Macodo views with traditional views to design and document the architecture of a complete system. The deployment of collaborations, for example, can be described using regular allocation views [Clements et al. 2010].

Table I provides an overview of how Macodo abstractions are mapped to architectural concepts in every view. Since each view takes a different perspective, not all abstractions are present in each view. To make the mapping more clear, we add the word 'module', 'component', or 'connector' to the Macodo terms. When working within a specific view, we typically omit these additional words. For example, in the Collaboration & Actor View, a 'collaboration' and 'collaboration connector' refer to the same concept. The rest of this section discusses the Macodo views in detail and illustrates them in the running example.

Table I. Mapping of Macodo abstractions to architectural concepts.

Macodo Model	Collaboration View	Collaboration & Actor View	Role & Interaction View
Actor	/	Actor Component	/
Collaboration	Collaboration Module	Collaboration Connector	/
Role	Role Module	Connector Role	Role Component
Behavior	Behavior Module	/	Behavior Component
Interaction	Interaction Module	/	Interaction Connector
Capability	Software Interfaces	Runtime Interfaces	Runtime Interfaces

5.1. Collaboration View

The Collaboration View is a Module View that models collaborations as reusable modules and how they are decomposed into reusable submodules (i.e., roles, interactions, and behaviors) (Fig. 3). Each module represents a reusable type of which concrete instances can be created. These instances can be modeled using the Collaboration & Actor View and the Role & Interaction View. The *uses*-relation [Bass et al. 2003] allows to express how a collaboration relies on roles, interactions, and behaviors.

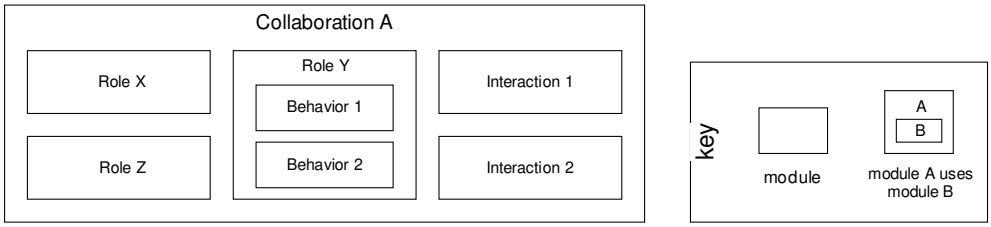


Fig. 3. Types of collaborations, roles, behaviors, and interactions are represented as modules.

Capabilities, which define the requirements to play a role, participate in an interaction, or execute a behavior, translate to software interfaces of role modules, interaction modules, and behavior modules. More specifically, each capability defines a pair of interfaces (Fig. 4): a *provider interface* and a *consumer interface*. A module that requires (or consumes) a capability realizes the consumer interface and uses the provider interface. A module that provides a capability realizes the provider interface and uses the consumer interface. Capabilities allow to group, structure, and reuse the interfaces of roles, interactions, and behaviors.

Usage. The Collaboration View is used to describe the collaborations in a system in terms of implementation units. The view promotes reuse, modularity, and modifiability, by decomposing collaborations into reusable modules. Architects can use this view to identify responsibilities of modules and to express and reason about commonalities and variations among modules.

Example. With the Collaboration View, we can model the *Vmi Collaboration* and *Cmi Collaboration* as reusable modules (Fig. 5). By modularizing these modules, we can also reuse submodules, such as the *Warehouse Role* and *Transporter Role*, and several interactions. In addition, the encapsulation of roles, behaviors, and interactions as separate modules allows improves the modifiability of a collaboration. We can, for example, alter the implementation of the *Inventory Reporting Behavior* without affecting the implementation of the *Inventory Reporting Interaction*.

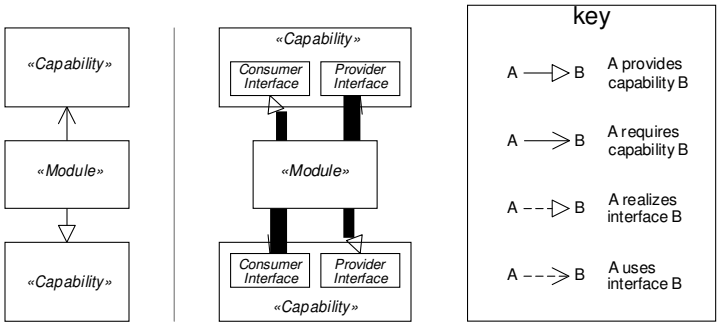


Fig. 4. Capabilities are used to organize, group, and reuse the interfaces of role modules, interaction modules and behavior modules.

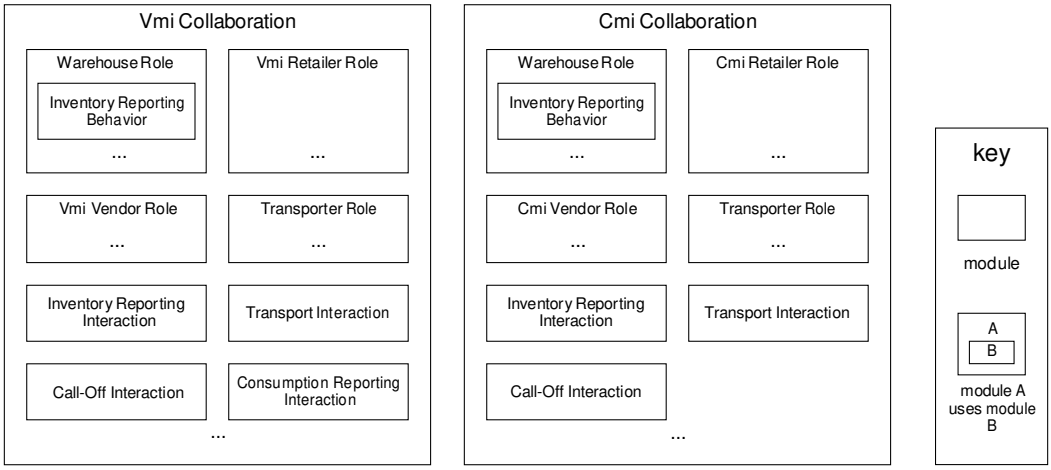


Fig. 5. The Collaboration View allows to model the collaborations of the running example as reusable modules, such *Vmi Collaboration* and *Cmi Collaboration* (dots ("...") indicate the specification is incomplete).

5.2. Collaboration & Actor View

The Collaboration & Actor View is a Component & Connector View that models the actors in a system and the concrete collaboration instances between them. In this view, actors are represented as components, and collaborations as connectors (Fig. 6). Actors have a set of ports (the runtime interfaces of a component), which correspond to a set of provided capabilities. The roles of a collaboration are mapped to connector roles (the runtime interfaces of a connector), which correspond to a required capability. The 'plays' relation of the Macodo model is represented as an attachment between the port of an actor, and a role of a collaboration. Since an actor can only play roles for which it provides the required capabilities, attachments are only valid if the port of the actor provides the capabilities required by the collaboration role.

Usage. The Collaboration & Actor View is used to describe the runtime architecture of a system in terms of actors and the collaborations between them. The view allows to assign responsibilities to actors while making abstraction of collaboration details. Architects can use this view to express and reason about runtime qualities of collaborations in terms of roles and actors, and to describe and document how collaborations and roles are created and destroyed.

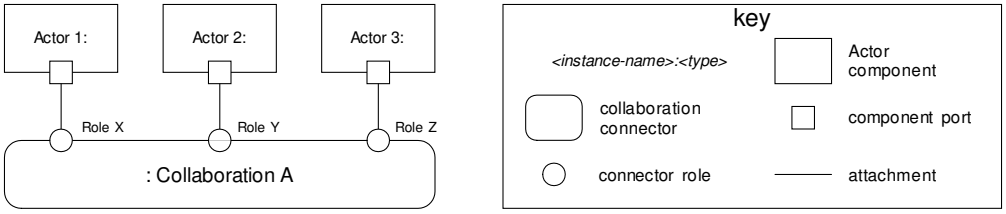


Fig. 6. Actors are represented as components and the concrete collaborations between these actors as connectors.

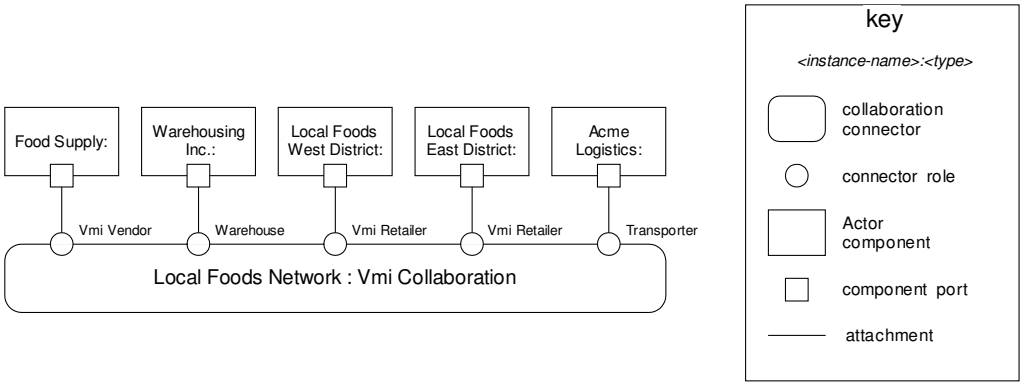


Fig. 7. The Collaboration & Actor View can be used to document the concrete collaborations in our running example.

Example. Using the Collaboration & Actor View, we can model concrete collaborations between actors, such as *Local Foods Network*, which is an instance of the *Vmi Collaboration* (Fig. 7). Each supply company is represented as an actor. *Acme Logistics*, for example, is a company playing the role of *Transporter*. *Acme Logistics* has a port that is attached to the corresponding role of the *Local Foods Network*. We can also use the Collaboration & Actor View to describe additional runtime qualities, by relating qualities or SLAs to specific actors and roles, such as defining a maximum delivery time for the *Transporter* role. These SLAs can be attached to the required capabilities of a role, making sure only actors that can deliver on time will play the role.

5.3. Role & Interaction View

The Role & Interaction View is a Component & Connector View that models the internal runtime architecture of a collaboration in detail. This view allows to document the concrete role and interaction instances in a collaboration, the active behaviors of roles, and how roles delegate the participation in interactions to behaviors. In the Role & Interaction View (Fig. 8), roles are represented as components[5] (shown as vertical lanes), interactions as connectors between roles (shown as horizontal lines), and behaviors as sub-components of roles (shown as blocks within roles).

A behavior is always executed in the context of a specific role, giving the actor of the role access to the interfaces of the behavior. This is modeled by placing the behavior inside the role. For example, in Fig. 8, *Role Y* has two active behaviors (*Behavior 1* and *Behavior 2*). Interactions have a set of connector roles, representing the runtime inter-

[5] Every connector role of a collaboration connector in the Collaboration & Actor View is internally realized by a role component in the Role & Interaction View.

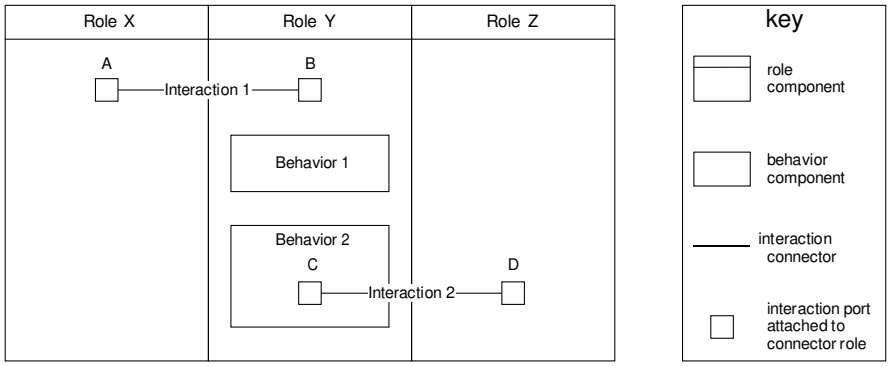


Fig. 8. Roles are represented as components, interactions as connectors between roles, and behaviors as sub-components of roles.

faces of the interaction. Roles have a set of interaction ports, representing the runtime interfaces to participate in interactions. When a role participates in an interaction, there is an attachment between an interaction port of the role and a connector role of the interaction. Connector roles that are attached to interaction ports are shown as small squares on the interactions. For example, in Fig. 8, interaction port *A* of *Role X* is attached to connector role *A* of *Interaction 1*, indicating that *Role X* participates in *Interaction 1*.

For every interaction in which a role participates, the corresponding interaction port can be delegated to the actor of the role (in which case the actor has direct access to the interaction), or to a behavior (in which case the behavior mediates the participation). To indicate that an interaction port is delegated to the actor, the interaction port is placed directly in the role. For example, interaction ports *A*, *B*, and *D* are delegated to the actor of the corresponding roles. To indicate that an interaction port is delegated to a behavior, the interaction port is placed in the behavior. For example, interaction port *C* is delegated to *Behavior2*.

Usage. The Role & Interaction View is used to describe the runtime architecture of a collaboration in detail. The view allows to focus on the internal architecture and runtime qualities of a collaboration without considering how it is used in a specific system. The Role & Interaction View can also be used to document collaboration and role dynamics, and the internal specification of behaviors and interactions. Collaboration and role dynamics are documented in terms of possible interactions between roles, role life-cycles, and possible role states. Behaviors and interactions can be documented using notations such as BPMN or UML sequence diagrams. Complex behaviors and interactions can also be documented in separate views.

Example. With the Role & Interaction Viewwe can model the internal runtime architecture of the *Vmi Collaboration* and *Cmi Collaboration* (Fig. 9). In the *Vmi Collaboration*, for example, we can document the possible interactions between the roles. The *Inventory Reporting Interaction*, for example, takes place between the *Vmi Vendor*, *Warehouse*, and *Vmi Retailer* role. In the *Cmi Collaboration*, the same interaction takes place between the between the *Warehouse* and *Cmi Retailer* role.

We can also document the possible behaviors of roles. The *Warehouse* role, for example, has two behaviors. The *Inventory Reporting Behavior* collects inventory levels and passes it to the inventory reporting interaction. By separating the behavior from the interaction role, we separate two concerns: collection of inventory data, and distribution of inventory data. The *Call-Off Fulfillment Behavior* encapsulates the functional-

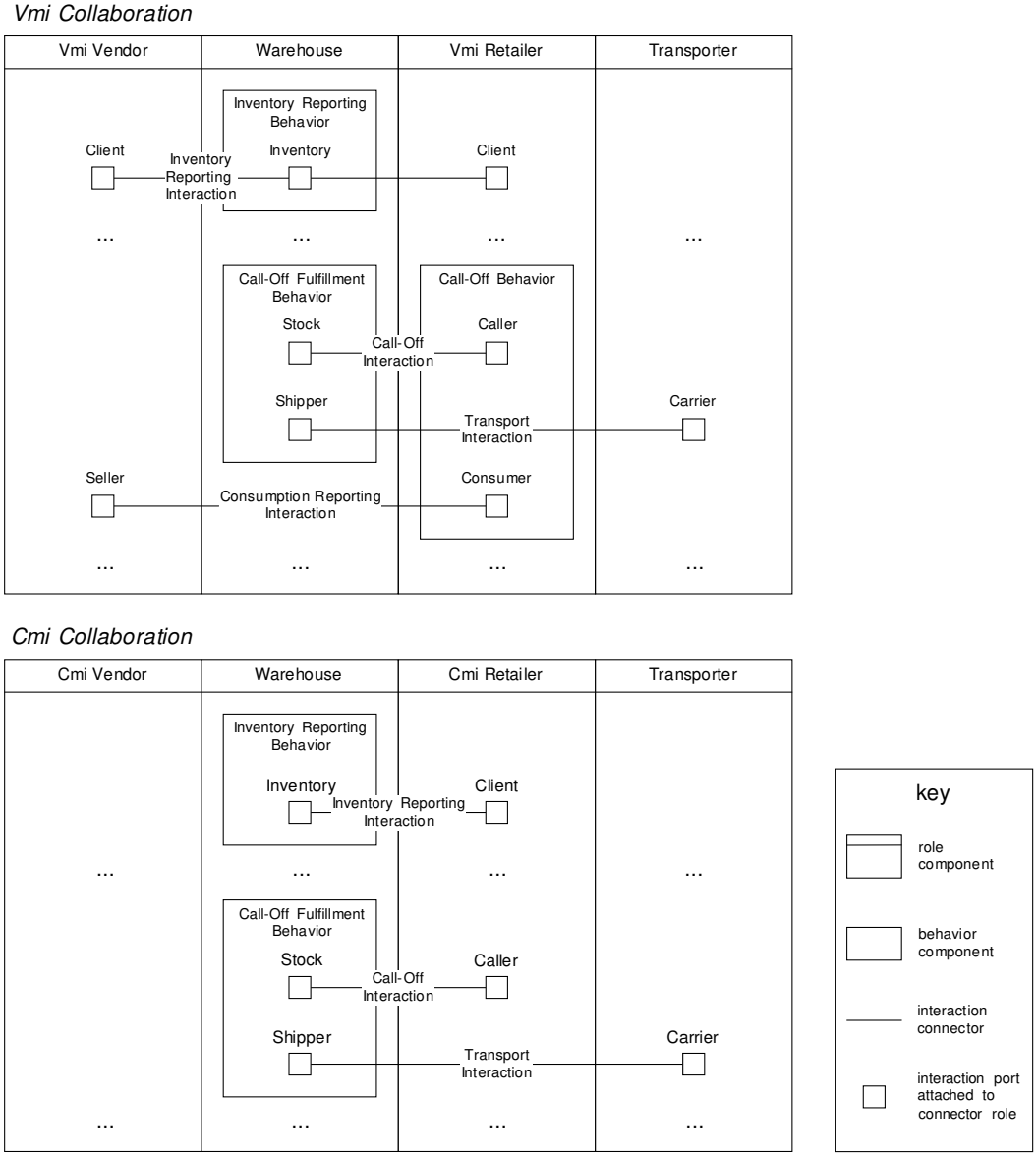


Fig. 9. The runtime architecture of the *Vmi Collaboration* and *Cmi Collaboration* connector using the alternative notation.

ity to initiate a *Transport Interaction* to fulfill the call-off. Other behaviors encapsulate similar functionalities. The *Call-Off Behavior* of *Retailer*, for example, initiates a *Consumption Reporting Interaction* after a successful call-off.

The Role & Interaction View can also be used to model additional runtime qualities, such as throughput of interactions or robustness of behaviors. We can, for example, specify that the *Call-Off Fulfillment Behavior* should always reply to a *Call-Off Interaction*, even if the actor of the *Warehouse* role is not reacting.

Table II. Mapping of Macodo architectural elements to Web service technology.

Architectural Elements	Web Service Technology
Collaboration Module	XML specification
Role Module	XML specification
Behavior Module	XML and BPEL specification
Interaction Module	XML and BPEL specification
Capability	partnerLinkType (WSDL)
Actor Component	External system
Collaboration Connector	Persistent data structure
Role Component	Persistent data structure
Behavior Component	BPEL process
Interaction Connector	BPEL process
Connector Role / Component Port	partnerLink (BPEL)

6. PROOF OF CONCEPT MIDDLEWARE FOR MACODO

The Macodo abstractions and architectural views allow to model and document collaborative applications. Without proper support at downstream design and implementation level, however, the abstractions and views quickly become useless. In this section, we present a proof of concept middleware for Macodo. This middleware provides a platform to design and implement collaborative applications that are modeled in the Macodo architectural views. The platform supports the Macodo abstractions as programming abstractions by mapping them to existing Web service technology. A prototype implementation of the middleware shows that Macodo can be integrated in the current technology stack without the need for new standards. We start by mapping the Macodo architectural elements to existing Web service technology. This provides the foundation of the Macodo middleware. Next, we discuss how to implement, deploy and use collaborations with the Macodo middleware. Finally, we give a brief overview of the middleware architecture and prototype implementation.

6.1. Middleware Mapping

Table II provides an overview of the mapping of the Macodo architectural elements to Web service technology. In this mapping, modules translate to XML and BPEL specifications. Capabilities are mapped to partnerLinkTypes, specified in WSDL, to define a set of 'bi-directional' Web services. Providing and requiring a capability thus translates to the ability to expose and use a set of Web services. At runtime, interaction connectors and behavior components are executed as BPEL processes. BPEL provides good support to model individual behaviors and interactions. Connector roles of interaction connectors and component ports of behavior components become partnerLinks of these BPEL processes. Actor components, collaboration connectors, and role components do not have a direct mapping to Web service technology. Collaboration connectors and role components map to persistent data structures maintained by the Macodo middleware. Actor components are not part of the middleware and become external systems.

An graphical representation of the middleware mapping is shown in Fig. 10. Actor components communicate with interactions and behaviors using SOAP over HTTP, a standard way for Web services to communicate. The middleware uses the persistent data structures, containing the current collaborations and roles, to mediate the information flow between actors, interactions, and behaviors accordingly. Actor components can also use a management service, exposed as a Web service by the Macodo middleware.

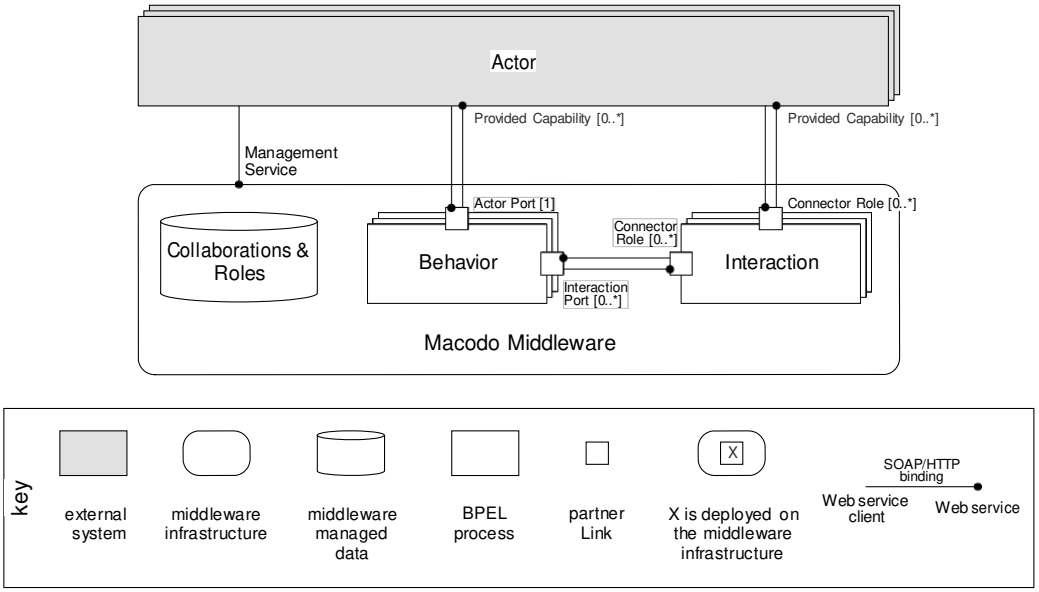


Fig. 10. An informal overview of the mapping of Macodo abstractions to Web service technology.

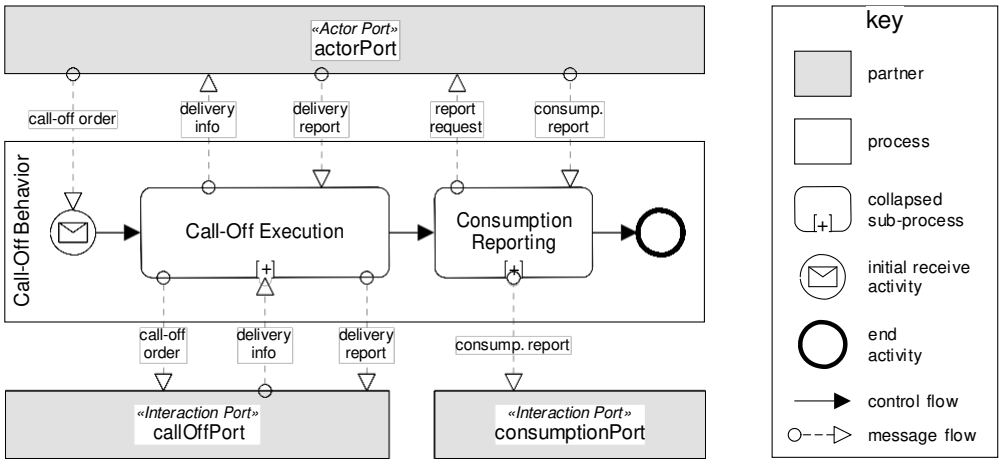


Fig. 11. The BPEL definition of the *Call-Off Behavior*.

6.2. Implementing Collaborations

The Collaboration View (Sect. 5.1) allows to model collaborative applications in terms of implementation units or modules. With the Macodo middleware, each of these modules, can be implemented using XML, WSDL, and WS-BPEL. Interactions and behaviors are specified by an XML file and an accompanying BPEL definition. Roles and collaborations are fully specified by an XML file. Capabilities are specified using WSDL.

Example. The *Call-Off Behavior* can be defined using an XML file and a BPEL definition. The BPEL definition (Fig. 11) provides the specification of the actual behavior in the form of a workflow. Each activity is a step that should be performed when executing the behavior. The XML file defines an actor port (the component port to interact

with the actor of the role) and the interaction ports of the behavior. Each port has a required or provided capability and maps to a corresponding partnerLink of the BPEL definition:

```
<behaviorModule name="CallOffBehavior"
                behaviorSpecification="InvRepBehavior.bpel">
  <actorPort requiredCapability="CallOffCapability"
             partnerLink="ActorPort" />
  <interactionPort name="callOffPort"
                   providedCapability="CallOffCallCapability"
                   partnerLink="callOffPort" />
  <interactionPort name="consumptionPort"
                   providedCapability=
                     "ConsumptionReportingConsumerCapability"
                   partnerLink="consumptionPort" />
</behaviorModule>
```

The *Vmi Retailer Role* can be defined using only an XML file. This file defines the interaction ports, the possible behaviors, and which interaction ports are delegated to behaviors:

```
<roleModule name="VmiRetailer">
  <interactionPort name="inventoryPort"
                   providedCapability="InventoryCapability" />
  <interactionPort name="callOffPort"
                   providedCapability="StockCapability" />
  <interactionPort name="consumptionPort"
                   providedCapability="ShipperCapability" />

  <behavior name="CallOffBehavior"
            behaviorModule="CallOffBehavior" />

  <interfaceDelegation behavior="CallOffBehavior"
                       behaviorInteractionPort="callOffPort"
                       roleInteractionPort="callOffPort" />
  <interfaceDelegation behavior="CallOffBehavior"
                       behaviorInteractionPort="consumptionPort"
                       roleInteractionPort="consumptionPort" />
  ...
</roleModule>
```

6.3. Deploying and Using Collaborations

Once specified, collaboration modules can be loaded in the Macodo middleware. The management service of the middleware (see Fig. 10) can then be used to register actors and to manage the life-cycle of concrete collaboration and role instances. After a role has been assigned to an actor, the actor can 'play' the role. To play a role, an actor uses interactions and behaviors. The information flow between the actors, interactions, and behaviors is mediated by the middleware, which routes messages to the correct interactions, behaviors, and actors (Fig. 12). Messages between the middleware and actors contain additional Macodo data, which uniquely identifies the role to which a message belongs.

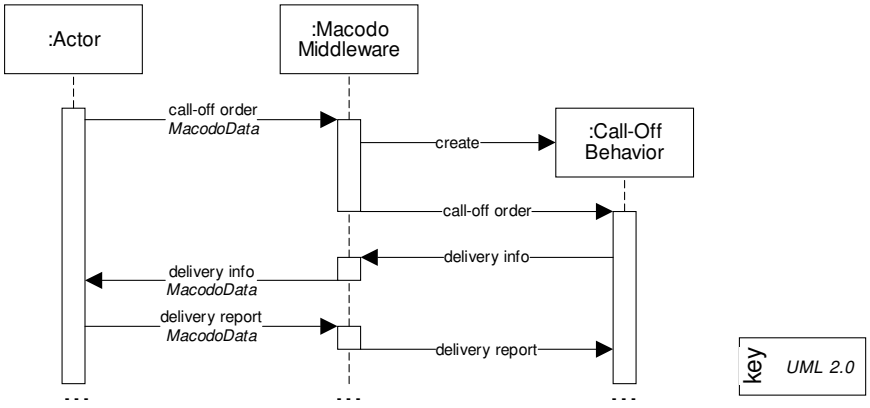


Fig. 12. An example of an actor executing the *Call-Off Behavior*. The middleware creates the behavior and mediates the messages.

6.4. Middleware Realization

A prototype implementation of the Macodo middleware was built using Java EE[6] and Open ESB[7]. An overview of the middleware architecture is given in Fig. 13. Interactions and behaviors are hosted in interaction and behavior containers. These containers, which are deployed on a separate BPEL engine, have three main responsibilities: configuring new interactions and behaviors, managing their life-cycle, and adding Macodo data to all outgoing messages[8]. To realize these responsibilities, the interaction and behavior containers use collaboration mediator components, which are responsible for mediating the interactions between actors, interactions, and behaviors, according to the current collaboration structure.

The collaboration manager components are responsible for managing collaborations and roles. They provide a management service which actors can use to register themselves, and to manage the life-cycle of roles and collaborations. To serve multiple clients in parallel, the collaboration manager and collaboration mediator components run as stateless session beans[9] on a Java EE server. Actors and middleware components interact using normal Web services. This allows to distribute and deploy actors and middleware components on different platforms. The collaboration manager and collaboration mediator components use a set of shared data repositories to store data, communicate, and synchronize their actions. This simplifies synchronization, and allows to replicate and deploy middleware services on multiple servers to improve performance and increase availability.

---

[6] Java EE (Enterprise Edition) provides a platform for developing and running enterprise software. Java EE includes support for distributed and multi-tier architectures, but also Web services.

[7] Open ESB is a Java-based open source enterprise service bus. Open ESB provides support for enterprise application integration and building service-oriented architectures.

[8] The prototype implementation relies on the instrumentation of the BPEL definitions of interactions and behaviors. 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 interactions and behaviors in a container, container logic is instrumented in the original BPEL definitions. The resulting instrumented BPEL definitions are directly deployed on a BPEL engine.

[9] Session beans are objects inside a J2EE server that perform work for clients of the server. Session beans can be exposed as a Web service to clients. A session bean is similar to an interactive session and can only have one client at a time. To serve multiple clients in parallel, session beans can be replicated. In contrast to a stateful session bean, a stateless session bean does not maintain a conversational state for a particular client.

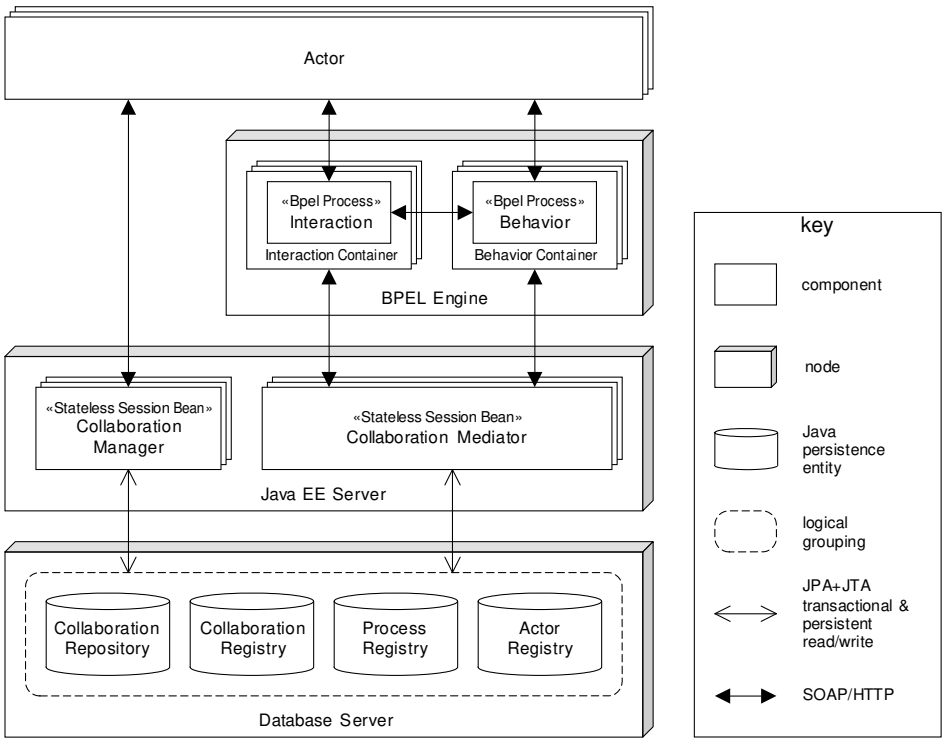


Fig. 13. A high-level overview of the Macodo middleware architecture.

7. EVALUATION: A CONTROLLED EXPERIMENT

We have presented a model, a set of architectural views, and a proof of concept middleware for Macodo. An important question that remains is whether Macodo does improve the development of collaborative applications. To answer this question, we performed an empirical study of Macodo. The goal of the empirical study is to address the following research questions:

> Do the architectural modeling abstractions provided by Macodo: (1) reduce fault density, (2) reduce design complexity, (3) increase the level of reuse, and (4) increase productivity, when designing centrally managed service collaborations?

As reference approach, we use standard SOA design principles and technologies. This reference approach consists of WSDL for Web service description, SOAP for Web service messaging and interaction, UDDI for Web service registry and discovery, WS-BPEL for describing and executing business processes, and BPMN for conceptual modeling of business processes. There are several arguments to motivate this selection as a fair point of reference:

— Each of the selected principles and technologies is considered the de facto standard in their domain [Alonso et al. 2004; Erl 2005; Papazoglou 2008].
— The reference approach not only represents the de facto standard, it also represents the current way of thinking in these domains. Most state of the art techniques build upon these technologies and paradigms [Ma and Leymann 2009; Eberle et al. 2009; Tran et al. 2012].

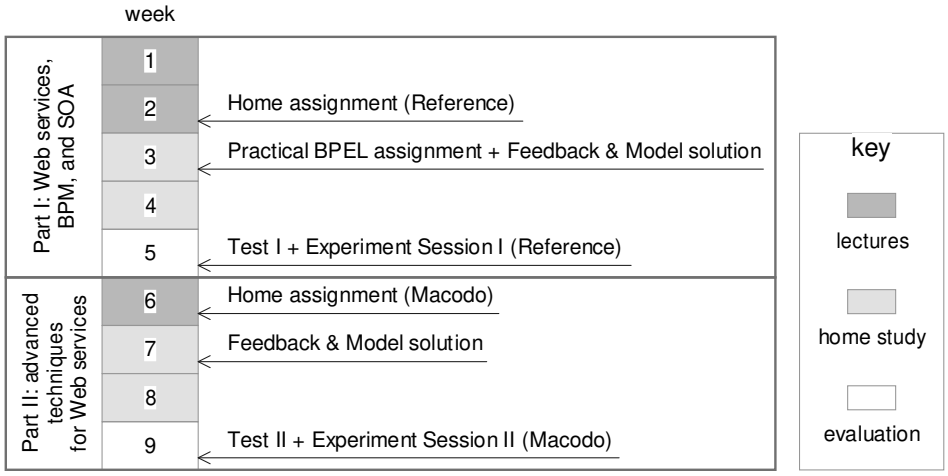


Fig. 14. Overview of the course in which the experiment takes place.

— The selected technologies and principles provide a stable basis for comparison that can be used by other researchers in their evaluation.
— Many evaluations, including the one presented in this section, are realized as 'classroom' experiments, where students have to learn the techniques used in the experiments. The reference approach allows students to learn relevant current techniques.

An alternative would be to select a more recent or novel approach as reference. However, this would make it hard to assess whether this approach is a good representative of the domain and state of the art in general. In addition, the available material and tools for such approaches tend to be too limited to perform a complete evaluation.

The empirical study is based on an extensive pilot study[10] of Macodo. The objects of the empirical study are Macodo and the reference approach. The subjects are 67 final year students of a Master in Software Engineering program from a university in Sweden and two universities in Ukraine. The study takes place as part of a nine-week master course on Web services. The course is split into two parts of five and four weeks (Fig. 14). In part I, students are educated on BPM, Web services, and SOA, which fully covers the reference approach. In part II, students are educated on advanced techniques for Web services, which includes Macodo.

The experiment itself is split into two sessions. In each session (of three hours), subjects receive an assignment to create (using pen and paper) a design (architecture + detailed design) of a system that supports a set of service collaborations, using a specific approach. The experiment is conducted as a block 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 [Campbell and Stanley 1963] 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.

[10] The pilot study was performed with 11 computer science master students, as part of an advanced course on software architectures for distributed systems. Students had to create and document two software architectures, one using the reference approach and standard architectural views, and another using the Macodo architectural views.

The rest of this section discusses the assignments, hypotheses, sample response, measures, analysis, discussion, threats to validity, and conclusions. All the material of the complete study can be found in [Haesevoets and Weyns 2012].

7.1. Assignments

In each experiment session, subjects receive an assignment to complete within three hours. 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.

We use two different assignments (A and B) with equal complexity and required functionality. Having multiple assignments, allows to have subjects solve a different assignment in multiple experiment sessions. Assignment A is based on an eHealth case in which different health care providers have to collaborate on different types of hospital floors. An extract of assignment A can be found in Appendix A. 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/processes, and (2) a detailed design of each module/process using a simplified version of a standard process notation[11]. All deliverables have to be written down using pen and paper on seven provided answering sheets. Section 7.4 discusses a sample response to the assignments.

7.2. 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.

$$H_{01} : \mu_{fault.density_{Ref}} = \mu_{fault.density_{Mac}} \quad (1)$$

$$H_{a1} : \mu_{fault.density_{Ref}} > \mu_{fault.density_{Mac}} \quad (2)$$

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

$$H_{02} : \mu_{complexity_{Ref}} = \mu_{complexity_{Mac}} \quad (3)$$

$$H_{a2} : \mu_{complexity_{Ref}} > \mu_{complexity_{Mac}} \quad (4)$$

[11] We use BPMN, but allow subjects to make abstraction of specific correlation mechanisms and trivial 'assign' activities can be omitted. To ease the notation on paper, a process is divided in swimlanes. Each swimlane represents an external partner of the process. Placing a send or receive activity in such a lane, means the activity sends to or receives from the corresponding partner.

— 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.

$$H_{03} : \mu_{reuse_{Ref}} = \mu_{reuse_{Mac}} \tag{5}$$

$$H_{a3} : \mu_{reuse_{Ref}} < \mu_{reuse_{Mac}} \tag{6}$$

— H_{04}: There is no difference in *productivity* when between the reference approach or using Macodo.

$$H_{04} : \mu_{productivity_{Ref}} = \mu_{productivity_{Mac}} \tag{7}$$

$$H_{a4} : \mu_{productivity_{Ref}} < \mu_{productivity_{Mac}} \tag{8}$$

7.3. Dependent Variables

The experiment measures four dependent variables to test our hypotheses.

Fault Density. We measure fault density as the amount of change that is required to make the design work [Fenton and Pfleeger 1998]. Size is measured as the amount of functionality (function points) that is supported by the design.

$$fault\ density = \frac{\#\ changes}{\#\ supported\ function\ points} \tag{9}$$

Design Complexity. We use two representative measures for complexity: (1) activity complexity (AC) per function point, and (2) average control flow complexity (CFC) per module. The activity complexity (AC) of a module or process is defined as the number of activities in the module or process [Cardoso 2005]. Control flow complexity (CFC) [Cardoso 2006] also takes splits, joins, loops, ending points, and starting points into account.

$$AC\ per\ function\ point = \frac{\sum_{m \in modules} AC(m)}{\#\ supported\ function\ points} \tag{10}$$

$$CFC\ per\ module = \frac{\sum_{m \in modules} CFC(m)}{\#\ modules} \tag{11}$$

Level of Reuse. We use a measure proposed by [Frakes and Terry 1996]. A module is considered reused only if it is used by more than one other procedure or module [Frakes and Terry 1996].

$$level\ of\ reuse = \frac{\sum_{m \in reused\ modules} AC(m)}{\sum_{m \in modules} AC(m)} \tag{12}$$

Productivity. Productivity is defined as the amount of functionality that developers can design per time unit. We measure productivity as follows [Fenton and Pfleeger 1998]:

$$productivity = \frac{\#\ supported\ function\ points}{time\ spend\ on\ design} \tag{13}$$

7.4. Sample Response

Before presenting the actual analysis of the results, we briefly discuss a typical, representative response to the assignments. Appendix B shows the (uncorrected) architec-

ture and detailed design (using the simplified process notation) of subject *x* for assignment *A* using the reference approach (Fig. 17 and Fig. 18) and for assignment *B* using Macodo (Fig. 19 and Fig. 20).

When using the reference approach, the subject modularizes the system into two processes: *MainProcess* and *GeneralPartOfReportInteractionProcess* (Fig. 17). These processes can interact with each other, a set of external partners (*Nurse*, *Head Nurse*, *Doctor on Call*, and *Doctor*), and a service repository (*Repository*). Although the assignment requires specific types of collaborations to be supported, the decomposition of the processes is practically random and lacks any underlying collaboration structure. *MainProcess* handles most of the functionality and *GeneralPartOfReportInteractionProcess* only handles a small non-reusable part. None of the two processes can be considered reused. As a result, the detailed design of the *MainProcess* becomes complex (given the relatively simple assignment) and prone to faults.

When using Macodo, the subject modularizes the system into three types of collaborations: *Basic Line*, *Normal Line*, and *Priority Line* (Fig. 19). These collaborations have a clear mapping to the collaborations required by the assignment. The collaborations are further decomposed into interactions (*Product Interaction*, *Direct Product Interaction*, and *Dispatch Interaction*) and behaviors (*Behavior 1*). Several of these interactions and behaviors are reused across the different types of collaborations. Due to the high level of modularity, the detailed design of each interaction and behavior (Fig. 20), becomes a lot less complex, which reduces the chance for faults.

7.5. Analysis

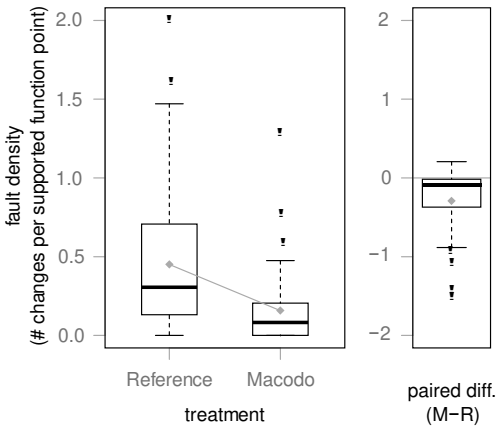
In total, 52 subjects provided usable data for paired comparison[12] of fault density, complexity, and reuse, and 42 subjects provided usable data for paired comparison of productivity. Table III and Fig. 15 describe the measurements for all dependent variables, and the paired difference between the treatments. The paired difference Z_i is defined as:

$$Z_i = M_i - R_i \quad for\ i = 1, \ldots, n \tag{14}$$

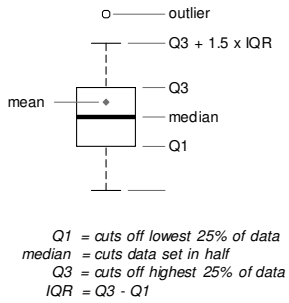
M_i and R_i are the measurements of subject i for respectively Macodo and the reference approach; n is the number of subjects that produced usable data for both treatments. Table III also provides the number of subjects that performed better, equal, or worse for Macodo

To select the proper statistical test for our hypotheses, we compared the distribution of Z_i for each dependent variable with the standard normal distribution, using the Anderson-Darling test [Gibbons and Wolfe 2003]. With a significance level (α) of 0.05, we can accept 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 [Wohlin et al. 2000] to test our hypotheses for 'AC per function point', and 'productivity', and the Wilcoxon signed-rank test [Hollander and Wolfe 1999] to test our hypotheses for the other dependent variables. This results in the following p-values for our hypotheses:

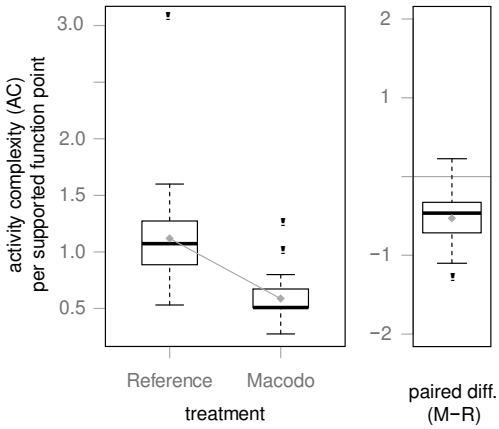
[12] Some subjects provided incomplete or inconsistent data or did not attent both experiment sessions, making their data unusable for paired comparison.



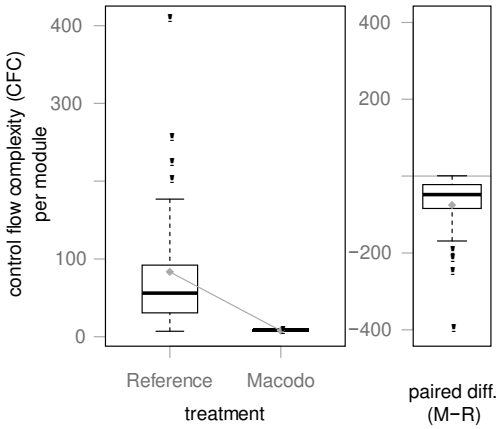
(a) Fault density.



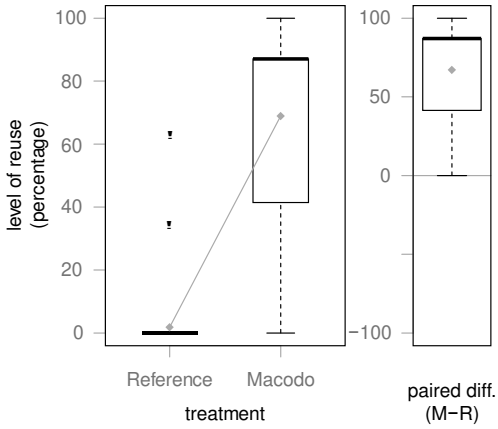
(b) Key.



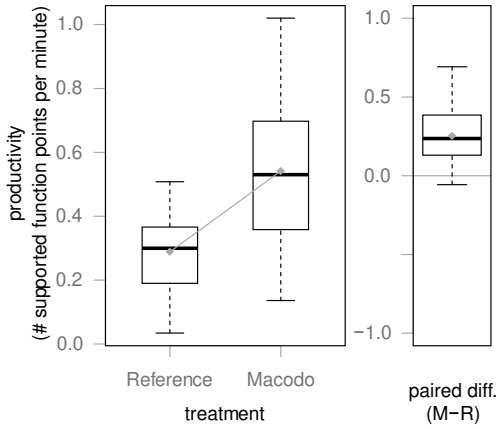
(c) Activity complexity (AC) per function point.



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



(e) Level of reuse.



(f) Productivity.

Fig. 15. Box plots for all measurements.

Table III. Measurements for all dependent variables. 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			
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			
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			
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			

	p-value	statistical test
Fault density	2.80×10^{-08}	Wilcoxon signed-rank test
AC per function point	2.04×10^{-15}	paired-t test
CFC per module	1.76×10^{-10}	Wilcoxon signed-rank test
Level of reuse	1.53×10^{-9}	Wilcoxon signed-rank test
Productivity	1.16×10^{-11}	paired-t test

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

7.6. Discussion

The descriptive analysis shows 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. This is confirmed by the statistical tests, which reject all four null hypotheses with a significance level (α) of 0.05.

There are several explanations for these improvements. Macodo allows to better modularize design, as illustrated in the example response. 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). In combination with a better separation of concerns, and the use of higher-level abstractions, this can reduce the design complexity. This reduction in complexity, has a clear effect on the fault density. Modularization not only effects complexity, it also improves reuse and productivity. In fact, for the

reference approach, only two subjects have any form of reuse, while for Macodo only two subjects have no reuse. Abstractions provided by the reference approach, such as sub-process or composite service, are not well suited to model reusable collaborations. Macodo, however, provides explicit collaboration concepts, and capabilities allow interaction and behaviors to be easily reused in different collaboration and role types.

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. The results underpin that Macodo can be learned in a reasonable amount of time, leading to a clear improvement in developer productivity.

7.7. Threats to Validity

The design of the experiment introduces some threats to validity [Cook and Stanley 1979; Wohlin et al. 2000]. We briefly discuss the main threats.

7.7.1. Threats to Construct Validity. Fault density and complexity are measured in terms of individual modules. Wiring of different modules, for example, is not accounted for. To avoid bias of subjects towards treatments, subjects never have any direct contact with the main author of Macodo, and students are not aware of the experiment. In addition, experiment sessions are executed by external instructors, and students are graded (and aware of this) on both experiment sessions equally.

7.7.2. Threats to Internal Validity. 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 three potential threats:

Increased Understanding. A subject's understanding of certain concepts can increase between the first and the second observation. In each experiment session, subjects have to create an architectural design and a detailed design of individual modules/processes. This threat mainly affects the detailed design, for which subjects use the same standard process notation in both experiment sessions. The architectural design is less affected, since subjects use different techniques in each experiment session. As a result, it is mainly fault density and possibly productivity that are affected by this threat, since the level of reuse and the complexity per module are largely determined in the architectural design and the decomposition of the system into individual processes. To reduce this threat, the students have five weeks to learn the process notation before the first experiment session. They are asked to fully master this notation and are aware that they will be graded on it. After the first experiment session, the course considers the notation to be known and no longer spends any time on it, limiting additional learning effects.

Maturing. Subjects can mature between the two observations, for example, by taking the experiment more serious. This threat is reduced by keeping the subjects unaware of the experiment (during the experiment), but aware that they will be graded on each experiment session.

Learning the type of Questioning. Subjects can learn the type of questioning, making them better prepared in the second experiment session. To reduce this threat, we use a home assignment for each part of the course. This home assignment uses the same questioning as the actual experiment assignments. The home assignment 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.

7.7.3. Threats to External Validity and Conclusion Validity. We use final year 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 [Kitchenham et al. 2002].

The experiment relies on a set of two assignments. To avoid a bias towards Macodo, the assignments were reviewed by two independent course instructors.

In addition, we made a number of decisions out of practical necessity that should be taken into account when generalizing our findings:

— using pen and paper to write down designs;

— allowing subjects to make certain assumptions about the systems and their context;

— using a simplified notation for process definitions;

— using predefined Web services to define external entities.

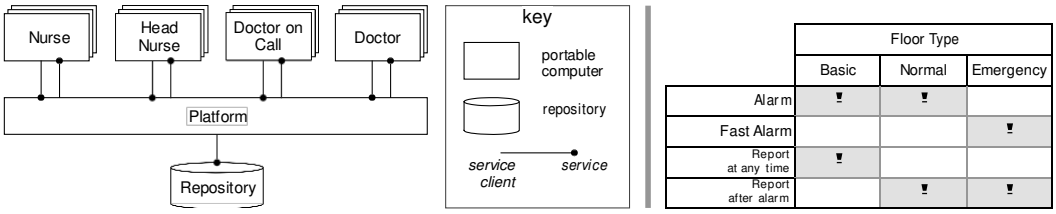
7.8. Conclusions of Study

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 a significant improvement over the reference approach in terms of fault density, design complexity, level of reuse, and productivity.

8. CONCLUSIONS AND FUTURE WORK

In this article, we argued that software architecture should play a more prominent role in the development of collaborative applications. This can help to better manage design complexity by modularizing complex collaborations and separating concerns. State of the art solutions do not provide proper support to model collaborations at architectural level, or do not reify architectural abstractions at downstream design and implementation level. To address these problems, we presented Macodo. Macodo supports the development of collaborative applications at architectural level by introducing a model, a set of views, and a proof of concept middleware. The Macodo model introduces abstractions to decompose complex collaborations into reusable units. The Macodo architectural views reify the Macodo modeling abstractions at architectural level and allow to design, document, and reason about collaborations and their qualities in terms of software elements. The Macodo middleware provides a proof of concept platform to implement collaborations defined using the Macodo architectural views. The evaluation of Macodo, an extensive empirical study, shows that Macodo can indeed improve the architectural support for developing collaborative applications. This improvement is evident from a reduction in fault density and design complexity, and an increase in reuse and productivity.

A number of challenges remain to be solved in order to provide complete architecture-centric support for developing collaborative applications. A key challenge is a better integration of Macodo with existing development techniques and methodologies. Other challenges relate to the current limitations of Macodo. Macodo focusses on service collaborations that take place in a restricted collaboration environment, managed by a trusted third-party. This excludes collaborative applications where there can be no central control or with an explicit need to decentralization. The middleware infrastructure presented in this article is intended as a proof of concept for Macodo. An essential step in providing complete architecture-centric support is to build a mature middleware infrastructure that fully supports Macodo and its features. Finally, an important trade-off of Macodo is that developers need to learn Macodo, the abstractions,



	Floor Type		
	Basic	Normal	Emergency
Alarm	▼	▼	
Fast Alarm			▼
Report at any time	▼		
Report after alarm		▼	▼

Fig. 16. (left) An overview of the platform. (right) 3 types of hospital floors and their properties.

the views, and the middleware. This can steepen the learning curve of Macodo. From the empirical study, however, it is clear that the Macodo basics can be learned in a reasonable amount of time.

APPENDIX

A. EXTRACT OF ASSIGNMENT

This appendix provides an extract of assignment A. Specifications of interactions, predefined Web services, and the repository are not included, but can be found in [Haesevoets and Weyns 2012].

An Orchestration Platform for an Automated Hospital Floor

A hospital has several floors. There are 4 healthcare providers on each floor that have to collaborate: 1 *Nurse*, 1 *Head Nurse*, 1 *Doctor on Call*, and 1 *Doctor*. To streamline the interactions on each hospital floor, the hospital has decided to give each healthcare provider a portable computer that can interact with an orchestration platform (Fig. 16, left), using a set of predefined Web services. The orchestration platform has to orchestrate the interactions between the different healthcare providers.
We consider 3 types of hospital floors (Fig. 16, right) and focus on a subset of interactions to be supported.

— **Basic:** The basic hospital floor enables 2 interactions: the *Alarm Interaction* and the *Report Interaction*. A *Nurse* can start an *Alarm Interaction* at any time, and a *Doctor on Call* can start a *Report Interaction* at any time.
— **Normal:** The normal hospital enables the same interactions as the basic hospital floor (*Alarm Interaction* and *Report Interaction*). But on this floor, the *Doctor on Call* is not free to start a *Report Interaction* at any time. Instead, the platform automatically starts a *Report Interaction* after each *Alarm Interaction* (if the *Head Nurse* confirmed the alarm).
— **Emergency:** The emergency hospital floor is similar to the normal floor (*Report Interaction* is automatically started), but instead of the *Alarm Interaction*, it uses the *Fast Alarm Interaction*.

The platform has to support multiple hospital floors of each type at the same time. For each hospital floor it supports, the platform has to interact with 1 *Nurse*, 1 *Head Nurse*, 1 *Doctor on Call*, and 1 *Doctor*.

The responsibility of the platform can be summarized as follows: when a *Nurse* sends an alarm to the platform, the platform starts and orchestrates an *Alarm Interaction* or a *Fast Alarm Interaction*, depending on the type of hospital floor, with the other healthcare providers of the hospital floor.

If the floor is *Normal* or *Emergency*, the platform automatically starts the *Report Interaction* after the *Alarm Interaction* (if the *Head Nurse* confirmed) or *Fast Alarm*

Interaction. If the floor is *Basic*, the *Doctor on Call* can send a report to the platform at any time, and the platform starts the *Report Interaction*.

To realize this orchestration, the platform has access to a repository, containing all information on the hospital floors and the involved health care providers. This repository can be used to get the other healthcare providers of a hospital floor and to find out the type of hospital floor.

B. SAMPLE RESPONSE

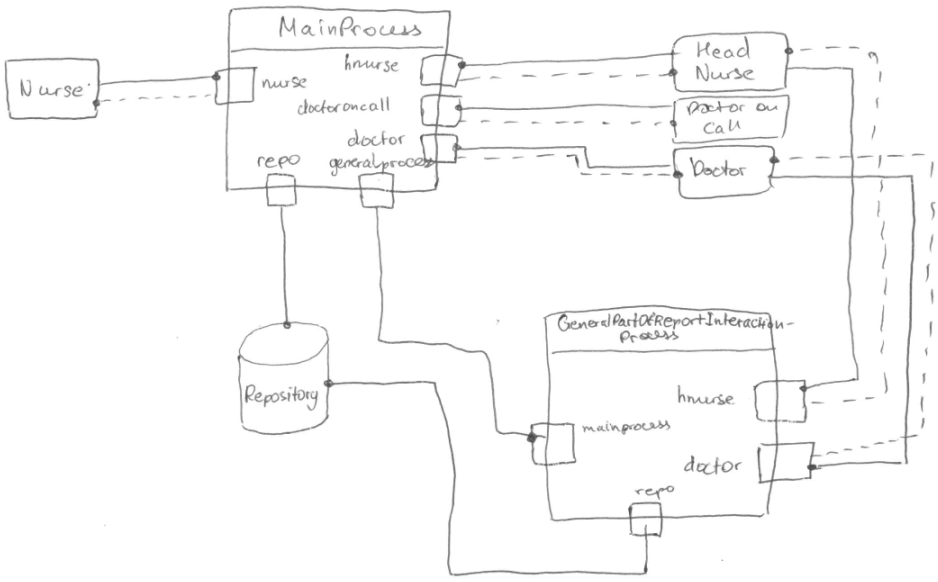


Fig. 17. Subject *x*'s architecture for assignment *A*, using the reference approach.

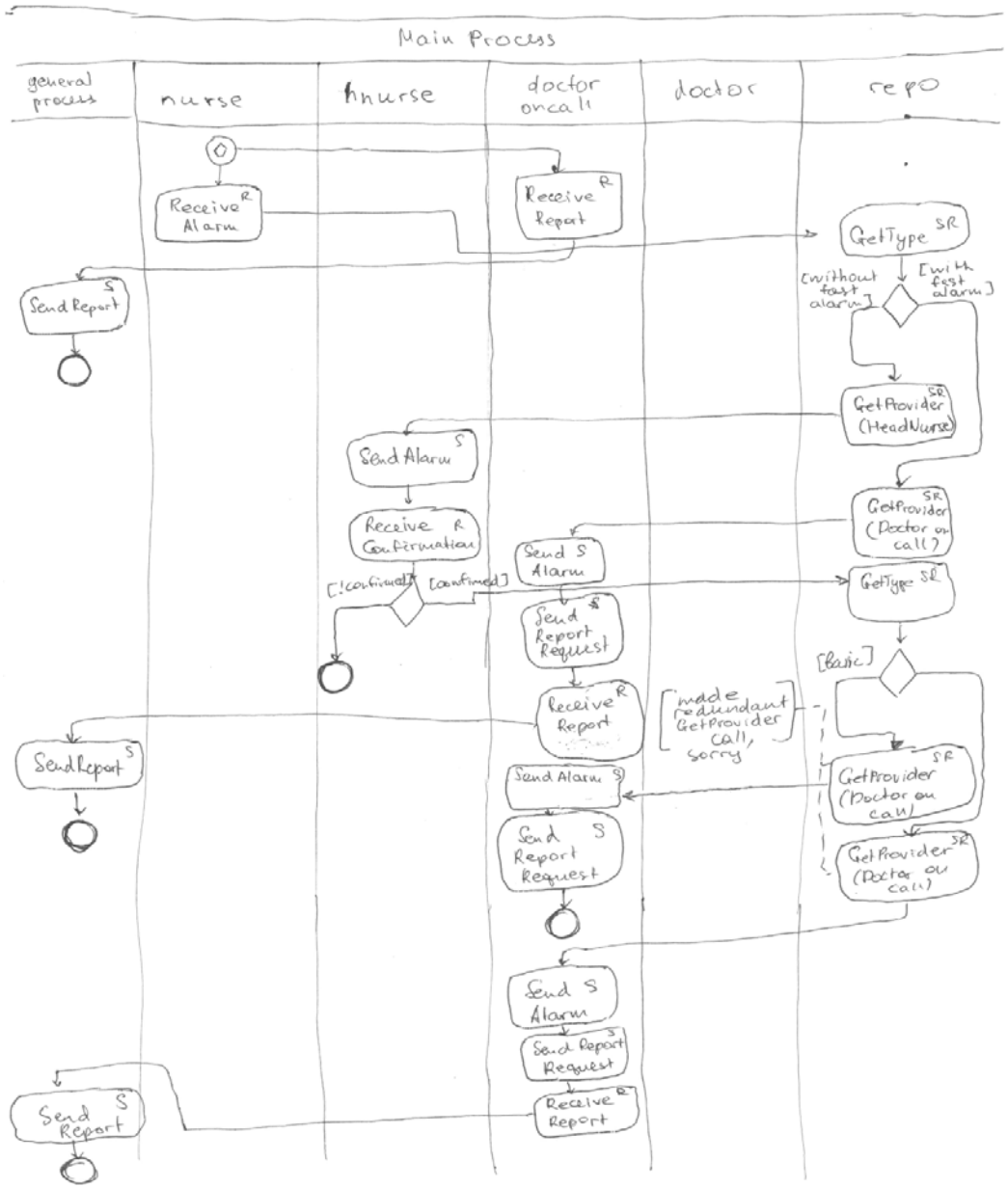


Fig. 18. Part of subject *x*'s detailed design (*Main Process*) for assignment *B*, using the reference approach. Note that the subject uses the simplified notation. Each swimlane represents an external partner of the process. Placing a send or receive activity in such a lane, means the activity sends to or receives from the corresponding partner.

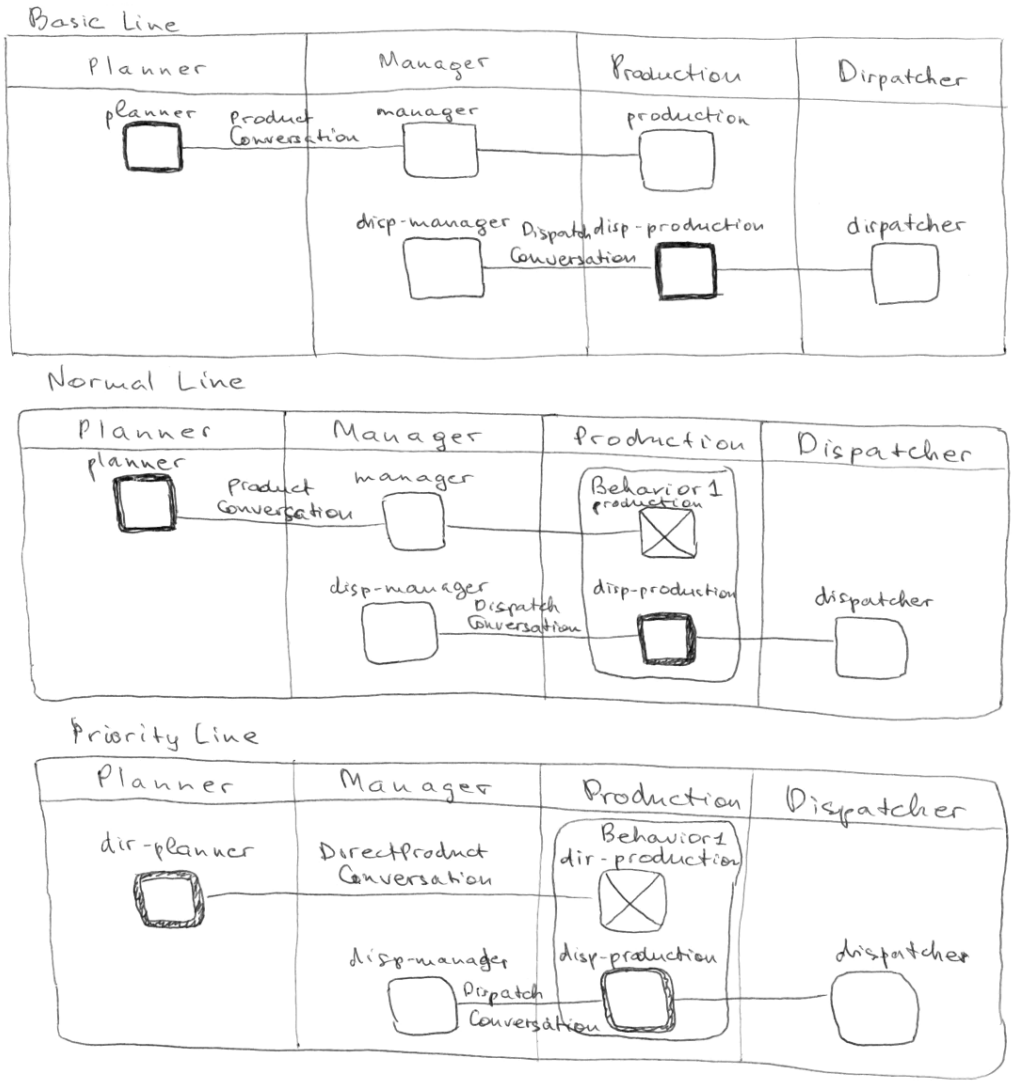


Fig. 19. Subject *x*'s architecture for assignment *B*, using Macodo.

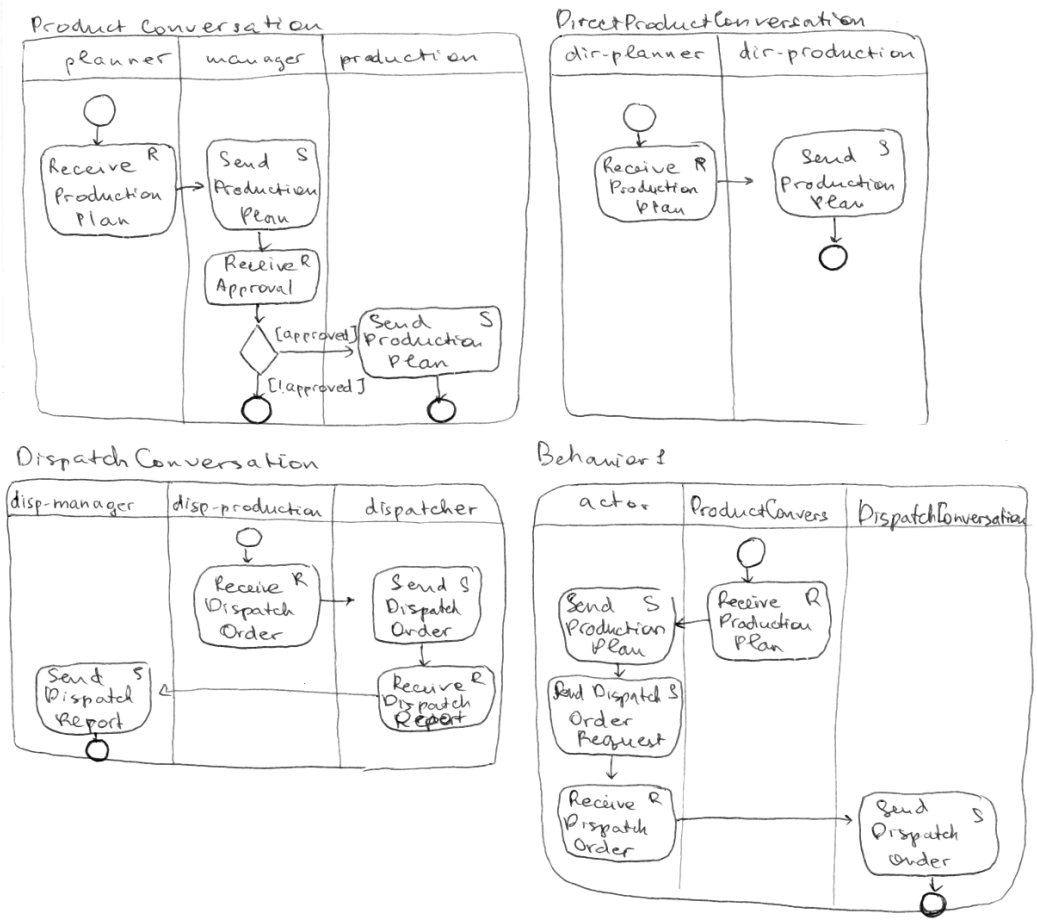


Fig. 20. Subject *x*'s detailed design (using the simplified notation) for assignment *B*, using Macodo.

References

M. Adams, A.H.M. Hofstede, D. Edmond, and W.M.P. van der Aalst. 2006. Worklets: A service-oriented implementation of dynamic flexibility in workflows. *On the Move to Meaningful Internet Systems CoopIS DOA GADA and ODBASE* 4275 (2006), 291–308.

Gustavo Alonso, Fabio Casati, Harumi Kuno, and V Machiraju. 2004. *Web Services: Concepts, Architectures and Applications*. Springer. 354 pages.

L. Bass, P. Clements, and R. Kazman. 2003. *Software Architecture in Practice, 2nd Edition*. Addison Wesley Publishing Comp.

Artur Caetano, Marielba Zacarias, Antonio Rito Silva, and Jose Tribolet. 2005. A Role-Based Framework for Business Process Modeling. In *Proceedings of the 38th Annual Hawaii International Conference on System Sciences*. IEEE.

Donald T. Campbell and Julian C. Stanley. 1963. *Experimental and quasi-experimental designs for research*. Vol. 20. Rand McNally. 84 pages.

Jorge Cardoso. 2005. Control-flow Complexity Measurement of Processes and Weyukerś Properties. *Engineering and Technology* 8, October (2005), 213–218.

Jorge Cardoso. 2006. Process control-flow complexity metric: An empirical validation. In *International Conference on Services Computing*. IEEE Computer Society, 167–173.

M. C. Carley and L. Gasser. 1995. Computational Organization Theory. Routledge, Chapter Social Dil, 201–253.

Cristiano Castelfranchi. 1998. Modelling social action for AI agents. *Artificial Intelligence* 103, 1-2 (1998), 157–182.

Anis Charfi and Mira Mezini. 2004. Aspect-oriented web service composition with AO4BPEL. *Web Services* 3250 (2004), 168–182.

Anis Charfi and Mira Mezini. 2007. AO4BPEL: An Aspect-oriented Extension to BPEL. *World Wide Web Internet And Web Information Systems* 10, 3 (2007), 309–344.

Anis Charfi and H. Müller. 2010. Aspect-Oriented Business Process Modeling with AO4BPMN. *ECMFA* (2010), 48–61.

I. Chebbi, S. Dustar, and S. Tata. 2006. The view-based approach to dynamic inter-organizational workflow cooperation. *Data & Knowledge Engineering* 56, 2 (2006), 139–173.

D.K.W. Chiu, Kamalakar Karlapalem, Qing Li, and Eleanna Kafeza. 2002. Workflow View Based E-Contracts in a Cross-Organizational E-Services Environment. *Distributed and Parallel Databases* 12, 2-3 (2002), 193–216.

Sunil Chopra and Peter Meindl. 2007. *Supply Chain Management: Strategy, Planning and Operation*. Pearson Prentice Hall. 567 pages.

Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. 2001. Web Services Description Language (WSDL) 1.1. (2001). http://www.w3.org/TR/wsdl

M. Christopher. 2005. *Logistics and supply chain management: creating value-added networks*. Pearson Education. 305 pages.

P. Clements, F. Bachman, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, and J. Stafford. 2010. *Documenting Software Architectures: Views and Beyond* (2 ed.). Addison-Wesley Professional. 592 pages.

T.D. Cook and J.C. Stanley. 1979. *Quasi-experimentation: Design & analysis issues for field settings*. Houghton Mifflin Company.

C. Courbis and A. Finkelstein. 2004. Towards an aspect weaving BPEL engine. *The Third AOSD Workshop on Aspects Components and Patterns for Infrastructure Software ACP4IS Lancaster UK* March (2004).

Manuela Cunha. 2009. Environments for Virtual Enterprise Integration. *International Journal of Enterprise Information Systems* 5, 4 (2009), 71–87.

P. Davidsson. 2001. Categories of Artificial Societies. In *Engineering Societies in the Agents World II (ESAW) (Lecture Notes in Artificial Intelligence)*, Vol. 2203. Springer-Verlag, 1 – 9.

Yves Demazeau and A.C.R. Costa. 1996. Populations and organizations in open multi-agent systems.. In *Proceedings of the 1st National Symposium on Parallel and Distributed A.I.* 1–13.

Nirmit Desai, Amit K. Chopra, and Munindar P. Singh. 2007. Representing and Reasoning About Commitments in Business Processes. *Artificial Intelligence* 22, 2 (2007), 1328–1333.

Nirmit Desai, Amit K. Chopra, and Munindar P. Singh. 2008. Amoeba: A methodology for modeling and evolving cross-organizational business processes. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 19, 2 (2008), 1–45.

Frank Dignum. 1999. Autonomous agents with norms. *Artificial Intelligence and Law* 7, 1 (1999), 69–79.

Frank Dignum, Virginia Dignum, Julian Padget, and Javier Vázquez-Salceda. 2009. 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* (2009), 225.

Virginia Dignum. 2009. *Handbook of research on multi-agent systems: semantics and dynamics of organizational models*. Information Science Reference, Hershey, New York, USA. 631 pages.

Virginia Dignum and Frank Dignum. 2011. A Logic for Agent Organizations. *Logic Journal of IGPL* 20, 1 (2011), 220–240 pp.

Virginia Dignum, Frank Dignum, and John Meyer. 2005. An agent-mediated approach to the support of knowledge sharing in organizations. *The Knowledge Engineering Review* 19, 02 (2005), 147–174.

Hanna Eberle, Tobias Unger, and Frank Leymann. 2009. Process Fragments. In *On the Move to Meaningful Internet Systems OTM 2009 Part I (Lecture Notes in Computer Science)*, R Meersman, T Dillon, and P Herrero (Eds.), Vol. 5870. Springer, 398–405.

Thomas Erl. 2005. *Service-Oriented Architecture (SOA): Concepts, Technology, and Design*. Prentice Hall. 792 pages.

Marc Esteva and Juan-Antonio Rodríguez-Aguilar. 2001. On the Formal Specification of Electronic Institutions. *Agent mediated electronic commerce* 1991 (2001), 126–147.

Norman E. Fenton and Shari L. Pfleeger. 1998. *Software Metrics: A Rigorous and Practical Approach, Revised*. Course Technology.

J. Ferber and O. Gutknecht. 1998. A meta-model for the analysis and design of organizations in multi-agent systems. In *Proceedings International Conference on Multi Agent Systems*. IEEE, 128–135.

Jacques Ferber, Tiberiu Stratulat, and John Tranier. 2009. 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 (2009), 1–23.

W. Frakes and C. Terry. 1996. Software reuse: metrics and models. *Comput. Surveys* 28, 2 (1996), 415–435.

Jean Dickinson Gibbons and Douglas A. Wolfe. 2003. Nonparametric Statistical Inference. *Technometrics* (2003), 185–194.

Martin Gudgin. 2003. SOAP Version 1.2. (2003). http://www.w3.org/TR/2007/REC-soap12-part1-20070427/

G. Guizzardi. 2005. *Ontological foundations for structural conceptual models*. Ph.D. Dissertation. University of Twente.

Robrecht Haesevoets. 2012. *Macodo: Architecture-Centric Support for Dynamic Service Collaborations*. Ph.D. Dissertation. Katholieke Universiteit Leuven, Leuven, Belgium. https://lirias.kuleuven.be/handle/123456789/332545.

Robrecht Haesevoets and Danny Weyns. 2012. *Evaluation of Macodo: A controlled experiment*. Technical Report CW 626. Department of Computer Science, Katholieke Universiteit Leuven, Heverlee, Belgium. http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW626.abs.html.

Robrecht Haesevoets, Danny Weyns, M. H. Cruz Torres, Alexander Helleboogh, Tom Holvoet, and Wouter Joosen. 2010. A middleware model in Alloy for supply chain-wide agent interactions. In *Agent Oriented Software Engineering (AOSE) (Lecture Notes in Computer Science)*, Vol. 6788. Springer, Toronto, Canada.

Mahdi Hannoun, Olivier Boissier, Jaime Simão Sichman, and Claudette Sayettat. 2000. MOISE: An organizational model for multi-agent systems. In *IBERAMIA-SBIA*. LNAI, Vol. 1952. Springer-Verlag, 156–165.

Stephan Herrmann. 2007. A precise model for contextual roles: The programming language ObjectTeams/Java. *Applied Ontology* 2, 2 (2007), 181–207.

M. Hollander and D. A. Wolfe. 1999. *Nonparametric statistical methods*. Vol. 2. Wiley-Interscience. 787 pages.

Jomi F. Hübner. 2010. *Moise specifications - draft*. Technical Report. 28 pages. http://moise.sourceforge.net/doc/moise-spec.pdf

Jomi F. Hübner, Olivier Boissier, and Rafael H. Bordini. 2011. A normative programming language for multi-agent organisations. *Annals of Mathematics and Artificial Intelligence* 62, 1-2 (2011), 27–53.

Jomi F. Hübner, Olivier Boissier, Rosine Kitio, and Alessandro Ricci. 2009. Instrumenting multi-agent organisations with organisational artifacts and agents. *Autonomous Agents and Multi-Agent Systems* 20, 3 (April 2009), 369–400.

Michael Hugos. 2011. *Essentials of Supply Chain Management* (3rd ed.). Wiley. 348 pages.

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. 2005. Research directions for service-oriented multiagent systems. *IEEE Internet Computing* 9, December (2005), 65–70.

ISO/IEC. 2007. *ISO /IEC 42010 - Systems and software engineering architectural description*. ISO, Geneva, Switzerland.

N.R. Jennings. 2000. On agent-based software engineering. *Artificial Intelligence* 177, 2 (2000), 277–296.

Nickolas (Oracle) Kavantzas, David (Commerce One) Burdett, Gregory (Novell) Ritzinger, Tony (Choreology) Fletcher, Yves (W3C) Lafon, and Charlton (Adobe Systems Incorporated) Barreto. 2005. *Web Services Choreography Description Language Version 1.0*. Technical Report. W3C. http://www.w3.org/TR/2005/CR-ws-cdl-10-20051109/

G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. 1997. Aspect-Oriented Programming. In *European Conference on Object-Oriented Programming (Lecture Notes in Computer Science, Vol. 1241)*. Springer-Verlag, Berlin, Heidelberg, New York.

B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg. 2002. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering* 28, 8 (2002), 721–734.

Matthias Kloppmann, Dieter Koenig, Frank Leymann, Gerhard Pfau, Alan Rickayzen, Claus Von Riegen, and Ivana Trickovic. 2005. WS-BPEL Extension for Sub-processes BPEL-SPE. *Joint white paper IBM and SAP* 2006, September (2005), 1–17.

P. Kruchten. 1995. The 4+1 View Model of Architecture. *IEEE Software* 12, 6 (1995), 42–50.

Victor R. Lesser. 1998. Reflections on the Nature of Multi-Agent Coordination and Its Implications for an Agent Architecture. *Autonomous Agents and Multi-Agent Systems* 1, 1 (1998), 89–111.

David S. Linthicum. 2000. *Enterprise application integration*. Addison-Wesley Longman Ltd., Essex, UK. 400 pages.

Zhilei Ma and Frank Leymann. 2009. BPEL Fragments for Modularized Reuse in Modeling BPEL Processes. *2009 Fifth International Conference on Networking and Services* (2009), 63–68.

A. Michlmayr, F. Rosenberg, C. Platzer, M. Treiber, and S. Dustar. 2007. Towards recovering recovering the broken SOA triangle: a software engineering perspective. In *2nd international workshop on Service oriented software engineering*. ACM, 22–28.

OASIS. 2007. *Web Services Business Process Execution Language (WS-BPEL) Version 2.0*. OASIS (Organization for Advancement of Structured Information Standards). http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

OMG. 2011. *Business Process Model and Notation (BPMN) version 2.0*. OMG (Object Management Group). http://www.omg.org/spec/BPMN/2.0/

Bart Orriëns, Jian Yang, and Mike P. Papazoglou. 2003. Model driven service composition. In *ICSOC*. Springer-Verlag, 75–90.

M. Ould. 1995. *Business processes: modelling and analysis for re-engineering and improvement*. John Wiley & Sons. 224 pages.

M. Ould. 2005. *Business Process Management: A Rigorous Approach*. Meghan-Kiffer Press.

M. P. Papazoglou. 2008. *Web Services: Principles and Technology*. 752 pages.

T. Parsons. 1956. Suggestions for a Sociological Approach to the Theory of Organizations-I. *Administrative Science Quarterly* 1, 1 (1956), 63–85.

Charles Petrie and Christoph Bussler. 2008. The Myth of Open Web Services: The Rise of the Service Parks. *IEEE Internet Computing* 12, 3 (2008), 96–95.

Massimo Pezzini and Benoit Lheureux. 2011. *Integration platform as a service: moving integration to the cloud*. Technical Report. Gartner, Inc., Stamford, CT.

Konrad Pfadenhauer, Schahram Dustdar, and Burkhard Kittl. 2005. Challenges and Solutions for Model Driven Web Service Composition. *Information Systems Journal* (2005), 126–134.

J. Pfeffer. 1997. *New Directions for Organization Theory: Problems and Prospects*. Oxford University Press. viii, 264p. pages.

Keith T. Phalp, P. Henderson, R. J. Walters, and G. A. Abeysinghe. 1998. RolEnact: role-based enactable models of business processes. *Information and Software Technology* 40, 3 (1998), 123–133.

Alessandro Ricci, Michele Piunti, Mirko Viroli, and Andrea Omicini. 2009. Environment Programming in CArtAgO. *Communication* (2009), 259–288.

Alan Rushton and Steve Walker. 2007. *International logistics and supply chain outsourcing: from local to global* (1 ed.). Kogan Page, London, UK. 424 pages.

Pierre Schmitt, Cédric Bonhomme, Jocelyn Aubert, and Benjamin Gâteau. 2011. Programming Electronic Institutions with Utopia. In *Information Systems Evolution*, Will Aalst, John Mylopoulos, Norman M. Sadeh, Michael J. Shaw, Clemens Szyperski, Pnina Soffer, and Erik Proper (Eds.). Lecture Notes in Business Information Processing, Vol. 72. Springer Berlin Heidelberg, 122–135.

David Schumm, Dimka Karastoyanova, Frank Leymann, and Steve Strauch. 2011. Fragmento: Advanced Process Fragment Library. In *Information Systems Development*, Jaroslav Pokorny, Vaclav Repa, Karel Richta, Wita Wojtkowski, Henry Linger, Chris Barry, and Michael Lang (Eds.). Springer New York, 659–670.

W. Richard Scott. 2003. *Organizations: Rational, Natural, and Open Systems*. Vol. 8. Prentice Hall. 340 pages.

D. Simchi-Levi. 2008. *Designing and Managing the Supply Chain: Concepts, Strategies and Case Studies* (3rd ed.). McGraw-Hill. 528 pages.

B. Singh and Gail L. Rein. 1992. Role Interaction Nets (RIN): A process description formalism. (1992).

Munindar P. Singh. 1999. An ontology for commitments in multiagent systems. *Artificial Intelligence and Law* 7, 1 (1999), 97–113.

Munindar P. Singh, Amit K. Chopra, and Nirmit Desai. 2009. Commitment-Based Service-Oriented Architecture. *Computer* 42, 11 (2009), 72–79.

Munindar P. Singh and Michael N. Huhns. 2005. *Service-oriented computing: semantics, processes, agents*. John Wiley & Sons.

Mario Solaz, Bruno Gui, Juan Rodrguez-Aguilar, Vicente Inglada, and Carlos Casamayor. 2011. Mixing Electronic Institutions with Virtual Organizations: A Solution Based on Bundles. In *Highlights in Practical Applications of Agents and Multiagent Systems*, Javier Prez, Juan Corchado, Mara Moreno, Vicente Julin, Philippe Mathieu, Joaquin Canada-Bago, Alfonso Ortega, and Antonio Caballero (Eds.). Advances in Intelligent and Soft Computing, Vol. 89. Springer Berlin / Heidelberg, 143–150.

F. Steimann. 2000. On the representation of roles in object-oriented and conceptual modelling. *Data & Knowledge Engineering* 35, 1 (2000), 83–106.

Pankaj R. Telang and Munindar P. Singh. 2012. Comma : A Commitment-Based Business Modeling Methodology and its Empirical Evaluation. In *AAMAS*. 4–8.

Huy Tran, Uwe Zdun, and Schahram Dustdar. 2007. View-based and Model-driven Approach for Reducing the Development Complexity in Process-Driven SOA. *Intl Working Conf on Business Process and* (2007), 105–124.

Huy Tran, Uwe Zdun, Talid Holmes, Ernst Oberortner, Emmanuel Mulo, and Schahram Dustdar. 2012. Compliance in service-oriented architectures: A model-driven and view-based approach. *Information and Software Technology* 54, 6 (2012), 531–552.

I. Trickovic. 2005. *Modularization and reuse in ws-bpel*. Technical Report. SAP Developer Network.

W.M.P. van der Aalst, A.H.M. Hofstede, and Mathias Weske. 2003. Business Process Management: A Survey. *Business* 2678, 1 (2003), 1–12.

Danny Weyns, Robrecht Haesevoets, and Alexander Helleboogh. 2010a. The MACODO organization model for context-driven dynamic agent organizations. *ACM Transactions on Autonomous and Adaptive Systems* 5, 4 (2010), 16:1–16:29.

Danny Weyns, Robrecht Haesevoets, Alexander Helleboogh, Tom Holvoet, and Wouter Joosen. 2010b. The MACODO middleware for context-driven dynamic agent organizations. *ACM Transactions on Autonomous and Adaptive Systems* 5, 1 (February 2010), 3:1–3:29.

A. Wise, A. G. Cass, B. Staudt Lerner, E. K. McCall, L. J. Osterweil, and Jr. S. M. Sutton. 2000. Using Little-JIL to Coordinate Agents in Software Engineering. In *Proceedings of the Automated Software Engineering Conference ASE 2000 Grenoble France*. IEEE Comput. Soc, 155–163.

C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. 2000. *Experimentation in Software Engineering: An Introduction*. Software Engineering, Vol. 15. Kluwer Academic Publishers. 228 pages.

F. Zambonelli, N. Jennings, and M. Wooldridge. 2003. Developing Multiagent Systems: The Gaia Methodology. *ACM Transactions on Software Engineering and Methodology* 12, 3 (2003), 317–370.