Die approbierte Originalversion dieser Diplom-/ Masterarbeit ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

TU UB WIEN Universitätsbibliothek

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

A Human Architecture Implementation Framework

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Sofware Engineering & Internet Computing

eingereicht von

Xiaolin Zhang, BSc
Matrikelnummer 0825548

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ. Prof. Mag.rer.soc.oec. Dr.rer.soc.oec. Schahram Dustdar
Mitwirkung: Univ. Ass. Mag.rer.soc.oec. Dr.techn. Christoph Mayr-Dorn

Wien, 13. Juli 2016

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

A Human Architecture Implementation Framework

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Xiaolin Zhang, BSc

Registration Number 0825548

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ. Prof. Mag.rer.soc.oec. Dr.rer.soc.oec. Schahram Dustdar
Assistance: Univ. Ass. Mag.rer.soc.oec. Dr.techn. Christoph Mayr-Dorn

Vienna, 13. Juli 2016

(Signature of Author) (Signature of Advisor)

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

Erklärung zur Verfassung der Arbeit

Xiaolin Zhang, BSc
Zwinzstrasse 4-6/1/12, 1160 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

____________________	____________________
(Ort, Datum)	(Unterschrift Verfasser)

Abstract

With increasing level of interconnections between software users due to the ubiquity of the web, software systems are forced to include and support emergent interaction patterns to enable effective and efficient collaboration between participants, that are humans and software agents. Numerous tools exist today to simplify collaborative undertakings and integrating them into software systems imposes a challenge to software architects and developers. Further more, collaborative tools are subject to fast changes thus human-intensive software systems have to be highly adaptable.

To this end, the human architecture implementation framework is proposed. It offers structural architectural documentation via hADL and can be combined with behavioural meta-models such as process description languages to provide composite means to prescribe and describe software architectures. The framework includes a runtime which offers process engine integration based on a resource model and manages architectural elements prescribed via hADL using surrogates (proxies). Additionally an architecture-driven software development approach is inherently enforced.

The evaluation of the framework includes a prototype implementation of an interest-based bargaining process. A prosaic assessment of the framework is conducted and experiences during development are shared.

Kurzfassung

Die stetige Steigerung des digitalen Vernetzungsgrades in der Gesellschaft führt unweigerlich zur Emergenz von innovativen gemeinschaftlichen Arbeitsweisen zwischen Mensch und Maschine. Softwaresysteme müssen diese neuen Strukturen und Verbindungen unterstützen damit die Sinnhaftigkeit der Systeme nicht in Frage gestellt wird und neue Unternehmungen effektiv und effizient durchgeführt werden können.

Unzählige digitale Werkzeuge existieren bereits um die Kooperation von Mensch zu Mensch, Mensch zu Maschine und Maschine zu Maschine zu vereinfachen. Diese sind jedoch ständigen Veränderungen ausgesetzt, weshalb Integrationslösungen einen gewissen Grad an Anpassungsvermögen mitbringen müssen. Dadurch ist die Entwicklung von Software in diesem Zusammenhang kompliziert und komplex.

Die vorliegende Arbeit schlägt eine organisatorische und technische Herangehensweise für die Entwicklung von Softwaresystemen, welche unter Anderem eine Vielzahl von menschlichen Akteuren verbindet, vor. Dazu wird hADL zur Beschreibung der Struktur eingesetzt, welche zusätzlich in Kombination mit einer Prozessbeschreibungssprache die architektonische Dokumentation des Systems darstellt. Ferner wird eine Laufzeitumgebung für die Verwaltung von Ressourcen, die mittels hADL deklariert werden, vorgestellt.

Zuerst wird das Design der Lösung beschrieben, anschließend wird sie anhand der Entwicklung eines beispielhaften Geschäftsprozesses ausgewertet. Die Evaluierung und kritische Würdigung erfolgen zuletzt als textuelle Argumentation.

Contents

List of Figures viii

List of Tables x

Listings x

1 Introduction 1
1.1 Motivation 1
1.2 Problem Statement 2
1.3 Motivating Scenario 3
1.4 Contribution 4
1.5 Structure of the Work 5

2 State of the Art 7
2.1 General Background 7
2.2 Designing Human Intensive Software Systems 9
2.3 Collaboration Tool Integration 12
2.4 Background 13

3 Methodology 21
3.1 Requirements 21
3.2 Design and Implementation 25
3.3 Evaluation 25

4 Design 27
4.1 Referencing collaboration patterns in process descriptions 30
4.2 Handle requests for collaboration patterns 31
4.3 Instantiate collaboration patterns 32
4.4 Framework Architecture 32
4.5 Runtime Architecture 34

5 Implementation 45
5.1 hADL Runtime 46
5.2 Marshaller 47

5.3 Surrogate Factory 49
5.4 Service Registry 49
5.5 Scope Manager 50
5.6 Operation Manager 53
5.7 Deployment and Execution 56

6 Evaluation **57**
6.1 **REQ-01**: Supporting Architecture Driven Development 57
6.2 **REQ-02**: Modelling the Architecture 58
6.3 **REQ-03**: Change in Collaboration Structure 65
6.4 **REQ-04**: Change in Collaboration Tool 67
6.5 **REQ-05**: Handling Human Faults 68
6.6 **REQ-06**: Process Engine Integration 68
6.7 Test Scenarios 68

7 Critical Reflection **69**

8 Conclusion **71**

Bibliography **73**

A Codelistings **81**
A.1 hADL 81
A.2 Surrogates 84

List of Figures

2.1 Relationship between communication, coordination and collaboration 8
2.2 LittleJIL icon to define a step obtained from [16] 16
2.3 LittleJIL notation to define substeps. The example has been obtained from [16] . . 16
2.4 LittleJIL states of steps based on [16] 18
2.5 Graphical representation of a hADL model of a chatroom 20

3.1 Interest-based bargaining process with its roles. Each step shows the involved actors and each party might consist of several individuals. 22
3.2 Overview of architecture driven development processes 24

4.1 Relationship between process description/engine and collaboration pattern/resource manager of the solution approach 29
4.2 Input/Output of process steps when referencing hADL collaboration patterns as resource. The tuple of hADL file and hADL elements identifies the pattern. A scope ID is assigned to the instantiated pattern. 30
4.3 The basic sequence of messages for using collaboration patterns as resources. The data of the messages are described in the arrow brackets. 31
4.4 Relationship and cardinality between hADL elements, their surrogates and the instances of each. 32
4.5 Overview of the human architecture implementation framework. 33
4.6 Overview of the components of the hADL runtime. The links between elements specify the cardinality between instances. 34
4.7 UML class diagram of the interfaces of surrogates. 36
4.8 The interface of operation managers. 37
4.9 The operation manager encapsulates incoming method calls into requests which are put into a queue, thus executing parallel calls consecutively. 38
4.10 Lifecycle states of surrogates which are facilitated by the operation manager. . . . 39
4.11 Interface of scope managers. 40
4.12 Interface of the hADL runtime. 41
4.13 Interface of the surrogate factory. 42
4.14 Interface of the service registry. 42
4.15 Interface of the marshaller. 43

5.1 Data model of resource descriptors. 46
5.2 UML sequence diagram for obtaining an scope instance from the hADL runtime. . 47
5.3 UML sequence diagram for obtaining an surrogate instance from the hADL runtime. 48
5.4 Process of converting hADL model instances into scope instances. 48
5.5 Step by step construction of a collaboration pattern instance. Coloured links indicate connections which have been added at the new step. 52
5.6 UML class diagram of the state pattern tracking surrogate states at the operation manager. 53
5.7 UML class diagram of the strategy and decorator pattern for executing surrogate requests at the operation manager. 54

6.1 Process model of the IBB process. 59
6.2 Publish/Subscribe collaboration pattern for IBB. 59
6.3 Shared artefact collaboration pattern for IBB. 61
6.4 Shared artefact collaboration pattern for IBB where all components may edit. . . . 63
6.5 Voting collaboration pattern for IBB. 65
6.6 IBB publish/subscribe pattern with an addtional party. 66
6.7 IBB publish/subscribe pattern where the moderator has been removed. 66

List of Tables

2.1 Time/Space taxonomy of groupware 9

4.1 Surrogate methods and the possible states after method execution. 37

5.1 Programming language, technologies and common libraries employed during implementation of the hADL runtime. 45

Listings

5.1 RxJava example for implementing ConnectRequest. 55
6.1 Input resources when starting the IBB process 60
6.2 Output resources of the IBB start step. 60
6.3 Input resources issue statements. 62
6.4 Output resources after consolidating issue statements. 62
6.5 Input resources when collecting interest items. 63
6.6 Output resources after collecting interest items. 64
6.7 Input resources when voting for options. 64
6.8 Output resources after voting for options. 65
6.9 Input resource of the IBB start step after adding an element to the collaboration pattern. 67
6.10 Output resource of the IBB start step after adding an element to the collaboration pattern. 67
A.1 hADL model instance of a shared artefact collaboration pattern. 81
A.2 Base class for asynchronous surrogates. 84

CHAPTER 1

Introduction

The advent of online collaboration has been observed in recent years. The web has enabled collaboration across geographical and to some extend temporal boundaries. Even though current collaborative activities focus on content generation and sharing, industries and scientific communities have already recognized the untapped potential of large scale collaborative undertakings. Nevertheless, software systems have not fully adopted yet to support human collaboration efficiently and effectively. This thesis tries to narrow the gap between the desire of individuals for collaboration and the software systems enabling it.

1.1 Motivation

The high level of interconnection between people through the web have shaped the way how they work and interact with each other. Emergent web based tools where many users can work together on tasks or products, even anonymously, indicate the advent of a shift from classical cooperation paradigms towards complex interaction patterns. Such changes can be observed in many problem domains, including the creation knowledge at Wikipedia[1], the distribution of information with Twitter[2] and social networking with LinkedIn[3]. Even writing articles can be done in real-time with multiple authors concurrently within a single document.

The relevance of modern collaboration and the *social technologies* enabling it, have also been emphasized in a survey published in 2013 by the McKinsey Global Institute. [18] They claim that social technologies have reached the main population and are being adopted by industries actively. Integrating collaboration tools into value chains can increase the performance of high-skill knowledge workers by up to 25 percent. A majority of surveyed companies in 2011 have already been using social technologies and nearly 90 percent have reported benefits. The institute postulated an untapped potential value of up to 1.3 trillion USD, which can be activated by fully implementing and integrating social technologies.

[1] wikipedia.org
[2] twitter.com
[3] linkedin.com

Positive effects of introducing collaboration tools have also been confirmed by the study of Evans et al. [31] in 2014 in the manufacturing domain. Team managers and members welcomed the new technologies and the authors concluded that modern groupwares offer the potential to deliver more effective collaborative environments and additional communication channels. Increasing impact of web-based collaboration tools also affects the domain of software engineering, which has already been conjectured by Whitehead in 2007. [76] The availability of novel communication channels and collaborative tools enables higher participation of stakeholders in the software development process and improves the documentation of rationale argumentations of design decisions.

A shift towards cross-institutional collaborations can also be observed in science. Jones et al. asserted the positive impact of collaboration on research quality. [41] Scientific research therefore profits from adequate tool support for collaboration. Research teams emerge in an ad-hoc fashion [37] and their average size is expected to increase monotonously. [9] Hence tools for coordination are gaining importance as well.

As human collaboration becomes increasingly normative and mandatory, software systems have to engage individuals, integrate means of collaboration and coordinate their interactions.

1.2 Problem Statement

Human intensive software systems (HISS) have gained momentum and their design and implementation is of major concern. [19] Nowadays collaboration of human participants is required in software systems and the integration of collaboration tools has become a necessity. Fully integrating collaboration tools also entails a change of view of traditional user roles. Regardless of the problem domain the collaboration tools are aiming at, they all have in common that humans are essential parts of the system. Traditional views of users only as consumers are no longer valid, instead humans have become both producer and consumer of services and content. They even gained importance as source of computational power. As Northrop et al. have predicted, the boundary between humans and computers are eroding. [60] Software designs have to incorporate the new role of users and developer teams have to model the interactions with and within the systems as well as communication between participants explicitly.

Humans as integral part of software systems have their own idiosyncrasies which has to be taken into account. Software system has no authority over their human participants, thus no behaviour can be enforced. The performance of tasks cannot be guaranteed and the networks responsible for delivering notifications of the systems which indicate state transitions are highly unreliable. Most software systems including human participation have to face similar challenges.

HISS which enable collaboration are confronted with a vast number of collaboration tools in heterogeneous problem domains. Tools differ in the number of participants and their interaction patterns as well as the tools' behaviour. The style of collaboration might emerge ad-hoc and does not always have a predefined structure. This stresses the flexibility of the means of system specification.

The number of collaboration tools might be considerably high, even in a single problem domain. Their underlying technologies often differ even though their behaviour or structure are similar. Actual programming interfaces and frameworks might vary substantially and could thus

be incompatible. Substituting collaboration tools is probably inevitable due to the volatile nature of web-based applications where the success of tools is determined by their popularity.

The aforementioned challenges have already been addressed in literature and industries, nevertheless no standardized approach exists to define and construct human intensive systems, in particular systems with innate collaboration of human participants. Current solutions are often incompatible or focus on particular problem domains, thus have limited general purpose. This thesis suggests a standardized framework for designing and developing collaboration-enabling software systems. In particular general solution approaches for reoccurring challenges are proposed.

A motivating scenario will be introduced in the next section to illustrate the challenging aspects mentioned above.

1.3 Motivating Scenario

In negotiation communities, interest-based approaches for resolving differences are highly acclaimed to be effective strategies. [72] In classical negotiations, two parties try to defend their position thus ultimately resulting in the loss of one party. By focusing on the underlying interests that are the cause of the position, solutions satisfying both parties might be found. [33] The process of interest-based bargaining (IBB) is an example of such approaches. IBB defines three actors: the *mediator*, *party A* and *party B* where the parties might consist of several individuals. The actors go through the following steps:

1. *Begin mediation session*: The mediator explains the process to both parties and gathers their contact information.
2. *Compose issue statements*: Both parties formulate a set of relevant issue statements independently usually in the form of open ended questions .
3. *Consolidate issue statements*: The mediator then tries to consolidate the issue statements from both parties. Results are new synthesized combined statements.
4. *Add interest items*: The parties have to identify interest items together, which have to be satisfied at the end of negotiation.
5. *Add options*: The process focuses on working towards a solution after the issue statements and interest items are identified. Both parties propose options which address the issues and satisfy some interests.
6. *Add options and questions*: Afterwards an immediate step to allow the parties to ask questions regarding the options and propose additional solutions follows.
7. *Identify acceptable options*: Then both parties determine acceptable options separately.
8. *Draft solution agreement*: Finally the mediator drafts a solution agreement based on the acceptable options of both parties.

Designing the System

A team of developers is assigned to implement a software system to support the IBB process. Process engines are already utilised by the company, thus the developers model the system with their process description language of choice. The sequence of IBB steps is straightforward but they struggle with the representation of the steps' internal architectures. Both negotiation parties have multiple members which have to collaborate internally in several steps.

Specifying the collaboration as sub process is tedious and results in a rigid design. Suppose synthesizing new issue statements in the IBB process involves the agreement of all parties. The combined statements can then be created iteratively with a shared document where issues are noted and a chatroom where issues are discussed. Describing the model with tasks and steps only requires considerable effort. Leaving it unspecified impedes the detection of architectural changes. If the participants, the shared document and the chatroom are passed as static resources to the step, their status cannot be monitored easily. Hence the need for adaptation is difficult to detect and events can only be communicated via exceptions.

1.4 Contribution

State of the art software systems nowadays have shift their focus towards user participation and collaboration. Process languages currently provide insufficient means to model collaboration efficiently and effectively. Inadequate models impede the development of adaptable systems. To this end, this thesis will make the following contributions to simplify the development of human intensive software systems:

1. This thesis introduces a framework for documenting, describing and prescribing the software architecture of human intensive software systems, which have innate collaboration of participants.

2. Standardized facilities to integrate and access collaboration tools.

3. Adaptability by abstracting concrete tools, thus enabling efficient exchange of implementations.

4. The framework offers a basic human fault handling model.

1.5 Structure of the Work

The remainder of the thesis is structured as follows:

Chapter 2 describes the state of the art of specifying human intensive software systems with respect to process modelling languages. Current literature and related work will be reviewed and discussed and background information will be provided.

Chapter 3 defines the method of evaluation for the resulting framework. Currently established methods for evaluating software architecture are introduced and utilized. In particular, a scenario based software architecture evaluation method is applied.

Chapter 4 details the architecture of the framework. Underlying design principles are elaborated and multiple views are employed to document the design decisions.

Chapter 5 summarizes used technologies, frameworks and tools while implementing the framework prototype.

Chapter 6 provides an qualitative evaluation of the framework prototype. The evaluation is based on the scenarios defined in Chapter 3.

Chapter 7 interprets the results of the evaluation and discusses the limitations and shortcomings of the prototype.

Chapter 8 summarizes the findings of the thesis and discusses opportunities for further research.

CHAPTER 2

State of the Art

This chapter provides theoretical background for collaboration in the context of computer systems. First, a brief introduction to collaboration in general will be given. Then tool support for collaboration will be discussed and current approaches for constructing collaboration-enabling software systems, rather human intensive software systems in general will be elaborated. Finally, concepts for documenting, prescribing and describing such systems will be reviewed.

2.1 General Background

Cooperation between humans is one major corner stone society is build upon. Fundamental achievements of and for humanity are hardly ever the result of actions and efforts of a single individual, but groups of people instead. Many projects are simply impossible to be managed by one person alone and the technological progress of society inevitably increases the number of such undertakings. The cooperation of many people is required to succeed in developing modern systems. Hence the role of efficient and effective cooperation is gaining momentum and importance. Cooperation can be seen as an agglomeration of three concepts which are *communication*, *coordination* and *collaboration*. Cooperative activities therefore can be divided into these categories. Individuals engaged in cooperative activities are *participants*. Communication represents the exchange of information between participants. Collaboration is the manipulation of information by the participants with focus on creating an outcome and coordination manages and relates communication and collaboration of participants. [46] These three concepts can also be viewed as subsets of each other (see Figure 2.1). Collaboration requires at least a minimum level of coordination between participants, thus collaborative activities are a subset of coordinated activities. Coordination on the other hand has (implicit) underlying communication, coordinated activities are therefore a subset of communicational activities.

The emergence of computers and networks influenced, and in some cases revolutionized, the way how work is done. Today it is almost a fact that modern technology supported by computers and communication networks has drastically improved efficiency and effectiveness

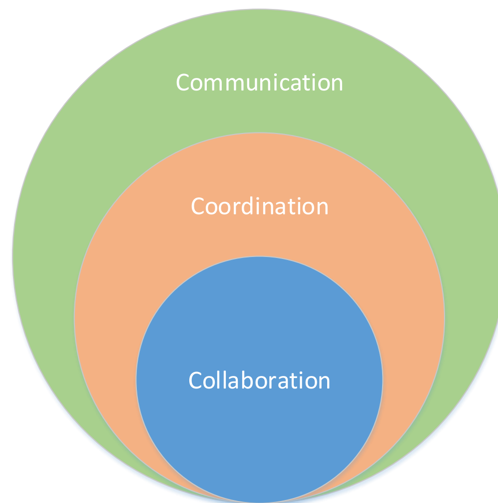

Figure 2.1: Relationship between communication, coordination and collaboration

of individuals working at a variety of tasks. [58] Numerous software intensive systems have been designed to support humans at their activities and to provide facilities to enable cooperation, but the resulting tools also had influence on the nature of cooperation itself.

The research domain of computer-supported cooperative work (CSCW) is trying to discover and utilize the interrelationships of users and their tools to further improve the quality of cooperation for humans with humans and also for humans with computers. CSCW applications are often referred to as *groupware*, although the exact boundaries of the definition is highly blurred in literature and depends on the level of abstraction. Researchers might categorize network file servers as groupware, while others might mention email as example. [36] Myriads of groupware applications exist nowadays and their classification imposes a challenge due to the variety of features and functionality. No single schema captures all aspects, although a few schemata are commonly used in literature. One of the earlier taxonomies to classify cooperation, and thus by extension also the supporting tools, defines a two by two matrix where *member proximity* and *group size* are used as dimensions. [21] This model has been refined later on and the dimensions were replaced by *space* and *time*. [36] Time determines the progression of interactions and space the location of the actors. The resulting taxonomy introduced a three by three model. Table 2.1 provides the schema and an example for each cell based on Borghoff et. al. [11] Activities can be carried out at the *same place*, at several *different but predictable places*, e.g. the locations are known to the participants, or *different and unpredictable places*, e.g. there might be unknown locations. Activities might also be performed at the *same time*, e.g. in real-time, at *different but predictable times*, e.g. with known deadlines or durations, or at *different and unpredictable times*, e.g. with unknown temporal boundaries. Nevertheless, the model is coarsely grained and some activities do not fit exactly into one category. Ellis et al. proposed a categorization based on application level functionality. [30] Their model classifies groupware as *message systems*, *multi-*

Space/Time	same time	different time	
		predictable	unpredictable
same place	face-to-face meeting	shift-work	blackboard
different place predictable	video conference	email	shared document
different place unpredictable	mobile phone conference	computer conference	workflow management

Table 2.1: Time/Space taxonomy of groupware

user editors, *group decision support systems*, *meeting rooms*, *computer conferencing*, *intelligent agents* and *coordination systems*. These classes however can also be interpreted as points on a triangular area defined through the vertices *communication*, *coordination* and *collaboration*.

Software systems have become ubiquitous in the past decades and have changed the way, how humans interact with each other. Communication was traditionally based on one-to-one channels and reaching larger numbers of people could only be achieved through broadcast, which exclusively provided unidirectional information flow. Bidirectional communication of groups of individuals demanded high effort if possible at all. Today vast peer groups can be contacted with relative ease through social software and social networks such as Twitter and Facebook. People can even reply to broadcast equivalent messages at any time and location, as long as internet connectivity is available. Coordination of cooperating humans traditionally could not surpass the boundaries of organizations and enterprises easily. Only humans within geographically limited areas could be coordinated effectively. These limitations also weakened with changing communication patterns and availability of supporting tools. *Crowdsourcing* for example, breaks down organizational boundaries and enables the coordination of large numbers of users to work at small tasks, which might be combined to an aggregated result. [42] It is also an example for harnessing collective intelligence. [61] The constraints of traditional coordination activities and the degradation of them also hold for collaboration due to the subset relationship. Modern collaboration tools allow geographically distributed participants to engage in joint activities. Even anonymous participation is supported. The advent of modern web applications transformed traditional groupware into current web-based collaboration tools. Their characteristics include the focus on user-generated content, sharing data and creating social networks. [50]

2.2 Designing Human Intensive Software Systems

Specifying and designing of human intensive software systems, in particular integrating human individuals and supporting collaboration are currently under active research. The following sections will elaborate common approaches to integrate humans in software systems and to facilitate collaboration. First, state of the art concepts will be discussed, then a brief overview of trends for collaboration tool integration will be covered.

Crowdsourcing

Crowdsourcing applications are currently prevalent software systems with large scale human participation. They aim at harnessing collective intelligence, which describes a group of individuals performing tasks collectively that seem intelligent in total. [55] Collective intelligence is commonly employed to solve predefined problems which are hard or currently impossible to automate, such as text translation and creative work. [22] Crowdsourcing is inherently task centric as the problems are split into smaller parts which are then solved by human workers. The coordination is commonly handled by workflows. [5, 43]

Bozzon et al. introduced *Reactive Control* as another model for coordination. [12] The authors describe crowdsourcing tasks as compositions of elementary task types such as labelling, liking or sorting, each with a predefined behavioural and data model. Descriptions are then transformed into features of a reactive execution environment. Their environment supports task planning, assignment and completion. Information about the runtime and the contexts are stored in a *Control Mart*. Instantiating workflows according to the data models yields the runtime model consisting of objects, performers and tasks, each with a corresponding control. The *Object Control* decides when and how responses are generated for each object. Determining the selection and rejection of performers is the responsibility of the *Performer Control* . Task execution, completion and replanning is overseen by the *Task Control*. Behaviour of controls are specified via rules expressed according to the event-condition-action paradigm. Most rules are automatically generated although custom rules can be developed and added on demand. Performers can be reached through multiple social platforms but direct interactions between performers are not foreseen.

Crowdsourcing systems can also be coordinated through models based on MapReduce. [44] Traditional crowdsourcing assumes solutions to problems to be simple aggregates of tasks. Nevertheless, real-life problems require more complex coordination. The authors claim that coordination needs can be classified as either shared resource, producer/consumer relationship and task/subtask relationship. All three dependency categories can be modelled with MapReduce. Ahmad et al. proposed a programming environment also based on MapReduce. [4] Their system provides a *Process Queue* where arbitrary processes based on MapReduce can be deployed. Each process consists of a given number of tasks which are submitted to a *Task Pool*. The submission information declares a specific *Service Adapter* to be used, which acts as a bridge to existing crowdsourcing platforms. Human participants can interact with the system via the crowdsourcing platform's user interface.

A variety of problems can be solved with crowdsourcing approaches, nevertheless the model is inherently limited as the problems are required to be decomposable into (independent) tasks. Problems which require intensive collaboration are thus hard to model.

Humans in Process Models

The most commonly employed software architectures in business environments are service oriented. In service oriented architectures (SOAs) process languages and their executing engines are often used to coordinate loosely coupled (web) services. SOAs include human individuals also as services, but treating them equally to software components could not satisfy industrial

requirements. A few solutions emerged to solve the issue and two approaches will be discussed in the following.

In industries the XML based Web Service Business Process Execution Language (WS-BPEL) is the most prominent process description language. WS-BPEL processes are essentially compositions of web-services with additional control structures. The main elements are *Activities* which can be *Basic Activities* and *Structured Activities*. Basic activities provide simple operations such as assigning values to variables or invoking a webservice asynchronously, whereas structured ones define flow control elements such as sequential, parallel and conditional execution. [39] The WS-BPEL's lack of support for human participation has been addressed in two complementary extensions, *WS-BPEL4People* and *WS-HumanTask*. [66] WS-BPEL4People adds a *HumanActivity* to the language which elevates human participants to first-class citizens. Human activities are implemented via *HumanTasks* introduced by the WS-HumanTask standard. The standard provides a state model for tasks, a coordination protocol for interactions and a role concept. Connecting process engines and humans requires the definition of two interfaces, one for exposing the services and another for managing and executing tasks by humans. Nevertheless the support for collaboration between humans is limited due to the low level of abstraction.

Shall et al. [67–70] proposed a novel approach for service oriented architectures. They introduced the concept of Human-Provided Services (*HPS*) to model human participation similar to software- or web-services. Their framework also includes an *HPS interaction model* consisting of an *activity model*, a *task model* and a *task execution model*. Collaboration can be defined through a series of hierarchical composed *activities* and their associated *tasks* which can be submitted to and executed by HPS. Although humans are integral parts of the system, arbitrary collaboration patterns are still tedious to specify. A Master/Worker pattern is incorporated in the frameworks design but intensive communication between workers is still unresolved. The level of details of the framework also does not support architecture driven development very well.

Lei et al. introduced a collaboration framework based on BPEL and also implemented a middleware prototype. [51] They extended BPEL activities with additional elements to allow the specification of human participants, in particular their role and admission policies. In addition, the interaction between participants can be described through a series of one-way and two-way communications. The extended elements were translated into conventional BPEL automatically. A middleware provides contextual sensitivity to determine the actual communication tools depending on the environment. This solution suffers from identical limitations as the aforementioned approaches, as BPEL's descriptiveness is limited in respect to integration of humans. All participants of activities must be known a priori and collaboration artefacts such as shared documents are omitted in the process definition.

Brambilla et al. proposed an approach to extend the business process model and notation (BPMN) to support social software. [13] Their extended notation allows the specification of pools of participants which can perform *Social Tasks* of *Social Activities*. Their solution offers a set of predefined task types, such as *Social broadcast* and *Voting*, and combines them to formulate common *social design patterns* for reuse purposes. This approach also supports code generation to accelerate application development. The solution does not support arbitrary collaboration types and is limited to social networks in particular.

2.3 Collaboration Tool Integration

The integration of tools into software systems is classically a subject of the domain of enterprise application integration (EAI). [54] Middlewares are traditionally used in order to integrate systems across organizational boundaries. Enterprise service buses (ESBs) in particular are employed in SOA environments. [38] ESBs provide an intermediate layer between proprietary technologies and systems requiring particular functionalities. They usually offer abstractions for messaging, protocol transformations and service containers. [17] Custom middleware implementations as integration solution for collaboration tools have also been evaluated in recent time. The next sections will discuss contemporary approaches.

Buford et al. introduced a middleware based on collaboration spaces. [15] A (collaboration) space defines a shared persistent container where users can conduct collaboration activities. It requires a set of resources which can be provided through third party collaboration platforms. Three layers are defined by the system. In the bottom layer collaboration platforms are integrated and consolidated models of the collaboration tools are provided as shareable views. It further provides resource management components for devices and a semantic storage for data mining. The mid-layer manages an unified representation of collaboration artefacts and offers user and session handling as well as eventing mechanisms and data stores. Collaborative applications which can coordinate and manipulate the objects in the mid-layer are contained in the top layer.

Buford et al. also designed an integration solution for connecting cloud-based and intra-organizational collaboration tools. [14] They proposed a three-tiered architecture where the first tier consists of the clients in the organisation's internal network. External tools and cloud-servers are in the third tier. In-between lies the second tier where mediation servers are located . Tools in the first and third tier are developed to match the interfaces of the mediators, thus no direct communication between internal and external collaboration tools exists. The authors implemented extensions for internal tools to realize the link to the mediator. Extensions were directly integrated into the graphical user interface, as most of tools were desktop applications. Thus users did not have to change their preferred tools. The cloud-based tools offered web-service interfaces which simplified their integration.

Similarly, the efforts of Wu et al. [81] and Mohit et al. [59] to integrate communication tools also offer concrete technical guidance. Both works introduce an abstraction layer to integrate communication tools. Another layer on top then coordinates and composes their abstract representation into larger systems.

Mashups

Providers of collaboration tools and platforms often offer publicly accessible application programming interfaces (APIs) to allow developers to integrate their services. Mashups are a recent trend to compose public APIs into individual workflows. [80,83] Due to the availability of graphical development environments even end-users are potentially capable of creating them. [35, 82] Integrating data is especially challenging for mashup developers. Their data-sources are often incomplete, inconsistent and heterogeneous and the developers are often no experts in the particular domain. [57] This challenge is usually addressed in five steps [74]:

1. Data Retrieval: Data are extracted from sources which might be unstructured, especially when they stem from HTML pages.

2. Source Modelling: The data are then structured to be distinguishable from other sources.

3. Data Cleaning: The extracted data have to be normalized and corrected if possible.

4. Data Integration: Penultimately the data sets from different sources have to be combined.

5. Data Visualization: Finally the preprocessed data is visualized in a graphical user interface.

The concept of mashups can also be applied to the coordination and integration of web-based collaboration tools, although the approaches are still unstructured and lack of appropriate standards. Nevertheless mashups can offer ad-hoc web-based collaboration.

2.4 Background

The remainder of the thesis requires in depth understanding of two architecture description languages, LittleJIL and hADL respectively. LitteJIL provides an integration context for evaluating the solution and is a graphical programming language which also follows the trend of development environments for mashups. In contrast, hADL is an integral part of the solution. Both languages will be exemplified in the following sections after a brief introduction to software architecture and their documentation in general.

Software Architecture

Software design can be documented in software architectures which are a set of principal design decisions about the system [73]. They describe software elements and relations among them. [8,10,34] Documentations of the software architecture can be descriptive, i.e. it reflects the architectural choices been made, or prescriptive, i.e. it limits the design choices. [62] The design decisions are made to meet external requirements, which can be functional or non-functional. Conformance to requirements is often seen as proxy for software quality. Software architectures can be classified into architectural styles based on their pattern of structural organisation. A pattern consists of components, which are loci of computation and state, and connectors, which are loci of communication. [2, 71]

Earlier attempts to document software architectures within a single model failed due to the innate complexity. The concept of views and viewpoints emerged instead, which focus on particular aspects of the system. A view represents the system under discourse from the perspective of a set of related concerns whereas a viewpoint establishes the purpose and audience of views and governs their construction and analysis. [1]

Kruchten introduced a framework based on multiple concurrent views on the systems design to achieve a more holistic description. [48] He defined 4+1 views, each targeting different concerns of the overall system.

As software systems are becoming more complex, the focus of software engineering shifted from lines of code to more coarse grained structures such as components and connectors. The change leads to new requirements on (formal) models and tools to encompass architecture-based development approaches. Hence many architecture description languages (ADLs), domain specific as well as general purpose architecture modelling languages, emerged. [56] ADLs can focus on any arbitrary aspect with any suitable notation, thus a single accurate definition is hard to formulate. An early study conducted by Vestal tried to identify common elements or features of ADLs. [75] The results of his analysis of the samples can be partially summarised in the following reoccurring concepts:

1. Components, rather their type are defined by interfaces which have an implementation aspect. Multiple instances of a component might occur. Component interfaces consist of entities where connections can be drawn from or to. The implementation of components can be defined though hierarchical composition. Leaf-level elements have to be developed in traditional programming languages. Features might exist to model the behaviour of components.

2. Connections between elements have no homogeneous semantic. Their meaning can be specified in the language to some degree.

3. Tool support is preferred to compose traditionally implemented components into larger executable systems.

4. Means for automated analysis, evaluation or verification of particular aspects might be offered.

A more detailed survey based on feature analysis by Clement classified reoccurring elements of ADLs into three categories: system-oriented, process-oriented and language-oriented features. [20] In his opinion, an ADL must support the creation, refinement and validation of architectures. They should be fit to represent common architectural styles and provide consistent views that express architectural information. In a recent study by Lago et al., the authors formulated concrete requirements for next generation ADLs. [49] They introduced a framework which clusters the requirements into the groups *language definition*, *language features* and *tool support*. Their demands on future ADLs shall help in establishing them in industrial environments.

LittleJIL

LittleJIL is a high level process language for representing and implementing processes in software systems. [79] The notion of process is not clearly defined, it is dependent on viewpoints of the system under discourse. Rolland utilized the four worlds framework, originating from the systems engineering discipline, to clarify the perception of processes. [65]. Therein the *subjects world* is the world of processes, *usage world* investigates rationales, *systems world* concerns their representation as process models and the *development world* focuses on the construction of process models.

The subjects world defines the system under discourse and its elements, which are processes in this context. A process is performed to produce a product and is seen as a route to be followed to reach a product in information systems contexts. The start and end position of the route can be described as points in a space with the three dimensions: *specification*, *representation* and *agreement*. [63] From an activity-oriented perspective, a process is a partially ordered set of steps intended to reach a goal. [32] Product oriented viewpoints define processes as a series of transformations to reach the desired product and decision oriented viewpoints define them as a set of related decisions conducted for product definition. [65]

The usage world determines the goals of processes. It defines the environment, where the processes are performed and the actors performing them. Further more the nature of evolution and change is also determined. Strong requirements on processes are imposed by the usage world.

The systems world defines the representation of the process, the representation's level of abstraction, its means and properties. Processes are classified as *process models*, which are type level abstractions. Thus a processes are instantiations of the corresponding process models, which on the other hand are instances of *process meta models*. [47] The systems world also determine the notation and organisation of process models.

The development world governs the construction of process models, i.e. the process of creating process models and the enactment of processes, i.e. the tool support needed for the execution of processes. It defines construction approaches and techniques as well as tool support for enactment and change. [65]

LittleJIL provides a process meta model and support for process enactment, thus it relates to the systems and development world. The following sections describe the characteristics of LittleJIL in respect to process meta models and their semantics.

LittleJIL Meta Model

Process models in LittleJIL have the goal to be understandable for non-programmers, thus they are specified graphically. The language's primary focus lies in the coordination of tasks, which are executed by agents. [16] A LittleJIL process model represents a process as a hierarchical composition of step types, i.e. a tree of step types, where each step type can be instantiated multiple times. The leaves of the tree are the smallest units of work and intermediate nodes are coordination structures. Thus the shape of the tree specifies the coordination and the leaves define the work to be done. [79]

Due to the graphical notation, a step is defined through a step icon (see Figure 2.2). A step can be annotated with badges which might have several variations with different icons and semantics [77–79]:

Prerequisite Badge: Prerequisites are preconditions of the step which are checked before execution. They can be specified through *predicates*.

Postrequisite Badge: Postrequisites are a set predicates which are checked after execution.

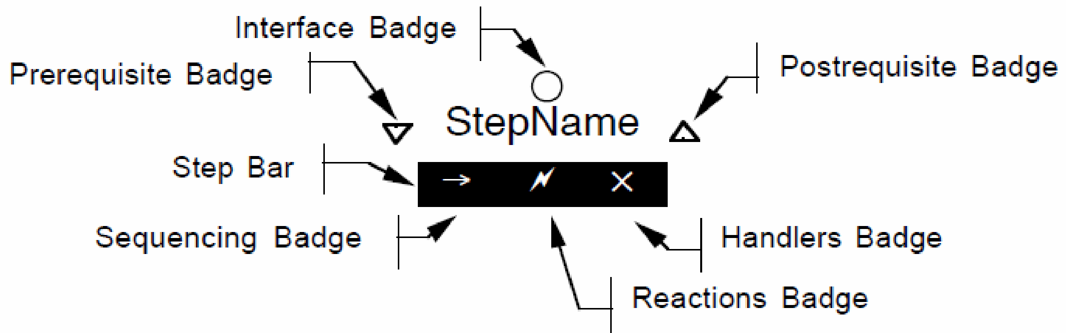

Figure 2.2: LittleJIL icon to define a step obtained from [16]

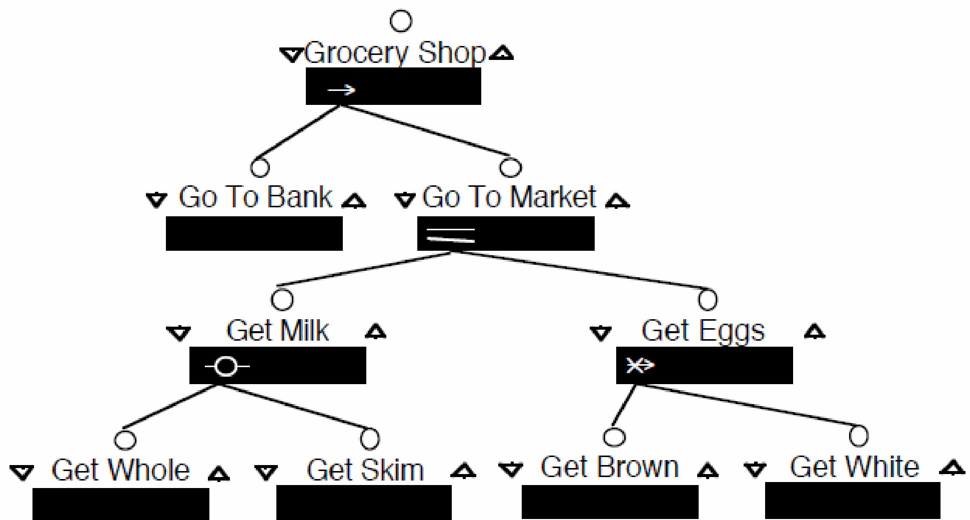

Figure 2.3: LittleJIL notation to define substeps. The example has been obtained from [16]

Interface Badge: The interface defines *Resources* used by the step, the *Parameters*, any declared *Channels* for communication, *Exceptions* that can be thrown and *Messages* that can be emitted. Agents can be passed to a step as a resource.

Sequencing Badge: The actual sequencing badge defines the executional semantics of the step. It can be of the kinds *None*: the step is a leave step and is executed by an agent, *Sequential*: substeps are executed sequentially from left to right, *Parallel*: substeps might be executed concurrently, *Choice*: one substep is selected by an agent and executed and *Try*: substeps are executed from left to right until one succeeds.

Handlers Badge: Handlers are callbacks which are executed when a specified exception is thrown.

Reactions Badge: Reactions define callbacks for emitted messages. Messages can signal the state of the step.

In LittleJIL, hierarchical composition can be achieved through definition of substeps (see Figure 2.3). The import and export capabilities of steps ease the management and reuse of process models.

Any runtime requires a set of additional components to be present to be able to execute processes as LittelJIL focuses only on the coordination of steps. [53] The problem domain specific behaviour have to be implemented separately. The following components are necessary:

Execution Agents: Agents perform the actual work and can be software or humans.

LittleJIL Interpreter: This component implements the runtime semantics of the process model.

Agenda Manager: The agenda manager governs the communication between interpreter and agents, such as notifying agents, when new steps are assigned.

Resource Manager: The resource manager is primarily responsible for resource acquisition and release. It handles resource requests of the process components.

Artefact Manager: Artefacts, such as type models for type verification, are managed by this component.

A step goes through several states of execution during runtime (see Figure 2.4). It is *Posted* when the unit of work is in the agenda (work queue) of the agent and required resources are available. The state transits to *Started* when the agent begins the task and to *Completed* when the task is finished successfully. Any unrecoverable faults lead to *Terminated*. The consequence of the runtime removing tasks before agents start them, is the state *Retracted*. If the agent denies the task before starting it, *Opted Out* is reached. Further language details of LittleJIL can be obtained from. [77]

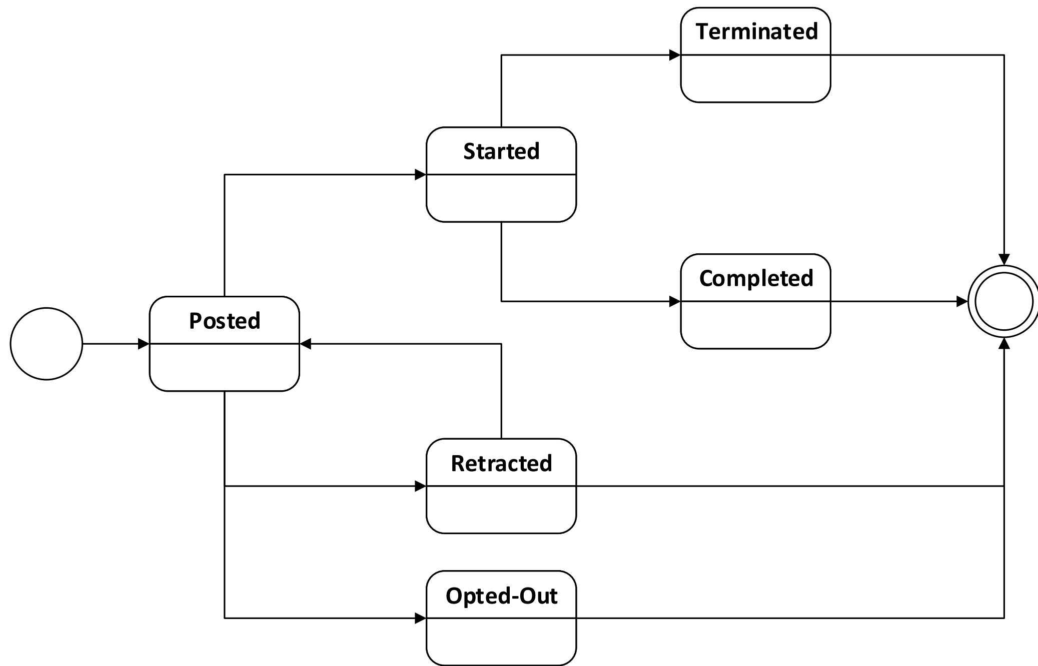

Figure 2.4: LittleJIL states of steps based on [16]

hADL

Current ADLs, process languages in particular, minimize human involvement in architectural descriptions to achieve reproducibility and reliable process outcomes. In contrast, the advent of large scale human intensive software systems demands adequate means of representation to promote flexibility and adaptability. Dorn et al. introduced the human architecture description language (hADL) to close this gap. [28] They extended the notion of architectural components into *human components* and connectors into *collaboration connectors* in order to emphasize the essential role of humans in such systems. By unifying communication and interaction of software with software, humans with software and humans with humans into *collaboration patterns*, higher flexibility and runtime adaptability can be achieved. [24, 29] Well known architectural styles can also be translated into collaboration patterns as done by Dorn and Taylor. [26] Each pattern has its scope of applicability, benefits and limitations dependent on the problem instance. Seven collaboration patterns have been exemplified:

Shared Artefact: Shared artefacts represent generic objects through which participants indirectly communicate, such as shared documents. Common operations are *Create*, *Read*, *Update* and *Delete*.

Publish/Subscribe: Participants communicate through events where *publishers* emit and *subscribers* receive them, such as mailing lists. Participants can have both roles and might

stay anonymous.

Master/Worker: A *client* provides and submits units of work called *tasks* and *masters* distribute them to a pool of *workers* which perform them. Crowdsourcing is considered an incarnation of this pattern.

Social Network: Large number of participants can form sparse networks, where multiple communication paths between nodes exist. Individuals can collaborate ad-hoc and in self-organized groups.

Workflow: A workflow consists of a *queue* where *workers* pick items of work and forward them to the next step after execution. Communication between participants might be reduced to the items of work only.

Secretary/Principal: This pattern introduces layering to collaboration. Direct communication with principals might not be possible. Instead a number of secretaries act as *proxies*.

Organizational Control: Organizational control describes strict hierarchical systems where *supervisors* apply proactive control to *subordinates*, which report reactively.

Models of hADL can also be used to generate configuration through model-to-model transformations. [27] The details of the hADL meta model will be described in the section to follow.

hADL Meta Model

The XML based specification of hADL is split into three modules: *hADLcore*, *hADLexecutable* and *hADLruntime*. The simplified corner stones of hADL are:

HumanComponent: The HumanComponent is the locus of computation and data. It can be a fully autonomous software component, an individual human or anything in between. A HumanComponent defines a set of *actions*.

CollaborationConnector: A CollaborationConnector manages interaction between HumanComponents and is responsible for efficiency and effectiveness. The possible levels of automation is equal to HumanComponents. It also defines a set of *actions*.

CollaborationObject: The CollaborationObject abstracts the means of interaction, such as messages or shared artifacts. Concrete semantics is captured through subtyping. They offer a set of *actions*.

Actions: Actions define capabilities a HumanComponent or CollaborationConnector requires to fulfil their role. In the case of CollaborationObject, actions determine the provided capabilities.

Meta model elements also include additional attributes, which are omitted for clarity and brevity reasons. The hADLexecutable extension defines supplementary properties to specify

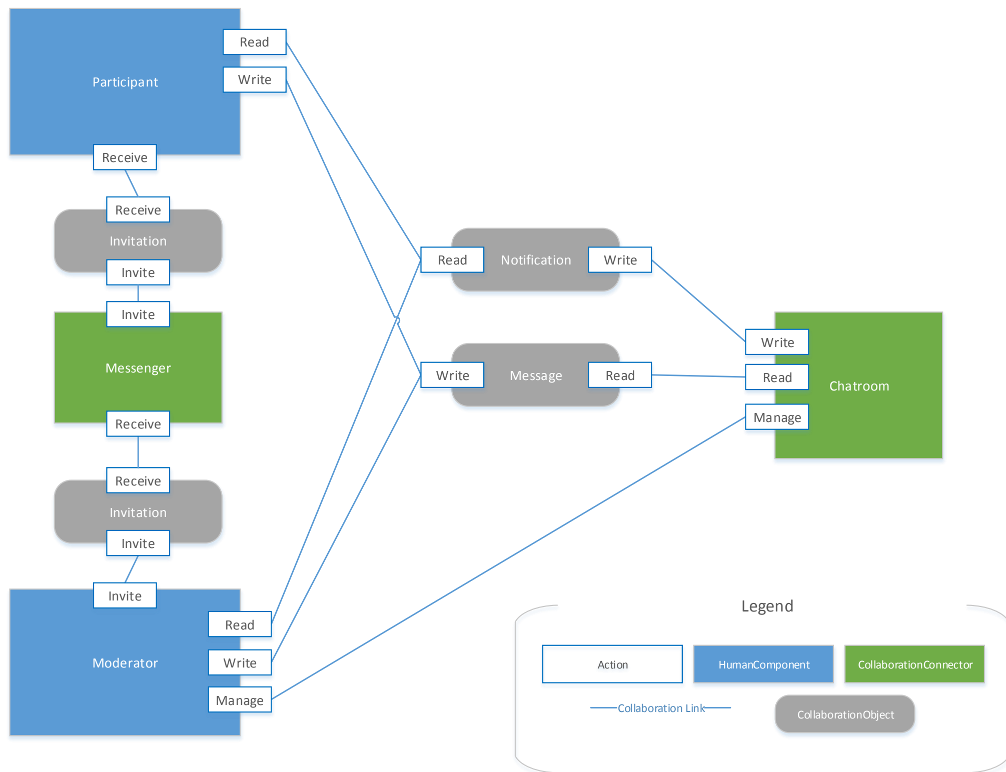

Figure 2.5: Graphical representation of a hADL model of a chatroom

mappings to implementations, whereas hADLruntime defines elements to capture the runtime environments and configurations.

A simple collaboration pattern might be a chatroom (see Figure 2.5). Both *participants* and *moderators* can connect to the *chatroom* and write to and read messages from it. The collaboration composition fits into the description of the Publish/Subscribe pattern where both components have two roles. Moderators can invite participants to a chat through the *messenger* connector, hence participants can receive invitations. Additionally moderators can manage the entire chatroom. In a runtime incarnation of the pattern, participants could be humans and moderators software components. The messenger could be build on e-mails and the chatroom could be a conference call over the web.

CHAPTER 3

Methodology

This chapter describes the process of designing a solution to the problem statements mentioned in the introductory section. The first step of the process is to identify the overall requirements for the framework which are presented as fictional scenarios in this thesis. Then an architectural design is developed and implemented later on, hence yielding the framework prototype. The prototype is evaluated in the last step based on the requirements.

3.1 Requirements

The requirements for business software systems are derived from the goals of the collaborative undertaking. Use cases are a popular modelling technique to capture requirements in the context of software engineering. [3, 40] They are formally defined and focus on actions and actors. Although this thesis employs this method, no formal definition will be provided. Instead the requirements will be encoded in an informal fashion as prosaic text which will be referred to as *scenarios*.

Scenarios

The scenarios are based on a fictional company, which uses the interest based-bargaining approach (see Figure 3.1) introduced in Chapter 1.3 in their business processes. Their research and development department is assigned to implement a workflow to support IBB. Developers have interviewed some stakeholders and identified the following scenarios.

REQ-01 *Supporting Architecture Driven Development*

The company has adopted an architecture driven development approach (ADD), which reflects current development trends, to implement their systems. Their development process is depicted in Figure 3.2 at an abstract level. First, architectural requirements have to be collected to define the overall goal of the system. Then a preliminary design of the system is drafted. The design has to be documented which yields a set of artefacts describing

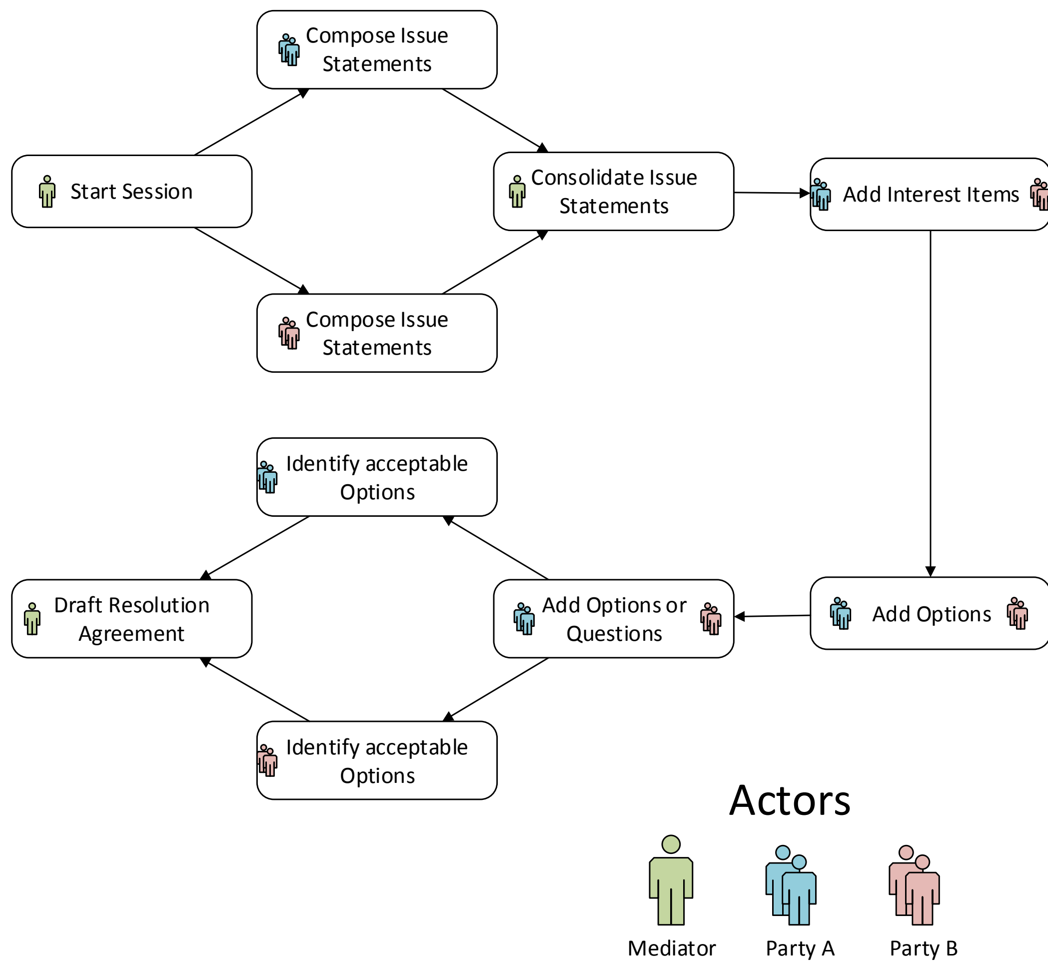

Figure 3.1: Interest-based bargaining process with its roles. Each step shows the involved actors and each party might consist of several individuals.

the system's behaviour, internal structures and interfaces. The design is evaluated against the requirements and refined according to the analysis' outcome. An adequate solution is thereby approached iteratively. The realization of the actual system starts when the architectural design has stabilized and the requirements have been met at an acceptable level. Finally the architecture of the implemented system has to be matched against the prescribed design to maintain the consistency and validity of the architectural design artefacts. Any deviation from the original design leads to a change of either the implemented system or the prescribed architecture. Whenever a new requirement emerges, the entire process is executed again. This cycle reflects the system's evolution over time. The framework shall support the intermediate steps of this development approach and the transitions between them.

REQ-02 *Modelling the Architecture*

The ADD approach requires adequate means to prescribe and describe the system. This includes in particular the artefacts created while designing the system. The means of documentation of the system under development (SUD) has to be suitable to represent the system's behaviour and structure. It has to reflect the system's agents, which can be individuals as well as software components, and their interactions.

REQ-03 *Change in Collaboration Structure*

The characteristics of human collaboration is innately spontaneous and therefore inherently difficult to predict. A prescribed collaboration structure might not fit the real interaction patterns of participants and might be rejected by them. Hence participants might resort to undocumented and unofficial communication channels thus leaving the sphere of influence of the SUD. The structure of interactions might also be not known a priori, emerge in an ad-hoc fashion or change drastically over time. In the context of IBB for example, adding interest items might be realized through a shared document which all participants of both parties might edit. At any point, the organisation might switch to a master/worker pattern where the master collects items from the workers and submits a complete list at once to be able to filter spam messages. The framework has to support architectural changes and facilitate simple adaptation mechanisms.

REQ-04 *Change in Collaboration Tool*

Software tools, especially web-based ones, are subjected to regular changes and updates. The preference of collaboration tools might also change rapidly depending on popularity of the tools. Effective collaboration requires user acceptance and adaptability, therefore simple exchange mechanisms for underlying tools are required. In the IBB process for example, the provider for shared documents which does not support real-time editing of multiple users, might be replaced by a modern one when introduced. A change to another tool might also be triggered by a shift of preference for specific user interfaces or by usability issues. The framework must provide adequate abstraction mechanisms to hide implementation details and to enable flexibility regarding the choice of collaboration tool.

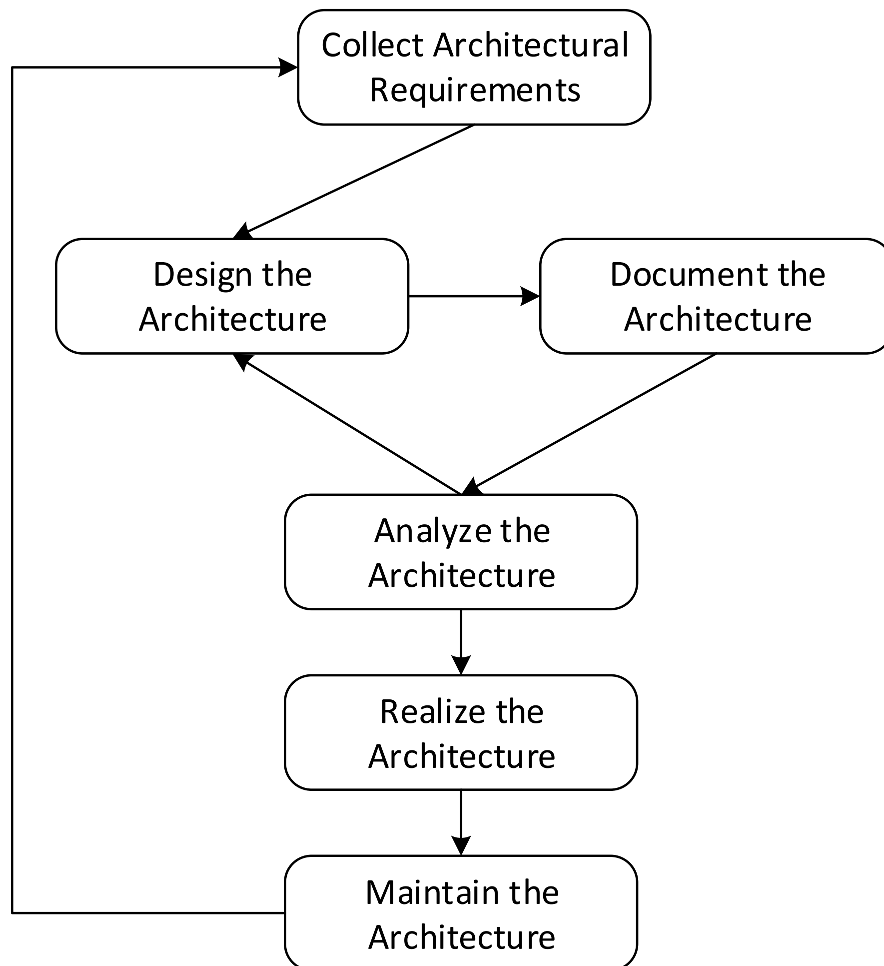

Figure 3.2: Overview of architecture driven development processes

REQ-05 *Handling Human Fault*

Humans are integral elements of the SUD but the system has no authority over them. Participants cannot be forced to execute tasks, especially not in a timely fashion. Their behaviour cannot be predicted and policies cannot be enforced. Definitions of acceptable behaviour depends on the sociological and corporal culture. The framework has to provide at least basic means to handle human faults and mechanisms to implement customized policies.

REQ-06 *Process Engine Integration*

Many companies heavily rely on process engines to execute their business workflows, so does the fictional company of this scenario. Using process engines implicates the presence of process models which already are essential parts of the architectural documentation. The acceptance or success of the framework is tied to it's ability to reuse existing architectural artefacts and to incorporate efforts already invested.

3.2 Design and Implementation

An architectural design addressing the aforementioned requirements is proposed. The software architecture is based on common architectural patterns at macroscopic and software design patterns at mesoscopic level. Details are described in Chapter 4.

Details of the implementation are located in Chapter 5. This chapter lists concrete technologies, tools and frameworks employed while implementing the framework.

3.3 Evaluation

Software architectures have high impact on the system's quality and therefore have to be evaluated to verify the capability of the system to fulfil the requirements and to detect design errors. [7] There is a plethora of evaluation methods, nevertheless no single method can assess the quality of architectures as a whole. They focus on specific aspects or quality attributes instead. Most approaches are scenario-based architecture evaluation methods (SAAMs), which encode requirements or quality attributes into textual scenarios. [23] Scenarios are then discussed with software architects to identify risks, tradeoffs and sensitivity points. [45] SAAMs are considered to have a high level of maturity which fosters the acceptance and confidence for these methods. [6] Nevertheless a fully fledged SAAM is out of the scope of this thesis. Instead the human architecture implementation framework is evaluated for compliance to the scenarios, or rather the requirements enlisted in Section 3.1. First, the IBB process is designed on the basis of the prototype. Afterwards the level of conformance to the requirements are argued in prosaic form, in particular in respect to **REQ-01**, **REQ-02**, **REQ-03** and **REQ-04**. Further more experiences gained from the application of the framework are also shared in Chapter 7.

CHAPTER 4

Design

Current human intensive software systems are often described through process languages which are then executed by process engines. Process languages such as LittleJIL model behavioural views of software systems with focus on coordination, underlying structural composition of the system are underspecified. The integration of collaboration tools on the other hand requires those to be explicitly modelled, which is hard to achieve with process languages only. This is mostly due to their innate structural and behavioural complexity. Structure-centric ADLs such as hADL on the other hand, are fit to create structural views but lack the ability to specify global or internal behaviour. A combination of structural and behavioural ADLs would create a more complete documentation of software architectures without tedious effort. [25] This thesis uses LittleJIL as process language as it focuses on coordination, is freely available and has a clear documentation of the language and its runtime. Nevertheless LittleJIL can be replaced with any process language as it serves only as conceptual point of reference. The structure-centric language of choice is hADL as it is a perfect fit for describing collaboration structures.

At a high level of abstraction, two solution approaches have been identified. The languages can either be integrated at *language level* or *runtime level*.

The integration of ADLs at language level had already been attempted in the past. An early work of Robbins et al. tried to integrate UML and various ADLs, although their intention was very different from this thesis'. [64] The authors promoted the use of standardized languages in order to benefit from their widespread application, which would also improve the overall quality of tools and thus the architectural views created. They examined the expressive equivalence of the ADLs by modelling specific aspects of the given languages in UML. Their results suggested that considerable effort was required for the integration, in particular modifications and extensions of UML were necessary. Thus a direct integration requires an adaptation of both LittleJIL's and hADL's meta-model. This approach also implies additional changes in language interpreters which may cause compatibility issues with existing applications.

The runtime integration approach does not require any changes in the languages' meta-model. Instead points for integration in their respective runtime environments have to be identified and exploited. This results in a looser coupling of both languages. Only existing interpreters

have to be modified or extended. No changes at language level also entails reusability of existing model instances and backward compatibility. Both approaches require interpreters for both languages which hADL does not provide yet. Thus a hADL runtime has to be developed in both cases. This thesis evaluates the runtime approach due to the simplicity and novelty as language level integration has already been attempted.

The human architecture implementation framework's means of architecture prescription and description requires two ADLs to work together and their respective roles have to be clarified and defined first. Dorn et al. have already identified three strategies to exploit the complementary nature of LittleJIL and hADL. A combined solution approach can either be based on a (1) task-driven, (2) interaction-driven or (3) artefact-driven strategy respectively [25]:

Strategy 1: The task-driven strategy uses LittleJIL to specify basic tasks and their dependencies as well as process artefacts and their data flows. Thus the macroscopic behaviour of the system is defined through LittleJIL. The primary purpose of hADL is to define the logical structure of leaf steps.

Strategy 2: The interaction-driven approach puts human participants in the foreground, thus hADL defines the overall system's architecture. LittleJIL specifies the internal logic of hADL elements.

Strategy 3: The artefact-driven strategy models the system as set of major artefacts with artefact manipulation capabilities. Coordination mechanisms are defined through hADL and LittleJIL specifies the internal logic of connectors or active collaboration objects.

Dorn et al. concluded that no single strategy dominates, thus all three strategies are viable. With *Strategy 1* a primary (composite) process model is developed and any collaboration structures required in leaf steps are defined through hADL. This strategy eases the migration from existing process models as the models can be simplified step by step to incorporate hADL but can generally be reused. The *Strategy 2* requires existing process models to be split into multiples which specify hADL elements' internal behaviour respectively. *Strategy 3* requires the split analogously to the second. All strategies require the creation of new hADL model instances. Based on the widespread application of process engines, **Strategy 1** seems to be the most pragmatic approach. The strategy requires only minimal changes of process engines and does not disrupt current approaches of process designers.

Employing *Strategy 1* limits the options of integration points. Process engines are focussing on assigning activities to agents and the **dataflow** required to carry them out. [52] Most process meta-models include mechanisms for resource handling thus integrating collaboration structures via a **resource model** becomes evident. The solution approach transforms collaboration patterns defined through hADL into resources usable by LittleJIL leaf steps. Figure 4.1 depicts the essential elements of the solution and the relationship between them. The system under development is described through a process description and a collaboration pattern description, which are created by the software developers. Then the process description can be instantiated and executed by a process engine. It requests instances of collaboration patterns from the framework during runtime which in turn has to load the pattern description to identify required elements. Links

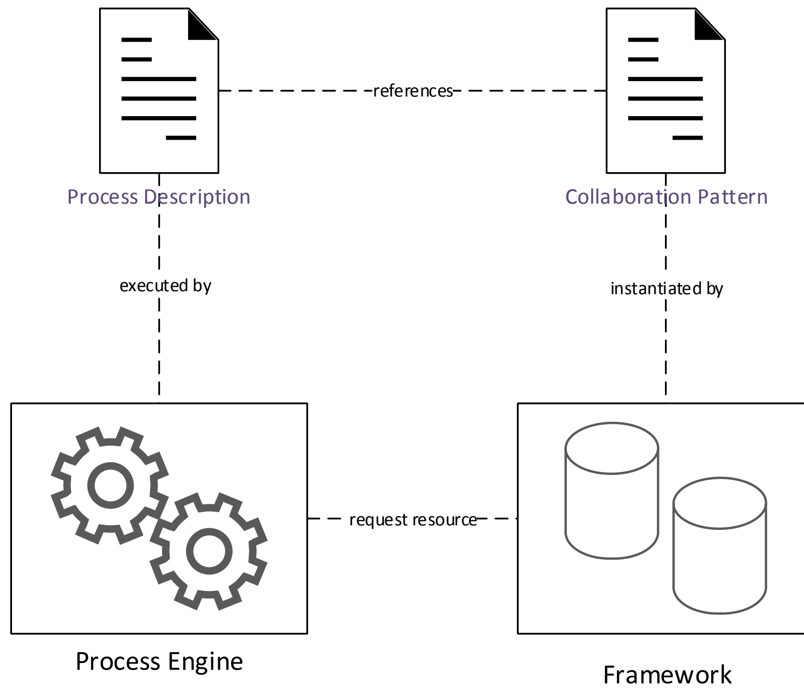

Figure 4.1: Relationship between process description/engine and collaboration pattern/resource manager of the solution approach

between elements in Figure 4.1 represent four sub-problems which have to be tackled in the solution approach:

1. *Referencing collaboration patterns in process descriptions*

   The framework must provide means to reference collaboration patterns and its elements from the process description.

2. *Executing process descriptions*

   The process engine is responsible for the execution of process descriptions. This sub-problem is implicitly solved by LittleJIL and thus does not have to be described explicitly by the framework.

3. *Handle requests for collaboration patterns*

   Any process engine, especially LittleJIL, has a defined resource model which determines the lifecycle of resources. The framework therefore must define a compatible model itself which can simulate the states and transitions of the process engine's resource model. It

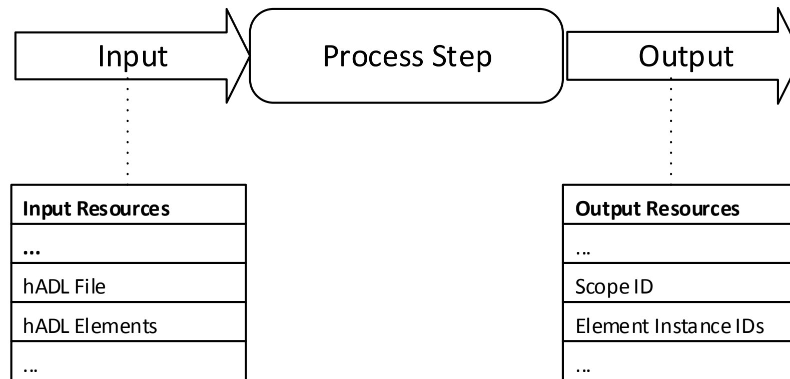

Figure 4.2: Input/Output of process steps when referencing hADL collaboration patterns as resource. The tuple of hADL file and hADL elements identifies the pattern. A scope ID is assigned to the instantiated pattern.

also has to provide resources such that they can be shared between steps. Thus instances of collaboration patterns and elements must be identifiable.

4. *Instantiate collaboration patterns*

   Instantiation of collaboration patterns is strongly tied to the resource model or rather the lifecycle. The actual logic required to realize patterns on specific platforms are very distinctive, thus capabilities for using individual implementations have to provided.

The following sections detail the solutions to the sub-problems previously mentioned. Only the execution of process descriptions is omitted, since it is the core functionality of process engines and as such it is already given.

4.1 Referencing collaboration patterns in process descriptions

Collaboration patterns can be seen as a named collection of (human) components, (collaboration) connectors and (collaboration) artefacts, thus each element of the collection can be referenced via an identifier unique only within each pattern. Collaboration pattern references in process descriptions have to at least specify an identifier for the pattern and a set of identifiers representing the elements of the pattern. Figure 4.2 depicts the input and output of process steps when a collaboration pattern is referenced. The step's input and output are sets of key/-value entries where the input has to specify the file which contains the hADL collaboration pattern. Additionally a list of hADL elements which should be initialized, can be provided. The outcome of the step then contains identifiers which are unique strings associated with the instantiated collaboration pattern and its elements. Hence the tuple of file path and element represents a reference from the process description to the hADL collaboration pattern.

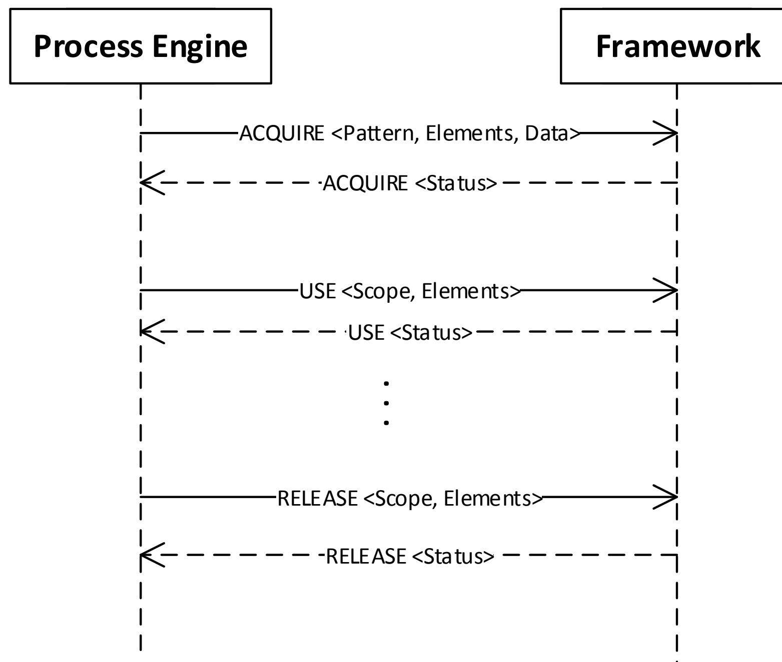

Figure 4.3: The basic sequence of messages for using collaboration patterns as resources. The data of the messages are described in the arrow brackets.

4.2 Handle requests for collaboration patterns

LittleJIL defines a simple lifecycle for resources where each resource can be *acquired*, *used* and *released*. The order of resource management operations is not enforced by LittleJIL and has to be modelled explicitly using individual steps. For that purpose additional symbols for defining resource acquisition and resource usage are provided. Releasing resource is requested implicitly when all steps using the resource are terminated. [77] Thus the framework has to support at least these three operations. Figure 4.3 illustrates the sequence of messages passed, when using a collaboration pattern in a process. The entire communication occurs in an asynchronous fashion to provide flexibility.

First an `ACQUIRE` message is sent, which contains the pattern description, the elements to be initialized and the data of the elements. The response contains the status, which indicates if the operation succeeded or failed. Statuses also contain IDs referencing the acquired pattern and element instances. References are usually memorized by the step's executing agent.

Then a number of `USE` messages are submitted which link element instances to scopes. The response contains a status indicating success or failure.

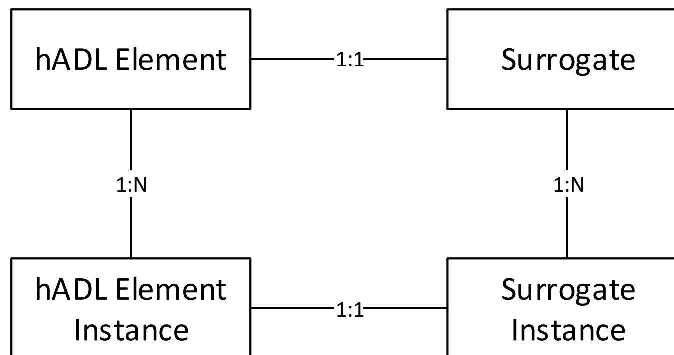

Figure 4.4: Relationship and cardinality between hADL elements, their surrogates and the instances of each.

Finally the resources are freed by issuing a `RELEASE` request containing the target scope and its elements. The response informs if the resources could be released.

4.3 Instantiate collaboration patterns

The actual logic to instantiate hADL elements depends on the problem domain and no general implementation can be provided by the framework. For example acquiring an individual human is fundamentally different to establishing a video conference for collaboration. Hence the logic has to be encapsulated in an extra element. The hADL executable extension augments hADL elements with an additional field to specify such an element called *surrogate*. Surrogates act as proxies between the runtime and the actual collaboration element. The behaviour of surrogates has to be provided by the developers which are using the framework to implement their target system. Figure 4.4 depicts the relationship between elements, surrogates and instances. Each hADL element has exactly one surrogate and each element instance has exactly one surrogate instance. Elements and surrogates can of course have arbitrary many instances.

4.4 Framework Architecture

In the previous sections, three major issues have been identified which have to be addressed by the framework. Figure 4.5 illustrates its basic components. The process engine executes the process description and interfaces with the *hADL runtime* to allocate, use or release resources. A runtime enforces asynchrony and responds with status messages which encapsulate the outcome. While handling a resource request, the runtime has to either obtain the resource via an existing surrogate or create a new one and query the instance according to the request. Creating a new surrogate requires the processing of the hADL specification which contains the collaboration pattern with its elements. Based on the specification, surrogate instances are created and

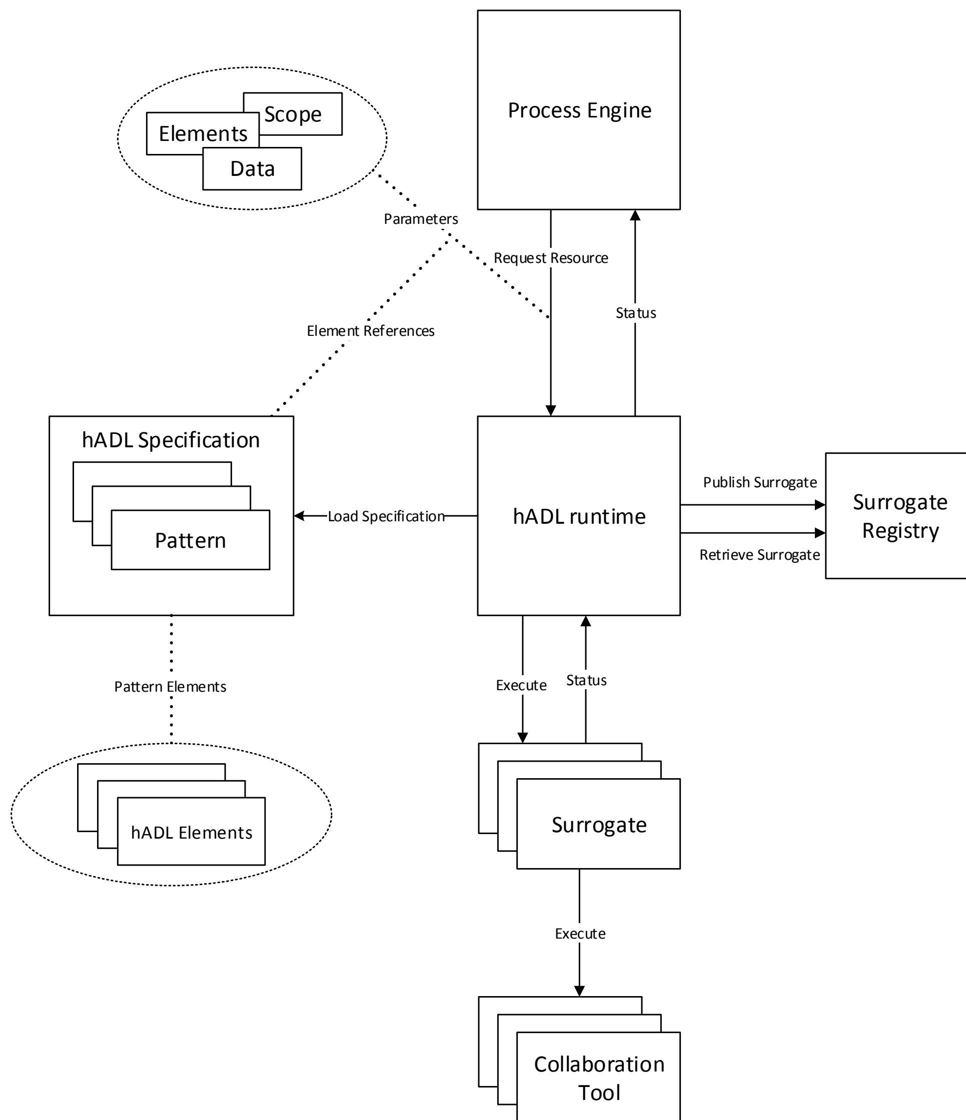

Figure 4.5: Overview of the human architecture implementation framework.

published to the *surrogate registry*. The registry enables the runtime to share existing instances between process steps and process instances. Then surrogates for managing operations, such as acquiring resources or releasing them, are requested by the runtime. Requests are handled asynchronously, thus status messages inform the runtime about the outcome. Finally surrogates are tied to the actual collaboration tools and use their specific interface to conduct the managing operations. The design of the runtime is introduced in the next section.

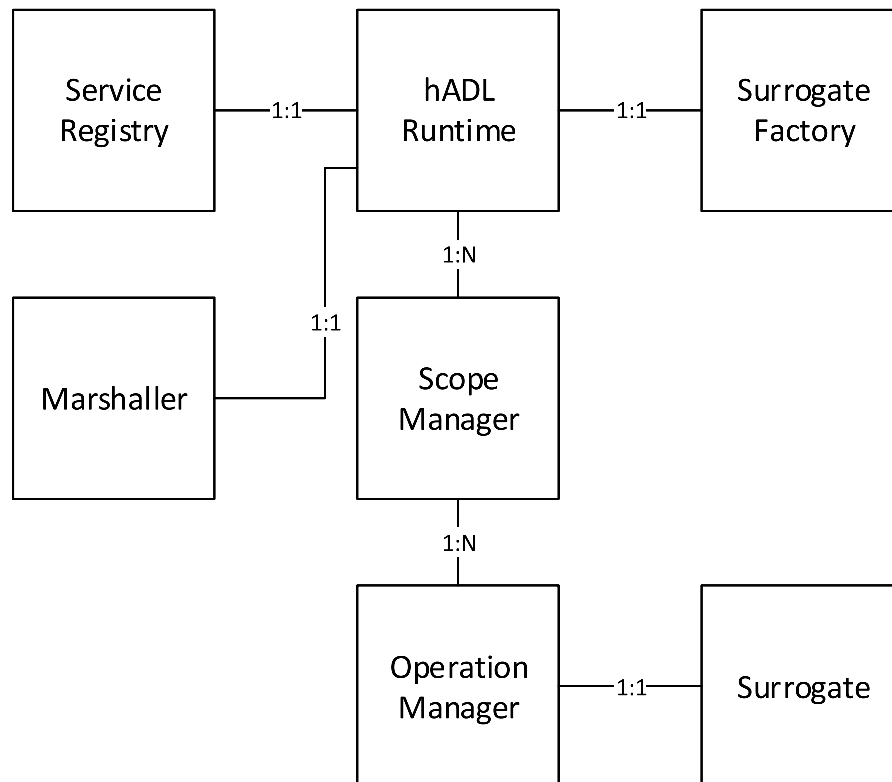

Figure 4.6: Overview of the components of the hADL runtime. The links between elements specify the cardinality between instances.

4.5 Runtime Architecture

The runtime described in Figure 4.5 has to provide access to the following functionalities:

1. Manage the lifecycle of resources of collaboration elements according to hADL model instances.

2. Manage the membership of collaboration element instances to process instances and their steps.

An architectural view of the runtime's components is illustrated in Figure 4.6. The `hADL Runtime` component acts as the single point of access to functionalities required by process engines. Collaboration pattern element instances which have been acquired are held in a `Service Registry`. A `Surrogate Factory` creates runtime instances of surrogates according to the specification of third parties. `Marshallers` transform hADL descriptions into runtime

objects. The `Scope Manager` determines the membership of allocated resources to process instances and their steps. `Operation Managers` wrap any requests to surrogates and provide basic error handling mechanisms. Finally the `Surrogates` facilitate the actual communication with collaboration tools. The next sections detail the interfaces, structure and behaviour of the runtime architecture components.

Surrogates

According to the resource model of LittleJIL, surrogates have to at least support three basic operations which are acquiring the resources, using them in process steps and releasing the resource allocations. Acquiring and releasing resources can be directly included into the surrogate's interface. Modelling resource usage on the other hand requires the operation to be split into two separate operations. First acquired resources can be used in different pattern instances (scopes) thus the framework must support linking element instances to any given scope. Second it also has to support the disconnection from arbitrary scopes and their elements. All operations employ the *observer pattern* as all communication have to be asynchronous. Figure 4.7 depicts the classes and interfaces defining surrogates. Any call to a `Surrogate` method results in an `Observable` which represents a stream of `SurrogateEvents`. The caller can register an `Observer` to the `Observable` which is notified when a new event has been raised. A `SurrogateEvent` contains the current `SurrogateStatus` indicating the state of the `Surrogate` and any `SurrogateExceptions` that have occurred. Table 4.1 contains the surrogate's methods and its corresponding statuses. Each method results in a success- or failed-message. The surrogate's behaviour has to be implemented by third party application developers and its methods require predefined and stable contracts which are given below:

`acquire`: This method allocates and initializes resources required by the collaboration component for a given scope. The resource descriptor contains the resources specified in the process step. This operation is called for each scope separately. Surrogate implementations have to relate resources to scopes.

`connectTo`: Connection of collaboration components in possibly different scopes is implemented in this method. The passed surrogate's resources have to be already acquired. Surrogate implementations have to track established connections.

`disconnectFrom`: The complementary method to `connectTo` disconnects two collaboration components. A corresponding connection has to be already established.

`release`: The counterpart to `acquire` releases all allocated resources for the given scope.

Operation Manager

Operation managers introduce an additional abstraction layer which provides basic error handling, rudimentary transactions and tracking of the surrogate's states. The surrogate's connect and disconnect methods only allow the specification of a single endpoint whereas the operation

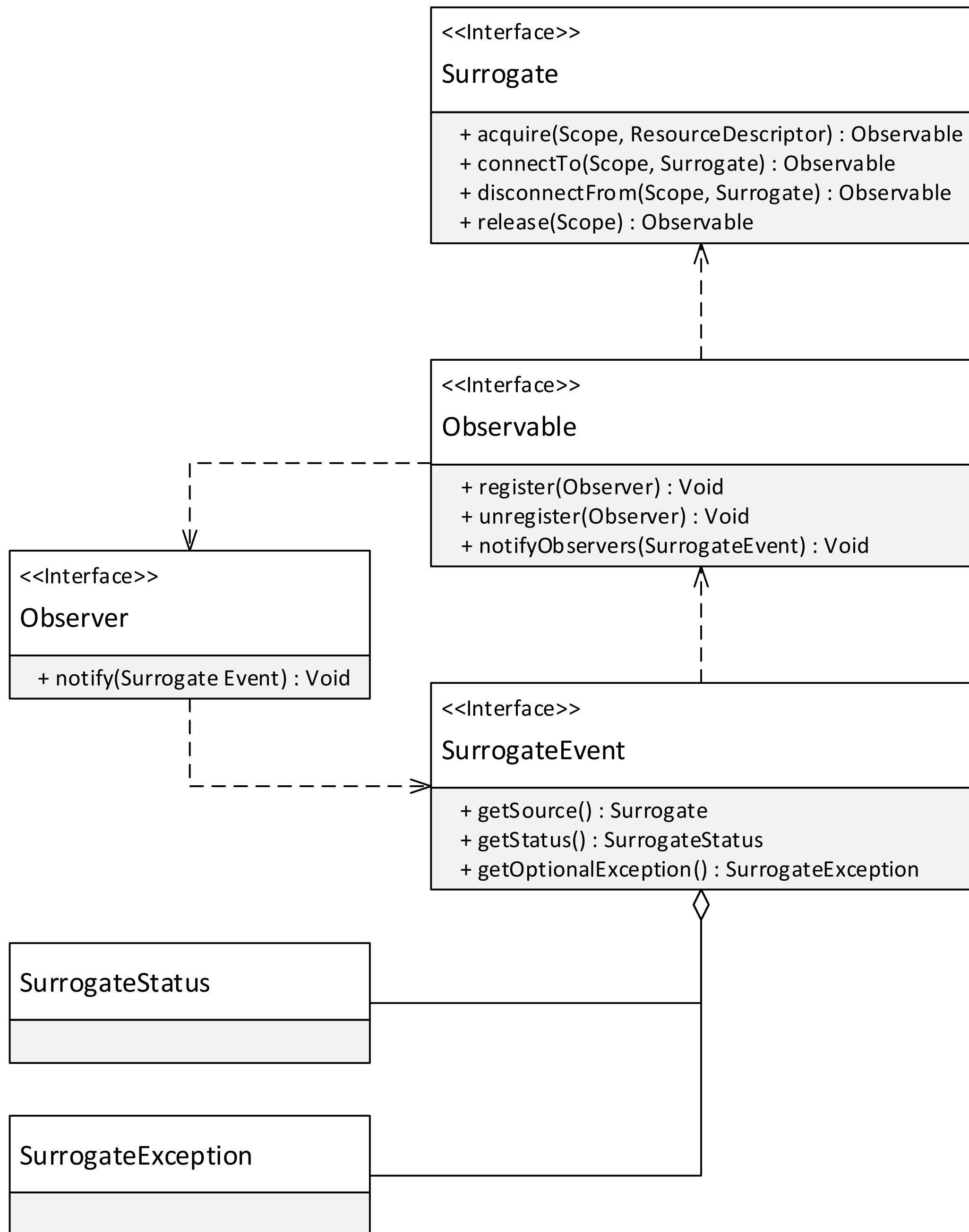

Figure 4.7: UML class diagram of the interfaces of surrogates.

Method	**Message**
acquire	ACQUIRING_SUCCESS
	ACQUIRING_FAILED
connectTo	WIRING_SUCCESS
	WIRING_FAILED
disconnectFrom	UNWIRING_SUCCESS
	UNWIRING_FAILED
release	RELEASE_SUCCESS
	RELEASE_FAILED

Table 4.1: Surrogate methods and the possible states after method execution.

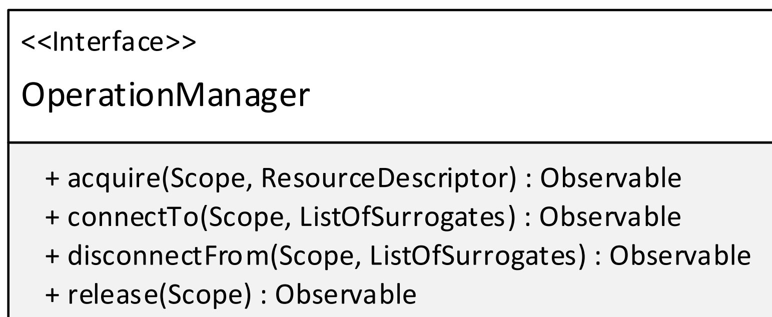

Figure 4.8: The interface of operation managers.

manager supports an arbitrary number (see Figure 4.8). Results of these methods are equal to the messages of surrogates (see Table 4.1), although the semantics differ slightly. The status messages indicate the success or failure of connecting and disconnecting from all endpoints at once. Thus the entire operation fails whenever a single endpoint fails.

Due to the fact that surrogates or rather the operation managers are shared between multiple processes or process steps, concurrency issues must be handled explicitly. Surrogate methods require a specific order of call to fulfil their duty and each call has to be atomic and thread-safe. Thus the operation manager wraps each call to its methods into a message and organizes them into a sequence of consecutive requests.

Figure 4.9 illustrates the mechanism. Any operation manager has two threads running concurrently which share a common queue. *Thread A* (which is the executing thread of the caller) handles incoming calls by creating a runtime object which encapsulates the called method and all parameters. The object is then put into the shared queue. *Thread B* takes objects from the queue one after another and calls the surrogate method according to the data of the object. After receiving a success or failure message from the surrogate, the next request is processed.

A operation manager also has to track the state of the surrogate to determine valid operations

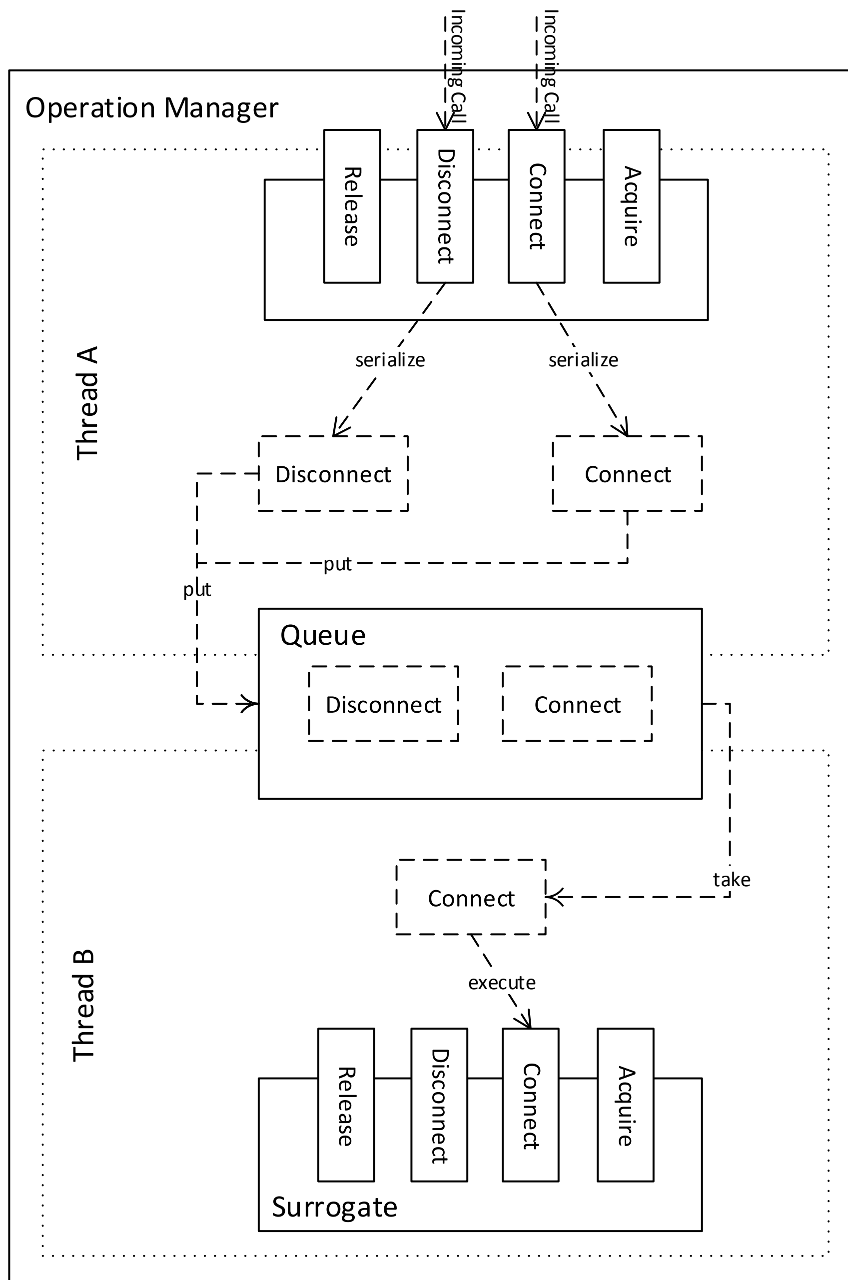

Figure 4.9: The operation manager encapsulates incoming method calls into requests which are put into a queue, thus executing parallel calls consecutively.

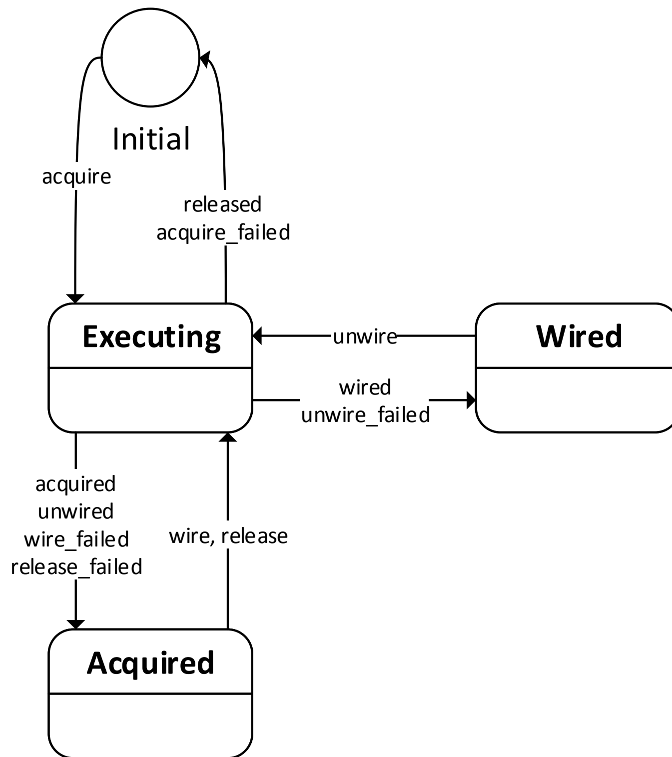

Figure 4.10: Lifecycle states of surrogates which are facilitated by the operation manager.

and to enable status queries by the runtime. States are modelled as a state machine where the transitions are based on the interface's operations and their results (see Table 4.1). Figure 4.10 depicts the state machine representing the lifecycle of surrogates. The following states are defined:

`Initial`-State: The initial state is reached whenever a surrogate is created or all acquired resources are released. This state is also the result of failing resource acquisitions due to any errors.

`Executing`-State: The execution of any requests leads to this state. It indicates an active operation and is left when a result is available regardless of success or failure.

`Acquired`-State: This state represents a successful acquisition of resources by the surrogate. The surrogate is not yet used in any scopes and thus is not connected to any other resources.

`Wired`-State: The surrogate is connected to at least one element in at least one scope when this state is reached. This state can only be left when all connections are terminated.

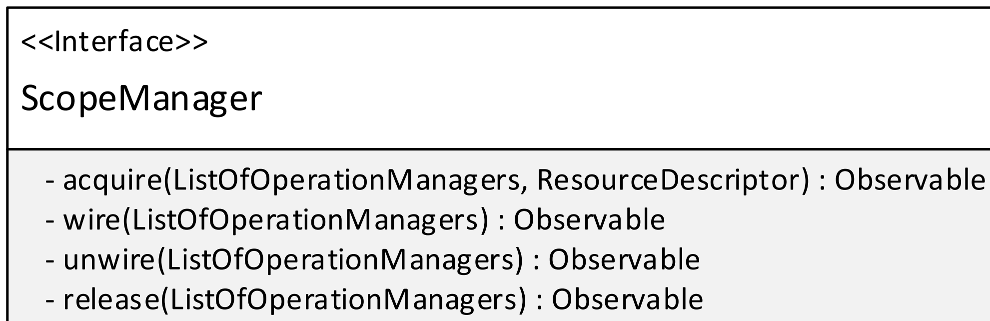

Figure 4.11: Interface of scope managers.

Scope Manager

At the next higher level of granularity in contrast to operation managers, the scope manager encapsulates an instance of a collaboration pattern, whereas an operation manager wraps a single element of the pattern. The scope manager has to keep track of element instances currently connected to it. Figure 4.11 illustrates the available operations on scope managers. All methods also return observables to support asynchronism. The following operations are defined:

`acquire`: Scope managers forward calls to the `acquire` method of the surrogates which are invoked via an operation manager. The results are returned through the observable along with any potential exceptions.

`wire`: This method connects the given surrogates or rather their operation managers to components of this scope. First all incoming and outgoing links of the element in the underlying hADL model instance have to be determined. Then elements at the opposite side of the links have to be collected resulting in a set of target elements. The operation manager has to connect to all instances of any of the target elements in the scope.

`unwire`: This operation is the counterpart to `wire`. Analogously all links of the corresponding hADL elements in the hADL model instance are collected first. The passed operation managers then have to disconnect from every available instances of those elements.

`release`: The scope manager releases the resources acquired by the surrogates for this scope by calling their corresponding method.

hADL Runtime

At the highest level of abstraction, the `hADLRuntime` interfaces with the process engine. The runtime has to parse hADL descriptions via the marshaller, create surrogate factory instances

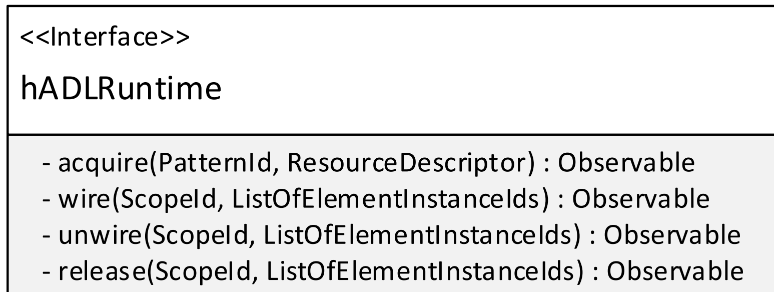

Figure 4.12: Interface of the hADL runtime.

and retrieve element instances based on identifiers. It also has to assign unique identifiers to new scopes and surrogate instances. The interface depicted in Figure 4.12 has been derived from the sequence of operation illustrated in Figure 4.3. Releasing resources has been split into `unwire` and `release` to provide finer granularity, although LittleJIL's resource model does not require it. The runtime defines the following operations:

`acquire`: This method has to either create a scope and a scope manager based on the `PatternId` or retrieve an existing scope from the service repository when a scope identifier is provided in the `ResourceDescriptor`. Analogously element instances are created or retrieved in the same fashion. The surrogate factory is employed whenever new elements need to be created. Each element instance is then wrapped into an operation manager and finally the acquire method of the scope manager is called.

`wire`: Handling the wiring request first requires the retrieval of instances from the service registry according to the identifiers passed. Then the wiring method of the scope manager is called and its result are returned.

`unwire`: Complementary to `wire`, the element instances and the scope manager are retrieved first. Then all operation managers are unwired from the scope.

`release`: After obtaining the object instances, the release method of the scope manager is called with the operation managers as parameter.

Surrogate Factory

Surrogate factories create surrogate instances based on the hADL specification. They basically map hADL elements to runtime surrogate objects. The surrogate implementations are provided by application developers, thus a mechanism to add new implementations during runtime has to be provided. In order to do that, developers have to supply an isolated binary library file

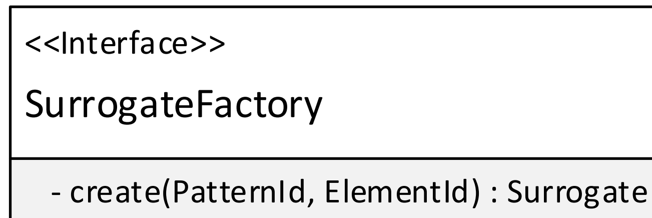

Figure 4.13: Interface of the surrogate factory.

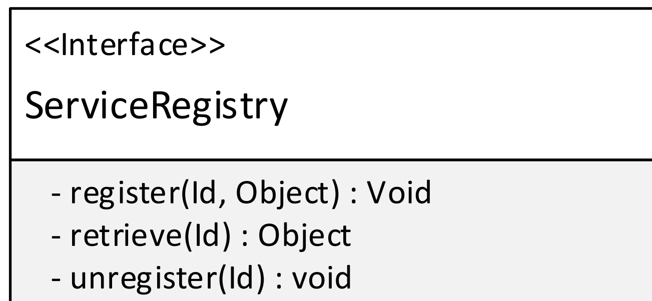

Figure 4.14: Interface of the service registry.

containing the surrogate implementations. The hADL model instances then have to reference the binary objects in the library. Figure 4.13 contains the simple surrogate factory interface definition which has only one method:

`create`: This method creates a surrogate instance based on the `PatternId`, which identifies the hADL description and `ElementId`, which determines the hADL element whose surrogate has to be instantiated. The factory has to keep track of available patterns autonomously.

Service Registry

The service registry is a facility to store and retrieve object instances. Multiple process instances might access the registry which requires it to be run in a separate process context. The registry enables sharing of scopes and surrogates between process instances and steps. Figure 4.14 defines the interface which has the following methods:

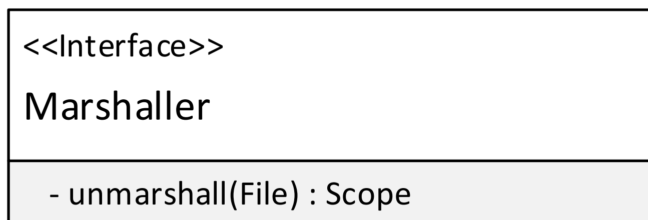

Figure 4.15: Interface of the marshaller.

`register`: The given object is published in the registry under the given `Id`. This method fails when the identifier has already been registered. The object on the other hand is allowed to be registered under arbitrary many identifiers.

`retrieve`: A published object is retrieved based on the passed `Id`. An exception is raised when no object is registered under the given ID.

`unregister`: A published object is removed from the registry based on the `Id` provided. If the ID has not been registered, an exception is raised instead of failing silently.

Marshaller

The last component of the framework is the marshaller which simply parses hADL descriptions and turns them into their runtime representation. Figure 4.15 describes its interface which has only one method:

`unmarshall`: This method parses the provided `File` and returns a scope object representing the hADL description if the file contains a valid hADL model instance.

CHAPTER 5

Implementation

This chapter elaborates and enlists the technologies employed to implement the components according to the design described in the last chapter. Similar to the structure of Section 4.5, each component defined in Figure 4.6, except of surrogates which are implemented by application developers, will be described in a separate section. Each section contains the structural and behavioural description of the implementation as well as the concrete technologies in use. The general programming environment and libraries along with their version can be obtained from Table 5.1. *Microsoft Windows* as operating system has been run on the development computer and the programming language of choice was *Java* along with *Eclipse* as IDE. *Apache Maven* has been used to manage dependencies and the build process. The *RxJava* library has been used to implement asynchrony of operations and the observer design pattern. *Google Guice* was employed as dependency injection framework and *Apache Camel* as service registry provider.

Category	**Technology**	**Version**
Platform	Microsoft Windows	7
Development Environment	Eclipse	4.4.1
Build Management	Apache Maven	3.2.3
Programming Language	Java	1.8
Libraries	Apache Camel	2.14.0
	Google Guice	4.0
	RxJava	1.0.2
hADL	hADL core	1.1.1
	hADL executable	1.2.0
	hADL runtime	1.0.2

Table 5.1: Programming language, technologies and common libraries employed during implementation of the hADL runtime.

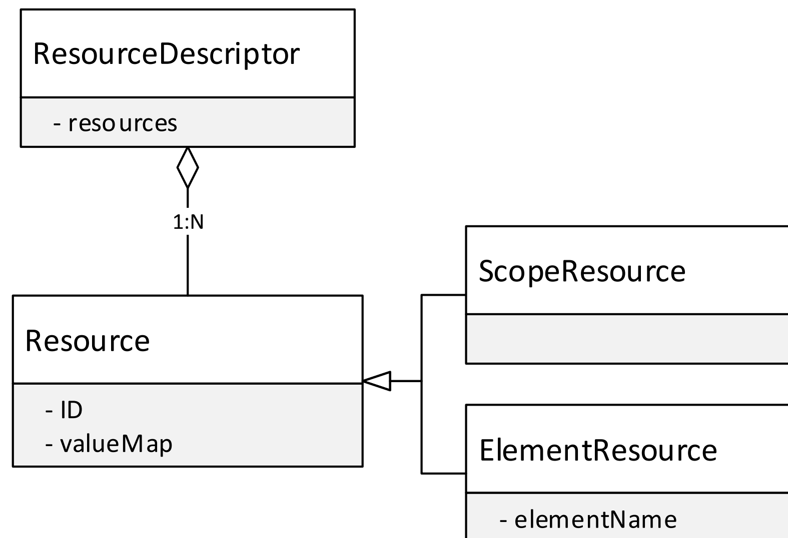

Figure 5.1: Data model of resource descriptors.

5.1 hADL Runtime

As described in Section 4.5, the functionality of the runtime can be reduced to two basic tasks. First, the runtime has to map instance identifiers to instances. Second, the requests from the process engine has to be forwarded to the scope managers. Retrieving or creating instances of scopes or surrogates requires the resource descriptors to contain identifiers for both. Multiple instances of each element might be used in a single scope and each instance of the elements has to be retrieved or created. The distinction between a new or an existing instance is determined by the absence or presence of an identifier for the element or scope under discussion.

Figure 5.1 depicts the data model of resource descriptors. A `ResourceDescriptor` consists of a set of `Resources` which have an `ID` attribute and a key/value data structure named `valueMap` for arbitrary string values. The resources are either `ScopeResource` holding data for a scope (hADL collaboration pattern) or an `ElementResource` containing data for a hADL element.

Figure 5.2 and Figure 5.3 illustrate the sequence of actions executed and the participating components when mapping identifiers to runtime objects for scopes and surrogates respectively. When a scope instance is required, the resource descriptor has to either contain a `ScopeResource` with a non-empty `ID` which directly references an existing instance, or a `ScopeId` referring to the hADL description of the collaboration pattern. If both identifiers are supplied, the scope ID takes precedence. Passing multiple `ScopeResources` leads to an exception. Each non-empty identifier triggers a query to the service registry to obtain the correspondent object instance. The marshaller is used to create new scope instances when an empty

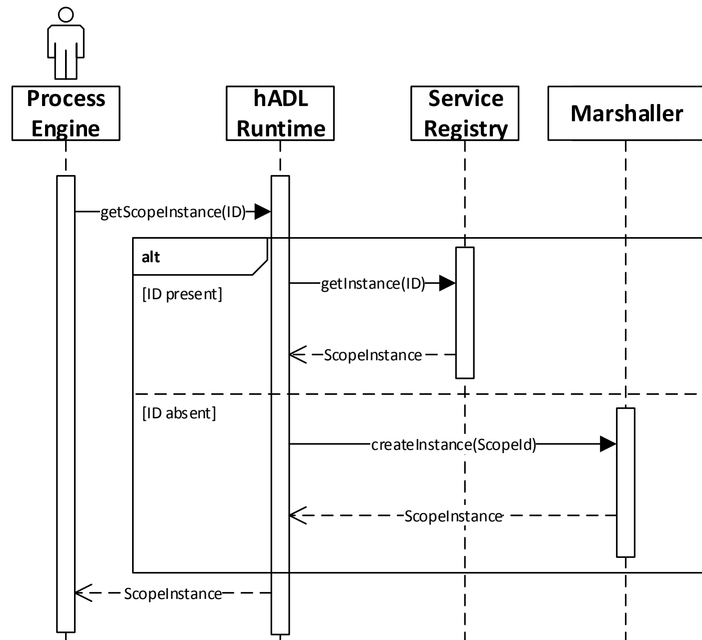

Figure 5.2: UML sequence diagram for obtaining an scope instance from the hADL runtime.

ID or no `ScopeResource` is supplied. Surrogate instances are obtained analogously, although the surrogate factory is used instead of the marshaller. New scope and surrogate instances are wrapped into scope manager and operation manager instances respectively, which in turn are registered in the service registry.

5.2 Marshaller

The marshaller is the component responsible for converting hADL descriptions which are provided as XML files into their runtime representation. Transforming runtime objects into a storeable format and vice versa is often called *marshalling* and *unmarshalling* respectively. Only the latter feature is required by the hADL runtime prototype.

The implementation of the marshaller employs the *Java Architecture for XML Binding (JAXB)* to handle XML processing. JAXB requires multiple steps to enable parsing hADL descriptions at runtime. Figure 5.4 describes the overall process which yields a scope instance. First the XML schemata of hADL has to be processed by JAXB which generates Java classes suitable to represent the XML equivalent. The classes are packaged along with the runtime which can then be started. During execution of the runtime, hADL model instances are submitted by clients which are then transformed into instances of the classes obtained from the schemata transforma-

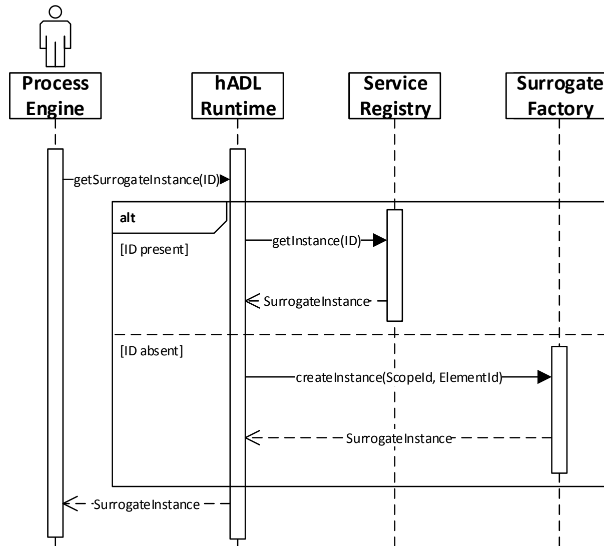

Figure 5.3: UML sequence diagram for obtaining an surrogate instance from the hADL runtime.

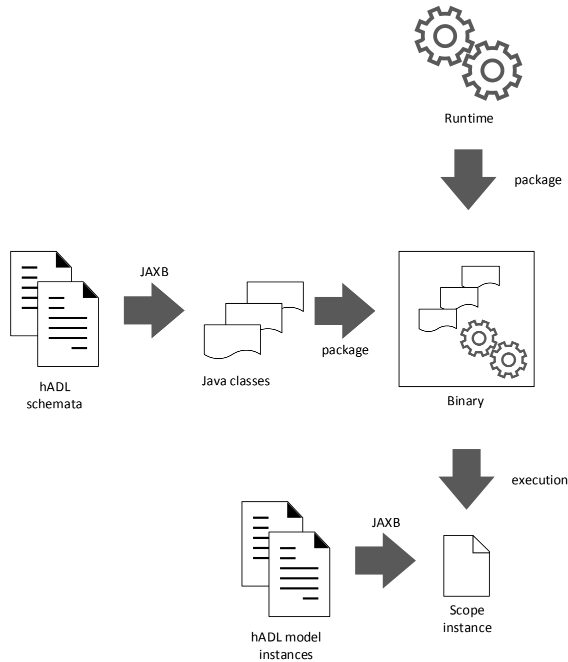

Figure 5.4: Process of converting hADL model instances into scope instances.

tion on demand. The `ScopeId` represents the filename of the hADL model instance. The class instances are finally composed into a scope instance by the marshaller.

5.3 Surrogate Factory

The surrogate factory creates surrogate instances according to scope instances. A scope instance has to define surrogate classes for each hADL element according to the *hADL executable* extension. The surrogate classes are instantiated by the factory on client requests. Thus third party developers have to specify the surrogate mappings and supply the implementations. In order to support that, the surrogate factory has to offer the following mechanisms:

Loading third party classes: Third party developers have to supply their classes as standard Java *JAR* archives. The surrogate factory defines a folder where all third party archives have to be located. This directory is monitored by a *watcher service* which notifies the surrogate factory when archives have been supplied, changed or removed. Then the factory adapts its *classloaders* according to the changes. The name of the archive defines the relationship to the collaboration pattern, i.e. the name of the hADL file and the archive have to match (excluding the file extension).

Manage multiple patterns: Each collaboration pattern has its own surrogate classes, thus each pattern has a separate classloader to minimize conflicts. The classloaders are held in a key/value store where the pattern name represents the key.

Instantiate third party classes: With the class definitions obtained from the classloaders, class instances can be constructed via the Java *Reflections* API. The surrogate factory requires the availability of parameterless class *constructors* or a constructor which takes a resource descriptor as argument.

5.4 Service Registry

The service registry provides access to scope manager and surrogate manager instances, to retrieve, share and reuse resources. This component is implemented via *Apache Camel* which is a routing and transport conversion framework for enterprise application integration contexts. It comes with a pluggable service registry [1] amongst other features. The hADL runtime service registry is based on the `SimpleRegistry` of Apache Camel which is basically a key/-value store. It is started along with the runtime. Further more, the hADL runtime publishes all instances to the registry where the instance's `ID` also serves as service identifier. Registry operations (see Figure 4.14) are simply implemented as follows:

`register`: Put the object into the map with the ID as key.

`retrieve`: Get the object with the given key from the map.

`unregister`: Remove the given ID from the map.

[1] http://camel.apache.org/registry.html

Algorithm 5.1: Algorithm for wiring surrogates within a scope manager.

name: `wire`
input: *operation manager* om

```
1 targetElements ← extractEndpoints(om);
2 for target in targetElements do
3 |   instances ← all wired instances of element target in scope;
4 |   om .connectTo(scope, instances);
5 end
```

Algorithm 5.2: Algorithm for unwiring surrogates within a scope manager.

name: `unwire`
input: *operation manager* om

```
1 targetElements ← extractEndpoints(om);
2 for target in targetElements do
3 |   instances ← all wired instances of element target in scope;
4 |   om .disconnectFrom(scope, instances);
5 end
```

5.5 Scope Manager

Scope managers represent instantiated collaboration patterns with their resources. They contain scope instances which describe pattern structures and keep track of instances of the pattern's elements. The implementation of operations defined in Section 4.5 are described in the following:

`acquire`: Each of the operation managers passed to this method is acquired for this scope in parallel and whenever the acquisition is successful, the instance is put into a key/value store where the key is the element's name and the value is a set of instances of operation managers for the underlying element's surrogate. Instances where the acquisition failed, are not registered in the store. The resulting observable emits a single event indicating if all instances have been successfully acquired for this scope.

`wire`: This operation connects the passed operation managers to element instances already wired in the scope. Algorithm 5.1 describes the behaviour of the method with the additional functions defined in Algorithm 5.3. Wired instances are put into a separate map.

`unwire`: As the counterpart to `wire`, the method disconnects the surrogates from the given scope (see Algorithm 5.2 and Algorithm 5.3).

`release`: The behaviour of this method is equal to `acquire` although the `release` method of the operation manager is called instead.

Algorithm 5.3: This algorithm generates a list of target elements for the input element according to the collaboration pattern structure represented by the scope instance.

```
   name  : extractEndpoints
   input : hADL collaboration element element, collaboration pattern scope
   output: Set of collaboration elements endpoints
 1 for link in scope .collaborationLinks do
 2 |  sourceElement ← mapToElement (link.objActionEndpoint, scope);
 3 |  targetElement ← mapToElement (link.objActionEndpoint, scope);
 4 |  if names of sourceElement and element are equal then
 5 |  |  add sourceElement to endpoints;
 6 |  end
 7 |  if names of targetElement and element are equal then
 8 |  |  add targetElement to endpoints;
 9 |  end
10 end
   name  : mapToElement
   input : hADL action endpoint endpoint, collaboration pattern scope
   output: hADL collaboration element result
11 for ce in scope .collaborationElement do
12 |  for action in ce.actions do
13 |  |  if names of action and endpoint are equal then
14 |  |  |  return ce as result;
15 |  |  end
16 |  end
17 end
```

The implementation of the wiring and unwiring features allows a step by step construction of the collaboration patterns, hence the instantiation of subgraphs of patterns is supported. It also allows an arbitrary number of instances per element to be wired at any given moment under the assumption that the resources have already been acquired for the particular scope. Figure 5.5 illustrates the wiring of a simplified *chatroom* pattern based on Figure 2.5. Collaboration objects have been omitted for clarity purposes. Coloured links between components instances indicate new links which are established when a surrogate instance is wired.

In the example, a client to the hADL runtime has already acquired the pattern elements and wants to wire them. The `wire` method of the scope manager (SM) is called with all element instances. First the *Moderator* has to be wired and since no other wired instances are available yet, no new connection is established and the moderator is put into the map. Then a *Messenger* needs to wired. A SM checks the hADL model instance and identifies a link between moderator and messenger which leads to a new connection between them. Then the messenger is put into the map. This process is repeated until all instances are wired.

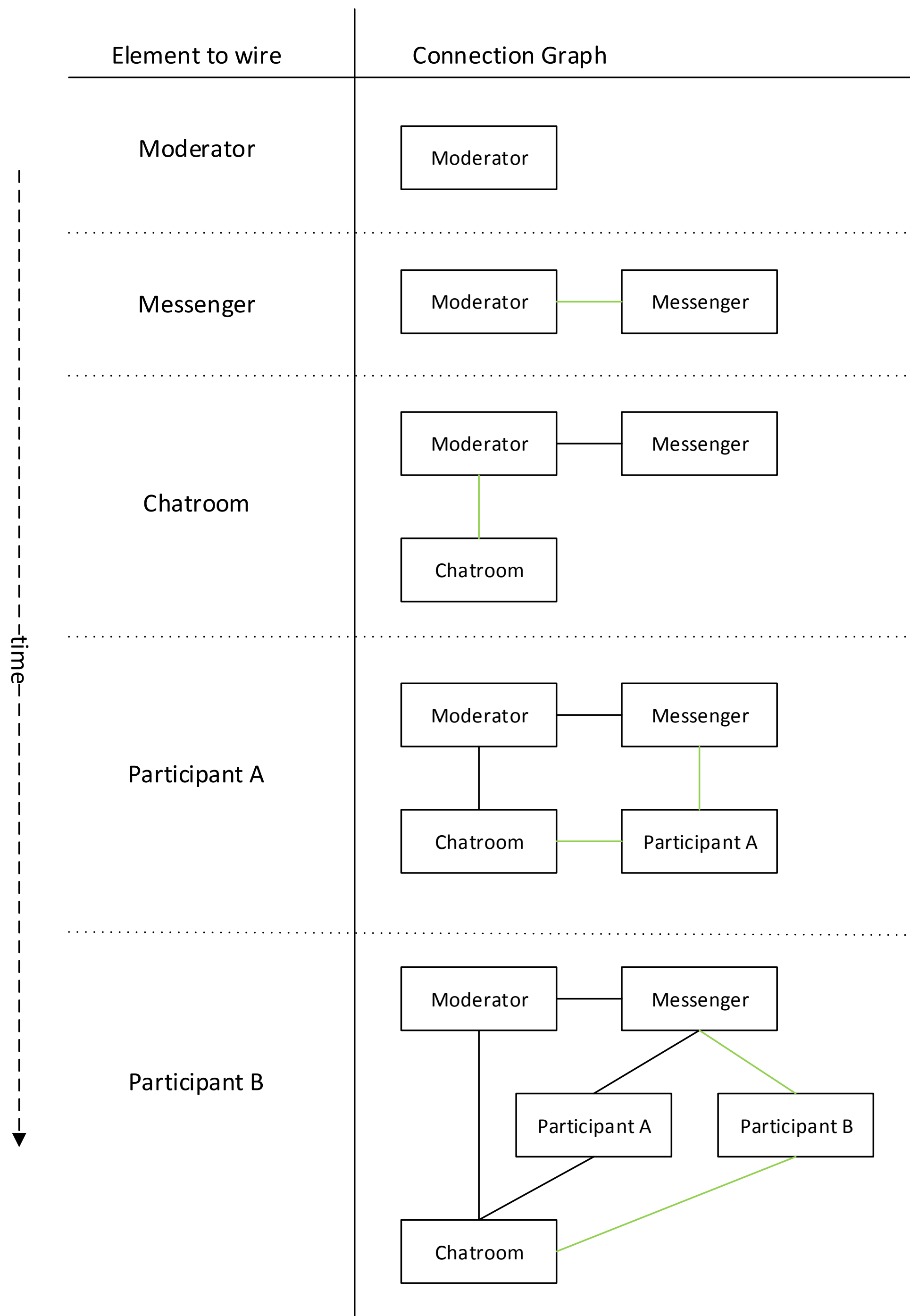

Figure 5.5: Step by step construction of a collaboration pattern instance. Coloured links indicate connections which have been added at the new step.

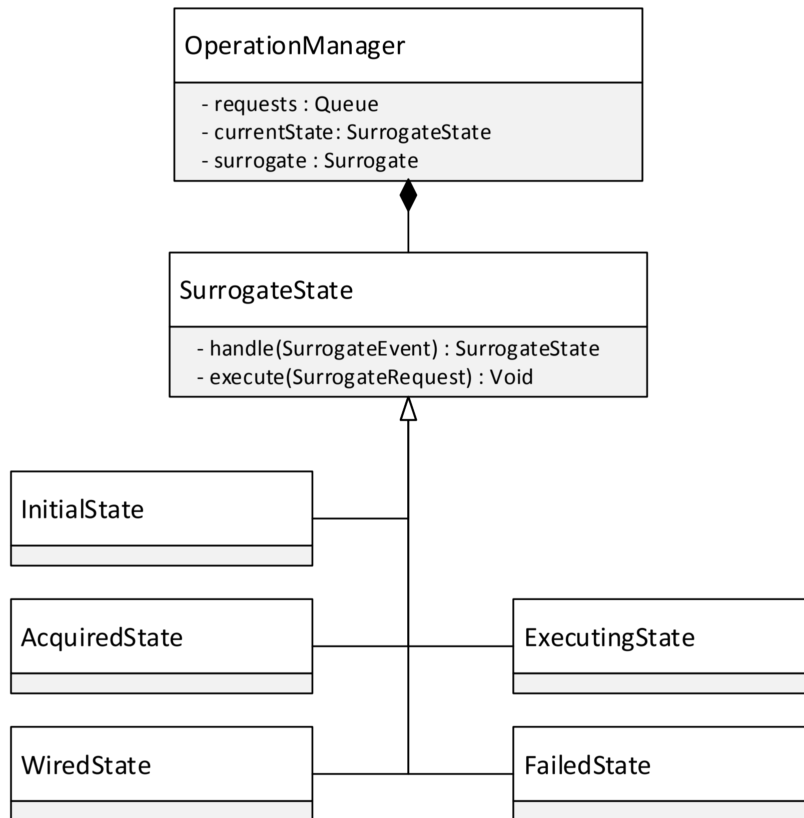

Figure 5.6: UML class diagram of the state pattern tracking surrogate states at the operation manager.

5.6 Operation Manager

As described in Section 4.5, the operation manager has to serialize incoming calls into requests, put them into a thread-safe queue and manage the state of the surrogates.

The state-machine is implemented via the *state*-design pattern (see Figure 5.6). Serialized requests are taken from the queue and executed by the current internal state. State transitions are triggered by events emitted by the underlying surrogate.

The requests are implemented with the *strategy*-design pattern (see Figure 5.7). Each surrogate method is encoded as separate request class and the parameters of the call are stored internally. The state's `execute` method calls the `executeFor` method of the request which in turn calls the surrogate implementation of third party developers.

The operation manager also provides basic facilities for error handling. By employing the *decorator*-design pattern (see Figure 5.7), the execution of the request can be wrapped into arbitrary means of error handling, such as retrying the requests and timeouts.

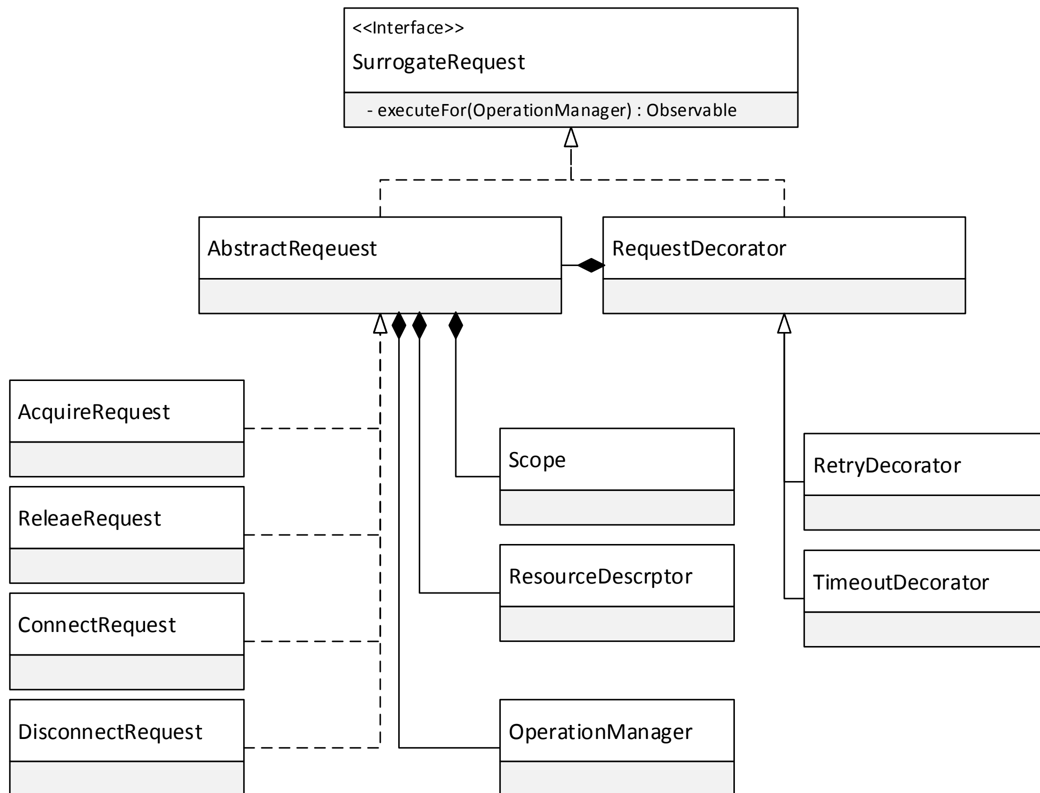

Figure 5.7: UML class diagram of the strategy and decorator pattern for executing surrogate requests at the operation manager.

Surrogates

The operation managers act as façades for the underlying surrogates, which only define an interface with asynchronous operations. A designated base class for third party surrogate implementations have been provided (see Appendix A.2). A base class enforces asynchrony via the *RxJava* framework which is *"... a library for composing asynchronous and event-based programs by using observable sequences."*[2] The core components of the framework are *Observables* which can be seen as a realisation of the publish/subscribe pattern. Observables also represent streams of objects of a given type, which are for example *SurrogateEvents* in the context of the human architecture implementation framework. RxJava advocates the application of the *reactive pattern* where the components react to events, such as property changes and user input to execute the business logic. Additionally convenience and utility functions are provided to simplify the coordination of multi-threaded tasks. Listing 5.1 showcases the synchronisation mechanism for multiple concurrent asynchronous method invocations. In *Line 8*, the `connectTo` method of the

[2]https://github.com/ReactiveX/RxJava/wiki

AsyncSurrogate (see Appendix A.2) is called which immediately returns an Observable. This line is executed for each target endpoint. The observables are collected in a list and passed to combineLatest which blocks until all observables have emitted at least one value. In *Line 19* a callback is specified which is invoked whenever a SurrogateEvent is raised by the observables. It simply counts failures which determines the overall result of the method (see *Line 24* and *Line 29*). The main advantage of *RxJava* in comparison to the *Java Concurrent API* are the *Observable Operators*. They offer stream manipulation behaviour such as merging multiple streams and transforming the objects and can be chained together. A plethora of operators is predefined and an exhaustive list can be obtained from the documentation.[3]

```
1  @Override
2  public Observable<SurrogateEvent> executeFor(Surrogate surrogate) {
3      final Set<SurrogateEvent> failedConnections = Collections
4          .synchronizedSet(new HashSet<>());
5
6      List<Observable<SurrogateEvent>> observables = new ArrayList<>();
7      for (final OperationManager man : endpoints) {
8          Observable<SurrogateEvent> obs = surrogate.connectTo(
9              scope, man.getSurrogate());
10         obs.subscribe(event -> {
11             if (!SurrogateStatus.WIRING_SUCCESS.equals(event.getStatus())) {
12                 failedConnections.add(event);
13             }
14         });
15         observables.add(obs);
16     }
17
18     Observable.combineLatest(observables, os -> os);
19
20     if (failedConnections.size() > 0) {
21         Set<Throwable> causes = failedConnections.stream()
22             .map(event -> event.getOptionalEx())
23             .collect(Collectors.toSet());
24         return Observable.just(new SurrogateEvent(surrogate,
25             SurrogateStatus.WIRING_FAILED,
26                 new MultiSurrogateException(surrogate,
27                     "connection failed", causes)));
28     } else {
29         return Observable.just(new SurrogateEvent(surrogate,
30             SurrogateStatus.WIRING_SUCCESS));
31     }
32 }
```

Listing 5.1: RxJava example for implementing ConnectRequest.

[3]http://reactivex.io/documentation/operators.html

5.7 Deployment and Execution

The human implementation framework has been developed using *JAVA 1.8* and can easily be build using *Apache Maven 3*. Build-artefacts can be generated using: `mvn clean install` executed in the top folder of the source code. This command generates a *JAR* which can be added to any third party projects as required. The source code of the framework can be found at `https://bitbucket.org/x_zhang/hadlruntime.git`.

CHAPTER 6

Evaluation

In the last chapters, the design (see Chapter 4) and implementation (see Chapter 5) of the framework have been proposed which have to be evaluated according to the methods defined in Chapter 3. This chapter is structured similar to the scenario descriptions at Section 3.1. Each scenario is described in a separate section where each section contains a solution addressing the requirement and a textual argumentation as evaluation.

6.1 REQ-01: Supporting Architecture Driven Development

The ADD approach has been introduced and illustrated in Figure 3.2. It is a macroscopic iterative approach to software engineering where the software architecture is developed, evaluated and changed along the lifespan of the system under development. Each of the steps will be evaluated against the framework in the following:

Collect Architectural Requirements: Collecting architectural requirements is essential to any system. It can be supported with established methods and software tools to document the findings, nevertheless it is ultimately an organizational issue which has to be solved via structured communication. A technical framework such as the proposed human architecture implementation framework cannot determine the requirements.

Design the Architecture: By enforcing the use of process descriptions and hADL, a structural and behavioural decomposition of the system can be guaranteed. Experiences in practice suggest the usefulness of process descriptions and its weakness regarding the system's structure. The shortcoming is addressed by hADL which employs the classical concept of components and connectors and extends them with aspects related to human collaboration. Thus additionally to the software structure, the structure of collaboration between human and software can also be designed at macroscopic level.

Document the Architecture: The design step yields at least a behavioural documentation based on the process language in use and a structural documentation based on hADL. Thus

higher level of completeness can be assured compared to an approach solely based on one of the languages. Mesoscopic and microscopic design decisions at process steps and within components, connectors and collaboration objects still have to be documented additionally and cannot be enforced.

Analyse the Architecture: The higher level of completeness regarding the architecture documentation favours the analysis as well. Traditional architecture analysis methods such as SAAMs (see Section 3.3) can still be applied to both models. The models can also be augmented with quality attributes to enable automated reasoning or verification.

Realize the Architecture: The implementation of the architecture can directly be derived from the process description and the hADL description. The process model and its engine are already essential parts of the implementation at top level. This holds equally for the hADL model instance and its runtime. Thus creating the documentation innately realizes the architecture as well.

Maintain the Architecture: The system's evolution over time is always synchronized with the architectural views at macroscopic level due to the direct mapping of architecture elements to their implementation by the runtime. Its architecture can thus always be reconstructed. Deviations of the descriptive view from the prescriptive view should only occur within hADL elements.

The argumentation above suggests that the human architecture implementation framework supports architecture driven software development and even enforces higher level of completeness in respect to software architecture documentation. The framework also simplifies the synchronization between prescriptive and descriptive view which represent the design as intended and the design as implemented respectively due to the direct mapping.

6.2 REQ-02: Modelling the Architecture

To evaluate the framework's capability to adequately model software systems with human participation, a interest based bargaining process is designed. The IBB process is illustrated in Figure 6.1. In process model, IBB is reduced to a sequence of phases (steps) each defining input and output resources. Any data defined in the following uses a format similar to *JSON* [1]. Each process step requires an executing agent which signals the success or failure of the step to the process engine. The exact internal architecture of the agent is not relevant as it depends on the process engine's API. Each subsequent section contains exemplified input and output data as well as behavioural (prosaic) and structural (hADL) descriptions of the elements. In particular the `acquire` and `wire` method of the surrogates will be elaborated.

The process and hADL model instances are kept simple for clarity and brevity reasons as the expressiveness of both ADLs for their respective domains have already been established in literature. Collaboration patterns introduced in Section 2.4 will be employed by the architecture.

[1] http://www.json.org/

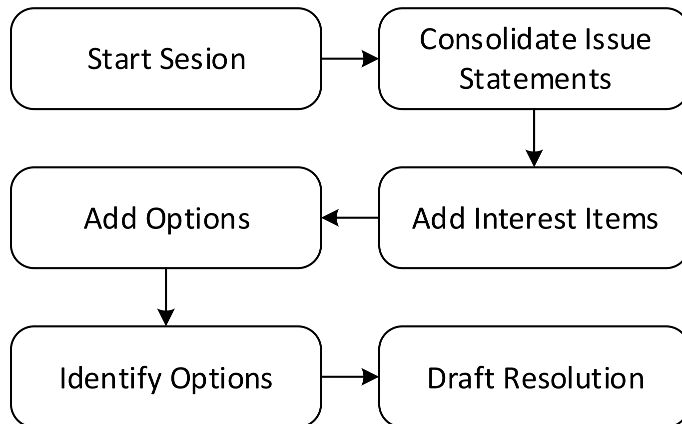

Figure 6.1: Process model of the IBB process.

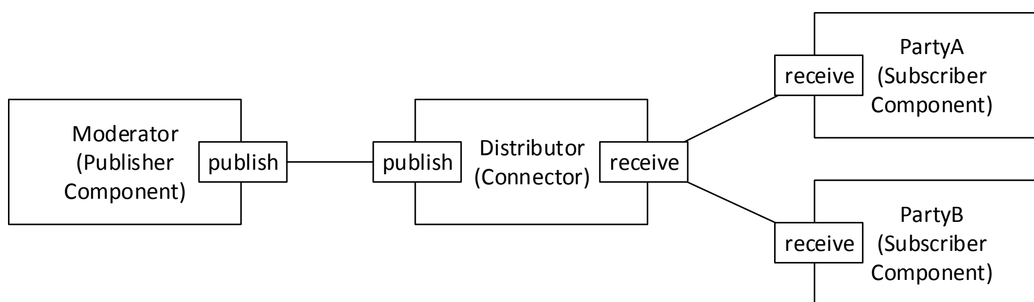

Figure 6.2: Publish/Subscribe collaboration pattern for IBB.

Start Session

The mediator of the IBB has to inform the parties about the process and gather their contact details, assuming that some contact information is available a priori (e.g. obtained from a different process). Informations about IBB could not be delivered otherwise.

Required resource for the first step is described in Listing 6.1. One instance of each element (see Figure 6.2) is requested. The runtime acquires and then wires all elements. Instance identifiers are in the step's outcome (see Listing 6.2).

Moderator : The surrogate acquires the moderator simply by sending an email indicating the start of a new IBB process instance along with the parties's information. After the moderator has been wired to the distributor, the parties can be informed about IBB via a mailing list.

PartyA/PartyB : Acquiring the party is also implemented as delivering an email notification. The functionality for wiring is equal to subscribing to the mailing list.

Distributor : The surrogate creates a new mailing list for the process when the distributor is acquired. When wiring, publishers and subscribers are registered for the list.

After the wiring is complete, the mediator component has to inform the participants and afterwards has to signal the end of the step to the step's executing agent. This holds analogously for all subsequent steps.

```
1  {
2   Scope: {
3    ScopeID: "ibb-start.xml",
4    Elements: {
5     Moderator: { ModeratorA: { email: "mod@example.com" } },
6     PartyA: { Bob: { email: "bob@partyA.com" } },
7     PartyB: { Alice: { email: "alice@partyB.com" } },
8     Distributor: {}
9    }
10  }
11 }
```

Listing 6.1: Input resources when starting the IBB process

```
1  {
2   Scope: {
3    InstanceID: "s1",
4    Elements: {
5     Moderator: {  ModeratorA: { InstanceID: "m1" } },
6     PartyA: { Bob: { InstanceID: "p1" } },
7     PartyB: { Alice: { InstanceID: "p2" } },
8     Distributor: { InstanceID: "d1" }
9    }
10  }
11 }
```

Listing 6.2: Output resources of the IBB start step.

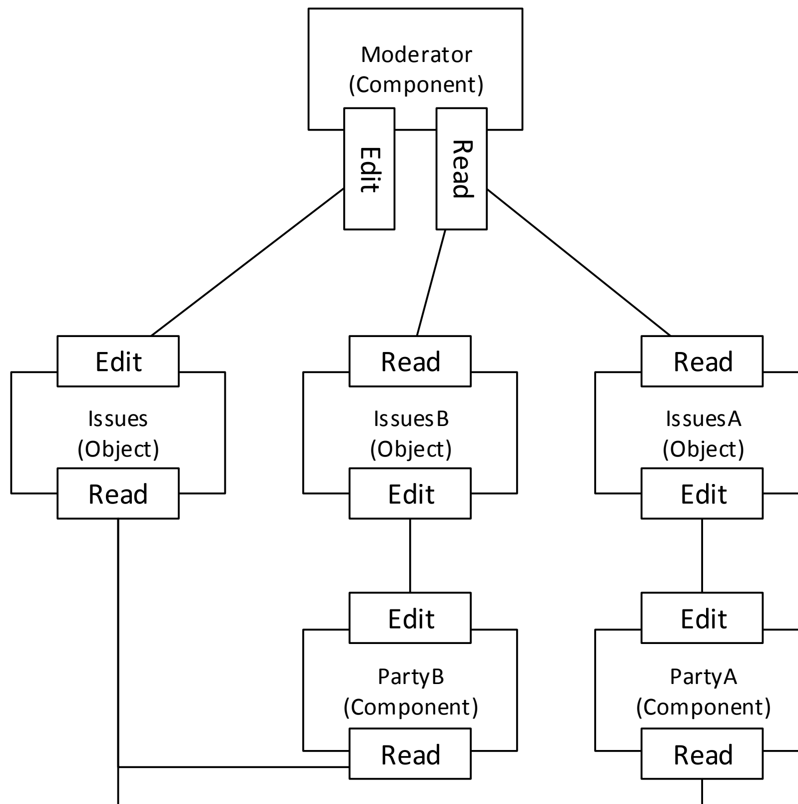

Figure 6.3: Shared artefact collaboration pattern for IBB.

Consolidate Issue Statements

In this step, the moderator collects the issue statements from the parties and consolidates them into a single document. The collaboration pattern is depicted in Figure 6.3. Each party has a separate artefact (`IssuesA` and `IssuesB`) shared with the moderator to collect their respective issue statements. Moderators consolidate the statements into `Issue` artefacts which both parties are allowed to read. Resource definitions can be obtained from Listing 6.3 and Listing 6.4. Moderator and both parties from the last step are reused.

The surrogates have to acquire the shared artefacts which are documents on *GoogleDocs* in this case. Thus a document has to be created on the platform according to the platform's API. When the components are wired, the surrogates have to give the relevant users read or write permissions as defined in the collaboration pattern.

```
{
 Scope: {
  ScopeID: "ibb-issues.xml",
  Elements: {
   Moderator: {  ModeratorA: { InstanceID: "m1" } },
   PartyA: { Bob: { InstanceID: "p1" } },
   PartyB: { Alice: { InstanceID: "p2" } },
   IssueA: { doc: {} },
   IssueB: { doc: {} },
   Issues: { doc: {} }
  }
 }
}
```

Listing 6.3: Input resources issue statements.

```
{
 Scope: {
  InstanceID: "s2",
  Elements: {
   Moderator: { ModeratorA: { InstanceID: "m1" } },
   PartyA: { Bob: { InstanceID: "p1" } },
   PartyB: { Alice: { InstanceID: "p2" } },
   IssueA: { doc: { InstanceID: "docA" } },
   IssueB: { doc: { InstanceID: "docB" } },
   Issues: { doc: { InstanceID: "docC" } }
  }
 }
}
```

Listing 6.4: Output resources after consolidating issue statements.

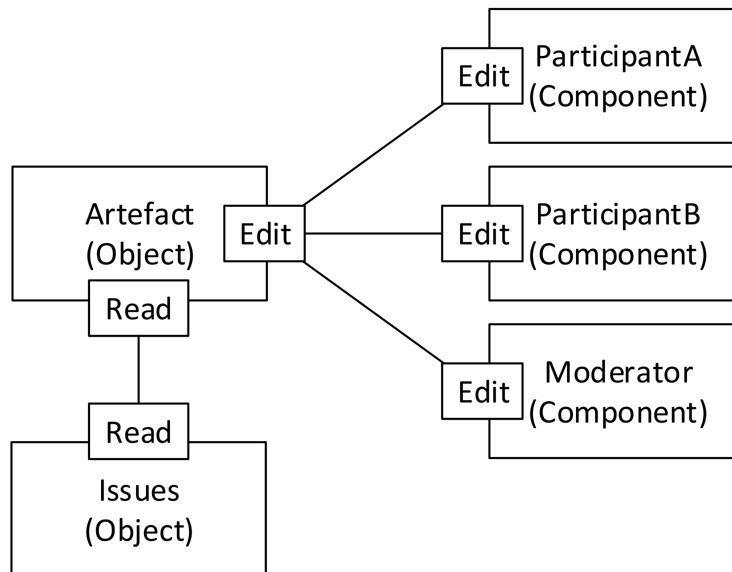

Figure 6.4: Shared artefact collaboration pattern for IBB where all components may edit.

Add Interest Items

After the consolidation of the issue statements, parties are allowed to add interest items to the newly created shared artefact. The collaboration pattern is illustrated in Figure 6.4. Shared artefacts grant every stakeholder edit permissions, i.e. read and write. Surrogates of the `Artefact` create new documents based on the data extracted from the collaboration objects of the last step (`Issues`). Input and output of this step are exemplified in Listing 6.5 and Listing 6.6 respectively.

```
1  {
2   Scope: {
3    ScopeID: "ibb-shared.xml",
4    Elements: {
5     Moderator: { ModeratorA: { InstanceID: "m1" } },
6     PartyA: { Bob: { InstanceID: "p1" } },
7     PartyB: { Alice: { InstanceID: "p2" } },
8     Issues: { doc: { InstanceID: "docC" } },
9     Artefact: { doc: {} }
10   }
11  }
12 }
```

Listing 6.5: Input resources when collecting interest items.

```
{
 Scope: {
  ScopeID: "s3",
  Elements: {
   Moderator: { ModeratorA: { InstanceID: "m1" } },
   PartyA: { Bob: { InstanceID: "p1" } },
   PartyB: { Alice: { InstanceID: "p2" } },
   Issues: { doc: { InstanceID: "docC" } },
   Artefact: { doc: { InstanceID: "docD"} }
  }
 }
}
```

Listing 6.6: Output resources after collecting interest items.

Add Options

When the interest items have been identified and fixated, the parties add options to the items which represent acceptable solutions for the issue and interest. The scope from the last step is reused as the collaboration pattern is very similar. Input and output are thus analogous to the previous step.

Identify Options

After all parties have added their options, a voting with the options is created to let the parties choose acceptable solutions. Figure 6.5 depicts the pattern. The `Survey` collaboration object reads the options from `Artefact` obtained at the previous step and creates a survey. When wiring the `Survey`, the location is transmitted to the participants. In this example, *Google Forms* is used as voting framework and a link to the form is send to the user. The resources are described in Listing 6.7 and Listing 6.8.

```
{
 Scope: {
  ScopeID: "ibb-vote.xml",
  Elements: {
   PartyA: { Bob: { InstanceID: "p1" } },
   PartyB: { Alice: { InstanceID: "p2" } },
   Survey: { form: {} },
   Artefact: { doc: { InstanceID. "docD"} }
  }
 }
}
```

Listing 6.7: Input resources when voting for options.

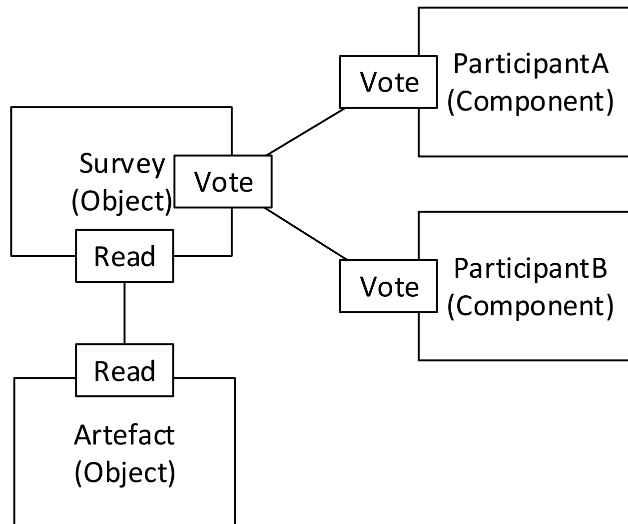

Figure 6.5: Voting collaboration pattern for IBB.

```
{
 Scope: {
  ScopeID: "s4",
  Elements: {
   Moderator: { ModeratorA: { InstanceID: "m1" } },
   PartyA: { Bob: { InstanceID: "p1" } },
   PartyB: { Alice: { InstanceID: "p2" } },
   Survey: { form: { InstanceID: "form1" } },
   Artefact: { doc: { InstanceID: "docD"} }
  }
 }
}
```

Listing 6.8: Output resources after voting for options.

Draft Resolution

In the last step a resolution is finally created from the chosen options. The moderator drafts a solution agreement for both parties and informs them using the mailing list acquired in the first step (see Figure 6.2).

6.3 REQ-03: Change in Collaboration Structure

The implementation of the IBB process introduced in the previous sections provides the basis for architectural change and evolution which is inevitable for long-lived software systems. This

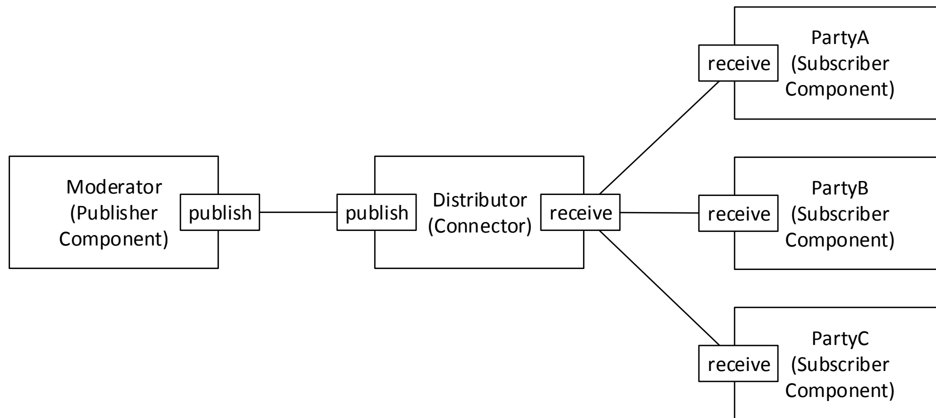

Figure 6.6: IBB publish/subscribe pattern with an addtional party.

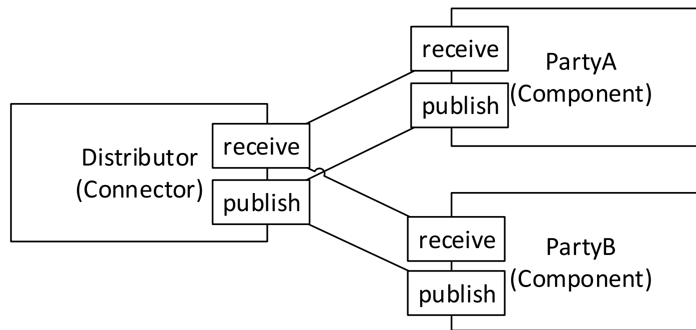

Figure 6.7: IBB publish/subscribe pattern where the moderator has been removed.

evaluation considers a change in the collaboration structure in the process's first step which is an incarnation of the publish/subscribe pattern (see Figure 6.2). In this context, at least two modifications might occur. First, a new element can be added (see Figure 6.6). Second, an existing element might be deleted from the model (see Figure 6.7).

When adding a component, regardless of leaf or non-leaf, the interface in respect to resource outputs stays compatible to subsequent steps. In Figure 6.6 the `PartyC` component is added to the model. Thus a new entry has to be added to the resource declaration in Listing 6.1 resulting in Listing 6.9. The output declaration does not have to be changed, if `PartyC` is not required afterwards. Otherwise the output would be similar to Listing 6.2. This also holds analogously for new connectors, collaboration objects, links and actions.

Removing elements from the hADL model triggers changes in all steps containing references to the deleted elements. The adaptation of the resource declarations clearly cannot be avoided.

```
{
 Scope: {
  ScopeID: "ibb-start.xml",
  Elements: {
   Moderator: { ModeratorA: { email: "mod@example.com" } },
   PartyA: { Bob: { email: "bob@partyA.com" } },
   PartyB: { Alice: { email: "alice@partyB.com" } },
   PartyC: { Alice: { email: "mike@partyC.com" } },
   Distributor: {}
  }
 }
}
```

Listing 6.9: Input resource of the IBB start step after adding an element to the collaboration pattern.

```
{
 Scope: {
  InstanceID: "s1",
  Elements: {
   Moderator: {  ModeratorA: { InstanceID: "m1" } },
   PartyA: { Bob: { InstanceID: "p1" } },
   PartyB: { Alice: { InstanceID: "p2" } },
   PartyC: { Alice: { email: "mike@partyC.com" } },
   Distributor: { InstanceID: "d1" }
  }
 }
}
```

Listing 6.10: Output resource of the IBB start step after adding an element to the collaboration pattern.

6.4 REQ-04: Change in Collaboration Tool

Whenever underlying tools of the collaboration pattern change in respect to their interfaces, the frameworks models have to be updated too, although only minor adaptations are necessary. For example, the publish/subscribe pattern described in previous sections is implemented via emails at first. The company then decides to use an instant messaging system instead, such as SMS via a dedicated gateway.

Two adaptations are required to switch the underlying implementations. First, the hADL model instance has to be modified. In particular, the new surrogate has to be registered in the XML tag `executableViaSurrogate` of the `Distributor` component. Second, the surrogate implementations have to be updated. In order to do that, the binaries in the surrogate factory's watch folder have to be updated. No changes in the process and hADL model are required otherwise.

6.5 REQ-05: Handling Human Faults

The runtime provides two fault handling mechanisms. First, whenever a request is submitted to surrogate, a timer is automatically started. If the timer has finished and no responses have been received, an exception indicating a timeout and failure is raised. Whenever a request fails, a retry strategy is applied. Three retries, i.e. new submissions of the original request, are attempted at default.

6.6 REQ-06: Process Engine Integration

The human architecture implementation framework is designed to be integrable into process engines by a resource model. Nevertheless during evaluation, no adapter for a real process engine has been implemented due to the considerable effort required. The process has been simulated by the scenario code instead.

6.7 Test Scenarios

The overall test cases can be found as *JUnit* tests in the repository of the hADL implementation framework.[2] The IBB process has not been developed entirely, as the logic of the steps are highly repetitive. Instead the allocation of hADL collaboration pattern and their lifecycle management have been simulated. A few surrogates have been implemented although no real-world collaboration platform or tool has been integrated.

[2] https://bitbucket.org/x_zhang/hadlruntime.git

CHAPTER 7

Critical Reflection

Based on the evaluation results from the previous chapter, the benefits of combining structural and behavioural ADLs in a single framework become more evident. It has been argued that architecture driven development is adequately supported due to the nature of the framework. The architectural models are essential parts of the developed systems. Consequently, the integration of process engines is also a non-optional aspect of the framework as well.

Changes in collaboration pattern structures are nearly limited only to adaptations in the hADL model instance. Minimal changes are propagated to the process model. The human architecture framework provides a clear approach for developing human intensive software applications and also offers a runtime to reduce developing repetitive components. Changes in implementations can easily be realised in the hADL model alone. Nevertheless a few open issues are still unresolved and are subject to further research.

First, hADL is not fully supported. The hADL executable extension contains the capabilities to assign a *collaboration platform* to each surrogate. Thus a platform is a subset of all available surrogates which share a common implementation platform such as *Google Documents*. The runtime could allow the specification of the platform during execution. This offers developers means to switch the implementation of multiple surrogates at once.

At core level, hADL does not allow direct connections between collaboration pattern elements. Only links between actions of elements are possible. The runtime on the other hand interprets the links as relationships between their parent elements and uses that fact as shortcut to wire the surrogates. It requires a finer granularity regarding connections and adequate means to oversee actions of surrogates.

Monitoring allocated resources and reconstructing the architecture at runtime are important features for real world applications. The framework does not support queries for runtime statuses explicitly although the internal representation of the structure can be mapped to a suitable representation such as *hADL runtime* for monitoring purposes. This extension assigns operational states to elements and summarizes their scope membership and active links.

Elements of each collaboration pattern are currently independent. Thus element instances within scopes cannot be reused in other scopes. By allowing the definition of mappings between

elements in different patterns, equivalence relations can be established. This approach is similar to creating aliases and enables the splitting of large hADL models.

The fault handling mechanisms are statically integrated into the runtime and cannot be easily extended by third party developers. Additional capabilities to allow specification and injection of fault handlers are still necessary.

Code generation as a feature has been entirely absent in the framework. Evaluating the generation of high level service interfaces based on actions of hADL elements is subject to future research. The generated code could be automatically enriched with authentication and authorization mechanisms to provide access control for surrogates.

Binaries, i.e. the implementations of the surrogates supplied by third party developers, cannot be updated during runtime. The system has to reboot and reset the Java classloaders to avoid class version conflicts which imposes an inconvenience.

The framework could come with predefined collaboration patterns and their implementations on top of common collaboration platforms. In particular the patterns introduced in Section 2.4 are reasonable candidates.

CHAPTER 8

Conclusion

This thesis has evaluated the synergy effects of combining structural and behavioural architecture description languages, especially hADL in the context of describing and prescribing software architectures for human intensive software systems. The expressiveness of a combined approach has been incorporated into a software development framework, the *Human Architecture Implementation Framework* (HAIF) which contains aspects of a development process and a technical framework. This thesis concludes that:

1. HAIF supports architecture driven development as architecture definitions are essential elements of the approach (see **REQ-01**).
2. The combination of behavioural and structural ADLs yields a more complete architectural documentation (see **REQ-02**).
3. A system based on HAIF embraces architectural evolution over time and thus eases the maintenance of the overall architecture (see **REQ-03**).
4. Systems based on HAIF have integrated adaptation mechanisms as implementations can easily be exchanged due to the standardised *Surrogate* interface (see **REQ-04**).

Human intensive software applications require systems to be adaptable and tolerant to human idiosyncrasies. To this end, the framework offers flexibility in architecture implementation and evolution as well as a simple extensible fault handling strategies.

The HAIF prototype can be seen as a proof of concept and the set of features provided are still very limited. The current implementation is thus not ready for production yet (see Chapter 7). Nevertheless a small step towards a standardised approach for implementing human intensive software applications has been made.

Bibliography

[1] IEEE recommended practice for architectural description of software-intensive systems. pages i–23.

[2] Gregory Abowd, Robert Allen, and David Garlan. Using style to understand descriptions of software architecture. 18(5):9–20.

[3] Steve Adolph, Alistair Cockburn, and Paul Bramble. *Patterns for Effective Use Cases*. Addison-Wesley Longman Publishing Co., Inc.

[4] Salman Ahmad, Alexis Battle, Zahan Malkani, and Sepander Kamvar. The jabberwocky programming environment for structured social computing. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, UIST '11, pages 53–64. ACM.

[5] Vamshi Ambati, Stephan Vogel, and Jaime Carbonell. Collaborative workflow for crowdsourcing translation. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, CSCW '12, pages 1191–1194. ACM.

[6] Muhammad Ali Babar and Ian Gorton. Comparison of scenario-based software architecture evaluation methods. In *Software Engineering Conference, 2004. 11th Asia-Pacific*, pages 600–607. IEEE.

[7] Muhammad Ali Babar, Liming Zhu, and Ross Jeffery. A framework for classifying and comparing software architecture evaluation methods. In *Software Engineering Conference, 2004. Proceedings. 2004 Australian*, pages 309–318. IEEE.

[8] Felix Bachmann, Len Bass, Jeromy Carriere, Paul C Clements, David Garlan, James Ivers, Robert Nord, and Reed Little. Software architecture documentation in practice: Documenting architectural layers.

[9] Albert-László Barabási. Network theory–the emergence of the creative enterprise. 308(5722):639–641.

[10] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. SEI Series in Software Engineering. Pearson Education.

[11] U.M. Borghoff and J.H. Schlichter. *Computer-Supported Cooperative Work: Introduction to Distributed Applications*. Springer Berlin Heidelberg.

[12] Alessandro Bozzon, Marco Brambilla, Stefano Ceri, and Andrea Mauri. Reactive crowdsourcing. In *Proceedings of the 22Nd International Conference on World Wide Web*, WWW '13, pages 153–164. International World Wide Web Conferences Steering Committee.

[13] Marco Brambilla, Piero Fraternali, and Carmen Vaca. BPMN and design patterns for engineering social BPM solutions. In Florian Daniel, Kamel Barkaoui, and Schahram Dustdar, editors, *Business Process Management Workshops (1)*, volume 99 of *Lecture Notes in Business Information Processing*, pages 219–230. Springer.

[14] J. Buford, K. Mahajan, and V. Krishnaswamy. Federated enterprise and cloud-based collaboration services. In *Internet Multimedia Systems Architecture and Application (IMSAA), 2011 IEEE 5th International Conference on*, pages 1–6.

[15] John Buford, Kishore Dhara, Venky Krishnaswamy, Xiaotao Wu, and Mario Kolberg. Work in progress: A CommunicationsEnabled collaboration platform. page 161.

[16] AG. Cass, AS. Lerner, E.K. McCall, L.J. Osterweil, S.M. Sutton, and A Wise. Little-JIL/juliette: a process definition language and interpreter. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 754–757.

[17] David Chappell. *Enterprise service bus*. O'Reilly Media, Inc.

[18] Michael Chui. *The social economy: Unlocking value and productivity through social technologies*. McKinsey.

[19] Lori A. Clarke, Leon J. Osterweil, and George S. Avrunin. Supporting human-intensive systems. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pages 87–92. ACM.

[20] Paul C. Clements. A survey of architecture description languages. In *Proceedings of the 8th International Workshop on Software Specification and Design*, IWSSD '96, pages 16–. IEEE Computer Society.

[21] Gerardine Desanctis and R Brent Gallupe. A foundation for the study of group decision support systems. 33(5):589–609.

[22] Anhai Doan, Raghu Ramakrishnan, and Alon Y. Halevy. Crowdsourcing systems on the world-wide web. 54(4):86–96.

[23] Liliana Dobrica and Eila Niemelä. A survey on software architecture analysis methods. 28(7):638–653.

[24] C. Dorn and R.N. Taylor. Coupling software architecture and human architecture for collaboration-aware system adaptation. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 53–62.

[25] Christoph Dorn, Schahram Dustdar, and LeonJ. Osterweil. Specifying flexible human behavior in interaction-intensive process environments. In Shazia Sadiq, Pnina Soffer, and Hagen Völzer, editors, *Business Process Management*, volume 8659 of *Lecture Notes in Computer Science*, pages 366–373. Springer International Publishing.

[26] Christoph Dorn and Richard N. Taylor. Analyzing runtime adaptability of collaboration patterns. In *International Conference on Collaboration Technologies and Systems (CTS 2012)*.

[27] Christoph Dorn and Richard N. Taylor. Architecture-driven modeling of adaptive collaboration structures in large-scale social web applications. In *13th International Conference on Web Information System Engineering (WISE 2012)*.

[28] Christoph Dorn, Richard N. Taylor, and Schahram Dustdar. Flexible social workflows: Collaborations as human architecture. 16(2):72–77.

[29] Christoph Dorn and R.N. Taylor. Co-adapting human collaborations and software architectures. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 1277–1280.

[30] Clarence A. Ellis, Simon J. Gibbs, and Gail Rein. Groupware: Some issues and experiences. 34(1):39–58.

[31] R.D. Evans, J.X. Gao, N. Martin, and C. Simmonds. Using web 2.0-based groupware to facilitate collaborative design in engineering education scheme projects. In *Interactive Collaborative Learning (ICL), 2014 International Conference on*, pages 397–402.

[32] P.H. Feiler and W.S. Humphrey. Software process development and enactment: concepts and definitions. In *Software Process, 1993. Continuous Software Process Improvement, Second International Conference on the*, pages 28–40.

[33] Roger Fisher, William L Ury, and Bruce Patton. *Getting to yes: Negotiating agreement without giving in*. Penguin.

[34] David Garlan and Dewayne E Perry. Introduction to the special issue on software architecture. 21(4):269–274.

[35] Lars Grammel and Margaret-Anne Storey. A survey of mashup development environments. In Mark Chignell, James Cordy, Joanna Ng, and Yelena Yesha, editors, *The Smart Internet*, volume 6400 of *Lecture Notes in Computer Science*, pages 137–151. Springer Berlin Heidelberg.

[36] Jonathan Grudin. Computer-supported cooperative work: History and focus. (5):19–26.

[37] Roger Guimera, Brian Uzzi, Jarrett Spiro, and Luis A Nunes Amaral. Team assembly mechanisms determine collaboration network structure and team performance. 308(5722):697–702.

[38] Wu He and Li Da Xu. Integration of distributed enterprise applications: A survey. 10(1):35–42.

[39] Dave Ings, Luc Clement, Dieter Koenig, Vinkesh Mehta, Ralf Mueller, Ravi Rangaswamy, Michael Rowley, and Ivana Trickovic. *WS-BPEL Extension for People (BPEL4People) Specification Version 1.1*. OASIS. Published: OASIS Committee Specification.

[40] Ivar Jacobson, Grady Booch, James Rumbaugh, James Rumbaugh, and Grady Booch. *The unified software development process*, volume 1. Addison-wesley Reading.

[41] Benjamin F Jones, Stefan Wuchty, and Brian Uzzi. Multi-university research teams: shifting impact, geography, and stratification in science. 322(5905):1259–1262.

[42] Aniket Kittur, Ed H. Chi, and Bongwon Suh. Crowdsourcing user studies with mechanical turk. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '08, pages 453–456. ACM.

[43] Aniket Kittur, Susheel Khamkar, Paul André, and Robert Kraut. CrowdWeaver: Visually managing complex crowd work. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, CSCW '12, pages 1033–1036. ACM.

[44] Aniket Kittur, Boris Smus, and Robert Kraut. CrowdForge: Crowdsourcing complex work. In *CHI '11 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '11, pages 1801–1806. ACM.

[45] J. Knodel and M. Naab. Software architecture evaluation in practice: Retrospective on more than 50 architecture evaluations in industry. In *Software Architecture (WICSA), 2014 IEEE/IFIP Conference on*, pages 115–124.

[46] N. Kock. *Encyclopedia of E-Collaboration*. ITPro collection. Information Science Reference.

[47] John Krogstie. Modelling languages: Perspectives and abstraction mechanisms. In *Model-Based Development and Evolution of Information Systems*, pages 89–204. Springer London.

[48] Philippe Kruchten. Architectural blueprints—the "4+ 1" view model of software architecture.

[49] P. Lago, I. Malavolta, H. Muccini, P. Pelliccione, and A. Tang. The road ahead for architectural languages. 32(1):98–105.

[50] LindaS.L. Lai and Efraim Turban. Groups formation and operations in the web 2.0 environment and social networks. 17(5):387–402.

[51] Hui Lei, D. Chakraborty, H. Chang, M.J. Dikun, T. Heath, J.S. Li, N. Nayak, and Yasodhar Patnaik. Contextual collaboration: platform and applications. In *Services Computing, 2004. (SCC 2004). Proceedings. 2004 IEEE International Conference on*, pages 197–206.

[52] B. S. Lerner, A. G. Ninan, L. J. Osterweil, and R. M. Podorozhny. Modeling and managing resource utilization for process, workflow and activity coordination TITLE2:.

[53] BarbaraStaudt Lerner, LeonJ. Osterweil, Jr. Sutton, StanleyM., and Alexander Wise. Programming process coordination in little-JIL. In Volker Gruhn, editor, *Software Process Technology*, volume 1487 of *Lecture Notes in Computer Science*, pages 127–131. Springer Berlin Heidelberg.

[54] David S Linthicum. *Enterprise application integration*. Addison-Wesley Professional.

[55] Thomas W Malone, Robert Laubacher, and Chrysanthos Dellarocas. Harnessing crowds: Mapping the genome of collective intelligence.

[56] Nenad Medvidovic and Richard N Taylor. A classification and comparison framework for software architecture description languages. 26(1):70–93.

[57] Duane Merrill. Mashups: The new breed of web app. pages 1–13.

[58] Kevin L Mills. Computer-supported cooperative work. In *ENCYCLOPEDIA OF LIBRARY AND INFORMATION SCIENCES (2ND EDITION*. Citeseer.

[59] C Mohit, Xiaotao Wu, and Venkatesh Krishnaswamy. Integrating enterprise communications into google wave. In *Proceedings of the 7th IEEE conference on Consumer communications and networking conference*, pages 1110–1111. IEEE Press.

[60] L. Northrop, P. Feiler, R. P. Gabriel, J. Goodenough, R. Linger, T. Longstaff, R. Kazman, M. Klein, D. Schmidt, K. Sullivan, K. Wallnau, and W. Pollak. Ultra-large-scale systems - the software challenge of the future.

[61] Tim O'really. Design patterns and business models for the next generation of software.

[62] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. 17(4):40–52.

[63] Klaus Pohl. The three dimensions of requirements engineering. In Janis Bubenko, John Krogstie, Oscar Pastor, Barbara Pernici, Colette Rolland, and Arne Sølvberg, editors, *Seminal Contributions to Information Systems Engineering*, pages 63–80. Springer Berlin Heidelberg.

[64] Jason E. Robbins, Nenad Medvidovic, David F. Redmiles, and David S. Rosenblum. Integrating architecture description languages with a standard design method. In *Proceedings of the 20th International Conference on Software Engineering*, ICSE '98, pages 209–218. IEEE Computer Society.

[65] Colette Rolland. A comprehensive view of process engineering. In Barbara Pernici and Costantino Thanos, editors, *Advanced Information Systems Engineering*, volume 1413 of *Lecture Notes in Computer Science*, pages 1–24. Springer Berlin Heidelberg.

[66] Nick Russell and Wil MP van der Aalst. Evaluation of the BPEL4people and WS-HumanTask extensions to WS-BPEL 2.0 using the workflow resource patterns. 513.

[67] D. Schall, S. Dustdar, and M.B. Blake. Programming human and software-based web services. 43(7):82–85.

[68] D. Schall, Hong-Linh Truong, and S. Dustdar. Unifying human and software services in web-scale collaborations. 12(3):62–68.

[69] Daniel Schall. A human-centric runtime framework for mixed service-oriented systems. 29(5):333–360.

[70] Daniel Schall, Hong-Linh Truong, and Schahram Dustdar. *The human-provided services framework*. Springer.

[71] Mary Shaw and David Garlan. *Software architecture: perspectives on an emerging discipline*, volume 1. Prentice Hall Englewood Cliffs.

[72] Borislava I Simidchieva, Lori A Clarke, and Leon J Osterweil. STORM2: Process-guided online dispute resolution.

[73] R.N. Taylor, N. Medvidovic, and E.M. Dashofy. *SOFTWARE ARCHITECTURE: FOUNDATIONS, THEORY, AND PRACTICE*. Wiley India Pvt. Limited.

[74] Rattapoom Tuchinda, Pedro Szekely, and Craig A. Knoblock. Building mashups by example. In *Proceedings of the 13th International Conference on Intelligent User Interfaces*, IUI '08, pages 139–148. ACM.

[75] Steve Vestal. A cursory overview and comparison of four architecture description languages.

[76] J. Whitehead. Collaboration in software engineering: A roadmap. In *Future of Software Engineering, 2007. FOSE '07*, pages 214–225.

[77] A Wise. Little-JIL 1.5 language report, lab. for advanced SW eng. research (LASER). dept. of comp. sci., UMass.

[78] Alexander Wise, AaronG. Cass, BarbaraStaudt Lerner, EricK. McCall, LeonJ. Osterweil, and Jr. Sutton, StanleyM. Using little-JIL to coordinate agents in software engineering. In Peri L. Tarr and Alexander L. Wolf, editors, *Engineering of Software*, pages 383–397. Springer Berlin Heidelberg.

[79] Alexander Wise, Barbara Staudt Lerner, Eric K McCall, Leon J Osterweil, and Stanley M Sutton Jr. Specifying coordination in processes using little-JIL. pages 99–71.

[80] Jeffrey Wong and Jason Hong. What do we mashup when we make mashups? In *Proceedings of the 4th International Workshop on End-user Software Engineering*, WEUSE '08, pages 35–39. ACM.

[81] Xiaotao Wu and V. Krishnaswamy. Widgetizing communication services. In *Communications (ICC), 2010 IEEE International Conference on*, pages 1–5.

[82] Nan Zang. Mashups on the web: end user programming opportunities and challenges. In *the proceedings of First Workshop on Evaluation and Usability of Programming Languages and Tools*, pages 25–29.

[83] Nan Zang, Mary Beth Rosson, and Vincent Nasser. Mashups: Who? what? why? In *CHI '08 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '08, pages 3171–3176. ACM.

APPENDIX A

Codelistings

A.1 hADL

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE xml>
3  <hADLmodel xmlns="http://at.ac.tuwien.dsg/hADL/hADLcore"
4    xmlns:exe="http://at.ac.tuwien.dsg/hADL/hADLexecutable"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5    xmlns:xlink="http://www.w3.org/1999/xlink"
6    xsi:schemaLocation="http://www.w3.org/2001/XMLSchema-instance
       hADLexecutable.xsd">
7    <name>hADL-shared artefact</name>
8    <description>Shared Artefact hADL example</description>
9    <extension>
10     <exe:Executables>
11       <exe:collabPlatform xlink:href="http://tuwien.ac.at"
12         id="example">
13         <exe:acceptableResourceDescriptor>
14           examplePlatform
15         </exe:acceptableResourceDescriptor>
16       </exe:collabPlatform>
17
18       <exe:surrogate id="participant_surrogate">
19         <exe:surrogateFQN>
20           at.ac.tuwien.hadl.ParticipantSurrogate
21         </exe:surrogateFQN>
22         <exe:collabPlatform>examplePlatform</exe:collabPlatform>
23       </exe:surrogate>
24
25       <exe:surrogate id="moderator_surrogate">
26         <exe:surrogateFQN>
27           at.ac.tuwien.hadl.ModeratorSurrogate
28         </exe:surrogateFQN>
29         <exe:collabPlatform>
```

```
30            examplePlatform
31          </exe:collabPlatform>
32        </exe:surrogate>
33
34        <exe:surrogate id="artefact_surrogate">
35          <exe:surrogateFQN>
36            at.ac.tuwien.hadl.DocumentSurrogate
37          </exe:surrogateFQN>
38          <exe:collabPlatform>
39            examplePlatform
40          </exe:collabPlatform>
41        </exe:surrogate>
42      </exe:Executables>
43    </extension>
44
45    <hADLstructure>
46      <component id="party" xsi:type="exe:tExecutableComponent">
47        <name>Participant</name>
48        <action id="party_edit">
49          <name>edit</name>
50        </action>
51        <exe:executableViaSurrogate>
52          participant_surrogate
53        </exe:executableViaSurrogate>
54      </component>
55      <component id="moderator" xsi:type="exe:tExecutableComponent">
56        <name>Moderator</name>
57        <action id="moderator_edit">
58          <name>edit</name>
59        </action>
60        <exe:executableViaSurrogate>
61          moderator_surrogate
62        </exe:executableViaSurrogate>
63      </component>
64      <object id="artefact" xsi:type="exe:tExecutableObject">
65        <name>Artefact</name>
66        <action id="artefact_edit">
67          <name>read</name>
68        </action>
69        <exe:executableViaSurrogate>
70          artefact_surrogate
71        </exe:executableViaSurrogate>
72      </object>
73      <link id="artefact_mod_edit">
74        <objActionEndpoint>
75          artefact_edit
76        </objActionEndpoint>
77        <collabActionEndpoint>
78          moderator_read
79        </collabActionEndpoint>
80      </link>
81      <link id="artefact_party_edit">
82        <objActionEndpoint>
```

```
83            artefact_edit
84         </objActionEndpoint>
85         <collabActionEndpoint>
86            party_read
87         </collabActionEndpoint>
88      </link>
89   </hADLstructure>
90 </hADLmodel>
```

Listing A.1: hADL model instance of a shared artefact collaboration pattern.

A.2 Surrogates

```java
package at.ac.tuwien.hadl.surrogates;

import at.ac.tuwien.hadl.domain.ResourceDescriptor;
import at.ac.tuwien.hadl.domain.Scope;
import at.ac.tuwien.hadl.domain.Surrogate;
import at.ac.tuwien.hadl.domain.SurrogateEvent;
import rx.Observable;
import rx.util.async.Async;

public abstract class AsyncSurrogate implements Surrogate {
    @Override
    public Observable<SurrogateEvent> acquire(Scope forScope,
            ResourceDescriptor surrogateFor) {
        return Async.start(() -> acquireSurrogate(forScope, surrogateFor));
    }

    @Override
    public Observable<SurrogateEvent> release(Scope inScope) {
        return Async.start(() -> releaseSurrogate(inScope));
    }

    @Override
    public Observable<SurrogateEvent> connectTo(Scope inScope,
            Surrogate endpoint) {
        return Async.start(() -> connectToSurrogate(inScope, endpoint));
    }

    @Override
    public Observable<SurrogateEvent> disconnectFrom(Scope inScope,
            Surrogate endpoint) {
        return Async.start(() -> disconnectFromSurrogate(inScope, endpoint));
    }

    public abstract SurrogateEvent acquireSurrogate(Scope forScope,
            ResourceDescriptor surrogateFor);

    public abstract SurrogateEvent releaseSurrogate(Scope inScope);

    public abstract SurrogateEvent connectToSurrogate(Scope inScope,
            Surrogate endpoint);

    public abstract SurrogateEvent disconnectFromSurrogate(Scope inScope,
            Surrogate endpoint);
}
```

Listing A.2: Base class for asynchronous surrogates.