FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

A Domain-Specific Language for Coordinating Collaboration

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Christoph Laaber
Matrikelnummer 1127107

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. April 2016

Christoph Laaber　　Schahram Dustdar

FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

A Domain-Specific Language for Coordinating Collaboration

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Christoph Laaber
Registration Number 1127107

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, 13th April, 2016

Christoph Laaber Schahram Dustdar

Erklärung zur Verfassung der Arbeit

Christoph Laaber
Marchetstraße 19, 2500 Baden

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.

Wien, 13. April 2016

Christoph Laaber

Danksagung

An erster Stelle möchte ich mich bei meinen Eltern, Karin und Manfred, bedanken. Ihr beide habt mich über all die Jahre in allen Lebensphasen, mit allen euch möglichen Ressourcen, ohne wenn und aber unterstützt. Worte können meine Dankbarkeit kaum beschreiben. Ich bin mir sicher, dass ich zu einem späteren Zeitpunkt, diese Unterstützung zurückgeben kann und werde.

Auf fachlicher Ebene, möchte ich mich besonders bei Christoph Mayr-Dorn für seine ununterbrochene, zeitlich unmittelbare und qualitativ hochwertige Unterstützung bedanken. Ohne ihn würde ich nicht hier, am Ende meiner Diplomarbeit, stehen. Dank gebührt auch meinem Betreuer Prof. Schahram Dustdar.

Besonderer Dank gilt meiner ganzen Familie, die, obwohl man sie sich nicht aussuchen kann, großartig ist. Daniel, du bist der Bruder, den ich nie hatte. Weiters ein herzliches Danke meinen Großeltern (Gerhild und Wolfgang, Maria und Josef), Anita und Fredi, Christa, Gertrude, Hildegard, Otto und Sabrina.

Franzl, Pauli, FPH oder wie auch immer ich dich gerade nennen mag: Mit dir gemeinsam in der Bibliothek zu sitzen, an meiner Diplomarbeit zu arbeiten, zu scherzen, Mädls zu betrachten, oder was auch immer, hat mich durch die Höhen und Tiefen der Monate, mit einem Lächeln auf den Lippen getragen. Danke!

Folgende Menschen, sind mir bis zum heutigen Tag besonders zur Seite gestanden. Auch bei ihnen, möchte ich mich für all die wertvollen Momente bedanken: Carlo, Dominik, Ello, Fabi, Gili, Joni, Karli, Martin, Matthias, Mischa, Raphi, Rono, Rudi, Simon, Wendi, Wolfi.

R.I.P. Oma, Opa und Daniel.

Kurzfassung

Seit einigen Jahrzehnten ist Software Design und Architektur im Fokus der Wissenschaft und der Industrie. Eine neuere Entwicklung in diesem Bereich ist die Einbeziehung von Menschen als integraler Bestandteil von Software Architektur. Eine solche Architekturbeschreibungssprache mit Fokus auf Menschen als Komponenten ist die Human Architecture Description Language (hADL). hADL gehört zu den struktur-orientierten Sprachen, die detaillierte, nicht restriktive Beschreibungen von Kollaborationen erlauben. Die Instanzierung von Kollaborationen ist jedoch komplex und brüchig. Prozess-orientierte Sprachen erlauben, im Vergleich zu struktur-orientierten Sprachen, bessere Unterstützung für Workflow-Modellierung und Ausführung von Kollaborationen. Der Nachteil von prozess-orientierten Sprachen ist die fehlende Flexibilität im Entwurf von Kollaborationen, welche struktur-orientierte Sprachen mit sich bringen. Eine Kombination dieser beider Paradigmen wird als überlegener Ansatz angesehen.

Diese Diplomarbeit untersucht eine Methode, um gültige hADL Programme spezifizieren zu können, die den Aufwand von Entwicklern reduziert. Eine weiteres Ziel ist, die beiden Architekturbeschreibungsparadigmen näher zusammen zu bringen.

Eine domänen-spezifische Sprache (DSL) wird mit Xtext, einem Framework zur Entwicklung von DSLs, entwickelt, welche es erlaubt Kollaborationsinstanzen in einer prägnanten Form zu spezifizieren. Der Funktionsumfang der DSL beinhaltet Verknüpfungs- (linkage) und Beobachtungsgrundfunktionalitäten (monitor) von hADL, Kontrollflussanweisungen, Variablen und einen Abstraktionsmechanismus. Mit Hilfe von automatischen Überprüfungen wird die Gültigkeit von DSL-Programmen zur Übersetzungszeit sichergestellt. Nur gültige Programme werden in Java/hADL-Client Programmcode übersetzt. Weiters wird der Entwicklungskomfort durch IDE Vorschläge und Auto-Vervollständigungen erhöht.

Die DSL wird anhand eines einfachem Scrum Prozesses evaluiert. Die Evaluierung zeigt auf, wie sehr sich der Aufwand eines Entwicklers reduziert, der die DSL statt Java zum Definieren von Kollaborationsinstanzen verwendet. Zwei Szenarien werden implementiert, die die notwendigen Kollaborationsstrukturen auf- und abbauen. Die Ergebnisse zeigen, dass ein signifikant geringerer geistiger (Überprüfungen im Kopf) und physischer (Tippen von Zeilen Code) Aufwand nötig ist, um Kollaborationsinstanzen mit der DSL zu definieren. Im Evaluierungsszenario führt die DSL 70 automatische Gültigkeits- und Konsistenzüberprüfungen durch und das DSL-Skript ist um einen Faktor 7,3 kürzer als der automatisch generierte hADL-Client Programmcode.

Abstract

Software design and architecture has been a focus of research and industry for a couple of decades. A more recent development in terms of software architecture is the inclusion of humans into software systems. A human-centered architecture description language that belongs to the structure-centric paradigm is the Human Architecture Description Language (hADL). Structure-centric languages and therefore hADL enable descriptions of detailed, non-restrictive collaboration mechanisms. Nevertheless instantiating collaborations described in hADL is brittle and verbose. Compared to structure-centric languages, process-centric languages offer better support for executing collaborative workflows, but do not have the flexibility for defining collaborations. A combination of those is considered being a superior solution compared to either paradigm on its own.

A method for specifying valid hADL programs is examined, that decreases the programmer's effort drastically. Furthermore the aim is narrowing the gap between process-centric and structure-centric languages.

This thesis introduces a Domain-Specific Language (DSL) developed with Xtext (a framework for language engineering), which allows developers to specify hADL collaboration instances in a concise way. The feature set of the DSL includes linkage and monitor primitives of hADL, control flow statements, variables and an abstraction mechanism. Validity and consistency of DSL programs is ensured by compile-time checks . Only valid programs are transformed into Java/hADL client code. Moreover development convenience is increased through IDE suggestions and auto-completions.

The DSL is evaluated with a simplified Scrum process. The evaluation highlights the effort a developer saves when using the DSL compared to defining collaborations with plain Java. Collaborations of two distinct scenarios of Scrum are implemented with the DSL. Required collaboration structures are set up and torn down later on. The results show that a significantly lower amount of mental work by the developer and Lines of Code (LOC) are necessary to define collaborations with the DSL compared to Java. The DSL performs 70 validity and consistency checks and the generated hADL client code is by a factor of 7.3 longer.

Contents

CHAPTER 1

Introduction

This thesis investigates how collaborations are defined through a DSL. This DSL is based on the hADL and enriches the traditional way of describing collaborations with Java and the hADL Application Programming Interface (API) by providing a significantly easier syntax, validity and consistency checks, Integrated Development Environment (IDE) features, little set up and an abstraction mechanism that is conveniently usable from Java.

1.1 Problem Statement

Since the beginning of software engineering, engineers and researchers put a lot of effort into to build and design software systems. The technical view on these systems dominated as opposed to a more human-centric view. Questions such as *what is the architecture of the software systems?* and *what are the components of the system and how are those connected?* were investigated in more detail than *how do humans use and interact with each other through the software system?*. In human-centric systems questions of the latter type are addressed. In human-centric architecture the design focuses on collaboration and coordination among human workers. Recent research deals with how to build software systems that integrate flexible behaviour of humans by means of software systems.

There are two general paradigms of how to integrate humans into software systems: behaviour-centric and structure-centric approaches [1]–[3]. Behaviour-centric approaches put the actual work humans carry out to the center of attention and deal with combination of their results. Examples are process-centric systems (see 2.1), Mashups which are often strongly connected to process-centric systems (see 2.2) and Crowdsourcing System (CS) (see 2.3). These approaches focus on *what is done?* where collaboration between humans is either defined by the system or not defined at all.

Structure-centric approaches on the other hand describe how collaboration and coordination is performed between human and/or software-based workers. An example is hADL (see 2.5). Those focus on *how is it done?*. Hence collaboration/coordination among humans is explicitly defined.

Either way of describing human architecture has advantages compared to the other. Hence an ideal solution would combine process-centric and structure-centric architecture description languages [1]. Dorn, Dustdar, and Osterweil [2] describe three strategies how a combination of them can be modeled. These strategies are either (i) task-driven, (ii) interaction-driven, or (iii) artifact-driven. Apart from specifying these strategies to combine the paradigms, no further research has been published to the best of our knowledge.

1.2 Motivating Scenario

Assume a standard agile software development process with Scrum at a medium sized software development company. Agile software development and therefore Scrum is an instance of a human-centric system. For example the Scrum master defines the collaborations of the Scrum process with the DSL. These collaborations support the developers in their process of creating software.

The workload of a specific project is divided into stories. Every week a new development iteration (sprint) starts where open stories are assigned to developers, who implement the stories until the end of the sprint. Multiple developers may be responsible for a single story and many stories may be assigned to a developer. For every story of a sprint a chat room is created and all responsible developers are invited to it, in order to discuss story related topics. At the end of a sprint all of the created chat rooms are deleted and the sprint concludes. After every sprint before the next starts, all developers that took part in the sprint are invited to the sprint retrospective meeting (a chat room). The purpose of this meeting is to discuss the quality of the scrum process and take note of what went well and how possible changes may improve it. These findings are documented on a wiki page.

In this scenario the software company uses a web-based collaboration tool for group communication such as HipChat. Sprint retrospective meeting notes are stored in form of wiki pages. As this process is about development of software it is apparent to use the wiki functionality of a source code repository hosting platform like Bitbucket. Management of the Scrum process is done through an online tool such as Agilefant. The important properties of these platforms are, that they have a web User Interface (UI) and an API. The UI for the humans to do the actual work and the API to retrieve needed information (sprints, stories and developers) and automatically set up the necessary communication artifacts (chat rooms, wikis).

Recall the above mentioned research deficiencies and the proposed strategies of handling flexible human behaviour in a process-oriented fashion. The interaction-driven strategy

[2] fits this scenario best as scrum is by design process-centered. Hence we start off by defining the process steps in a process-oriented language choice (e.g. Business Process Model and Notation (BPMN) or Little-JIL). Those steps are (i) starting the sprint and creating the chat rooms for each story, (ii) wait for completion of the sprint, (iii) tear down the chat rooms and finish the sprint, and (iv) hold sprint retrospective meeting. For simplicity reasons steps such as story selection and assignment (sprint planning meeting), daily scrum meetings and the sprint review meeting are excluded from the example process.

While the process steps are defined, there is no explicit notion of how collaboration is performed within and across those steps. Therefore a structure-centric language with flexible collaboration descriptions is used. In the next step the collaborations of each step are defined with the structure-centric language of choice (e.g. hADL) and executed. If hADL is used the collaboration structure is defined as a hADL model. Operating on hADL structures is done through the hADL runtime framework, which is quite verbose, can be error-prone and process developers have to call the appropriate hADL API calls.

Those problems are addressed in this work by introducing a DSL, which is a concise way of specifying valid hADL programs that are easily callable from Java code. Therefore the process developer can focus just on the process itself and has a defined entry point (a method call) to hADL code.

1.3 Contributions

This thesis defines a DSL for specifying valid collaboration instances with hADL. Furthermore it aims at narrowing the gap between process-centric languages (e.g. BPMN and Little-JIL) and hADL in a way that hADL programs can easily be defined and used from regular Java code. Furthermore reducing the developer's mental and physical effort is a goal of this thesis. These contributions are addressed by the following research questions:

RQ 1 *How can we define valid hADL programs and what benefits does a programmer gain from such a program? How is validity of a hADL program achieved?*

RQ 2 *How can collaborations in hADL be abstracted so that they are usable from process-oriented languages with little set up costs?*

Methodology In order to achieve the contributions and provide answers to the research questions asked, this thesis takes the following methodology: First we look at different scenarios and use cases for this work's potential DSL and deduct requirements from it and set the scope. The initial scenarios serve as input for the evaluation scenario. The requirements and the scope define the feature set of the DSL. An important property of the DSL is that the feature set is concise and fully functional as opposed to broad and only operative under special circumstances. Based on the feature set the DSL is designed and implemented. The implementation follows the design by adding validity and consistency checks. This DSL is then evaluated with the help of a use case scenario. It is measured how much benefit the DSL brings compared to using Java with the hADL API and the results are presented and discussed accordingly.

1.4 Thesis Organization

The remainder of this thesis is structured as follows.

Chapter 2 provides the background and related work. hADL is the basis on which this thesis and its DSL build. The concepts that are important for the rest of this thesis are described in detail (see 2.5). Furthermore this chapter introduces and compares hADL and the DSL with approaches in the areas of (i) human involvement processes (see 2.1), (ii) mashups (see 2.2), (iii) crowdsourcing (see 2.3) and (iv) software engineering and collaboration/coordination support.

Chapter 3 is about the design of the DSL. It introduces the approach to designing the DSL and the benefits we desire to achieve with it. Next the language is specified in terms of syntax and semantics. The necessary constraints are discussed which lead to a valid hADL program. Finally error handling in the DSL is described.

Chapter 4 describes the implementation of the language designed in chapter 3. It starts with an introduction to Xtext/Xtend and how it is used to achieve the design decisions. The other implementation section introduces the hADL synchronous library a wrapper for the hADL runtime framework. This library provides the functionality of the hADL runtime framework in a way that suits the DSL more.

Chapter 5 introduces a use case scenario and how it is implemented with the DSL. This scenario is then used to evaluate the DSL. It shows the effort a developer saves when using the DSL compared to Java. This chapter wraps up by describing assumptions that were made and how malicious users can interfere while writing or executing a DSL script.

Finally this thesis concludes in Chapter 6 by summarising the achievements and give possible directions for future work.

CHAPTER 2

Related Work and Background

This chapter introduces related work in the area of human collaboration/coordination systems and compare it to hADL. Sections 2.1 to 2.4 deal with research work and industry solutions. Furthermore it provides background information which serves as the basis of this thesis' work. hADL, described in 2.5, is the underlying language and runtime framework which the DSL is targeting.

2.1 Human Involvement Processes

This first related work section deals with process-centric approaches. In the following Subject-oriented Business Process Management (S-BPM), Social Business Process Management (BPM), Little-JIL and Web Services Human Task (WS-HumanTask)/BPEL4People are investigated and compared to hADL/this thesis' approach.

A business process is a set of interrelated tasks/steps handled by workers (human, computational or mixed) in a logical (with respect to the business) and chronological order. In each task a worker uses resources (information) to reach the goal of the business process and therefore satisfy the customer. Additionally every business process has a defined start state, input data and an end state with a result [4].

Subject-oriented Business Process Management - S-BPM

BPM is important for the success of organizations. With BPM organizational strategies and business models are implemented in a process-centric way. Opposed to BPM S-BPM brings the subjects (actors) who carry out the actual work to the center of attention. In S-BPM there are four roles: (i) Governors act as bridges between executives and operational business. They take responsibility and assure the quality of the process; (ii) Actors do the actual work in business processes. They are supported by Experts and Facilitators; (iii) Experts are specialist in a particular field and are triggered by any other role; and (iv) Facilitators support Actors in organizational development to satisfy stakeholder's needs [4].

In a first step the subjects of business processes are identified and described with a collection of Subject Behaviour Diagrams (SBDs). Next the communication among those need to be represented (subject-to-subject communication). Communication in S-BPM is done through messages which optionally contain business objects (structured information). As a result we get a Subject Interaction Diagram (SID) or alternatively called Communication Structure Diagram (CSD). Messages between subjects are either asynchronously or synchronously exchanged, hence the subjects do their work autonomously and in parallel, whereas traditional BPMs assumes a central control flow [4], [5]. Fleischmann, Schmidt, Stary, *et al.* [4] furthermore describe how to model S-BPM with subprocesses and other complex process networks, subject and their behaviour's description including exception handling, validation of processes and models and optimization aspects.

As business processes are mostly described in natural language [4] by non-technical personnel it takes another level of indirection to bring those informal definitions into correct S-BPM models. In [6] an approach based on physical card boards is introduced to create S-BPM models. This method generates semantically incomplete models such that simulation and refinement of the model is applied as last step. This yields semantically and syntactically correct S-BPM models.

In [4] it is described how to implement S-BPM processes. Raß, Kotremba, and Singer [7] show how a S-BPM process can be executed using Windows Workflow Foundation (WF).

In traditional S-BPM subjects are always represented by a single entity (human, computational or mixed). Fleischmann, Kannengiesser, Schmidt, *et al.* [5] propose a system that unifies multi agent-systems with the Agent/Group/Role (AGR) model and S-BPM. The benefit is that the same business process can be implemented using different organizational structures. Reversely agents or agent structures can implement different processes. Hence we gain a more variable and flexible system where reusability of processes and agents is possible.

Krauthausen and Krauthausen [8] take a different approach to subject-orientation in business processes than S-BPM. Relying on baskets that act as a collection of business objects (input/result of work) with publish and subscribe semantics on them for either business tasks or step tasks. Business objects are routed to business tasks through the Subject Communication Engine. A business task is always associated with a human worker and can be divided into sub tasks and action items that are carried out by a worker. The result of an action item is then published to a basket by the worker. On the other hand business objects are assigned to step tasks by the Subject Task Execution Engine. A step task is a software automated function consuming a business object and publishing the result to a basket.

S-BPM is conceptually based on process algebras like Calculus of Communicating Systems (CCS) and Communicating Sequential Processes (CSP) where communication between subjects has message-based semantics [4]. Krauthausen and Krauthausen's

[8] approach uses publish/subscribe-based communication. Compared to those two hADL does not have any restrictions on how communication between humans is realised. The way of communication is only dependent on the implementation of the respective CollaborationObject.

Social BPM

Social BPM combines BPM with social software such as Wikipedia or social networks, changing the traditional approach from closed to an open/social one. This fusion can be applied to design and/or enactment phase.

Dengler, Koschmider, Oberweis, *et al.* [9] propose a system where Wikis are used to textually describe business processes and keep them in sync with the process model and find/coordinate collaborators on social networks. Wikis in this approach are based on Semantic MediaWiki (SMW) which uses semantic annotations for connecting Wikis. Each Wiki represents an activity in the process. In an export step Wikis are transformed with the Resource Description Framework (RDF) into Petri Nets which are then executed by a process execution engine. Coordination and communication in social networks are handled by a model called Community Process. In Community Process there are three different stages: finding partners, building relationships and executing collaboration. Each Community Process has a set of Community Process Objects (Users and Data) which are transferred between activities. An extension based on case-based reasoning integrates Community Process with services of business information systems. Changes to the process model are exported back to the Wiki. Therefore the Wiki and the process model are kept in sync.

In [10] [11] BPMN is extended to support social interactions through patterns. The following social tasks are defined by this extension: social broadcast, social posting, invitation to activity, commenting, voting, login to join, invitation to join a network and search for actors's information. These tasks are performed by either a single or multiple users. Furthermore social design patterns are defined by the extension which can be assembled to form social processes. The introduced solution is implemented in WebRatio that transforms BPMN models via WebML into JavaEE applications.

PROWIT [12] implements a user-centric process system based on Liferay portals. All collaborations are done through a single platform. A BPM model is loaded into the platform, user tasks are created and assigned to the specified roles. Users of the platform get tasks for their roles presented in a portal.

All social BPM solutions presented provide means for collaboration between humans through a social platform. The differences to hADL are that collaborations are only possible within a single task [10] [11] and/or collaboration is used for designing a process [9]. PROWIT [12] is a social platform for executing processes, whereas hADL does not make any assumption on which tools are used for communication.

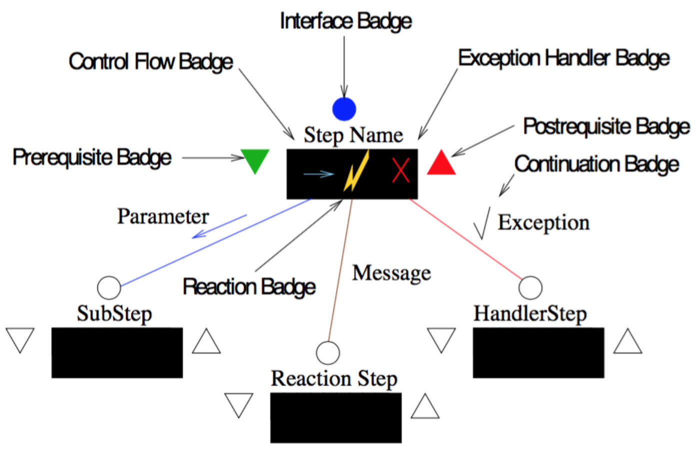


Figure 2.1: Little-JIL Step [14]

Little-JIL

Little-JIL is a graphical process-centric language for defining collaborations. The main abstraction is a step, which is similar to a task in BPMN. A Little-JIL program is a tree of steps where the leaf steps represent the smallest work loads and the structure of the tree indicates how coordination is done. Each step has exactly one agent associated with it. An agent can be either human or computational. A step can be in many states of which the most important are: posted to an agent, started by agent, successfully completed or terminated with exception. Figure 2.1 shows a graphical representation of a step with all possible badges and connections to other steps. Steps always have at most one parent step and 0 or more sub-steps. All non-leaf steps can have one of the following control flow kinds: sequential, parallel, try, choice. The control flow defines how sub-steps are executed. Parameters are passed from the parent-step to the sub-steps. Additionally each step can handle exceptions through a handler-step. Through Requisites checks are added before (Prerequisite) and after (Postrequisite) a step's execution. Each step uses a set of resources necessary to execute it. Resources are acquired during runtime adding dynamic behaviour to Little-JIL programs [13] [14].

Communication between humans (agents) is always through parameter passing of steps. Compared to hADL Little-JIL does not allow specification of how humans interact with each other. The DSL has the notion of tasks (see 3) that can be executed by a step in Little-JIL.

WS-HumanTask and BPEL4People

Business Process Execution Language (BPEL) - more accurately Web Services (WS)-BPEL - defines business processes focussing on interaction between web services. Standard BPEL does not define how humans are incorporated into business processes. Therefore an extension to BPEL has been introduced named WS-BPEL4People [15]. BPEL4People utilizes WS-HumanTask specification [16] as well as other WS-* specifications.

WS-HumanTask [16] describes humans in a service-oriented fashion through Web Services Description Language (WSDL) interfaces. Hence integrating "human services" into service-oriented applications is possible. WS-HumanTask defines two types of interfaces: (i) an interface that provides the service described by the task and (ii) another interface that lets people handle the tasks. Each human task has a set of people assigned to it. Sequential and parallel assignment of tasks is possible. Notifications are a special form of Human Tasks that send information concerning the business process to people. Furthermore WS-HumanTask defines decomposition of tasks into subtasks (sequential or parallel), Lean Tasks which have reduced capabilities and input/output data type descriptions. Gerhards, Skorupa, Sander, *et al.* [17] propose a security framework that adds authentication and authorization capabilities to WS-HumanTask processors.

BPEL4People [15] builds on top of WS-HumanTask. It adds People Activities to BPEL that are either inline Human Tasks or standalone Human Tasks (as described by WS-HumanTask [16]).

WS-HumanTask and BPEL4People communication and coordination between humans are on-task basis. Whereas hADL and specifically the proposed notion of combining it with a process-centric language (see 2.5) does not have this limitation. Through this work's DSL collaboration and coordination within a process task can be described with hADL's capabilities.

2.2 Mashups

Mashups in general are systems that incorporate a number of web services, merge, transform, filter and/or augment the data they have received and provide that data in a new way (possibly through a new web service). Traditionally mashups are tightly coupled with BPEL. Daniel, Koschmider, Nestler, *et al.* [18] describe mashups with respect to three dimensions: multi-user support, multi-page navigation and workflow support. From these dimensions the following types of mashups are identified: simple mashups, multi page mashups, guided mashups, page flow mashups, shared page mashups, shared space mashups, cooperative mashups and process mashups. Only mashups that support multiple users, have a multi-page navigation and support workflows are considered process mashups.

Torres, Pérez, Koschmider, *et al.* [19] do not use a graphical way to create mashups, but extend BPMN to support coordination between actors (humans) that work on a shared task. As this approach uses BPMN it highly relates to the previous section (see 2.1). Collaborative tasks are introduced where multiple actors with potentially different roles work on a single task together. Therefore three kinds of models are used (as shown in Figure 2.2): (i) a Business Process Model (BPMN); (ii) a Roles Model that describes which roles participate in collaborative task, which are optional or mandatory and which

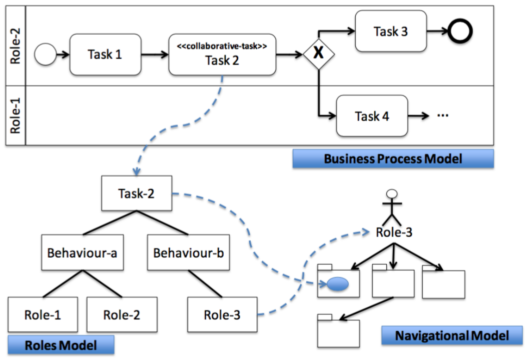


Figure 2.2: Collaborative Tasks in Process Mashups [19]

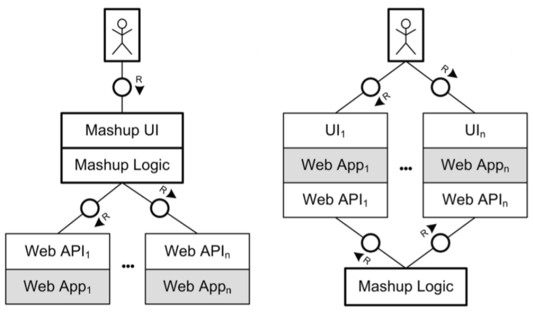


Figure 2.3: Architectural Mashup Styles: (left) Standard, (right) Reverse [20]

is task responsible; and (iii) a Navigational Model that describes the UIs, which are necessary for the roles to participate in the process.

Kunze, Overdick, Grosskopf, *et al.* [20] propose a reversed architecture of mashups compared to traditional mashups. In traditional mashups, as seen on the left of Figure 2.3, content from different web services are aggregated into a single page or a collection of pages. Whereas the reversed architecture, as seen on the right in Figure 2.3, uses the UIs of the web services for user interaction. The mashup logic accesses the APIs of those web services to run the process. The business process is also specified with BPMN, but compared to [19] no collaboration tasks are available. Hence only collaboration on a mutual overall task, but with distinct subtasks is possible. Communication between humans, signals to start a task and to end a task are sent through public Twitter messages. Such a message always includes the recipient (through Twitter's mentioning feature), the Uniform Resource Locator (URL) of the task and a command (e.g. start, stop, decide, ...).

Process mashups compared to hADL provide UIs through which actors (humans) work on the specified tasks, whereas hADL does not provide any UIs. A collaboration defined with hADL uses the respective collaboration platforms and their UIs for user input. Furthermore process mashups integrate process-centric language capabilities. This work relies on external process-centric languages to invoke the DSL's generated Java code. Torres, Pérez, Koschmider, *et al.* 's [19] approach does not specify how roles of a collaborative task communicate (e.g. Shared Artifact, Message, Stream). The second technique [20] is similar to hADL in the sense, that existing websites/collaboration platforms are utilized for user interaction. Nevertheless it does not provide means to define how collaboration is accomplished. Communication between humans as well as process coordination is done through Twitter. hADL has no constraint on that front.

2.3 Crowdsourcing

Crowdsourcing is a form of human computation where a task is distributed to a group of people. It is a way of distributed problem solving where humans play the integral part. Furthermore crowdsourcing is a valuable production model for businesses. Typical CS are (i) the ones which provide the functionality to build CSs (e.g. Amazon Mechanical Turk (MTurk)) or (ii) the actual systems where particular tasks are solved through human workers (e.g. Linux, Wikipedia, Yahoo! Answers, Threadless) [21].

Doan, Ramakrishnan, and Halevy [22] describe CSs by identifying genes that form genomes. A Gene is the basic building block which describes a task of a CS by answering four questions: (i) Who is performing the task?; (ii) Why are they doing it?; (iii) What is being accomplished?, and (iv) How is it being done?

Malone, Laubacher, and Dellarocas [23] classify CSs by looking at 9 dimensions: (i) Nature of collaboration; (ii) Type of target problem; (iii) How to recruit and retain users?; (iv) What can users do?; (v) How to combine user input?; (vi) How to evaluate users?; (vii) the degree of manual effort for a user; (viii) the role of the user; and (ix) if the system piggy-backs on another system or if it is standalone.

Jabberwocky [24] is a framework that treats machines and humans as first class citizens and enables building workflows with those. It is built consisting of three layers. The bottom layer - Dormouse - is the virtual machine that provides a unified view and interface to machines and humans. Dormouse interacts with existing crowdsourcing platforms and abstracts away the details of specific APIs. The middle layer is a map-reduce inspired framework called ManReduce. It provides ways to decompose complex tasks into micro-tasks by specifying map steps and aggregation of micro-tasks with reduce steps. Both, map and reduce steps, can be either machine or human. The top layer consists of the scripting language Dog, which provides a high abstraction for requesting work from either machines or humans as well as creating, using and sharing functions and micro-task templates. Programmers can either choose ManReduce or Dog to implement CSs. ManReduce scripts are written in a Ruby-like programming language, whereas Dog uses a Structured Query Language (SQL)-inspired language where collaboration patterns (e.g. Vote, Label, Compare, Extract, Answer) are used to operate over a specified data set. Examples for ManReduce and Dog scripts can be seen in Figure 2.4.

CrowdForge [25] is a framework for decomposing complex work into interconnected micro-tasks in map-reduce style (similar to Jabberwocky). It makes use of the existing micro-task crowdsourcing platform MTurk. The basic building blocks are partition, map and reduce steps. Partition divides a task into smaller sub tasks. Map is the phase where the tasks are solved by at least one human worker. In the Reduce phase results from previous tasks are combined into a single result. Complex flows with an arbitrary sequence of partition/map/reduce steps are possible. Compared to Jabberwocky, CrowdForge workflows are designed using a web UI. CrowdForge only allows human workers to solve tasks.

```
map :name => :extract_disease_facts do |key,
    value|
  facts = RiskExtractor.extract (value)

  for fact in facts do
    emit (fact["disease"], fact["risk_factor"
        ])
  end

end

reduce :name => :summarize do |key, values|

  task = SummarizeFacts.prepare
    :task_name => "Summarize disease risks:
        #{key}"
  task.facts = values

  task.ask do |answer|
    emit (key, answer)
  end

end
```

(a) ManReduce Extraction Script [24]

```
students = PEOPLE FROM gates
reviews = ASK students TO Review ON
    uist_submissions SUCH THAT uist_submission.
    topic IS IN student.areas_of_expertise
reviewers = FIND PEOPLE FROM reviews
advisors = PEOPLE FROM gates WHERE advisees
    CONTAINS reviewers
validated_reviews = ASK advisors TO Validate ON
    reviews SUCH THAT review.reviewer = advisor.
    student
```

(b) Dog Script [24]

Figure 2.4: Jabberwocky Script Examples

Turkomatic [26] is another tool for handling complex tasks by splitting them up into micro-tasks (map-reduce style), which are then solved with help of MTurk. The novel feature is that the design of the workflow itself is done by the crowd. The requester enters a task description (e.g. a question) into a web input field. The submitted task is posted to MTurk where a worker has to decide either (i) is the task easy enough to solve it directly, or (ii) is it too complex. If the task is too complex, the worker subdivides the task into smaller subtasks. This algorithm - named Price-Divide-Solve algorithm - is then repeatedly applied to every subtask until every task is easy enough to solve directly and therefore a result for the overall task can be obtained. Through the whole process of collaboratively defining and solving the task, the requester has capabilities to monitor the current status of and edit the workflow in real time. If a task was edited, all downstream subtasks are removed and potential upstream task solutions are invalidated.

CrowdLang [27] is an executable, model-based programming language and framework for handling complex crowdsourcing applications, which handles interdependencies between steps. It supports task decomposition, identifies coordination mechanisms, integrates several dimensions of design attributes and supports recombination of existing patterns. Both human and software-based computation of tasks are supported. CrowdLang provides a set of basic operators and elements for task decomposition, routing and distribution (Divide-and-Conquer, Aggregate, Multiply, Merge, Router and Reduce). Furthermore the following elements of collective intelligence are provided: Job, Contest, Iterative Collaboration, Parallelized Interdependent Subproblem Solving, Independent Decision and Group Decision. CS are built with those provided operators and elements. User-defined complex elements are supported through combination of basic operators, elements and existing complex elements.

In “A human-centric runtime framework for mixed service-oriented systems” Schall [28] takes a service-oriented approach to combine human computation - called human provided services - and machine computation - called software-based services. The presented system consists of the following components: a Middleware Layer for managing data collections and eXtensible Markup Language (XML) artifacts, an API Layer exposing services for user form generation and XML Schema Definition (XSD) transformation, and a Runtime Layer for activity, user management and interactions through WS technology. Communication between services and therefore humans is done in a service-oriented fashion through SOAP messages and WSDL contracts. Additionally an expertise ranking is used to differentiate between the skills and interests of workers. It brings the concepts of previously discussed WS-HumanTask (see 2.1) to CS.

CS by its design coordinate humans to solve independent tasks and therefore direct communication/collaboration between humans is not desired. However some of the above described solutions provide some collaboration/coordination capabilities. CrowdLang [27] has the operator GroupDecision which implies collaboration and coordination between humans, but does not specify how it is achieved. Schall’s [28] method provides coordination within a task through a ControlInstance. The paper does not go into detail how the ControlInstance is created and used. Communication is always through one-to-one messages implied by the service-oriented architecture. Turkomatic’s [26] requester has coordination duties, but no communication between human workers is possible. Jabberwocky [24] and CrowdForge [25] do not provide any means of collaboration/coordination between humans. In Jabberwocky, CrowdForge and Turkomatic indirect communication between humans is possible through the input/output of tasks.

2.4 Software Engineering and Collaboration/Coordination Support

This thesis’ DSL is designed for all kinds of collaborations defined with hADL, but is evaluated with a Software Engineering (SE) scenario (Scrum). Therefore this related work section on research about collaboration and SE is of interest.

Whitehead [29] gives an overview of collaboration tools in SE up to the year 2007. According to him collaboration in SE can be divided into four main categories: (i) model-based collaboration tools spanning all stages of the SE life cycle. The most prominent candidate in this category might be the Unified Modeling Language (UML); (ii) process-centered collaborations; (iii) collaboration aware systems which let developers know which task, issue, branch, etc. is currently worked on by other developers; and (iv) collaboration infrastructure that helps developers integrate the collaborative workflows into their tools (e.g. IDE plug-ins for git, a bug tracker or a Continuous Integration (CI)-platform).

LaToza, Towne, Adriano, *et al.* [30] use crowdsourcing to build software collaboratively, which lowers the entry barrier of traditional open-source software projects by removing the need to acquire knowledge on how to contribute. In terms of dynamics the subdivision of tasks into subtasks is similar to Turkomatic (see 2.3) as the number of subtasks is not known at time of requesting the task. The central part are artifacts (functions and function tests) for which micro-tasks are created. These micro-tasks are created whenever an event on a particular artifact occurs. Every micro-task is a short amount of work that is done by a worker on a single artifact. Completing a micro-task triggers new events, and therefore creates new micro-tasks, on the respective artifact and on the ones that depend on it. The authors implemented their approach with the online IDE CrowdCode. With CrowdCode it is possible and desired to write parts of the function's body in pseudocode, which in turn creates new micro-tasks to implement those parts. Calling another function from a function can only be done by a pseudocall. This prevents workers from writing a whole program within a single micro-task and enforces development parallelism. Communication between workers is possible through a group chat with all currently logged-in workers.

Dorn and Egyed [31] propose an approach that deals with inconsistencies between design/architectural artifacts and source code artifacts by monitoring and analysing artifacts and communication between developers. Three types of analysis are done: (i) temporal, (ii) collaboration pattern and (iii) artifact dependencies. Based on that analysis, recommendations for adaption of artifacts are sent to developers.

This work's evaluation scenario combines the aforementioned categories of model-based collaboration (hADL) and process-centered collaboration by conveniently describing model-based collaborations with the DSL and generating Java source code that is then called from a step/task in a process. The method in [30] compares to hADL the same way as regular CS do (see 2.3). Communication between workers is only possible in a group chat, whereas hADL does not restrict communication to a particular type.

2.5 Human Architecture Description Language - hADL

The Human Architecture Description Language is the basis on top of which this work builds. Introduced by Dorn and Taylor [32] in their work *Architecture-driven modeling of adaptive collaboration structures in large-scale social web applications* hADL is a language to specify flexible human behaviour in a structure-centered way [1].

hADL consists of two parts: the language and the runtime framework. The language or schema defines the elements to describe collaborations and coordinations. The runtime framework facilitates instantiation and connection of language elements as well as life cycle handling. A runtime element has a corresponding entity on a particular collaboration platform. Such platforms are for example Bitbucket, HipChat or Agilefant. The elements and concepts introduced in the following are the ones integral to this thesis. The list is non-exhaustive. Some of the information below can be found in [3]. The rest is based on the source code of Dorn's git repository on Bitbucket [33].

To get a better understanding of the elements and concepts they are described with help

hADL element	XML tag	Java type
HumanComponent	`<component>`	`THumanComponent`
CollaborationConnector	`<connector>`	`TCollabConnector`
CollaborationObject	`<object>`	`TCollabObject`
Action	`<action>`	`TAction`
Link	`<link>`	`TCollabLink`
CollaboratorReference	`<collabRef>`	`TCollabRef`
ObjectReference	`<objectRef>`	`TObjectRef`

Table 2.1: hADL-Elements and Corresponding XML Tags and Java Types

of an exemplary chat room use-case. If possible a reference to the corresponding Java class of the hADL project is provided.

Language Elements

The core language of hADL defines collaborators (HumanComponents and CollaborationConnectors), interactions in form of messages, streams or shared artifacts (CollaborationObjects) and connections (Links and References) between those. The human architecture described through the language elements forms a collaboration pattern or a combination of patterns [32]. Such patterns are identified and described in [34], [35]. hADL uses an XML representation to represent a human architecture (see Appendix Scrum hADL Model). HumanComponents, CollaborationConnectors and CollaborationObjects are called hADL-Elements. Links and References are called hADL-Connectives.

Table 2.1 shows which language element corresponds to which XML tag and Java type. The tags and types are specified in the schema project [33]. All Java types are classes in the package `at.ac.tuwien.dsg.hadl.schema.core.*`.

HumanComponents are the building blocks that carry out specific tasks, which are essential to the collaboration. They act as the business logic and provide data for the collaboration. HumanComponents can be seen as the "decision makers" in a collaboration. In a chat room regular chat users are considered HumanComponents.

CollaborationConnectors are secondary or non-essential, replaceable entities that have coordination duties of the collaboration. Efficient and effective interaction between HumanComponents is the main goal of CollaborationConnectors. They can be anything from purely human to fully automated software scripts. A moderator of a chat room would be an instance of a CollaborationConnector.

CollaborationObjects define the form of interaction among HumanComponents/-CollaborationConnectors. Typical examples of interaction forms are Messages, Streams or shared Artifacts, all of which are instances of a CollaborationObject. A description of what Messages, Streams and shared Artifacts are and what differentiates those from each other can be found in [32]. The chat room by itself can be considered as a CollaborationObject of type Stream.

Actions are specified rights on HumanComponents, CollaborationConnectors and CollaborationObjects. There are two kinds of Actions: HumanActions and ObjectActions. HumanActions are access rights that Collaborators require in order to fulfill their roles. ObjectActions are rights that CollaborationConnectors expose for Collaborators to use. Actions can be seen as connection points. HumanActions are connected to ObjectActions through Links, which are described below. Multiple connections to a single Action are possible. A HumanAction on a chat user could be *ChatroomInvite* and an ObjectAction on the chat room could be *Invite*.

Links connect Collaborators with CollaborationObjects or more precisely HumanActions with ObjectActions. A link that connects the HumanAction *PostChatroomMessage* with the ObjectAction *Post* could be named *postToChatroom*.

References are relations such as inheritance, composition or containment between either Collaborators or CollaborationObjects. Relations between Collaborators are called CollaboratorReferences. Relations between CollaborationObjects are called ObjectReferences. An administrator HumanComponent could have an inheritance relation with a regular chat user HumanComponent.

Runtime Elements and Concepts

The previous section describes the Language Elements of hADL. This section introduces the elements and concepts that are part of the hADL-Runtime. A hADL-Client makes use of the language elements defined by the hADL-model in XML form (see Appendix Scrum hADL Model), and instantiates, connects, loads and releases instances of those.

LinkageConnector The LinkageConnector provides mechanisms to acquire, link, reference, unlink, dereference and release hADL language elements. Furthermore it starts and stops the scope of Operationals, which are described below.
Java class: `at.ac.tuwien.dsg.hadl.framework.runtime.impl.`
`HADLLinkageConnector`

RuntimeMonitor The RuntimeMonitor is responsible for loading (sensing) Operationals from existing Operationals via Links or References. This is done by Sensors, which are described below.
Java class: `at.ac.tuwien.dsg.hadl.framework.runtime.impl.`
`HADLruntimeMonitor`

hADL-Element	hADL-Operational-Element	Java type
HumanComponent	OperationalComponent	`TOperationalComponent`
CollaborationConnector	OperationalConnector	`TOperationalConnector`
CollaborationObject	OperationalObject	`TOperationalObject`
Action	OperationalAction	`TOperationalAction`
Link	OperationalLink	`TOperationalCollabLink`
CollaboratorReference	OperationalCollabRef	`TOperationalCollabRef`
ObjectReference	OperationalObjectRef	`TOperationalObjectRef`

Table 2.2: hADL-Elements and Corresponding hADL-Operational-Elements and Java Types

Operationals are the runtime equivalent of the language elements. Hence for every language element an Operational exists. A hADL-Client can retrieve Operationals either through the LinkageConnector or the RuntimeMonitor. OperationalCollaborators (OperationalComponents and OperationalConnectors) and OperationalObjects are either acquired (LinkageConnector) or loaded (RuntimeMonitor) from existing Operationals. Operationals for Links (OperationalLink) are returned by the LinkageConnector when an OperationalCollaborator is linked with an OperationalObject. Operationals for References are returned when either OperationalCollaborators (OperationalCollabRef) or OperationalObjects (OperationalObjectRef) are referenced.

Operationals of Collaborators and CollaborationObjects always have a corresponding Resource Descriptor (described below). All Operationals are in a particular state at any time and keep a reference to a Surrogate (described below). The different states are described in [3].

Table 2.2 shows the hADL-Elements with their corresponding hADL-Operational-Element and Java types. All Java types are in the package `at.ac.tuwien.dsg.schema.runtime.*`.

ResourceDescriptors encapsulate information that is necessary for Surrogates and Sensors to relate to corresponding elements on a collaboration platform. A Resource Descriptor needs to be supplied to the LinkageConnector when acquiring a hADL-Element. A ResourceDescriptor can be any class implementing `at.ac.tuwien.dsg.hadl.schema.runtime.TResourceDescriptor`.

Surrogates are proxy objects that are triggered by the hADL runtime framework upon acquire, link, reference, unlink, dereference and release as well as starting and stopping the Surrogate scope through the LinkageConnector. Their task is to map changes of the hADL runtime model to changes on the respective collaboration platforms. For example a Surrogate for a chat room calls the API of the collaboration platform (e.g. HipChat) when a chat user is invited to a chat room. An invitation can be modeled as a link between HumanAction *ChatroomInvite* and ObjectAction *Invite*.

A Surrogate is at any time in one of the following states described in [3]. Starting and stopping the Surrogate Scope (LinkageConnector) transitions the states of all Surrogates. This is done after an acquired element is changed (link, unlink, reference, dereference). Changes are applied to the collaboration platform on starting the Surrogate Scope.

Surrogates are classes that implement either `at.ac.tuwien.dsg.hadl.framework.runtime.intf.IObjectSurrogate` or `at.ac.tuwien.dsg.hadl.framework.runtime.intf.ICollaboratorSurrogate`.
The framework needs to know about an instance of `at.ac.tuwien.dsg.hadl.framework.runtime.intf.SurrogateFactory` which can be either specified within the hADL XML or provided programmatically (via Dependency Injection (DI)).

Sensors are used in combination with the previous introduced RuntimeMonitor to load Operationals from an existing Operational. Therefor the implementation of a sensor accesses the collaboration platform to get the connected elements, creates respective Resource Descriptors and emits a LoadEvent. The RuntimeMonitor intercepts emission and acquires the Operationals for the LoadEvent. With SensingScopes the hADL-Client can limit the Links, References or Actions that are being considered for loads.

Sensors are classes that implement either `at.ac.tuwien.dsg.hadl.framework.runtime.events.ICollaboratorSensor` or `at.ac.tuwien.dsg.hadl.framework.runtime.events.ICollabObjectSensor`.
Sensors are created by the framework with a registered instance (via DI) of `at.ac.tuwien.dsg.hadl.framework.runtime.events.SensorFactory`.

Process Support

hADL itself supports a model-driven approach to collaborations, but lacks the possibility to run collaborations in a process-centric fashion. In [1], [2], [35] a concept is introduced to combine process-centric languages such as Little-JIL or BPMN with structure-centric collaboration languages like hADL. This thesis' work narrows the gap between those two by introducing a DSL that creates functions containing sequential hADL statements to be called by tasks/steps of a process. The process of design (see 3), implementation (see 4) and evaluation (see 5) of the DSL is the main part of this thesis.

CHAPTER 3

Design

This chapter focusses on the design and specification of a DSL for hADL collaboration instance handling. First it shows in Section 3.1 the contributions this thesis makes to the existing hADL framework. Second the main aspects for using the DSL over Java/hADL API are described in Section 3.2. Third the language is described with its syntax and semantic in Section 3.3. Forth in Section 3.4 it gives insight into what and how validity and consistency is checked by the DSL. Finally it provides information about errors and how those are handled in Section 3.5.

3.1 Contributions to hADL

Section 2.5 introduces hADL's elements and concepts which are essential to this thesis. It is essential to figure out in which way this DSL should augment the existing hADL features. The focus of the DSL is on handling the LinkageConnector's, the RuntimeMonitor's responsibilities and aspects for integrating hADL with a process-centric language (e.g. Little-JIL or BPMN). The following list describes the contributions this thesis makes through its DSL to the current state of hADL as seen in Figure 3.1.

- hADL clients are currently implemented through the Java API exposed by the hADL runtime framework. A DSL script is translated into that Java hADL client.
- hADL primitives as exposed by the hADL LinkageConnector and the hADL RuntimeMonitor are represented by single line statements of the DSL syntax.
- The previous mentioned primitive statements are wrapped in a synchronous library for easier handling from the DSL generated hADL client code. This library can be used without the DSL from Java as well.

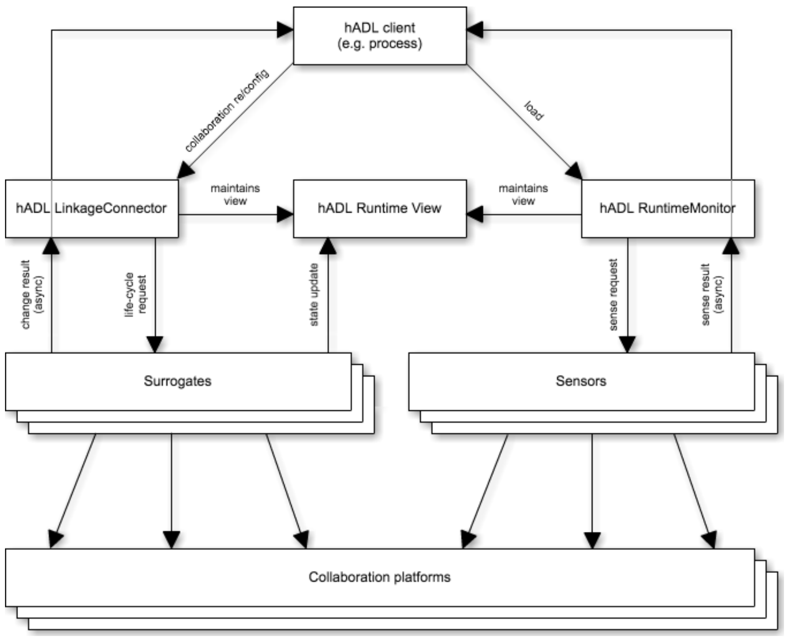


Figure 3.1: Architecture of hADL Framework

- The hADL client is possibly executed from tasks of a process. Therefore an abstraction mechanism is provided by the DSL. The translated Java equivalent of this mechanism are methods. These methods are conveniently invokable from process' task implementations.

- The DSL adds additional checks, such that DSL code emits only valid Java/hADL client code.

- A basic error handling mechanism provides the calling code (e.g. task) with a list of errors that occurred during execution of collaboration changes.

- The implementation of the evaluation scenario provides working Surrogates and Sensors for HipChat, Bitbucket and Agilefant.

3.2 Main Aspects for using a Domain-Specific Language

This work's DSL offers two major aspects which extend the current state of hADL. These two purposes are (i) providing means for modeling valid and consistent hADL instances, and (ii) generating Java hADL client code that interacts with the hADL framework. The following subsections discuss the two purposes of the DSL below, and show the benefits using it over plain Java by means of examples.

Valid and consistent hADL instance modeling

In order to achieve valid and consistent hADL instances, the DSL provides a concise but expressive syntax. The syntax provides hADL linkage primitives, a mechanism to load (sense) hADL-Elements from existing hADL-Operational-Elements, and a task abstraction defined by a sequence of hADL statements with input and output parameters. All the elements of this feature set are described in detail later in this chapter (see 3.3). On top of the new syntax, the DSL adds constraint and type checks that ensure validity of DSL/hADL programs (see 3.4). IDE features (auto-completion and syntax highlighting) are added such that the production of DSL code is more convenient for the developer (see 4). These aspects are completely independent from the actual implementation of hADL.

Figure 3.2 shows a simple hADL model, where two CollaborationObjects (`scrum.obj.Sprint` and `scrum.obj.Story`) are connected through a ObjectReference (`code.obj.containsStory`). Assume we want to load all stories of a sprint. Apart from the actual lines of code necessary to write that statement(s), the programmer needs to take care in the Java version that the hADL-Elements are identified with the correct Identifiers (IDs) and that these IDs are actually valid for the load statement(s). For these checks the hADL framework exposes the `ModelTypesUtil` class, which provides Java access to the hADL model. Hence validity checks of all hADL statements are obligatory. Furthermore these checks are performed at runtime which yield in potential errors at runtime when using the Java API.

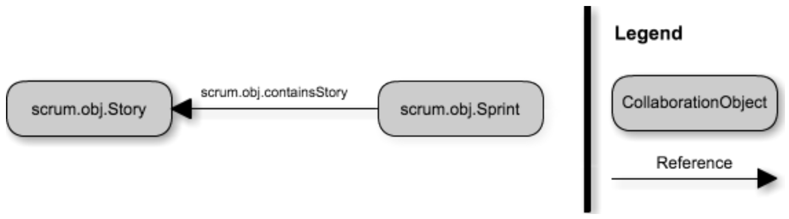


Figure 3.2: hADL Model Sprint to Story

The biggest benefit of the DSL is that these validity checks are performed at compile time without additional work to be done by the developer. As checks are done at compile time, the developer is presented with appropriate errors while defining collaboration instances. This removes errors while executing hADL collaborations, that are solely due to model to instance incongruities. Hence only valid instance declarations are possible when defining collaborations with the DSL. Furthermore the IDE features offer helpful suggestions to the developer during design time of collaboration instances.

hADL client code generation

This second aspect includes (i) the actual translation of DSL syntax into hADL client code (e.g. Java) that calls the hADL API, (ii) the abstraction mechanism through Java methods, (iii) runtime checks where compile time checks are not possible due to the design of the hADL framework (e.g. hADL type checks for input parameters), (iv) a runtime error handling system, (v) a way to pass a hADL context to and retrieve the context from collaboration abstractions, and (vi) hADL set up code.

The main benefit of the client code generation is that the developer of collaboration instances has to do a lot less typing. The example given below demonstrates the effort reduction by the DSL. Listing 3.1 shows the Java version and Listing 3.2 shows the DSL version of the acquirement of a sprint.

Listing 3.1 shows how a CollaborationObject, here the one that represents the Sprint with the ID 156935, is acquired. For simplicity reasons the example does not include loading the hADL-model (line 2), setting up the Guice injector and therefore the hADL runtime framework (line 3) and initialising the LinkageConnector (line 4).

```
 1 public void acquireSprint() {
 2   HadlModel model = loadModel();
 3   Injector i = setUpInjector(model);
 4   HADLLinkageConnector lc = setUpLinkageConnector(i,
      model);
 5
 6   ModelTypesUtil mtu = i.getInstance(ModelTypesUtil.
      class);
 7
 8   THADLarchElement sprintType = mtu.getById("scrum.obj.
      Sprint");
 9   // check if sprintType exists
10
11   TAgilefantResourceDescriptor rd = new
      TAgilefantResourceDescriptor();
12   rd.setId("sprint1");
13   rd.setName("Sprint1");
14   rd.setAgilefantId(156935);
15
16   List<Entry<THADLarchElement, TResourceDescriptor>>
      mapping = new ArrayList<>();
17   mapping.add(new AbstractMap.SimpleEntry<>(sprintType,
      rd));
18
19   Observable<SurrogateEvent> o = lc.
      acquireResourcesAsElements(mapping);
20
21   o.subscribe(new Observer<SurrogateEvent>() {
22     @Override
23     public void onCompleted() {
      // all elements acquired
    }

    @Override
    public void onError(Throwable e) {
      // error while acquiring
      // check what kind of error and handle it
    }

    @Override
    public void onNext(SurrogateEvent t) {
      // element acquired
      // safe element somewhere for later access
    }
  });
}
```

Listing 3.1: hADL Example for Acquiring a Sprint

The example in Listing 3.1 clearly indicates that a lot of boilerplate code needs to be written for a trivial task like acquiring an element. Setting up the hADL framework, dealing with hADL types through the ModelTypesUtil, putting together the required data types (variables mapping and rd), handling errors and using the asynchronous API of RXJava [36] in a callback fashion is a burden for developers.

Listing 3.2 shows the equivalent (also a bit simplified) DSL version of acquiring a sprint opposed to the Java version in Listing 3.1. Only three statements are needed to obtain the same effect as in the Java example. (i) Line 1 defines which model should be loaded; (ii) line 5 specifies a factory for obtaining ResourceDescriptors; and (iii) lines 10-12 (split into three lines for improving readability) defines which type with which ResourceDescriptor is acquired and assigned to which variable (s).

```
1 hADL "/Users/Christoph/TU/Diplomarbeit/Code/Evaluation/
     src/main/resources/agile-hadl.xml"
2
3 ...
4
5 resourceDescriptorFactory agilefantRd
6   class net.laaber.mt.hadl.agilefant.resourceDescriptor.
       AgilefantResourceDescriptorFactory
7
8 ...
9
```

```
acquire scrum.obj.Sprint
  with agilefantRd.agilefant("Sprint1", 156935)
  as s
```

Listing 3.2: DSL Version of Acquiring a Sprint

3.3 Language Specification

This section describes what the language elements (syntax) are and how the semantics of those is. First it describes elements that are necessary for set up (see 3.3.1). It does not provide information on how the grammar is implemented. This is described in Section 4.1.2. After that Java types and hADL types are discussed (see 3.3.3). Third the main abstraction mechanism is introduced (see 3.3.2). Following the definition of that abstraction mechanism, the elements for providing input and defining output to it are described (see 3.3.4). Fifth the hADL functionalities exposed by the LinkageConnector and the RuntimeMonitor is shown as hADL primitives (see 3.3.5). Section 3.3.6 describes what kind of variables are available in the DSL. And last it gives insight in how iteration is implemented and what it is needed for (see 3.3.7).

3.3.1 Set Up Statements

Before starting to define collaborations, set up of the hADL runtime framework and the IDE is necessary. Because it is the Design chapter, it just briefly mentions what the IDE set up is for, but does not go into detail how it is used. This is subject of Chapter 4. The set up language elements are described below.

hADL Model File

The hADL model is essential for all further features of the DSL. These features include: handling hADL types through the ModelTypesUtil (see 3.4) and therefore IDE features such as type checks and auto-completions; and set up code for the hADL runtime framework (see 4). As this model is crucial for everything else in the script it is the very first statement DSL.

Listing 3.3 shows how the path to the hADL model file is defined in the DSL. *path_to_hadl_file* is the absolute file system path in Unix format (e.g. /Users/Christoph/TU/Diplomarbeit/Code/Evaluation/src/main/resources/agile-hadl.xml).

```
hADL "path_to_hadl_file"
```

Listing 3.3: Definition of hADL Model File Location

Generated Java File Name

The next setting is the file name of the generated Java file. The keyword name `className` implies that the DSL is implemented to produce a single Java file, hence a single Java class (see 4). This set up parameter is mandatory that the DSL-to-Java-compiler knows where to create what class. *`fully_qualified_name`* is a Java class name declaration with fully qualified path (e.g. net.laaber.mt.evaluation.Tasks) (see Listing 3.4).

```
className fully_qualified_name
```

Listing 3.4: Definition of Generated Java File

ResourceDescriptor Factories

ResourceDescriptors encapsulate information that is needed during runtime to identify a hADL-Element with a resource on a collaboration platform (see 2.5). In order to provide the best possible User Experience (UX) the DSL introduces ResourceDescriptor factories. This statement introduces a new global variable with the name *`var_name`* that references the Java class at *`fully_qualified_name`* (e.g. net.laaber.mt.hadl.agilefant. resourceDescriptor.AgilefantResourceDescriptorFactory) (see Listing 3.5). Multiple ResourceDescriptor Factories can be defined.

```
resourceDescriptorFactory var_name class
    fully_qualified_name
```

Listing 3.5: Definition of a ResourceDescriptor Factory

The specified ResourceDescriptor factory is used to create ResourceDescriptors when acquiring a hADL-Element (see 3.3.5). Listing 3.16 shows how a ResourceDescriptor factory is used to create a ResourceDescriptor. *`var_name`* is a variable name referring to a ResourceDescriptor factory (e.g. *`var_name`* in Listing 3.5). After the variable name a dot is required. Methods of the ResourceDescriptor factory, which return an instance of `TResourceDescriptor` (for fully qualified name see Section 2.5), are allowed after the dot (instead of `resourceDescriptor`). Then, surrounded by braces, a list of Java-type parameters of arbitrary size are required. The parameters have the same restrictions as described in Section 3.3.3.

```
var_name.resourceDescriptor(param1, param2)
```

Listing 3.6: Creation of a ResourceDescriptor

Parameters of type `String` can have a value of type `int/Integer` (*`var1`*) appended to it (see Listing 3.7). This feature is of interest when acquiring hADL-Elements within an iteration (see 3.3.7) by using the iteration's counter variable.

```
var_name.resourceDescriptor("a String" + var1, param2)
```

Listing 3.7: Appending Numeric Values to ResourceDescriptor String Parameters

Sensor Factory

Section 2.5 introduces two runtime concepts that interact with collaboration platforms: (i) Surrogates and (ii) Sensors. Both are created by registered instances of `SurogateFactory` or respectively `SensorFactory` (for fully qualified names see 2.5). The runtime framework has a resolver built in, that extracts information about the Surrogate factory from the hADL model file. As Sensors are a newer feature of hADL, there is no resolver yet available. Hence specification of the Sensor factory with the DSL is required.

Listing 3.8 shows the statement to define the location of the Sensor factory. *`fully_qualified_name`* specifies the location of the Java file that contains the Sensor factory class (e.g. net.laaber.mt.evaluation.sensor.DslSensorFactory).

```
sensorFactory fully_qualified_name
```

Listing 3.8: Definition of the Sensor Factory

3.3.2 Tasks

After having set up the DSL and the hADL runtime framework, the only other top-level construct that can be specified are tasks. A Task is the logical unit, which is supposed to be called from a process-centric language. Figure 3.3 shows an exemplary process with tasks *Process Task A* and *Process Task B* that call the respective DSL's tasks *DSL Task C* and *DSL Task D*.

Listing 3.9 shows the definition of Tasks (i.e. *DSL Task C* and *DSL Task D*) with the DSL. Every Task is identified by its *`task_name`* which must be unique per DSL script. The body of a Task, enclosed by curly braces, defines *`inputs`* (see 3.3.4), *`statements`* and *`outputs`* (see 3.3.4). The possible *`statements`* are hADL primitives (see 3.3.5), variables (see 3.3.6) and iterations (see 3.3.7). At least one *`statement`* needs to be present. The execution order of the *`statements`* is sequentially.

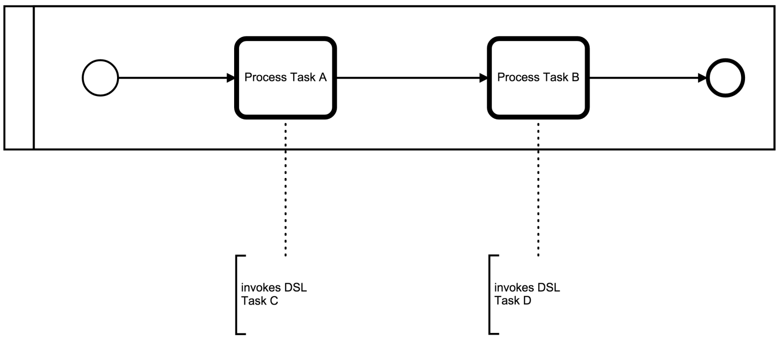


Figure 3.3: Process-Centric Language to DSL

```
task task_name {
  inputs
  statements
  outputs
}
```

Listing 3.9: Definition of a Task

3.3.3 Types

The DSL supports two kinds of types: hADL types and regular Java types. A simple hADL type is specified by a hADL-Language-Element's ID. Listing 3.10 shows the definition of an ID on a HumanComponent by supplying an id-Tag. The actual ID is represented by *hadl_id*. CollaborationConnectors, CollaborationObjects, References, Links and Actions also have id-Tags. The DSL also supports List-types for hADL types, where the *hadl_id* is enclosed by square brackets (denoted as [*hadl_id*]).

```
<component id="hadl_id">
  ...
</component>
```

Listing 3.10: hADL HumanComponent Tag

Listing 3.11 shows variable declarations with a simple hADL type on line 1 and with a list-type on line 2. *hadl_type* corresponds to *hadl_id* in Listing 3.10. Section 3.3.6 gives a detailed explanation on variables.

Every hADL-Language-Element has a corresponding Java type, which is described in Section 2.5 and is visually presented in Table 2.1.

```
var var1 : hadl_type
var var2 : [hadl_type]
```

Listing 3.11: Simple and List-Type hADL Variables

Java types are supported as input parameters (see 3.3.4), as parameters to ResourceDescriptor factory calls (see ResourceDescriptor factories in 3.3.1), in the form of iteration counters (see 3.3.7) and as output names (see 3.3.4). For simplicity reasons the design restricts the set of Java types to `String`, `int/Integer`, `float/Float` and `double/Double`.

3.3.4 Input and Output

Section 3.3.2 mentions that every task specifies *`inputs`* and *`outputs`* (see Listing 3.9) for providing data to a collaborative task and getting data as a result of a collaborative task. Input parameters can either be a hADL type or a Java type. Any variable from the Task can be returned as output parameter.

Listing 3.12 shows the different ways to specify input parameters to a Task. Line 1 shows the definition of an input parameter of simple hADL type *`hadl_type`*, line 2 shows the definition of an input parameter of hADL List-type *`hadl_type`*, and line 3 shows the definition of an input parameter of Java type *`java_type`*. *`var1`*, *`var2`* and *`var3`* are placeholders for variable names, and *`java_type`* represents a Java type (for restrictions see 3.3.3). Zero or more input parameters are eligible per Task. Input parameters must be the first statements of a Task body.

```
1 in var1 : hadl_type
2 in var2 : [hadl_type]
3 javaIn var3 : java_type
```

Listing 3.12: Definition of Input Parameters

As with input parameters, output parameters are optional and must be specified as the last statements of a Task body. *`var1`* represents a variable name that is defined within the scope of the Task (more on variable scopes in 3.3.6). The value of this variable is returned from the Task, which can be referred to by the caller through *`var2`*. *`var2`* is called output parameter name, which must be unique within a Task.

```
out var1 as "var2"
```

Listing 3.13: Definition of Output Parameter

3.3.5 hADL Primitives

The hADL primitives described in this section include functionality exposed by the LinkageConnector and the RuntimeMonitor of the runtime framework [33]. These are (i) acquire, (ii) release, (iii) link, (iv) unlink, (v) reference, (vi) dereference, (vii) stopping and (viii) starting of Surrogate scopes, and (ix) loading hADL-Operational-Elements from existing ones.

Acquire and Release

Acquire and release are the initial and terminal operations in the lifetime of HumanComponents, CollaborationConnectors and CollaborationObjects (see 2.5). Hence only hADL types that are HumanComponents, CollaborationConnectors or CollaborationObjects are eligible as *hadl_type* in Listing 3.14 on line 1. *rd* on the same line refers to a ResourceDescriptor as created in Listing 3.16. After the `as` keyword the name of the variable is specified (*var_name*), which holds the result of the acquire statement. Depending on the hADL type the result is either an OperationalComponent, OperationalConnector or OperationalObject (see 2.5).

Line 3 of Listing 3.14 shows how already acquired hADL-Operational-Elements can be released again. Therefore the variable name *var_name* needs to refer to an OperationalComponent, OperationalConnector or OperationalObject.

```
1 acquire hadl_type with rd as var_name
2 ...
3 release var_name
```

Listing 3.14: Acquire of hADL-Elements and Release of hADL-Operational-Elements

Link and Unlink

Link creates a hADL link between an OperationalCollaborator and an OperationalObject. Line 1 in Listing 3.15 shows the link statement. *var1* is a variable that references an OperationalCollaborator and *var2* is one that references an OperationalObject. *hadl_type* specifies which Link between *var1* and *var2* is used for linking those together. Again the variable name of the result is specified after the `as` keyword (*var3*). The result of a link operation is an OperationalLink.

Unlink removes an existing OperationalLink. Line 3 shows how unlinking is done. *var3* is the name of an OperationalLink variable.

```
link var1 and var2 by hadl_type as var3
...
unlink var3
```

Listing 3.15: Link and Unlink of hADL-Operational-Elements

Reference and Dereference

Reference is the polymorphic statement for creating both CollaboratorReferences and ObjectReferences. Line 1 in Listing 3.16 shows the statement for referencing hADL-Operational-Elements. If `var1` is an OperationalCollaborator, then `var2` is an OperationalCollaborator and vice versa. Conversely if `var1` is an OperationalObject, then `var2` is an OperationalObject and vice versa as well. Similar to Link, `hadl_type` defines the Reference (either CollaboratorReference or ObjectReference) that is used to relate `var1` and `var2`. The result of the reference statement is saved to `var3`. Depending on whether `var1` and `var2` are OperationalCollaborators or OperationalObjects the result is either an OperationalCollabRef or an OperationalObjectRef.

Line 3 shows the polymorphic dereference statement, which takes as input a variable name (`var3`) that references either an OperationalCollabRef or an OperationalObjectRef.

```
1 reference from var1 to var2 with hadl_type as var3
2 ...
3 dereference var3
```

Listing 3.16: Reference and Dereference of hADL-Operational-Elements

Stop and Start Surrogate Scope

Before any changes to Links (link and unlink) and References (reference and dereference) are performed the Surrogate scope must be stopped. To apply the changes made since the last `stopScope`, `startScope` must be called. Listing 3.17 shows the statements for stopping and starting Surrogate scopes.

```
stopScope
...
startScope
```

Listing 3.17: Stopping and Starting of Surrogate Scope

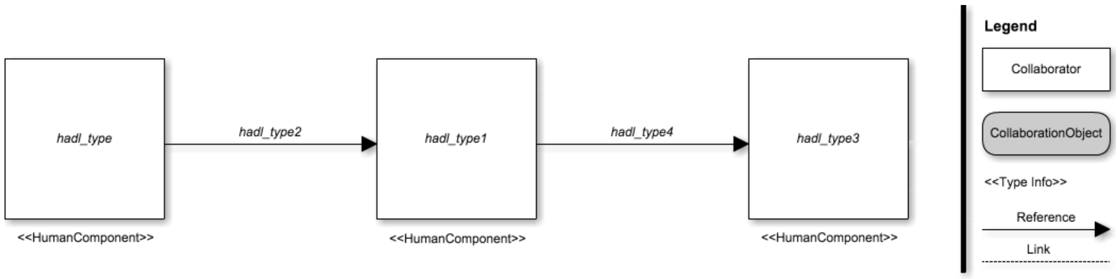


Figure 3.4: hADL Load Example Model

Load

All previous statements affected the LinkageConnector of the hADL runtime framework [33]. The load statement is the first and only one that concerns the RuntimeMonitor (as described in Section 2.5). Figure 3.4 shows a graphical representation of a hADL model. It consists of three HumanComponents (white squares) connected through CollaboratorReferences. The left HumanComponent of type *`hadl_type`* is connected through a CollaboratorReference of type *`hadl_type2`* to the middle HumanComponent of type *`hadl_type1`*. Which is in turn connected through a CollaboratorReference of type *`hadl_type4`* to the right HumanComponent of type *`hadl_type3`*.

There are two types of load scenarios. Scenario 1 (see Listing 3.18) is when the hADL-Operational-Element to load is directly connected (through a Link or a Reference) to the hADL-Operational-Element we start from (*`var1`*). Scenario 2 (see Listing 3.19) is when the hADL-Operational-Element to load is indirectly (through other hADL-Elements) connected to the hADL-Operational-Element we start from (*`var1`*). In general *`var1`* in both scenarios is a variable referring to either an OperationalCollaborator or an OperationalObject. In the example (see Figure 3.4) it is an OperationalComponent. The types in the hADL model in Figure 3.4 refer to the same named types in Listings 3.18 and 3.19.

Scenario 1 defines that the starting point (*`var1`*) is immediately succeeded by a `load` keyword. After `load` a construct is inserted, that lists the hADL type of the hADL-Element to load (*`hadl_type1`*) separated by a colon (`:`) from the hADL type of the hADL-Connective (*`hadl_type2`*). The result is a hADL List-type value (*`hadl_type1`*) of the loaded hADL-Operational-Elements, which is saved in *`var2`*.

```
startingFrom var1 load hadl_type1:hadl_type2 as var2
```

Listing 3.18: Scenario 1: Loading of Directly Connected hADL-Elements

The difference between Scenario 1 and Scenario 2 is that hADL-Operational-Elements are loaded through other hADL-Elements. These load steps between the starting point (*var1*) and the desired hADL-Operational-Elements (after `load` keyword) are defined with the `via` keyword. It is eligible to define an arbitrary amount of `via` elements. The syntactic construct needed after a `via` keyword is identical to the construct after the `load` keyword. The result is the same as in Scenario 1, but with hADL type *hadl_type3*.

```
startingFrom var1 via hadl_type1:hadl_type2 load
    hadl_type3:hadl_type4 as var2
```

Listing 3.19: Scenario 2: Loading of Indirectly Connected hADL-Elements

To process the results of a load statement the iteration statement is used (see 3.3.7).

3.3.6 Variables

Previous Sections introduced ResourceDescriptor factories (see 3.3.1), types of the DSL (see 3.3.3), input parameters (see 3.3.4), and variables created by hADL primitives acquire, link, reference and load (see 3.3.5). All of these concepts are either directly related to variables (ResourceDescriptor factories, input parameters and hADL primitives), as they create variables, or indirectly related, as all variables have types.

This section describes how we can explicitly declare variables, assign values to it and how these explicit variables differ from input parameters and variables created through hADL primitives.

Every variable in the DSL has five properties: (i) a name, (ii) a hADL type, (iii) a Java type, (iv) an assignment semantics, and (v) a scope availability. First scoping is defined, second the previous mentioned variable declarations are discussed, and third real variables are introduced.

Variable scopes are a well known concept in programming languages. The DSL also supports different variable scopes where a particular variable is visible and accessible. Theoretically with the DSL an arbitrary depth of scopes is possible. All variables from outer scopes are accessible from nested scopes. The outer most scope (scope-level 0) is where set up statements are specified. The only variables from this scope are ResourceDescriptor factory variables. The next inner scopes are the ones defined by Tasks (scope-level 1). Every Task defines its own scope, therefore in two different Tasks it is possible to define variables with the same name. On the same scope-level (and enclosing scopes) every variable name must be unique. Within a Task nested scopes are created through Iterations (see 3.3.7).

ResourceDescriptor factories create variables that have value semantics. They do not have a hADL type, and their Java type is the class they refer to. As they are defined in the outermost scope, they are accessible everywhere.

Input parameters also have value semantics. If they are defined with the keyword `in`, then they have a hADL type which is specified after the colon (`:`) and are called hADL input parameters. The hADL type can either be simple or of List-type and the Java type is inferred from the hADL type depending on the hADL-Language-Element. Table 2.2 in Section 2.5 shows which hADL-Language-Element corresponds to which operational Java type. If the input parameter is defined with the keyword `javaIn` then it does not have a hADL type und the Java type is specified after the colon (`:`). These input parameters are called Java input parameters. See Listing 3.12 in Section 3.3.4.

hADL primitives variables are created through acquire, link, reference and load statements. The assignment semantic is also value semantics.

Acquire's hADL type depends on which hADL-Element was acquired (*`hadl_type`* in Listing 3.14), so does the Java type.

Link's hADL type depends on the link that was defined to link the OperationalCollaborator and the OperationalObject (*`hadl_type`* in Listing 3.15). Link's Java type is always `TOperationalCollabLink`.

Reference's hADL type depends on the reference that was used to relate the OperationalCollaborators or OperationalObjects (*`hadl_type`* in Listing 3.16). Reference's Java type is `TOperationalCollabRef` or `TOperationalObjectRef` depending on the hADL-Operational-Elements that are related.

Load's hADL type depends on the type of the hADL-Element specified after the `load` keyword (*`hadl_type1`* in Listing 3.18 and *`hadl_type3`* in Listing 3.19). As load returns a hADL List-type, the corresponding Java type is of `List<T>` type, where `T` is the corresponding Java type of the loaded hADL-Operational-Elements (see Table 2.2).

Real variables This paragraph introduces the new concept of real variables. All the previous mentioned variables have one thing in common, which is that they all have value semantics. Meaning that the are immutable or do not support destructive assignment. The DSL has a feature which supports real variables with destructive assignment semantics.

Real variables are defined similar to hADL input parameters, but with the `var` instead of the `in` keyword. `var` is followed by the variable name (*`var1`*). Then after a colon (`:`) the hADL type is specified, which can either be simple or of List-type.

Listing 3.20 shows on line 1 the declaration of a simple hADL variable. To be of use, real variables support not just declaration, but also assignment. Assignment of a real variable is shown on line 3. The value of *`var2`* is assigned to the newly created real variable *`var1`*. Therefore the hADL types of *`var1`* and *`var2`* must match.

Listing 3.21 shows the other option for defining real variables. On line 1 a variable of hADL List-type is declared. Reassignments (`assign`) of List-type variables is supported.

```
1 var var1 : hadl_type
2 ...
3 assign var2 to var1
```

Listing 3.20: Defining and Using Simple Variables

A second operation which adds elements to a List-type variable is exclusive to this variables. Therefore *var2* must be of *hadl_type* from line 1.

```
1 var var1 : [hadl_type]
2 ...
3 add var2 to var1
```

Listing 3.21: Defining and Using List-Type Variables

What do we need real variables for? Recall the definition of variable scopes from above: Variables are only accessible from the same and enclosing scope-levels. When a new scope-level is introduced, through iterations, all variables within this new scope-level are not accessible from the enclosing scope (the Task scope-level). Hence these nested variables are not accessible by `out` statements. Real variables come to help by allowing to define a variable on the Task scope-level and assigning (`assign`) or adding (`add`) a value from an inner scope. For an example see Section 3.3.7.

3.3.7 Iteration

Iteration is the last statement in this language specification that is introduced. We know from basic programming language courses, that if a list of elements needs processing of each contained element an iteration mechanism is needed. Depending on the programming language paradigm, loops in iterative languages or higher-order functions in functional languages are used to accomplish iteration.

This DSL takes an iterative approach, as the definition of tasks already relies on an iterative approach to which statements it consists of. It is best compared to Java's *enhanced for* statement [37] which is used for iterating over Java Collections and Arrays. The DSL's iteration statement comes in two forms, with and without access to a loop counter variable.

The simple form without loop counter is shown in Listing 3.22. *var1* represents a previously defined variable of hADL List-type `[hadl_type]`. *var2* represents a different element of the list in every iteration, where the type has to be `hadl_type`. The element's variable (*var2*) is accessible from within the loop body, which is enclosed by curly braces.

The body itself consists again of an arbitrary number of statements. The set of statements include all hADL primitives, real variable declarations and usage of those and iterations.

```
for all var1 as var2 {
  statements
}
```

Listing 3.22: Iterating over List-Type Variables

The other form where a loop counter variable is defined is shown in Listing 3.23. The name of the counter variable (*var3*) is declared after the keyword `counter`. Its Java type is `int` and it is accessible within the iteration's body. The rest of the iteration statement is exactly the same as the simple form.

```
for all var1 as var2 counter var3 {
  statements
}
```

Listing 3.23: Iteration with Loop Counter

As iterations are part of the statements allowed in iteration bodies, an arbitrary number of nested iterations is possible. Because every iteration statement creates a new scope-level, an arbitrary number of nested scopes can be constructed. Variables (all types) defined within an iteration are not visible to enclosing scopes.

Listing 3.24 shows an example where variables defined on an outer scope-level are accessed from inner scope-levels (iteration). In addition it shows the typical usage of loop counter variables.

In order to access those variables created within an iteration we need to declare a real variable on the surrounding scope-level (`acquiredElements` on line 1). After a new element is acquired and stored in variable `acqEl` on line 3, it is added to the List-type variable `acquiredElements` on line 4. Concerning the types for the `add` statement: (i) line 1 shows that the hADL type of `acquiredElements` is `[`*`hadl_type`*`]` and (ii) line 3 shows that the hADL type of `acqEl` is *`hadl_type`*. Therefore the `add` statement on line 4 is valid. It is possible to e.g. return the acquired elements (`acquiredElements`) from the Task through an `out` statement.

Line 3 shows how a loop counter variable (`i`) is used as input parameter of the ResourceDescriptor factory method. In this way it is possible to acquire elements within iterations and have their ResourceDescriptor properties (e.g. ID or name) altered.

```
var acquiredElements : [hadl_type]
for all loadedElements as el counter i {
  acquire hadl_type with rd.resourceDescriptor("acq" + i
      ) as acqEl
  add acqEl to acquiredElements
}
```

Listing 3.24: Iteration Example: Nested Scopes and Usage of Loop Counter Variable

3.4 Validity and Consistency Checks

Section (see 3.2) presented the main aspects for using the DSL over writing hADL clients with Java. This section is about how valid and consistent hADL collaboration instances are achieved.

In section 3.3 on language elements remarks are made on conditions that have to be met in order for the DSL to work properly. The following sections present all these constraints and Section 4.1.4 shows how these checks are implemented.

3.4.1 Name Checks

This section describes the names that the DSL introduces and how they are checked. There are three kinds of names: (i) variable names, (ii) task names, and (iii) output names.

Variable names need to be unique in the scope they are in. This is important for two reasons, first that they can be clearly identified when they are referenced and second that variables from inner scope-levels do not shadow over variables with the same name of outer scope-levels. Variable names that need to be unique are ResourceDescriptor variables (see 3.3.1), input parameters (see 3.3.4), variables from hADL primitives (see 3.3.5), real variables (see 3.3.6) and iteration variables (see 3.3.7).

ResourceDescriptor variables are on scope-level 0 and are therefore accessible everywhere in a DSL file. Input parameters and variables created directly in a task must be unique on a per Task basis (scope-level 1). For example two different Tasks can both have an input parameter with the name *foo* (see Listing 3.25), but a single task with an input parameter *foo* and a variable from an acquire statement named *foo* is prohibited (see Listing 3.26).

The iteration statement introduces one (simple form) or two (form with loop counter variable) new variables itself and possibly contains multiple variables in its body. These need to be unique on the scope-level of the iteration. If the iteration statement itself is on

```
...

task A {
  in param1 : [hadl_type]
  ...
}

task B {
  in param1 : [hadl_type]
  ...
}
```

Listing 3.25: Good Example: Input Parameter Names

```
...

task A {
  in var1 : [hadl_type]

  acquire hadl_type2 with rd as var1
  ...
}
```

Listing 3.26: Bad Example: Duplicate Variable Names

scope-level 1, the variable names must be unique from scope-level 2 (the iteration body) onwards. Two iterations on the same scope-level can have the same element variable name, loop counter variable name and introduce the same variable names in its body (see Listing 3.27).

```
...

task A {
  ...
  for all list as var1 counter i {
    var var2 : hadl_type
    ...
  }

  for all list as var1 counter i {
    var var2 : hadl_type
    ...
  }
  ...
}
```

Listing 3.27: Good Example: Same Variable Names in Different Scopes

Listing 3.28 shows a bad example, because *var1* on line 7 is already declared on line 1 and*var2* on line 8 is already declared on line 7 and line 5.

```
1  ...
2
3  task A {
4    in var1 : hadl_type
5    in var2 : hadl_type2
6    ...
7    for all list as var1 counter var2{
8      var var2 : hadl_type3
9    }
10 }
```

Listing 3.28: Bad Example: Same Variable Names in Same Scope

Task names must be unique on a per file basis. It is possible to define multiple Tasks in one file and every name must be unique. This is necessary for identification purposes (see 3.3.2).

Output names which are defined after the `as` keyword in `out` statements (see 3.3.4), must be unique on a per Task basis. A single task can not have multiple outputs, where the output name is identical, because otherwise the caller of the Task can not differentiate between the different outputs.

3.4.2 Type Checks

The second checks that the DSL performs are type checks. As Section 3.3.3 explains there are two different types that are of interest: (i) hADL types that are identified by the id-tag of the XML element, and (ii) Java types of the variables. In the following the different type checks executed by the DSL compiler are listed.

hADL type checks

hADL type checks are performed to ensure that statements, which make use of hADL types within a Task description, are aware of those types and only allow correctly typed programs. When using the hADL runtime framework to construct a hADL client, the programmer specifies hADL types as Java `Strings` and has only the type safety provided by the Java type system at compile-time. hADL types are not checked until runtime execution of the collaboration.

The thesis already presented the conditions that have to be met for every statement in Section 3.3, in order to write hADL type correct statements. In the following it showcases the differences between the Java version and the DSL version and highlight where type checks are performed in the DSL version. Sections 4.1.4 and 4.1.5 present how these type checks are implemented.

Acquire Listing 3.29 shows what steps are necessary to acquire an element. Compared to the Java version, we can see the DSL version in Listing 3.30. Line 1 in the Java version retrieves the Java object that represents the hADL type `scrum.obj.Sprint`. The Java version returns a generic `THADLarchElement` object and not the object of the most specific type. As the hADL type is provided as `String`, we can not be sure that this type even exists.

```
 1    THADLarchElement sprintType = mtu.getById("scrum.obj.
       Sprint");
 2    // check if sprintType exists
 3
 4    TAgilefantResourceDescriptor rd = new
       TAgilefantResourceDescriptor();
 5    rd.setId("sprint1");
 6    rd.setName("Sprint1");
 7    rd.setAgilefantId(156935);
 8
 9    List<Entry<THADLarchElement, TResourceDescriptor>>
       mapping = new ArrayList<>();
10    mapping.add(new AbstractMap.SimpleEntry<>(sprintType,
       rd));
11
12    Observable<SurrogateEvent> o = lc.
       acquireResourcesAsElements(mapping);
```

Listing 3.29: Java Acquire Example

The DSL version provides the hADL type after the `acquire` statement and checks if a hADL-Language-Element of that type exists and if it is a hADL-Element and therefore qualifies as one that can be acquired. Therefore the DSL version is always type correct. If in the Java version the element `sprintType` would not exist or is an element that can not be acquired (Links, References, Actions), an error would be sent to the observable `o`.

```
acquire scrum.obj.Sprint with rd.resourceDescriptor("
    sprint1", "Sprint1", 156935) as var1
```

Listing 3.30: DSL Acquire Example

Reference Listing 3.31 shows a Java example on how to create a reference *`hadl_type`* between two hADL-Operational-Elements (*`var1`* and *`var2`*). It specifically shows how two OperationalObjects are referenced. `var1` and `var2` refers to OperationalObjects and `hadl_type` refers to an OperationalObjectRef. The problem is that the Java version does not have any information about the variables' (`var1`, `var2` and `hadl_type`) hADL types, and if `var1` and `var2` have a reference `hadl_type` in the hADL model connecting them. Listing 3.32 shows the DSL version of referencing two hADL-Operational-Elements. The DSL checks if *`var1`* and *`var2`* are connectable through a reference of type *`hadl_type`*. Therefore the reference statement is always hADL type correct.

```
List<Entry<TOperationalObject, TOperationalObject>>
    fromTo = new ArrayList<>();
fromTo.add(new AbstractMap.SimpleEntry<
    TOperationalObject, TOperationalObject>(var1, var2);

Observable<SurrogateEvent> o = lc.buildRelations(fromTo
    , hadl_type, false);
```

Listing 3.31: Java Reference Example

```
reference from var1 to var2 with hadl_type as var3
```

Listing 3.32: DSL Reference Example

Link Listing 3.33 depicts that Java version of how a link (`hadl_type`) between an OperationalCollaborator (`var1`) and an OperationalObject (`var2`) is established. Similar to the Java version of the reference statement, there are no compile-time checks if `var1` and `var2` can be linked with `hadl_type`. The DSL (see Listing 3.34) adds those compile-time type checks, such that *`var1`* and *`var2`* can be linked by *`hadl_type`*.

```
Observable<SurrogateEvent> o = lc.wire(var1, var2,
    hadl_type);
```

Listing 3.33: Java Link Example

```
link var1 and var2 by hadl_type as var3
```

Listing 3.34: DSL Link Example

Load Loads with the Java API of hADL are quite cumbersome to do. Listing 3.35 shows a simplified version on how to accomplish that. If only a particular hADL-Connective is of interest to follow for loading hADL-Operational-Elements, this hADL-Connective (`hadl_type2`) has to be added to a `SensingScope` (line 1). In a next step the load is executed with the starting hADL-Operational-Element and the previously specified `SensingScope`. The loaded elements are then retrieved through the Observable `o`. If consecutive loads starting from the loaded elements are desired (via functionality of the load statement of the DSL), the programmer must perform a new load from within the Observable with a new `SensingScope`. Again the Java version does not do any compile-time checks whether `var1` has a hADL-Connective of type `hadl_type2`. These compile-time checks are added by the DSL (see Listing 3.36), such that only *`hadl_type1`* and *`hadl_type2`* types are possible if *`var1`* has a hADL-Element of type *`hadl_type1`* connected to it through a hADL-Connective of type *`hadl_type2`*.

```
1 SensingScope s = new SensingScope();
2 s.getObjectRefs().add(hadl_type2);
3
4 Observable<LoadEvent> o = rm.loadFrom(var1, s);
```

Listing 3.35: Java Link Example

```
startingFrom var1 load hadl_type1:hadl_type2 as var2
```

Listing 3.36: DSL Link Example

Variables The variables declared with the keyword `var` also support hADL type checking. Java only knows about operational element types, the DSL also is aware of the particular hADL type of each variable. For example a Java list of OperationalComponents may contain OperationalComponents of different hADL types. A hADL List-type variable only contains elements of the particular hADL type. The compile-time type checks guarantee that an `assign` or `add` statement only allows assignment and inclusion of the right hADL types.

Java type checks

The DSL also supports Java type checks besides from hADL type checks. It does not add anything to the overall contributions of hADL, because the Java version naturally supports Java type checks. Nonetheless the next paragraphs list the Java type checks done in the DSL.

The Java type checks of hADL variables are implicitly successful, because if the hADL types match the Java types must match as well.

In the set up section in DSL scripts we must specify the location of the Sensor factory. The DSL checks whether the specified path leads to a Java class that implements the required interface `SensorFactory` (for fully qualified path see Section 2.5).

Creation of ResourceDescriptors as part of acquiring new hADL-Elements needs Java type checking as well. The method used for creating a ResourceDescriptor is required to return an instance of `TResourceDescriptor` (for fully qualified path see Section 2.5).

The last Java type checks that are performed by the DSL are when Java input parameters (`javaIn`) are used as parameters to ResourceDescriptor factory methods or as names for output parameters (`out`). ResourceDescriptor factory methods support the Java types `String`, `int/Integer`, `float/Float` and `double/Double`. Output parameter names must be of Java type `String`.

3.5 Error Handling

This last section in the chapter on design of the DSL is about errors and how they are treated by the language. There are two types of situations that need special treatment. On one hand there are exceptions, where exceptional situations occur which need treatment by the programmer. Such situations are the ones that indicate bugs in the DSL implementation or the hADL model, Surrogate or Sensor implementations. On the other hand there are failures that occur during execution of the hADL framework, such as Surrogate state change failures. Depending whether the error is an exception or a failure the DSL behaves differently. Errors may occur during the execution of hADL primitives (see 3.3.5).

Exceptions

Exceptions indicate bugs and therefore the program that calls the Tasks must stop immediately as no further execution makes sense. The programmer has to debug the application and fix the error before the program is restarted.

Failures

Failures are the more interesting case of errors as the program, that calls the Tasks, itself can decide which strategy is taken to deal with that failure. For example an acquire statement might fail because the network was down and therefore the collaboration platform could not be reached. A sophisticated program inspects the failure, understands that the failure was caused by unavailability of the network and retries the acquire statement after a few seconds.

The next paragraphs present a simple failure situation model that only informs the caller of a Task about the occurrence of a failure. A sophisticated failure handling system remains out of scope of this thesis.

A Task is a sequential composition of possibly multiple hADL primitives. If some hADL-Elements or hADL-Connectives have been created it is not as simple as just return the failure and let the caller handle it. Handling failure situations requires that the already successfully created elements are returned in addition to the failure, if an out statement for that variable exists.

Listing 3.37 shows the complete definition of Task A, where only set up statements are omitted. On lines 3 and 4 two hADL-Elements of the same kind (HumanComponent, CollaborationConnector, CollaborationObject) are acquired. Lines 7 and 8 define that the results of lines 3 and 4 are returned to the caller of Task A. Line 5 creates a reference between the previously acquired elements (*var1* and *var2*) and line 9 specifies that the created reference is returned as well.

Assume that the acquire statement on line 4 returns with a failure. In this case it is not possible to just return the failure, but also the already successfully created *var1* needs to be returned because of line 7.

The solution is that in addition to the explicitly specified output parameters an additional failure list is returned. The caller of a Task needs to check whether the failure list contains any elements. A Task succeeded if the failure list is empty. All specified output parameters that refer to variables that are defined before the failure occurred are returned as well.

In the example from above the output of Task A includes *var1* (specified on line 7) and the failure list with one element that specifies the failure which occurred during execution of line 5.

```
1 ...
2 task A {
3   acquire hadl_type1 with rd.resourceDescriptor("el1")
       as var1
4   acquire hadl_type2 with rd.resourceDescriptor("el2")
       as var2
5   reference from var1 to var2 with hadl_type3 as var3
```

```

    out var1 as "el1"
    out var2 as "el2"
    out var3 as "l"
}
```

Listing 3.37: Failure Handling Scenario

Failures in iterations require enhanced handling. If a failure occurs within an iteration, the failure is added to the failure list and the iteration immediately continues with the next element. After the iteration where the failure occurred iterated through all elements the Task returns. This behaviour applies to nested iterations as well.

CHAPTER 4

Implementation

Previous chapters discussed the design of the DSL on a conceptual level. What functionality should be present, how the syntax looks, what the semantics are, which validity and consistency checks it provides and how errors are handled. This chapter is about how the DSL is implemented such that all the design decisions are met. Section 4.1 introduces the used tools for implementing the DSL and point out, which features of the tool chain are used to implement which feature of the DSL. Section 4.2 presents a library that acts as an adapter to the hADL runtime framework. That library's purpose is to transform the hADL's functionality required by the DSL into a form which suits the design best.

The source code for the implementation of the DSL is split into two projects. The first being the one that implements the DSL and the second is the one that implements the library of Section 4.2. In the following the first project is referred to as the DSL-project [38] and the second is referred to as hADL-lib-project [39].

4.1 DSL Development with Xtext/Xtend

After deciding the domain of the DSL and designing the concepts of it, the selection of a suitable DSL development tool is crucial. Besides supporting the features defined in Chapter 3, the ease of development, relatively flat learning curve and editor/IDE support are important characteristics as basis for making a choice. There are two ways of implementing a DSL [40]. The internal and the external approach. The internal DSL is a particular API as part of a host language, whereas an external DSL is independently parsed from the host language. As internal host language Scala [41] and Haskell [42] were considered because those languages have a rich type system. Tools for external DSL development that were considered are Xtext [43] and Epsilon [44]. We decided to use Xtext as the DSL development framework because the documentation seemed good and the assistant advisor of this thesis had previous experience working with it.

Software	Version
Eclipse IDE	4.5.1
Xtext Software Development Kit (SDK)	2.8.4
Xtend	2.8.4

Table 4.1: Software Versions used for Development

Table 4.1 shows the versions of the main languages and frameworks that are used for development.

Xtend [45] is the programming language that is primarily used for developing with Xtext. The Eclipse Foundation, Inc. is responsible for developing the language. They advertise Xtend as a modernised Java where features such as extension methods, multiple dispatch (multi methods), everything is an expression and advanced type inference are supported. A language primer of Xtend is out of scope of this thesis. For developers familiar with Java it should be straight forward to learn the basics of Xtend. The Xtend website [45] and the Xtext website [43] provide plenty information to learn Xtend. Code examples which use features that are not common to Java developers are explained as they are presented. Xtend compiles into Java source code and therefore supports seamless integration of Java libraries with Xtend programs. This integration feature is of high interest to this work as the hADL framework is written in Java itself and tight integration of it and the DSL is essential.

Xtext is an Eclipse-based framework that provides all features required for developing programming languages and DSLs. The Xtext framework shows its power through many examples. One use case is Xtend, which is implemented using Xtext. The Xtext infrastructure includes a powerful grammar language, parser, linker, type checker, compiler, and IDE features for Eclipse. The following sections describe the syntax specification (see 4.1.2), the translation of DSL code into Java code (see 4.1.3), constraint checks through validation (see 4.1.4) and scopes (see 4.1.5), and syntax proposals and auto-completions (see 4.1.6).

4.1.1 Project Structure

This section describes how the DSL-project is structured. The DSL-project consists of multiple sub-projects. All sub-projects but the *net.laaber.mt.DSL.lib* sub-project are defined by Xtext. The sub-projects are listed below.

- *net.laaber.mt.DSL* is the main sub-project. It includes the grammar file, DI set up, the workflow file which defines the set up for the whole generation process and classes for java generation, scoping and validation. Additionally the sub-project includes classes for handling variables and types. Those additional classes are called from the validator, scope provider and proposal provider.

- *net.laaber.mt.DSL.lib* is the only sub-project that is not required by Xtext. It is the central point for dealing with external dependencies for all the other projects. Xtext is built as an Eclipse plug-in, which uses OSGi. OSGi handles dependencies between bundles through manifest files (META-INF/MANIFEST.MF). Maven and OSGi do not mix well, therefore it is not possible letting Maven handle the dependencies. All dependencies required by the DSL-project are in the *libs* folder.

- *net.laaber.mt.DSL.sdk* is an automatically generated sub-project where no modifications were made. Hence it is of no further interest to us.

- *net.laaber.mt.DSL.tests* is the sub-project where unit tests for the parser, validators and scopes are put.

- *net.laaber.mt.DSL.ui* deals with all UI specific functionality. Those include proposal providers, labeling, outline and quick fix. This work only customises the proposal provider. The default implementations of the other features are sufficient for this thesis' purpose.

4.1.2 Grammar Specification

The syntax as described in Section 3.3 is defined in Xtext through their special grammar file. The file is located in the main sub-project in the package `net.laaber.mt` and named *DSL.xtext*. A typical parser can be divided into four stages: (i) lexing, (ii) parsing, (iii) linking and (iv) validating. The first two are defined in the grammar file, the third is declared in the grammar file but needs additional semantic handling (see 4.1.5) and the fourth is solely defined through validators (see 4.1.4).

Not all features of the grammar are used in the DSL. The implementation uses existing terminal rules (lexer), defines custom parser rules, calls custom parser rules from other parser rules and defines cross-references which are later used by the linker (see 4.1.5).

Terminal Rules that the grammar specification uses are `ID` and `STRING`. `ID` specifies an identifier that starts with either a lowercase letter, an uppercase letter or an underscore, followed by a possibly empty set of characters that are either the same as the first character or a number from 0 to 9. `STRING` represents a String enclosed by double quotes (`"`). The DSL's grammar uses `ID`s for Task names, Variable names and ResourceDescriptor factory method names. `STRING`s are used as hADL file path, Output variable names and ResourceDescriptor factory method parameters.

Custom Parser Rules are the main building blocks in the DSL's grammar file. The first rule specified serves as the entry rule for the parser. Listing 4.1 shows the definition of the entry rule, which specifies the set up statements from Section 3.3.1. `Domainmodel` is the name of the custom parser rule. Between the colon (`:`) and the sem-colon (`;`) the body of the rule is defined. Between single quotes (`'`) keywords are defined. A hADL model file path is specified (line 2) by starting with the keyword `hADL` followed by a String (as specified by the terminal rule `STRING`). The String is assigned to the variable `file` through which it is accessible during Java code generation phase. Line 3 is similar to line 2 with the exception that instead of the terminal rule `STRING` the predefined parser rule `QualifiedName` is called. `QualifiedName` is valid for qualified names such as *net.laaber.mt.Tasks*. Line 4 introduces a new assignment with the operator +=, which assigns multiple values to the variable `rdFacs`. In combination with the + at the end of the line this assignments means that one or more `ResourceDescriptorFactory` rules must be present. Line 5 again is a single value assignment of the custom parser rule `SensorFactory`. Finally Line 6 defines that after the previous set up statements 0 or more `Task` specifications are required. The asterisk (`*`) at the end of the line indicates that 0 or more are expected.

```
1 Domainmodel:
2   'hADL' file = STRING
3   'className' className = QualifiedName
4   rdFacs += ResourceDescriptorFactory+
5   sensorFac = SensorFactory
6   tasks += Task*
7 ;
```

Listing 4.1: Xtext: Set Up Grammar Rule

Cross-references Listing 4.1 shows all grammar features used in the DSL grammar except cross-references. Usage of cross-references are shown in Listing 4.2. Lines 1 - 3 define the parser rule for hADL input parameters, where a new variable (`Var`) is created. Lines 5 - 7 show the parser rule for output parameters, where a previously created variable is referenced (`[Var]`). Xtext ensures that only variables (`Var`) are valid at this point. By default all created variables are allowed to insert there. If only specific variables are desired to be available at that point, a new scope provider for the parser rule `Output` must be implemented (see 4.1.5).

```
1 HadlInput:
2   'in' variable = Var ':' type = HadlType
3 ;
4
```

```
Output:
  'out' variable = [Var] 'as' key = OutputKey
;
```

Listing 4.2: Xtext: Cross-Reference Example

For the complete grammar specification see Appendix DSL Grammar Specification.

4.1.3 Java-generation

The next step after specifying the grammar is to translate DSL scripts into Java source code. First the following sections define how the desired output should look like and then they give a brief introduction how this is achieved with Xtext. From the chapter on language specification (see 3.3) we know that the DSL provides some basic set up statements first and then continues with Task specifications. There are two obvious strategies how translations can be done.

The next subsections contain references to type names of the Java SE API [46].

Possible Compilation Strategies

The first strategy is to translate every Task into an own Java class, where the Task name is equal to the class name. Input parameters could translate to properties with setters and output parameters are properties with getters, or input parameters are constructor parameters and the task body translates to the only method of `java.util.concurrent.Callable<V>` where `V` are the output parameters. Furthermore the code generator needs to compile the set up functionality to either a super class of all Tasks or into a separate class that is accessed from the Tasks through composition. The elegance of this strategy is that the different parts (Tasks and set up) is spread over multiple classes and with every Task being an instance of `Callable<V>` concurrency is easily applicable (through `java.util.concurrent.ExecutorServices`). The downsides are that the Java code generation process is more complex and the developer that calls the Tasks must instantiate an object per Task.

The other strategy is to compile everything into a single class. Tasks are translated into methods of that class and input parameters are translated to method parameters. Per design the DSL supports multiple output parameters, but Java only supports a single output parameter. The compiler tackles that problem by always returning an instance of `java.util.Map<String, Object>`. This strategy does not separate concerns as elegant as the first one, but makes translation easier. Furthermore clients that invoke Tasks only have to instantiate a single object.

The DSL implementation takes the second strategy and compiles all Tasks into a single Java class. The name of the class is defined in the set up section with the keyword `className`.

Xtext Compilation Infrastructure

Xtext provides an automatically generated stub which is the entry point to compiling your own language into Java source code. This class is located in the main sub-project in the package `net.laaber.mt.jvmmodel`. It is named `DSLJvmModelInferrer` and extends `AbstractModelInferrer`. The entry method which is called when translation starts has the signature as shown in Listing 4.3. The keyword `dispatch` indicates that this method supports multiple dispatch. `Domainmodel` is the name of the entry parser rule defined in the grammar file which is provided by Xtext as a class with all variables defined in the grammar accessible. The parameter `acceptor` is used for generating Java code by calling the method `accept` on it and providing the desired translations as parameter.

```
def dispatch void infer(Domainmodel element,
    IJvmDeclaredTypeAcceptor acceptor, boolean
    isPreIndexingPhase)
```

Listing 4.3: Entry Method for Java Code Generation

Compilation Steps

The first step of the compilation is to load the hADL model that was provided in the DSL script and initialise the `ModelTypesUtil` from the hADL runtime framework with it. The `ModelTypesUtil` exports convenience functions to work with the hADL model such as retrieving types from IDs. After that the variable manager is initialised (see 4.1.4). As last step the actual translation is performed. Translation consists of three main steps: (i) class creation, (ii) set up instructions, and (iii) Task creation.

Class Creation and Set Up Steps are the first steps during code translation. Listing 4.4 shows the usage of the parameter `acceptor`, the generation the class with the provided set up statement (`className`), retrieving variables from the parameter `element` and invoking the `addSetUp` method. `element` and `acceptor` are the method parameters from Listing 4.3. Line 2 shows the method invocation that returns an instance of a subtype of `org.eclipse.xtext.common.types.JvmDeclaredType`. The parameter is the class name that is provided in the set up section of the DSL script after the `className` statement. Line 3 calls a custom method that adds the `setUp` method to the created class, which set ups the Guice injector needed for the hADL runtime framework. It returns a data object that includes all necessary objects (e.g. LinkageConnector, HADL-runtimeMonitor) of type `net.laaber.mt.dsl.lib.hadl.ProcessScope` which is defined in the hADL-lib-project. Line 5 shows a placeholder comment where the actual generation of Tasks is placed.

```
acceptor.accept(
  element.toClass(element.className) [
    addSetUp(element, members)

    // continue with Task generation
  ]
)
```

Listing 4.4: Java Class Generation and Set Up

Task Creation The biggest part of the Java code generation is the creation of Tasks. The second scenario describes that every Task is compiled to a method of the previously created class. A code example with detailed description of how Tasks are translated is out of scope of this thesis. The steps of the Task translation are: (i) for every Task a new method with the provided name is created; (ii) Task methods have a `throws` clause for `java.lang.Throwable` types; (iii) a parameter of type `ProcessScope` is added to the signature; (iv) all defined input parameters (`in` and `javaIn`) are added to the signature; (v) the body of the Task method is specified.

In (iii) the DSL implementation adds an input parameter of type `ProcessScope`. This parameter is necessary that the hADL runtime state with all referenced Java objects can be passed between different Tasks. Otherwise objects might be removed by the Java garbage collector even though they are conceptually still needed.

The Tasks' method body is translated in the following steps into Java code: (i) check if `ProcessScope` input is provided. If not call the `setUp` method which returns a freshly initialised `ProcessScope` variable; (ii) hADL input parameter type checks. As the Java type system does not support hADL types, the generated code needs to check at runtime if the provided hADL input parameters match the specified hADL types of the DSL script; (iii) create the output map and add the `ProcessScope` variable and the failure list to it; and finally (iv) translate the statements (see 3.3) into their Java equivalents.

Algorithm 4.1 shows the above described DSL to Java translation as pseudo code.

4.1.4 Validation

Validation is the static analysis that ensures that a program of a language (e.g. the DSL) is syntactically and semantically correct. These checks produce errors if they fail, which are helpful to the programmer for writing valid programs. Xtext offers three different kinds of validation: (i) automatic validation that is performed due to grammar specification such as syntax and cross-references, (ii) custom validation that is added to the Xtext workflow file (`net.laaber.mt.GenerateDSL.mwe2`), and (iii) manual validators that are defined programmatically with Xtend.

Algorithm 4.1: DSL to Java Translation

```
1  if hADL model not loaded then
2      load hADL model
3  end
4  init variable manager
5  create Java class
6      add setUp func
7      foreach t of Tasks do
8          create method for t
9              addThrows Throwable
10             addParam ps : ProcessScope
11             add other parameters
12             addBody
13                 if ps == null then
14                     ps = setUp()
15                 end
16                 check hADL parameters for types
17                 add ps and failure list to output map
18                 foreach s of Statements do
19                     generate Java code
20                 end
21             end
22         end
23     end
24 end
```

This work's DSL makes use of automatic validation and manual validators. The following sections focus on the latter and describe them in more detail. Section 3.4 introduces validity and consistency checks that are applied automatically by the compiler. Those are divided into name and type checks.

VariableManager

This subsection describes how variable information including names and types (Java and hADL) is stored and accessed, before it describes how manual validations are defined. The VariableManager (`net.laaber.mt.VariableManager`) encapsulates that functionality. It is located in the main sub-project of the DSL-project. Listing 4.5 shows the interface that the VariableManager exposes. Line 1 and line 3 are the methods for initialising and resetting the VariableManager. Line 5 defines a method where the operational Java type of a hADL type is returned if it exists, and `Option.none()` otherwise. Lines 9 - 15 define methods that check if certain properties hold true for variables. Line 9 checks if a variable is deletable. Variables that are deletable are:

hADL-Operational-Elements and hADL-Operational-Connectives. The method on line 11 checks wether a variable `v` exist on the scope-level `container`. Line 13 defines the method that checks if a variable name already exists (with respect to the scope). The method on line 15 checks if the provided output name of `o` already exists within the Task it is specified in. Lines 7 adds a new variable with its Java type and hADL type. Line 17 adds a new output name. Finally line 19 retrieves all information on a provided variable if the variable exists and `Option.none()` otherwise. That information includes the variable's: (i) name, (ii) Java type, (iii) hADL type, and (iv) the scope it is defined in.

```
 1  def void setUp(Domainmodel model)
 2
 3  def void reset()
 4
 5  def Option<Class<? extends THADLarchElement>>
       operationalTypeForHadlInput(String hadlId)
 6
 7  def void addVariable(Var v, Clas<?> type, String hadlId)
 8
 9  def boolean deletable(Var v)
10
11  def boolean variableExists(Var v, EObject container)
12
13  def boolean variableNameExists(Var v)
14
15  def boolean outputKeyExists(Output o)
16
17  def void addOutputKey(Output o)
18
19  def Option<VariableInfo> variableInfo(Var v)
```

Listing 4.5: VariableManager's Interface

Checks

Xtext supports manual validators as methods of the class `net.laaber.mt.validation.DSLValidator` that are annotated with `@Check`. Both name and type checks are implemented with these methods. The method name can be arbitrary. Each method must have a single parameter, that corresponds to a subclass of `EObject`. These classes are automatically generated by Xtext for every parser rule. The Xtext runtime calls the validators for every occurrence where the specific parser rule is valid.

See Section 3.4 for a list of checks that are performed on the DSL program code. Name checks are performed by calling the VariableManager's methods `variableNameExists` and outputKeyExists. If the methods return false, a new variable or output name is added to the VariableManager by invoking the methods addVariable and addOutputKey.

hADL Type checks are performed by validators for References, Links and Loads. The corresponding methods in the DSLValidator are named refIsPossible, linkIsPossible and loadIsPossible. The basic idea for these three checks is, first to retrieve type information from VariableManager (variableInfo(v)) for all variables defined by the statements, and then check with the ModelTypesUtil if the hADL types can be referenced, linked, or loaded in the occurring definition.

The correct type of the SensorFactory is checked by the validator method sensorFactoryValid. ResourceDescriptor factory methods are checked for each occurrence in an acquire statement. The validator method is named checkResourceDescriptor.

4.1.5 Scopes

Scoping is strongly connected to cross-references introduced in the Section on grammar above (see 4.1.2). That section gives an example how cross-references are defined in the grammar. The referencing side (e.g. [Var]) is where scopes come into play. Through ScopeProviders the programmer can programmatically define which elements are valid as cross-reference.

An example is displayed in Listing 4.6 where the grammar definitions of the link and unlink statements are defined. In this example there are three cross-reference [Var]s defined (in square brackets) and one created Var. With a ScopeProvider we restrict the first [Var] on line 2 to a variable referencing an OperationalCollaborator, and the second [Var] on line 2 to a variable referencing an OperationalObject which is linked in the hADL model to the first [Var]. The [Var] on line 6 is restricted to a variable referencing an OperationalCollabLink. Such a variable can either be created with a link statement or passed to the Task as input parameter.

```
1 Link:
2   'link' collaborator = [Var] 'and' object = [Var] 'by'
        link = QualifiedName 'as' variable = Var
3 ;
4
5 Unlink:
6   'unlink' link = [Var]
7 ;
```

Listing 4.6: Scoping Example

The scope definitions are defined in Xtext in the class `net.laaber.mt.scoping.DSLScopeProvider`. For the Xtext version used in this project some adaptions are needed in order to get it working. The `DSLScopeProvider` has to extend `org.eclipse.xtext.xbase.scoping.batch.XbaseBatchScopeProvider`. Furthermore the `XBaseBatchScopeProvider` must be bound to the DSL-project's `DSLScopeProvider` in the class `net.laaber.mt.DSLRuntimeModule`.

Whenever a cross-reference occurs in a DSL script the method `getScope(EObject context, EReference reference)` of `DSLRuntimeModule` is invoked. That method returns an instance of `IScope`. Depending on the `context` and `reference` the ScopeProvider knows for which cross-reference the method was invoked. The implementation then dispatches the calls to methods of the form as seen in Listing 4.7. The naming convention is that it starts with "scope_" followed by the type of the context. The first parameter is the parser rule object and the second parameter is the reference to figure out the position of the cross-reference (compare to Listing 4.6 line 2).

```
def IScope scope_Link(Link l, EReference r)
```

Listing 4.7: ScopeProvider Method

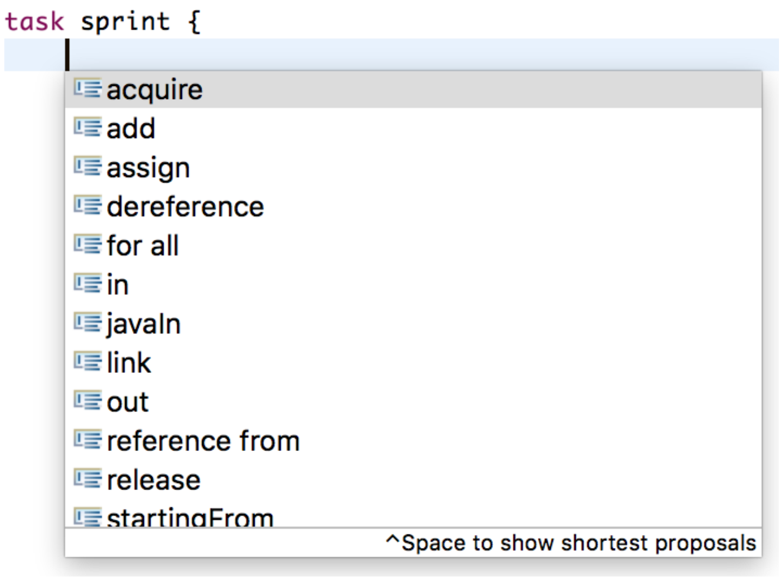
The implementation of the ScopeProvider methods relies on the utility class `net.laaber.mt.DslUtil` and the `ModelTypesUtil`. The `DslUtil` provides methods to conveniently retrieve variables by different criteria. Those include: (i) simple hADL type, (ii) hADL List-type, (iii) Java type, and (iv) desired scope. The `ModelTypesUtil` is used to perform checks that concern the specified hADL model. An example for such a check is given above where the cross-references of the `link` statement are described.

A nice side-effect of implementing scopes besides validity is that at the cross-reference positions the IDE provides content proposals. From a DSL script developers perspective scopes act the same as a combination of validation (see 4.1.4) and content proposals (see 4.1.6) do.

4.1.6 Content Proposals

Content proposals or otherwise known as auto-completions are the third and last form that support a DSL script developer at writing code. Validations are compile-time checks whether a program is valid, Scopes provide content proposals and do validity checks, and content proposals only provide suggestions.

As with validations Xtext comes with built in content proposals, where the developer of a language does not have to do anything in order to take advantage of those. These content proposals concern the syntax which is defined in the grammar file. At any cursor position in a DSL script file one can hit the key combination *CTRL + SPACE* (on OSX) to trigger a content proposal. If at that cursor position a proposal is possible based on the grammar, a context menu with all the options appears. Figure 4.1 shows a content proposal in Eclipse.



```
task sprint {
```

Figure 4.1: Content Proposal in Eclipse

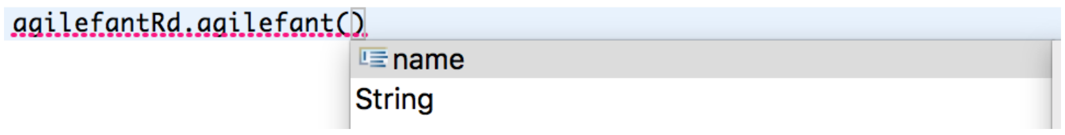
Custom content proposals in Xtext are part of the UI sub-project. The class `net.laaber.mt.ui.contentassist.DSLProposalProvider` defines the content proposals by overriding methods of a super class. The signature of these methods is shown in Listing 4.8. `{TypeName}` is exchanged with the type name of the parser rule the content proposal is for.

```
def override complete_{TypeName}(EObject model, RuleCall
    ruleCall, ContentAssistContext context,
   ICompletionProposalAcceptor acceptor)
```

Listing 4.8: Content Proposal Methods

This work's DSL supports content proposals for hADL types, ResourceDescriptor methods and output parameter names. The hADL type proposals are shown for statements `acquire`, `link`, `reference`, `in`, `var` and whenever a hADL List-type is inserted. The constraints which types are valid for each statement are defined in Sections 3.3 and 3.4.

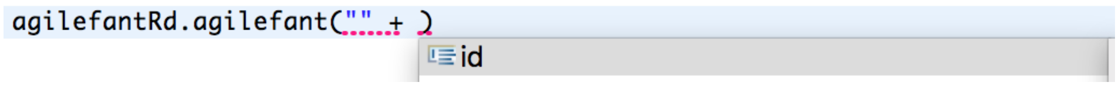
Section 3.3.1 describes how ResourceDescriptor factories are specified and used to create ResourceDescriptors. Content proposals play an important role concerning the usability. First only methods of the factory that return a `TResourceDescriptor` are suggested. Second the types of the parameters and possible Java input parameters are suggested



```
agilefantRd.agilefant()
```

name
String

(a) Simple Parameter



```
agilefantRd.agilefant(""+ )
```

id

(b) Appendable Parameter

Figure 4.2: Content Proposal ResourceDescriptor Parameter Types

(see Figure 4.2). Subfigure a suggests either the Java input parameter *name* or an inline String. The inline variant inserts two double quotes with the curser between those.

Output parameter names are suggested the same way as String parameters in ResourceDescriptor factory methods.

4.2 hADL Synchronous Library

This chapter already described how the syntax is defined through grammar, how Java code generation works, how constraints are checked through validators and scopes, how content is proposed (scopes and content proposals). This section discusses how hADL primitives (see 3.3.5) call the hADL runtime framework.

Throughout this thesis it illustrates a couple of Java code examples that include calls of the LinkageConnector and the RuntimeMonitor. The invocation of those method calls with the proper parameters and the handling of the return values is quite cumbersome. Hence the Java code generation would get more complex and congested. All the calls to the hADL runtime framework return instances of `rx.Observable<T>` as result. `rx.Observable<T>` is part of the RXJava library [36] which enforces an asynchronous programming model. The design of the DSL on the other hand is inherently synchronous as statements within a Task may rely on the output of a previous statement. The code generation for asynchronous code would be more complex than synchronous code.

In order to have an API that closely resembles the input of the DSL and to have synchronous calls into the hADL runtime framework, the implementation has a layer between the generated Java code and the hADL runtime framework. The library representing that layer is substance of the second project, the hADL-lib-project. Figure 4.3 shows an overview of this thesis' work. It shows how the hADL synchronous lib fits into the overall picture.

The hADL-lib-project defines its classes and interfaces in the package `net.laaber.mt.dsl.lib.hadl`. The methods of the contained interfaces and classes are designed to take input parameters that closely resemble the design of the DSL. The output parameters

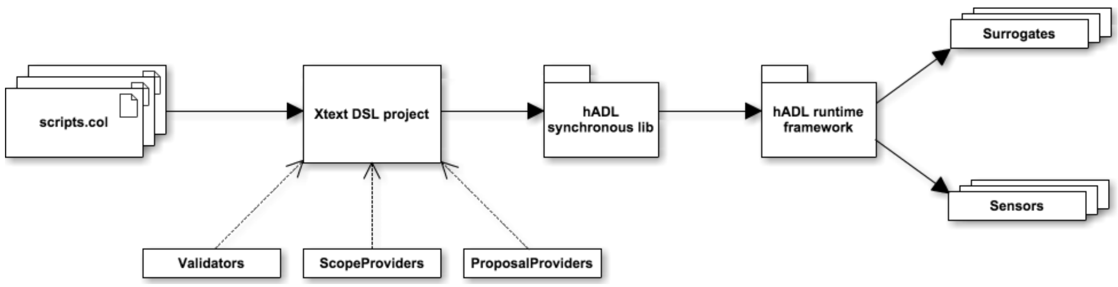


Figure 4.3: Overview of this Thesis' Work

are designed in a functional programming style. Two kinds of output parameters are used: (i) methods that have a return value return `Either<Throwable, O>`, and (ii) methods that have no return value return `Option<Throwable>`. The idea is that error handling is more explicit than doing it with raised exceptions and try-catch blocks. Hence the developer using the library must explicitly check the output of a method if it has returned an error (`Throwable`). The following subsections describe the contents of the package in detail. References to the language specification (see 3.3.5) are made that point out which parameters of the hADL primitives are used for which parameters of the library.

`HadlModelInteractor` and `HadlModelInteractorImpl`

`HadlModelInteractor` and `HadlModelInteractorImpl` are the interface and implementation for interacting with a hADL model file. The provided functionality includes loading a model, storing runtime and executable models, and retrieving a scope of a model by its ID.

`HadlLinker` and `HadlLinkerImpl`

`HadlLinker` and `HadlLinkerImpl` define the functionality which is offered by the LinkageConnector (`HADLLinkageConnector` of the hADL runtime framework). Listing 4.9 shows the methods that are available. Line 1 is the set up function which must be called once before any other method is called.

Acquire and Release Line 3 acquires a hADL-Elements. `archElem` is the hADL type (*`hadl_type`* in 3.14) and the ResourceDescriptor (*`rd`* in 3.14). Lines 4 - 6 release acquired hADL-Operational-Elements. The parameter to the `release` methods is the variable that is referenced in the `release` statement of the DSL (*`var_name`* in 3.14).

Reference and Dereference Line 8 creates an OperationalCollabReference between to OperationalCollaborators (`from` and `to`) with the hADL type `refId`. `from` and `to` must be either an OperationalComponent or an OperationalConnector. `from` relates to *`var1`*, `to` relates to *`var2`*, and `refId` is *`hadl_type`* in Listing 3.16. Line 9 is the dereference operation that removes an OperationalCollabRef (*var3* in 3.16). Lines 10 and 11 are the respective reference and dereference operations for OperationalObjects and OperationalObjectRefs.

Link and Unlink Line 13 creates an OperationalLink between an OperationalCollaborator (`collaborator`) and an OperationalObject (`object`) through a link (`linkId`). `collaborator` refers to *var1*, `object` refers to *var2*, and `linkId` refers to *hadl_type* in Listing 3.15. Line 14 is the unlink operation that removes an OperationalLink (*var3* in 3.15).

Stop and Start Surrogate scope Lines 16 stops (`stopScope` in 3.17) and line 17 starts (`startScope` in 3.17) the Surrogate scope.

Return Values `acquire`, `reference` and `link` have `Either` return vales. If the respective operation was successful the right part of the `Either` holds the reference to the resulting operational element. Which is than saved in the variable defined with the DSL (after the `as` keyword). If the operation was unsuccessful the error which occurred is returned as the left part of the `Either` return value. The other operations (`setUp`, `release`, `dereference`, `unlink`, `startScope` and `stopScope`) do not have a specific return value. If the operation was successful `Option.none()` is returned, otherwise the occurring error is returned.

```
1  Option<Throwable> setUp(String scopeId);
2
3  Either<Throwable, THADLarchElement> acquire(String
       archElem, TResourceDescriptor resourceDescriptor);
4  Option<Throwable> release(TOperationalComponent c);
5  Option<Throwable> release(TOperationalConnector c);
6  Option<Throwable> release(TOperationalObject o);
7
8  Either<Throwable, TOperationalCollabRef> reference(
       TCollaborator from, TCollaborator to, String refId);
9  Option<Throwable> dereference(TOperationalCollabRef ref)
       ;
10 Either<Throwable, TOperationalObjectRef> reference(
       TOperationalObject from, TOperationalObject to,
       String refId);
11 Option<Throwable> dereference(TOperationalObjectRef ref)
       ;
12
```

```
Either<Throwable, TOperationalCollabLink> link(
    TCollaborator collaborator, TOperationalObject object
    , String linkId);
Option<Throwable> unlink(TOperationalCollabLink link);

Option<Throwable> startScope();
Option<Throwable> stopScope();
```

Listing 4.9: HadlLinker Methods

`HadlMonitor` and `HadlMonitorImpl`

`HadlMonitor` and `HadlMonitorImpl` are the interface and implementation that wrap the functionality provided by the RuntimeMonitor (`HADLruntimeMonitor` from the hADL runtime framework). As already discussed in Section 3.3.5 the load operations from the RuntimeMonitor only supports directly connected elements and uses `SensingScopes` to control what elements are loaded from hADL-Operational-Element. The DSL's approach for loading is quite different, as it supports loading of indirectly connected elements (with `via` keyword) and only loading of one particular list of hADL-Elements (see Figure 3.4 and Listings 3.18 and 3.19). Listing 4.10 shows the library's methods for loading hADL-Operational-Elements.

There are 2 different kinds of operations: (i) the standard loads which just follow the path and load all the hADL-Operational-Elements that are found; and (ii) the loads which follow the path in the same way but exclude duplicate hADL-Operational-Elements. Duplicate hADL-Operational-Elements are possible when indirect connected elements are loaded. Recall the example from Figure 3.4. Suppose the first load (from *hadl_type* to *hadl_type1*) returns two distinct hADL-Operational-Elements. The second load (from *hadl_type1* to *hadl_type3*) returns one element for each of elements from the first load. It is possible that these two elements from the second load are the same hADL-Operational-Element (refer to the same entity on a collaboration platform). If this is desired behaviour than use the methods on lines 1, 3 and 5, if a mathematical set is desired use the methods on lines 7, 9 and 11. The input parameters of all six methods is equal.

The first parameter (`what`) is a tuple (`Map.Entry<String, String>`) where the left side denotes the type of the hADL-Element to load and the right side denotes the type of the hADL-Connective that leads to that element (after DSL's `load` keyword). The second parameter (`startingFrom`) is the hADL-Operational-Element (after DSL's `startingFrom` keyword) the load is started from. The last parameter (`fromVias`) is a list of tuples that represent the intermediary elements on the path of the load (after DSL's `via` keywords). Each tuple of the list is of the same form as the `what` parameter.

```
Either<Throwable, List<THADLarchElement>> load(Map.Entry
    <String, String> what, TOperationalComponent
    startingFrom, List<Map.Entry<String, String>>
    fromVias);

Either<Throwable, List<THADLarchElement>> load(Map.Entry
    <String, String> what, TOperationalConnector
    startingFrom, List<Map.Entry<String, String>>
    fromVias);

Either<Throwable, List<THADLarchElement>> load(Map.Entry
    <String, String> what, TOperationalObject
    startingFrom, List<Map.Entry<String, String>>
    fromVias);

default Either<Throwable, Set<THADLarchElement>>
    loadWithoutDuplicates(Map.Entry<String, String> what,
     TOperationalComponent startingFrom, List<Map.Entry<
    String, String>> fromVias) { ... }

default Either<Throwable, Set<THADLarchElement>>
    loadWithoutDuplicates(Map.Entry<String, String> what,
     TOperationalConnector startingFrom, List<Map.Entry<
    String, String>> fromVias) { ... }

default Either<Throwable, Set<THADLarchElement>>
    loadWithoutDuplicates(Map.Entry<String, String> what,
     TOperationalObject startingFrom, List<Map.Entry<
    String, String>> fromVias) { ... }
```

Listing 4.10: HadlMonitor Methods

`ProcessScope`

`ProcessScope` is the already introduced (see 4.1.3) data object that is passed to Tasks and returned from Tasks, which references the various hADL schema/runtime framework objects and objects of this library. Those include the `HADLmodel`, `HADLlinkageConnector`, `HADLruntimeMonitor`, `HadlLinker` and `HadlMonitor` besides others.

`SurrogateStateChangeFailure`

Section 3.5 describes how the DSL handles errors and introduces the two types of errors that can occur. The ones that are exceptional situations and the ones that occur if a Surrogate state change fails. The second type is represented by the class `SurrogateStateChangeFailure`. Listing 4.11 shows a simplified version of the generated Java code for dealing with errors from hADL primitives. Line 1 tries to acquire an element. Line 2 immediately checks if an error was returned. If an error was returned line 4 checks whether it is a `SurrogateStateChangeFailure`, if so it returns normally from the Task with the error added to the failureList. Any other error triggers a return from the Task by throwing that error.

```
 1 Either<Throwable, THADLarchElement> res = linker.acquire
       (archElem, rd);
 2 if (res.isLeft()) {
 3   Throwable err = res.left().value();
 4   if (err instanceof SurrogateStateChangeFailure) {
 5       failureList.add((SurrogateStateChangeFailure) err)
      ;
 6       return res;
 7     } else {
 8       throw err;
 9     }
10 }
```

Listing 4.11: Generated Java Error Handling Code

CHAPTER 5

Evaluation

The previous two chapters describe how the DSL is designed, of which syntactical elements it is comprised of, what behaviour these elements have (see 3), and how these features are implemented (see 4). This chapter is about evaluating the results of the work that has been done so far. Recall the overview of this work (see Figure 4.3): Chapters 3 and 4 describe the elements of Xtext DSL project (with its Validators, ScopeProviders and ProposalProviders) and the hADL synchronous lib. The hADL runtime framework is developed by Christoph Mayr-Dorn with some small additions from us that came up during this work. That leaves the very left elements from the overview, the actual scripts written in the DSL, and the very right elements that wrap access to the collaboration platforms, the Surrogates and Sensors. In order to evaluate the DSL this chapter presents a use case for it, a corresponding DSL script which defines the collaborations of the use case, the Surrogates and Sensors, and a simple Java program that brings together the Tasks.

The two aspects that are investigated as part of the evaluation concern the benefits a developer has when using the DSL over the hADL runtime framework directly from Java. Section 3.2 introduces the main aspects to use a DSL: (i) valid and consistent hADL instance modeling, and (ii) hADL client code generation. Those aspects are central to the following evaluation.

Section 5.1 introduces the use case used to evaluate the DSL. Section 5.2 compares the DSL version vs. the equivalent Java version by looking at the lines of code that are necessary to achieve the same result. Finally Section 5.3 shows ways how a developer can break the DSL despite all the checks.

5.1 Use Case

This section introduces the use case chosen for evaluating the DSL. First it describes the scenario. Second it depicts the collaboration platforms utilised to support the scenario. Then it describes the implementation of the use case. There are three parts that complete the implementation: the DSL script, the Surrogates and the Sensors.

5.1.1 Scenario

The scenario is defined in two steps. On one hand there is the idea what would be a good scenario to test the capabilities of the DSL. On the other hand there is the hADL model that supports the collaborations of the scenario.

Idea Typical workflows often used in software engineering are agile software development methods. Arguably the most popular method today is Scrum. A full implementation of a Scrum process as well as a detailed introduction to Scrum, as described in [47], is out of scope of this thesis.

In Scrum a product backlog is maintained, which is a prioritised list of features to develop. The actual work is divided into fixed time slots (in a sprint planning meeting), called iterations or sprints, where a group of developers implement the assigned stories or tasks. At the end of each iteration the developers present the implemented features to the rest of the group including the product owner and the Scrum master. This meeting is called the sprint review. After the sprint review another meeting is held, the sprint retrospective where the team discusses what is going well and what is not concerning the Scrum process. After that a new sprint starts with a sprint planning meeting. Additionally every day daily stand-up meetings take place, where every developer presents in a couple of words what he or she has been doing since the last daily stand-up and what the next steps are.

This evaluation scenario contains the set up and tear down of two concrete collaborations within the Scrum process. (i) create chat rooms for developers assigned to a particular story of a sprint at the beginning of the sprint execution phase and remove the chats at the end of the execution phase, and (ii) provide the collaboration infrastructure for a sprint retrospective meeting for all sprint participants, which includes a chat room and a wiki page.

hADL Model Figure 5.1 shows the hADL model required for implementing the scenario. The model is designed with two collaboration platforms in mind. The first one offers functionality for organising Scrum processes. The second one provides communication and collaboration mechanisms for software development teams. These two platforms and the features used for the scenario are described in Section 5.1.2.

The model consists of four CollaborationObjects representing a *Sprint*, a *Story*, a *Chat* and a *Wiki*, and two HumanComponents namely *AgileUser* and *DevUser*. The *Sprint*

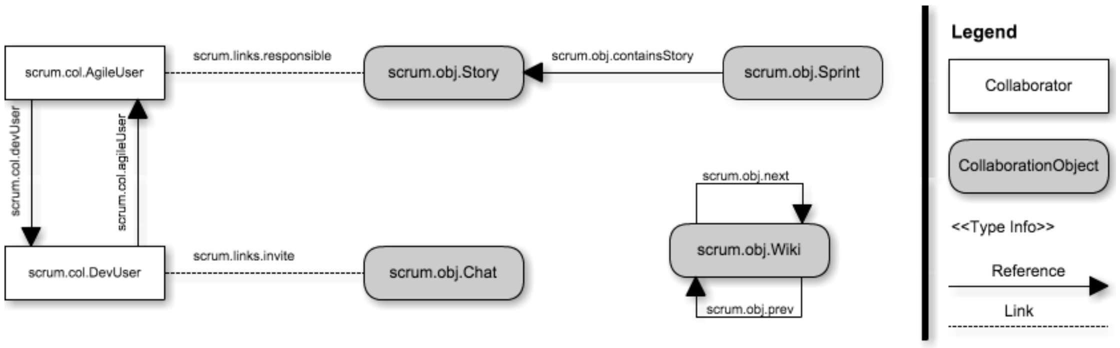


Figure 5.1: Scrum hADL Model

references *Story* with the ObjectReference *containsStory*. This reference is a one-to-many reference indicating that a *Sprint* can contain many *Stories*. A *Story* in turn is linked with *AgileUser* by the Link *responsible*. This Link has many-to-many semantics meaning that a *Story* can have many *responsible AgileUsers* assigned. So can a single *AgileUser* be *responsible* for many *Stories*. Every *AgileUser* is related to a single *DevUser* (CollaboratorReference *devUser*) and otherwise round (CollaboratorReference *agileUser*). *DevUser* may be linked to *Chat* with a many-to-many semantics. The Link *invite* indicates that a *DevUser* is invited to a *Chat*. The last CollaborationObject is *Wiki* which has two ObjectReferences to the *previous* and *next* wiki page. The hADL types of the hADL-Elements are depicted in the center of the objects.

For the XML representation of the hADL model visualised in Figure 5.1 see Appendix Scrum hADL Model.

5.1.2 Collaboration Platforms

In the previous section (see 5.1.1) it is mentioned that the hADL model is designed with the collaboration platforms in mind. The scenario requires two functionalities: Scrum process management and collaboration mechanisms for developers. An ideal collaboration platform would support both functionalities and expose an API to those. Such a platform does not exist hence the functionalities are split over multiple platforms. Agilefant is the platform of choice for handling the Scrum related tasks. Bitbucket offers an API for wiki pages, and HipChat provides communication channels.

Agilefant

Agilefant is a platform that offers the functionality to manage software development through Scrum. It provides a web interface and a Representational State Transfer (REST)-based API to access its functionality. The resources of interest to us are backlogs, stories, and users. Backlogs are either products, projects or iterations. Table 5.1 shows the resources (URLs and Hypertext Transfer Protocol (HTTP) verbs) of the Agilefant API that are of interest to us and their resulting hADL-Element types. The base URL of

URL	HTTP verb	hADL-Elements types
`backlogs/`	GET	scrum.obj.Sprint
`backlogs/{id}/stories`	GET	List of scrum.obj.Story
`stories/{id}`	GET	scrum.obj.Story
`users/{id}`	GET	scrum.obj. AgileUser

Table 5.1: Agilefant API Resources and Corresponding hADL-Element Types

the Agilefant API is `https://cloud.agilefant.com/<accountname>/api/v1/`, where `<accountname>` is substituted with the actual account name. The `id` parameter of the URLs is substituted by the ID of the desired entity.

Dorn [48] implemented a Java abstraction of the Agilefant REST API which is used to implement the Agilefant Sensors (see 5.1.4).

Bitbucket

Bitbucket is a source code hosting platform with basic means for collaboration. The service run and developed by Atlassian supports users, teams, code repositories, wiki pages, issues and comments for most of those. For the evaluation scenario the user management (`scrum.col.DevUser`) and wiki pages (`scrum.obj.Wiki`) are of interest. Bitbucket exposes two versions of its REST API which are both needed for the scenario. Version 1 exposes the wiki resource, which is used for creating and updating wiki pages. Version 2 provides access to the user resource. The base url to both APIs is `https://api.bitbucket.org/{version}/` where `version` is replaced by `1.0` for version 1 and by `2.0` for version 2. Version 1 uses basic HTTP authentication and version 2 uses OAuth 2.

The REST client for Bitbucket is implemented in the project Atlassian4Hadl [49].

HipChat

HipChat is a communication platform by Atlassian. It offers user management, chat rooms, direct message, groups, notifications and sharing of documents. For the use case scenario the user management (`scrum.col.DevUser`) feature and chats (`scrum.obj.Chat`) are of interest. HipChat offers two API versions, but only version 2 is recommended for usage. The base url is `https://api.hipchat.com/v2/` and OAuth 2 is used for authentication.

The REST client for HipChat is implemented in the project Atlassian4Hadl [49].

5.1.3 Script Implementation

Previous sections introduce the scenario, the corresponding hADL model and the collaboration platforms that support the use case. In order to evaluate the DSL by means of the scenario, an implementation of the scenario with the DSL and a script that calls the created Java code are necessary. The following sections introduce the script in detail which is defined in the project MTEvaluationDSL [50]. Appendix Scrum DSL Script shows the DSL script in full.

Set Up Section

Code Segment 5.1 shows the set up section of the DSL script. The hADL model file can be seen in Appendix Scrum hADL Model and the generated Java file `net.laaber.mt.evaluation.Tasks` (line 3) can be seen in the Evaluation project [51]. Lines 4 and 5 define the ResourceDescriptor factories exposed by the Agilefant4Hadl [52] and Atlassian4Hadl [49] projects. The SensorFactory defined on line 6 is part of the Evaluation project [51], which furthermore contains the overall evaluation process.

```
1 hADL "/Users/Christoph/TU/Diplomarbeit/Code/Evaluation/
     src/main/resources/agile-hadl.xml"
2
3 className net.laaber.mt.evaluation.Tasks
4 resourceDescriptorFactory agilefantRd class net.laaber.
     mt.hadl.agilefant.resourceDescriptor.
     AgilefantResourceDescriptorFactory
5 resourceDescriptorFactory atlassianRd class net.laaber.
     mt.hadl.atlassian.resourceDescriptor.
     AtlassianResourceDescriptorFactory
6 sensorFactory net.laaber.mt.evaluation.sensor.
     DslSensorFactory
```

Listing 5.1: Set Up Section of Evaluation Script

Sprint Acquirement

The first Task defined in the script (see Code Segment 5.2) `sprint` returns a `scrum.obj.Sprint` object. The Java input parameters defined on lines 2 and 3 serve as input parameters to the ResourceDescriptor factory method on line 5. Because all Scrum elements are represented on Agilefant, the acquirement of a `scrum.obj.Sprint` needs an Agilefant ResourceDescriptor.

```
1 task sprint {
2   javaIn id : Integer
3   javaIn name : String
4
5    acquire scrum.obj.Sprint with agilefantRd.agilefant(
        name, id) as s
6
7    out s as "sprint"
8  }
```

Listing 5.2: `sprint` Task of Evaluation Script

Story Chats Creation

The Task `createChatsForStoriesOfSprint` in Code Segment 5.3 creates a chat for each story of a sprint and invites all responsible developers of the story to the created chat. The only input of this Task is the sprint for which the chats should be created (`s` parameter on line 2). Line 4 stops the Surrogate scope indicating that changes to hADL-Operational-Elements are about to be made. As the first step all stories of the sprint (input parameter `s`) are loaded on line 6 and saved to the variable `stories`. Lines 8 and 9 define hADL List-type variables referencing created chats and invites.

Lines 10 to 19 define the iterations for creating the chats and inviting users to it. The iteration started on line loops over the previously loaded `stories`. Line 11 creates a new chat for each story using the `counter` variable (see 3.3.7). Line 12 adds the created chat to the variable from line 8. The next steps are about inviting the responsible users of a story to the newly created chat. On line 14 all `scrum.col.DevUsers` are loaded from the `story` via the `links.responsible` Link and `scrum.col.devUser` Reference. Lines 15 to 19 define the iteration that invites the loaded `users` to the created chat (variable `c`) (line 16) and adds the invitation (line 17) to the variable defined on line 9.

Finally the changes made to the Surrogates are applied by starting the Surrogate scope (line 21) and the variables `chats` and `invites` are returned from the Task (lines 23 and 24).

```
1  task createChatsForStoriesOfSprint {
2    in s : scrum.obj.Sprint
3
4    stopScope
5
6    startingFrom s load scrum.obj.Story:scrum.obj.
        containsStory as stories
7
8    var chats : [scrum.obj.Chat]
9    var invites : [scrum.links.invite]
10   for all stories as story counter i {
11     acquire scrum.obj.Chat with atlassianRd.hipChat("
          StoryChat" + i, "StoryChat" + i) as c
```

```
    add c to chats

    startingFrom story via scrum.col.AgileUser:scrum.
       links.responsible load scrum.col.DevUser:scrum.
       col.devUser as users
    for all users as u {
      link u and c by scrum.links.invite as invite
      add invite to invites
    }
  }

  startScope

  out chats as "chats"
  out invites as "invites"
}
```

Listing 5.3: Creation of Story Chats of Evaluation Script

Story Chats Deletion

The next Task `releaseChatsAndInvites` (see Code Segment 5.4) removes the collaborations created in Task `createChatsForStoriesOfSprint`. Therefore the hADL input parameters on lines 2 and 3 correspond to the output parameters of Task `createChatsForStoriesOfSprint`. Lines 7 to 9 remove the invites to the chat before lines 13 to 15 the chats are removed.

```
 1 task releaseChatsAndInvites {
 2   in chats : [scrum.obj.Chat]
 3   in invites : [scrum.links.invite]
 4
 5   stopScope
 6
 7   for all invites as i {
 8     unlink i
 9   }
10
11   startScope
12
13   for all chats as c {
14     release c
15   }
16 }
```

Listing 5.4: Deletion of Story Chats of Evaluation Script

Sprint Retrospective Set Up

The set up of the sprint retrospective meeting's collaborations is split over three Code Segments 5.5, 5.6 and 5.7. Code Segment 5.5 defines the Task `sprintRetrospectiveSetUp` with one hADL input parameter on line 2 and two Java input parameters specifying the current sprint number (line 3) and the number of the previous sprint (line 4). Lines 6 to 13 acquire the wikis of the current sprint (line 8) and the previous sprint's wiki (line 9). Lines 10 and 11 establish the previous (`scrum.obj.prev`) Reference and the next (`scrum.obj.next`) Reference between the two wikis. Line 13 applies the changes made on lines 10 and 11.

```
 1 task sprintRetrospectiveSetUp {
 2   in sprint : scrum.obj.Sprint
 3   javaIn sprintNumber : Integer
 4   javaIn previousSprintNumber : Integer
 5
 6   stopScope
 7
 8   acquire scrum.obj.Wiki with atlassianRd.bitbucket("
        SprintWiki" + sprintNumber, "SprintWiki" +
        sprintNumber) as currentWiki
 9   acquire scrum.obj.Wiki with atlassianRd.bitbucket("
        SprintWiki" + previousSprintNumber, "SprintWiki" +
        previousSprintNumber) as previousWiki
10   reference from currentWiki to previousWiki with scrum.
        obj.prev as p1
11   reference from previousWiki to currentWiki with scrum.
        obj.next as p2
12
13   startScope
```

Listing 5.5: Set Up of Sprint Retrospective (1/3)

Code Segment 5.6 continues the definition of the Task. After the establishment of the References between the two wikis a chat for the sprint retrospective meeting is created on line 1. Line 2 loads all the `scrum.col.DevUsers` that have task responsibilities in the current `sprint` as `users`. The variable `invites` (line 6) holds all the invites to the previous created chat (`chat`). Lines 7 to 10 create the invitations to the chat (line

8) and adds the invites to the variable `invites` (line 9). Lines 4 and 12 again control the Surrogate scope.

```
 1   acquire scrum.obj.Chat with atlassianRd.hipChat("
        RetrospectiveChat" + sprintNumber, "
        RetrospectiveChat" + sprintNumber) as chat
 2   startingFrom sprint via scrum.obj.Story:scrum.obj.
        containsStory via scrum.col.AgileUser:scrum.links.
        responsible load scrum.col.DevUser:scrum.col.
        devUser as users
 3
 4   stopScope
 5
 6   var invites : [scrum.links.invite]
 7   for all users as u {
 8     link u and chat by scrum.links.invite as i
 9     add i to invites
10   }
11
12   startScope
```

Listing 5.6: Set Up of Sprint Retrospective (2/3)

The definition of Task `releaseChatsAndInvites` concludes in Code Segment 5.7 by releasing the variable for the previous wiki and returning the created chat (`chat` on line 3), the wiki of the current sprint (line 4) and the invites (line 5) to the caller of the Task.

```
1   release previousWiki
2
3   out chat as "chat"
4   out currentWiki as "wiki"
5   out invites as "invites"
6 }
```

Listing 5.7: Set Up of Sprint Retrospective (3/3)

Sprint Retrospective Tear Down

The last Task of the evaluation script tears down the sprint retrospective collaborations (see Code Segment 5.8). It takes the chat hADL as input and the invites to that chat (lines 2 and 3). Lines 7 to 9 removes the invitations by iterating over the list and unlinking every element. On line 13 the chat is released.

```
task sprintRetrospectiveTearDown {
  in chat : scrum.obj.Chat
  in invites : [scrum.links.invite]

  stopScope

  for all invites as i {
    unlink i
  }

  startScope

  release chat
}
```

Listing 5.8: Tear Down of Sprint Retrospective

Overall Process

The previous sections describe the DSL script in detail. This section describes the overall process as defined by the class `net.laaber.mt.evaluation.Evaluation`. It brings together the generated Tasks from the DSL script in a basic process-like fashion. It is part of the Evaluation project [51]. The Tasks are called by the process in the same order as they are defined in the DSL script: `sprint`, `createChatsForStoriesOfSprint`, `releaseChatsAndInvites`, `sprintRetrospectiveSetUp` and `sprintRetrospectiveTearDown`. In addition after every Task invocation the returned `Map` is checked if failures occurred and the output parameters are extracted from the `Map`. Furthermore outputs from previous Tasks are used as inputs for successive Tasks. A complex failure handling or even an implementation with a process engine remains out of scope of this thesis.

5.1.4 Surrogate and Sensor Implementations

The very left elements of Figure 4.3 are described in the previous section (see 5.1.3). This only leaves the very right elements open for discussion. This section describes the implementation of the Surrogates and the Sensors required for the evaluation scenario. Section 2.5 introduces the concepts of Surrogates and Sensors. Recall the Scrum hADL model (Figure 5.1) with the scenario (see Sections 5.1.1 and 5.1.3) in mind, we conclude that for the hADL-Elements the Surrogate and Sensor implementations in Table 5.2 are necessary. Table 5.2 furthermore shows the project the Surrogates and Sensors implementations are part of. hADL-Elements which do not need a specific Surrogate have either `at.ac.tuwien.dsg.hadl.framework.runtime.utils.`

hADL-Element	Surrogate	Sensor	Project
`scrum.obj.Sprint`	no	yes	Agilefant4Hadl [52]
`scrum.obj.Story`	no	yes	Agilefant4Hadl
`scrum.col.AgileUser`	no	yes	Agilefant4Hadl
`scrum.col.DevUser`	yes	no	Atlassian4Hadl [49]
`scrum.obj.Chat`	yes	no	Atlassian4Hadl
`scrum.obj.Wiki`	yes	yes	Atlassian4Hadl

Table 5.2: hADL-Element Surrogate and Sensor Implementations

`DefaultAcceptingObjectSurrogate` or `at.ac.tuwien.dsg.hadl.framework.runtime.utils.DefaultAcceptingCollaboratorSurrogate` in the hADL model file specified (see Appendix Scrum hADL Model). For hADL-Elements that do not need a specific Sensor, the SensorFactory does not return a Sensor (`null`). Sensors are created at the time when they are needed.

Surrogates

This subsection textually describes what functionality the surrogates implement. An in depth description of how to implement Surrogates is out of this thesis' scope. In addition to the tasks described below, every Surrogate handles its own state transitions and the state transitions of the corresponding hADL-Operational.

The `scrum.obj.DevUser` Surrogate combines the access to both Bitbucket and HipChat. Therefore a composition ResourceDescriptor is needed. A composition ResourceDescriptor combines multiple ResourceDescriptors into one. This is necessary as the hADL LinkageConnector does not support acquiring hADL-Elements with multiple ResourceDescriptors. This composition ResourceDescriptor is represented by the class `net.laaber.mt.hadl.atlassian.resourceDescriptor.TAtlassianResourceDescriptor`. It consists of ResourceDescriptors for Bitbucket (`TBitbucketResourceDescriptor`) and HipChat (`THipChatResourceDescriptor`) which are both located in the same package. The Surrogate only checks on acquire if the provided ResourceDescriptor is valid for the collaboration platforms. If not, acquiring fails. The Surrogate is represented by the class `net.laaber.mt.hadl.atlassian.surrogate.UserSurrogate`.

The `scrum.obj.Chat` Surrogate has more tasks to perform compared to the previous Surrogate. On acquire it checks wether a chat for the provided ResourceDescriptor already exists on the collaboration platform. If existing, it just loads the data, otherwise it creates a new chat. Furthermore it implements the invitation feature, where new users are invited and removed from existing chats. The Surrogate is represented by the class `net.laaber.mt.hadl.atlassian.surrogate.ChatSurrogate`.

The `scrum.obj.Wiki` Surrogate loads existing wiki pages or creates new ones otherwise. The `scrum.obj.next` and `scrum.obj.prev` References are handled by this Surrogate. Bitbucket does not support saving additional data to wiki pages, hence the information about the previous and next wiki page is stored as first line of a wiki page's content. The Surrogate is represented by the class `net.laaber.mt.hadl.atlassian.surrogate.WikiSurrogate`.

Sensors

Same as with the Surrogates, the Sensors are just textually described. Sensors access the collaboration platforms to retrieve the elements that can be loaded from a hADL-Operational-Element via a particular Link or Reference. From the retrieved data about the elements a Sensor creates the corresponding ResourceDescriptors. The hADL runtime framework in turn intercepts the load and creates the hADL-Operational-Elements for the ResourceDescriptors.

The `scrum.obj.Sprint` Sensor loads the stories which are part of a sprint. It is represented by the class `net.laaber.mt.hadl.agilefant.sensor.SprintSensor`.

The `scrum.obj.Story` Sensor loads all responsible `scrum.col.AgileUsers` of a story. It is represented by the class `net.laaber.mt.hadl.agilefant.sensor.StorySensor`.

The `scrum.col.AgileUser` Sensor loads a single `scrum.col.DevUser` that represents the same physical person. The thesis introduces the concept of composition ResourceDescriptors to deal with multiple entities that are represented by a single hADL-Operational-Element. This is the second approach to deal with that case by creating different hADL-Elements, referring to each other and make some assumption how these two elements are equivalent. This use case assumes that a `scrum.col.AgileUser` and a `scrum.col.DevUser` have the same Email address and therefore are uniquely identifiable on all three collaboration platforms used. This Sensor is represented by the class `net.laaber.mt.hadl.agilefant.sensor.AgileUserSensor`.

The `scrum.obj.Wiki` Sensor reads the first line of its own wiki page, retrieves the information about previous and next wiki page and depending on the Reference to follow creates the corresponding ResourceDescriptor. The Sensor is represented by the class `net.laaber.mt.hadl.atlassian.sensor.WikiSensor`.

5.1.5 Prerequisites

The subsections of Section 5.1 describe how the evaluation use case is implemented. The following prerequisites concerning the collaboration platforms are necessary for the evaluation program to succeed.

Set Up Properties

The Agilefant REST client and Sensors as well as the Atlassian REST clients, Surrogates and Sensors need set up in order to work properly. Listing 5.9 shows the XML property file necessary for the Agilefant4Hadl project [52] to work. Line 4 specifies the account name and lines 5 and 6 specify the login credentials.

```
1 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2 <!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/
     properties.dtd">
3 <properties>
4   <entry key="account">...</entry>
5   <entry key="admin.pw">...</entry>
6   <entry key="admin.login">...</entry>
7 </properties>
```

Listing 5.9: Agilefant4Hadl Project Properties

Listing 5.10 shows the properties required for the Atlassian4Hadl project [49] to work. Lines 4 and 5 specify the Bitbucket OAuth 2 credentials required for the Bitbucket API version 2. Lines 6 and 7 are the login credentials needed for the Bitbucket API version 1. Lines 8 and 9 define the team name and the repository name in which the wiki pages are created. Only users that are members of the team have access to the wiki pages. Line 10 specifies the OAuth 2 access token for HipChat. On line 11 and 12 are the Gmail user name and password for sending Emails. Some features are not exposed through an API, therefore an Email is sent for those to manually add the requested feature via the collaboration platform.

Required Entities

A couple of entities have to be present on the collaboration platforms for the evaluation program to succeed. The following paragraphs present the required entities by collaboration platform grouped.

Agilefant First a product *MT_Evaluation* must be created. Within this product a project named *MT_Evaluation_Product* must exist, which itself contains an iteration named *Sprint1*. *Sprint1* has three stories defined: *Story1*, *Story2* and *Story3*. *Story1* has one responsible user *hadltestuser1*. *Story2* has two responsible users, *hadltestuser1* and *hadltestuser3*. *Story3* has a single responsible user *hadltestuser2*.

Bitbucket must have a team created with the same name as specified as `bb.teamname` in the Atlassian4Hadl project properties (see Listing 5.10). That team has a single repository named *MT_Evaluation*. The repository has three users in the Developer

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/
    properties.dtd">
<properties>
  <entry key="bb.apiKey">...</entry>
  <entry key="bb.apiSecret">...</entry>
  <entry key="bb.admin.login">...</entry>
  <entry key="bb.admin.password">...</entry>
  <entry key="bb.teamname">...</entry>
  <entry key="bb.repository">...</entry>
  <entry key="hc.accessToken">...</entry>
  <entry key="gmail.user">...</entry>
  <entry key="gmail.pwd">...</entry>
</properties>
```

Listing 5.10: Atlassian4Hadl Project Properties

group, *hadltestuser1*, *hadltestuser2* and *hadltestuser3*. These users must have the same Email addresses as the Agilefant users. The only other requirement is, that after each evaluation script execution the created wiki pages are manually deleted from Bitbucket.

HipChat only needs three users added to a team. The team name does not matter. The users must have the same Email addresses as the ones on Agilefant and Bitbucket.

5.2 DSL vs. Java

The previous section (see 5.1) introduces the use case scenario which is the basis for the evaluation of the DSL. This section is about the comparison between the use case implemented with the DSL and the Java code that would be necessary to write when using the hADL framework [33] instead. Furthermore it investigates how many mental checks a developer would have to make in order to achieve the consistency provided by the DSL. The sections present on a per Task basis (i) the effort a developer saves when using the DSL by comparing the DSL LOC versus the generated Java LOC; and (ii) the consistency checks the DSL takes care of. The Java code generation is measured in LOC (empty lines and comments excluded) and the consistency checks are measured by the amount of checks triggered. Calls to the hADL synchronous library are measured as if the library calls were just regular function calls. Only consistency checks that provide added benefits for the developer are counted. The checks that are performed to provide a Java like experience (variable name checks, iterations, uniqueness constraints, lexical scopes) are not measured. Surrogates and Sensors (see 5.1.4) are excluded from the evaluation as the work for them is identical in both situations, using the DSL or Java with the hADL runtime framework.

5.2.1 Code Effort Reduction

This section depicts the effort reduction in terms of LOC between DSL and Java. First the fixed effort reduction in form of called library code (hADL-lib) and second the actual reduction due to hADL client code generation is introduced.

Fixed Effort Reduction

This subsection illustrates the fixed reduction of effort a developer has when defining collaborations with the DSL, before it presents the effort a developer saves per Task. This reduction is called fixed because it does not depend on the size of the DSL script. Table 5.3 shows the lines of code per hADL primitive (see 3.3.5) the hADL synchronous library [39] is accountable for. In total the hADL synchronous library decreases the effort for the developer by 511 LOC.

hADL primitive	LOC
Acquire	45
Release	35
Reference	43
Dereference	39
Link	61
Unlink	37
Stop Surrogate Scope	35
Start Surrogate Scope	35
Load	181
Total	511

Table 5.3: hADL Primitive to hADL Runtime Framework through hADL Synchronous Library

Code Generation Reduction

This section explains the effort reduction due to hADL client Java code generation. The data in Table 5.4 is provided on a per task basis. The first column shows the names of the Tasks (or for the first row the set up section). The second row shows the LOC that are required to implement the use case scenario with the DSL. Column 3 lists the LOC produced by the Java code generator. The first row of the table (apart from the heading) shows the set up section (see 5.1.3) of the hADL runtime framework which takes only 5 LOC with the DSL compared to 151 LOC in Java. This is mostly due to DI set up code that is required for the hADL runtime framework. The next five rows show the comparison between the DSL LOC and the Java LOC of the tasks of the evaluation scenario: *Sprint Acquirement*, *Story Chat Creation*, *Story Chats Deletion*, *Sprint Retrospective Set Up*, *Sprint Retrospective Tear Down* described in Section 5.1.3. The second to the last row (*Total*) shows the added numbers of DSL LOC and Java LOC. In the last row the fixed effort reduction is added to the overall Java LOC.

Task	DSL [LOC]	Java [LOC]
Set Up Section	5	151
Sprint Acquirement	6	23
Story Chat Creation	19	98
Story Chats Deletion	12	71
Sprint Retrospective Set Up	24	147
Sprint Retrospective Tear Down	10	63
Total	76	553
Total + hADL-lib	76	1064

Table 5.4: Code Generation Effort Reduction

From the data of Table 5.4 we conclude, that the effort reduction concerning LOC decreases in the evaluation scenario by a factor of approximately 7.3. This factor does not include the fixed effort reduction. The LOC of the fixed effort reduction is constant with respect to the DSL script size. Hence the fixed effort reduction is negligible for the factor.

Effort Reduction Factor: 7.3

5.2.2 Mental Check Effort Reduction

While writing a hADL client in Java, a programmer has to perform numerous mental checks in order to write valid code. These checks concern the consistency of hADL clients with respect to correct hADL types used with hADL primitives (see 3.4). When specifying collaboration instances with the DSL, these checks are automatically performed by the DSL compiler. This section highlights how many mental checks a programmer would have to do in the evaluation scenario, if the hADL client was written in Java and not with the DSL.

Xtext provides three mechanisms to do consistency checks: (i) Content Proposals (see 4.1.6) provide the developer with choices that are valid at a particular position in the DSL script, (ii) Validators (see 4.1.4) are compile-time checks whether a DSL statement is valid in its entirety, and (iii) Scopes (see 4.1.5) provide suggestions like Content Proposals and additionally do compile-time validity checks like Validators, but only for a single position within a DSL statement. All three checks plus the built-in checks that are deduced from the grammar guarantee that a DSL script produces valid hADL instances. Table 5.5 shows how many consistency checks per DSL element are performed.

DSL Language Element	Content Proposals	Validators	Scopes
Acquire	1	1	0
Release	0	0	1
Reference	1	1	2
Dereference	0	0	1
Link	1	1	2
Unlink	0	0	1
Load	1	1	1
Load (additional per via)	1	0	0
hADL Input, hADL Variable	1	0	0
Assign	0	0	2
Add	0	0	2
Iteration	0	0	1
Output	0	0	1

Table 5.5: Consistency Checks per DSL Language Element

Based on Table 5.5 we calculate the mental effort reduction on a per task basis of the evaluation scenario as shown in Table 5.6. From the figures in the table we conclude that a significant amount of mental work is required from the programmer when writing a hADL client in Java. For this work's evaluation scenario the DSL performs 70 individual consistency checks. Hence defining collaboration instances with the DSL is easier compared to Java.

Task	Content Proposals	Validators	Scopes	Total
Sprint Acquirement	1	1	1	3
Story Chat Creation	8	4	12	24
Story Chats Deletion	2	0	4	6
Sprint Retrospective Set Up	11	7	14	32
Sprint Retrospective Tear Down	2	0	3	5
Total	24	12	34	70

Table 5.6: Consistency Checks per Task

Performed Validity/Consistency Checks: 70

5.3 Ways to break the DSL

In the first two section of the evaluation chapter the thesis introduces a scenario, the implementation with the DSL and the evaluation in terms of effort reduction. Validity and consistency checks ensure that a DSL script is valid. But there are ways to break the DSL or the execution despite those checks. This section is about these situations, which can be divided into three areas: (i) create a malicious DSL script, (ii) implement incorrect Surrogates and Sensors, and (iii) interference during runtime.

5.3.1 DSL script

The Content Proposals, Validators and Scopes, in combination with the checks automatically performed due to the grammar, check many cases in order to get a valid program. There are still some assumptions the implementation makes, that must hold for a valid program.

Incorrect hADL model file The DSL checks if a file at the specified location (after `hADL` keyword) is readable and therefore exists. The DSL does not check whether the provided file is a valid hADL model file. If an invalid file is provided the `DSLJvmModelInferrer` (see 4.1.3) just throws an exception if the `HadlModelInteractor` (see 4.2) does not return an instance of `HADLmodel`.

Invalid ResourceDescriptor factory After the keyword `resourceDescriptorFactory` a ResourceDescriptor factory can be provided. In the hADL project exists no notion of ResourceDescriptor factory. The hADL synchronous library also does not specify an interface or class that defines a ResourceDescriptor factory. At the time of specification of a ResourceDescriptor factory in the DSL script are no checks about the class name provided. Later when a ResourceDescriptor is created (when acquiring an hADL-Element) the used ResourceDescriptor factory's static methods are checked if they return an instance of `TResourceDescriptor`. Therefore the implementation assumes that the factory provided after the keyword `resourceDescriptorFactory` exposes static methods that return such an instance. The DSL checks during compile time whether the defined `acquire` statement is valid.

Invalid SensorFactory For all Loads in a DSL script the required Sensors must be provided by the SensorFactory. The DSL does not check whether the Loads performed within the script are valid with regard to the SensorFactory. It only checks whether the Loads are valid concerning the hADL model.

Invalid hADL-Element - ResourceDescriptor combination The `acquire` statement is checked if the hADL-Element can be acquired and the factory method returns an instance of `TResourceDescriptor`. Neither the DSL nor the hADL runtime framework check if the combination is valid. In the evaluation scenario it is possible to use a

ResourceDescriptor for Agilefant to acquire a hADL-Element that represents an entity on Bitbucket.

Cyclic Loads Assume the hADL model has two HumanComponents that are referencing each other. Hence it is possible to perform a circular load. If this is done often enough the DSL program will crash due to stack overflow as the implementation is based on recursion. The DSL assumes that the regular case is that loads are in the order of a couple and therefore a stack overflow should never occur.

5.3.2 Surrogates and Sensors

The implementations of Surrogates, Sensors and their factories are an ideal point for developers to crash the DSL. Especially because the DSL relies on sequential execution of the statements of a Task. For example if any method of a Surrogate or Sensor blocks, the whole DSL execution will block.

State changes Surrogates are responsible for handling (changing and reacting to different states) the Operational state and the Surrogate state. If the implementation does not do that properly, the resulting behaviour can be unexpected. Furthermore if an Operational was already released, consecutive start/stop Surrogate scope operations still trigger methods of the Surrogate (`asyncBegin`, `asyncStop`, `asyncLinkTo`, `asyncDisconnectFrom` and `asyncRelating`).

Factories Both Surrogate and Sensor factories are provided to the DSL. The Surrogate factory is specified within the hADL model and the Sensor factory is defined through the DSL itself (`sensorFactory` keyword). The hADL runtime framework and therefore the DSL relies on proper implementations of the factories. This means, that for every Collaborator and CollaborationObject that is acquired a corresponding Surrogate must be returned. Moreover for every hADL-Operational-Element that is used as a starting point of a Load a corresponding Sensor must be returned.

Network connection As hADL-Operational-Elements have corresponding entities on collaboration platforms it is assumed that a network connection is available. Temporary outages of the network may be handled by the Surrogate/Sensor implementations, but in order to successfully execute collaborations a functional network is required.

5.3.3 Runtime

This section deals with attacks a developer can do during runtime from a Task callers perspective to hinder the execution of the collaborations.

Altered ProcessScope The `ProcessScope` object passed to every Task and returned from every Task (see 4.1.3) holds the whole state of the hADL runtime. It is assumed that the ProcessScope with all its containing objects are not accessed nor altered from Java code outside of Tasks. Only Tasks are allowed to alter the ProcessScope and therefore the state of the hADL runtime.

hADL input parameters The DSL assumes that the passed hADL input parameters (Operationals) to Tasks are part of the `HADLruntimeModel` which is itself part of the ProcessScope. A hADL object where this is not the case can not be used with the ProcessScope.

Invalid hADL input object states The Operationals passed as hADL input parameters to Tasks are assumed to be in a state that is valid. Validity of the states depends on which operations (hADL primitives) are executed on it. Whenever a hADL operation is executed with a particular Operational it must be in the correct state, otherwise the program ends with an exception.

Collaboration Platform The implementation assumes that only the generated code from the DSL uses the ProcessScope. In addition to that it assumes that the collaboration structures handled by the hADL runtime framework are only altered through the runtime framework as well. Assume we create a chat with the DSL, immediately after that delete it through the collaboration platform, and in a consecutive step use the chat with the DSL (Link, Reference, etc). Such inconsistencies between the Operational and its corresponding entity on a collaboration platform can only be handled through the Surrogate implementations.

CHAPTER 6

Conclusions

This last chapter concludes by summarising the achieved results throughout the thesis. Moreover the research questions introduced in Section 1.3 are revisited to point out which part of this work answers them. Finally it provides possible directions for future work.

6.1 Summary

The thesis starts off by giving a motivation for valid collaboration programs defined with hADL and the need for embedding collaborations in process-centric environments. It is followed by a discussion of related work, which includes human involvement processes (see 2.1), mashups (see 2.2), crowdsourcing (see 2.3), and software engineering and collaboration/coordination support (see 2.4). The section on human involvement processes explores areas on S-BPM, social BPMN, Little-JIL, and WS-HumanTask and BPEL4People. After related work hADL is introduced, which serves as the basis for the rest of the thesis. The discussed related work differs from hADL and this work's DSL in the way that it has a process-centric view on collaboration, is a form of distributed computation for and coordinate among humans, but do not specify how actual collaboration is performed. hADL compared to that does not have the process-centric view, but focusses on how collaboration and coordination is achieved. The DSL approaches the process-centric way by providing an easy way to invoke collaborations through Tasks from tasks/steps of a business process.

Chapter (see 3) introduces the design of the DSL that achieves the validity of hADL programs and removes effort from the developer. First the contribution to the current state of hADL and the main aspects for using a DSL is given. The language is specified in the next section in terms of syntax and semantics. The language includes elements for setting up the DSL and the hADL runtime framework, specifying callable abstractions, input and output parameters, the hADL primitives exposed by the LinkageConnector and the RuntimeMonitor, variables, and iterations. Furthermore types, validity and consistency checks, and error handling is discussed.

The next chapter (see 4) shows how the design of the DSL from the previous chapter is implemented. It describes how Xtext/Xtend is used to implement the syntax in form of their grammar, the validity and consistency checks are performed by Validators and Scopes, Content Proposals are defined for better usability through an IDE. Furthermore the hADL synchronous library is introduced which wraps the hADL runtime framework to be more closely to the design of the DSL and transform an inherent asynchronous API (hADL runtime framework) into a blocking version.

The results of the design and the implementation phase is a DSL that is used to define collaborations based on a hADL model. Through its concise syntax it reduces a lot of effort for the programmer and adds checks such that only valid hADL programs are valid DSL scripts.

Chapter 5 takes the DSL and evaluates it in terms of effort reduction for the programmer. First a scenario from agile software development is introduced. The use case takes two parts from a Scrum process: (i) inviting all responsible users of the stories of a sprint to dedicated chat rooms, and (ii) holding a sprint retrospective meeting by inviting all developers of a sprint into a single chat room and providing a wiki for notes. The collaboration structures of both parts are first set up and later torn done. Moreover the required steps to implement the use case with the DSL are described. These implementation steps include designing the structure in a hADL model, defining the collaboration instances with the DSL and implementing the Surrogates and Sensors to access the respective entities on the collaboration platforms Agilefant, Bitbucket and HipChat.

The implemented use case (with the DSL) is then evaluated against the Java version that uses the hADL runtime framework instead. Therefore the development effort in LOC and the mental work savings are measured. For the overall use case scenario the DSL version is by a factor of 7.3 less effort than the Java version. Furthermore the DSL carries out 70 validity and consistency checks, that a programmer would have to do when using Java.

The last section discusses assumptions that where taken when designing the DSL and consequently ways how a developer can break DSL scripts and executions.

6.2 Research Questions revisited

This section revisites the research questions stated in Section 1.3 and indicates which parts of this thesis answers the questions.

RQ 1 *How can we define valid hADL programs and what benefits does a programmer gain from such a program? How is validity of a hADL program achieved?*

Valid hADL programs are written with the DSL which is described in Chapter 3. When defining collaboration instances with the DSL the programmer only writes a small amount of set up instructions, uses a concise syntax compared to verbose Java code, does not have to use an asynchronous/call back programming model and can easily embed and invoke collaboration instances from business process through the Tasks concept. Overall the developer's effort writing code lines is reduced dramatically (see Sections 4.1.3 and 5.2). Compared to the Java type system the DSL is aware of hADL types identified by IDs of elements of hADL and checks those. This checks lead to valid programs, because only elements with correct hADL types are allowed at specific positions of the script. What kind of checks are defined and how they are implemented is part of Sections 3.4, 4.1.4 and 4.1.5.

RQ 2 *How can collaborations in hADL be abstracted so that they are usable from process-oriented languages with little set up costs?*

A superior way to define collaborations is a mix of process-oriented languages and structure-centric languages such as hADL. A complete merge of these two concepts is out of scope of this thesis, but by introducing the concept of Tasks (see 3.3.2), the input and output of the hADL state (`ProcessScope`) and the according Java code generation (see 4.1.3), the DSL brings those approaches closer together. The set up section of the DSL is only a few lines compared to many lines necessary in Java (see Sections 3.3.1 and 5.2).

6.3 Future Work

This section points into possible directions for work that can follow this thesis. Section 5.3 introduces malicious steps a developer can take to interfere with the DSL. The following paragraphs explain how these situations could be avoided in future work and additionally mention areas which are worth exploring in successive research.

Robustness through static checks There is room for a lot of improvement concerning static checks. Checking the correctness of the hADL model as soon as it is specified as first statement of a DSL script. Adding ResourceDescriptor factories to the hADL-project and checking on acquire if the specified hADL-Element is compatible with the provided ResourceDescriptor. Information on what hADL-Element requires which ResourceDescriptor could be added as an `extension` to the hADL model. The parser could do a second run and check whether the provided Surrogate and Sensor factories return valid objects for the input DSL script. Concerning the cyclic Loads the parser could limit the via steps to a fixed number.

Sophisticated error handling As described in Section 3.5 error handling is rudimentary in the DSL. A more sophisticated error handling mechanism is desirable. Possible solutions include (i) try-catch behaviour or (ii) error handling strategies such as retry or alternative. Different reactions to different (possibly predefined) hADL errors are of interest. There could be explicit error handling which is defined by the developer and reactions to errors that the DSL runtime can handle on its own (failure transparency). Moreover a rollback strategy is interesting. Rollback would need additional methods in the Surrogates such as acquire and unacquire (not necessarily the same as release).

Support for all hADL features As of now the DSL does not support different types of References such as composition and inheritance. Moreover sub structures are currently not supported.

Type safe input and output parameters Output parameters are returned in form of a Java `Map<String, Object>`. Hence the developer has to access the Map by providing a String and then casting the returned object to the correct type. This adds complexity for the programmer. A better solution would be that for every Task an output class is generated with type correct getters for the returned parameters. To take it even further the code generator could create a unique class for every hADL type (maybe a subtype of the corresponding Java type). This would remove the runtime check of hADL input parameters and would simplify the hADL type checks (`VariableManager`).

Process support The DSL makes it easy to define collaborations and group them in callable entities (Tasks). A step further is to combine the already working functionality with a process engine. Two possible approaches are: (i) take the DSL script as input and include the Tasks in a existing process engine (e.g. Little-JIL, BPMN) by extending it, or (ii) add process primitives to the existing DSL.

Additional statements Apart from error handling (try-catch) additional statements would be of interest for a future version of the DSL. To us the most appealing statement would be a conditional statement (if-else). This could be a worthy addition especially if the states of Operationals and Surrogates were accessible from within the DSL. Another idea for a statement is one that has XOR semantic, where multiple statements could be listed and only one should succeed (would need rollback functionality).

List of Figures

List of Tables

List of Listings

List of Algorithms

Glossary

hADL-Connective is either a Link or a Reference in hADL. 15, 31, 41, 43, 60, 93

hADL-Element is either a Collaborator or a CollaborationObject. 15, 17, 21, 25, 26, 29, 31–33, 39, 41–43, 58, 60, 65, 66, 72–74, 80, 81, 85, 87, 88, 93

hADL-Language-Element is either a hADL-Element, hADL-Connective or an Action. 27, 33, 39

hADL-Operational-Connective is the corresponding Operational of a hADL-Connective. 53

hADL-Operational-Element is the corresponding Operational of a hADL-Element. 17, 21, 29–33, 40, 41, 53, 58, 60, 68, 74, 81, 87, 88

Agilefant is a Scrum online tool. 2, 14, 20, 65–67, 75, 76, 81, 84, 87

Bitbucket is a git hosting and collaboration platform by Atlassian. 2, 14, 20, 65, 66, 73–76, 81, 84

Collaborator is either a HumanComponent or CollaborationConnector in hADL. 16, 17, 29–31, 33, 40, 54, 58, 59, 81, 93

git is a free and open source distributed version control system. 13, 14, 93

Guice a lightweight DI framework developed by Google.. 22, 50

HipChat is a team collaboration platform by Atlassian, that provides communication mechanisms such as chat rooms and private messages. 2, 14, 17, 20, 65, 66, 73, 75, 76, 84

Maven a dependency and build tool for Java.. 47

OSGi a dynamic component/module system for Java.. 47

OSX Operating System (OS) developed by Apple.. 55

RXJava Reactive Extensions for Java developed by Netflix Inc.. 23, 57

Scrum is an agile software development methodology. xi, 2, 13, 64, 65, 67, 72, 84, 93

SOAP a protocol for exchanging XML based messages. It is highly used in Web Services.. 13

Unix an OS originally developed by Bell Labs.. 24

Acronyms

AGR Agent/Group/Role. 6

API Application Programming Interface. 1–4, 10, 11, 13, 17, 19, 21, 23, 41, 45, 49, 57, 65, 66, 75, 84, 87

BPEL Business Process Execution Language. 8, 9, 83

BPM Business Process Management. 5–7

BPMN Business Process Model and Notation. 3, 7–10, 18, 19, 83, 86

CCS Calculus of Communicating Systems. 6

CI Continuous Integration. 13

CS Crowdsourcing System. 1, 11–14

CSD Communication Structure Diagram. 6

CSP Communicating Sequential Processes. 6

DI Dependency Injection. 18, 46, 77, 93

DSL Domain-Specific Language. xi, 1–5, 8–10, 13, 14, 18–27, 32–34, 36, 38–42, 45–53, 55–60, 62–64, 66, 67, 72, 76–89, 91

hADL Human Architecture Description Language. ix, xi, 1–5, 7–10, 13–36, 38–43, 45–48, 50–58, 60–74, 76–88, 93

HTTP Hypertext Transfer Protocol. 65, 66

ID Identifier. 21, 22, 27, 35, 50, 58, 66, 85

IDE Integrated Development Environment. xi, 1, 13, 14, 21, 24, 45, 46, 55, 84

LOC Lines of Code. xi, 76–78, 84

MTurk Amazon Mechanical Turk. 11, 12

OS Operating System. 93, 94

RDF Resource Description Framework. 7

REST Representational State Transfer. 65, 66, 75

S-BPM Subject-oriented Business Process Management. 5, 6, 83

SBD Subject Behaviour Diagram. 6

SDK Software Development Kit. 46

SE Software Engineering. 13

SID Subject Interaction Diagram. 6

SMW Semantic MediaWiki. 7

SQL Structured Query Language. 11

UI User Interface. 2, 10, 11, 47, 56

UML Unified Modeling Language. 13

URL Uniform Resource Locator. 10, 65, 66

UX User Experience. 25

WF Windows Workflow Foundation. 6

WS Web Services. 8, 9, 13

WS-HumanTask Web Services Human Task. 5, 8, 9, 13, 83

WSDL Web Services Description Language. 9, 13

XML eXtensible Markup Language. 13, 15, 16, 18, 38, 65, 75, 87, 94, 96

XSD XML Schema Definition. 13

Bibliography

[1] C. Dorn, S. Dustdar, and L. J. Osterweil, “Specifying flexible human behavior in interaction-intensive process environments”, in *Business Process Management*, ser. Lecture Notes in Computer Science 8659, S. Sadiq, P. Soffer, and H. Völzer, Eds., DOI: 10.1007/978-3-319-10172-9_24, Springer International Publishing, Sep. 7, 2014, pp. 366–373, ISBN: 978-3-319-10171-2 978-3-319-10172-9. [Online]. Available: `http://link.springer.com/chapter/10.1007/978-3-319-10172-9_24` (visited on 12/02/2015).

[2] ——, “Strategies for specifying flexible human behavior in interaction-intensive process environments”, vol. Submitted,

[3] C. Mayr-Dorn and S. Dustdar, “A framework for model-driven execution of collaboration structures”, in *Proceedings of the 28th International Conference on Advanced Information Systems Engineering (CAiSE)*, Ljubljana, Slovenia: Springer, Jun. 2016.

[4] A. Fleischmann, W. Schmidt, C. Stary, S. Obermeier, and E. Börger, *Subject-Oriented Business Process Management.* Springer Berlin Heidelberg, 2012, DOI: 10.1007/978-3-642-32392-8_12, ISBN: 978-3-642-32391-1 978-3-642-32392-8. [Online]. Available: `http://link.springer.com/chapter/10.1007/978-3-642-32392-8_12` (visited on 12/16/2015).

[5] A. Fleischmann, U. Kannengiesser, W. Schmidt, and C. Stary, “Subject-oriented modeling and execution of multi-agent business processes”, in *Proceedings of the 2013 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT) - Volume 02*, ser. WI-IAT ’13, Washington, DC, USA: IEEE Computer Society, 2013, pp. 138–145, ISBN: 978-0-7695-5145-6. DOI: `10.1109/WI-IAT.2013.102`. [Online]. Available: `http://dx.doi.org/10.1109/WI-IAT.2013.102` (visited on 12/16/2015).

[6] S. Oppl, “Articulation of subject-oriented business process models”, in *Proceedings of the 7th International Conference on Subject-Oriented Business Process Management*, ser. S-BPM ONE ’15, New York, NY, USA: ACM, 2015, 2:1–2:11, ISBN: 978-1-4503-3312-2. DOI: `10.1145/2723839.2723841`. [Online]. Available: `http://doi.acm.org/10.1145/2723839.2723841` (visited on 12/02/2015).

[7] S. Raß, J. Kotremba, and R. Singer, "The s-BPM architecture: A framework for multi-agent systems", in *Proceedings of the 2013 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT) - Volume 03*, ser. WI-IAT '13, Washington, DC, USA: IEEE Computer Society, 2013, pp. 78–82, ISBN: 978-0-7695-5145-6. DOI: 10.1109/WI-IAT.2013.154. [Online]. Available: http://dx.doi.org/10.1109/WI-IAT.2013.154 (visited on 12/02/2015).

[8] C. Krauthausen and S. Krauthausen, "Engines supporting the subject-oriented paradigm", in *Proceedings of the 7th International Conference on Subject-Oriented Business Process Management*, ser. S-BPM ONE '15, New York, NY, USA: ACM, 2015, 11:1–11:7, ISBN: 978-1-4503-3312-2. DOI: 10.1145/2723839.2723851. [Online]. Available: http://doi.acm.org/10.1145/2723839.2723851 (visited on 12/16/2015).

[9] F. Dengler, A. Koschmider, A. Oberweis, and H. Zhang, "Social software for coordination of collaborative process activities", in *Business Process Management Workshops*, ser. Lecture Notes in Business Information Processing 66, M. z. Muehlen and J. Su, Eds., DOI: 10.1007/978-3-642-20511-8_37, Springer Berlin Heidelberg, Sep. 13, 2010, pp. 396–407, ISBN: 978-3-642-20510-1 978-3-642-20511-8. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-642-20511-8_37 (visited on 12/01/2015).

[10] M. Brambilla, P. Fraternali, and C. Vaca, "BPMN and design patterns for engineering social BPM solutions", in *Business Process Management Workshops*, ser. Lecture Notes in Business Information Processing 99, F. Daniel, K. Barkaoui, and S. Dustdar, Eds., DOI: 10.1007/978-3-642-28108-2_22, Springer Berlin Heidelberg, Aug. 29, 2011, pp. 219–230, ISBN: 978-3-642-28107-5 978-3-642-28108-2. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-642-28108-2_22 (visited on 12/01/2015).

[11] M. Brambilla, P. Fraternali, and C. K. Vaca Ruiz, "Combining social web and BPM for improving enterprise performances: The BPM4people approach to social BPM", in *Proceedings of the 21st International Conference on World Wide Web*, ser. WWW '12 Companion, New York, NY, USA: ACM, 2012, pp. 223–226, ISBN: 978-1-4503-1230-1. DOI: 10.1145/2187980.2188014. [Online]. Available: http://doi.acm.org/10.1145/2187980.2188014 (visited on 12/02/2015).

[12] S. Schwantzer and N. Faltin, "Prowit: Integrated web 2.0 business process collaboration service-platform", in *Proceedings of the 11th International Conference on Knowledge Management and Knowledge Technologies*, ser. i-KNOW '11, New York, NY, USA: ACM, 2011, 27:1–27:4, ISBN: 978-1-4503-0732-1. DOI: 10.1145/2024288.2024322. [Online]. Available: http://doi.acm.org/10.1145/2024288.2024322 (visited on 12/02/2015).

[13] A. Cass, A. Lerner, E. McCall, L. Osterweil, S. Sutton, and A. Wise, “Little-JIL/juliette: A process definition language and interpreter”, in *Proceedings of the 2000 International Conference on Software Engineering, 2000*, 2000, pp. 754–757. DOI: 10.1109/ICSE.2000.870488.

[14] A. Wise, A. G. Cass, B. Staudt Lerner, E. K. McCall, L. J. Osterweil, and S. M. Sutton Jr., “Using little-JIL to coordinate agents in software engineering”, presented at the Automated Software Engineering Conference (ASE 2000), Grenoble, France, Sep. 2000, pp. 155–163.

[15] *WS-BPEL extension for people (BPEL4people) specification version 1.1*, Aug. 17, 2010. [Online]. Available: http://docs.oasis-open.org/bpel4people/bpel4people-1.1.pdf (visited on 12/16/2015).

[16] *Web services human task (WSHumanTask) specification version 1.1*, Jul. 24, 2012. [Online]. Available: http://docs.oasis-open.org/bpel4people/ws-humantask-1.1.html (visited on 12/16/2015).

[17] M. Gerhards, S. Skorupa, V. Sander, P. Pfeiffer, and A. Belloum, “Towards a security framework for a WS-HumanTask processor”, in *Proceedings of the 7th International Conference on Network and Services Management*, ser. CNSM ’11, Laxenburg, Austria, Austria: International Federation for Information Processing, 2011, pp. 484–488, ISBN: 978-3-901882-44-9. [Online]. Available: http://dl.acm.org/citation.cfm?id=2147671.2147761 (visited on 12/16/2015).

[18] F. Daniel, A. Koschmider, T. Nestler, M. Roy, and A. Namoun, “Toward process mashups: Key ingredients and open research challenges”, in *Proceedings of the 3rd and 4th International Workshop on Web APIs and Services Mashups*, ser. Mashups ’09/’10, New York, NY, USA: ACM, 2010, 9:1–9:8, ISBN: 978-1-4503-0418-4. DOI: 10.1145/1944999.1945008. [Online]. Available: http://doi.acm.org/10.1145/1944999.1945008 (visited on 03/24/2015).

[19] V. Torres, J. M. Pérez, A. Koschmider, and F. Daniel, “Dealing with collaborative tasks in process mashups”, in *Proceedings of the 5th International Workshop on Web APIs and Service Mashups*, ser. Mashups ’11, New York, NY, USA: ACM, 2011, 4:1–4:8, ISBN: 978-1-4503-0823-6. DOI: 10.1145/2076006.2076011. [Online]. Available: http://doi.acm.org/10.1145/2076006.2076011 (visited on 12/02/2015).

[20] M. Kunze, H. Overdick, A. Grosskopf, and M. Weidlich, “Lightweight collaboration management”, in *Proceedings of the 3rd and 4th International Workshop on Web APIs and Services Mashups*, ser. Mashups ’09/’10, New York, NY, USA: ACM, 2010, 3:1–3:8, ISBN: 978-1-4503-0418-4. DOI: 10.1145/1944999.1945002. [Online]. Available: http://doi.acm.org/10.1145/1944999.1945002 (visited on 12/02/2015).

[21] M.-C. Yuen, I. King, and K.-S. Leung, "A survey of crowdsourcing systems", in *2011 IEEE Third International Conference on Privacy, Security, Risk and Trust (PASSAT) and 2011 IEEE Third Inernational Conference on Social Computing (SocialCom)*, Oct. 2011, pp. 766–773. DOI: 10.1109/PASSAT/SocialCom.2011.203.

[22] A. Doan, R. Ramakrishnan, and A. Y. Halevy, "Crowdsourcing systems on the world-wide web", *Commun. ACM*, vol. 54, no. 4, pp. 86–96, Apr. 2011, ISSN: 0001-0782. DOI: 10.1145/1924421.1924442. [Online]. Available: http://doi.acm.org/10.1145/1924421.1924442 (visited on 03/23/2015).

[23] T. W. Malone, R. Laubacher, and C. Dellarocas, "Harnessing crowds: Mapping the genome of collective intelligence", Social Science Research Network, Rochester, NY, SSRN Scholarly Paper ID 1381502, Feb. 3, 2009. [Online]. Available: http://papers.ssrn.com/abstract=1381502 (visited on 03/23/2015).

[24] S. Ahmad, A. Battle, Z. Malkani, and S. Kamvar, "The jabberwocky programming environment for structured social computing", in *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '11, New York, NY, USA: ACM, 2011, pp. 53–64, ISBN: 978-1-4503-0716-1. DOI: 10.1145/2047196.2047203. [Online]. Available: http://doi.acm.org/10.1145/2047196.2047203 (visited on 03/23/2015).

[25] A. Kittur, B. Smus, S. Khamkar, and R. E. Kraut, "Crowdforge: Crowdsourcing complex work", in *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '11, New York, NY, USA: ACM, 2011, pp. 43–52, ISBN: 978-1-4503-0716-1. DOI: 10.1145/2047196.2047202. [Online]. Available: http://doi.acm.org/10.1145/2047196.2047202 (visited on 03/23/2015).

[26] A. Kulkarni, M. Can, and B. Hartmann, "Collaboratively crowdsourcing workflows with turkomatic", in *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, ser. CSCW '12, New York, NY, USA: ACM, 2012, pp. 1003–1012, ISBN: 978-1-4503-1086-4. DOI: 10.1145/2145204.2145354. [Online]. Available: http://doi.acm.org/10.1145/2145204.2145354 (visited on 05/04/2015).

[27] P. Minder and A. Bernstein, "CrowdLang — first steps towards programmable human computers for general computation", in *Workshops at the Twenty-Fifth AAAI Conference on Artificial Intelligence*, Aug. 24, 2011. [Online]. Available: https://www.aaai.org/ocs/index.php/WS/AAAIW11/paper/view/3891 (visited on 03/23/2015).

[28] D. Schall, "A human-centric runtime framework for mixed service-oriented systems", *Distributed and Parallel Databases*, vol. 29, no. 5, pp. 333–360, Mar. 29, 2011, ISSN: 0926-8782, 1573-7578. DOI: 10.1007/s10619-011-7081-z. [Online]. Available: http://link.springer.com/article/10.1007/s10619-011-7081-z (visited on 03/23/2015).

[29] J. Whitehead, “Collaboration in software engineering: A roadmap”, in *2007 Future of Software Engineering*, ser. FOSE ’07, Washington, DC, USA: IEEE Computer Society, 2007, pp. 214–225, ISBN: 0-7695-2829-5. DOI: `10.1109/FOSE.2007.4`. [Online]. Available: `http://dx.doi.org/10.1109/FOSE.2007.4` (visited on 03/23/2015).

[30] T. D. LaToza, W. B. Towne, C. M. Adriano, and A. van der Hoek, “Microtask programming: Building software with a crowd”, in *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST ’14, New York, NY, USA: ACM, 2014, pp. 43–54, ISBN: 978-1-4503-3069-5. DOI: `10.1145/2642918.2647349`. [Online]. Available: `http://doi.acm.org/10.1145/2642918.2647349` (visited on 04/01/2015).

[31] C. Dorn and A. Egyed, “Towards collaboration-centric pattern-based software development support”, in *2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, May 2013, pp. 109–112. DOI: `10.1109/CHASE.2013.6614743`.

[32] C. Dorn and R. N. Taylor, “Architecture-driven modeling of adaptive collaboration structures in large-scale social web applications”, ISR Technical Report UCI-ISR-12-5, May 2012. [Online]. Available: `http://isr.uci.edu/tech_reports/UCI-ISR-12-5.pdf` (visited on 03/26/2015).

[33] C. Dorn. (). Christophdorn / hADL, [Online]. Available: `https://bitbucket.org/christophdorn/hadl` (visited on 12/09/2015).

[34] C. Dorn and R. N. Taylor, “Analyzing runtime adaptability of collaboration patterns”, *Concurrency and Computation: Practice and Experience*, vol. 27, no. 11, pp. 2725–2750, Aug. 10, 2015, ISSN: 1532-0634. DOI: `10.1002/cpe.3438`. [Online]. Available: `http://onlinelibrary.wiley.com/doi/10.1002/cpe.3438/abstract` (visited on 12/01/2015).

[35] C. Dorn, R. Taylor, and S. Dustdar, “Flexible social workflows: Collaborations as human architecture”, *IEEE Internet Computing*, vol. 16, no. 2, pp. 72–77, Mar. 2012, ISSN: 1089-7801. DOI: `10.1109/MIC.2012.33`.

[36] Netflix Inc. (). ReactiveX/RxJava, GitHub, [Online]. Available: `https://github.com/ReactiveX/RxJava` (visited on 01/11/2016).

[37] Oracle. (). The for statement, The Java^{TM} Tutorials, [Online]. Available: `https://docs.oracle.com/javase/tutorial/java/nutsandbolts/for.html` (visited on 01/14/2016).

[38] C. Laaber. (). Chrstphlbr / MT_dsl, [Online]. Available: `https://bitbucket.org/chrstphlbr/mt_dsl` (visited on 01/22/2016).

[39] ——, (). Chrstphlbr / MT_dsl-lib, [Online]. Available: `https://bitbucket.org/chrstphlbr/mt_dsl-lib` (visited on 01/22/2016).

[40] M. Fowler, *Domain-Specific Languages*, 1st ed. Addison-Wesley Professional, Sep. 23, 2010, ISBN: 978-0-321-71294-3.

[41] École Polytechnique Fédérale de Lausanne (EPFL). (2016). Scala, The Scala Programming Language, [Online]. Available: `http://www.scala-lang.org/` (visited on 01/18/2016).

[42] haskell.org. (). Haskell, Haskell Language, [Online]. Available: `https://www.haskell.org/` (visited on 01/18/2016).

[43] Eclipse Foundation, Inc. (). Xtext, Xtext, language engineering for everyone!, [Online]. Available: `https://eclipse.org/Xtext/index.html` (visited on 01/18/2016).

[44] ——, (). Epsilon, Epsilon, [Online]. Available: `http://www.eclipse.org/epsilon/` (visited on 01/18/2016).

[45] ——, (). Xtend, Xtend - Modernized java, [Online]. Available: `http://www.eclipse.org/xtend/` (visited on 01/18/2016).

[46] Oracle. (). Java documentation, Java$^{\text{TM}}$ Platform, Standard Edition 8 API Specification, [Online]. Available: `https://docs.oracle.com/javase/8/docs/api/index.html` (visited on 01/18/2016).

[47] K. S. Rubin, *Essential Scrum: A practical guide to the most popular agile process.* Addison-Wesley Professional, Jul. 26, 2012, ISBN: 978-0-321-70040-7.

[48] C. Dorn. (). Christophdorn / agilefant 4 hADL, [Online]. Available: `https://bitbucket.org/christophdorn/agilefant-4-hadl` (visited on 01/22/2016).

[49] C. Laaber. (). Chrstphlbr / MT_atlassian4hadl, [Online]. Available: `https://bitbucket.org/chrstphlbr/mt_atlassian4hadl` (visited on 01/22/2016).

[50] ——, (). Chrstphlbr / MT_evaluation_dsl, [Online]. Available: `https://bitbucket.org/chrstphlbr/mt_evaluation_dsl` (visited on 01/22/2016).

[51] ——, (). Chrstphlbr / MT_evaluation, [Online]. Available: `https://bitbucket.org/chrstphlbr/mt_evaluation` (visited on 01/22/2016).

[52] ——, (). Chrstphlbr / MT_agilefant4hadl, [Online]. Available: `https://bitbucket.org/chrstphlbr/mt_agilefant4hadl` (visited on 01/22/2016).

Appendices

DSL Grammar Specification

```
grammar net.laaber.mt.DSL with org.eclipse.xtext.xbase.
   Xbase

generate dSL "http://www.laaber.net/mt/DSL"
import "http://www.eclipse.org/xtext/xbase/Xbase"
import "http://www.eclipse.org/xtext/common/JavaVMTypes"
    as jvmTypes

Domainmodel:
  'hADL' file = STRING
  'className' className = QualifiedName
  rdFacs += ResourceDescriptorFactory+
  sensorFac = SensorFactory
  tasks += Task*
;

Var:
  name = ID
;

OutputKey:
  name = STRING | keyVariable = [Var]
;

ResourceDescriptorFactory:
  'resourceDescriptorFactory' name = ID 'class' type =
     JvmTypeReference
;

SensorFactory:
  'sensorFactory' type = JvmTypeReference
;

Task:
  'task' name = ID '{'
    inputs += Input*
    statements += Statement*
    outputs += Output*
  '}'
;
```

```

// input variables are values (final)
Input:
  JavaInput | HadlInput
;

JavaInput:
  'javaIn' variable = Var ':' javaType =
     JvmTypeReference
;

HadlInput:
  'in' variable = Var ':' type = HadlType
;

HadlType:
  HadlSimpleType | HadlListType
;

HadlSimpleType:
  element = QualifiedName
;

HadlListType:
  '[' element = QualifiedName ']'
;

Output:
  'out' variable = [Var] 'as' key = OutputKey
;

Statement:
  HadlStatement | StructuralStatement
;

HadlStatement:
  Acquire | Release | Link | Unlink | Reference |
     Dereference | Load | StartScope | StopScope
;

StructuralStatement:
  Iteration | HadlVariableDeclaration |
     HadlVariableAssignment | HadlListVariableAppend
```

```
;

Method:
  name = ID
;

Param:
  s = STRING | i = INT | f = Float | variable = [Var]
;

// variable must be an Integer
AppendableParam:
  param = Param ('+' add = [Var])?
;

Float:
  INT '.' (INT)+
;

/* hADL Statements */

// acquire variables are values (final)
Acquire:
  'acquire' element = QualifiedName 'with' rd =
      ResourceDescriptor 'as' variable = Var
;

ResourceDescriptor:
  factory = [ResourceDescriptorFactory] '.' method =
      Method '(' (params+=AppendableParam)? (',' params+=
      AppendableParam)* ')'
;

Release:
  'release' variable = [Var]
;

// link variables are values (final)
Link:
  'link' collaborator = [Var] 'and' object = [Var] 'by'
      link = QualifiedName 'as' variable = Var
;

```

```
Unlink:
  'unlink' link = [Var]
;

// load variables are values (final)
Load:
  'startingFrom' startingFrom = [Var] (vias += LoadVia)*
      'load' what = LoadFromVia 'as' variable = Var
;

LoadVia:
  'via' via = LoadFromVia
;

LoadFromVia:
  from = QualifiedName':' via = QualifiedName
;

// reference variables are values (final)
Reference:
  'reference from' from = [Var] 'to' to = [Var] 'with'
     ref = QualifiedName 'as' variable = Var
;

Dereference:
  'dereference' ref = [Var]
;

StartScope:
  {StartScope}'startScope'
;

StopScope:
  {StopScope}'stopScope'
;

/* Structural Statements */

Iteration:
  'for all' list = [Var] 'as' element = Var ('counter'
     loopVariable = Var)?
  '{'
    statements += Statement+
  '}'
;

// hadl variables are true variables (destructive
   assignment possible)
HadlVariableDeclaration:
  'var' variable = Var ':' type = HadlType
;

HadlVariableAssignment:
  'assign' right = [Var] 'to' left = [Var]
;

HadlListVariableAppend:
  'add' right = [Var] 'to' left = [Var]
;
```

Scrum hADL Model

```xml
<?xml version="1.0" encoding="UTF-8"?>
<hADLmodel id="scrum"
      xmlns="http://at.ac.tuwien.dsg/hADL/hADLcore"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:hADLexe="http://at.ac.tuwien.dsg/hADL/hADLexecutable"
      xmlns:xlink="http://www.w3.org/1999/xlink"

      xsi:schemaLocation="http://at.ac.tuwien.dsg/hADL/hADLcore     ../schema/hADLcore.xsd  ">
  <name>scrum</name>
  <description />
  <extension>
    <hADLexe:Executables>
      <hADLexe:collabPlatform xlink:href="http://www.scrum.org" xlink:type="simple" id="scrumPlatform">
        <name>ScrumPlatforms</name>
        <description>Scrum Platforms</description>
        <extension />
        <hADLexe:acceptableResourceDescriptor>tempuri.org</hADLexe:acceptableResourceDescriptor>
      </hADLexe:collabPlatform>
      <hADLexe:surrogate id="AgilefantSprintSurrogate">
        <name>Agilefant Sprint Surroagate</name>
        <description></description>
        <extension />
        <hADLexe:surrogateFQN>at.ac.tuwien.dsg.hadl.framework.runtime.utils.DefaultAcceptingObjectSurrogate</hADLexe:surrogateFQN>
        <hADLexe:surrogateFactoryFQN>at.ac.tuwien.dsg.hadl.framework.runtime.impl.DefaultSurrogateFactory</hADLexe:surrogateFactoryFQN>
        <hADLexe:collabPlatform>scrumPlatform</hADLexe:collabPlatform>
      </hADLexe:surrogate>
      <hADLexe:surrogate id="AgilefantStorySurrogate">
        <name>Agilefant Story Surrogate</name>
        <description></description>
```

```
     <extension />
     <hADLexe:surrogateFQN>at.ac.tuwien.dsg.hadl.framework.runtime.utils.DefaultAcceptingObjectSurrogate</hADLexe:surrogateFQN>
     <hADLexe:surrogateFactoryFQN>at.ac.tuwien.dsg.hadl.framework.runtime.impl.DefaultSurrogateFactory</hADLexe:surrogateFactoryFQN>
     <hADLexe:collabPlatform>scrumPlatform</hADLexe:collabPlatform>
   </hADLexe:surrogate>
   <hADLexe:surrogate id="AgilefantUserSurrogate">
     <name>Agilefant User Surrogate</name>
     <hADLexe:surrogateFQN>at.ac.tuwien.dsg.hadl.framework.runtime.utils.DefaultAcceptingCollaboratorSurrogate</hADLexe:surrogateFQN>
     <hADLexe:surrogateFactoryFQN>at.ac.tuwien.dsg.hadl.framework.runtime.impl.DefaultSurrogateFactory</hADLexe:surrogateFactoryFQN>
     <hADLexe:collabPlatform>scrumPlatform</hADLexe:collabPlatform>
   </hADLexe:surrogate>
   <hADLexe:surrogate id="AtlassianUserSurrogate">
     <name>AtlassianUserSurroagate</name>
     <description></description>
     <extension />
     <hADLexe:surrogateFQN>net.laaber.mt.hadl.atlassian.surrogate.UserSurrogate</hADLexe:surrogateFQN>
     <hADLexe:surrogateFactoryFQN>at.ac.tuwien.dsg.hadl.framework.runtime.impl.DefaultSurrogateFactory</hADLexe:surrogateFactoryFQN>
     <hADLexe:collabPlatform>scrumPlatform</hADLexe:collabPlatform>
   </hADLexe:surrogate>
   <hADLexe:surrogate id="AtlassianChatSurrogate">
     <name>Atlassian Chat Surrogate</name>
     <description></description>
     <extension />
     <hADLexe:surrogateFQN>net.laaber.mt.hadl.atlassian.surrogate.ChatSurrogate</hADLexe:surrogateFQN>
```

```xml
        <hADLexe:surrogateFactoryFQN>at.ac.tuwien.dsg.hadl.framework.runtime.impl.DefaultSurrogateFactory</hADLexe:surrogateFactoryFQN>
        <hADLexe:collabPlatform>scrumPlatform</hADLexe:collabPlatform>
      </hADLexe:surrogate>
      <hADLexe:surrogate id="AtlassianWikiSurrogate">
        <name>Atlassian Wiki Surrogate</name>
        <hADLexe:surrogateFQN>net.laaber.mt.hadl.atlassian.surrogate.WikiSurrogate</hADLexe:surrogateFQN>
        <hADLexe:surrogateFactoryFQN>at.ac.tuwien.dsg.hadl.framework.runtime.impl.DefaultSurrogateFactory</hADLexe:surrogateFactoryFQN>
        <hADLexe:collabPlatform>scrumPlatform</hADLexe:collabPlatform>
      </hADLexe:surrogate>
    </hADLexe:Executables>
  </extension>
  <hADLstructure id="scrum.obj">
    <name>obj</name>
    <description />
    <extension />
    <object id="scrum.obj.Sprint" type="ARTIFACT" xsi:type="tSimpleCollabObject">
      <name>Sprint</name>
      <description />
      <extension>
        <hADLexe:ExecutableRefExtension>
          <hADLexe:executableViaSurrogate>AgilefantSprintSurrogate</hADLexe:executableViaSurrogate>
        </hADLexe:ExecutableRefExtension>
      </extension>
    </object>
    <object id="scrum.obj.Story" type="ARTIFACT" xsi:type="tSimpleCollabObject">
      <name>Story</name>
      <description />
      <extension>
        <hADLexe:ExecutableRefExtension>
          <hADLexe:executableViaSurrogate>AgilefantStorySurrogate</hADLexe:
```

```
executableViaSurrogate>
     </hADLexe:ExecutableRefExtension>
   </extension>
   <action id="scrum.obj.Story.userResponsible">
     <name>userResponsible</name>
     <description />
     <extension />
   </action>
 </object>
 <object id="scrum.obj.Wiki" type="ARTIFACT" xsi:type
="tSimpleCollabObject">
   <name>Wiki</name>
   <description />
   <extension>
     <hADLexe:ExecutableRefExtension>
       <hADLexe:executableViaSurrogate>
AtlassianWikiSurrogate</hADLexe:
executableViaSurrogate>
     </hADLexe:ExecutableRefExtension>
   </extension>
 </object>
 <object id="scrum.obj.Chat" type="ARTIFACT" xsi:type
="tSimpleCollabObject">
   <name>Chat</name>
   <description />
   <extension>
     <hADLexe:ExecutableRefExtension>
       <hADLexe:executableViaSurrogate>
AtlassianChatSurrogate</hADLexe:
executableViaSurrogate>
     </hADLexe:ExecutableRefExtension>
   </extension>
   <action id="scrum.obj.Chat.inviteUser">
     <name>inviteUser</name>
     <description />
     <extension />
   </action>
 </object>
 <objectRef id="scrum.obj.containsStory" xsi:type="
tObjectCompositionRef">
   <name>containsStory</name>
   <objFrom>scrum.obj.Sprint</objFrom>
   <objTo>scrum.obj.Story</objTo>
```

```
      <compositionType>FROM_REFERENCES_TO</compositionType>
    </objectRef>
    <objectRef id="scrum.obj.prev" xsi:type="tObjectCompositionRef">
      <name>prev</name>
      <objFrom>scrum.obj.Wiki</objFrom>
      <objTo>scrum.obj.Wiki</objTo>
      <compositionType>FROM_REFERENCES_TO</compositionType>
    </objectRef>
    <objectRef id="scrum.obj.next" xsi:type="tObjectCompositionRef">
      <name>next</name>
      <objFrom>scrum.obj.Wiki</objFrom>
      <objTo>scrum.obj.Wiki</objTo>
      <compositionType>FROM_REFERENCES_TO</compositionType>
    </objectRef>
  </hADLstructure>
  <hADLstructure id="scrum.col">
    <name>col</name>
    <description />
    <extension />
    <component id="scrum.col.AgileUser">
      <name>AgileUser</name>
      <description />
      <extension>
        <hADLexe:ExecutableRefExtension>
          <hADLexe:executableViaSurrogate>AgilefantUserSurrogate</hADLexe:executableViaSurrogate>
        </hADLexe:ExecutableRefExtension>
      </extension>
      <action id="scrum.col.AgileUser.storyResponsible">
        <name>storyResponsible</name>
        <description />
        <extension />
      </action>
    </component>
    <component id="scrum.col.DevUser">
      <name>DevUser</name>
      <description />
    <extension>
      <hADLexe:ExecutableRefExtension>
        <hADLexe:executableViaSurrogate>
 AtlassianUserSurrogate</hADLexe:
 executableViaSurrogate>
      </hADLexe:ExecutableRefExtension>
    </extension>
    <action id="scrum.col.DevUser.inviteChat">
      <name>inviteChat</name>
      <description />
      <extension />
    </action>
  </component>
  <collabRef id="scrum.col.devUser" xsi:type="
 tCollabCompositionRef">
    <name>devUser</name>
    <compconnFrom>scrum.col.AgileUser</compconnFrom>
    <compconnTo>scrum.col.DevUser</compconnTo>
    <compositionType>FROM_DEPENDS_ON_TO</
 compositionType>
  </collabRef>
  <collabRef id="scrum.col.agileUser" xsi:type="
 tCollabCompositionRef">
    <name>agileUser</name>
    <compconnFrom>scrum.col.DevUser</compconnFrom>
    <compconnTo>scrum.col.AgileUser</compconnTo>
    <compositionType>FROM_DEPENDS_ON_TO</
 compositionType>
  </collabRef>
</hADLstructure>
<hADLstructure id="scrum.links">
  <name>links</name>
  <description />
  <extension />
  <link id="scrum.links.responsible">
    <name>responsible</name>
    <objActionEndpoint>scrum.obj.Story.userResponsible
 </objActionEndpoint>
    <collabActionEndpoint>scrum.col.AgileUser.
 storyResponsible</collabActionEndpoint>
  </link>
  <link id="scrum.links.invite">
    <name>invite</name>
```

```
      <objActionEndpoint>scrum.obj.Chat.inviteUser</
    objActionEndpoint>
      <collabActionEndpoint>scrum.col.DevUser.inviteChat
    </collabActionEndpoint>
    </link>
  </hADLstructure>
</hADLmodel>
```

Scrum DSL Script

```
hADL "/Users/Christoph/TU/Diplomarbeit/Code/Evaluation/
   src/main/resources/agile-hadl.xml"

className net.laaber.mt.evaluation.Tasks
resourceDescriptorFactory agilefantRd class net.laaber.
   mt.hadl.agilefant.resourceDescriptor.
   AgilefantResourceDescriptorFactory
resourceDescriptorFactory atlassianRd class net.laaber.
   mt.hadl.atlassian.resourceDescriptor.
   AtlassianResourceDescriptorFactory
sensorFactory net.laaber.mt.evaluation.sensor.
   DslSensorFactory

task sprint {
  javaIn id : Integer
  javaIn name : String

  acquire scrum.obj.Sprint with agilefantRd.agilefant(
     name, id) as s

  out s as "sprint"
}

task createChatsForStoriesOfSprint {
  in s : scrum.obj.Sprint

  stopScope

  startingFrom s load scrum.obj.Story:scrum.obj.
     containsStory as stories

  var chats : [scrum.obj.Chat]
  var invites : [scrum.links.invite]
  for all stories as story counter i {
    acquire scrum.obj.Chat with atlassianRd.hipChat("
       StoryChat" + i, "StoryChat" + i) as c
    add c to chats

    startingFrom story via scrum.col.AgileUser:scrum.
       links.responsible load scrum.col.DevUser:scrum.
```

```
      col.devUser as users
    for all users as u {
      link u and c by scrum.links.invite as invite
      add invite to invites
    }
  }

  startScope

  out chats as "chats"
  out invites as "invites"
}

task releaseChatsAndInvites {
  in chats : [scrum.obj.Chat]
  in invites : [scrum.links.invite]

  stopScope

  for all invites as i {
    unlink i
  }

  startScope

  for all chats as c {
    release c
  }
}

task sprintRetrospectiveSetUp {
  in sprint : scrum.obj.Sprint
  javaIn sprintNumber : Integer
  javaIn previousSprintNumber : Integer

  stopScope

  acquire scrum.obj.Wiki with atlassianRd.bitbucket("
     SprintWiki" + sprintNumber, "SprintWiki" +
     sprintNumber) as currentWiki
  acquire scrum.obj.Wiki with atlassianRd.bitbucket("
     SprintWiki" + previousSprintNumber, "SprintWiki" +
     previousSprintNumber) as previousWiki
  reference from currentWiki to previousWiki with scrum.
     obj.prev as p1
  reference from previousWiki to currentWiki with scrum.
     obj.next as p2

  startScope

  acquire scrum.obj.Chat with atlassianRd.hipChat("
     RetrospectiveChat" + sprintNumber, "
     RetrospectiveChat" + sprintNumber) as chat
  startingFrom sprint via scrum.obj.Story:scrum.obj.
     containsStory via scrum.col.AgileUser:scrum.links.
     responsible load scrum.col.DevUser:scrum.col.
     devUser as users

  stopScope

  var invites : [scrum.links.invite]
  for all users as u {
    link u and chat by scrum.links.invite as i
    add i to invites
  }

  startScope

  release previousWiki

  out chat as "chat"
  out currentWiki as "wiki"
  out invites as "invites"
}

task sprintRetrospectiveTearDown {
  in chat : scrum.obj.Chat
  in invites : [scrum.links.invite]

  stopScope

  for all invites as i {
    unlink i
  }

  startScope

  release chat
}
```