FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Ansatz zur Analyse der Auswirkung von kollaborativem Nutzerverhalten mittels Simulation von Softwarearchitekturen

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering/Internet Computing

eingereicht von

BSc Michael Vodep
Matrikelnummer 0530311

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dr. Schahram Dustdar
Mitwirkung: Univ.Ass. Dr. Christoph Mayr-Dorn

Wien, 27. Oktober 2015

Michael Vodep	Schahram Dustdar

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

An approach to simulating software architectures for analyzing the impact of collaborative user behaviour

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering/Internet Computing

by

BSc Michael Vodep
Registration Number 0530311

to the Faculty of Informatics

at the Vienna University of Technology

Advisor: Univ.Prof. Dr. Schahram Dustdar
Assistance: Univ.Ass. Dr. Christoph Mayr-Dorn

Vienna, 27th October, 2015

Michael Vodep

Schahram Dustdar

Erklärung zur Verfassung der Arbeit

BSc Michael Vodep
Badeggerweg 12, 8501 Lieboch

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, 27. Oktober 2015

Michael Vodep

Kurzfassung

Das Verhalten von Systemen, welche in einem sicherheitskritischen Systemumfeld die Kollaboration der Benutzer unterstützen, ist meist schwer vorhersehbar. Im Bereich Public-Safety erfolgt die Kollaboration u.a. über ein Einsatzleitsystem. Einsatzleitsysteme dienen in erster Linie zum Informationsaustausch und in weiterer Folge zur Koordination zwischen Disponenten. Neben der hohen Softwarevariabilität (getrieben primär durch funktionale Anforderungen), hat ein Einsatzleitsystem auch viele **nicht-funktionale Anforderungen** wie u.a. Verfügbarkeit, Skalierbarkeit und niedrige Latenzzeiten. Es ist oft nicht ohne weiteres möglich, Aussagen über die Einhaltung dieser Anforderungen zu treffen. Neben den technischen Aspekten stellt sich auch die Frage nach **optimalen Arbeits-Workflows** zwischen Notrufern (e.g. verbalen Notrufer), Disponenten und Ressourcen (e.g. Einsatzleiter). Zusammengefasst: Die Kollaborationsanalyse und Simulation kann für technische als auch für organisatorische Fragestellungen in einer Leitstelle angewandt werden.

Eine Lösung, um das reale Arbeitsumfeld abbilden und dokumentieren zu können, sind Modelle. Modelle können in eine ausführbare Simulation übergeführt werden, um das Verhalten und weitere Eigenschaften des Systems vorhersagen zu können, ohne einen kostspieligen Prototyp bauen zu müssen. Ein Ansatz, um kollaborative Umgebungen abbilden zu können, wurde von Dorn, Dustdar und Osterweil diskutiert. Dabei kamen die Sprachen hADL und LittleJIL zum Einsatz. Ziel der vorliegenden Arbeit ist es, Modelle, welche in hADL und LittleJIL beschrieben wurden, in eine ausführbare Simulation überzuführen.

Die Umsetzung erfolgte in mehreren Schritten, da das gewählte Simulationstool nur generische Hilfsmittel zur Simulations-Modellierung zur Verfügung stellt. Es wurden initial Umsetzungsmuster festgelegt, welche Elemente des Ursprungsmodells (Human Architecture Description Language (hADL) und LittleJIL) auf Elemente des Ziel-Modells beschreiben. Das umgesetzte Feature-Set wurde Eingangs priorisiert - im Zuge der Arbeit wurde ein Teil des Sets umgesetzt. Das in der Arbeit angewandte, exemplarische Szenario beschäftigt sich mit Großschadensereignissen, bei denen selbe Vorkommnisse mehrfach gemeldet werden und eine Koordination zwischen Disponenten erfordert, um eine Doppeldisposition zu vermeiden.

Die Ausgabe des Transformationstools kann von einem Entwickler in DomainPro Designer bearbeitet und verfeinert werden. Der automatisch generierte Code ist dabei eine minimale

Implementierung, welche als Grundlage für weitere Änderungen dient und kann vom Entwickler beliebig angepasst werden. Das im Zuge der Arbeit betrachtete Fallbeispiel konnte mit Hilfe des Transformationstools erfolgreich umgesetzt werden.

Abstract

The behavior of systems, which support the collaboration of users in a safety critical system environment, are mostly hard to predict. In the area of public safety, collaboration happens via an incident management system. Incident management systems primarily serve the purpose of information exchange and coordination between dispatchers. In addition to high software variability (primarily driven by functional requirements), an incident management system has a variety of non-functional requirements, such as availability, scalability and low latency. It is not necessarily possible to make statements about the compliance of such requirements. Besides the technical aspects, there is also the question of what is the optimal workflow between emergency callers (e.g. verbal emergency callers), dispatchers and resources (e.g. incident commander). In summary, the analysis of collaboration and simulation can be applied to technical and non-technical questions in an emergency call center.

Models are a solution for representing and documenting real working environments. They can be transformed into an executable simulation to predict the behavior and additional properties of the system without the need to build costly prototypes. An approach to representing collaborative environments is discussed by Dorn, Dustdar and Osterweil. Thereby the languages hADL and LittleJIL are used. The goal of the current thesis is to transform models described in hADL and LittleJIL into a single executable simulation.

Implementation is done in several steps due to the chosen simulation tool only providing generic simulation primitives. Patterns, which describe which elements of the source model (hADL and LittleJIL) map to the target model, were selected at the beginning. The overal feature set was prioritized, as in the end this thesis implemented only the most important ones. The case study, which is applied to the implementation, deals with major catastrophic events. During a major catastrophic event, the occurrence is reported several times to the emergency call center, a process which requires coordination between dispatchers to prevent a double dispatch.

The output of the transformation can be edited and refined by a developer in DomainPro Designer. The generated code is a minimal implementation, which serves as a base for further changes and can be modified arbitrarily by the developer. The case study can be successfully implemented with the help of the transformation tool.

Inhaltsverzeichnis

KAPITEL 1

Problemstellung

Das Laufzeitverhalten von Systemen, welche in einem sicherheitskritischen Systemumfeld die Kollaboration der Benutzer unterstützen, ist meist schwer vorhersehbar. Im Bereich Public-Safety erfolgt die Kollaboration u.a. über ein Einsatzleitsystem. Einsatzleitsysteme dienen zum Informationsaustausch zwischen Disponenten. Sie haben dabei u.a. folgende Aufgaben:

- Aktuelle Einsätze bzw. bereits abgeschlossene Einsätze verwalten und dem Disponenten eine Gesamtübersicht geben
- Ressourcenverwaltung (e.g. Fahrzeuge, Personen)
- Dokumentation

Neben der hohen Softwarevariabilität (primär getrieben durch funktionale Anforderungen), hat ein Einsatzleitsystem auch viele nicht-funktionale Anforderungen wie u.a. Ausfallsicherheit, Skalierbarkeit und niedrige Latenzzeiten. Diverse Softwaresystemhäuser haben bereits unterschiedliche Architekturen entworfen, um den Anforderungen gerecht zu werden. Beim Entwurf, als auch während der Entwicklung, ist es daher von großem Interesse, Qualitätsattribute der Architektur ständig messen zu können, um das Verhalten im Feldbetrieb bestmöglich vorhersagen zu können.

Beschreibt man nicht-funktionale Anforderungen aus Sicht **eines einzigen Disponenten**, so kann es im Feldbetrieb zu unerwarteten Szenarien kommen, in denen es durch kollaboratives Zusammenarbeiten von mehreren Disponenten zu unvorhersehbaren Verhalten des Systems kommt. Schränkt das Verhalten die Arbeit der Disponenten ein, so ist das im Bereich Public-Safety nicht akzeptabel, da es die Arbeitsqualität beeinträchtigt und somit in weiterer Folge den hilfesuchenden Personen Nachteile bringt (e.g. längere Wartezeiten, was gesundheitsschädigende Folgen haben kann). Ziel sollte es sein, dass ein

Einsatzleitsystem ein wohldefiniertes Verhalten in allen erdenklichen Szenarien hat, in denen Disponenten zusammenarbeiten.

Neben den technischen Aspekten sind auch **Optimierungen im Arbeitsfluss** von Interesse. In Public-Safety ist einer der herausforderndsten Tätigkeiten jene des Großschadensereignisses. Eine hohe koordinative Interaktion zwischen Disponenten ist in diesem Fall erforderlich. Dabei ist von Interesse, welche Informationen ein Disponent von anderen benötigt, um seine Arbeit zu vollrichten bzw. wie der Ablauf im System durch Zuweisung von Rollen (e.g. Calltaker und Dispatcher) optimiert werden kann.

Die Beschreibung der Umgebung ist facettenreich. U.a. gibt es diverse Bemühungen, Beschreibungen von Szenarien in natürlicher Sprache (e.g. Gherkin Notation), ausführbar zu machen (Natural Language Processing). Durch den Hype von Behavior Driven Development (BDD) sind eine Vielzahl an Tools entstanden, welche es erlauben, Akzeptanzkriterien in natürlicher Sprache zu formulieren und anschließend auszuführen.

Mit Modellen geht man einen Schritt weiter. Man beschreibt die Entitäten (e.g. Disponenten) und deren koordinative Wechselwirkung. Die Modelle können anschließend in eine ausführbare Simulation transformiert und ausgeführt werden. Das Einsatzleitsystem kann als Black-Box betrachtet werden oder es können Teile des Systems modelliert und simuliert werden, wenn diese für die kollaborative Arbeit von Interesse sind (wie Eingangs erwähnt e.g. nicht-funktionale Anforderungen eines Einsatzleitsystems). Ein Einsatzleitsystem kommuniziert mit vielen externen Systemen (e.g. Sprachvermittlungssystem, Hausautomation). Auch kann der Einfluss der koordinativen Arbeit der Disponenten und die Auswirkung auf externe Systeme betrachtet werden. Ebenfalls von Interesse könnten auch Fehlerszenarien in einer kollaborativen Umgebung sein (e.g. Teilausfälle von technischen Komponenten und die Auswirkung auf die Arbeit der Disponenten). Für all die Fragestellungen ist es ratsam, Teile des Systems oder das ganze System zu modellieren und anschließend zu simulieren.

1.1 Introduction

Waren es Anfang der 80iger Jahre noch wenige hundert Computer, die vernetzt waren, sind es heute fast 1 Milliarde Computer, die übers Internet verbunden sind. Plattformen wie Office-Online[1] oder Wikipedia erlauben es dem Benutzer, mit anderen Benutzern gemeinsam Aufgaben zu erledigen. Doch nicht nur im Internet findet eine Kollaboration von Benutzern statt. Auch andere Industriezweige haben den Vorteil des "vernetzt seinërkannt. Hier bedient man sich meist abgeschotteter Netzwerke, wie einem Local Area Network (LAN) oder Wide Area Network (WAN).

Eines der Kernaufgaben eines Computersystems ist die Verarbeitung und Verteilung von Informationen. Der Benutzer (oder andere externe Systeme) kann auf Daten reagieren und dadurch neue Daten generieren, welche wiederum von einem anderen Benutzer wahrgenommen werden. Diese computergestützte Art des Arbeitens - besser bekannt als

[1] http://en.wikipedia.org/wiki/Office_Online

Koordination (Coordination) - ist nichts Neues und wurde schon vor fast zwei Jahrzehnten u.a. von Malone und Crowston untersucht [MC94]. Sie definierten den Begriff Coordination wie folgt:

> Coordination is managing dependencies between activities.

Als Grundlage der Forschung dienten Ideen aus den Bereichen Computertechnologie, Organisationslehre, Wirtschaft, Sprache und Psychologie, wo man ähnliche Ansätze wahrgenommen hat. Die daraus gewonnenen Erkenntnisse werden im Laufe der Arbeit noch genauer beleuchtet.

Neben der Koordination von Aktivitäten, gibt es auch die Kollaboration (Collabaoration), welche sich als Ziel setzt, mehrere Parteien gemeinsam zu einem Ziel zu verhelfen. Schon in den 90iger Jahren sind einige Arbeiten [LRS99] über *human-computer interaction* zu finden, bei denen der Benutzer seine Aktionen mit einem Computer (meist auch Agent genannt) koordiniert, um ein gemeinsames Ziel zu erreichen (Kollaboration). Durch den hohen Vernetzungsgrad ist es heute gang und gäbe, dass die Kollaboration auch über die Computergrenzen hinweg durchgeführt wird. Benutzer versuchen über vernetzte Computer gemeinsam eine Aufgabe zu lösen. Die Arbeit beschäftigt sich mit einer speziellen kollaborativen Domäne: einer Notrufzentrale, welche ein Einsatzleitsystem zur Koordination von Disponenten verwendet. Um den Leser ein besseres Verständnis über die Domäne zu geben, wird diese kurz vorgestellt.

1.2 Motivating Scenario - Domäne Feuerwehr Einsatzleitsystem

Ein Einsatzleitsystem ist eine Software, welche in Leitstellen (Polizei, Feuerwehr, Rettung, ÖAMTC, ...) u.a. zur Koordination von Ressourcen und zur Dokumentation verwendet wird. Ein Einsatzleitsystem interagiert in den meisten Fällen mit externen Systemen und dient als Informationsquelle für den Bediener. In den genannten Organisationen bzw. Firmen wird eine Arbeitsabfolge durch ein Ereignis ausgelöst (e.g. Verkehrsunfall, Autopanne, Banküberfall, ...). Durch das Absetzen einer Benachrichtigung an die Organisation bzw. Firma wird eine interne Arbeitsabfolge gestartet. Die gängigsten Benachrichtigungstypen sind:

- verbale Verständigung per Notruf 122 / 133 / 144 / ...
- Datenübertragung per Druckknopfmelder, Rufhilfe Armband, eCall in PKWs, Gehörlosenfax, ...

Der Notruf wird meist von einem Disponenten (engl. Dispatcher) entgegengenommen und bearbeitet. Dies kann entweder manuell durch Eingabe der Informationen in ein

Einsatzleitsystem (engl. Computer Aided Dispatch System (CAD)) passieren, oder durch automatische Annahme der Daten über das Einsatzleitsystem, wie es bei Benachrichtigungen des Typs *Datenübertragung* der Fall ist.

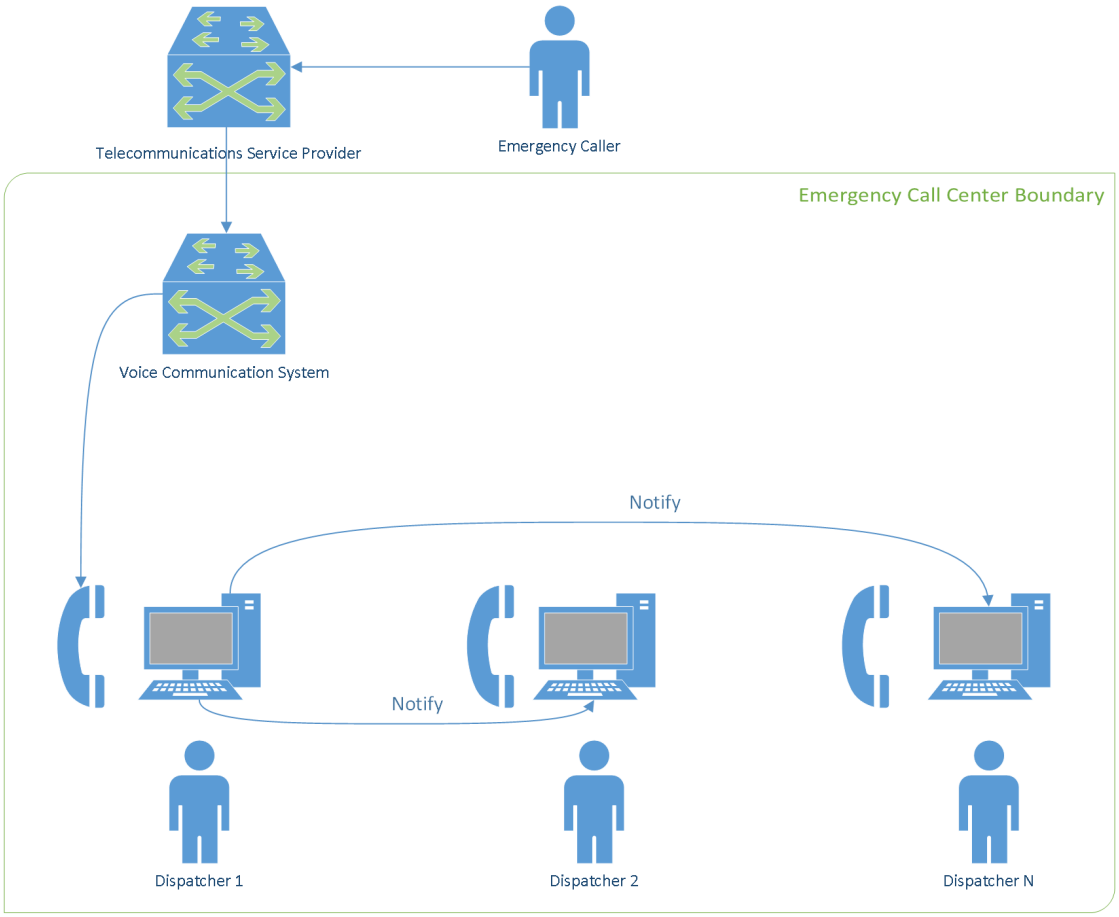

Abbildung 1.1: Ein Beispiel für verbale Verständigung per Notruf

Abbildung 1.1 zeigt grob den Ablauf bei der Bearbeitung eines Schadensereignisses. Folgende Schritte sind dabei essentiell:

1. Ein Schadensereignis ereignet sich (e.g. ein Verkehrsunfall). Ein Notrufer (Emergency Caller, allgemein: Annunciator) meldet das Ereignis über den Notruf 122.

2. Der Telekommunikations Service Provider (TSP) entscheidet anhand der Lokalität des Notrufers zu welcher Leitstelle der Notrufer weitergeleitet wird.

3. Ein Sprachvermittlungssystem (Voice Communication System (VCS)) nimmt den Anruf entgegen und verteilt ihn - geregelt durch Policies - an einen oder mehrere Disponenten. Der Anruf wird von einem Disponenten angenommen.

4. Dieser trägt den Schadensfall in das Einsatzleitsystem ein.
5. Das Einsatzleitsystem schlägt dem Disponenten aufgrund des globalen Status (u.a. verfügbare Ressourcen) weitere Schritte vor.

Die Erfahrung zeigt, dass ein Schadensfall oft von verschiedenen Personen gemeldet wird (e.g. wenn Personen bei einem Verkehrsunfall vorbeifahren und diesen über Notruf melden). Somit ist folgendes Szenario denkbar:

1. Ein Verkehrsunfall Höhe Ketzergasse 473, 1230 Wien passiert.
2. Ein Anwohner (Notrufer 1) meldet das Ereignis über 122.
3. Das Voice-Communication-System leitet den Notruf an Disponent 1 weiter.
   a) Disponent 1 legt den Einsatz an.
4. Ein Fußgänger leistet erste Hilfe und meldet das Ereignis ebenfalls über 122.
5. Das Voice-Communication-System leitet den Notruf an Disponent 2 weiter.
   a) Disponent 2 legt den Einsatz an.

Würde das Einsatzleitsystem diesen Missstand nicht erkennen, würde Disponent 1 als auch Disponent 2 jeweils Ressourcen zum gleichen Einsatzort disponieren. Im Idealfall findet eine Koordination durch das Einsatzleitsystem statt. Betrachtet man die Gesamtsituation, so findet zwischen Notrufer 1 und 2 und den Einsatzkräften (Rettungswagen, Feuerwehrfahrzeugen) eine Kollaboration statt - mit dem Ziel, den verunfallten Personen zu helfen. Das soeben exemplarisch gezeigte Kollaborations-Szenario bildet - neben vielen anderen ähnlichen Szenarien - die Grundlage an Anforderungen für ein Einsatzleitsystem. Ziel ist es, ein besseres Gesamtverständnis über das System zu erlangen, in dem man nicht nur die Sicht eines Disponenten beschreibt, sondern deren Kollaboration. Im Zuge der Arbeit werden noch weitere Szenarien genauer beleuchtet, um die Problematik zu verfeinern.

> Die Betrachtung der Kollaboration erhöht das Gesamtverständnis über das System für alle Beteiligten und erlaubt Rückschlüsse durch Simulation zu tätigen.

1.2.1 Die Software Architektur

Eine Architektur hat Einflüsse auf die Qualitätsattribute, welche durch e.g. Sicherheit, Änderbarkeit wahrgenommen werden. Die Wahl einer guten Architektur kann das Erreichen der Qualitätsattribute vereinfachen. Die Frage nach "Wie viel Software Architektur ist genug?"betrifft jedes Softwareprojekt. Sie wird nicht in jedem Projekt gestellt -

wenngleich sie auch immer präsent ist. Das Spektrum der Möglichkeiten weist zwei Extreme auf: man verzichtet zur Gänze auf architekturelle Planung oder man plant alles bis ins letzte Detail. Fairbanks [Fai10] kategorisiert das Spektrum in drei Abschnitte:

- **Architecture-Indifferent Design**: In diesem Ansatz wird der Architektur nur wenig Aufmerksamkeit geschenkt. Eine Ursache kann sein, dass Entwickler die Architektur ignorieren und einfach von einem vorherigen Projekt übernehmen oder eine in der Domäne übliche Architektur anwenden (Presumptive Architecture)
- **Architecture-Focused Design**: Hier wählt und entwirft man die Architektur bewusst. Die Architektur wird so gestaltet, dass sie die gesetzten Ziele erfüllt und die funktionalen und nicht-funktionalen Attribute erfüllt. Der Architektur Aufmerksamkeit schenken, bedeutet nicht zwangsläufig, diese zu dokumentieren - wenngleich Dokumentation in großen Teams eine Hilfe sein kann. Auch kann der Code vom Architektur-Design abweichen.
- **Architecture Hoisting**: Hier fokussiert man sich auf die Architektur, um ein Ziel (Goal) oder eine Eigenschaft des Systems zu garantieren (gleich wie bei Architecture-Focused Design). Allerdings werden die Eigenschaften des Architektur-Designs garantiert (d.h. modellierte Eigenschaften finden eine Abbildung im Code - der Entwickler muss keinen zusätzlichen Code dafür schreiben).

Fairbanks [Fai10] nutzt einen Risk-Driven-Approach, bei dem der Entwurfsaufwand der Architektur anhand der Risiken festgelegt wird. Der Ansatz schlägt vor, zuerst Risiken zu identifizieren und priorisieren. Wurde dies gemacht, können geeignete Techniken ausgewählt und angewandt werden.

Die hier angewandte Methode ist Architecture Hoisting. Dabei werden folgende Ziele verfolgt:

- Die entworfene Architektur in einem kollaborativen Umfeld durch Simulation verstehen (Dokumentation). Die Architektur umfasst in diesem Kontext u.a. auch menschliche Entitäten und weitere Entitäten, die zum Erreichen des kollaborativen Ziels notwendig sind.
- Offene Fragen durch Simulation beantworten
  - Fragen im nicht-technischen Kontext (e.g. "Welche Auswirkung hat die Separierung der Rollen Calltaker und Dispatcher bei geringem Einsatzaufkommen?")
  - Fragen im technischen Kontext (e.g. "Wie hoch ist der Gewinn durch Caching?")
- Ebenfalls ist die Generierung eines Code-Frameworks aus der Architektur-Beschreibung (umgesetzt mit einer Architecture Description Language (ADL)) möglich - wird in der Arbeit aber nicht weiter verfolgt.

1.2.2 Anforderungen dokumentieren und simulieren

Um größere Systeme wie ein Einsatzleitsystem testen (und dokumentieren) zu können, gibt es unterschiedlichste Ansätze. U.a. empfiehlt es sich, Anforderungen automatisiert zu verifizieren. Eine mögliche Technik ist unter dem Namen *Specification by example*[2] (oder auch als Behavior Driven Development (BDD)) bekannt. Ziel ist es, Anforderungen in natürlichsprachlicher Form zu beschreiben, welche anschließend ausführbar sind. Man spricht von einer Living Documentation - diese kann von Nicht-Entwicklern gelesen werden und hat den Vorteil, dass sie (im optimalen Fall) immer aktuell ist. Der Anwendungsbereich ist sehr facettenreich. Man kann funktionale und nicht-funktionale Anforderungen beschreiben, Anforderungen aus Sicht eines Disponenten oder aus Sicht der ganzen Notrufkette.

Neben der textuellen Beschreibung gibt es auch noch die Möglichkeit, aus Modellen, ausführbaren (Test-)Code zu erzeugen. Diese Vorgehensweise ist unter Model Based Software Development (MBSD) bekannt und wurde u.a. von Zheng und Taylor [ZT13] beschrieben und in mehrere Kategorien unterteilt. Dabei sind folgende beiden Methoden von Interesse:

- **Architecture-Centric Development**: Die Granularität der Modellierung beschränkt sich auf die Architektur. Aus dem Modell kann ein Code-Skeleton generiert werden, welches von Entwicklern mit Details versehen werden kann. UML ist eine mögliche Modellierungssprache - aber auch domainspezifische Sprachen sind möglich.
- **Generative and componentbased development**: Diese Modelle erlauben es, die Verbindungen zwischen Komponenten zu modellieren und anschließend den Glue-Code zu generieren. Es erfolgt eine Komposition von Komponenten.

Die Ideen der Modellierung wurden bereits von Dorn und Taylor aufgegriffen und es wurden die domainspezifischen Sprachen hADL (Dorn) [DT13] bzw. LittleJIL (Taylor) [CLM+00] entworfen. Diese Sprachen wurden mit dem Designziel entwickelt, Umgebungen und darin getätigte Abläufe in einem Modell zu beschreiben. Dadurch wäre es möglich, die eingangs erwähnte Kollaboration zwischen Notrufer, Disponenten und Einsatzkräfte zu modellieren und anschließend zu simulieren. Dies hat den Vorteil, dass die Kollaborations-Modelle auch von Nicht-Entwicklern gelesen werden können und dies somit eine gute Dokumentations-Basis bildet. Ebenfalls ermöglicht es die Generierung eines Code-Skeletons (wie es beim Typ Architecture-Centric Development der Fall ist). Dieses Skeleton dient als Grundlage für die Simulation. Diese Idee wird im Zuge der Arbeit aufgegriffen, umgesetzt und analysiert.

1.2.3 Vorteile Anwendungsbereiche der Simulation

Seit Agile Entwicklungsmethoden immer mehr an Beliebtheit gewinnen, wurden Methoden wie Big Design Up Front (BDUF) oder Evolutionäre Architektur immer öfters diskutiert.

[2] `http://martinfowler.com/bliki/SpecificationByExample.html`

Dabei geht es um den Umfang des Architektur-Designs in einem Softwareprojekt. Das Meinungsspektrum ist facettenreich und wird in der Arbeit nicht genauer beleuchtet. Analysen wurden bereits von Kruchten et al. [ABK10] durchgeführt. Die Analysen zeigten, dass architekturelle Entscheidungen im Kontrast zu Software-Designentscheidung meist gering ausfallen. Die Menge an architekturellen Entscheidungen ist je nach Softwareprodukt unterschiedlich. Architekturelle Entscheidungen sollten aber in einer frühen Phase getroffen werden [ABK10], da diese meist schwer zu ändern und Änderungen mit Kosten verbunden sind. Simulation erlaubt es, wichtige architekturelle Entscheidungen zu überprüfen. Auch kann Simulation als eine Art Dokumentation gesehen werden, welche ein *Common-Understanding* über die Funktionalität bzw. Designentscheidungen gewährleistet.

Betrachtet man den Einsatz von Simulation in einem frühen Projektstadium, so kristallisieren sich folgende Vorteile für ein Einsatzleitsystem heraus:

- Kollaborative Szenarien können dokumentiert und simuliert werden. Aus den Modellen kann Code generiert werden, mit dem Szenarien ständig ausgeführt werden können (e.g. für Regressions Tests). Die Dokumentation von Anforderungen (engl. Requirements) aus Benutzersicht ist essentiell, da sie über Design-Evolutionen hinweg gültig ist.

- Durch Simulation können nicht-funktionale Eigenschaften einer Architektur überprüft werden.

Die Erzeugung einer Simulations-Umgebung zum Zwecke der Evaluierung von Anforderungen ist einer der Schwerpunkte der Arbeit. Bildet der Software-Architekt End-to-End Szenarien (Notrufer bis Einsatzkräfte ab), so kann durch Simulation festgestellt werden,

- welche Daten
- durch welche Aktion im System
- welchen Disponenten
- zu welchem Zeitpunkt

zur Verfügung stehen. Stellt man fest, dass das Ergebnis unbefriedigend ist, so kann die Architektur schrittweise verbessert werden. Würde die selbe Erkenntnis erst in einem sehr späten Projektstadium erlangt werden, so wären die Änderungen nur mit einem hohen (finanziellen) Aufwand durchführbar.

1.3 Challenges

Die in der Arbeit verwendeten Sprachen zum Modellieren der Umgebung ist hADL (Human Architecture Description Language) bzw. LittleJIL für die Modellierung der Arbeitsabfolge-Hierachien. Die Synergien dieser beiden Sprachen wurde bereits untersucht [DDO14]. Die Herausforderung besteht in der Überführung in eine DomainPro-Struktur. Im DomainPro Designer können Software-Architekten anschließend Implementierungsdetails zur Simulation der Umgebung bzw. des Systems hinzufügen und anschließend simulieren (DomainPro Analyst).

Ziel der Arbeit ist es, die Disponenten-Kollaboration in den Modellierungssprachen hADL (Modellierung der Struktur) und LittleJIL (Modellierung des Verhaltens) zu modellieren und anschließend in eine ausführbare Simulation zu transformieren. Es soll festgestellt werden, ob die daraus entstandene Simulation die erwarteten Aspekte zufriedenstellend abdecken kann.

1.4 Herangehensweise

Um den Funktionsumfang des Tools, welches aus hADL und LittleJIL ein DomainPro Modell generiert, festlegen zu können, ist es erforderlich, die Eigenschaften eines Einsatzleitsystems zu analysieren. Sind die Eigenschaften eines Einsatzleitsystems bekannt, können Szenarien definiert werden, welche in weiterer Folge modelliert werden. Betrachtet werden ausschließlich kollaborative Szenarien. Um die Interaktion mit dem Einsatzleitsystem zu ermöglichen, wird dieses teilweise ebenfalls modelliert. Die Simulation des Einsatzleitsystems ist allerdings nicht Gegenstand der Arbeit und wird daher nur rudimentär durchgeführt.

Sind die Modelle vorhanden, welche die Szenarien architekturell wiederspiegeln, kann das Tool zur Generierung der DomainPro Modelle entwickelt werden. Durch dieses ist die Transformation von hADL bzw. LittleJIL in eine DomainPro Simulation möglich, in der anschließend der Glue-Code (Szenariendetails) entwickelt werden können. Die Evaluierung, ob die Szenarien ausreichend beschrieben werden konnten, ist der letzte Teil des Zyklus.

Die angeführten Schritte werden iterativ anhand der Beispiel-Domäne durchgeführt, um die Transformation von einem Modell in eine Simulation sukzessive verbessern zu können. Dabei wird in jedem Iterationsschritt entschieden, ob die Input Modelle (hADL und LittleJIL) angepasst oder Features zum Tool, welches das Simulations-Skeleton generiert, hinzugefügt werden müssen.

KAPITEL 2

State of the art

Dieses Kapitel befasst sich mit aktuellen Methodiken für die Lösung des Eingangs beschriebenen Problems. Dabei sind die Themenschwerpunkte wie folgt:

- Modellierung
- Synthetisieren und Interpretation
- Simulation
- Softwarearchitektur

2.1 Model Based Software Development

> Model-based engineering is the creation and analysis of models of your system such that you can **predict and understand** its capabilities and operational quality attributes (e.g., its performance, reliability, or security). ... Analyzable models provide a foundation for prediction and design assurance by enabling meaningful assessment and reasoning about their properties and the properties of the systems they represent. Model properties are addressed in the development process and focus on quality attributes such as modifiability and maintainability, while system properties address operational quality attributes such as performance, safety, and reliability. [FG12]

MBSD ist ein Überbegriff für modellgetriebene Softwareentwicklung. Bei einem Software-Model handelt es sich um eine Abstraktion von einem Teilaspekt eines Softwareprodukts [ZT13], welches u.a. die Aufgabe der Dokumentation hat. Neben den klassischen Anwendungsgebieten gibt es bei MBSD auch die Ausprägungen der Source-Code Synthetisierung

[ZT13] - bekannt unter Model-Driven Engineering (MDE). Dabei ist es möglich, viel näher an der Problemdomäne zu entwickeln und entwicklungsspezifische Details zu verstecken. Taylor und Zheng haben die unterschiedlichen Methodiken in vier Klassen eingeteilt [ZT13]:

- Specification-driven development
- Model-driven development
- Architecture-centric development
- Generative and componentbased development

Bei *Specification-driven development* erfolgt eine vollständige Codegenerierung aus den Anforderungen (Requirements). *Model-driven development* generiert ebenfalls vollständig den Code, stützt sich aber auf *Software Design Models* anstatt auf Anforderungen. UML wäre ein Beispiel für die Modellierung in *Architecture-centric development* oder *Model-driven development*. Bei *Generative and componentbased development* geht es in erster Linien um die Generierung von Glue-Code zwischen Komponenten, welcher in einer Composition-Specifications beschrieben wird.

2.2 Model-Driven Engineering

MDE generiert aus abstrakten Modellen von Software konkrete Implementierungen [FR07]. Die Motivation hinter Model-Driven Engineering ist die immer komplexer werdende Software - es wird versucht, die Domäne und Implementierung so nahe wie möglich zueinander zu bringen. Dabei werden Modelle unterschiedlicher Sichtweisen genutzt, um das System zu beschreiben. MDE beschränkt sich nicht nur auf die Nutzung gängiger Modellierungssprachen wie Unified Modeling Language (UML) - auch Sprachen wie Aloi[1] oder Simulink[2] sind für die Umsetzung möglich. Entwickeln der Software in einer Programmiersprache kann ebenfalls als Modellierung gesehen werden.

2.2.1 Zusammenhang zwischen Model Based Software Development und Model-Driven Engineering

> Finally, we use "model-based engineering"(or "model-based development") to refer to a softer version of MDE. That is, the MBE process is a process in which software models play an important role although they are not necessarily the key artifacts of the development (i.e., they do NOT "drive"the process as in MDE). An example would be a development process where, in the analysis phase, designers specify the domain models of the system but

[1] `http://alloy.mit.edu/alloy/`
[2] `http://de.mathworks.com/products/simulink/`

> then these models are directly handed out to the programmers as blueprints to manually write the code (no automatic code-generation involved and no explicit definition of any platform-specific model). In this process, models still play an important role but are not the central artifacts of the development process and may be less complete (i.e., they can be used more as blueprints or sketches of the system) than those in a MDD approach. MBE is a superset of MDE. All model-driven processes are model-based, but not the other way round. [BCW12]

Brambilla, Cabot und Wimmer [BCW12] sehen *Model-Based* Ansätze als relaxte Form von *Model-Driven* Ansätzen, bei denen keine Code-Generierung erfolgt. Diese Definition deckt sich auch mit jener von Zhend und Taylor [ZT13], welche Model-Driven Development auch als eine Ausprägung von Model-Based Software-Development sehen.

2.3 Architecture-centric software development

Hier ist die Software-Architektur der Kernaspekt der Modellierung [ZT13]. Es gibt unterschiedliche Ausprägungen (nicht vollständig):

2.3.1 Style-Based Architecture Refinement [MQR95]

Eine Refinement-Pattern besteht dabei aus einer Standardroutine, welche einen geprüften Ansatz für ein Standard-Architekturproblem darstellt. Eine Software-Architektur besteht dabei aus folgenden Konzepten [MQR95]:

- **Component**: Software, welche über ein Interface Service-Funktionalitäten anbietet [Nie05]. E.g. Module, Prozesse.
- **Interface**: Ein wohldefinierter Berührungspunkt für die Kommunikation mit anderen Komponenten oder der Umgebung.
- **Connector**: Verbindung zwischen Interfaces und Components.
- **Configuration**: Eine Menge an Constraints, welche Objekte in einer spezifischen Architektur verbinden.
- **Mapping**: Beschreibt die Umwandlung von einer abstrakten Architektur in eine konkrete.
- **Architectural style**: Besteht u.a. aus Design-Elementen, Constraints.

Ziel des Refinement ist es, die Architektur - bestehend aus den oben aufgezählten Konzepten - in eine konkrete Architektur, in kleinen Schritten, überzuführen. Eine textuelle Beschreibung könnte wie in Abbildung 2.1 aussehen [MQR95].

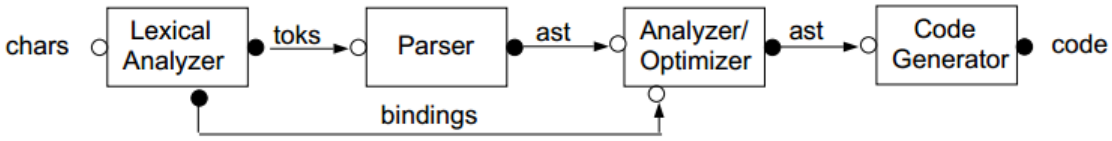

Abbildung 2.1: Beispiel einer Architektur

Als Rechteck dargestellt werden Funktionale-Komponenten (Components). Die kleinen Kreise kennzeichnen Input- und Output-Ports, zwischen denen Daten fließen können (über Connectors). Daten, welche zwischen Ports übertragen werden sollen, müssen Typ-Kompatibel sein. Wie die Verbindungen (Connectors) umgesetzt sind, wird im Beispiel nicht spezifiziert.

2.3.2 Framework und Middleware-Based Development [MOT97]

Hierbei handelt es sich um einen Architecture-Style zum Wiederverwenden von Software-Komponenten. Komponenten sind von Komponenten niedriger Kategorie nicht abhängig. Diese teilen Komponenten höherer Hierarchie über Notifications Aktivitäten und Ereignisse mit. Ziel ist es, Off-The-Shelf (OTS) Softwarekomponenten wieder zu verwenden. Der Style ist unter Chiron-2 oder C2 bekannt [TMA+96] und basiert auf den Konzepten von Komponenten bzw. Messages (siehe Abbildung 2.2). Mehrere Komponenten sind über Connectors verbunden, welche als Message-Routing Device agieren. Folgende Eigenschaften erhöhen die Wiederverwendbarkeit [TMA+96] (Prinzipien nicht nur bei C2 gültig):

- **Component Heterogeneity**: Keine Einschränkung bzgl. der Sprache, in der die Komponenten geschrieben sind. Auch die Granularität der Komponenten ist irrelevant.
- **Substrate Independence**: Komponenten wissen nichts von anderen Komponenten niedriger Hierachie und sind daher unabhängig von diesen.
- **Internal Component Architecture**: Die interne Architektur einer C2-Komponenten separiert Kommunikation von der Verarbeitung. Benachrichtigungen (Notifications) teilen anderen Komponenten interne State-Änderungen mit. Domain Translator kümmern sich dabei um die Übersetzung in die jeweilige Domain-Sprache.
- **Asynchronous Messages**: Das Verteilen der Nachrichten über Connectors geschieht asynchron.
- **Keine Annahmen über Shared Adress Space**: Komponenten können nicht annehmen, dass sie im selben Adressraum ausgeführt werden.
- **Keine Annahmen über Single Thread of Control**: Komponenten haben einen oder mehrere eigene Threads.

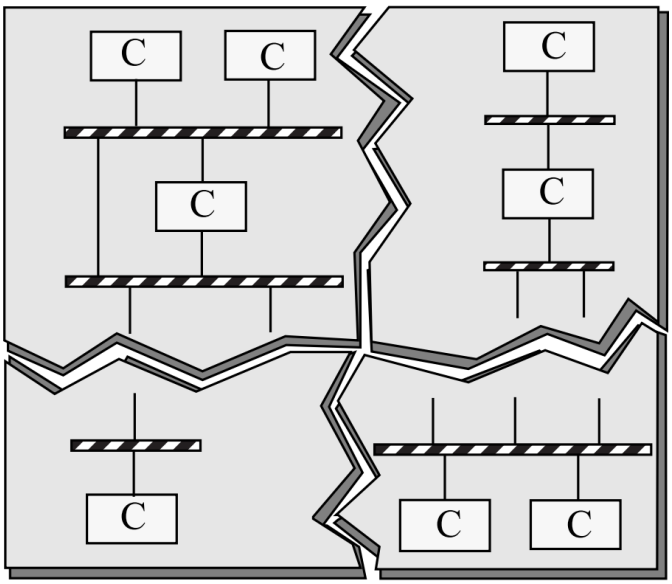

Abbildung 2.2: Der C2-Style

Architecture Refinement und Framework Techniques sind Architecture-Style spezifisch und können daher nur auf Architekturen des selben Styles angewandt werden.

2.3.3 Architecture Language Support

ADL dienen dazu, Architekturentscheidungen zu dokumentieren bzw. zu modellieren. Neben C2 gibt es auch noch viele andere Sprachen [MT00] - ein Auszug: Aesop, ArTek, Darwin, LILEANNA, MetaH, Rapide, SADL, UniCon, Weaves und Wright. Die Sprachen haben alle einen unterschiedlichen Fokus [DHT05] - so spezialisiert sich Wright e.g. auf die Komponentenkommunikation. Dies geschieht durch die Repräsentation der Kommunikation in Communicating Sequential Processes (CSP), was erlaubt, e.g. Rückschlüsse auf Deadlocks zu machen. Rapide erlaubt die Beschreibung vom Komponenten durch Partially Ordered Sets of Events (Posets) [DHT05], was für die Simulation des Systems verwendet werden kann.

Die Architecture Analysis & Design Language (AADL) ist ebenfalls ein Vertreter dieser Kategorie. Die AADL [FG12] ist ein SAE International Standard (AS5506A1). Bei AADL handelt es sich um ein Framework, welches es erlaubt, die Software-Architektur bis hin zur Computer-Plattform-Architektur - auf welcher die Software deployed wird - zu modellieren. Der Schwerpunkt der Sprache liegt auf Embedded-Systeme, die Konzepte sind aber auch auf andere Typen von Softwaresystemen anwendbar. Ein interessanter Aspekt von Embedded-Systemen ist, dass nicht nur die Modellierung der statischen Software-Architektur von Interesse ist, sondern auch die Umgebung, auf der die Software läuft. Unter anderem sind Aspekte von Interesse, wie lange eine Software für die Ausführung auf einem bestimmten Prozessor benötigt.

2.4 Gemeinsamkeiten von architekturbeschreibenden Sprachen

Betrachtet man die angeführten Modellierungssprachen für Architekturen, so kristallisieren sich Gemeinsamkeiten heraus. Medvidovic und Taylor haben diese analysiert und zusammengefasst [MT00].

- **Components**: Sind eine *Unit Of Computation* und/oder speichern Daten. Komponenten bestehen also aus Berechnungen und einem State. Komponenten können die Größe einer einzelnen Prozedur haben oder die Größe einer ganzen Applikation. Des Weiteren kann eine Komponente u.a. folgende Eigenschaften haben:
  - Interface: Das Bindeglied zwischen interner und externen Welt. Das Interface wird meist über ein oder mehrere Ports angeboten. In manchen ADL kann spezifiziert werden, ob ein Interfaces synchron oder asynchron ist. Die Kommunikation erfolgt u.a. über Messages. ADL unterscheiden typischerweise zwischen benötigten und zur Verfügung gestellten Interfaces.
  - Semantics: Sprachen wie Rapide erlauben es, das Verhalten der Komponente zu spezifizieren. Bei Rapide wird dies über partial geordnete Events umgesetzt (siehe Beschreibung oben).
  - Constraints: Neben Interfaces und Semantics können Constraints noch weitere Einschränkungen erlauben. E.g. hat C2 die Einschränkung, dass es einen Top- und Bottom-Port hat. Auch nicht funktionale Einschränkungen wie ExecutionTime, Deadline und Criticality sind möglich.
  - Nonfunctional Properties: Auch wenn viele ADLs es erlauben beliebige nichtfunktionale Eigenschaften zu spezifizieren - die Interpretation bzw. Nutzung der Information ist eingeschränkt. MetaH und UniCon interpretieren diese e.g. für das Laufzeitverhalten (u.a. bei AADL - siehe Beschreibung oben).
- **Connectors**: Connectors sorgen für die Verbindung zwischen Komponenten. Diese können implizit sein, wie Remote Procedure Call (RPC), oder explizit ähnlich einer Komponente [DHT05]. Einfache Connectors können e.g. über Messaging umgesetzt werden, komplexere e.g. durch eine Middleware. Es kann daher zu einer Vermischung der Definition zwischen Connectors und Components kommen - allgemein gilt: **Connectors können die Syntax oder Form der Daten ändern, aber nicht die Applikations-Semantik**. Ähnliche Ansätze unter unterschiedlichen Namen sind in allen ADLs vorhanden. Darwin und Rapide erlauben es, komplexes Verbindungsverhalten in eine Connector Component zu abstrahieren.
  - Interface: Die Modellierung von *Component* als auch von *Connector Interface* geschieht meist auf die gleiche Art und Weise. Connector Interfaces sind in ACME, Aesop, UniCon, und Wright e.g. Roles, welche einen Namen haben und typisiert sind. Es erfolgt anschließend in der Architectural Configuration

eine explizite Verbindung zwischen Component Ports und Roles. In C2 und Weaves können beliebige Nachrichten geschickt werden - hier geht es primär um die Vermittlung und Koordination zwischen Komponenten.

  - Semantics: Die Möglichkeiten sind hier facettenreich. Ein Beispiel wäre C2, wo Nachrichten e.g. gefiltert werden können. Rapide nutzt Posets, um die Communication-Patterns zwischen Komponenten zu beschreiben.

- **Modeling Configurations**: Hier geht es um die Beschreibung der architekturellen Struktur. Dabei werden alle Elemente (u.a. Components und Interface) der Architektur zu einer Topologie vereint, über die anschließend Rückschlüsse (per Analyse) getroffen werden können. Folgende Aspekte sind dabei von Interesse:

  - Understandable Specifications: E.g. kann explizite Konfiguration (im Kontrast zu Inline-Konfiguration) das Verständnis erleichtern. Grafische Notationen können ebenfalls das Verständnis über die Gesamtarchitektur stark vereinfachen.
  - Compositionality: Die meisten ADLs erlauben die Komposition von mehreren Komponenten. ADLs wie Darwin und UniCon haben keine speziellen Konstrukte, um eine Architektur zu modellieren. Die Architektur wird hier zur Gänze als eine Komposition aus Komponenten umgesetzt.
  - Refinement and Traceability: Die Unterstützung dieser beiden Eigenschaften ist in ADL nur gering ausgeprägt. SADL und Rapide erlauben es, durch Mapping unterschiedliche Level von Abstraktion zu erzeugen. Beide Sprachen erlauben es somit, Entscheidungen auf unterschiedlichen Design-Levels zu dokumentieren.

Sprachen wie Wright, Rapide und Darwin repräsentieren die erste Generation an ADLs [DHT05]. Diese Sprachen sind meist durch eine eigene Syntax charakterisiert und Tools, wie u.a. Compiler und Analyse-Tools.

2.5 xADL

xADL hat es sich als Ziel gesetzt, eine erweiterbare ADL zu sein [DHT05]. Dabei sollte nicht - wie es bei vielen anderen ADLs üblich ist - eine eigene Sprache designed werden, sondern man stützt sich auf vorhandene Techniken. Die Repräsentation ist in Extensible Markup Language (XML), wobei xADL die Möglichkeiten der Erweiterungen sehr stark nutzt.

Das Architektur-Modell wird dabei im xArch-Instances-Schema und xADL-2.0-Structure & Types Schema beschrieben. xADL unterscheidet zwischen Design- und Laufzeit-Modellen einer Architektur. So kann das Design-Modell Metadaten, oder Einschränkungen bzgl. der Anordnung der Elemente haben [DHT05]. Im Kontrast dazu könnte das Laufzeit-Modell die physikalische Verteilung der Komponenten beinhalten. xADL 2.0 beschreibt

einige generische Konstrukte (u.a. Components, Connectors, Interfaces) mit denen eine Architektur beschrieben werden kann. Das Verhalten von e.g. Components und Connectors wird dabei nicht genau spezifiziert. Ebenfalls gibt es ein Implementation Mappings, welche die Verbindung zwischen ausführbaren Code und ADL herstellt. xADL bietet Basisfunktionalität für Typen - Equality und Composition. Die Nutzung von Typen ist allerdings optional.

2.6 hADL

Die bisher betrachteten Sprachen wurden zur Beschreibung der Architektur verwendet. Betrachtet man die Kollaboration als Ganzes, so müssen auch Benutzer oder andere externe Komponenten betrachtet werden. Die Sprache human Architecture Description Language (hADL) wurde genau mit diesem Designkriterium entworfen [DT12]. hADL verfolgt ebenfalls das Konzept der Components und der Connectors [DDO14]. Components erfüllen die gleichen Aufgaben wie in einer ADL für Software - sie beinhalten einen Status bzw. Logik für Berechnungen [MT00]. Selbiges gilt für Connectors. Es gibt in hADL u.a. folgende Elemente:

- **HumanComponents**: Sind die klassischen Entscheidungsträger (decision maker). Sie sind der essentielle Baustein einer Architektur und nehmen eine bestimmte kollaborative Rolle ein, um das kollaborative Ziel zu erreichen [DT12]. Interfaces werden durch HumanActions repräsentiert, welche die Berechtigungen (in Form von Create, Read, Update, und Delete) angeben, welche eine HumanComponent benötigt, um ihre Aktionen auszuführen.
- **CollaborationConnectors**: In der realen Welt agieren Manager oder Teamleader als Koordinator zwischen anderen Mitarbeitern / Teilnehmern (HumanComponents). CollaborationConnectors sind somit für die effiziente und effektive Interaktion zwischen HumanComponents verantwortlich [DDO14] und entkoppeln diese. Es gibt dabei mehrere Ausprägungen: Menschliche-, Softwareunterstützte- und rein Softwarebasierte-CollaborationConnectors [DT12].

Eine Human Architecture beschreibt die Configuration der Components und Connectors. Die Architektur spiegelt dabei ein Collaboration-Pattern wieder - Beispiele hierfür sind: Master/Worker, Publish/Subscribe, Shared Artifact und Peer-to-Peer.

Ein weiterer elementarer Bestandteil von hADL sind CollaborationObjects, welche zwischen zwei oder mehreren HumanComponents und/oder CollaborationConnectors fungieren [DT12]. Dies steht im Kontrast zu ADLs, welche ComponentInterfaces und ConnectorInterfaces direkt verbinden. HumanComponents kommunizieren dabei über

- **Messages**: Es handelt sich um eine nicht veränderbare Nachricht wie e.g. ein Email.

- **Streams**: Ist eine Abfolge von Nachrichten zwischen Sender und Empfänger. Streams werden grob in zwei Subkategorien unterteilt:
  - Subscriptions-Charakter besteht aus unabhängigen Nachrichten (e.g. RSS Updates)
  - Multimedia-Streams sind Nachrichten, welche beim Empfänger als Ganzes wahrgenommen werden (e.g. Video-Chat)
- **SharedArtifacts**: Ist ein langlebiges Objekt, welches durch kollaborierende Entitäten manipuliert werden kann.

welche von CollaborationObjects umgesetzt werden. CollaborationObjects repräsentieren die Zugriffsrechte über ObjectActions. Die Rechte von HumanActions und ObjectActions müssen übereinstimmen, wenn diese verbunden werden.

Andere Techniken, wie e.g. Business Process Model and Notation (BPMN), beinhalten auch einige Aspekte über menschliche Beteiligung an einem Prozesse [DT13]. BPMN wird meist in der Business Process Execution Language (BPEL) formuliert, um sie in ausführbare Form zu bringen. Die Erweiterung BPEL4People bietet die Möglichkeit, menschliche Aktivitäten abzubilden. Beide Sprachen sind aber primär für die Service Orientierte Architektur (SOA) gedacht und haben nur wenig Support für andere Architekturen. Auch sind die Möglichkeiten für die Abbildung kollaborativer Arbeit nicht Kernziel von BPMN und BPEL.

2.7 Synergien zwischen xADL und hADL

xADL dient der Beschreibung einer Software Architektur, hADL der Beschreibung der Umgebung - welche u.a. Menschen, welche mit der Software interagieren, beinhaltet. Verbindungen zwischen xADL und hADL Komponenten zur Designzeit sind möglich, um eine Interaktion mit der Software zu modellieren [DT13].

Wie in Abbildung 2.3 zu sehen ist, spezifiziert der Software Architekt zur Designzeit, wie sich Softwareelemente und kollaborative Elemente zueinander verhalten. Zur Laufzeit wird durch Monitoring (e.g. Anzahl der Benutzerzugriffe) und Analyse (e.g. Verhalten des kollaborativen Verhaltens der Benutzer) eine Adaptation der Architektur durchgeführt (e.g. weitere Instanzen starten). Diese Technik wird im Zuge der Arbeit nicht weiter verfolgt.

2.8 LittleJIL

LittleJIL kann zur Koordination von Prozessen verwendet werden [CLM+00]. LittleJIL besteht aus eine hierarchischen, baumartigen Anordnung aus sogenannten Steps (siehe Abbildung 2.4). Blätter im Baum repräsentieren dabei Units-Of-Work und spiegeln das

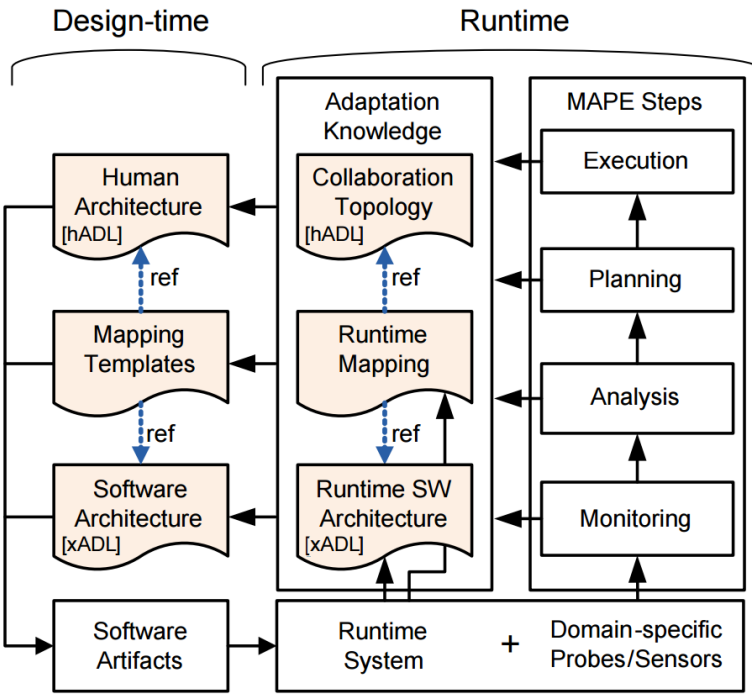

Abbildung 2.3: Synergien zwischen xADL und hADL

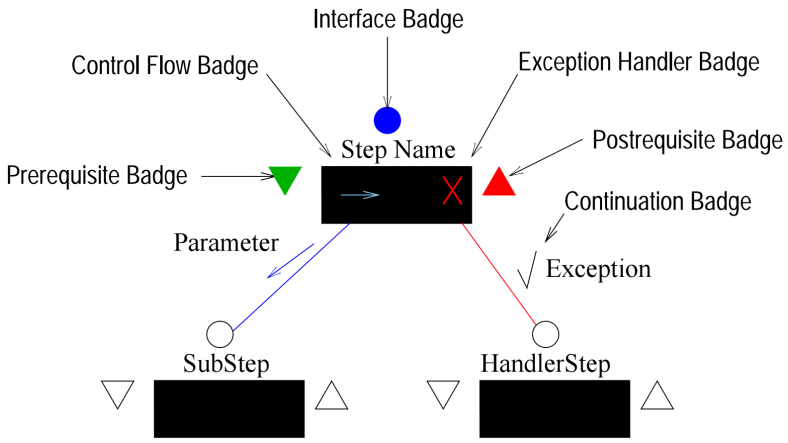

Abbildung 2.4: Ein LittleJIL Step

wieder, was ausgeführt werden soll. Die Koordination findet zwischen den einzelnen Steps statt.

Ein Step besteht aus folgenden Eigenschaften [CLM+00]:

- **Interface Badge**: Hier können übergeordnete Steps verbunden werden. Der Kreis

ist ausgefüllt, wenn lokale Deklarationen (Ressourcen und Parameter) zugeordnet sind.

- **Prerequisite Badge**: Ist die Vorbedingung eines Steps. Diese Bedingung erlaubt es zu definieren, wann ein Step ausgeführt werden darf.

- **Postrequisite Badge**: Ist die Nachbedingung eines Steps. Diese Bedingung erlaubt es zu definieren, welcher Zustand erreicht sein muss, wenn ein Step ausgeführt wurde.

- **Control Flow Badge**: Gibt an, in welcher Reihenfolge die Sub-Steps ausgeführt werden sollen. Es gibt dabei folgende Flows:

  - Sequentiell ($\rightarrow$)
  - Parallel in einer beliebigen Reihenfolge (||)
  - Try führt von links nach rechts Steps aus und hört auf, sobald der erste Step erfolgreich ausgeführt wurde
  - Choice - hier kann der Agent dynamisch entscheiden, welcher Step als nächstes ausgeführt wird

- **Exception Handler Badge**: Erlaubt es, Exception-Handler mit dem Step zu verbinden.

2.9 Anwendung von ADL in der Praxis

Woods und Hillard [WH05] haben 2005 einen Practice Session Report präsentiert, in dem im Zuge einer Konferenz u.a. die Alltagstauglichkeit von ADLs diskutiert wurde. Als Hauptaufgabe für ADL wurde die Dokumentation hervorgehoben, um das Wissen rund um die Architektur allen Beteiligten zu kommunizieren. Ebenfalls wurde die Frage diskutiert, warum ADL so wenig genutzt wurde - ein Auszug:

- Die meisten ADLs sind sehr restriktiv und zwingen die Nutzung eines bestimmen Architektur Modells

- Alle der Gruppe bekannten ADL haben nur eine Sichtweise unterstützt - mehrere Ansichten wären aber wünschenswert

- Die meisten ADLs sind sehr generisch und berücksichtigen nicht die Anforderungen spezieller Domänen

2.10 Modelltransformation und Interpretation

Unter Modelltransformation versteht man den Vorgang der Transformation aus einem oder mehreren Input-Modellen in ein oder mehrere Output-Modelle [SK03]. Ein Verständnis der Syntax und Semantik der Input- und Output Modelle muss gegeben sein. Dabei wird Metamodeling genutzt. Ein Metamodel ist ein Modell, welches in einer Metalanguage geschrieben ist und eine bestimmte Modeling-Language beschreibt [ZT13]. Es gibt Maschinen die Möglichkeit, Modelle zu interpretieren.

> One of the best ways to combat complexity of software development is through the use of abstraction, problem decomposition, and separation of concerns. [SK03]

Es gibt mehrere Strategien für die Modelltransformation. Diese werden meist durch Tools unterstützt, welche folgende Strategien anbieten [SK03]:

- **Direct Model Manipulation**: Ein Zugriff auf die interne Repräsentation ist gegeben und eine Manipulation per API ist möglich. Der Zugriff erfolgt meist in einer General-Purpose Sprache wie e.g. C#. Der Entwickler muss sich keine neuen Sprachen aneignen, ist aber durch die API bei der Transformation eingeschränkt.
- **Intermediate Representation**: Das Modell wird in ein Standard-Format - meist XML - exportiert. Eine Weiterverarbeitung erfolgt dann aus den XML Dateien. XSLT ist eine Möglichkeit der Manipulation und Überführung in ein Output-Modell. Der Aufwand zum Erstellen von XSLT ist allerdings nicht zu unterschätzen.
- **Transformation Language Support**: Das Tool bietet eine Sprache an, welche es erlaubt, die Transformation zu beschreiben und auszuführen. Eine Domain spezifische Sprache (DSL) wird zum Beschreiben der Transformation verwendet.

2.11 Black-Box Test Automation

System-Tester sind oft Applikations-Domänen Experten, sind aber nicht zwangsläufig mit dem System-Design vertraut [IAB13]. Bei Real-Time Embedded Systems (RTES) ist es daher schon lange gang und gäbe, dass die Softare-Under-Test (SUT) als Black-Box betrachtet wird. Es wird zusätzlich die Umgebung (Environment) modelliert und simuliert. Environment-Modelle helfen u.a. dabei, ohne echte Hardware zu testen. In einem sicherheitskritischen Umfeld kann durch die Simulation der Umgebung eine Vielzahl von Szenarien abgedeckt werden, die mit echter Hardware unter realistischen Rahmenbedingungen nicht realisierbar wäre [IAB13]. Die in [IAB13] angewandte Modellierungsstrategie erfasst alle strukturell wichtigen Details der Umgebung - inkl. deren Charakteristik und Beziehungen.

Im Kontrast zu traditionellen Black-Box Tests, wo Tests manuell aus den Anforderungen (Requirements) geschrieben werden, wird bei Model-Based-Testing ein Modell erzeugt, welches das erwartete Verhalten der SUT wiederspiegelt [UL10]. Eine Definition von Testen ist [UL10]:

> Software testing consists of the **dynamic** verification of the behavior of a program on a **finite** set of test cases, suitably **selected** from the usually infinite executions domain, against the **expected** behavior.

Die Bedeutung der Wörter kann wie folgt interpretiert werden [UL10]:

- **Dynamic**: Das bedeutet, dass das Programm ausgeführt wird (im Kontrast zu statischer Analyse).
- **Finite**: Alle möglichen Input- / Output-Werte zu testen ist für die meisten Programme nicht praktikabel. Dazu kommen noch unendlich viele ungültige oder unerwartete Eingaben.
- **Selected**: Die Anzahl der möglichen Tests ist unendlich - die Kunst besteht darin, jene Tests zu wählen, welche am ehesten Fehler aufzeigen. Hier ist das Gespür des Testers gefragt.
- **Expected**: Nach jedem Test muss festgestellt werden, ob das beobachtete Verhalten in Ordnung war oder nicht. Dieses Problem ist als Oracle-Problem bekannt und wird meist anhand von Analysen der Output-Werte gelöst.

Model-Based-Testing wurde in den letzten Jahren für eine Vielzahl von Teststrategien verwendent [UL10]:

- **Test Input-Data aus dem Domain-Model generieren**: Hier werden Test-Input-Values aus dem Domain-Model generiert. Allerdings ist es schwer zu sagen, wann der Test fehlgeschlagen ist oder nicht, da diese Information nicht vorhanden ist.
- **Tests aus dem Environment-Model generieren**: Aus dem Environment können ebenfalls Testfälle generiert werden, allerdings fehlt hier auch wieder die Information der erwarteten Ausgabewerte.
- **Tests mit einem Oracle aus dem Behaviour-Model**: Hier beinhaltet das Oracle die erwarteten Ausgabewerte. Hier muss der Testgenerator ausreichend Wissen über das Verhalten der SUT haben.

- **Test Scripts von Abstract-Tests**: Hier dient eine abstrakte Beschreibung eines Test-Cases (e.g. UML Sequenzdiagramm oder High-Level Procedure Calls) als Grundlage und es erfolgt eine Transformation in Low-Level Scripts, welche ausführbar sind.

Utting und Legeard haben speziell den dritten Fall untersucht [UL10]. Aus den Modellen (wie e.g. UML Class und State Diagramm, spezifizieren der Postcondition mit OCL) ist es möglich, automatisch Test-Input-Values zu generieren und das Test-Oracle. Beispiele für Tools wären: Microsoft Spec Explorer, NModel, ModelJUnit, Smartesting CertifyIt. Um das Behaviour-Model zu entwickeln, haben sich zwei Techniken etabliert:

- Transition-based Notations Umsetzung durch Finite-State-Machines (FSM). Gut geeignet für Control-Oriented Systeme. Hier sind nur gewisse Operationen in gewissen States möglich.
- Pre/post Notations: Umsetzung durch Pre-/Post-Conditions im Code. Gut geeignet für Data-Oriented Systeme, welche viele States haben.

2.12 Coordination

Malone und Crowston haben eine Untersuchung über Coordination in diversen Bereichen (Computertechnologie, Organisationslehre, Wirtschaft, Sprache und Psychologie) durchgeführt [MC94]. Dabei kann Coordination als der Prozess des Managens von Aktivitätsabhängigkeiten gesehen werden. Die grundlegende Frage der Coordination-Theory ist die Frage nach der Veränderung, welche durch Einbinden der Informations-Technologie in menschliche Arbeitensprozesse geschieht. Die grundlegenden Coordination-Processes sind (ein Auszug):

- **Shared Resources verwalten**: Wenn mehrere Aktivitäten eine gemeinsame, limitierte Ressource teilen (e.g. Geld, Speicher), so muss die Ressource-Allocation geregelt werden. Sehr wichtig ist dieser Prozess auch bei der Zuweisung von Aufgaben (Tasks) an ausführende Einheiten (Actors).
- **Producer/Consumer verwalten**: Wenn eine Activity etwas produziert, was von einer anderen Activity konsumiert wird, spricht man von einem Producer/Consumer Relationship. Folgende Punkte sind bei Producer/Consumer Relationships essentiell:
  - Prerequisite Constraints: Die Producer-Activity muss beendet sein, bevor die Consumer-Acivity beginnen kann. Durch eine Benachrichtung (Notification) kann eine Beendigung mitgeteilt werden.
  - Transfer: Wenn eine Activity etwas macht, was von einer anderen Acivity gebraucht wird, muss es zwischen den beiden Activities transferiert werden.
  - Usability: Das Resultat der Producer-Activity sollte von der Consumer-Acivity konsumierbar sein.

2.13 Collaboration Patterns

Anwender sind schon lange nicht mehr einfache Benutzer von Software - sie sind inzwischen ein integraler Teil. Besonders auffällig ist die größer werdende Unschärfe der Grenzen zwischen Benutzer und Software [DT14]. In einem kollaborativen Umfeld ist von Interesse, wer Tätigkeiten ausführt, wer sie koordiniert und wie das Zusammenspiel der einzelnen Instanzen gestaltet ist. Menschliche Akteure in einem kollaborativen Umfeld können **work-focused** oder **coordination-focused** Rollen einnehmen.

Es haben sich einige Kollaborations-Pattern etabliert. Die wichtigsten sind [DT14]:

- **Shared Artifact**: Ein Tuple-Space ist einem Shared Artifact sehr ähnlich. Sie ermöglichen starke Entkoppelung der Komponenten. Es können damit Aktivitäten abgebildet werden, wie die Manipulation von Dokumenten, Source Code oder eine Diskussionsplattform. In einem kollaborativen Umfeld können Entitäten dynamisch dem Shared Artifact beiwohnen, oder selbst eines erzeugen.
- **Master/Worker**: Sind unabhängige Arbeitsschritte gegeben, so können diese auf unterschiedliche Worker (Komponenten, welche die Arbeit erledigen) zugewiesen werden. Es ist möglich, dass Worker ihre Arbeit zugewiesen bekommen (Push-Style) oder Worker sich ihre Arbeit selbst holen (Pull-Style). Master/Worker ist in diesem Kontext auch als Crowdsourcing bekannt.
- **Publish/Subscriber**: Es findet auch hier eine starke Entkoppelung zwischen Publisher und Subscriber statt. Subscriber empfangen Events und produzieren ihrerseits u.U. wieder neue Events. In großen Umgebungen sind Plattformen wie Twitter ein gutes Beispiel für dieses Pattern.

Kollaboration in Sozialen-Netzwerken oder in anderen öffentlich zugänglichen Plattformen wurde schon untersucht [DT14] [DTD12].

Emergency-Call-Centers sind ein weiteres Beispiel einer kollaborativen Umgebung. Der Aufbau und die Abarbeitung sind von Land zu Land unterschiedlich. Dugdale, Pavard und Soubie haben als Beispiel eine Leitstelle in einem Außenbereich von Paris beschrieben [DPS00]. Es gibt zwei Notrufnummern: 18 für die Feuerwehr und 15 für den Rettungsdienst. Notrufe der Nummer 18 (nicht medizinische Notrufe) werden zuerst von einem Disponenten gefiltert und anschließend zu einer passenden Person weiter geleitet. Das selbe gilt für Notruf 15, wobei hier der Anruf von medizinisch geschulten Personal entgegen genommen wird. Es gibt auch Disponenten, welche beratend den Disponenten vom Notruf 15 und 18 zur Seite stehen [DPS00]. Die Verständigung mit den Einsatzkräften passiert heuzutage sehr oft nur mehr über Mobile Data Terminal (MDT), sodass Einschränkungen, welche es mit der verbalen Funktechnik gegeben hat, nicht mehr präsent sind.

NG9-1-1 ist eine Initiative in der USA und Kanada, die bestehende 9-1-1 Infrastruktur zu modernisieren[3]. Es wurde festgestellt, dass die Zeit, in der Notrufer ausschließlich

[3] `http://www.its.dot.gov/ng911/index.htm`

analoge Telefonleitungen nutzten, nicht mehr existent sind. Medien wie Voice, Text oder Video (e.g. Skype) liefern heute die Informationen - über verschiedene Gerätetypen (e.g. Telefone, Smartphones). Notrufe per Twitter sind kein abwegiges Szenario mehr[4].

> While the NG911 conversation continues, there are several communities beginning to embrace alternative forms to receive emergency information from citizens that incorporates the social media and Web 2.0 concepts of two-way information exchange and documentation. [Cro12] ... There is a growing interest by the public safety community to allow the public to send photos and videos to a dispatcher, showing conditions at an incident scene with more detail than a verbal description can provide, pictures of perpetrators or suspect vehicles [which] could be sent for immediate distribution to emergency responders in an area as well as for later use by investigators. [Cro12]

Es gibt bereits erste Entwicklungen, welche Smartphone-Apps[5] für die Video und Audio Kommunikation von 911-Notrufen verwenden. Auch in Österreich gibt es Konzepte hierfür[6]. Dem Public-Safety Answering Point (PSAP) - der Leitstelle - wird dadurch eine neue Bedeutung zugeordnet: Notrufer werden je nach Auslastung und Verfügbarkeit zur besten Leitstelle geroutet. Die Lokalität von IP-Endgräten kann allerdings nicht mehr trivial festgestellt werden [KSS+08].

Zusammenfassend kann festgehalten werden, dass die Kommunikationskanäle zu einer Leitstelle viel facettenreicher als noch vor einem Jahrzehnt sind und somit neue Herausforderungen für die kollaborative Zusammenarbeit bringen.

2.14 Simulation / Discrete Event Simulation / Markovian Analysis

Simulation ist die Imitation von Operationen eines Real-World Prozesses oder Systems über die Zeit [Ban98]. Bei einer Simulation wird eine künstliche Historie erzeugt, welche anschließend zur Interpretation herangezogen wird. Gründe für die Simulation können die Analyse des Verhaltens sein oder um "was wäre wennFragen zu beantworten. Ein Modell repräsentiert das System - das Modell sollte nur so komplex sein, damit es alle gestellten Fragen beantworten kann. Ein Event repräsentiert eine State-Änderung im System. Es wird dabei zwischen internen- und externen Events unterschieden. Jerry konzentriert sich in seiner Arbeit [Ban98] auf Discerete-Event Simulation Models, welche u.a. eine Möglichkeit neben mathematischen Modellen, deskriptiven Modellen oder statistischen Modellen darstellt.

[4] http://www.emsworld.com/news/10339592/atlanta-councilman-chooses-twitter-over-911-to-report-emergency

[5] http://m-urgency.umd.edu

[6] http://www.leitstelle-tirol.at/Notfall-App.52.0.html

System-State-Variablen spiegeln den internen State eines Systems zu einem gegebenen Zeitpunkt wieder. Sind die State-Variables definiert, kann eine Unterscheidung zwischen

- Discrete-Event Models: Diese Werte bleiben über ein Zeitintervall konstant und ändern sich nur an wohldefinierten Punkten - den Event-Times.
- Continous-Event Models: Werden durch Gleichungen ausgedrückt, welche den Wert über die Zeit hinweg ändern.

getroffen werden. Eine Simulation hat dabei folgende Vorteile [Ban98] - ein Auszug:

- Eine Simulation erlaubt es, sein Design zu überprüfen, bevor kritische Änderungen durchgeführt werden - eine nachträgliche Änderung ist mit hohen Kosten verbunden.
- Eine Simulation erlaubt es, die Zeit zu beschleunigen und zu verlangsamen. Das erlaubt, interessante Zeitabschnitte genauer zu betrachten.
- Möglichkeiten ausloten, ohne das Echtsystem zu beeinträchtigen.
- Das Verständnis über das System stärken.

Eine Simulation besteht aus folgenden drei Entitäten [Ban98] (siehe Abbildung 2.5):

- Das System der realen Welt
- Ein theoretisches Modell des Systems. Den Schritt vom realen Modell auf theoretisches Modell nennt man Simulation Modeling.
- Ein computerbasiertes Modell - das Simulation Programm. Den Schritt von einem theoretischen Modell auf ein computerbasiertes Modell nennt man Simulation Programming.

Das Input-Modell ist einer der wichtigen Kriterien einer Simulation, welche über deren Aussagekraft entscheidet. Die Daten für das Input-Modell können dabei wie folgt gewonnen werden:

- Erfahrung aus der Vergangenheit
- Theorie
- Beispieldaten aus dem realen Prozess

Eine Input-Variable eines Simulations-Modells kann ein stochastischer Prozess sein.

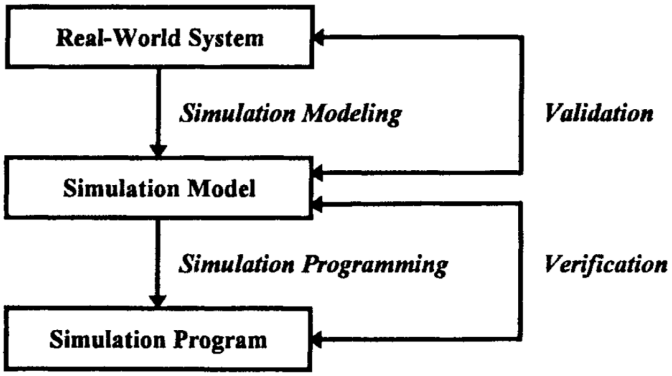

Abbildung 2.5: Schritte einer Simulation

2.15 Simulation in kollaborativen Umgebungen

In großen Deployments von kollaborativen Systemen sind oft Qualitätsmetriken von Interesse: der Durchsatz von Nachrichten, Anzahl der Änderungen usw. [DEM12]. E.g. könnte ein Architekt einer kollaborativen Plattform in der Anzahl der Schreibkonflikte von Personen mit einer gewissen Eigenschaft (e.g. Zugriffsrechte) interessiert sein. Nicht jedes Verhalten kann im System vorhergesagt werden - es kann auch zu unvorhersehbaren kollaborativem Verhalten kommen. System-Designer müssen durch ausgeklügelte Analysen dem Grund dieses Verhaltens auf die Spur gehen, um geeignete Maßnahmen setzen zu können.

2.16 Markov Kette

Der Vollständigkeit halber seien auch Markov-Ketten erwähnt, welche in der Arbeit allerdings nicht mehr weiter verfolgt werden. Bei der Markov-Kette handelt es sich um einen stochastischen Prozess. Die Übergange zwischen States sind mit Wahrscheinlichkeiten versehen. Diese sind aber gedächtnislos - die Vorgeschichte der Events spielt keine Rolle. Die Definition von Markov-Ketten ist facettenreich - eine Definition ist:

> Markov-Ketten haben einen endlichen, zählbaren State-Space und ist für Discrete-Time gedacht. Es gibt auch die Ausprägung des Continuous-Time Markov Process. [SCS14]

Lachlan, Comans und Scuffham haben Markov-Modeling und Discrete Event Simulation für die Simulationen im Gesundheitswesen untersucht. Eines der Hauptnachteile für diese Domäne war, dass Markov-Ketten gedächtnislos sind - Discrete Event Simulation allerdings nicht [SCS14]. Ebenfalls wurde festgestellt, dass mit Discrete Event Simulation komplexere Systeme modelliert werden können.

KAPITEL 3

Anforderungen und Abgrenzung

Dieses Kapitel befasst sich mit den Anforderungen und der Abgrenzung des Transformationstools. Der erste Teil des Kapitels analysiert das Fallbeispiel der Einsatzleitstelle. Der zweite Teil befasst sich mit den Anforderungen an das Transformationstool.

3.1 Domäne Einsatzleitzentrale von BOS (Behörden und Organisationen mit Sicherheitsaufgaben)

Wie bereits in Kapitel 1 beschrieben, dienen Leitstellen zur Koordination von Einsatzmittel (e.g. Feuerwehr- oder Rettungsfahrzeuge) oder als telefonische Auskunftsstelle (e.g. Öffnungszeiten von Apotheken oder Ärzten). Beispiele für Leitstellen in Österreich sind:

- **Berufsrettung Wien**: Erreichbar über 122 innerhalb des Stadtgebiets Wien
- **Berufsrettung Wien**: Erreichbar über 144 innerhalb des Stadtgebiets Wien
- **Leitstelle Tirol**: Hier handelt es sich um eine integrierte Leitstelle. Das bedeutet, dass die Leitstelle mehrere Institutionen betreut: Rettungsdienst, Feuerwehr, Flugrettung, Wasserrettung Landesverband Tirol, Bergrettungsdienst Tirol, Höhlenrettung Landesverband Tirol, Grubenwehr Silberberg Tirol und Ärztenotdienst der Ärztekammer Tirol.
- **Landesleitstelle Florian Steiermark**: Diese Leitstelle ist für die Koordination der Feuerwehren in der Steiermark verantwortlich und nimmt Notrufe 122 in der Steiermark entgegen. Die einzelnen Bezirke in der Steiermark verfügen über Bezirksleitstellen, welche bei erhöhten Einsatzaufkommen besetzt werden und direkt mit der Landesleitstelle verbunden sind.

Wie man sieht, ist alleine in Österreich der Aufbau der Leitstellen sehr facettenreich. Anhand der Steiermark sieht man auch, dass Leitstellen über große Distanzen hinweg miteinander verbunden sein können und die Koordination der Ressourcen gemeinsam bewerkstelligen. Das Routing der Notrufe erfolgt durch den Telekommunikationsprovider und wird entweder anhand des Standorts oder anhand von Policys geregelt (e.g. aktuelle Auslastung einer Leitstelle). Die Leitstelle ist auch unter dem Namen Public-Safety Answering Point (PSAP) bekannt. Die Koordination zwischen Leitstellen wird in der Arbeit nicht weiter betrachtet - Hauptaugenmerk liegt in der Koordination innerhalb einer Leitstelle.

3.1.1 Aufbau einer Leitstelle

Der Aufbau der Leitstellen kann historisch bedingt sehr unterschiedlich sein. Meist wird ein verbaler Notruf von einem Sprachvermittlungssystem entgegen genommen und in der Leitstelle verteilt. Die Verteilung erfolgt anhand von dynamischen Regeln (e.g. Tages-/Nachtbetrieb, aktuelle Auslastung). Der Disponent (Dispatcher) nimmt den Anruf entgegen und wird sich als erstes nach der Lokalität des Schadensereignisses erkundigen. Anschließend setzt er geeignete Maßnahmen zur Minderung des Schadensereignisses. Die Aufteilung der Arbeit auf die Rollen **Calltaker** (nimmt Notrufe entgegen und gibt erste Informationen in das Einsatzleitsystem ein) und **Dispatcher** (weist Ressourcen dem Schadensereignis zu) ist auch eine häufig anzutreffende Arbeitsweise.

Ebenfalls können Hilfesuchende die Leitstelle über andere Wege erreichen (e.g. Gehörenlosenfax, SMS). Hier wird dem Disponent (Dispatcher) die Nachricht auf einem abgesetzten Client des Sprachvermittlungssystems oder im Einsatzleitsystem angezeigt. Auch gibt es schon Tendenzen, dass Leute über Social Media Hilfe anfordern (e.g. Twitter, Facebook - auf den jeweiligen Seiten der Organisation bzw. mit entsprechendem Hash Tag[1]).

Wie in Abbildung 3.1 zu sehen ist, versehen Disponenten (Dispatchers) in einer Leitstelle Dienst. Viele Leitstellen sind rund um die Uhr besetzt. Auch gibt es Szenarien, in denen Dispatcher die Leitstelle von zu Haus aus unterstützen (e.g. Silvester).

Es ist auch für Hilfesuchende möglich, an Hilfe zu gelangen, ohne aktiv mit der Leitstelle Kontakt aufzunehmen. In vielen Gebäuden Österreichs sind Brandmeldeanlagen vorgeschrieben. Der schematische Aufbau ist in Abbildung 3.2 zu sehen.

Die Brandmeldezentrale (BMZ) ist die zentrale Sammelstelle für Alarme in einem Gebäude. Wie im Bild links zu sehen ist, kann sie e.g. durch einen Druckknopfmelder (DKM) oder Brandmelder (e.g. Rauchmelder, Wärmemelder) ausgelöst werden. Je nach Konfiguration leitet die Brandmeldezentrale den Alarm direkt in die nächste Leitstelle oder löst hausintern einen Alarm aus. In diesem Fall sieht der Disponent sofort den Einsatzort und u.U. auch weitere Informationen, wie Gebäudetrakt oder Auslöseart (DKM oder Brandmelder).

[1] http://www.pcworld.com/article/261579/help_japan_mulls_911_emergency_calls_from_twitter_social_networks.html

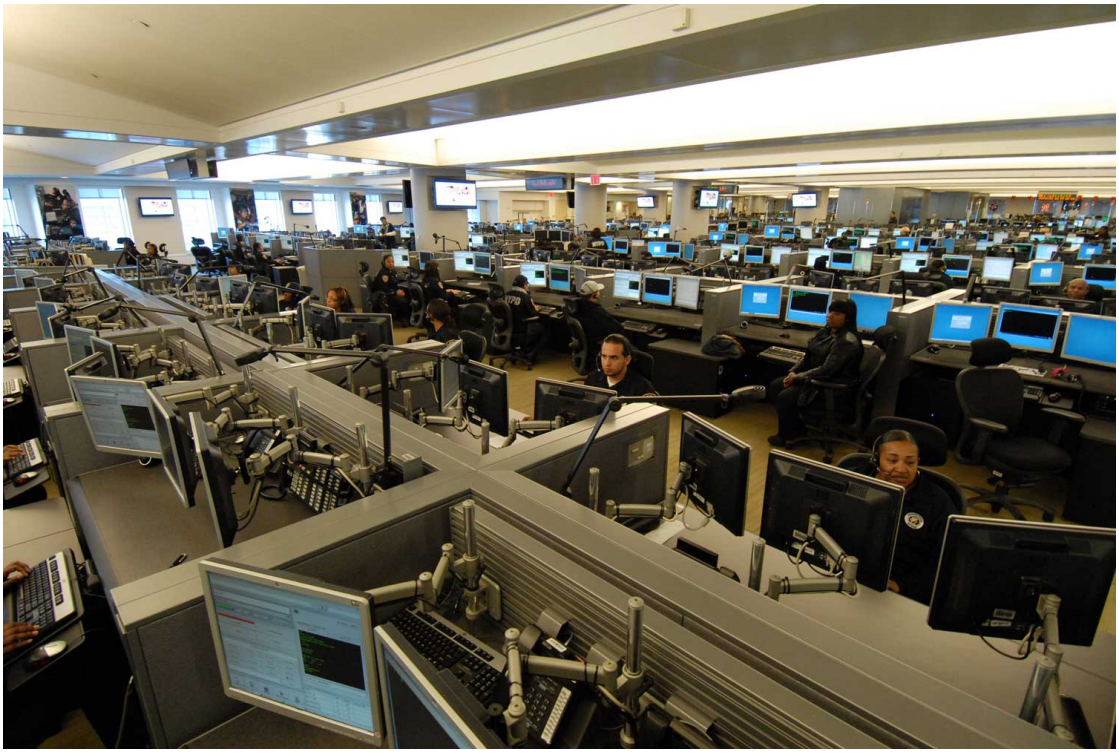

Abbildung 3.1: Leitstelle 911 New York. Quelle: `http://www.nyc.gov`

3.1.2 Schematische Darstellung einer Leitstelle

Abbildung 3.3 zeigt eine schematische Darstellung einer Leitstelle. Die verwendete Modellierungssprache ist frei erfunden. Details wie Telekommunikationsdienstanbieter oder das Sprachvermittlungssystem in der Leitstelle wurden dabei abstrahiert. Ebenfalls ist für die bevorstehende Fragestellung nicht von Interesse, über welchen Kommunikationsweg (e.g. Mobilfunkanbieter, BOS TETRA) der Einsatzleiter seine Daten aus der Leitstelle bezieht. Das Modell kann wie folgt interpretiert werden:

- In einer Leitstelle gibt es mehrere **Bedienplätze**, welche sich aus einem **Einsatzleitrechner** und einer Kommunikationseinheit (in diesem Fall nur ein Headset) zusammensetzen. Die Steuerung der Kommunikation (e.g. Notrufe annehmen, Funken, SMS lesen) erfolgt ebenfalls über das **Einsatzleitsystem**. Wie das Einsatzleitsystem aufgebaut ist, ist aus dem Modell nicht ersichtlich. Es kann aus einem Verbund aus Rechnern bestehen (e.g. Peer-To-Peer) oder nur einem Server. Das dargestellte Symbol zeigt eine logische Recheneinheit.
- **SMS**, **Telefonanruf** und **Brandmeldezentrale** agieren als Informationsquellen für die Leitstelle. Sie informieren im Szenario über verbale Kommunikation und dem Datenweg über ein Schadensereignis.

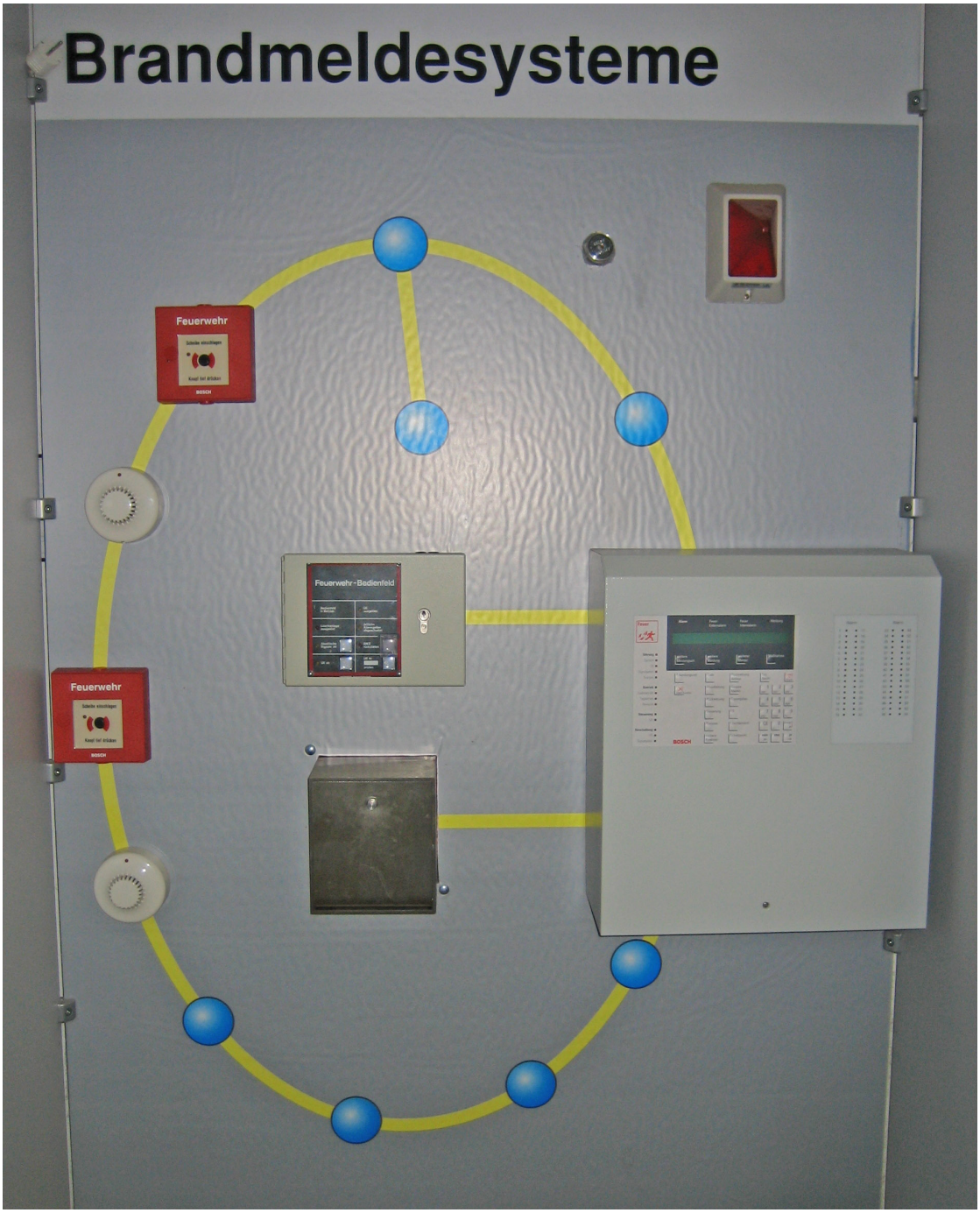

Abbildung 3.2: Schemantischer Aufbau einer Brandmeldezentrale, Quelle: Wikipedia

- Der **Einsatzleiter** und **Schichtleiter** überwachen die Ereignisse mit Hilfe eines Datenterminals, können aber auch aktiv eingreifen. **Feuerwehrfahrzeuge** (ausgestattet mit einem Datenterminal) können Informationen des Einsatzes abfragen und dokumentieren.
- Damit eine **Feuerwache** weiß, wann und mit welchen Fahrzeugen sie ausrücken muss, bekommt diese auch die Informationen über den aktuellen Einsatz.

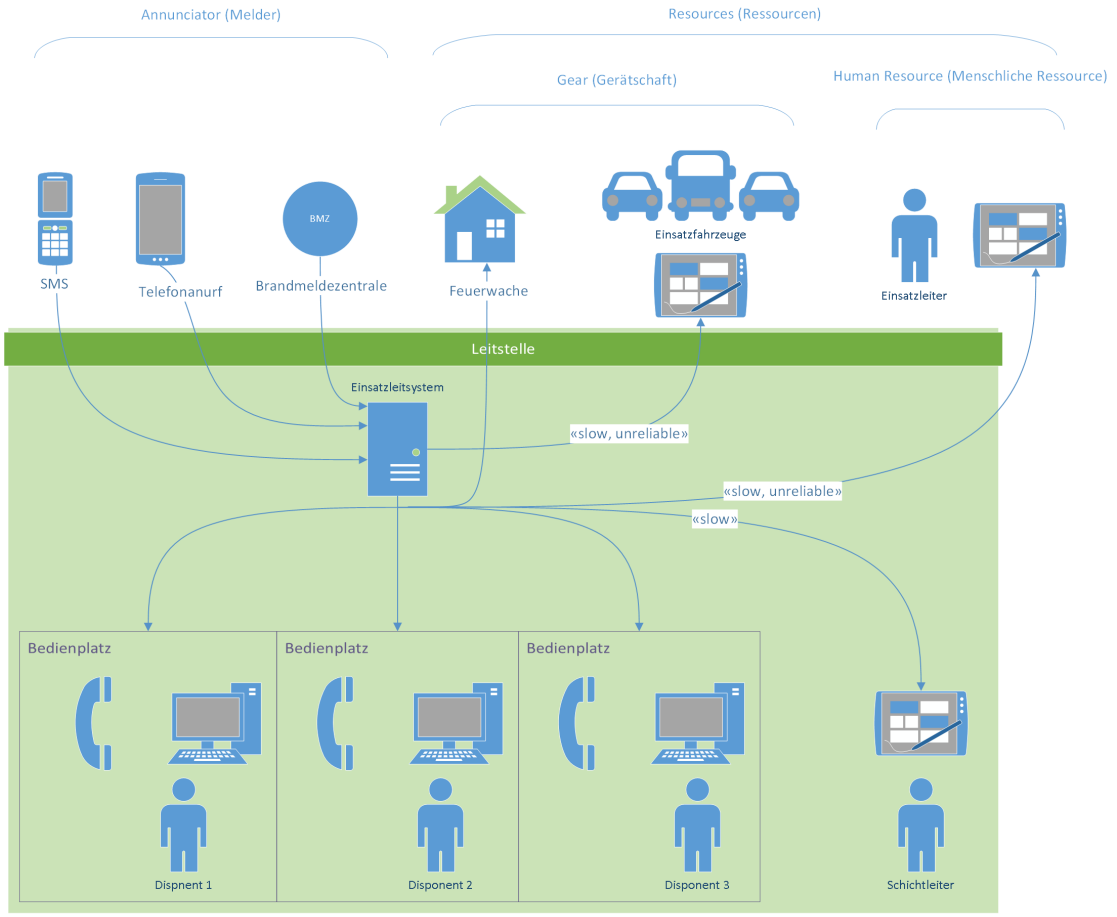

Abbildung 3.3: Schemantischer Aufbau einer Leitstelle

Ebenfalls dem Modell zu entnehmen ist, dass die Verbindungsgeschwindigkeiten des Schichtleiters, der Feuerwehrfahrzeuge und des Einsatzleiters langsam sind. Der Einsatzleiter und die Feuerwehrfahrzeuge verfügen zusätzlich über eine unzuverlässige Verbindung (e.g. Mobilfunk).

Aus dem Modell als auch aus der textuellen Beschreibung (fetter Schriftsatz) haben sich Rollen heraus manifestiert. Diese Rollen sind für die weitere Fragestellung von

großem Interesse, da sie Teil des kollaborativen Netzwerkes sind. Durch Zusammenarbeit kann ein gemeinsames Ziel erreicht werden: die Minderung des Schaden am Objekt (Firmengebäude).

3.1.3 Modelle

In den folgenden Abschnitten wird durch Modelle die reale Domäne der Leitstelle abstrahiert. Ein Modell stellt immer eine Abstraktion dar. Wichtig ist, dass die Details im Modell so gewählt werden, dass sie ausreichend sind, um die gestellte Frage zu beantworten. Nicht benötigte Details sollten weggelassen werden, um das Verständnis nicht unnötig zu erschweren. Modelle können von unterschiedlichen Leuten für unterschiedliche Dinge wiederverwendet werden. [Fai10]

> Essentially, all models are wrong, but some are useful. – George Box [BD87]

3.1.4 Gründe zur Kollaborationsanalyse

Fairbanks [Fai10] zählt in seinem Buch gebräuchliche Elemente einer Architektur auf (ein Auszug): Modules, Components, Connectors, Ports, Protocols, Quality Attributes und Models (e.g. Security Policies, Concurrency Models). All diese Elemente sind sehr generisch und können in einer Architektur individuell eingesetzt werden. Die Frage, die sich ein Softwarearchitekt stellen muss ist, ob die getroffenen Entscheidungen den gewünschten Erwartungen entsprechen. Da architekturelle Entscheidungen u.U. schwer zu ändern sind, ist eine frühe Abklärung von Interesse. Simulation ist eine Möglichkeit und wird in der Arbeit verfolgt.

3.1.5 Canonical Model Structure

Um Modelle gruppieren zu können, ist festzustellen, welche unterschiedlichen Ausprägungen von Modellen es in einer Architektur gibt. Abbildung 3.4 zeigt eine Separierung der Modelle, welche von Fairbanks vorgeschlagen wurde [Fai10] und unter dem Namen Canonical Model Structure bekannt ist.

Abbildung 3.4 zeigt mehrere Modelle, welche im Folgenden kurz erklärt werden:

- Das **Domain-Model** spiegelt die Wahrheit über die Domäne wieder. Ein Beispiel wäre, dass ein Einsatz immer eine Lokalität hat. Das Domain-Model drückt Details aus, welche u.U. nichts mit der konkreten Systemimplementierung zu tun haben.
- Das **Boundary-Model** spiegelt das wieder, was von außen gesehen werden kann (e.g. Behavior, Interchange Data und Quality Attributes). Das ist wichtig, da es nur jene Details zeigt, welche zum Zugriff auf das System benötigt werden, aber Interna nicht Preis gibt. Es ist ein View des Design-Modells.
- Das **Internals-Model** ist ein weiterer View des Design-Models, welche aber im Kontrast zum Boundary-Model sämtliche Details enthält.

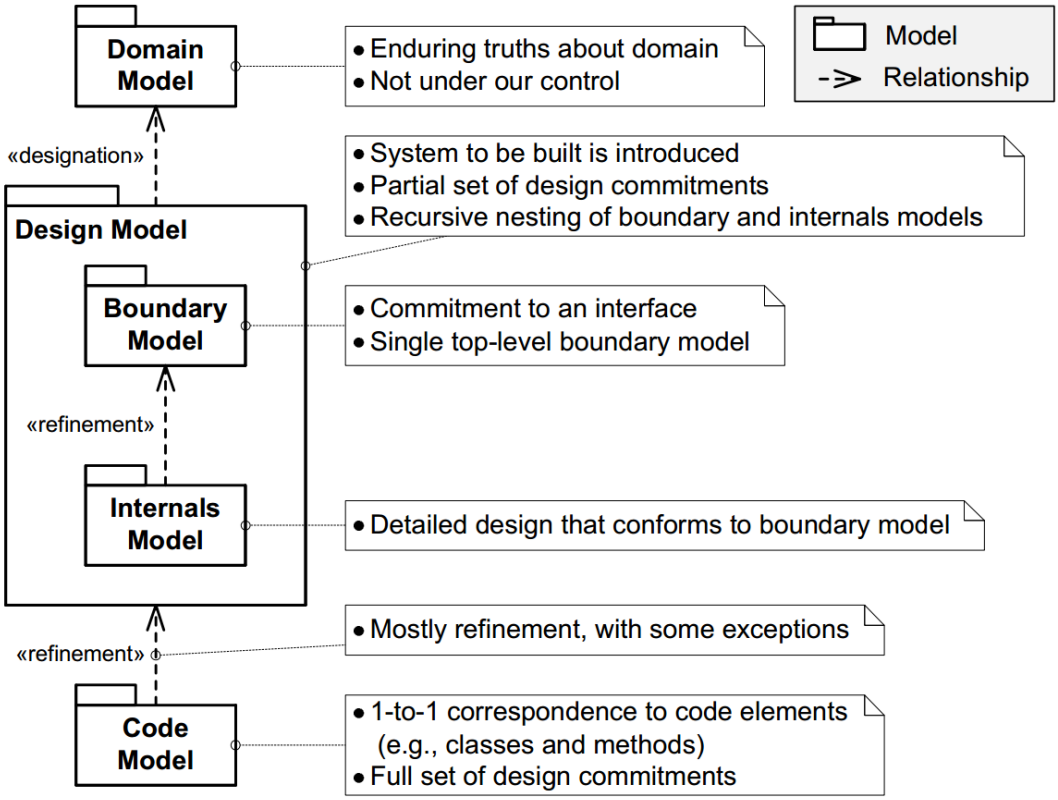

Abbildung 3.4: Canonical Model Structure nach Fairbanks

Wie aus Abbildung 3.4 zu entnehmen ist, besteht zwischen Domain-Model und Design-Model eine Verbindung mit der Anmerkung Designation. Das bedeutet, dass gleiche Dinge in unterschiedlichen Modellen das Gleiche ausdrücken. Das bedeutet, dass Entitäten aus dem Design-Model auch Abbildungen der Entitäten im Domain-Model haben. Die vorliegende Arbeit nutzt größtenteils das Domain-Model.

3.1.6 Szenrario

Ein Szenario soll überblicksmäßig den Ablauf eines Einsatzes wiederspiegeln. Das betrachtete Szenario spiegelt einen Einsatz in Wien wieder. Es handelt sich um einen Bürogebäudebrand. Der fiktive Einsatz findet am 25.03.2015 statt. Ein Mitarbeiter leert den Aschenbecher in einen Mistkübel. Zuvor hat ein anderer Mitarbeiter Zetteln einer Besprechung in den Mistkübel geworfen, sodass ein Glimmbrand entsteht. Nach einiger Zeit entwickelt sich ein Feuer, welches auf die gesamte Einrichtung übergreift - der installierte Brandmelder löst aus. Die Brandmeldezentrale meldet den Alarm an die Berufsfeuerwehr Wien. Zum selben Zeitpunkt entdeckt eine Reinigungskraft die starke Rauchentwicklung und meldet den Vorfall per Notruf 122. Ein Mitarbeiter nimmt starken Rauchgeruch durch die Klimaanlage wahr und meldet den Vorfall ebenfalls. Die Brandmeldezentrale

ist so programmiert, dass sie ein SMS an den Brandschutzbeauftragten (Mitarbeiter der Firma) schickt. Dieser befindet sich allerdings zu diesem Zeitpunkt in einer wichtigen Besprechung. Verunsichert leitet er das SMS an den Notruf 122 weiter.

Nach dem die Feuerwehr laut Alarmplan Fahrzeuge zum Ort des Geschehens geschickt hat und weitere Einsatzkräfte (e.g. Rettung, Polizei) verständigt hat, unterstützt diese die Einsatzkräfte vor Ort. Der Einsatzleiter der Feuerwehr vor Ort kommuniziert mit der Leitstelle über Funk. Er fordert zwei weitere Hilfeleistungslöschgruppenfahrzeug (HLF) an. Er wird durch 2 Mitglieder der Berufsfeuerwehr unterstützt, welche jeweils einen Datenterminal haben und die Position und den Alarmierungsstatus nachverfolgen. Über das Datenterminal kann ebenfalls mit der Leitstelle kommuniziert werden, sodass eine Drehleiter nachbestellt wird.

Wie man sieht, sind Disponenten (Dispatcher) auf Information angewiesen, um die Ressourcen koordinieren zu können. Oft ist der Informationsfluss allerdings redundant.

3.1.7 Domäne der Feuerwehr

Um die Begrifflichkeiten rund um das Feuerwehrwesen besser verstehen zu können, kann ein Domain-Modell verwendet werden. Dieses spiegelt nicht zwangsläufig alles wieder, was im zu bauenden System umgesetzt wird. Des Weiteren werden nur Details erfasst, welche für die vorliegende Arbeit eine konkrete Bedeutung haben. Umgesetzt wird die Domäne im Programm Code meist in Englisch, da internationale Entwickler beteiligt sind. Das Domänenmodell in der Arbeit enthält die englischen Begriffe mit deutscher Beschreibung (siehe Abbildung 3.5).

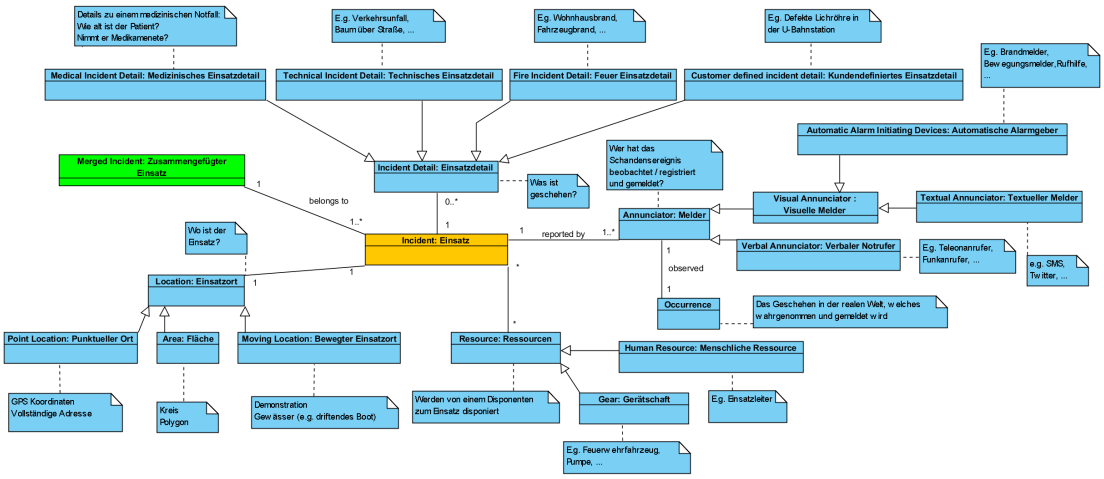

Abbildung 3.5: Domain-Model von Public-Safety Incidents

Um die Laufzeit Instanzen (aus Domänensicht) besser verstehen zu können, wird ein beliebiges Szenario herangezogen und diskutiert. Abbildung 3.6 zeigt ein Objektdiagramm, welches die Zusammenhänge darstellt. Zu beachten ist, dass Links (Verbindungen

zwischen Objektinstanzen) bedeuten, dass die Objektinstanzen zur Laufzeit miteinander kommunizieren können.

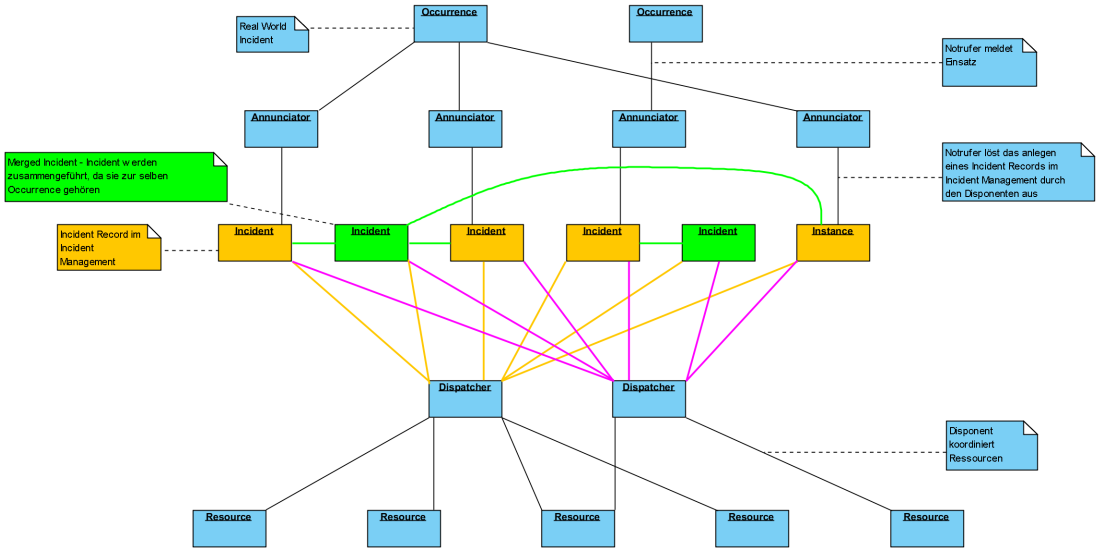

Abbildung 3.6: Objektdiagramm von Public-Safety Incidents

Folgendes kann aus dem Objektdiagramm-Szenario entnommen werden:

- Es geschieht eine Occurrence (Vorkommnis) in der realen Welt (e.g. Verkehrsunfall, Wohnhausbrand, ...)
- Dieses wird von einem Annunciator (Melder) einer Leitstelle gemeldet (die Leitstelle ist nicht abgebildet)
- Der Dispatcher legt einen Incident im System an (blaue Instanzen). Da im Objektdiagramm mehrere Annunicator die gleiche Occurrence gemeldet haben, legen unterschiedliche Dispatcher - aufgrund der Verteilung der Aufgaben - für eine Occourence mehrere Incidents an (blaue Instanzen). Warum das so ist, wird im Zuge der Arbeit noch detailliert analysiert.
- Da es nun mehrere Incidents zu einer Occurrence gibt, werden diese nach Erkennung des Missstands zu einem Incident zusammengeführt. Grüne Incident Instanzen zeigen einen zusammengeführten Incident (mehrere blaue Incidents ergeben einen grünen Incident).
- Jeder Dispatcher hat Zugriff auf jeden Incident
- Der Dispatcher entscheidet anhand des Incident welche Ressourcen er verständigt

Die Herausforderung für den Disponenten liegt in mehreren Schritten. Der Disponent hat die Aufgabe, verbale Notrufe (verbal Annunciator) e.g. über ein Telefonat abzufragen und

in das System einzutragen, sodass dieses die Eingaben semantisch verwerten kann. Das ist notwendig, um e.g. Einsätze in der Umgebung finden zu können (womöglich der gleiche Einsatz) und erlaubt es dem System, Ressourcenvorschläge machen zu können. Das System wird für einen Brand andere Fahrzeuge vorschlagen, als für einen Verkehrsunfall. Umso besser die Eingaben sind (semantisch hochwertiger), umso besser ist der Vorschlag. Abbildung 3.7 zeigt ein mögliches Aktivitätsdiagramm für einen verbalen Notruf. Alternativ zum Ablauf in Abbildung 3.7 kann auch bei jedem Notruf automatisch ein neuer Einsatz angelegt werden (wie in Abbildung 3.6 dargestellt). Welches Szenario zum Tragen kommt, hängt von der Größe der Leitstelle ab. Die Absprache mit zwei Disponenten ist weitaus geringer, als jene mit 50 Disponenten und erlaubt es, Occurrence-Duplikate frühzeitig zu erkennen.

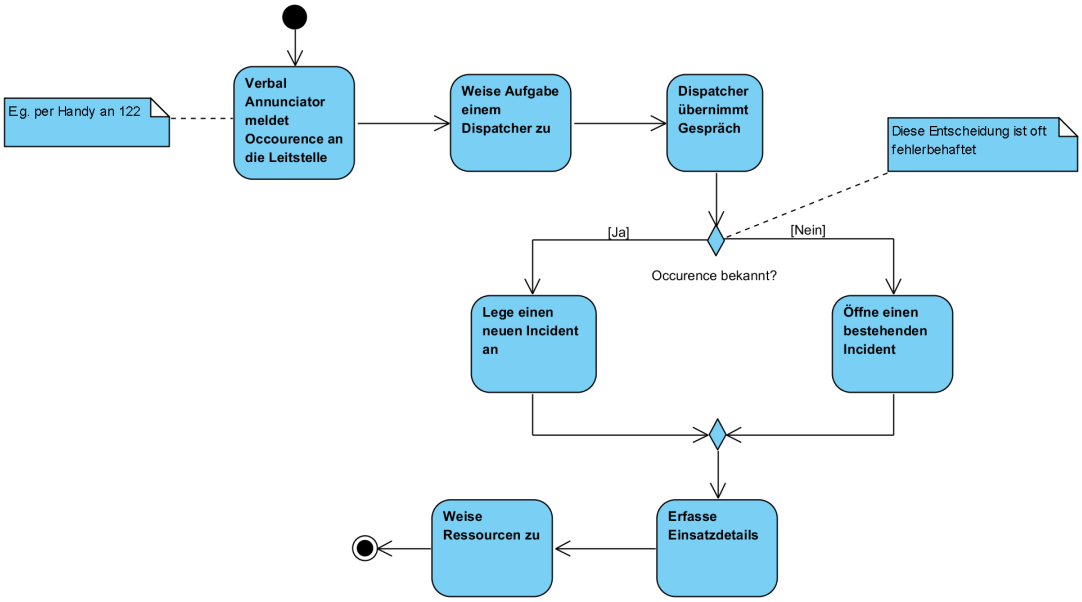

Abbildung 3.7: Aktivitätsdiagramm von einem Vorfall

Wie aus dem Aktivitätsdiagramm ebenfalls zu entnehmen ist, muss ein Disponent recht früh die Entscheidung treffen, ob das Vorkommnis (Occurrence) ihm bereits bekannt ist oder nicht. Da in der Arbeit eine größere Leitstelle analysiert wird, wird angenommen, dass bei jedem Notruf automatisch ein Incident angelegt wird. Es ist daher ein Merge-Vorgang erforderlich, um Duplikate zu entfernen. Um die Problematik besser beleuchten zu können, wird zuerst der verbale Anruf bzw. die Art und Weise wie der Disponent (Dispatcher) an Informationen über Vorkommnisse kommt, analysiert.

3.2 Abstraktion

Im folgenden Abschnitt werden die Abstraktionen der Entitäten beschrieben. Zur Erinnerung: Jedes Modell stellt eine Abstraktion dar.

3.2.1 Abstraktion eines Telefonanrufs (Verbal Annunciator)

Ein Telefonanruf ist ein verbales Gespräch über einen gewissen Zeitraum. In diesem Zeitraum teilt der Anrufer dem Disponenten Informationen über seine Beobachtungen mit. Die Abfrage erfolgt anhand er 5-W Fragen[2]:

- Wo ist etwas geschehen?
- Was ist geschehen?
- Wie viele Personen sind betroffen?
- Welche Art von Erkrankung/Verletzung/Schaden liegt vor?
- Warten auf Rückfragen! (das Gespräch nicht unaufgefordert beenden) / Wer meldet?

Das "Was ist geschehen?"kann auf unterschiedliche Art und Weise abgefragt werden - entweder es ist dem Disponenten frei überlassen oder er kann ein standardisiertes Abfrageverfahren (e.g. Advanced Medical Priority Dispatch System (AMPDS)) nutzen. Dies spielt aber für die Fragestellung der vorliegenden Arbeit keine Rolle, da es lediglich zu einem Informationsfluss zwischen Anrufer und Disponent kommt (Abbildung 3.8).

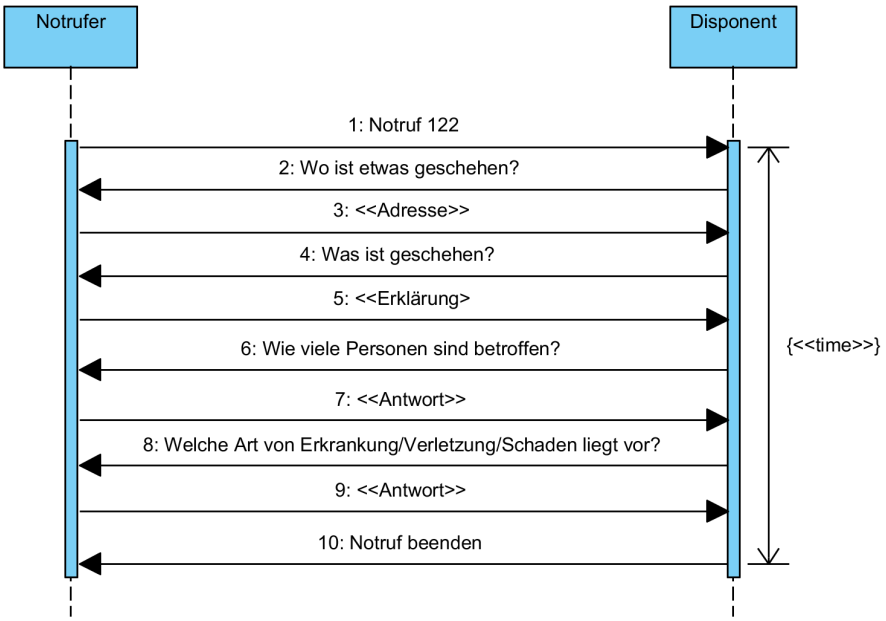

Abbildung 3.8: Sequenzdiagramm eines Notrufs

Für das Modell spielt es keine Rolle, dass zwischen Notrufer und Disponent ein Wechselgespräch stattfindet (Abbildung 3.9). Stattdessen kann festgehalten werden, dass der Notruf

[2] http://de.wikipedia.org/wiki/Notruf

einen Disponenten eine gewisse Zeiteinheit beschäftigt und dass ein Informationsfluss statt findet. Der Notruf kann daher auch als unidirektionaler Fluss von Informationen vom Notrufer zum Disponent gesehen werden.

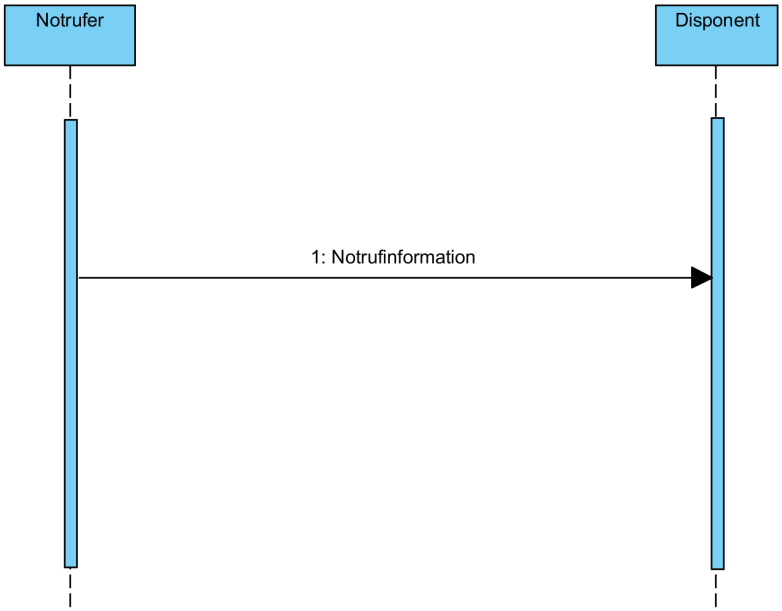

Abbildung 3.9: Vereinfachung des Notrufs

Die dabei überlieferte Information kann als Objekt gesehen werden (Abbildung 3.10).

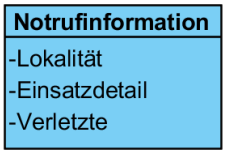

Abbildung 3.10: Objekt des Einsatzdetails

Der Disponent wird im Regelfall den Notrufer als Erstes nach der Lokalität fragen - würde nämlich das Telefongespräch abbrechen, so könnte man zumindest ein Fahrzeug zur Nachschau disponieren. Gibt der Disponent als erstes immer die Lokalität ein, so kann auch recht früh ein Konflikt erkannt werden: Rufen nämlich zwei Notrufer für das gleiche Schadensereignis zeitgleich an, so gelangen sie zu jeweils einem anderen Disponenten. Würden beide Disponenten ohne Kenntnis des jeweils anderen Disponenten wie gewohnt weiter arbeiten, so würden doppelt so viele Ressourcen (e.g. Feuerwehrfahrzeuge) wie benötigt zum Ort des Geschehens geschickt werden. Das System könnte allerdings schon sehr früh erkennen, dass zwei Disponenten versuchen, einen gleichen Einsatz anzulegen. Ist die Adresse gleich, so kann es je nach Einsatztyp ausreichend sein:

- Melden zwei Notrufer einen Dachstuhlbrand in der Stumpergasse 31 kann von einem identen Einsatz ausgegangen werden.
- Melden zwei Notrufer eine Person in Zwangslage (e.g. Person in der Wohnung gestürzt) - Einsatzadresse Stumpergasse 31 - so kann ohne Kenntnis der Türnummer nicht mehr davon ausgegangen werden, dass es sich um den selben Einsatz handelt. (Anmerkung: Es ist davon auszugehen, dass Disponenten bei Türöffnungen nach einer Stiege / Stockwerk / Türe fragen)
- Melden zwei Notrufer jeweils einen Einsatz an der gleichen Adresse, aber einen unterschiedlichen Einsatztyp, so hängt es vom Einsatztyp ab, wie vorgegangen wird. E.g. könnten ein Dachstuhlbrand und ein Zimmerbrand im gleichen Gebäude als gleicher Einsatz gesehen werden. Eine Türöffnung und ein Rohrbruch allerdings nicht.

Sind die Adressen unterschiedlich (aber das gleiche Schadensereignis), wird die Sache noch etwas komplizierter. Eine Strategie wäre eine Umgebungssuche anhand der Adresse:

- Notrufer 1 meldet Stumpergasse 31
- Notrufer 2 meldet Stumpergasse 33

Abbildung 3.11 zeigt vereinfacht ein Sequenzdiagramm der Abfrage. Man kann die Abfrage hier als Protokoll betrachten, bei dem in zwei Phasen eine Abfrage passiert. Auch werden beide Disponenten so viele Informationen wie nur möglich vom Notrufer abfragen. Dass ein Disponent die Abfrage aufgrund einer positiven Duplikatserkennung abbricht, ist unwahrscheinlich, da er immer weitere brauchbare Informationen erfragen kann.

Es kann festgehalten werden, dass das System den Disponenten auf jeden Fall informativ unterstützen sollte, wenn es um die Erkennung von womöglich identen Einsätzen in der Umgebung geht. Aus Sicht des Autors ist das allerdings ein Usability-Problem ("Wie stelle ich zwei idente Einsätze übersichtlich dar?"), und nicht ein Kollaborations-Problem.

Ebenfalls von Interesse ist, wie ein Disponent überhaupt zu einem Notruf gelangt. Der Notruf wird nicht direkt an einen Disponenten gerichtet. Für das Modell ist es ausreichend, dass man annimmt, dass der Notruf an eine Leitstelle weiter geleitet wurde - Details werden nicht modelliert. Ist der Notruf in der Leitstelle angekommen, wird er ausgewählten Disponenten signalisiert (im einfachsten Fall allen). Der erste Disponent, welcher den Anruf annimmt, bekommt ihn zugewiesen. Diese Art der Koordination ist unter dem Master/Worker Pattern bekannt und wurde in Kapitel 2 analysiert. In diesem Fall wird ein Pull-Style angewandt, bei dem der Worker (Disponent / Einsatzleitrechner) seine Arbeit beim Master anfordert (Abbildung 3.12). In realen Umgebungen ist der Master sehr komplex, da Telefonate Prioritäten haben und u.U. weitergleitet oder von einem Disponenten erneut in die Warteschleife gelegt werden. Diese Details werden allerdings nicht modelliert, da sie nicht von Interesse sind. Stattdessen wird der Master in Form einer einfachen Queue umgesetzt.

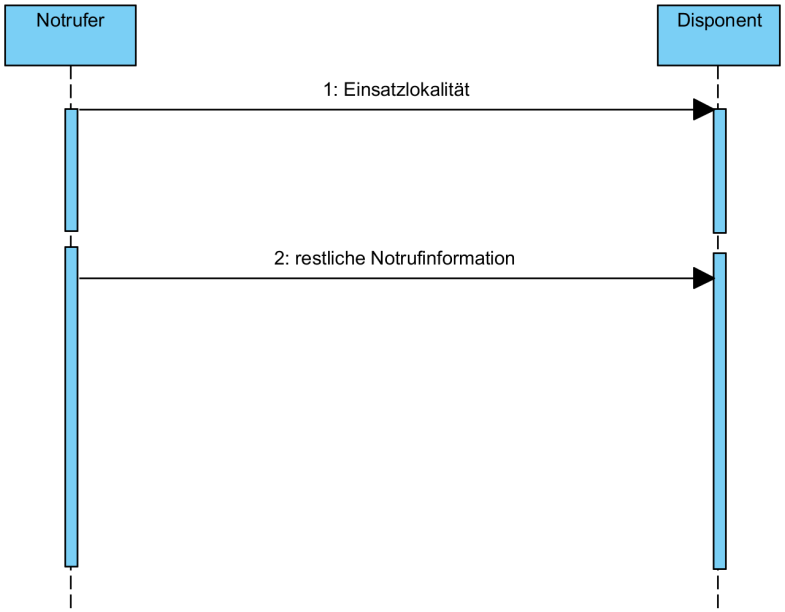

Abbildung 3.11: Abstraktion des Notrufs inkl. Aufteilung unterschiedlicher Informationen

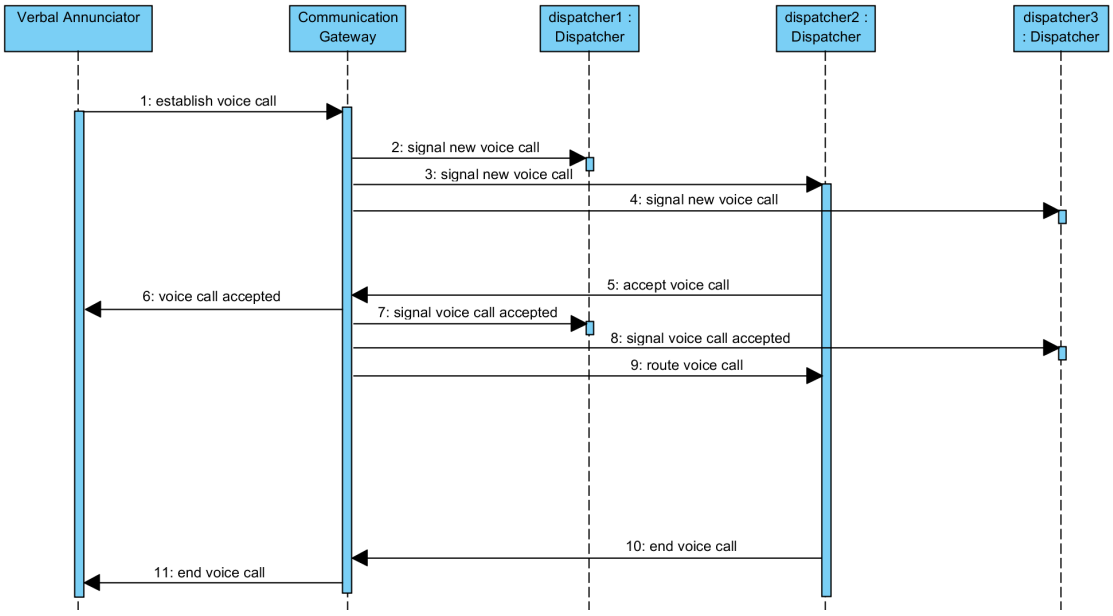

Abbildung 3.12: Zuweisen eines Telefonats im Pull-Style

Der Vollständigkeit halber seien noch Ausprägungen des Masters erwähnt. Neben den Pull-Style gibt es auch einen Push-Style. Bei diesem werden Anrufe einem freien Disponenten zugewiesen. Nimmt der Disponent den Anruf nach einer gewissen Zeit nicht an, so wird er erneut in die Verteiler-Warteschlange gegeben. Ein Disponent kann auch seinen Status künstlich verzögern, um e.g. eine Tätigkeit zu erledigen (e.g. Einsatzdokumentation vervollständigen, Toilette aufsuchen). Der Push-Style ist schwieriger zu modellieren, da er weitaus mehr Edge-Cases beinhaltet. Er wird allerdings in einigen Leitstellen erfolgreich eingesetzt.

3.2.2 Abstraktion SMS, Brandmelder (Visual Annunciator)

Diese beiden Arten der Meldung werden als unidirektionale Variante des Meldens gesehen. Es wird angenommen, dass eine SMS und ein Brandmelder sehr ähnliche Informationen wie ein verbaler Anruf beinhalten. Die Art und Weise, wie die Information an den Disponenten gelangt, ist allerdings unterschiedlich. Jeder Disponent bekommt die Information, dass eine Meldung vorliegt. Jener Disponent, welcher als erstes die Meldung akzeptiert, bekommt diese zugewiesen. Malone und Crowston haben dieses Patterns bereits analysiert (siehe Kapitel 2). Dort wird es unter Managing Shared Resources kategorisiert:

> Whenever multiple activities share some limited resource, a resource allocation process is needed to manage the interdependencies among these activities.
>
> **Task Assignment**. One very important special case of resource allocation is task assignment, that is, allocating the scarce time of actors to the tasks they will perform.

Es kann festgehalten werden, dass nur einem Disponenten der Task zugewiesen wird - und zwar jenem, der ihn als erster akzeptiert (Pull-Style).

Wie in Abbildung 3.13 zu sehen ist, gewinnt der erste Disponent, der das Ereignis (SMS, Brandmelder, ...) akzeptiert. Das Protokoll kann auch so gestaltet sein, dass es protokolliert, wenn ein Disponent das Ereignis ablehnt - ansonsten ist diese Information nicht erforderlich.

Wie lange soll ein Kommunikationssystem auf eine Antwort warten? Prinzipiell unendlich lang. Allerdings sollte ein Eskalationmechanismus entworfen werden, falls zu lange gewartet wird. Dies wird allerdings in der Arbeit nicht berücksichtigt.

3.2.3 Konflikt Erkennung

Das Erkennen von Konflikten ist einer der wichtigsten Aufgaben eines Einsatzleitsystems, da es bei Nichterkennung unter Umständen zur Doppel-Disposition (e.g. es werden zwei Wachen statt einer disponiert) kommt. Das Wort Konflikt wird im Zuge der Arbeit mehrdeutig verwendet:

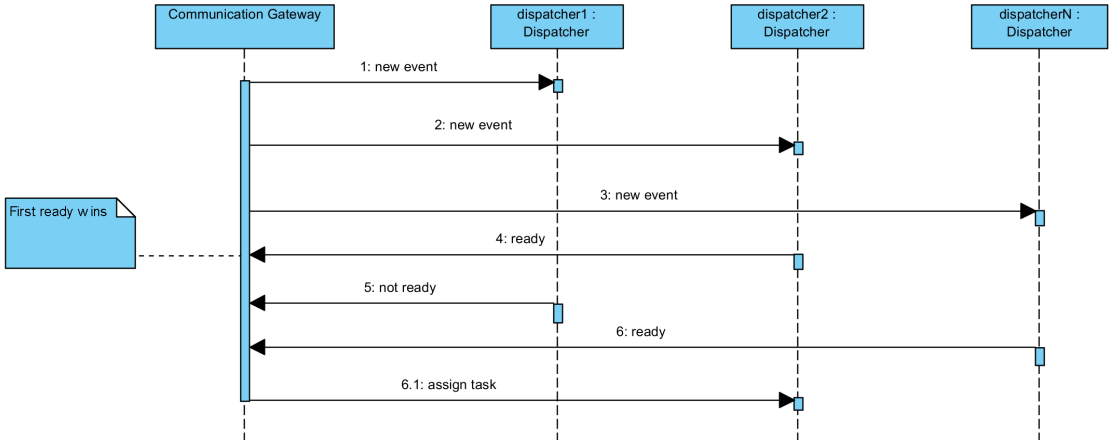

Abbildung 3.13: Sequenzdiagramm zeigt die Zuweisung eines SMS oder Brandmelderalarms

- Ein Konflikt entsteht, wenn zwei Disponenten einen Notruf eines gleichen Schadensereignisses bekommen und den Einsatz disponieren.

- Ein Konflikt entsteht, wenn zwei Disponenten am gleichen Einsatz (gemeint ist hier Incident und nicht Occurrence) die selben Daten (e.g. Adresse) zeitgleich ändern.

Im Zuge der Arbeit wird der kollaborative Einfluss des ersten Punkts genauer betrachtet. Wie bereits bei der Analyse von verbalen Gesprächen festgestellt wurde, ist die **Lokalität** eines Einsatzes einer der wichtigsten Kriterien zum Erkennen eines Konflikts. Es werden drei unterschiedliche Methoden zur aktiven Präventation diskutiert und analysiert:

- **Möglichkeit 1: Verbal**: Disponenten sitzen nebeneinander und können sich verbal verständigen. In diesem Fall können sie sich bis zu einem gewissen Einsatzaufkommen koordinieren, um Doppeldisposition zu vermeiden. Dies kann e.g. nötig sein, wenn das Einsatzleitsystem nur eine Liste von Einsätzen anzeigt und diese nur Adresse (Wehrgasse 11, 1050 Wien) und Einsatztyp (FEUY - Feuer Menschenleben in Gefahr) enthält. Hier müsste der Disponent die Liste jener Einsätze durcharbeiten, welche er nicht disponiert hat, um sicherzustellen, dass ein Kollege den Einsatz nicht bereits disponiert hat. Dies ist bei einer längeren Liste (welche wie beschrieben nur wenig Information zur Verfügung stellt) oft zeitlich nicht vertretbar bzw. fehlerbehaftet.

- **Möglichkeit 2: Chat**: Durch eine große Menge an Disponenten oder einem abgesetzten Arbeitsplatz (Leitstelle über ein WAN-Netzwerk verbunden), können Disponenten nicht verbal miteinander kommunizieren. Ein Chat kann Abhilfe schaffen. Auch kann hier die Ursache ähnlich jener von Möglichkeit 1 sein.

- **Möglichkeit 3: Automatische Detektion**: Bei dieser Methode versucht das System anhand einer Umgebungssuche den Disponent auf mögliche Konflikte hinzuweisen.

Um die Auswirkungen der Kollaboration besser verstehen zu können, wird der Ansatz der Simulation angewandt. Bevor die Möglichkeiten allerdings im Detail betrachtet werden, sollen zwei generische Szenarien definiert werden, welche anschließend auf alle drei Möglichkeiten angewandt werden.

3.3 Generische Szenarien

Ein generisches Szenario gibt im Großen und Ganzen ein kollaboratives Framework zwischen Notrufer, Disponent und Ressourcen an. In der vorliegenden Arbeit wird folgende Strategie angewandt:

- Es gibt zwei **Einsatztypen**: Small Occurrence (Kleinschadensereignis) und Big Occurrence (Großschadensereignis) (Abbildung 3.14)
  - Ein **Kleinschadensereignis** ist dadurch charakterisiert, da es nur von einem verbal Annunciator (verbaler Notrufer - e.g. Telefon über 122) gemeldet wird. Der Disponent nimmt den Notruf entgegen, gibt die Daten ins Einsatzleitsystem ein und disponiert Ressourcen.
  - Im **Großschadensereignis** informieren mehrere Annunciator (Melder) die Leitstelle. Dies können Automatic Alarm Initiating Devices (Automatische Alarmgeber wie e.g. Brandmelder) sein, oder Verbal Annunciator (Verbale Notrufer).
- Im Regelfall passieren deutlich mehr Kleinschadensereignisse als Großschadensereignisse. Wir nehmen in der Simulation ein Ratio zwischen **1 zu 10**.

Bei einem **Kleinschadensereignis** können ebenfalls vermeintliche Konflikte auftreten (False-Positiv-Rate). Folgendes Beispiel zeigt die Problematik: Ein Notrufer meldet Wassereintritt im Keller Franzensgasse 13, 1050 Wien. Ein zweiter Notrufer meldet herunterfallende Fassadenteile in der Franzensgasse 13, 1050 Wien. Die einfachste Form einer Konfliktdetektion (Berücksichtigung der Adresse) würde einen Konflikt anzeigen - durch Berücksichtigung des Einsatztypes würde das System allerdings schnell zwischen zwei unterschiedlichen Einsätzen klassifizieren können. Des Weiteren ist die Wahrscheinlichkeit dieses Szenarios so gering, dass es vernachlässigbar ist. Diese Details werden daher abstrahiert - die Klassifizierung des Einsatzes erfolgt **a priori** und besagt daher, dass das System bei Kleinschadensereignissen keinen Konflikt detektiert.

Hat der Disponent bei einem **Kleinschadensereignis** den Notruf beendet, führt er weitere schadensminimierende Tätigkeiten durch (e.g. die Disposition von Ressourcen).

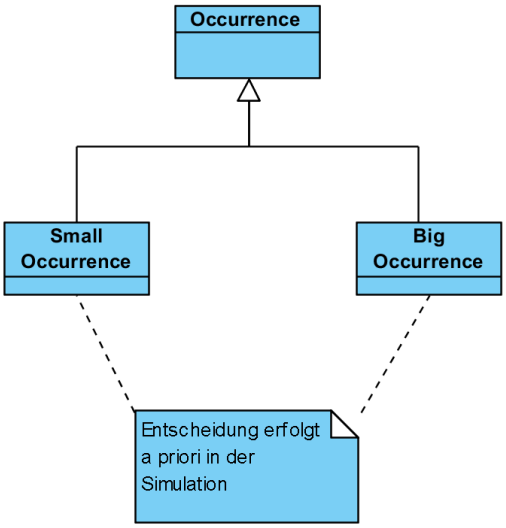

Abbildung 3.14: Zeigt die unterschiedlichen Typen. Zu beachten ist, dass in einem realen Einsatz die Klassifizierung erst im Nachhinein (Ex post) festgestellt werden kann

Dieses Detail wird dahingehend abstrahiert, dass der Disponent eine zeitlang beschäftigt ist. Das Kleinschadensereignis lässt sich wie in Abbildung 3.15 zusammenfassen.

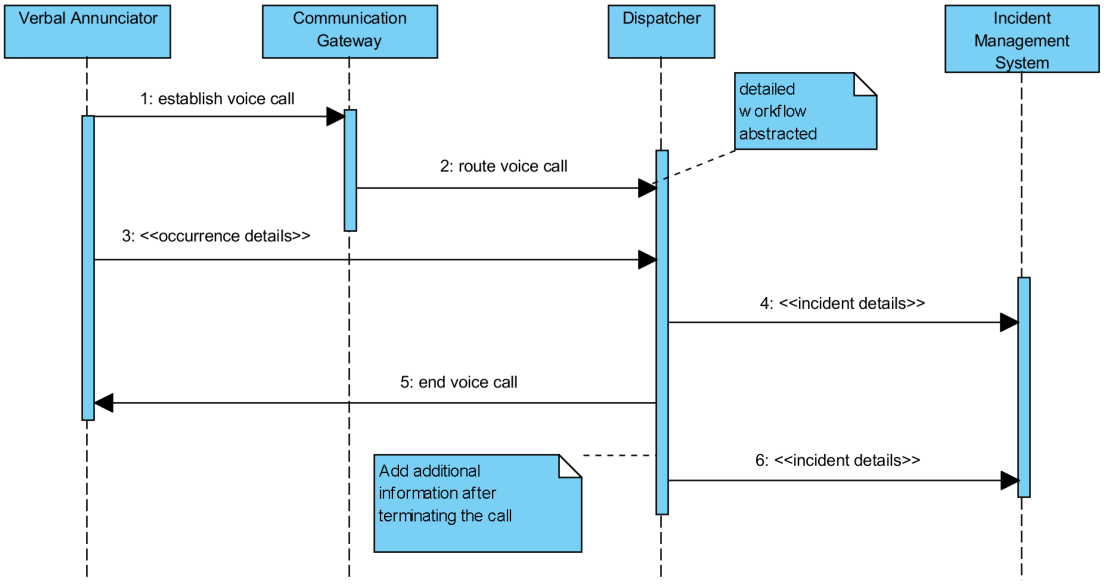

Abbildung 3.15: Ein Sequenzdiagramm eines Kleinschadensereignisses

Das **Großschadensereignis** gestaltet sich etwas schwieriger. Bei einem Großschadensereignis gibt es mehrere Instanzen von Annunciator und Dispatcher. Als Grundlage der

Analyse soll ein Sequenzdiagramm dienen, welches sich am Eingangs erwähnten Szenario orientiert (Abbildung 3.16).

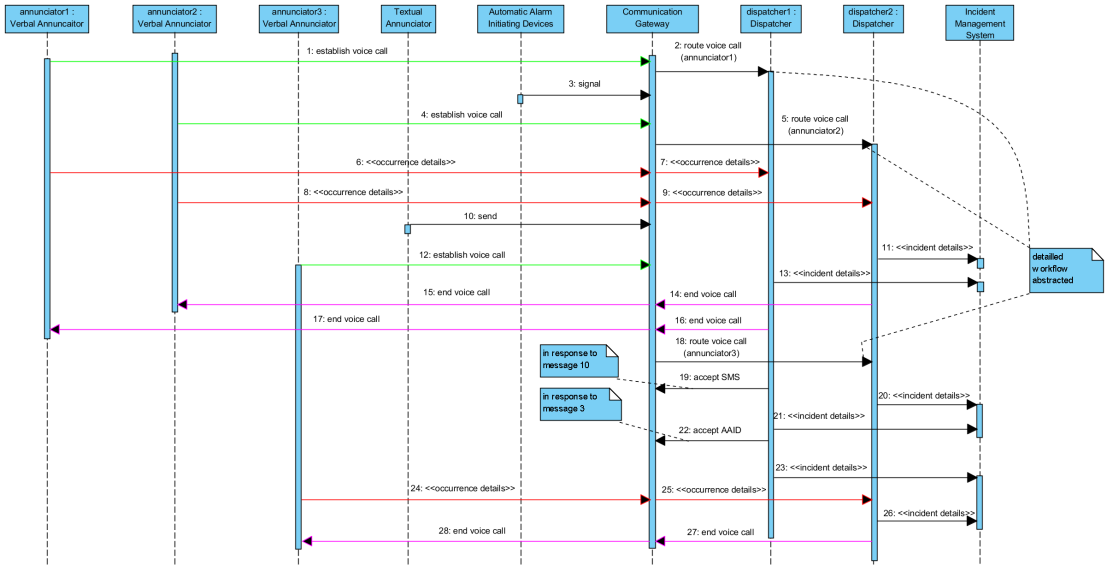

Abbildung 3.16: Sequenzdiagramm eines Großschadensereignisses.

Folgendes kann aus dem Großschadensereignis aus Abbildung 3.16 entnommen werden:

- Der Zeitpunkt von Establish Voice Call des Verbal Annunciators ist **zufällig**.
- Der Ressource Alocator nimmt den verbalen Notruf entgegen (grüne Nachricht 1,4 und 12) und verteilt ihn nach der Reihenfolge des Auftreffens im System (Nachricht 2,5 und 18)
- Wie bei der Analyse des Telefonats festgestellt, findet zwischen Verbal Annunciator und Dispatcher nur ein **unidirektionaler Informationsfluss** statt. Dieser kann durchaus (wie analysiert) auch in zwei Teilflüsse aufgeteilt werden (rote Nachrichten 6, 8 und 24 bzw. 7,9 und 25).
- Die Verteilung der Anrufe ist abstrahiert. 2, 5 und 18 wird durch den Pull-Style umgesetzt.
- Der Disponent kann dem Incident-Management-System mehrfach Incident-Details schicken. Er kann e.g. zuerst nur die Adresse schicken und dann weitere Details. Je nach dem, wie semantisch hochwertig die Daten sind und um so früher das System die Daten hat, um so früher kann das Incident-Managament-System auf Konflikte reagieren.

Um das komplexe Verhalten am Besten darstellen zu können, werden autonome Instanzen und deren Eigenschaften beschrieben - sogenannte **Software-Agenten**. Agenten in der

vorliegenden Umsetzung besitzen keine Möglichkeit des Lernens, sondern verfolgen ein fixes Regelwerk.

3.4 Simulations Entitäten

Im folgenden Abschnitt werden die Simulations Entitäten beschrieben. Simulations Entitäten haben u.U. mehr (in manchen Fällen auch weniger) Informationen als reale Entitäten.

3.4.1 Environment Entität

Ist die Umgebung bzw. die reale Welt. In dieser realen Welt gibt es Vorfälle (Occurrences). Die Aufgaben der Welt sind (Abbildung 3.17):

- Erstelle in zufällig gewählten Zeitabständen beliebig viele Occurrences. Dieser Vorgang setzt sich unendlich lang fort.
- Das Verhältnis zwischen den Occurrence-Eigenschaften Großschadensereignis und Kleinschadensereignis soll dabei zwischen 1 zu 10 sein.
- Erstelle in einer zufällig gewählten Zeit Annunciators, welche einer Occurrence zugewiesen werden (in anderen Worten: der Annunciators beobachtet eine Occurrence). *Verbal Annunciators* sollen häufiger als *Textual Annunciators* und Automatic *Alarm Initiating Devices* vorkommen. Die Anzahl der Annunciators soll zufällig sein.

3.4.2 Occurrence Entität

Die Occurrence ist das Vorkommnis in der realen Welt. Es besitzt folgende Eigenschaften (Abbildung 3.18):

- **Time**: Zeitpunkt, an dem die Occurrence passiert ist.
- **Dimension**: Gibt an, ob es sich um ein Großschadensereignis (e.g. Brand Hochhaus) oder Kleinschadensereignis (e.g. Brand Mistkübel) handelt. Diese Eigenschaft kann zwei Werte beinhalten:
  - Small Occurrence
  - Big Occurrence

Die Occurrence selbst besitzt keine Verhalten - sie dient dem Annunciator u.a. nur als Informationsquelle.

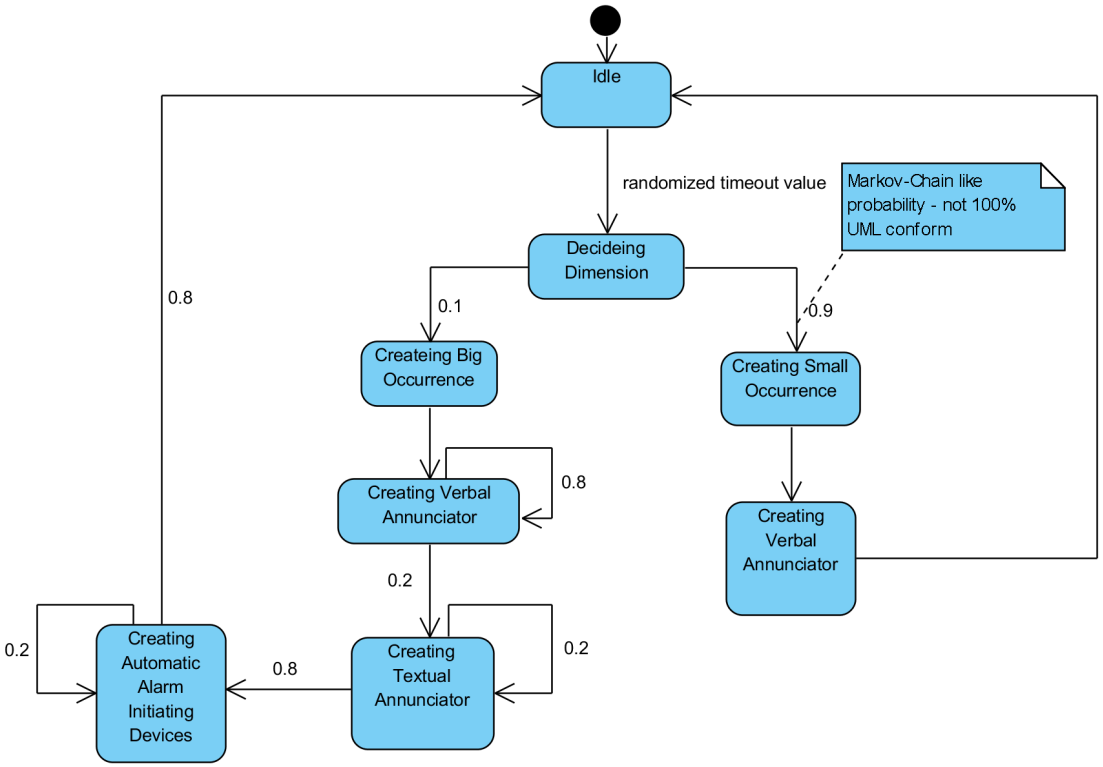

Abbildung 3.17: Eine State-Maschine des Environment aus Sicht des Simulationskontext. Die Übergänge geben Wahrscheinlichkeiten an.

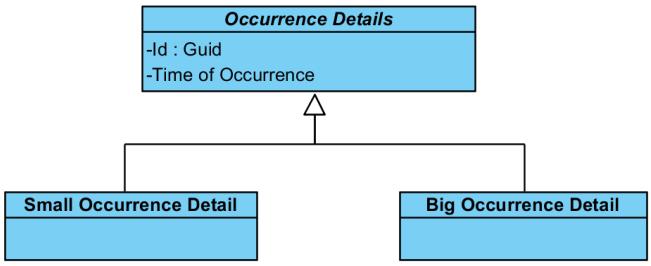

Abbildung 3.18: Die Occurrence Hierachie.

3.4.3 Annunciator Entität

Ein Annunciator ruft bzw. verbindet (im Simulationskontext schickt er eine Nachricht) nach einer gewissen (im Simulationskontext: zufälligen) Zeit zur Leitstelle. Anschließend übermittelt er (im Simulationskontext: nach einer zufälligen Zeit) die ihm zur Verfügung stehenden Informationen.

Anmerkung zur Simulation: Der Annunciator weiß in der realen Welt nicht, ob er eine Small Occurrence (Kleinschadensereignis) oder Big Occurrence (Großschadensereignis) beobachtet hat. Der Disponent kann durch richtige Fragen und Erfahrungen nur Vermutungen anstellen. Folgende Beispiele zeigen dies:

- Ein ausgedehnter Zimmerbrand in einem bewohnten 3 stöckigen Haus in Wien mit starker Rauchentwicklung
  - Es ist davon auszugehen, dass sich mehrere Notrufer melden werden
  - Der koordinative Aufwand der Einsatz-Ressourcen ist hoch
  - Die Informationsquelle zwischen Außenwelt und Leitstelle ist hoch
  - Es sind höchstwahrscheinlich mehrere Disponenten involviert
  - Es werden mehrere Incidents für die gleiche Occurrence angelegt
  - **Klasssifikation**: Big Occurrence
- Ein Brand einer Mülltone auf einer freistehenden Müllinsel
  - Es ist je nach Tageszeit davon auszugehen, dass sich nur eine Person meldet. Bleibt diese vor Ort stehen, werden andere Passanten davon ausgehen, dass dieser bereits angerufen hat und werden keine weiteren Maßnahmen setzen.
  - Der koordinative Aufwand der Einsatz-Ressourcen ist gering
  - Es ist höchstwahrscheinlich nur ein Disponent involviert
  - Die Informationsquelle zwischen Außenwelt und Leitstelle ist niedrig
  - Es wird nur in Incident für die Occurrence angelegt
  - **Klasssifikation**: Small Occurrence

Ein Disponent wird den Notrufer so viele Fragen wie möglich stellen, um zu erfahren, welche Ausmaße das Schadensereignis hat. Davon wird im Simulationskontext die Wahrscheinlichkeit abgleitet, wie viele Leute das Schadensereignis noch melden werden. Erfährt der Disponent, dass es sich um eine Big Occurrence handelt, versucht er nach Beendigung des Gesprächs festzustellen, ob es einen Konflikt gibt. Wie bereits festgestellt, kann dies durch drei unterschiedliche Arten geschehen: verbal, per Chat oder automatisch.

Der Inhalt von Verbal Annunciator Details ist in Abbildung 3.19 noch einmal zusammengefasst.

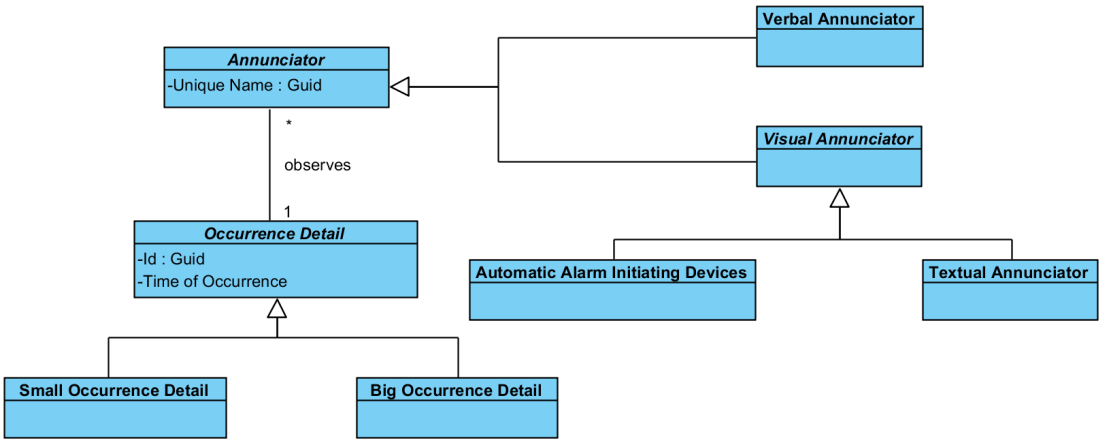

Abbildung 3.19: Zeigt die Informationen, welche ein Annunciator einer Leitstelle im Simulationskontext übermittelt

3.4.4 Communication Gateway Entität

Das Communication Gateway dient als Schnittstelle zwischen Außenwelt und Leitstelle. Die genaue Arbeitsweise wurde bereits für verbale und visuelle Notrufer analysiert.

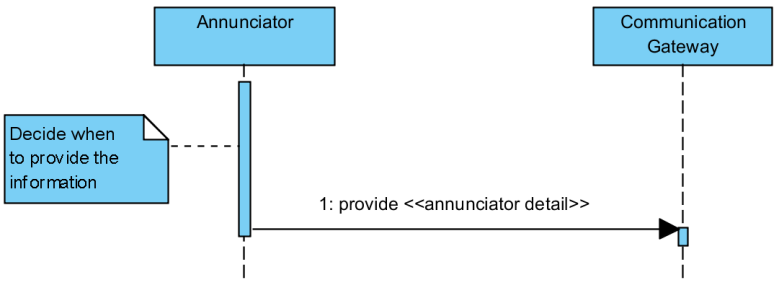

Abbildung 3.20: Sequenzdiagramm eines Verbal-Annunciators

Das Verhalten von einem Annunciator ist in Abbildung 3.20 noch einmal zusammengefasst. Im Fall von einem Verbal Annunciators wird das Wechselgespräch durch das Senden der Gesamtinformation abstrahiert.

Das Communication Gateway kann auch Anfragen vom Dispatcher bekommen. Dieser signalisiert mit *ready*, dass er bereit ist für die nächste Aufgabe. Das *Communication Gateway* weist im Simulationskontext die Aufgabe nach dem Eintreffzeitpunkt zu.

Für die Simulation ist es ausreichend, dass die unterschiedlichen Melder (Annunciator) dasselbe melden. Der Effekt macht sich in der Länge der Abarbeitung bemerkbar - Details werden in der Disponenten Entität abgehandelt.

3.4.5 Dispatcher Entität

Der Disponent (Dispatcher) hat die größte Wichtigkeit im System, da er jene Entität ist, welche zwischen Außenwelt und Ressourcen als koordinative Rolle agiert. Wie bereits erwähnt, besteht die Aufgabe des Disponenten darin, eingehende Daten zu interpretieren und Tätigkeiten zur Schadensminderung zu veranlassen. Dieser Vorgang ist sehr komplex, da die Entscheidung, welche Ressourcen am besten zur Minderung des Schadens eingesetzt werden können, sehr viel Erfahrung und Wissen voraussetzt. Eine Möglichkeit der Abstraktion im Zuge einer Simulation - wie bereits beschrieben - ist jene des Wissens im Nachhinein (Ex post). Hier geht man davon aus, dass der Disponent weiß, ob es sich um ein Großschadensereignis handelt oder nicht.

3.4.6 Small Occurrence

Hat der Disponent im Simulationkontext also das abstrahierte Wissen, dass es sich um eine Small Occurrence handelt, so muss er keine Überprüfung bezüglich Konflikte durchführen. Abbildung 3.21 zeigt die Interaktion mit einem Disponenten in einer Small Occurrence.

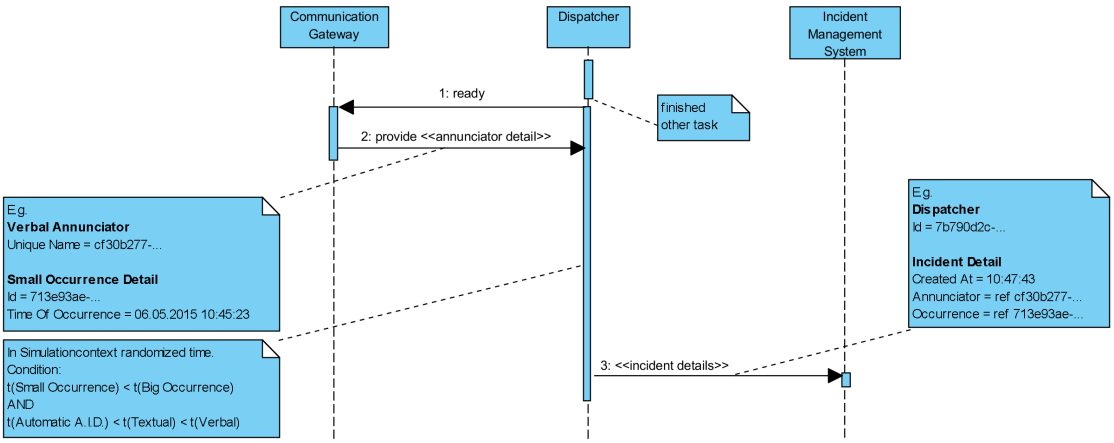

Abbildung 3.21: Ein Sequenzdiagramm einer Small Occurrence im Simulationskontext

Die Daten, welche im Zuge der Simulation zum Incident Management System geschickt werden, sind in Abbildung 3.22 zu sehen

Da die Daten Annunciator und Occurrence **idempotent** sind, werden sie der Einfachheit halber im Dispatcher-Detail nur referenziert.

3.4.7 Big Occurrence

Im Falle einer Big Occurrence ist mit Konflikten zu rechnen - sogar mit Sicherheit je nach Arbeitsprozess. Dazu muss der Arbeitsprozess eines Disponenten analysiert werden (An-

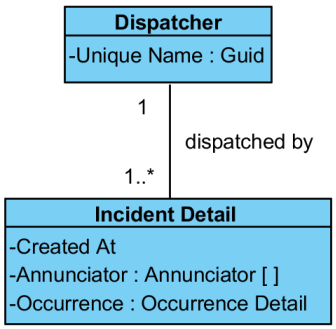

Abbildung 3.22: Incident-Detail und Dispatcher Entitäten und deren Zusammenhang.

merkung: die Arbeitsprozesse sind sehr facettenreich und können nicht alle berücksichtigt werden. Es wird nur ein in der Praxis angewandter Arbeitsprozess herangezogen):

- Wenn der Disponent einen Notruf annimmt, öffnet das CAD Programm einen neuen Incident und füllt initial Gesprächsbeginn und Notrufnummer aus.
- Der Disponent fragt den Occurrence-Details
- Während des Gesprächs kann das System bereits eine **Konfliktwarnung** anzeigen
- Durch die Warnung überprüft der Disponent vor der Disposition der Ressourcen auf Konflikte

Abbildung 3.23 zeigt ein mögliches Szenario aus der Sicht des Simulationskontextes.

Merging kann wie folgt beschrieben werden:

- Es erfolgt eine Gruppierung auf alle Incident Details, welche die gleiche Occurrence haben.
- *Created At* wird auf die aktuelle Zeit gesetzt.
- Alle Annunciators werden in den gemergten Incident kopiert.

Möglichkeit 1 im Detail: Verbal

Sitzen Disponenten in Reichweite, können sie miteinander kommunizieren. Es gelten dabei folgende Annahmen:

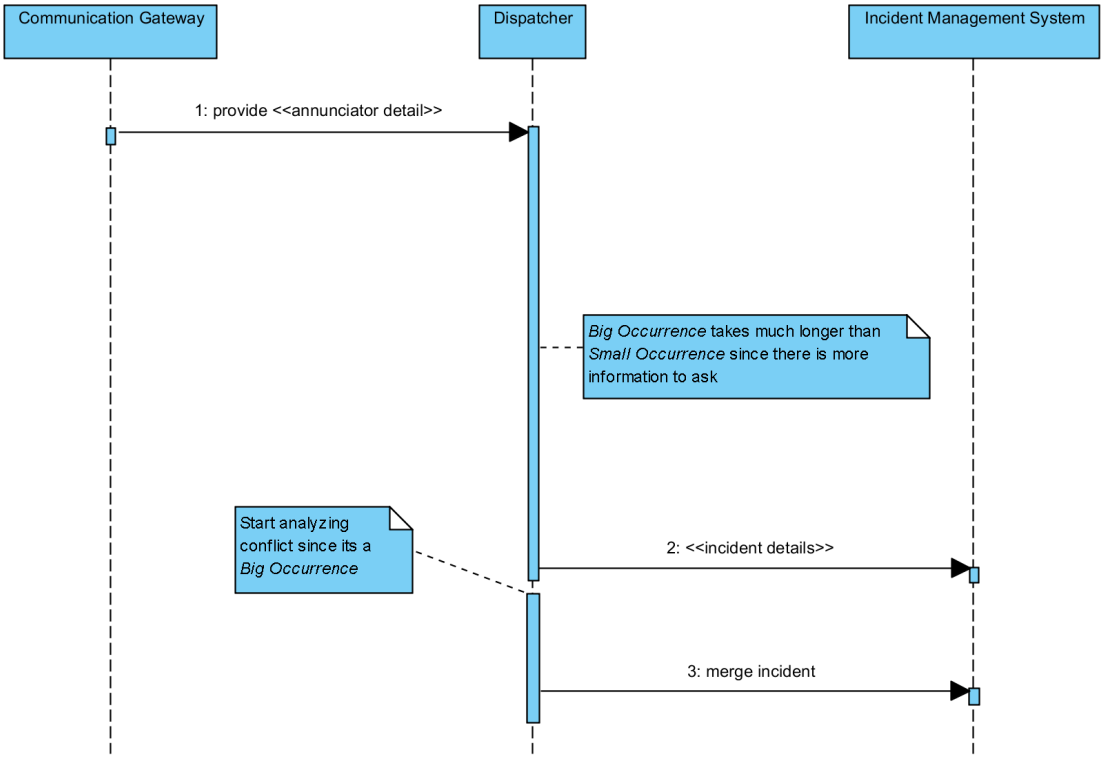

Abbildung 3.23: Sequenzdiagramm eines Big-Occurrence.

- Ein Disponent fragt alle anderen Disponenten gleichzeitig eine Frage. E.g. "Hat jemand schon einen Einsatz in der Stumpergasse 31 disponiert?"
- Disponenten antworten wenn sie Zeit haben. Um **Deadlocks** zu vermeiden, können andere Disponenten auch antworten, wenn sie selbst eine Frage gestellt haben.
- Der fragende Disponent kann erst wieder weiter machen, wenn er eine Antwort von allen Disponenten bekommen hat.
- Die Zeit der Antwortdauer pro Disponent ist abhängig von der Anzahl der Incidents, welcher der Disponent angelegt hat.

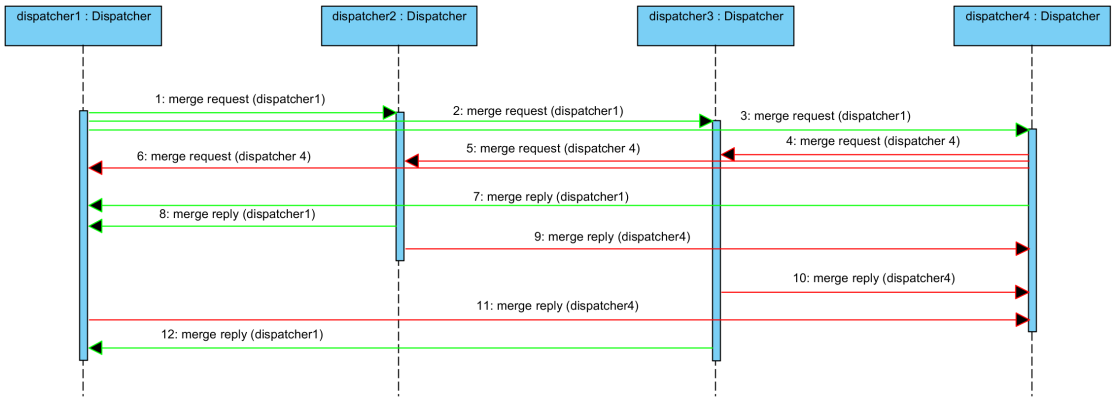

Abbildung 3.24: Sequenzdiagramm eines verbalen Merge-Prozesses. Es gibt immer eine Anfrage und nach einer gewissen Zeit eine Antwort. Es gibt keine kausalen Zusammenhänge im System - die Antwortzeit ist beliebig.

Abbildung 3.24 zeigt einen exemplarischen Ablauf. Aus dem Beispiel aus Abbildung 3.24 lässt sich folgendes entnehmen:

- Dispatcher 1 und Dispatcher 4 starten fast zeitgleich eine Anfrage an alle anderen Dispatcher
- Wie am Beispiel von Dispatcher 3 zu erkennen ist, kann die Reihenfolge der Antwort auch umgekehrt sein (10 vor 12). Dies spiegelt die mögliche Komplexität einer Fragestellung wieder.
- Anhand von Dispatcher 1 bzw. Dispatcher 4 ist zu erkennen, dass diese auch einen Request beantworten können, wenn sie selbst noch ausstehende Antworten haben.

Möglichkeit 2 im Detail: Chat

Sind Arbeitsplätze abgesetzt bzw. über ein WAN verbunden (unterschiedliche Räumlichkeiten), so ist Möglichkeit 1 nur über Telefon etc. möglich. Es soll an dieser Stelle auch ein Versuch einer textuellen Kommunikation untersucht werden. Diese gestaltet sich gleich jener der verbalen, braucht allerdings viel länger.

Möglichkeit 3 im Detail: Automatisch

Als dritte Option soll ein automatisches System untersucht werden. Dieses gestaltet sich im Kontrast zu Methode 1 und 2 zur Gänze anders. Im Fall einer automatischen Detektion ändert sich das Kollabortions-Pattern, da nicht mehr alle Disponenten befragt werden, sondern nur mehr das Incident Management System - im Speziellen eine Komponente namens *Conflict Detector*. Abbildung 3.25 zeigt den Ablauf von Methode 3.

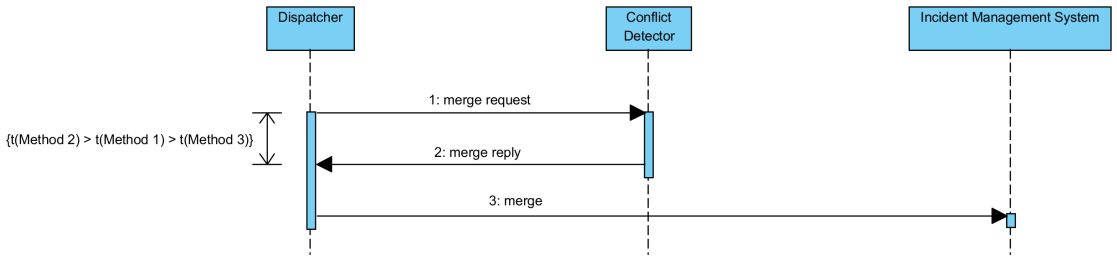

Abbildung 3.25: Automatisches Mergen

Wie in Abbildung 3.25 zu entnehmen ist, wurde im Kommentar ein zeitliches Constraint eingetragen.

3.5 hADL, LittleJIL und DomainPro

Die Beschreibung des oben analysierten kollaborativen Szenarios wird in zwei Teile unterteilt:

- einen strukturellen Teil, welcher die einzelnen Entitäten beschreibt
- einen Verhaltensteil, welcher Abfolge von Tätigkeiten beschreibt

Der strukturelle Teil wird mit Hilfe von hADL umgesetzt. Es werden die Elemente Connector, Component bzw. Artifact und Message verwendet, um die Entitäten einer Leitstelle zu beschreiben. Folgende Features sollen umgesetzt werden:

- Transformation von ConnectorType, ComponentType bzw. ArtifactType und MessageType
- Transformation von Relations: Für Relations sollen Variablen angelegt werden. Der Typ von Variablen soll automatisch bestimmt werden.
- Transformation von Actions: Diese werden nur für ArtifactType oder MessageType umgesetzt. Die Umsetzung soll in Form von Methoden erfolgen.

Für das Verhalten der Disponenten wird die Sprache LittleJIL verwendet. Es sollen folgende Features umgesetzt werden:

- Umsetzen von beliebigen Step-Hierachien. Ein Step kann Child-Steps haben - diese sollen berücksichtigt werden.
  - Child-Steps können sequentiell oder parallel ausgeführt werden. Welche Variationen im Detail umgesetzt werden, wird in den folgenden Punkten spezifiziert.

- Berücksichtigen von Agenten für jeden Step. Agenten (ausführende Instanzen) sollen für jeden Step ermittelt und erzeugt werden. Der Entwickler soll über das Simulationstool DomainPro (Simulations Initialisierung) oder zur Laufzeit weitere Instanzen erstellen können. Ist der Agent nicht explizit in einem Step gesetzt, so soll ein Agent vom ersten Parent Knoten genommen werden, bei dem der Agent gesetzt ist. Ist kein Agent gesetzt, soll die Transformation abgebrochen werden.
- Data-Binding von einfachen Variablen (ausgenommen sind Listen). Data-Binding soll für sequentielle als auch für parallele Steps funktionieren.
  - Eingehende Referenzen: Es soll möglich sein, Variablen von einem Parent Step zu übernehmen. Diese sollen automatisch gesetzt werden. Der Parent-Step muss der unmittelbare Parent sein.
  - Ausgehende Referenzen: Es soll möglich sein, Variablen an den Parent Step zu übergeben. Auch hier muss der Parent der unmittelbare Parent sein.
- Für sequentielle Steps kann über Ranges iteriert werden. Es sollen Ranges der Form a..b unterstützt werden, wobei a und b nummerische Werte sind (bzw. b kann auch * sein)
- Für sequentielle Steps kann über Listen aus Ressourcen iteriert werden (e.g. *has Cardinality: 0..* for elements in: masterTaskReferenceList*). Convention: Listen haben den Postfix *List*. Die Listen Elemente der Iteration werden automatisch an jene Child-Steps übergeben, welche die Liste im Data-Binding haben.
- Für sequentielle Steps kann über Listen aus Agenten iteriert werden. Die Methoden werden anschließend auf die Agenten angewandt.
- Für parallele Steps kann über Ranges iteriert werden - selbe Anforderungen wie für sequentielle Steps.
- Für parallele Steps kann über Listen iteriert werden - selbe Anforderungen wie für sequentielle Steps.

Für die jeweiligen Anforderungen müssen spezifische Lösungen gefunden werden, da DomainPro Step-Hierachien nicht direkt unterstützt. Lösungen werden im nächsten Kapitel diskutiert.

3.6 Unterstützte Coordination Patterns

Im Zuge der Transformation werden folgende Patterns unterstützt (Beschreibung der Patterns siehe Kapitel 2):

- **Shared Artifact**: Es soll möglich sein, ein Artifact instanzieren zu können und Methoden auf der Instanz aufrufen zu können. Des Weiteren soll ein Artifact versionierbar sein und man soll es als gelöscht markieren können.

- **Master/Worker**: Es soll möglich sein, über eine Liste von Aufgaben sequentiell iterieren zu können und diese Agenten zuzuweisen.
- **Publish/Subscriber**: Auch hier soll die sequentielle Iteration über eine Liste von Agenten möglich sein. Subscribing / Unsubscribing muss vom Simulationsentwickler selbst umgesetzt werden.

Ziel des Transformationstools soll es sein, die Patterns zu unterstützen - wenn auch nur in semi-automatischer Transformation (Eingriff des Simulations-Entwickler erforderlich).

3.7 Simulationaspekte im Skeleton

Die Elemente, welche für DomainPro generiert werden, sollen automatisch angeordnet werden. Es ist dabei ausreichend, die Verschachtelung der Elemente zu berücksichtigen.

Neben den Elementen, welche generiert werden, wird auch Code in den Methoden generiert. Um die Arbeit dem Simulationsentwickler zu erleichtern, sollen Default-Agent Instanzen in jeder Methode generiert werden. Diese können vom Entwickler beliebig geändert werden. Um die Verfolgung des Simulationsablaufs zu erleichtern, soll jede Methode ihren vollständigen Namen (Methodenname und Agent) ausgeben. Container-Step-Instanzen (welche sich im Agent-Container befinden) sollen automatisch erzeugt werden.

Ebenfalls soll es dem Simulationsentwickler einfach möglich sein, auf den Child-Step bzw. Parent-Step zuzugreifen.

Die Nomenklatur für diverse Elemente ist wie folgt:

- ConnectorType, ComponentType bzw. ArtifactType und MessageType werden gleich benannt wie in hADL spezifziert.
- Agenten von Steps werden auch so benannt wie in LittleJIL bzw. hADL spezifziert.
- Container für Steps werden mit dem Postfix *Container* benannt.
- Methodennamen für Steps sollen einen sprechenden Namen haben, sodass der Entwickler diese einfach der LittleJIL Struktur zuordnen kann.

3.8 Anforderungen an das Tool

Da das Tool iterativ entwickelt wird, soll jede Iteration über ausreichend Integrationstests (in Form von Regressions-Tests) verfügen. Tests sollen die Transformation von einer DSL zum fertigen Modell beschreiben. Es soll dem Simulations-Entwickler einfach möglich sein, Glue-Code einzufügen.

Das Tool soll in .NET / C# entwickelt werden und unter Windows lauffähig sein.

KAPITEL 4

Entwurf

Wie im Kapitel 2 festgehalten wurde, gibt es generell drei Arten der Modell-Transformation:

- Direct Model Manipulation
- Intermediate Representation
- Transformation Language Support

Die in der vorliegenden Arbeit angewandte Strategie entspricht jener der Intermediate Representation. Das Modell (welches in einer DSL repräsentiert ist), wird im ersten Schritt nach XML (Extensible Markup Language) exportiert. Es erfolgt anschließend eine Transformation in das Simulations-Modell, welches ebenfalls XML als Intermediate Representation nutzt. Die Transformation erfolgt anhand eines Regelwerks, welches anschließend diskutiert wird.

Die transformierte XML Struktur kann anschließend mit dem Tool DomainPro eingelesen werden. Dieses interpretiert das XML und wandelt es in C# Code um, welcher anschließend ausgeführt werden kann. Zu beachten ist, dass es sich bei DomainPro um ein generisches Simulationstool handelt. Es ist in vielen Fällen daher nicht möglich, eine Eins-zu-eins Umsetzung vom Ausgangsmodell ins Zielmodell zu erreichen.

4.1 Die Transformation in einer groben Übersicht

Die Transformation erfolgt in mehreren Schritten (Abbildung 4.1). Im ersten Schritt werden aus dem Ausgangsmodell alle hADL Elemente (ConnectorType, ComponentType bzw. ArtifactType und MessageType) verarbeitet und in das Zielmodell transformiert. Das hADL Feature der Realtions und Links werden ebenfalls direkt in das Zielmodell umgesetzt.

Der weitaus komplexere Transformationsschritt ist jener von LittleJIL. Eine Eins-zu-eins Umsetzung in das Zielmodell ist nicht möglich, da SSubstepsünd bedingte Ausführungen nur über Umwege möglich sind. Um ein gutes Scoping zu erreichen, wird für jeden Step (inkludiert auch Substeps und Rootsteps) ein Container angelegt (selber Name wie Step - mit dem Postfix *Container*). Die Umwandlung der LittleJIL Features "parallele Abarbeitungünd ßequentielle Abarbeitung"der ChildSteps erfordert eine unterschiedliche Herangehensweise. Ersteres kann nur durch Generierung von Code in der aufrufenden Methode umgesetzt werden, Letzteres durch das im Zielmodell vorhandene Feature der Method-Sequences (Pfeile).

Ebenfalls ist die Transferierung der Daten eine Herausforderung. Diese erfordert wieder eine Separierung von paralleler bzw. sequentieller Abarbeitung der Child-Steps. Die Transferierung der Daten im parallelen Fall muss in der aufrufenden Methode erledigt werden - jene im sequentiellen Fall kann in der Pfeil Eigenschaft *Transfer* umgesetzt werden. Die Transformation muss mehrere Fälle berücksichtigen:

- Transferierung von ëinfachen"Daten
- Transferierung von Agenten
- Transferierung / Iteration über Listen

Letzteres stellt eine große Herausforderung da, weil der Fall nicht unmittelbar erkennbar ist. Eine Convention wird daher eingeführt: Listen müssen mit dem Postfix *List* enden.

Der letzte Schritt ist die Überführung des Glue-Codes in das Zielmodell.

4.2 Struktur und Verhalten

Wie bereits im Kapitel 2 festgehalten wurde, besteht das Ausgangsmodell (welches das Environment beschreibt) aus einem Strukturteil (hADL) und einem Verhaltensteil (LittleJIL). Begonnen wird mit dem Strukturteil.

4.2.1 Das Architecture Element (hADL)

hADL verfügt über mehrere Architecture Elemente (kurz ArchElement). Diese wurden im Kapitel 2 genau erläutert. Sie bestehen in der Intermediate Representation aus folgenden Elementen:

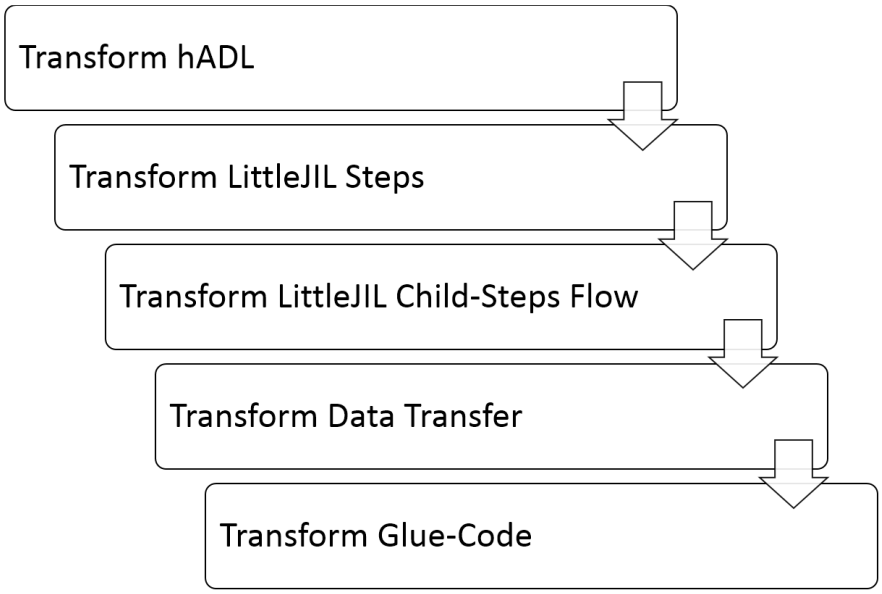

Abbildung 4.1: Übersicht des Transformationsprozesses

- **ObjectType** (entspricht CollaborationObjects)
  - ArtifactType: Entspricht SharedArtifacts.
  - StreamType: Entspricht Streams.
  - RequestType: Entspricht dem Request-Pattern
  - MessageType: Entspricht Messages.
- **CollaboratorType**
  - ConnectorType: dies entspricht den CollaborationConnectors. Sind somit für die effiziente und effektive Interaktion zwischen HumanComponents verantwortlich.
  - ComponentType: dies entspricht den HumanComponents. Sind die klassischen Entscheidungsträger (decision maker).

Besondere Behandlung erfährt der ArtifactType, da ihm noch zwei Variablen hinzugefügt werden:

- **IsDeleted**: gibt wieder, ob ein Artifact bereits gelöscht ist oder nicht.
- **ArtVersion**: wird bei jeder Änderung um eins erhöht und spiegelt somit den Versionsstand des Artifact wieder.

Actions werden nur für Artifact und Messages umgesetzt. Ist die Kardinalität gesetzt, so wird zusätzlich eine Variable mit dem Präfix *cvar* angelegt, welche Referenzen des Aufrufers halten kann (Beispiel in den folgenden Unterpunkten).

Relations (hADL)

In der hADL DSL gibt es Relations, welche Abhängigkeit zwischen Komponenten abbilden. Ein Beispiel wäre, wenn ein Einsatz (engl. Emergency) einen Vorfall (engl. Occurrence) referenziert (Codebeispiel 4.1).

Algorithm 4.1: Beispiel einer Struktur in hADL DSL

```
1  package cad
2  hADL Model name: cad
3  {
4    Structure name: world
5    {
6      // A Occurrence is something that happened in the environment. It may be a traffic accident or
           a structural fire or something else.
7      Artifact name: Occurrence
8
9      // An Emergency Call happens between an arbitrary Annunciator and a CommunicationGateway. It
           may be a voice call or a simple flow of data. A voice call is abstracted by an
           unidirectional flow of information
10     Message name: EmergencyCall
11
12     Relations {
13       [occurrenceRef : EmergencyCall references cad.world.Occurrence]
14     }
15   }
16 }
```

Das hätte zur Folge, dass im Emergency Container eine Variable *occurrenceRef* vom Typ *cad.world.Occurrence* angelegt wird. Die XML Repräsentation des soeben gezeigten Beispiels ist im Codebeispiel 4.2 zu sehen.

Algorithm 4.2: Intermediate Representation des Codebeispiels 4.1

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <dsl:Model xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:dsl="http://www.tuwien.at/
       dsg/Dsl" package="cad">
3    <hadl name="cad">
4      <structures name="world">
5        <elements xsi:type="dsl:ArchElement" name="Occurrence">
6          <type xsi:type="dsl:ArtifactType"/>
7        </elements>
8        <elements xsi:type="dsl:ArchElement" name="EmergencyCall">
9          <type xsi:type="dsl:MessageType"/>
10       </elements>
11       <relations name="occurrenceRef" from="dispatchXML.dsl#//@hadl/@structures.0/@elements.1" to="
             dispatchXML.dsl#//@hadl/@structures.0/@elements.0">
12         <type xsi:type="dsl:References"/>
13       </relations>
14     </structures>
15   </hadl>
16 </dsl:Model>
```

Das im Zuge der Arbeit entwickelte Tool interpretiert das XML mit den bis jetzt erklärten

Schritten und erzeugt ein XML, welches mit dem DomainPro Designer geöffnet werden kann. Es wird ein Modell wie in Abbildung 4.2 dargestellt.

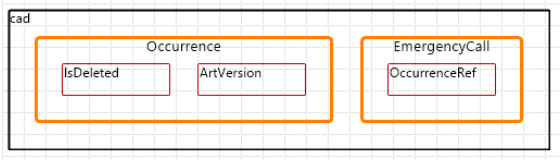

Abbildung 4.2: Ergebnis der Transformation

Für das Articfact Occurrence wurde eine DataObjType-Komponente (orange) erzeugt. Diese dient als Container für weitere Elemente, welche in den folgenden Abschnitten diskutiert werden. Wie bereits erwähnt, werden für ein Artifact zusätzlich zwei Variablen eingefügt (rote Container *IsDeleted* und *ArtVersion*). Die Variable *occurrenceRef* wurde aus der Relation des Ausgangsmodells abgeleitet.

4.2.2 Actions (hADL)

Das Konzept der HumanActions und ObjectActions wurde im Kapitel 2 bereits detailliert beleuchtet. HumanActions werden in der jetzigen Version nicht unterstützt und werden ignoriert. ObjectActions werden in Methoden umgewandelt (Codebeispiel 4.3).

Algorithm 4.3: Beispiel mit hADL Actions

```
1  package cad
2  hADL Model name: cad
3  {
4    Structure name: world
5    {
6      Component name: Dispatcher
7        Actions
8          [someTriggerAction primitives: [C]]
9
10     // A Occurrence is something that happened in the environment. It may be a traffic accident or
           a structural fire or something else.
11     Artifact name: Occurrence
12       Actions
13         [anExampleActionForAnOccurrence primitives: [R] cardinality: 0..*]
14
15     // An Emergency Call happens between an arbitrary Annunciator and a CommunicationGateway. It
           may be a voice call or a simple flow of data. A voice call is abstracted by an
           unidirectional flow of information
16     Message name: EmergencyCall
17       Actions
18         [anExampleActionForAnEmergencyCall primitives: [C] cardinality: 0..*]
19
20     Link [doSomething: cad.world.Dispatcher.someTriggerAction => cad.world.EmergencyCall.
           anExampleActionForAnEmergencyCall]
21     Relations {
22       [occurrenceRef : EmergencyCall references cad.world.Occurrence]
23     }
24   }
25 }
```

Codebeispiel 4.3 wird in das Modell, wie in Abbildung 4.3 gezeigt, transformiert. Blaue Boxen repräsentieren Methoden, welche aufgerufen werden können. Die Verwendung

dieser Methoden wird in diesem Kapitel noch klarer werden, wenn diese im LittleJIL Teil verwendet werden.

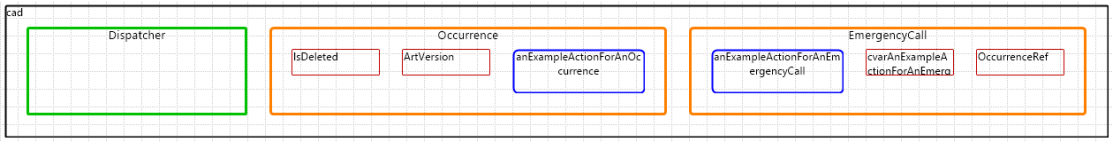

Abbildung 4.3: Ergebnis der Transformation

Es wurde ebenfalls eine Variable *cvarAnExampleActionForAnEmergencyCall* vom Typ *List<Dispatcher>* angelegt. Dies passiert nur, wenn

- Die Kardinalität gesetzt ist
- Der Datentyp ermittelt werden kann

Der Datentyp kann durch den Link ermittelt werden (Codebeispiel 4.4). Die Kardinalität besagt, dass ein Emergency mehrere Instanzen eines Dispatcher halten kann und der Dispatcher Emergencies erstellen kann.

Algorithm 4.4: Beispiel eines Links in der DSL

```
1  Link [doSomething: cad.world.Dispatcher.someTriggerAction => cad.world.EmergencyCall.
       anExampleActionForAnEmergencyCall]
```

4.3 Transformation von hADL

Nachdem die wesentlichen Transformationsschritte diskutiert wurden, werden nun die Typen des Ausgangsmodells genauer betrachtet. Diese werden vom Simulationstool DomainPro zur Verfügung gestellt.

ConnectorType als auch ComponentType werden im Simulationskontext als *Simulation.DomainProAbstractStructure* umgesetzt (implementiert in Simulation.ComponentType). Neben ConnectorType und ComponentType werden im Zuge der Arbeit weiters ArtifactType und MessageType umgesetzt, da das Verhalten der anderen Typen eine Vereinfachung der umgesetzten Typen ist. Diese werden jeweils in ein sogenannten *Simulation.DomainProAbstractType* - mit der Implementierung *Simulation.DataObjType* - umgewandelt.

Die Ableitungshierachie (Abbildung 4.4) ist vom Simulationstool vorgegeben und wird vom Transformationstool erzeugt und anschließend in XML serialisiert. Die wichtigsten Bausteine der Simulations Hierarchie sind:

- DomainProBase: Jedes Element in der Simulation hat eine eindeutige Id (umgesetzt durch einen Globally Unique Identifier - kurz GUID). Des Weiteren wird zwischen Typen und strukturellen Elementen unterschieden.
  - DomainProAbstractStructure: Enthält die wichtigsten Strukturen. Jede Struktur besteht aus beliebig vielen DomainProAbstractSemanticType Elementen (siehe unten).
    - VariableTypeStructure: zur Umsetzung von Variablen
    - ComponentTypeStructure: zur Umsetzung von Komponenten
    - MethodSequenceStructure: zur Umsetzung von Abläufen in Form von unidirektionalen Pfeilen
    - DataObjTypeStructure: zur Umsetzung von Datenobjekten
    - MethodTypeStructure: zur Umsetzung von Methoden
  - DomainProAbstractType: Besteht unter anderem aus:
    - DomainProAbstractSemanticType: Hier befinden sich die wichtigsten Typen, für den Aufbau der Simulation
    - VariableType: besteht aus einem Namen, Type und einem initialen Wert
    - MethodSequence: Besteht aus einem Origin und einer Destination - jeweils Methoden der Simulation
    - MethodType: Besteht aus einem Namen
    - ComponentType: Besteht aus einem Namen
    - DataObjType: Besteht aus einem Namen
    - DomainProAbstractModelType: Das Wurzel Element der Simulation

Zu beachten ist, dass eine beliebige Verschachtelung möglich ist - die gewählte Aufzählung jedoch nur die Verwendung im Zuge der Arbeit wiederspiegelt.

Der Pseudo-Code in Algorithmus 4.5 fasst die Schritte nochmals zusammen.

Algorithm 4.5: Pseudocode für die Transformation von hADL Elementen

```
 1  for each element in InputModel if type is ConnectorType or ComponentType
 2    create a Simulation.ComponentType
 3    create variables for relations
 4
 5  for each element in InputModel if type is ArtifactType or MessageType
 6    create a Simulation.DataObjType
 7
 8    if type is ArtifactType
 9      Create Delete and Version variable
10    fi
11
12    create variables for relations
13    create methods for actions
14       if cardinality from action is set
15          create cvar variable
```

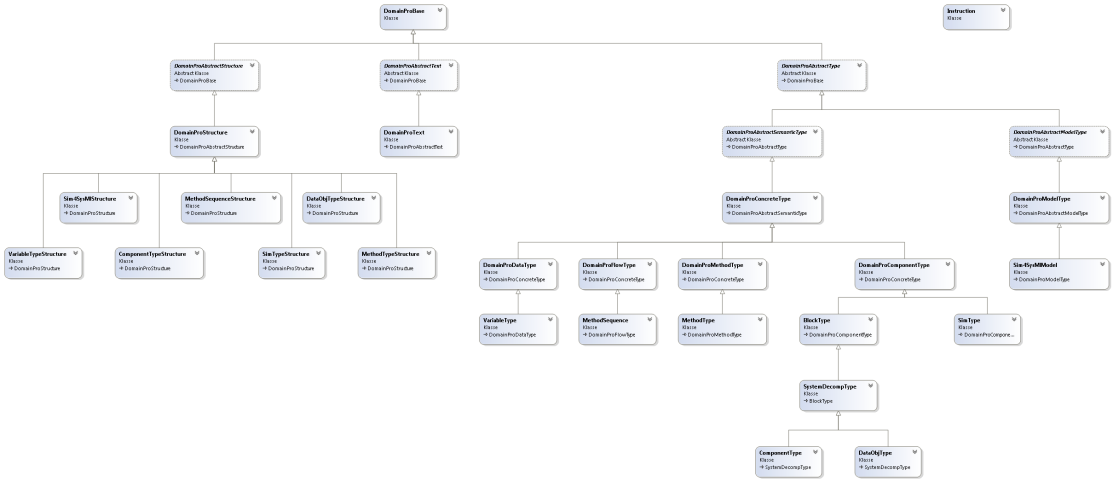

Abbildung 4.4: Das Klassendiagram der Ableitungshierachie

4.4 Steps Hierachie (LittleJIL)

Mit Hilfe der Prozessbeschreibungssprache LittleJIL kann hierarchisch ein Prozess abgebildet werden. Im Zuge der Arbeit wurden nicht alle Features von LittleJIL umgesetzt.

Ein essentieller Baustein von LittleJIL sind Steps - wie im Codebeispiel 4.6 gezeigt.

Das im Codebeispiel 4.6 gezeigte Beispiel enthält folgende Kernaspekte:

- Es handelt sich um eine sequentielle Abarbeitung der Steps (im Kontrast zur parallelen Abarbeitung)
  - Es wird zuerst *getCash* aufgerufen, nach Beendigung *getIngredient*. *getIngredient* wiederum ruft *getMilk*, *getSalt* und *getMeat* auf.
- Jeder Step hat einen zugewiesenen Agenten. Ein Agent kann ein Mensch oder ein automatisiertes System sein (e.g. Kommunikationssystem). Ein Agent ist eine Entität, welcher Arbeit zugewiesen werden kann und welche diese abarbeitet. Ist kein Agent angegeben, wird jener des Vorgängers genommen (beliebige Tiefe). Das Beispiel verfügt über zwei Agenten: *Person* und *EnvControl*

4.5 Transformation einer Steps Hierachie (LittleJIL)

Die Transformation erfolgt in mehreren Schritten. Im ersten Schritt werden für Agenten Container angelegt - für das Beispiel (Abbildung 4.5) wären das zwei Container: *Person* und *EnvControl*. Anschließend werden für jeden Step Container angelegt (mit dem Postfix *_Container*), welche sich im jeweiligen Container des Agenten (ausführende Instanz) befinden. Abbildung 4.5 zeigt das Resultat.

Algorithm 4.6: Beispiel von Step-Hierachien

```
1  package simpleSeq
2  hADL Model name: loop
3  {
4    Structure name: ex
5    {
6      Component name: Person
7      Artifact name: EnvControl
8    }
9  }
10 LittleJIL Process name: shopping
11   Root Steps {
12     Step name: shoppingRoot of type Sequential
13     Local Data: {
14       [ agent as Resource of hADLType loop.ex.EnvControl]
15     }
16     Child Steps: {
17       Step name: getCash of type Leaf
18       Local Data: {
19          [ agent as Resource of hADLType loop.ex.Person]
20       }
21       Step name: getIngredient of type Sequential
22       Local Data: {
23         [ agent as Resource of hADLType loop.ex.Person]
24       }
25       Child Steps: {
26         Step name: getMilk of type Leaf
27         Step name: getSalt of type Leaf
28         Step name: getMeat of type Sequential
29         Child Steps: {
30           Step name: getPork of type Leaf
31           Step name: getBeaf of type Leaf
32       }
33     }
34   }
35 }
```

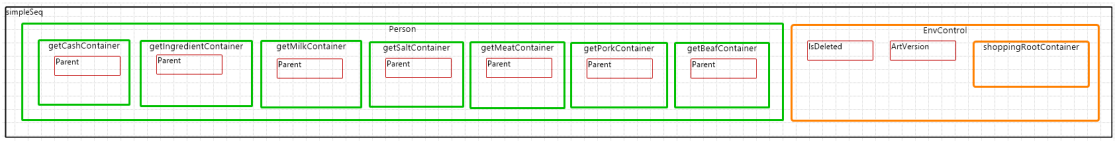

Abbildung 4.5: Ergebnis der Transformation

Wie aus Abbildung 4.5 zu entnehmen ist, wurde auch in jedem Step-Container eine Parent Variable angelegt, welche den Typ des übergeordneten Step-Container besitzt. So hat die Parent Variable des Containers *getPorkContainer* den Typ *PersonContext.getMeatContainer*. Algorithmus 4.7 zeigt den Pseudocode des Vorgangs.

Container alleine machen noch nichts. Daher ist es erforderlich, Methoden in den Container anzulegen und die hierarchische Abfolge abzubilden. Die Umsetzung wird anhand Abbildung 4.6 diskutiert.

Wie in Abbildung 4.6 zu erkennen ist, haben vor allem Steps, welche ChildSteps besitzen, mehrere Methoden. Dies ist damit zu begründen, dass das Simulations-Tool nur unidirektionale Methodenaufrufe kennt. In den "ReturnMethoden (e.g. *getIntgredient_getSalt*) kann der Entwickler Code hinterlegen, welcher ausgeführt werden soll, wenn e.g. *getSalt*

Algorithm 4.7: Pseudocode für das Erzeugen von Step-Container

```
1  input: ArtifactType / MessageType resp. ComponentType / ConnectorType from previous Step as
       component
2
3  for each step in the InputModel
4    if agent from step equals component
5      create a step container
6      add parent variable to reference parent step
7    fi
```

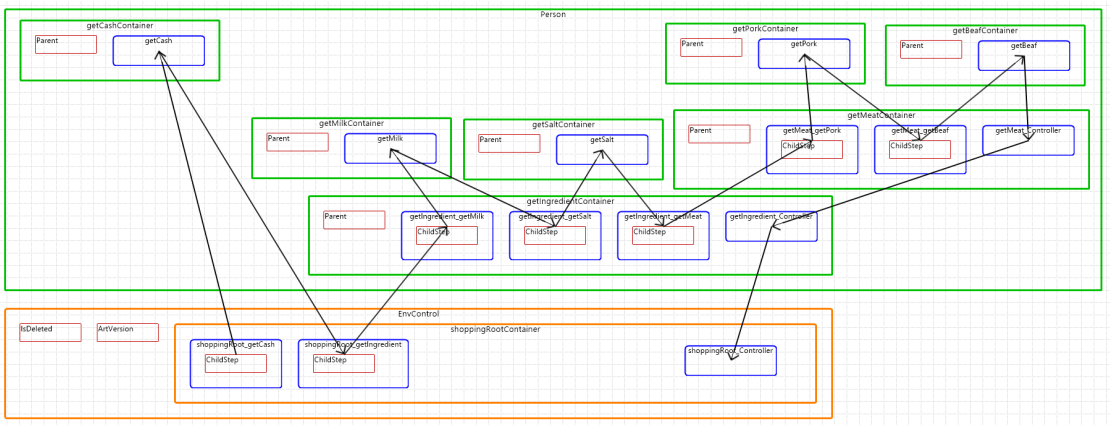

Abbildung 4.6: Step-Container mit Method-Sequences

abgearbeitet wurde. Im Ausgangsmodell hatte *getIngredient* folgende ChildSteps:

- getMilk
- getSalt
- getMeat

Es wurde für den getIngredient Step ein Container *getIngredientContainer* angelegt, welcher für jedes ChildStep Methoden beinhaltet. Die Nomenklatur lautet wie folgt: *<name of step>_<name of child step>*. Die Methoden mit dem Postfix *_Controller* dienen als Rückkehrmethode für Child-Steps.

Die Methoden haben eine Variable *ChildStep*, welche die Referenz auf den Child-Step gesetzt haben.

4.6 Methoden Sequenzen

Wie bereits erwähnt, unterstützt das Simulationstool undirektionale Pfeile, welche es erlauben zu spezifizieren, welche Methode nach der Abarbeitung aufgerufen werden soll. Der Pfeil hat drei Werte:

- Trigger: Wann soll der Pfeil ausgeführt werden?
- Transfer: Daten-Referenzen, welche übertragen werden sollen
- Resolve: Die Destination Instanz

Abbildung 4.7 zeigt einen Ausschnitt aus dem obigen Beispiel. Es soll der Pfeil zwischen *getIngredient_getMilk* und *getMilk* betrachtet werden.

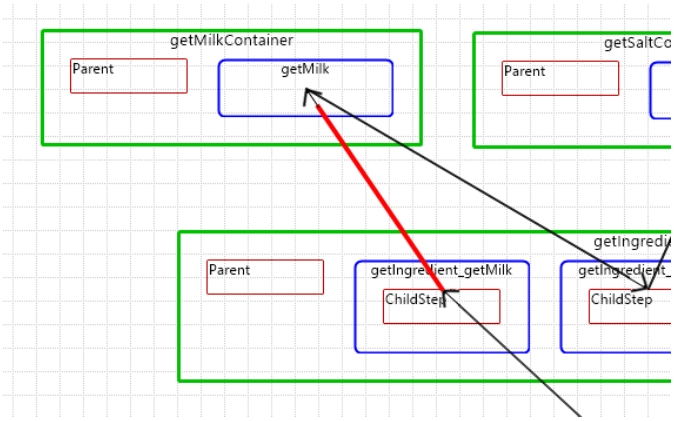

Abbildung 4.7: Method-Sequence Beispiel

- **Trigger**: *getIngredient_getMilk.ChildStep != null* - ChildStep wird nur ausgeführt, wenn er vorher gesetzt wurde
- **Transfer**: Es werden keine Daten übertragen
- **Resolve**: *getIngredient_getMilk.ChildStep* - die Instanz, auf welcher die Methode aufgerufen wird, ist jene, die vorher erstellt wurde

Es steht dem Simulationsentwickler frei, diese Werte zu ändern.

4.7 Ausführende Instanzen (Agenten)

Simulationsentitäten müssen instanziert werden. Dies gilt vor allem für Container und Agenten-Container. In der gewählten Umsetzungsmethodik werden Agenten Instanzen beim Starten der Simulation im Simulationstool gesetzt. Step-Container (jene Container mit dem Postfix *_Container*) werden automatisch erstellt. Codebeispiel 4.8 zeigt den generierten Code für *getIngredient_getMilk* aus dem *getIngredientContainer* (entnommen aus dem obigen Beispiel).

Wie man im Codebeispiel 4.8 sieht, wird in den Methoden eine neue ChildStep Instanz des *getMilkContainer* erstellt. Beim Ausführungspfeil zwischen *getIngredient_getMilk*

Algorithm 4.8: Automatisch generierter Code für Agenten

```
1  DomainProAnalyst.Instance.Report("Person.getIngredientContainer.getIngredient_getMilk");
2
3  // Default agent instances - modify if needed
4  var agentInstanceEnvControl = getIngredientContainer.Person.simpleSeq.EnvControlList[0];
5  var agentInstancePerson = getIngredientContainer.Person;
6
7  // Modify agentInstance if other Child should be called
8  ChildStep = (PersonContext.getMilkContainer)agentInstancePerson.Create("getMilkContainer");
9  ChildStep.Parent = getIngredientContainer;
```

und *getMilk* wird die Instanz anschließend bei Resolve returniert (*return getIngredient_getMilk.ChildStep;*).

4.8 Transfer von Daten in sequentiellen Abläufen

LittleJIL erlaubt es, Daten zwischen Steps zu transferieren. Codebeispiel 4.9 zeigt ein Beispiel.

Das Beispiel enthält folgende wichtige Punkte:

- *ObserveWeather* hat einen *ParameterOut* vom Typ *Weather*. Dieser wird im *Local to Parent Data Binding* dem Parent-Step zugewiesen
- *CreateStoryAboutWeather* bekommt das *Weather* als *ParameterIn*. Im *Local to Parent Data Binding* erfolgt die Zuweisung aus dem Parent-Step. Der Schritt bekommt ebenfalls eine bestehende Story als Input, modifiziert diese und weist sie wieder dem Parent zu. Dies wurde durch *ParameterInOut* realisiert bzw. durch *storyReference <> masterStoryReference* an den Parent-Step gebunden.
- *ReadStory* funktioniert analog zu den anderen Steps und bekommt die Story als Input

Abbildung 4.8 zeigt das transferierte Modell. Der Transfer der Daten passiert in den ein- bzw. ausgehenden Pfeilen. Betrachtet man den eingehenden Pfeil von *CreateStoryAboutWeather*, so befindet sich in der Transfer Eigenschaft der Code wie in Codebeispiel 4.10 gezeigt.

Dies spiegelt auch die verbale Erklärung der Zuweisung wieder. Im ausgehenden Pfeil findet man die Zuweisung wie im Codebeispiel 4.11.

4.9 Sequentielle Schleifen (Loops)

Schleifen werden durch Kardinalitäten ausgedrückt. Diese besagen, wie oft ein Child Step (oder mehrere ChildSteps) ausgeführt werden soll. Codebeispiel 4.12 zeigt ein Beispiel - Abbildung 4.9 zeigt die dazugehörige Ausgabe.

Algorithm 4.9: Beispiel für den Transfer von Daten zwischen Steps

```
1  package simpleSeq
2  hADL Model name: loop
3  {
4    Structure name: ex
5    {
6      Component name: Person
7      Artifact name: EnvControl
8      Artifact name: Story
9      Artifact name: Weather
10   }
11 }
12 LittleJIL Process name: MakeDataTransfer
13   Root Steps {
14     Step name: MasterStep of type Sequential
15     Local Data: {
16       [ agent as Resource of hADLType loop.ex.EnvControl]
17       [ masterStoryReference as Subtree of hADLType loop.ex.Story]
18       [ masterWeatherReference as Subtree of hADLType loop.ex.Weather]
19     }
20     Child Steps: {
21       Step name: ObserveWeather of type Leaf
22       Local Data: {
23         [ agent as Resource of hADLType loop.ex.EnvControl]
24         [ weatherReference as ParameterOut of hADLType loop.ex.Weather]
25       }
26       Local to Parent Data Bindings: {
27         [ weatherReference > masterWeatherReference ]
28       }
29       Step name: CreateStoryAboutWeather of type Leaf
30       Local Data: {
31         [ agent as Resource of hADLType loop.ex.EnvControl]
32         [ storyReference as ParameterInOut of hADLType loop.ex.Story]
33         [ weatherReference as ParameterIn of hADLType loop.ex.Weather]
34       }
35       Local to Parent Data Bindings: {
36         [ storyReference <> masterStoryReference ]
37         [ weatherReference < masterWeatherReference ]
38       }
39       Step name: ReadStory of type Leaf
40       Local Data: {
41         [ agent as Resource of hADLType loop.ex.Person]
42         [ personStoryReference as ParameterIn of hADLType loop.ex.Story]
43       }
44       Local to Parent Data Bindings: {
45         [ personStoryReference < masterStoryReference ]
46       }
47     }
48   }
```

Algorithm 4.10: Beispiel für den Transfer von Daten zwischen Steps

```
1  CreateStoryAboutWeather.CreateStoryAboutWeatherContainer.storyReference =
       MasterStep_CreateStoryAboutWeather.MasterStepContainer.masterStoryReference;
2  CreateStoryAboutWeather.CreateStoryAboutWeatherContainer.weatherReference =
       MasterStep_CreateStoryAboutWeather.MasterStepContainer.masterWeatherReference;
```

Algorithm 4.11: Beispiel für den Transfer von Daten zwischen Steps

```
1  MasterStep_ReadStory.MasterStepContainer.masterStoryReference = CreateStoryAboutWeather.
       CreateStoryAboutWeatherContainer.storyReference;
```

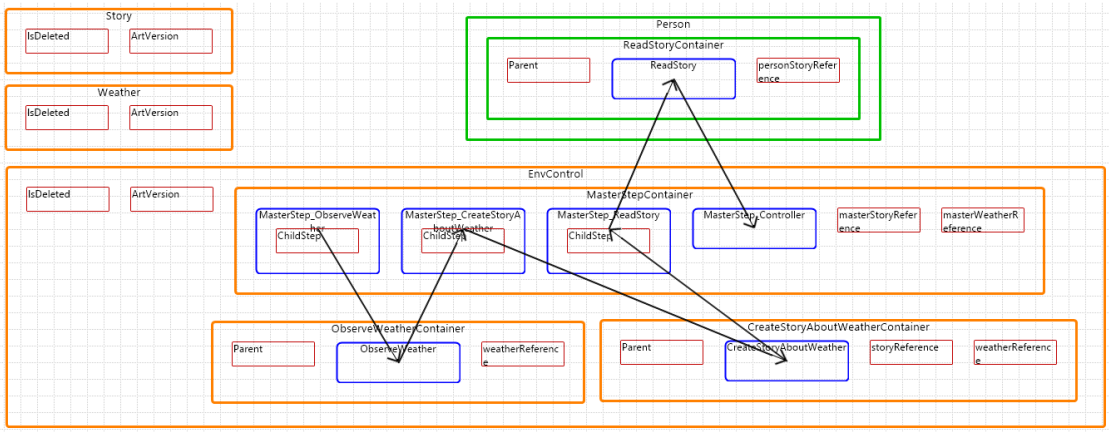

Abbildung 4.8: Beispiel von Datentransfer bei sequentiellen Schritten

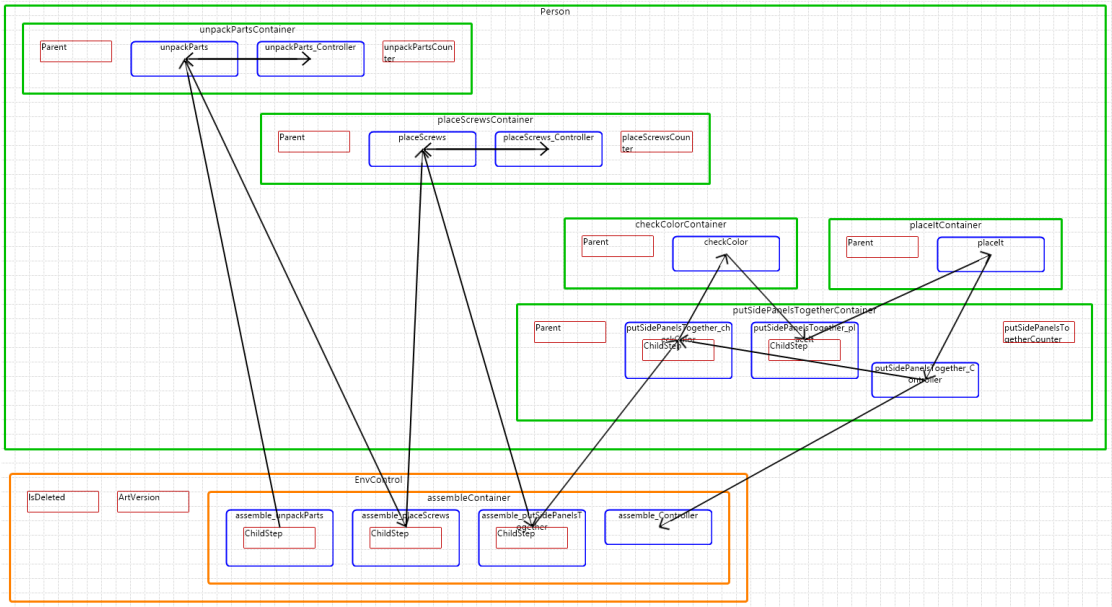

Abbildung 4.9: Beispiel einer sequentiellen Schleife

Betrachtet man e.g. *placeScrews* so erkennt man folgendes:

Algorithm 4.12: Beispiel von sequentiellen Loops über Ranges

```
1  package simpleSeq
2  hADL Model name: loop
3  {
4    Structure name: ex
5    {
6      Component name: Person
7      Artifact name: EnvControl
8    }
9  }
10 LittleJIL Process name: AssembleBilly
11 Root Steps {
12   Step name: assemble of type Sequential
13   Local Data: {
14     [ agent as Resource of hADLType loop.ex.EnvControl]
15   }
16   Child Steps: {
17     Step name: unpackParts of type Leaf
18     has Cardinality: 5..5
19     Local Data: {
20       [ agent as Resource of hADLType loop.ex.Person]
21     }
22     Step name: placeScrews of type Leaf
23     has Cardinality: 4..8
24     Local Data: {
25       [ agent as Resource of hADLType loop.ex.Person]
26     }
27     Step name: putSidePanelsTogether of type Sequential
28     has Cardinality: 3..6
29     Local Data: {
30       [ agent as Resource of hADLType loop.ex.Person]
31     }
32     Child Steps: {
33       Step name: checkColor of type Leaf
34       Step name: placeIt of type Leaf
35     }
36   }
37 }
```

- Es wurde eine nummerische Variable *placeScrewsCounter* angelegt, welche pro Iteration um 1 erhöht wird
- Die Erhöhung passiert in *_Controller*-Methode des Steps
- Die Kardinalität beträgt 4 bis 8. Das heißt, dass es mindestens 4 Mal - maximal 8 Mal ausgeführt werden soll. Es obliegt dem Entwickler die Ausführung in diesem Bereich zu steuern.
- Die Steuerung erfolgt in den Trigger der Pfeile. So ist der Trigger des Pfeils *placeScrews* und *assemble_putSidePanelsTogether* mit *placeScrews.placeScrewsContainer.placeScrewsCounter >= 8* angegeben
- Der sequentielle Ablauf bei Schritten mit Child-Steps kann aus dem Beispiel *putSidePanelsTogetherContainer* entnommen werden. Der *putSidePanelsTogether_Controller* erhöht die Counter-Variable. Wie man sieht, hat der *putSidePanelsTogether_Controller* zwei ausgehende Pfeile. Einer dieser Pfeile enthält die Bedingung für *>= 8* (returniert zum Parent Step) und der andere für *< 8* (führt eine weitere Iteration aus).

Ebenfalls ist zu erwähnen, dass auch Leaf-Knoten eine *_Controller* Methode beinhalten können, wenn diese öfters ausgeführt werden sollen (siehe e.g. *unpackParts*).

4.9.1 Sequentielle Schleifen über Listen

Neben den Ranges (e.g. 4..8) ist es auch möglich, über Listen zu iterieren. Codebeispiel 4.13 zeigt ein Beispiel.

Algorithm 4.13: Iteration über Listen

```
1  package simpleSeq
2  hADL Model name: loop
3  {
4    Structure name: ex
5    {
6      Component name: Node
7      Artifact name: EnvControl
8      Artifact name: Task
9    }
10 }
11 LittleJIL Process name: CalculationCluster
12 Root Steps {
13   Step name: MasterStep of type Sequential
14     Local Data: {
15       [ agent as Resource of hADLType loop.ex.EnvControl]
16       [ masterTaskReferenceList as Subtree of hADLType loop.ex.Task]
17     }
18     Child Steps: {
19       Step name: DistributeTasks of type Sequential
20         has Cardinality: 0..* for elements in: masterTaskReferenceList
21         Local Data: {
22           [ agent as Resource of hADLType loop.ex.EnvControl]
23           [ taskReference as ParameterOut of hADLType loop.ex.Task]
24         }
25         Local to Parent Data Bindings: {
26           [ taskReference < masterTaskReferenceList ]
27         }
28         Child Steps: {
29           Step name: CalculateTaskAndReturnResult of type Leaf
30           Local Data: {
31             [ agent as Resource of hADLType loop.ex.Node]
32             [ newTaskReference as ParameterIn of hADLType loop.ex.Task]
33           }
34           Local to Parent Data Bindings: {
35             [ newTaskReference < taskReference ]
36           }
37         }
38     }
39   }
```

Der Step *DistributeTasks* weist dem Node (Default wird die erste Node Instanz genommen) mehrere Tasks zu, welche in *CalculateTaskAndReturnResult* abgearbeitet werden. Die wichtigsten Zeilen sind im Codebeispiel 4.14 zu sehen.

Algorithm 4.14: Zuweisung der Liste

```
1  has Cardinality: 0..* for elements in: masterTaskReferenceList
2  ...
3  [ taskReference < masterTaskReferenceList ]
```

Die erste Zeile (Codebeispiel 4.14) besagt, dass über alle Elemente einer Liste iteriert werden soll. Es wurde eine Convension eingeführt, welche besagt, dass Namen, welche mit *List* Enden als Liste interpretiert werden. Diese zweite Zeile besagt, dass eine Liste einem einzelnen Task zugewiesen wird - was als Zuweisung eines einzelnen Elements der Liste interpretiert werden kann. Das Ergebnis der Transformation ist in Abbildung 4.10 zu sehen.

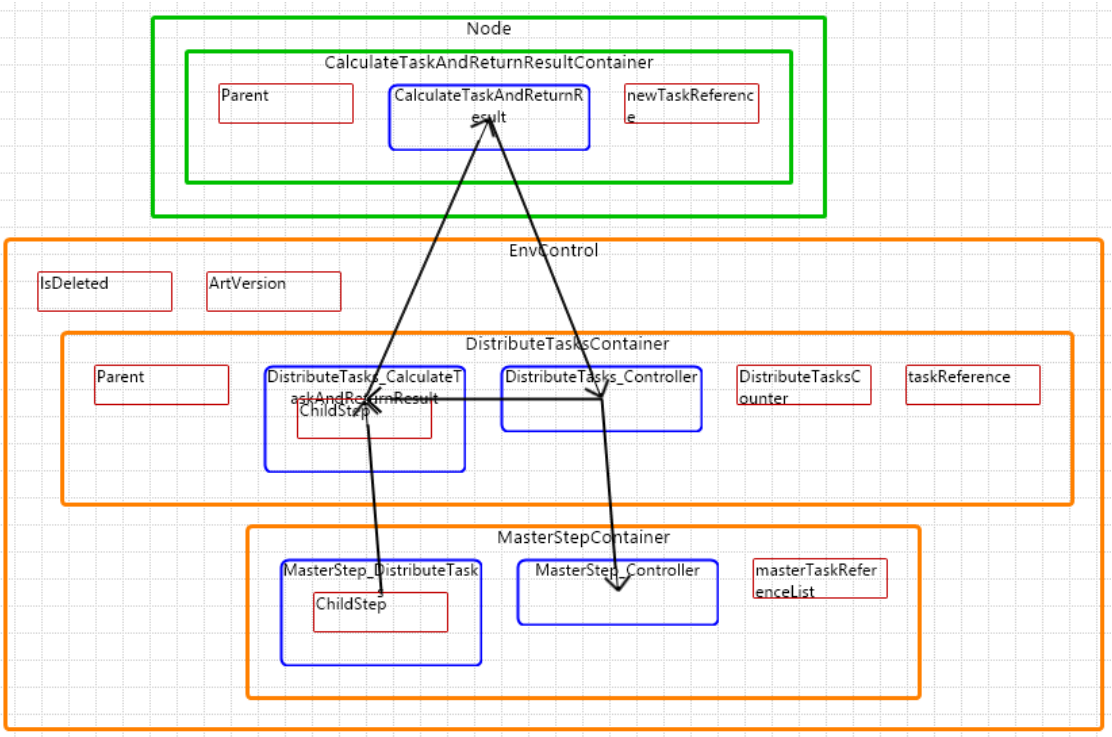

Abbildung 4.10: Beispiel einer sequentiellen Schleife über eine Liste

Der essentielle Code befindet sich in *DistributeTasks_Calling* und ist im Codebeispiel 4.15 dargestellt.

Man sieht, dass mit Hilfe des *DistributeTasksCounter* das aktuelle Item gewählt wird und anschließend der *taskReference* zugewiesen wird. Die Referenz muss nur noch in den Child-Step gegeben werden. Dies passiert durch ein weiteres wichtiges Feature von LittleJIL: dem Transfer von Daten zwischen Steps. Dazu soll der Step *CalculateTaskAndReturnResult* genauer betrachtet werden (Codebeispiel 4.16).

Codebeispiel 4.16 besagt, dass sich der Step einen eingehenden Parameter vom Typ Task erwartet.

Codebeispiel 4.17 besagt, dass diese Variable vom Parent zugewiesen werden soll. Dies wird im eingehenden Pfeil von CalculateTaskAndReturnResult in der Transfer Eigenschaft gesetzt (Codebeispiel 4.18).

Algorithm 4.15: Iteration über eine Liste

```
1  DomainProAnalyst.Instance.Report("EnvControl.DistributeTasksContainer.
       DistributeTasks_CalculateTaskAndReturnResult");
2
3  // Default agent instances - modify if needed
4  var agentInstanceEnvControl = DistributeTasksContainer.EnvControl;
5  var agentInstanceNode = DistributeTasksContainer.EnvControl.simpleSeq.NodeList[0];
6
7  if(DistributeTasksContainer.DistributeTasksCounter < DistributeTasksContainer.Parent.
       masterTaskReferenceList.Count)
8  {
9    DistributeTasksContainer.taskReference = DistributeTasksContainer.Parent.masterTaskReferenceList[
         DistributeTasksContainer.DistributeTasksCounter];
10 }
11
12 // Modify agentInstance if other Child should be called
13 ChildStep = (NodeContext.CalculateTaskAndReturnResultContainer)agentInstanceNode.Create("
       CalculateTaskAndReturnResultContainer");
14 ChildStep.Parent = DistributeTasksContainer;
```

Algorithm 4.16: Iteration über eine Liste

```
1  [ newTaskReference as ParameterIn of hADLType loop.ex.Task]
```

Algorithm 4.17: Iteration über eine Liste

```
1  Local to Parent Data Bindings: {
2    [ newTaskReference < taskReference ]
3  }
```

Neben dem ParameterIn gibt es auch noch ParameterOut und ParameterInOut. Out-Parameter werden immer über die Transfer-Eigenschaft eines Pfeiles gesetzt.

4.9.2 Sequentielle Schleifen über Agenten

Wie in den letzten Beispielen erwähnt, wird immer der erste Agent für die Ausführung eines Child-Steps herangezogen. Allerdings ist es oft erforderlich, mehrere Agenten zu benutzen. Codebeispiel 4.19 zeigt ein Beispiel, in dem mehrere Personen (Agenten) befragt werden.

Im Beispiel werden Personen über ihre Laune befragt. Die Personen befinden sich in der *masterPersonReferenceList*. Es wird durch die Liste iteriert und der Agent jeder Person gesetzt(Codebeispiel 4.20 in *askedAboutMood*).

Abbildung 4.11 zeigt das Ergebnis der Transformation.

Algorithm 4.18: Iteration über eine Liste

```
1  CalculateTaskAndReturnResult.CalculateTaskAndReturnResultContainer.newTaskReference =
       DistributeTasks_CalculateTaskAndReturnResult.DistributeTasksContainer.taskReference;
```

Algorithm 4.19: sequentielle Iteration über eine Liste von Agenten

```
1  package simpleSeq
2  hADL Model name: loop
3  {
4    Structure name: ex
5    {
6      Component name: Person
7      Artifact name: EnvControl
8    }
9  }
10 LittleJIL Process name: AskAboutMooPerson
11 Root Steps {
12   Step name: askAboutMoodRoot of type Sequential
13   Local Data: {
14     [ agent as Resource of hADLType loop.ex.EnvControl]
15     [ masterPersonReferenceList as Subtree of hADLType loop.ex.Person]
16   }
17   Child Steps: {
18     Step name: askAboutMooLoop of type Sequential
19     has Cardinality: 0..* for elements in: masterPersonReferenceList
20     Local Data: {
21       [ agent as Resource of hADLType loop.ex.EnvControl]
22       [ personReference as ParameterOut of hADLType loop.ex.Person]
23     }
24     Local to Parent Data Bindings: {
25       [ personReference < masterPersonReferenceList ]
26     }
27     Child Steps: {
28       Step name: askedAboutMood of type Leaf
29         Local Data: {
30           [ agent as Resource of hADLType loop.ex.Person]
31         }
32         Local to Parent Data Bindings: {
33           [ agent < personReference ]
34         }
35     }
36   }
37 }
```

Algorithm 4.20: sequentielle Iteration über eine Liste von Agenten

```
1  [ agent < personReference ]
```

Der generierte Code von *askAboutMooLoop_askedAboutMood* ist in Codebeispiel 4.21 dargestellt. Wie man sieht, wird die *agentInstancePerson* durch die aktuelle Instanz der Liste überschrieben. Anschließend wird auf dieser Instanz ein *askedAboutMoodContainer* erzeugt. Der Aufruf erfolgt mit Hilfe der Pfeile.

4.10 Parallele Schleifen (Loops)

Auch können Child-Steps parallel ausgeführt werden. Codebeispiel 4.22 zeigt ein Beispiel. In Abbildung 4.12 sieht man das Transformationsergebnis in DomainPro.

Wie dem Beispiel zu entnehmen ist, werden einer Person (da 1 Agent Instanz) maximal 5 Aufgaben zugewiesen.

Die Steuerung erfolgt diesmal nicht direkt mit den Pfeilen, sondern mit einem Auto

Algorithm 4.21: Generierter Code von einer sequentiellen Iteration über eine Liste von Agenten

```
1  DomainProAnalyst.Instance.Report("EnvControl.askAboutMooLoopContainer.
       askAboutMooLoop_askedAboutMood");
2
3  // Default agent instances - modify if needed
4  var agentInstanceEnvControl = askAboutMooLoopContainer.EnvControl;
5  var agentInstancePerson = askAboutMooLoopContainer.EnvControl.simpleSeq.PersonList[0];
6
7  if(askAboutMooLoopContainer.askAboutMooLoopCounter < askAboutMooLoopContainer.Parent.
       masterPersonReferenceList.Count)
8  {
9    askAboutMooLoopContainer.personReference = askAboutMooLoopContainer.Parent.
         masterPersonReferenceList[askAboutMooLoopContainer.askAboutMooLoopCounter];
10 }
11
12 // Agent instances set per Transfer
13 agentInstancePerson = askAboutMooLoopContainer.personReference;
14
15 // Modify agentInstance if other Child should be called
16 ChildStep = (PersonContext.askedAboutMoodContainer)agentInstancePerson.Create("
       askedAboutMoodContainer");
17 ChildStep.Parent = askAboutMooLoopContainer;
```

Algorithm 4.22: Beispiel einer parellelen Schleife über Ranges

```
1  package simpleSeq
2  hADL Model name: loop
3  {
4    Structure name: ex
5    {
6      Component name: Person
7      Artifact name: EnvControl
8    }
9  }
10 LittleJIL Process name: AssignWorkToPerson
11 Root Steps {
12   Step name: assignWorkRoot of type Sequential
13   Local Data: {
14     [ agent as Resource of hADLType loop.ex.EnvControl]
15   }
16   Child Steps: {
17     Step name: assignWorkLoop of type Parallel
18     has Cardinality: 0..5
19     Local Data: {
20       [ agent as Resource of hADLType loop.ex.EnvControl]
21     }
22     Child Steps: {
23       Step name: assignWork of type Leaf
24       Local Data: {
25         [ agent as Resource of hADLType loop.ex.Person]
26       }
27     }
28   }
29 }
```

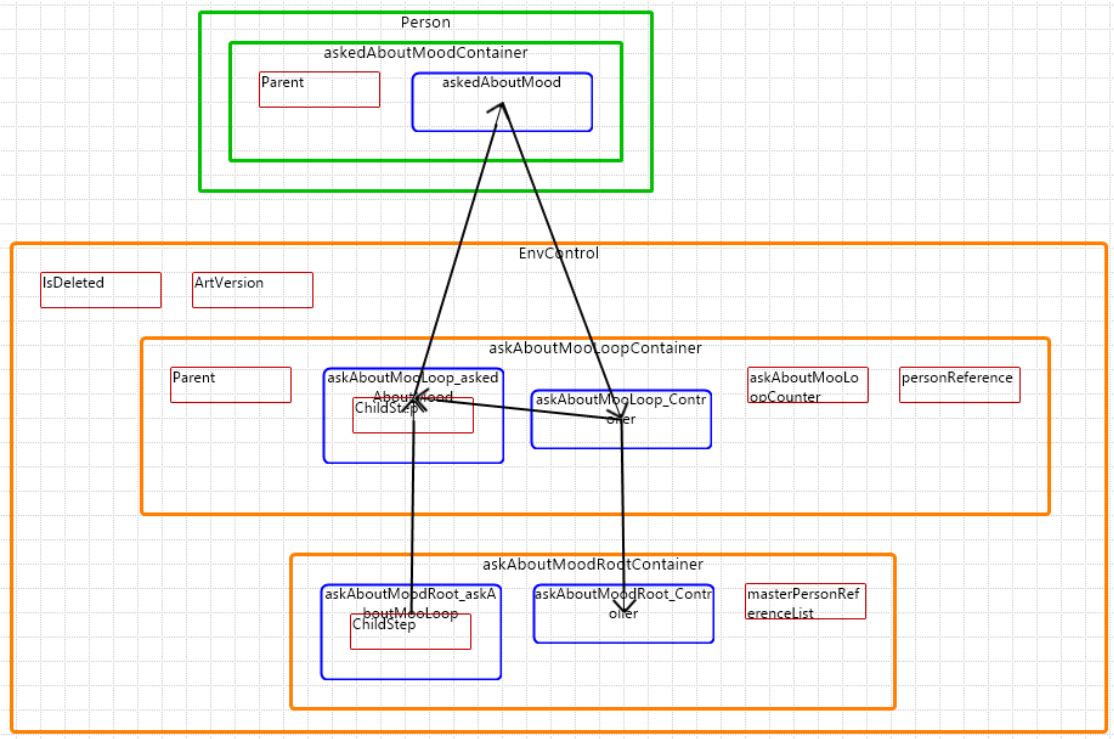

Abbildung 4.11: Beispiel einer sequentiellen Schleife über Agenten

generierten Code, welcher sich in der aufrufenden Methode *assignWorkLoop_Calling* befindet (Codebeispiel 4.23).

Wie man im Codebeispiel 4.23 sieht, wird die erste Personen Instanz (Agent) für die Erzeugung von *assignWorkContainer* genommen. Dies kann jedoch vom Entwickler angepasst werden. Ebenfalls ist es erforderlich, sich die Instanzen zu merken, welche aufgerufen werden (*CalledChildStepsList*). Dies ist erforderlich, um feststellen zu können, wann alle Aufrufe getätigt wurden. Um sicher zu stellen, dass alle Instanzen in der *CalledChildStepsList* sind, wird erst nach Beendigung der ersten Schleife der Aufruf der Methoden gemacht.

Jeder Prozess der returniert, setzt die *ReturningCallerReference* der *assignWorkLoop_Controller* Komponente. Dieser enthält den autogenerierten Code wie in Codebeispiel 4.24 gezeigt.

Der Trigger des darüber liegenden Pfeils ist im Codebeispiel 4.25 gezeigt. Er returniert, wenn alle asynchronen Methoden erfolgreich returniert haben.

Der Output eines Simulationsdurchlaufs ist im Codebeispiel 4.26 gezeigt.

Algorithm 4.23: Parallele Schleife über Ranges

```
1  DomainProAnalyst.Instance.Report("EnvControl.assignWorkLoopContainer.assignWorkLoop_Calling");
2
3  // Default agent instances - modify if needed
4  var agentInstanceEnvControl = assignWorkLoopContainer.EnvControl;
5  var agentInstancePerson = assignWorkLoopContainer.EnvControl.simpleSeq.PersonList[0];
6
7  var asynchronActionsToCallList = new List<Action>();
8
9  for(int calls = 0; calls < 5; calls++)
10 {
11   var hostAssignWork = (PersonContext.assignWorkContainer)agentInstancePerson.Create("
         assignWorkContainer");
12   hostAssignWork.Parent = assignWorkLoopContainer;
13   assignWorkLoopContainer.CalledChildStepsList.Add(hostAssignWork);
14   asynchronActionsToCallList.Add(() => hostAssignWork.assignWork(false));
15 }
16
17 // Call methods asynchron when all instances are stored in CalledChildStepsList
18 foreach(var actionToCall in asynchronActionsToCallList)
19 {
20   actionToCall();
21 }
```

Algorithm 4.24: Parallele Schleife über Ranges

```
1  DomainProAnalyst.Instance.Report("EnvControl.assignWorkLoopContainer.assignWorkLoop_Controller");
2
3  // Default agent instances - modify if needed
4  var agentInstanceEnvControl = assignWorkLoopContainer.EnvControl;
5  var agentInstancePerson = assignWorkLoopContainer.EnvControl.simpleSeq.PersonList[0];
6
7  if(ReturningCallerReference != null)
8  {
9    assignWorkLoopContainer.CalledChildStepsList.Remove(ReturningCallerReference);
10 }
```

Algorithm 4.25: Parallele Schleife über Ranges

```
1  assignWorkLoop_Controller.assignWorkLoopContainer.CalledChildStepsList.Count == 0;
```

Algorithm 4.26: Parallele Schleife über Ranges

```
1  Starting simulation NewSimulation...
2  EnvControl.assignWorkRootContainer.assignWorkRoot_assignWorkLoop
3  EnvControl.assignWorkLoopContainer.assignWorkLoop_Calling
4  Person.assignWorkContainer.assignWork
5  Person.assignWorkContainer.assignWork
6  Person.assignWorkContainer.assignWork
7  Person.assignWorkContainer.assignWork
8  Person.assignWorkContainer.assignWork
9  EnvControl.assignWorkLoopContainer.assignWorkLoop_Controller
10 EnvControl.assignWorkLoopContainer.assignWorkLoop_Controller
11 EnvControl.assignWorkLoopContainer.assignWorkLoop_Controller
12 EnvControl.assignWorkLoopContainer.assignWorkLoop_Controller
13 EnvControl.assignWorkLoopContainer.assignWorkLoop_Controller
14 EnvControl.assignWorkRootContainer.assignWorkRoot_Controller
15 Simulation "NewSimulation" completed.
```

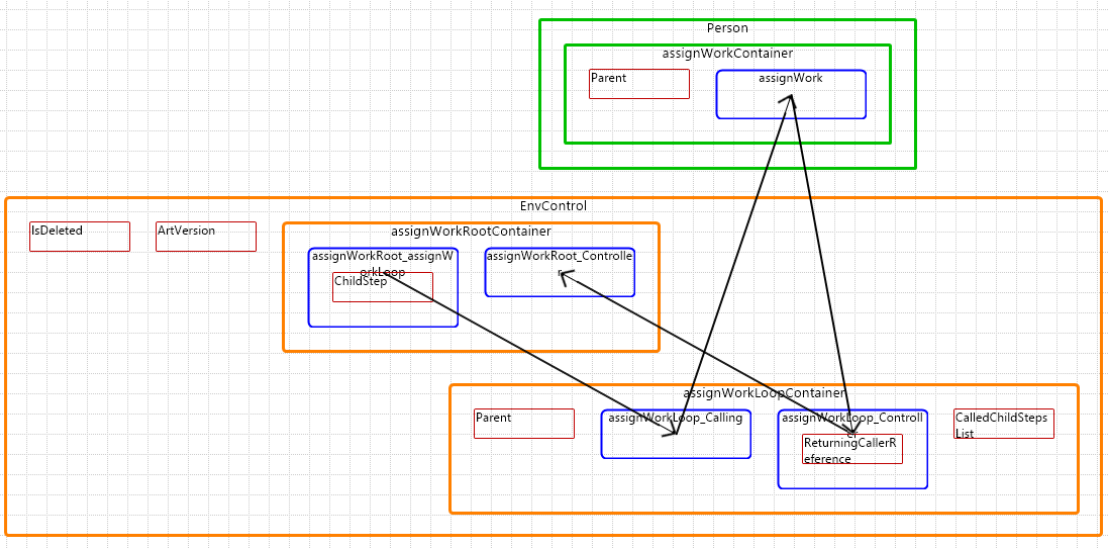

Abbildung 4.12: Beispiel einer parallelen Schleife über eine Range

4.10.1 Parallele Schleifen über Listen

Neben den Ranges (e.g. 4..8) ist es auch möglich, über Listen zu iterieren. Codebeispiel 4.27 zeigt ein Beispiel. Es ähnelt dem sequentiellen Beispiel, allerdings werden die ChildSteps parallel ausgeführt.

Das Ergebnis der Transformation ist in Abbildung 4.13 zu sehen.

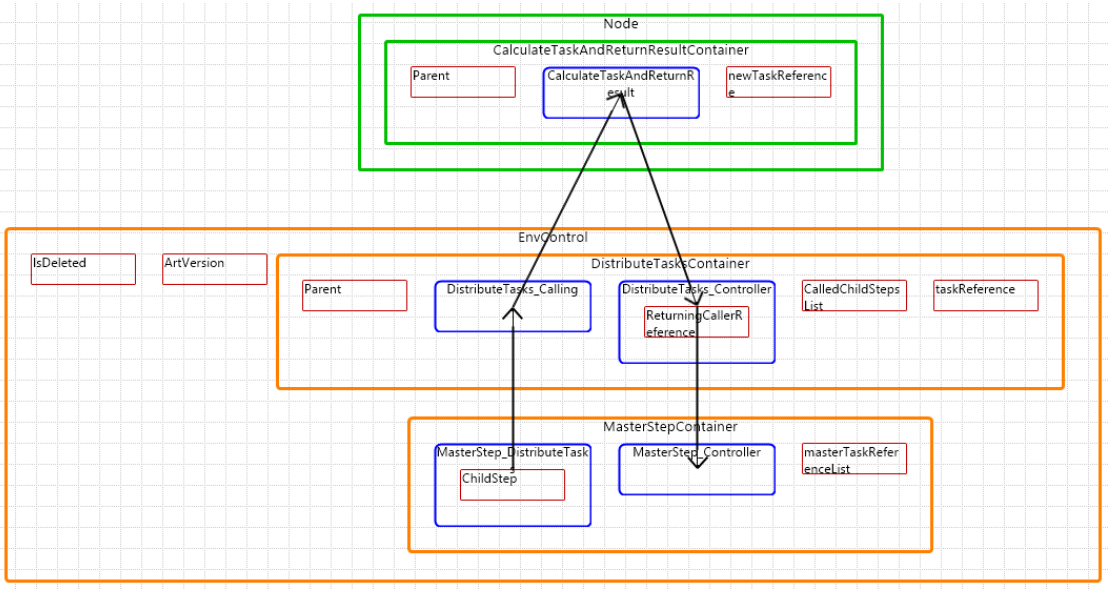

Abbildung 4.13: Beispiel einer parallelen Schleife über Ressourcen

Algorithm 4.27: parallele Iteration über Liste

```
1  package simpleSeq
2  hADL Model name: loop
3  {
4    Structure name: ex
5    {
6      Component name: Node
7      Artifact name: EnvControl
8      Artifact name: Task
9    }
10 }
11 LittleJIL Process name: CalculationCluster
12 Root Steps {
13   Step name: MasterStep of type Sequential
14   Local Data: {
15     [ agent as Resource of hADLType loop.ex.EnvControl]
16     [ masterTaskReferenceList as Subtree of hADLType loop.ex.Task]
17   }
18   Child Steps: {
19     Step name: DistributeTasks of type Parallel
20     has Cardinality: 0..* for elements in: masterTaskReferenceList
21     Local Data: {
22       [ agent as Resource of hADLType loop.ex.EnvControl]
23       [ taskReference as ParameterOut of hADLType loop.ex.Task]
24     }
25     Local to Parent Data Bindings: {
26       [ taskReference < masterTaskReferenceList ]
27     }
28     Child Steps: {
29       Step name: CalculateTaskAndReturnResult of type Leaf
30       Local Data: {
31         [ agent as Resource of hADLType loop.ex.Node]
32         [ newTaskReference as ParameterIn of hADLType loop.ex.Task]
33       }
34       Local to Parent Data Bindings: {
35         [ newTaskReference < taskReference ]
36       }
37     }
38
39   }
40 }
```

Im parallelen Fall muss der Aufruf der Child-Steps per Code passieren und kann nicht - wie im sequentiellen Beispiel - per Pfeile delegiert werden. Der Code befindet sich in *DistributeTasks_Calling* und ist im Codebeispiel 4.28 dargestellt. Die Transformation der beiden Zeilen aus der DSL spiegeln sich auch im generierten Code wieder (Codebeispiel 4.29). Der Transfer der Daten kann nicht wie im sequentiellen Fall per Pfeil-Transfer passieren, sondern muss im Code gesetzt werden (Codebeispiel 4.30).

Algorithm 4.28: parallele Iteration über Liste

```
1  DomainProAnalyst.Instance.Report("EnvControl.DistributeTasksContainer.DistributeTasks_Calling");
2
3  // Default agent instances - modify if needed
4  var agentInstanceEnvControl = DistributeTasksContainer.EnvControl;
5  var agentInstanceNode = DistributeTasksContainer.EnvControl.simpleSeq.NodeList[0];
6
7  var asynchronActionsToCallList = new List<Action>();
8
9  foreach(var subStepInstance in DistributeTasksContainer.Parent.masterTaskReferenceList)
10 {
11   DistributeTasksContainer.taskReference = subStepInstance;
12
13   var hostCalculateTaskAndReturnResult = (NodeContext.CalculateTaskAndReturnResultContainer)
         agentInstanceNode.Create("CalculateTaskAndReturnResultContainer");
14   hostCalculateTaskAndReturnResult.Parent = DistributeTasksContainer;
15   DistributeTasksContainer.CalledChildStepsList.Add(hostCalculateTaskAndReturnResult);
16   hostCalculateTaskAndReturnResult.newTaskReference = DistributeTasksContainer.taskReference;
17   asynchronActionsToCallList.Add(() => hostCalculateTaskAndReturnResult.
         CalculateTaskAndReturnResult(false));
18 }
19
20 // Call methods asynchron when all instances are stored in CalledChildStepsList
21 foreach(var actionToCall in asynchronActionsToCallList)
22 {
23   actionToCall();
24 }
```

Algorithm 4.29: parallele Iteration über Liste

```
1  foreach(var subStepInstance in DistributeTasksContainer.Parent.masterTaskReferenceList)
2  {
3    DistributeTasksContainer.taskReference = subStepInstance;
```

Algorithm 4.30: parallele Iteration über Liste

```
1  hostCalculateTaskAndReturnResult.newTaskReference = DistributeTasksContainer.taskReference;
```

KAPITEL 5

Implementierung

Das folgende Kapitel beschäftigt sich mit den Implementierungsaspekten des Transformationstools. Das Tool ermöglicht es, die in der DSL beschriebenen kollaborativen Modelle auszuführen (simulieren). Als Simulationstool wird DomainPro Analyst verwendet.

5.1 Implementierungsdetails Transformations-Tool

Die Transformation der DSL nach XML erfolgt mit dem Eclipse Modeling Framework - ist aber nicht Teil der vorliegenden Arbeit. Als Input für das Transformationstool wird das von EMF generierte XML verwendet. Da die XML Schema Definition (XSD) vorhanden war, konnte mit Hilfe des Tools `http://www.xsd2code.com` ein Code Skeleton generiert werden, welche es erlaubt, das XML in Objekte umzuwandeln.

Ebenfalls erlaubt das Tool XSD2Code, dass eine Basisklasse für alle Entitäten angegeben werden konnte. Diese Basisklasse ist im Codebeispiel 5.1 zu sehen.

Algorithm 5.1: Basisklasse der XML Entitäten

```
1  namespace Contracts.Xsd
2  {
3    public class EntityBase<T>
4    {
5      [XmlIgnore]
6      public AbsoluteXPathModel XPath { get; set; }
7    }
8  }
```

Zur jeder Identität wird der XPath gespeichert. Codebeispiel 5.2 zeigt einen Auszug der Step Klasse. Wie man sieht, besitzt die Klasse keine Eigenschaft, welche den Parent-Step wiederspiegelt. Durch die rekursive Top-Down Suche nach Referenzen könnte dieses Problem auch gelöst werden. Allerdings hat die XPath Variante zwei Vorteile:

- Einfaches debuggen, da man sofort sieht, um welches Element es sich handelt
- Einige Referenzen (e.g. Bindings) werden auch als Pfad angegeben und sind daher über XPath sehr leicht zu finden - mehr dazu im Laufe des Kapitels.

Algorithm 5.2: Ein Auszug aus der Step Klasse

```
1  [System.CodeDom.Compiler.GeneratedCodeAttribute("System.Xml", "4.0.30319.34234")]
2  [Serializable()]
3  [DebuggerStepThrough()]
4  [DesignerCategory("code")]
5  [XmlType(Namespace = "http://www.tuwien.at/dsg/Dsl")]
6  [XmlRoot(Namespace = "http://www.tuwien.at/dsg/Dsl", IsNullable = false)]
7  public partial class Step : EntityBase<Step>
8  {
9    ...
10
11   [XmlElement(Form = System.Xml.Schema.XmlSchemaForm.Unqualified, ElementName = "cardinality")]
12   public StepCardinality Cardinality { ... }
13
14   [XmlElement("interfaces", Form = System.Xml.Schema.XmlSchemaForm.Unqualified, ElementName = "
         interfaces")]
15   public List<StepInterface> Interfaces { ... }
16
17   [XmlElement("bindings", Form = System.Xml.Schema.XmlSchemaForm.Unqualified, ElementName = "
         bindings")]
18   public List<Binding> Bindings { ... }
19
20   [XmlElement("substeps", Form = System.Xml.Schema.XmlSchemaForm.Unqualified, ElementName = "
         substeps")]
21   public List<Step> Substeps { ... }
22
23   [XmlElement("exceptionHandlers", Form = System.Xml.Schema.XmlSchemaForm.Unqualified, ElementName
         = "exceptionHandlers")]
24   public List<ExceptionHandler> ExceptionHandlers { ... }
25
26   [XmlElement("stepExceptions", Form = System.Xml.Schema.XmlSchemaForm.Unqualified, ElementName = "
         stepExceptions")]
27   public List<StepException> StepExceptions { ... }
28
29   [XmlAttribute(AttributeName = "name")]
30   public string Name { ... }
31
32   [XmlAttribute(AttributeName = "kind")]
33   public StepType Kind { ... }
34
35   [XmlIgnore()]
36   public bool KindSpecified { ... }
37 }
```

Das Setzen des XPath wurde direkt nach dem Deserialisieren gemacht. Die generierten Instanzen wurden rekursiv durchwandert und der Pfad wurde generiert. Codebeispiel 5.3 zeigt den rekursiven Teil für das Generieren des XPath für Step Elemente.

5.2 Generieren der Container

Wie bereits im Kapitel 4 diskutiert, wird das Erstellen der Container in zwei Phasen gemacht:

Algorithm 5.3: Rekursion für das Berechnen des XPath

```
1  private static void RecoverXPathForLittleJilElements(Model model)
2  {
3    model.Lj.XPath = new AbsoluteXPathModel(model.XPath + "/lj[1]");
4
5    uint stepCounter = 1;
6
7    foreach (var step in model.Lj.Steps)
8    {
9      step.XPath = new AbsoluteXPathModel(model.Lj.XPath + string.Format("/steps[{0}]", stepCounter
         ++));
10     GenerateXPathForSubStepsRecursive(step);
11   }
12 }
13
14 private static void GenerateXPathForSubStepsRecursive(Step step)
15 {
16   uint subStepCounter = 1;
17
18   foreach (var substep in step.Substeps)
19   {
20     substep.XPath = new AbsoluteXPathModel(step.XPath + string.Format("/substeps[{0}]",
         subStepCounter++));
21     GenerateXPathForSubStepsRecursive(substep);
22   }
23 }
```

- Erstellen der Container für ArtifactType und MessageType
- Erstellen der Container für ComponentType und ConnectorType

Da das XML wie bereits erwähnt in Objekte umgewandelt wurde, kann .NET Language Integrated Query (LINQ) verwendet werden. Im Codebeispiel 5.4 werden alle Komponenten für ArtifactType und MessageType abgefragt.

Algorithm 5.4: Abfrage aller ArtifactType und MessageType Elemente

```
1  var artifactOrMessageTypes = InputModel.Hadl.Structures.SelectMany(i => i.Elements).OfType<
       Contracts.Xsd.ArchElement>().Where(e => e.Type is ArtifactType || e.Type is MessageType);
```

Die Klassenstruktur für das Ausgabemodell musste manuell erstellt werden. Die Hierarchie wurde im Kapitel 4 beschrieben. Die Erstellung für ArtifactType und MessageType ist im Codebeispiel 5.5 zu sehen.

Um Probleme bei Unit Tests zu vermeiden, wurde auch die Globally Unique Identifier (GUID) abstrahiert und kann durch ein Stub[1] ersetzt werden.

5.2.1 Weitere Schritte für ArtifactType und MessageType

Für den ArtifactType werden noch die Variablen IsDeleted und ArtVersion hinzugefügt. Wie im Kapitel 3 erwähnt, werden auch noch Variablen für Kardinalitäten angelegt, wenn

[1] http://xunitpatterns.com/Test%20Stub.html

Algorithm 5.5: Erstellen eines DataObjType

```
var simulationDataObjectType = new DataObjType
{
  Name = artifactOrMessageType.Name,
  Id = GuidGenerator.NewGuid(),
  Structure = new DataObjTypeStructure
  {
    Id = GuidGenerator.NewGuid(),
    Types = new List<DomainProAbstractSemanticType>()
  },
  Text = new DomainProText
  {
    Id = GuidGenerator.NewGuid(),
    Instructions = startupInstructions
  }
};
```

deren Typ ermittelt werden kann. Der Typ wird über einen Link in der DSL spezifiziert. Ein Beispiel ist im Codebeispiel 5.6.

Algorithm 5.6: Example of a link

```
Link [doSomething: cad.world.Dispatcher.someTriggerAction => cad.world.EmergencyCall.
    anExampleActionForAnEmergencyCall]
```

Die Zwischenrepräsentation des Beispiels ist im Codebeispiel 5.7 dargestellt.

Algorithm 5.7: Zwischenrepräsentation des Links

```
<elements xsi:type="dsl:CollabLink" name="doSomething" collabActionEndpoint="dispatchXML.dsl#//
    @hadl/@structures.0/@elements.0/@actions.0" objActionEndpoint="dispatchXML.dsl#//@hadl/
    @structures.0/@elements.2/@actions.0"/>
```

Das verwendete Format ist XPath ähnlich. Der Unterschied besteht darin, dass XPath u.a. beim Zähler 1 beginnt - die vorliegende Variante bei 0. Die Extraktion der Werte funktioniert mit RegEx wie im Codebeispiel 5.8 gezeigt.

Algorithm 5.8: Regular Expressions zum Auflösen der Ecore-Path

```
private static readonly Regex EcorePathRegex = new Regex(@"^(?:((\w+).dsl#\/\/))(?:@(?<value>(\w
    +(\.\d+)?))\/?)+$", RegexOptions.Compiled);
private static readonly Regex EcoreNumberRegex = new Regex(@"^(?<elementname>\w+)(\.(?<level>\d+))*
    $", RegexOptions.Compiled);
```

Anschließend muss der Wert nur mehr umgewandelt werden. Im Codebeispiel 5.9 befindet sich der dazugehörige XPath des Beispiels.

Da nun der XPath der verlinkten Elemente bekannt ist, kann im Eingangs-XML ganz einfach eine XPath-Abfrage gestartet werden, um den Typ herauszufinden. Alternativ könnte man auch die geparsten Instanzen rekursiv durchlaufen und nach einem XPath Treffer suchen.

Algorithm 5.9: XPath des Beispiels

```
/dsl:Model/hadl[1]/structures[1]/elements[1]/actions[1]
/dsl:Model/hadl[1]/structures[1]/elements[3]/actions[1]
```

5.2.2 Generieren der Container für Steps

Für jeden Step muss ein Container innerhalb eines Agenten Container erstellt werden. Codebeispiel 5.10 zeigt, wie man dank LINQ einfach an alle Steps (und deren Child-Steps) gelangt.

Algorithm 5.10: Abfrage aller Child-Steps

```
var steps = InputModel.Lj.Steps.SelectManyRecursive(s => s.Substeps);
```

Bei SelectManyRecursive handelt es sich um eine selbst geschriebene Extention Methode wie im Codebeispiel 5.11 zu sehen ist.

Algorithm 5.11: Implementierung von SelectManyRecursive

```
public static class IEnumerableExtensions
{
  public static IEnumerable<T> SelectManyRecursive<T>(this IEnumerable<T> source, Func<T,
      IEnumerable<T>> selector)
  {
    if (source == null)
    {
      throw new ArgumentNullException("source");
    }

    if (selector == null)
    {
      throw new ArgumentNullException("selector");
    }

    var sourceArray = source.ToArray();

    return sourceArray.Any() ? sourceArray.Concat(sourceArray.SelectMany(i => selector(i) ??
        Enumerable.Empty<T>()).SelectManyRecursive(selector)) : sourceArray;
  }
}
```

Als nächstes wird überprüft, ob der Agent explizit für den Step gesetzt wurde. Wenn nicht, wird er über XPath ermittelt (Codebeispiel 5.12). Dazu wird das erste Parent Element gesucht, bei dem der Agent gesetzt ist.

Algorithm 5.12: Suche nach dem Agent

```
var parentElementWithAgent = stepElement.XPathSelectElement("ancestor::*[name()='substeps' or name
    ()='steps'][interfaces[@name='agent' and @xsi:type='dsl:HadlTypedStepInterface']][1]",
    NamespaceManager);
```

Alternativ wäre auch eine rekursive Suche möglich gewesen - allerdings bietet XPath diese von Haus aus an. Wurde der Container im Agent erstellt (mit dem Postfix *_Container* im Name), können Methoden für die einzelnen Schritte generiert werden.

5.3 Methoden für Schritte generieren

Die Erzeugung für die Methoden der einzelnen Schritte erfolgt pro Step. Codebeispiel 5.13 zeigt die Unterscheidung.

Algorithm 5.13: Unterscheidung zur Erzeugung von Step-Methoden

```
1  internal void GenerateMethodsForLeafAndRootSteps(Step step, SystemDecompType stepContainerComponent
       )
2  {
3    if (step.Kind == StepType.Leaf)
4    {
5      _leafStepActionMethodGenerator.GenerateMethodsForLeafStep(step, stepContainerComponent);
6    }
7
8    if (step.Kind == StepType.Sequential)
9    {
10     _sequentialStepActionMethodGenerator.GenerateMethodsForSequentialStep(step,
           stepContainerComponent);
11   }
12
13   if (step.Kind == StepType.Parallel)
14   {
15     _parallelStepActionMethodGenerator.GenerateMethodsForParallelStep(step, stepContainerComponent)
           ;
16   }
17 }
```

5.3.1 Parent-Knoten ermitteln anhand des Beispiels von Leaf-Methoden

Im Kapitel 4 wurde bereits erwähnt, dass Leaf-Knoten auch *_Controller* Methoden haben können. Ob diese Methode erstellt wird, hängt von der Kardinalität des Parent-Step ab - die Kardinalität muß *>1* sein. Dank LINQ kann der Parent Knoten leicht identifiziert werden (Codebeispiel 5.14).

Algorithm 5.14: Bestimmung des Parent Knote

```
1  public static class ModelExtentions
2  {
3    public static Step GetParentStep(this Model inputModel, Step step)
4    {
5     return inputModel.Lj.Steps.SelectManyRecursive(s => s.Substeps).SingleOrDefault(s => s.Substeps.
          Any(x => x.XPath == step.XPath));
6    }
7  }
```

5.3.2 Generieren von sequentiellen Methoden

Das Generieren der sequentiellen Methoden erfordert schon mehr Aufwand. Wie bereits im Kapitel 4 analysiert wurde, benötigt der Parent-Step für jeden Child-Step eine Methode, um den Aufruf bzw. die Retournierung steuern zu können. Ein weiterer essentieller Bestandteil ist das Generieren des Codes in den Methoden.

Ist die Kardinalität gesetzt, muss zusätzlicher Code generiert werden. Die aufrufende Methode ist für die Iteration verantwortlich. Ein Beispiel des generierten Codes ist im Codebeispiel 5.15 zu sehen.

Algorithm 5.15: Generierter Code zum Iterieren über Listen

```
1  if(askOthersContainer.askOthersCounter < askOthersContainer.Parent.dispatcherReferenceList.Count)
2  {
3    askOthersContainer.dispatcherReference = askOthersContainer.Parent.dispatcherReferenceList[
         askOthersContainer.askOthersCounter];
4  }
```

Der Code wird allerdings nur generiert, wenn

- die Kardinalität gesetzt ist
- der UpperBound * entspricht
- und ein Parameter-Controller vorhanden ist

Die dazugehörige DSL ist im Codebeispiel 5.16 abgebildet.

Algorithm 5.16: DSL für das Iterieren über eine Liste

```
1  has Cardinality: 0..* for elements in: dispatcherReferenceList
```

Der Parameter-Controller entspricht in diesem Beispiel der *dispatcherReferenceList*. Als nächstes werden Default-Agent-Instanzen im Code generiert, damit der Entwickler die Möglichkeit hat, diese leichter anzupassen bzw. der Code auch ohne Modifikation lauffähig ist. Der generierte Code eines Beispiels ist im Codebeispiel 5.17 zu sehen.

Algorithm 5.17: Default Agent Instanzen

```
1  // Default agent instances - modify if needed
2  var agentInstanceEnvironment = askOthersContainer.Dispatcher.cad.EnvironmentList[0];
3  var agentInstanceCommunicationGateway = askOthersContainer.Dispatcher.cad.CommunicationGatewayList
       [0];
4  var agentInstanceDispatcher = askOthersContainer.Dispatcher;
5  var agentInstanceAnnunciator = askOthersContainer.Dispatcher.cad.AnnunciatorList[0];
```

Zu beachten ist, dass das Codebeispiel 5.17 von einem Step ist, welcher als Agent den Dispatcher hat. Dort wird die Agent-Instanz auf die aktuelle Instanz gesetzt (siehe

agentInstanceDispatcher im Beispiel). Die Ermittlung aller Agenten ist relativ einfach, da der Objektbaum rekursiv durchsucht werden kann (Codebeispiel 5.18).

Algorithm 5.18: Bestimmen aller Agenten

```
1  var agentTypes = InputModel.Lj.Steps
2    .SelectManyRecursive(s => s.Substeps)
3    .SelectMany(i => i.Interfaces.OfType<HadlTypedStepInterface>())
4    .Where(i => i.Name == SimulationConstants.Agent)
5    .Select(i => _typeResolver.GetTypeNameFromEcorePath(i.Type))
6    .Distinct();
```

Ein weiterer Teil des generierten Codes sind Zuweisungen des Agenten. Dazu wird überprüft, ob der Child-Step eine Agenten-Instanz erwartet (Codebeispiel 5.19).

Algorithm 5.19: Suchen eines Agent Bindings

```
1  var agentBinding = substep.Bindings.Where(s => s.BindingType is InOutBinding || s.BindingType is
       InBinding).FirstOrDefault(b => _typeResolver.GetTypeNameFromEcorePath(b.Left) ==
       SimulationConstants.Agent);
```

Ist dies der Fall, wird die Default-Agent-Instanz mit der gesetzten Instanz überschrieben. Weitere Schritte sind:

- Füge eine *__Controller* Methode ein
- Wenn die Kardinalität gesetzt ist und mehr als 1 Iteration erfordert
  - Erzeuge eine Counter Variable
  - Erhöhe den Counter um 1 im *__Controller*

5.3.3 Generieren von parallelen Methoden

Das Generieren der Methoden von parallel Steps unterscheidet sich stark von sequentiellen Steps. Es ist hier nicht erforderlich, für jeden Step eine Methode zu erzeugen. Es werden stattdessen nur zwei Methoden erzeugt:

- eine Methode mit dem Postfix *__Calling* welche die Child-Steps aufruft
- eine Methode mit dem Postfix *__Controller* welche kontrolliert, ob alle Child-Steps returniert haben

Des Weiteren werden zwei Variablen angelegt:

- Eine Variable mit dem Namen *ReturningCallerReference*. Diese wird vom ChildStep gesetzt, wenn er fertig ist. Damit weiß die *__Controller* Methode, von welchem Child-Step sie gerade aufgerufen wurde

- Eine Variable mit dem Namen *CalledChildStepsList*. Diese enthält alle ChildSteps - ebenfalls für die *_Controller* Methode, damit diese weiß, ob alle Methoden returniert sind (Codebeispiel 5.20).

Algorithm 5.20: Anlegen der Variable CalledChildStepsList

```
1  private void AddCalledChildStepsList(SystemDecompType stepContainerComponent)
2  {
3    var counterVariable = new VariableType
4    {
5      Name = SimulationConstants.CalledChildStepsList,
6      Id = GuidGenerator.NewGuid(),
7      Structure = new VariableTypeStructure
8      {
9        Id = GuidGenerator.NewGuid(),
10       Types = new List<DomainProAbstractSemanticType>()
11     },
12     Text = new DomainProText
13     {
14       Id = GuidGenerator.NewGuid(),
15       Instructions = new List<Instruction>()
16     },
17     ImplementationType = "IList<DP_IComponent>",
18     InitialValue = "new List<DP_IComponent>()"
19   };
20
21   stepContainerComponent.Structure.Types.Add(counterVariable);
22 }
```

5.3.4 Code Generierung in parallelen Methoden

Nach dem alle Container (Leaf, Sequential, Parallel) erfolgreich erstellt worden sind, wird noch Code in die parallelen Methoden eingefügt (Codebeispiel 5.21).

Algorithm 5.21: Erzeugen des Codes für parallel Steps

```
1  internal void GenerateCodeForParallelRangeIfNecessary()
2  {
3    var parallelSteps = InputModel.Lj.Steps.SelectManyRecursive(s => s.Substeps).Where(s => s.Kind ==
         StepType.Parallel);
4
5    foreach (var parallelStep in parallelSteps)
6    {
7      GenerateCodeForCallingVirtualMethod(parallelStep);
8      GenerateCodeForControllerMethod(parallelStep);
9    }
10 }
```

Zuerst wird die *_Calling* Methode betrachtet. Ist die Kardinalität gesetzt, so muss zwischen zwei Fällen unterschieden werden:

- Wird über eine Ressource iteriert
- Gibt es einen Wertebereich über den iteriert wird

Im ersten Fall wird die Schleife, wie im Codebeispiel 5.22 gezeigt, generiert. Voraussetzung ist wieder, dass

- die Kardinalität gesetzt ist
- der UpperBound * entspricht
- und ein Parameter-Controller vorhanden ist.

Algorithm 5.22: Generieren des Schleifen-Codes

```
1  private string GenerateLoopHeaderForRessource(Step parallelStep, ParameterController
       referenceElement)
2  {
3    return string.Format("foreach(var subStepInstance in {0}Container.Parent.{1})", parallelStep.Name
         , _typeResolver.GetTypeNameFromEcorePath(referenceElement.Ref));
4  }
```

Ist ein *Binding* im aktuellen Schritt vorhanden, so wird dieses generiert. Das bedeutet, dass einer gegebenen Variable die aktuelle Instanz der Iteration zugewiesen wird.

Ist ein Wertebereich angegeben, so wird eine Schleife für den Wertebereich erstellt (Codebeispiel 5.23).

Algorithm 5.23: Loop-Header für Schleifen mit Wertebereich

```
1  private string GenerateLoopHeaderForRange(int loopCounterVariableValue)
2  {
3    return string.Format("for(int calls = 0; calls < {0}; calls++)", loopCounterVariableValue);
4  }
```

Die Generierung des Schleifen-Body erfordert auch noch die Miteinbeziehung der Child-Steps (Codebeispiel 5.24). Dies ist erforderlich, da je nach Typ, die aufrufende Methode anders heißt.

Algorithm 5.24: Generieren jener Methode, welcher aufgerufen werden soll

```
1  if (substep.Kind == StepType.Sequential || substep.Kind == StepType.Parallel)
2  {
3    methodNameToCall = ((VirtualNodeStepInformations)substep.Information).VirtualMethods.First().
       MethodName;
4  }
5  else
6  {
7    methodNameToCall = ((LeafNodeStepInformation)substep.Information).MethodName;
8  }
```

VirtualNodeStepInformations als auch LeafNodeStepInformation enthalten Informationen, welche Methoden für welchen Step generiert worden sind (Codebeispiel 5.25). Das ist erforderlich, da das Zielmodell auf diese Methoden per Id verweist und diese daher zwischengespeichert werden muss.

Der Inhalt des Body wurde bereits im Kapitel 4 diskutiert. Es wurde auch festgestellt, dass der Transfer von Daten nicht mit Pfeilen möglich ist. Der Parent muss dem Child-Step Daten in der Schleife übergeben. Codebeispiel 5.26 zeigt den Vorgang.

Algorithm 5.25: Klassen für das Speichern der zusätzlichen Methoden zum Abbilden von Methoden-Sequenzen

```
1  public class VirtualNodeStepInformations : StepInformation
2  {
3    public VirtualNodeStepInformations()
4    {
5      VirtualMethods = new List<VirtualNodeStepInformation>();
6    }
7
8    public IList<VirtualNodeStepInformation> VirtualMethods { get; private set; }
9  }
10
11 public class VirtualNodeStepInformation : IEquatable<VirtualNodeStepInformation>
12 {
13   public VirtualNodeStepInformation(Guid methodId, string methodName)
14   {
15     MethodId = methodId;
16     MethodName = methodName;
17   }
18
19   public Guid MethodId { get; private set; }
20   public string MethodName { get; private set; }
21
22   public override bool Equals(object right)
23   { ... }
24
25   public bool Equals(VirtualNodeStepInformation other)
26   { ... }
27
28   public override int GetHashCode()
29   { ... }
30 }
31
32 public class LeafNodeStepInformation : StepInformation
33 {
34   public Guid MethodId { get; set; }
35   public string MethodName { get; set; }
36 }
```

Algorithm 5.26: Generierung des Codes für die Datenübergabe an den Child-Step

```
1  private void AppendDataTransferCode(Step step, Step substep, StringBuilder codeStringBuilder,
       string instanceName)
2  {
3    foreach (var binding in substep.Bindings.Where(s => s.BindingType is InOutBinding || s.
         BindingType is InBinding))
4    {
5      var left = _typeResolver.GetTypeNameFromEcorePath(binding.Left);
6      var right = _typeResolver.GetTypeNameFromEcorePath(binding.Right);
7
8      if (left != SimulationConstants.Agent)
9      {
10           codeStringBuilder.AppendLineFormat("  {0}.{1} = {2}Container.{3};", instanceName, left,
                 step.Name, right);
11     }
12   }
13 }
```

5.4 Erzeugung der Pfeile (Method Sequences)

Wie bereits im Kapitel 4 festgestellt, können Methoden andere Methoden über Pfeile aufrufen. Die Pfeile werden erstellt, nachdem alle Container und Methoden erstellt wurden (Codebeispiel 5.27).

Algorithm 5.27: Erzeugung der Method-Sequences

```
1  internal void CreateMessageSequenceLinks()
2  {
3    var steps = InputModel.Lj.Steps.SelectManyRecursive(s => s.Substeps);
4
5    foreach (var step in steps)
6    {
7      if (step.Kind == StepType.Sequential)
8      {
9        _sequentialStepMethodSequenceGenerator.HandleSequentialStep(step);
10     }
11
12     if (step.Kind == StepType.Parallel)
13     {
14       _parallelStepMethodSequenceGenerator.HandleParallelStep(step);
15     }
16
17     if (step.Kind == StepType.Leaf && step.IsCardinalityRangeGreaterOne())
18     {
19       _leafStepMethodSequenceGenerator.HandleCardinalityOfLeafNode(step);
20     }
21   }
22 }
```

5.4.1 Leaf Nodes

Ist die Kardinalität eines Leaf-Nodes gesetzt und ist die Anzahl der Iterationen größer 1, so ist es erforderlich, zwischen den beiden Methoden im Leaf eine Abfolge durch Pfeile zu machen. Die Code ist im Codebeispiel 5.28 abgebildet.

Algorithm 5.28: Verarbeitung der Kardinalität im Leaf-Node

```
1  internal void HandleCardinalityOfLeafNode(Step step)
2  {
3    var leafInformation = (LeafNodeStepInformation)step.Information;
4
5    var trigger = string.Format("return {0}.{0}Container.{0}Counter < {1};", step.Name, step.
         Cardinality.UpperBound);
6
7    CreateMethodSequenceAndAddToOutputModel(leafInformation.MethodId, step.CardinalityStepInformation
         .ControllerMethodId, string.Format("{0}_Cardinality_Calling", step.Name), trigger);
8    CreateMethodSequenceAndAddToOutputModel(step.CardinalityStepInformation.ControllerMethodId,
         leafInformation.MethodId, string.Format("{0}_Cardinality_Returning", step.Name));
9  }
```

Ebenfalls ist dem Code zu entnehmen, dass im ersten Pfeil der Trigger für die Prüfung des Counters passiert. Das ist erforderlich, um die Iteration mehrfach wiederholen zu können.

5.4.2 Sequential Nodes

Das Erzeugen der Pfeile bei sequentiellen Methoden ist etwas komplexer. Die wichtigsten Aspekte wurden im Codebeispiel 5.29 herausgehoben.

Algorithm 5.29: Erzeugen von Method-Sequences bei sequentiellen Knoten

```
1  foreach (var substep in step.Substeps)
2  {
3    var callingMethod = stepInformation.VirtualMethods[stepCounter];
4    var returningMethod = stepInformation.VirtualMethods[stepCounter + 1];
5
6    ...
7
8    if (substep.Kind == StepType.Leaf)
9    {
10     CreateSequencesForLeaf(step, substep, callingMethod, returningMethod, callingTrigger,
           returnTrigger);
11   }
12
13   if (substep.Kind == StepType.Sequential || substep.Kind == StepType.Parallel)
14   {
15     CreateSequencesForSequentialOrParallelStep(step, substep, callingMethod, returningMethod,
           callingTrigger, returnTrigger);
16   }
17
18   stepCounter++;
19 }
```

Um sich das Ganze besser vorstellen zu können, zeigt Abbildung 5.1 einen Ausschnitt eines Beispiels.

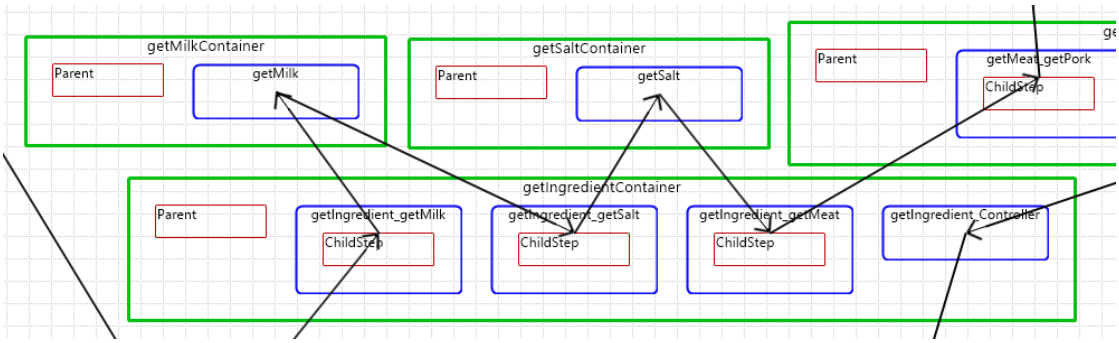

Abbildung 5.1: Method Sequences

Da für jeden Substep eine eigene "virtuelle"Methode erstellt wurde, kann man jeweils von dieser zum Child-Step (*getIngredient_getMilk* → *getMilk*) und vom Child-Step (*getMilk*) zur nächsten "virtuellen"Methode (*getIngredient_getSalt*) einen Pfeil erzeugen.

Wie aus dem Codeausschnitt auch zu entnehmen ist, erfährt ein Child-Step des Typs Parallel oder Sequentiell eine besondere Behandlung: der Pfeil geht nicht - wie bei Leaf Child-Step Knoten - zur gleichen Methode, sondern jeweils zur ersten bzw. letzten Methode.

Ein weiterer wichtiger Aspekt bei sequentiellen Pfeilen sind die Transfer-Zuweisungen. Codebeispiel 5.30 zeigt die Zuweisung der Referenzen für den Child-Step (eingehende Zuweisung).

Algorithm 5.30: Generieren der Zuweiung für eingehende Bindings

```
1  private string GenerateTransferValueForBindingsIn(Step step, Step subStep, string stepMethodName,
       string subStepMethodName)
2  {
3    var stringBuilder = new StringBuilder();
4
5    foreach (var binding in subStep.Bindings.Where(s => s.BindingType is InOutBinding || s.
         BindingType is InBinding))
6    {
7      var left = _typeResolver.GetTypeNameFromEcorePath(binding.Left);
8      var right = _typeResolver.GetTypeNameFromEcorePath(binding.Right);
9
10     var isListBinding = (left.EndsWith("List") && right.EndsWith("List") == false) || (left.
           EndsWith("List") == false && right.EndsWith("List"));
11     var isAgentBinding = left.Equals(SimulationConstants.Agent) || right.Equals(SimulationConstants
           .Agent);
12
13     if (isListBinding == false && isAgentBinding == false)
14     {
15       stringBuilder.AppendLineFormat("{0}.{1}Container.{2} = {3}.{4}Container.{5};",
             subStepMethodName, subStep.Name, left, stepMethodName, step.Name, right);
16     }
17   }
18
19   return stringBuilder.ToString();
20 }
```

Wie man dem Code entnehmen kann, wird der Code für den Transfer nur generiert, wenn

- Es sich um keine Liste-Einzelelement-Zuweisung handelt (zur Erinnerung: Listen erkennt man anhand der Convension, da sie mit *List* enden)
- Es sich um keinen Agenten handelt (zur Erinnerung: auch die Zuweisung der Agenten passiert in Bindings. Allerdings will man hier keine Zuweisung per Transfer, sondern einen Aufruf des richtigen Agenten - und das passiert im Code der Methode)

Der Code für den ausgehenden Transfer passiert analog.

Eine weitere Eigenschaft von Pfeilen sind Trigger, welche einen *bool* Wert returnieren, welcher angibt, ob der Pfeil ausgeführt wird oder nicht. Der aufrufende Pfeil kann immer ausgeführt werden. Für den retournierenden Pfeil ist folgendes zu beachten:

- Wenn eine Kardinalität gesetzt ist, der UpperBound * entspricht und ein Controller (e.g. Liste) vorhanden ist, so wird der Pfeil so lange ausgeführt, bis der Counter kleiner ist, als die Anzahl der Elemente in der Liste
- Falls der UpperBound eine nummerische Zahl ist, wird diese als obere Grenze übernommen.

5.4.3 Parallel Steps

Wie bereits im Kapitel 4 und 5 festgestellt, befindet sich die Logik der Iterations-Steuerung im Code der aufrufenden Methode und nicht in den Pfeilen. Allerdings kann ein parallel Step auch sequentielle Steps aufrufen, sodass die retournierenden Pfeile gleich wie bei sequentiellen Steps behandelt werden. Ebenfalls muss der retournierende Pfeil Informationen über den Child-Step beinhaltet, damit der *_Controller* des Parallel-Step feststellen kann, welche Methode terminiert hat (Codebeispiel 5.31).

Algorithm 5.31: Generieren der Transfer Eigenschaften für einen parallelen Step

```
1  private static string GenerateTransferValue(Step step, Step substep)
2  {
3    string transfer = SimulationConstants.DefaultMethodTransferValue;
4
5    if (substep.Kind == StepType.Parallel || substep.Kind == StepType.Sequential)
6    {
7      transfer = string.Format("{0}_Controller.ReturningCallerReference = {1}_Controller.{1}Container
           ;", step.Name, substep.Name);
8    }
9    else
10   {
11     transfer = string.Format("{0}_Controller.ReturningCallerReference = {1}.{1}Container;", step.
           Name, substep.Name);
12   }
13
14   return transfer;
15 }
```

Der aufrufende Trigger bei parallelen Steps ist immer *false*. Das bedeutet, dass die Pfeile keine Funktion erfüllen, sondern nur der Vollständigkeit halber angeführt sind. Die Steuerung erfolgt durch die aufrufende Methode (wie im Kapitel 5 bereits ausführlich diskutiert).

5.5 Anordnen der Elemente

Die Elemente, welche das Transformationstool generiert, beinhalten keine Information über Größe und Position. D.h., sie würden alle übereinander in minimaler Größe im Simulationstool erscheinen und dies führt unweigerlich zum Absturz des DomainPro Designers. Elemente werden daher von links nach rechts angeordnet. Verschachtelte Elemente haben jeweils 25 Pixel Abstand (in allen Richtungen) zu ihrem Parent. Codebeispiel 5.32 zeigt die wichtigsten Methoden der Style-Generierung.

Codebeispiel 5.32 enthält folgende wichtige Punkte:

Algorithm 5.32: Generieren des Styles für alle Elemente

```
1  public static class StyleGenerator
2  {
3    ...
4
5    public static void ApplyStyleOnDocument(XDocument document)
6    {
7      Recursion(document.XPathSelectElement(".//anyType[not(@xsi:type='MethodSequence')]",
           XmlNamespaceManager));
8      ApplyMinimumStyleOnMethodSequences(document.XPathSelectElements(".//anyType[@xsi:type='
           MethodSequence']", XmlNamespaceManager));
9    }
10
11   private static void ApplyMinimumStyleOnMethodSequences(IEnumerable<XElement> elements)
12   { ... }
13
14   private static void Recursion(XElement xPathSelectElement)
15   {
16     foreach (var childElement in GetChildsOfFirstGrade(xPathSelectElement))
17     {
18       Recursion(childElement);
19     }
20
21     var type = xPathSelectElement.Attribute(XsiNamespace + "type");
22
23     if (type == null || string.IsNullOrWhiteSpace(type.Value))
24     {
25       throw new ArgumentException("type cant be null");
26     }
27
28     var anyTypeChildElementsOfCurrentElement = GetChildsOfFirstGrade(xPathSelectElement).ToArray();
29
30     int width;
31     int height;
32
33     InitializeVariablesForType(type.Value, out width, out height);
34
35     if (anyTypeChildElementsOfCurrentElement.Any())
36     {
37       width = anyTypeChildElementsOfCurrentElement.Select(GetWidthPropertyOfElement).Sum() + 25 * (
             anyTypeChildElementsOfCurrentElement.Count() + 1);
38       height = anyTypeChildElementsOfCurrentElement.Select(GetHeightPropertyOfElement).Max() + 50;
39
40       // If we have more than 1 childs move each of them side by side to the right
41       if (anyTypeChildElementsOfCurrentElement.Count() > 1)
42       {
43         XElement leftChild = null;
44
45         foreach (var child in anyTypeChildElementsOfCurrentElement)
46         {
47           if (leftChild != null)
48           {
49             GeXLocationProperty(child).Value = (GetXLocationPropertyOfElement(leftChild) +
                   GetWidthPropertyOfElement(leftChild) + 25).ToString();
50           }
51
52         leftChild = child;
53       }
54     }
55   }
```

Algorithm 5.33: Generieren des Styles für alle Elemente cont.

```
1     var location = new XElement("Location",
2     new XElement("X", 25),
3     new XElement("Y", 25));
4
5     var size = new XElement("Size",
6     new XElement("Width", width),
7     new XElement("Height", height));
8
9     xPathSelectElement.Add(size);
10    xPathSelectElement.Add(location);
11    }
12
13    private static void InitializeVariablesForType(string type, out int width, out int height)
14    { ... }
15
16    /// <summary>
17    /// Gets anyType childs which have the given anyType element as parent
18    /// </summary>
19    private static IEnumerable<XElement> GetChildsOfFirstGrade(XElement element)
20    {
21      if (element == null)
22      {
23        throw new ArgumentNullException("element");
24      }
25
26      if (element.Name != "anyType")
27      {
28        throw new ArgumentException("element should by of anyType");
29      }
30
31      var childElements = element.XPathSelectElements(".//anyType[not(@xsi:type='MethodSequence')]",
            XmlNamespaceManager);
32
33      foreach (var childElement in childElements)
34      {
35        var parentElement = childElement.XPathSelectElement("ancestor::anyType[not(@xsi:type='
              MethodSequence')][1]", XmlNamespaceManager);
36
37        if (parentElement.Equals(element))
38        {
39          yield return childElement;
40        }
41      }
42    }
43
44    ...
45  }
```

- Ein Pfeil braucht ein Icon, da das Simulationstool sonst abstürzt
- *GetChildsOfFirstGrade* gibt anyType Elemente zurück, welche ein gegebenes anyType Element als Vorgänger haben. In der XML Struktur sind diese Elemente allerdings nicht direkt verwandt - es befinden sich dazwischen andere Knoten. Ausgenommen werden anyType Knoten vom Typ MethodSequence - diese werden nicht verschachtelt angeordnet.
- Die Recursion durchwandert rekursiv alle Elemente und setzt den Style des tiefsten Kind-Element als erstes. Alle weiteren Elemente übernehmen die Breite der Kinderelemente inkl. eines entsprechenden Abstands.

5.6 Injezierung des Glue-Codes

Bei jeder Transformation des erstellten Tools werden neue Globally Unique Identifier (GUID) Werte für diverse Elemente erstellt. Daher ist es nicht ohne weiteres möglich, den Code, welcher ein Entwickler nachträglich im Simulationstool eingetragen hat, zu übernehmen. Da Entwickler meistens den Code in Methoden schreiben wollen, erfolgt die Injezierung des Codes unter Bekanntgabe von Containername und Methodenname. Des Weiteren muss man zwischen zwei Methoden der Injezierung unterscheiden:

- Code zum bestehenden Code anhängen. Dies ist die einfachste Form, da die Manipulation des bestehenden Codes einfach ist.
- Bestehenden Code ändern. Diese Variante ist etwas schwieriger, da der Entwickler die Zeilen / Position im vorhanden Code kennen muss.

Codebeispiel 5.34 zeigt ein Beispiel, in dem eine Schleife angepasst wurde. Ziel war es, eine Schleife, welche durch die Kardinalität 1..5 spezifiziert war, durch einen Zufallswert zu ersetzen.

Wie man sieht, wurde zuerst auf den Code der Methode *createAnnunciators_Calling* des *createAnnunciatorsContainer* referenziert. Codebeispiel 5.35 zeigt die Iteration über alle vorhandenen Codezeilen.

Algorithm 5.35: Iteration über die Zeilen des bestehenden Codes

```
1  foreach (string line in new LineReader(new StringReader(currentCode)))
```

Entspricht der Code dem Schleifenkopf, wird dieser ersetzt. Wird e.g. der Container oder die Methode umbenannt und kann das Transformationstool die Elemente nicht eindeutig finden, bricht das Tool mit einem Fehler ab.

5.7 Testen des Codes

Initial wurden zwei Testmethoden identifiziert:

Algorithm 5.34: Änderung des automatisch generierten Codes

```
1  private static void MakeAnnunciatorAmountRandom(TransformationWorkflow transformationWorkflow)
2  {
3    transformationWorkflow.ManipulateRunCodeOfComponentModel(
4    "createAnnunciatorsContainer",
5    "createAnnunciators_Calling",
6    currentCode =>
7    {
8      var codeStringBuilder = new StringBuilder();
9
10     foreach (string line in new LineReader(new StringReader(currentCode)))
11     {
12       // Replace this line
13       if (line.StartsWith("for(int calls = 0; calls < 5; calls++)"))
14       {
15         codeStringBuilder.AppendLine("int amountOfAnnunciators = 1;");
16         codeStringBuilder.AppendLine();
17         codeStringBuilder.AppendLine("if(createAnnunciatorsContainer.occurrenceRef.IsBigOccurrence)
               ");
18         codeStringBuilder.AppendLine("{");
19         codeStringBuilder.AppendLine("  amountOfAnnunciators = createAnnunciatorsContainer.
               Environment.EnvironmentRandomGenerator.Next(1,5);");
20         codeStringBuilder.AppendLine("}");
21         codeStringBuilder.AppendLine();
22         codeStringBuilder.AppendLine("for(int calls = 0; calls < amountOfAnnunciators; calls++)");
23       }
24       else
25       {
26         codeStringBuilder.AppendLine(line);
27       }
28     }
29
30     return codeStringBuilder.ToString();
31   });
32 }
```

- Im Test die Erwartungen bzgl. des Ausgabemodels spezifizieren (Validierung der Objekt-Struktur).
- Die Ausgabe gegen ein vorhandenes XML vergleichen.

Es wurde zugunsten der zweiten Methode entschieden, da initial die Struktur nicht zur Gänze klar war. Der Nachteil der Methode liegt definitiv darin, dass alle Aspekte der Transformation überprüft wurden. Dadurch haben Tests plötzlich nicht mehr funktioniert, weil neue Transformationsfeatures noch nicht in der Referenz-XML des Tests waren. Allerdings war dieser Nachteil gleichzeitig ein Vorteil, da Tests in weiterer Folge automatisch gezeigt haben, wo sich etwas geändert hat. Die Ausgabe der fehlgeschlagenen Tests wurde anschließend in DomainPro manuell auf Sinnhaftigkeit überprüft und angepasst.

Ein weiteres Problem war, dass sich Globally Unique Identifier (GUID) bei jeder Generierung geändert haben. Der erste Ansatz war, dass man diese Guid durch einen Stub ersetzt und im Testfall immer eine leere Guid generiert (00000000-0000-0000-0000-000000000000). Dies funktionierte bis zum Erstellen der Methodent-Sequenz-Transformation, da Pfeile immer per Guid auf andere Methoden verweisen und somit eine Zuordnung zu den Methoden nicht mehr möglich war. Daher generiert das Tool momentan das Ausgabedokument mit Guids. Diese werden anschließend alle durch eine leere Guid ersetzt, damit

die Dokumente leichter vergleichbar sind. Codebeispiel 5.36 zeigt die Ersetzung, die dank LINQ to XML sehr einfach ist.

Algorithm 5.36: Ersetzen der GUIDs mit einer Empty-GUID

```
1  private static void ReplaceAllGuidsWithEmptyGuid(XDocument sourceXDocument)
2  {
3    ReplaceAllGuidsWithEmptyGuidByElementName(sourceXDocument, "Id");
4    ReplaceAllGuidsWithEmptyGuidByElementName(sourceXDocument, "Role1Id");
5    ReplaceAllGuidsWithEmptyGuidByElementName(sourceXDocument, "Role2Id");
6  }
7
8  private static void ReplaceAllGuidsWithEmptyGuidByElementName(XDocument sourceXDocument, string
       elementName)
9  {
10   foreach (var idElement in sourceXDocument.Root.Descendants(elementName))
11   {
12     Guid currentValue;
13
14     // Only replace it if its an valid Guid
15     if (Guid.TryParse(idElement.Value, out currentValue))
16     {
17       idElement.Value = Guid.Empty.ToString();
18     }
19   }
20 }
```

Der Vergleich der XML Dokumente wurde mit dem Microsoft XML Diff & Patch GUI Tool[2] gemacht. Dieses Tool führt keinen textuellen Vergleich durch, sondern erlaubt es auch, Optionen zu spezifizieren (e.g. die Reihenfolge der Kindelemente ignorieren). Codebeispiel 5.37 zeigt den Vergleich des generierten Modells mit einer XML Referenz File.

Algorithm 5.37: Vergleichen von XML Dokumenten

```
1  internal bool IsXmlEqual(Sim4SysMlModel systemModel, FileInfo expectedXml)
2  {
3    var sourceXDocument = XDocument.Parse(Serialisation.ConvertInstanceToString(systemModel));
4
5    var referenceXmlDocument = new XmlDocument();
6    referenceXmlDocument.Load(expectedXml.FullName);
7
8    ReplaceAllGuidsWithEmptyGuid(sourceXDocument);
9
10   var xmlDiff = new XmlDiff(XmlDiffOptions.IgnoreChildOrder | XmlDiffOptions.IgnoreWhitespace) {
         Algorithm = XmlDiffAlgorithm.Auto };
11
12   return xmlDiff.Compare(sourceXDocument.ToXmlDocument(), referenceXmlDocument);
13 }
```

Wie im Codebeispiel 5.37 zu sehen ist, werden Leerezeichen (*XmlDiffOptions.IgnoreWhitespace*) und die Reihenfolge der Kindelemente (*XmlDiffOptions.IgnoreChildOrder*) ignoriert.

[2] https://msdn.microsoft.com/en-us/library/aa302295.aspx

5.8 Design & Simulations Tool

Das erzeugte Modell kann im DomainPro Designer geladen und modifiziert werden (Abbildung 5.2).

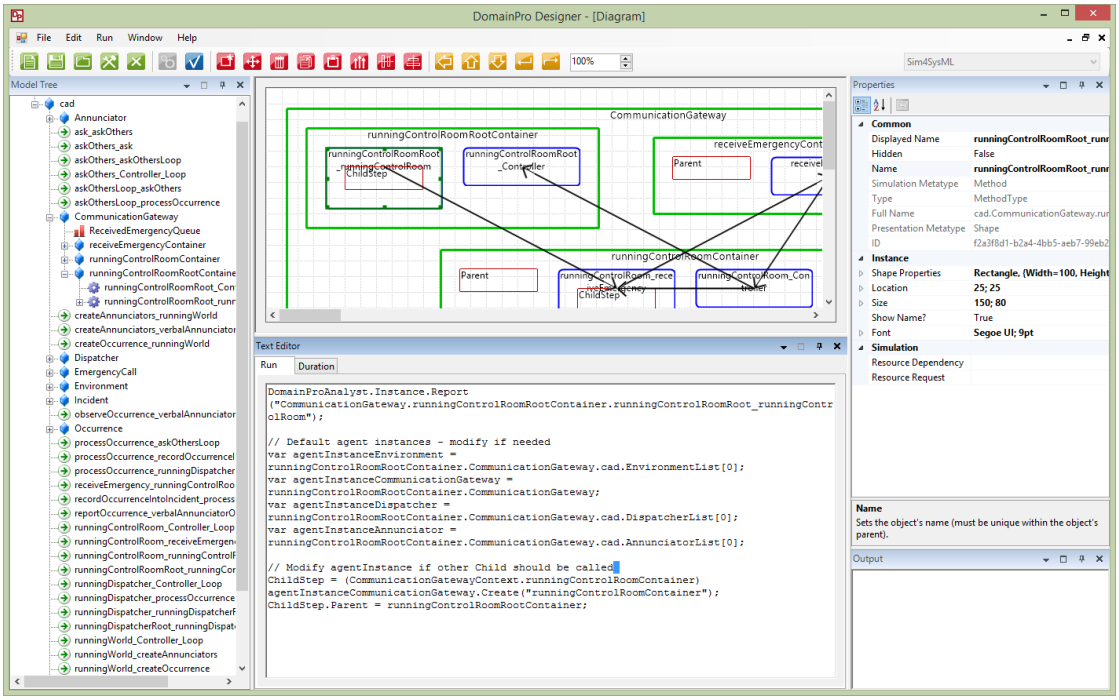

Abbildung 5.2: DomainPro Designer mit Bearbeitung des Glue-Codes

Der Entwickler kann weitere Elemente einfügen, oder den Glue-Code erweitern bzw. ändern. Die Ausführung der Simulation passiert im DomainPro Analyst. Um die Simulation starten zu können, ist es erforderlich, die Startinstanzen anzugeben. (Abbildung 5.3).

Ebenfalls muss angegeben werden, wie lange die Simulation läuft (Abbildung 5.4).

Wenn die Simulation ausgeführt wird, wird der vom Transformationstool generierte Debug-Code ausgegeben. Dieser zeigt die einzelnen Methodenaufrufe (Abbildung 5.5).

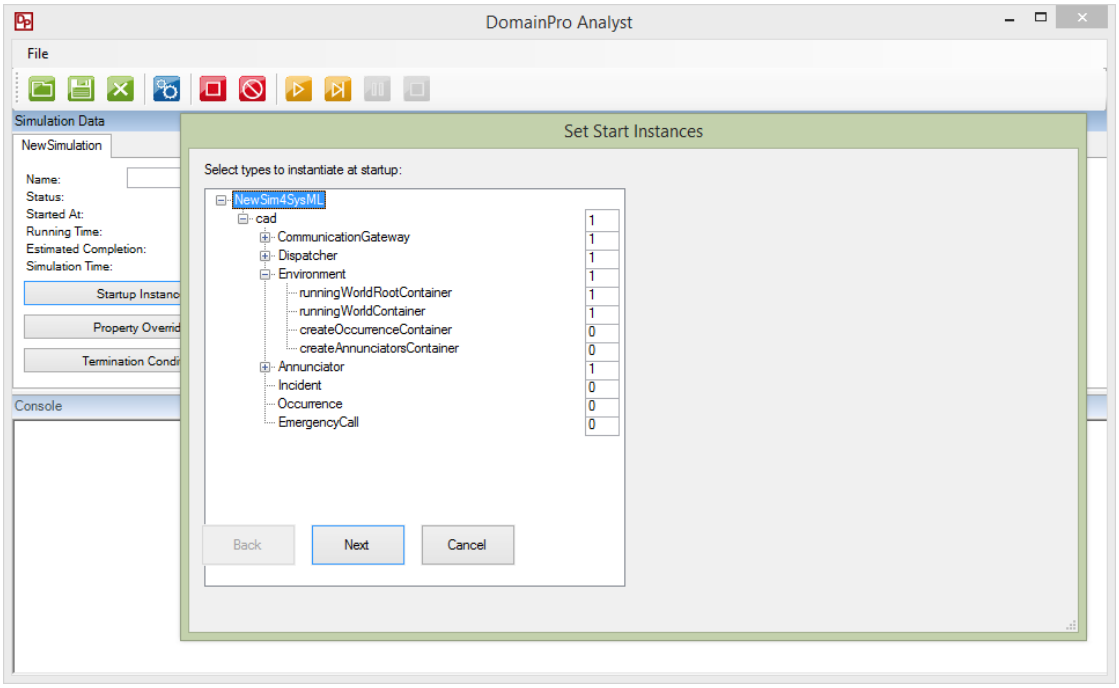

Abbildung 5.3: DomainPro Designer mit der Konfiguration der Startinstanzen

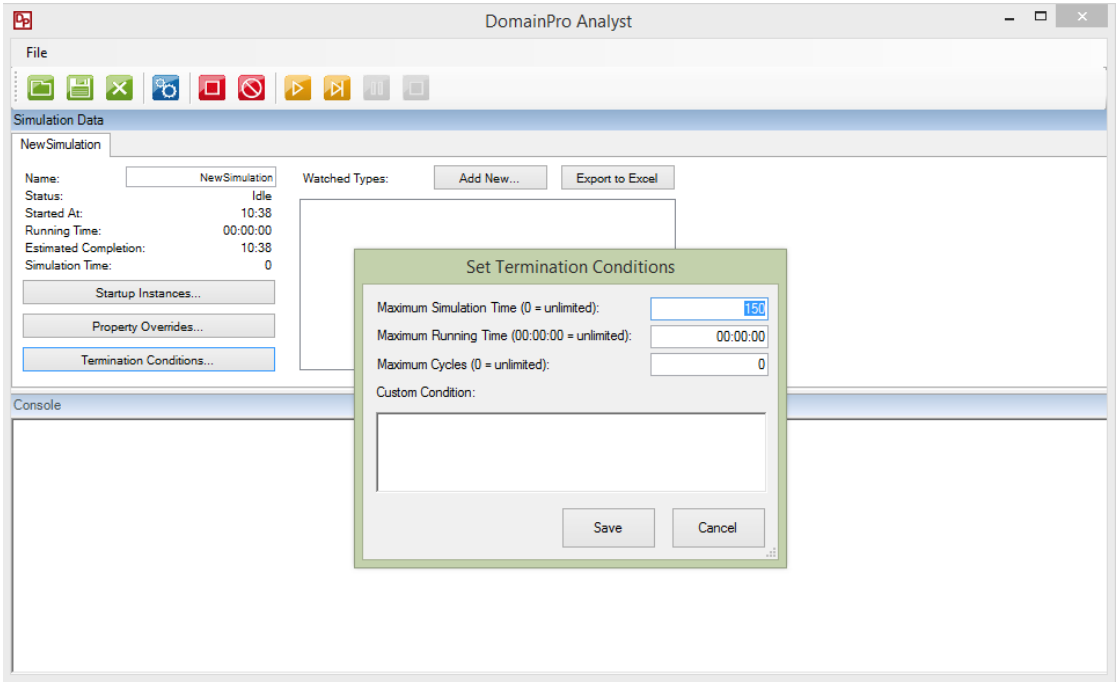

Abbildung 5.4: DomainPro Designer mit Konfiguration der Simulationsdauer

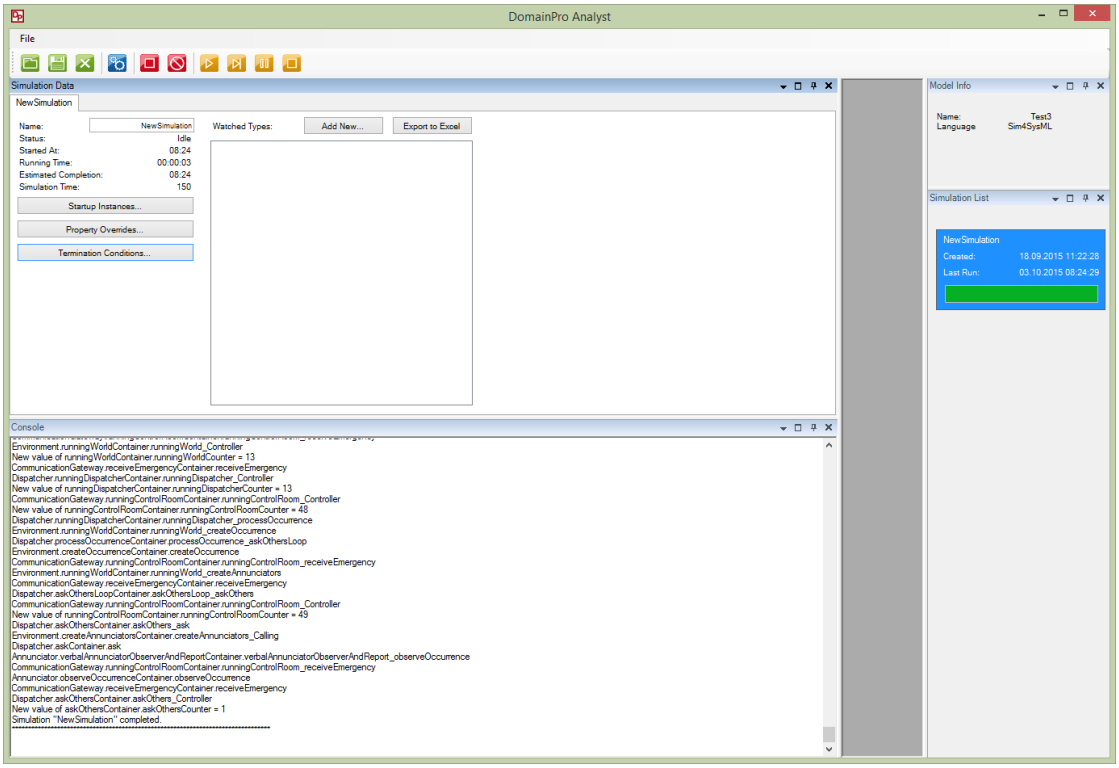

Abbildung 5.5: DomainPro Analyst beim Ausführen der Simulation

KAPITEL 6

Analyse

Das Transformationstool wurde auf das Eingangs diskutierte Problem der Feuerwehrleitstelle angewandt. Dabei wurde die Interaktion mit dem Notrufer und dem Disponenten mit Hilfe von LittleJIL und hADL analysiert. Anhand des Fallbeispiels soll in diesem Kapitel festgestellt werden, wie gut kollaborative Umgebungen, welche in hADL und LittleJIL beschrieben sind, in eine DomainPro Struktur umgewandelt werden konnten.

6.1 Beschreibung der Komponenten und Abläufe

Die Komponenten wurden in zwei unterschiedliche Strukturen aufgeteilt: dem *Controlroom* und der *World* (Codebeispiel 6.1).

Algorithm 6.1: Die Grundstrultur

```
1  package cad
2  hADL Model name: cad
3  {
4    Structure name: controlroom
5    {
6      ...
7    }
8
9    Structure name: world
10   {
11     ...
12   }
13 }
```

Der *Controlmroom* besteht aus mehreren Komponenten, welche die Kollaborations-Entitäten wiederspiegeln (Codebeispiel 6.2).

Die Aufgaben der Komponenten sind wie folgt angedacht (siehe auch Kapitel 3) - eine Zusammenfassung:

Algorithm 6.2: Aufbau des *controlroom*

```
1  // A Communication Gateway is the entry point for any kind of communication into the control room
       and also provides functionality for the Dispatcher to establish calls with the outside world.
2  Component name: CommunicationGateway
3
4  // A Dispatcher is a person which is responsible for coordinating Resources (e.g. fire trucks). He
       is also responsible for answering all kinds of Annunciator requests and handle them i a
       predefined way.
5  Component name: Dispatcher
6    Actions
7      [createEmergencyAndEnterInformations primitives: [C] cardinality: 0..*]
8
9  // An Incident is a system internal representation of an Occurrence containing data about the
       Occurrence
10 Artifact name: Incident
11   Actions
12     // A Dispatcher transfers information (semi-automatic via voice or by triggering an automatic
           process for textual)
13     [transferInformationObservedByDispatcherToIncident primitives: [C] cardinality: 1..*]
14
15 Link [ transfering : cad.controlroom.Dispatcher.createEmergencyAndEnterInformations => cad.
       controlroom.Incident.transferInformationObservedByDispatcherToIncident]
16
17 Relations {
18   [incidentToOccurrenceRef : Incident references cad.world.Occurrence]
19 }
```

- **CommunicationGateway**: Nimmt die Notrufe entgegen (egal in welcher Form) und kümmert sich um die Weiterleitung an jene Disponenten, welche gerade kein Ereignis (Occurrence) bearbeiten.
- **Dispatcher**: Ein Dispatcher ist für die Koordination zuständig. Ebenfalls nimmt er Notrufe aller Art entgegen und kümmert sich um die Eingabe und Verarbeitung im System.
- **Incident**: Repräsentiert eine Occurrence in der Domäne der Leitstelle. Meist in Form eines Datensatzes im Einsatzleitsystem, welcher von mehreren Disponenten bearbeitet werden kann.

Neben der *Controlroom* Komponente gibt es auch noch die *World* Komponente (Codebeispiel 6.3):

- **Environment**: Kann als Container gesehen werden, welcher es erlaubt, Aktivitäten der Welt zu modellieren.
- **Occurrence**: Ist der Vorfall / das Ereignis in der realen Welt.
- **Annunciator**: Ist der Melder des Vorfalls
- **EmergencyCall**: Der Notruf, welcher durch den Annunciator an die Leitstelle getätigt wird.

Das Resultat der Transformation wird in den Folgekapiteln gezeigt und diskutiert.

Algorithm 6.3: Aufbau der *world*

```
1  Structure name: world
2  {
3    // The Environment represents the real world. In this world things happen (Occurrences) and need
         attention
4    Component name: Environment
5
6    // A Occurrence is something that happened in the environment. It may be a traffic accident or a
         structural fire or something else.
7    Artifact name: Occurrence
8      Actions
9        [observe primitives: [R] cardinality: 1..*]
10
11   // A Annunciator is a generic term for a person or device in the Environment which observes the
         Occurrence
12   Component name: Annunciator
13     Actions
14       // A verbal annunciator may be able to observe multiple Occurrences - but since all other
             types of annunciator can not, we assume, that a verbal annunciator also observers only 1
              Occurrence during
15       // a simulation run
16       [observe primitives: [R] cardinality: 1..1]
17
18   // An Emergency Call happens between an arbitrary Annunciator and a CommunicationGateway. It may
         be a voice call or a simple flow of data. A voice call is abstracted by an unidirectional
         flow of information
19   Message name: EmergencyCall
20     Actions
21       [provideInformationToEmergencyCallCenter primitives: [C]]
22
23   Link [ observing : cad.world.Annunciator.observe => cad.world.Occurrence.observe]
24
25   Relations {
26      [emergencyCallToOccurrenceRef : EmergencyCall references cad.world.Occurrence]
27   }
28 }
```

6.1.1 Analyse

Die Umsetzung der Komponenten in die DomainPro-Strukturen erfolgt direkt, da DomainPro auch Komponenten als Basismodellierungselement kennt. Beschreibungen zu den Elementen lassen sich leider nicht hinzufügen, da die Information schon bei der Transformation zwischen DSL und XML verloren gehen. Dies macht es erforderlich, die Bedeutung mancher Container im Ursprungsmodell nachzulesen.

6.2 Analyse der Abläufe

Die Interaktion zwischen den Komponenten stellt den weitaus komplexeren Teil dar. Auf Basis der exemplarischen Umsetzung, soll eine Analyse durchgeführt werden.

6.2.1 Die Root-Steps

Um autonome Abläufe modellieren zu können, wurden mehrere Root-Steps erstellt (Codebeispiel 6.4).

Algorithm 6.4: Die Root-Steps des Fallbeispiels

```
1  LittleJIL Process name: dailyWork
2    Root Steps {
3      Step name: runningWorldRoot of type Sequential
4      Local Data: {
5          [ agent as Resource of hADLType cad.world.Environment]
6        }
7        Child Steps: {
8        ...
9      }
10
11     Step name: runningControlRoomRoot of type Sequential
12       Local Data: {
13         [ agent as Resource of hADLType cad.controlroom.CommunicationGateway]
14       }
15       Child Steps: {
16         ...
17       }
18
19     Step name: runningDispatcherRoot of type Sequential
20       Local Data: {
21         [ agent as Resource of hADLType cad.controlroom.Dispatcher]
22       }
23       Child Steps: {
24         ...
25       }
26   }
```

- **runningWorldRoot**: Kümmert sich um das Erzeugen der Occurrence und Annunciator Entitäten.
- **runningControlRoomRoot**: Bildet alle Leitstellenaktivitäten ab.
- **runningDispatcherRoot**: Der Dispatcher wurde als autonome Instanz abgebildet.

6.2.2 Analyse

Wie man der DSL entnehmen kann, werden alle drei Steps auf unterschiedlichen Containern-Instanzen (Agenten) ausgeführt. Das Konzept der Agenten ließ sich gut in DomainPro umsetzen, da es den Instanzen der Step-Container entspricht. Auch wäre es e.g. möglich, dem Dispatcher einen Namen (in Form einer Variable) zu geben, damit die Agent-Instanzen einfacher identifiziert werden können.

Eine Herausforderung bestand im Transformationstool in der Abbildung der Step-Hierachien. Hier wäre es optimal, wenn eine Instanzhierachie (serialisierte Elemente des Ausgangsmodells) aus Immutable-Objects erstellt worden wäre, um irrtümliche Änderungen während des Transformationsprozesses zu verhindern. Jede XPath-Abfrage kann auch in eine Abfrage in der Instanzhierachie umgewandelt werden. Aufgrund der Typsicherheit von Language Integrated Query (LINQ) kann das Arbeiten mit deserialisierten Objekten als eine gute Wahl gesehen werden.

6.3 Analyse von runningWorldRoot

Das Modell ist im Codebeispiel 6.5 abgebildet. Das Ergebnis der Transformation in Abbildung 6.1.

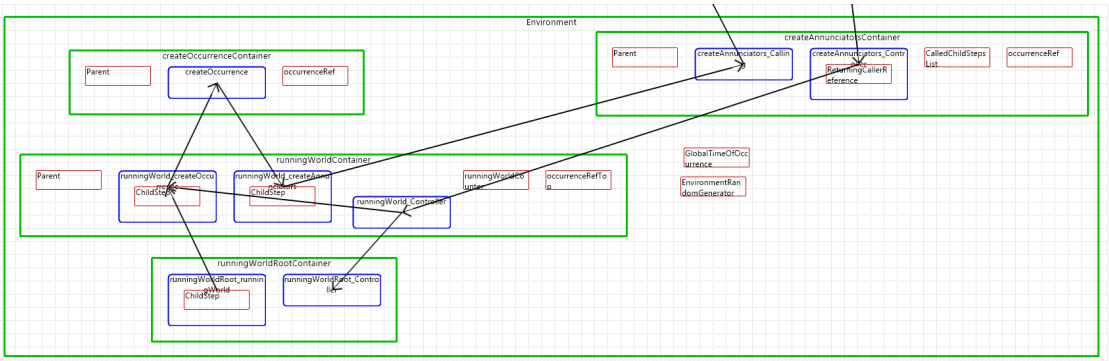

Abbildung 6.1: runningWorldRoot in DomainPro

6.3.1 Analyse

Aufgrund der begrenzten umgesetzten Features im Zuge der Diplomarbeit, konnte der Use-Case nicht zur Gänze umgesetzt werden. Das Erzeugen der unterschiedlichen Annunciator-Typen hätte eine 1:1 Kopie der Child-Steps erfordert. Das hätte eine große Verkomplizierung des Simulations-Modells zur Folge gehabt und wurde daher nicht gemacht. LittleJIL erlaubt es, Steps zu referenzieren[1]:

> Each step in a Little-JIL program is defined exactly once; however, it may be used multiple times. These uses are represented by references. A reference is represented with italicized text and without badges.

Dieses Feature kann für eine zukünftige Version als wichtig eingestuft werden, da dieser Use-Case vermutlich öfters vorkommt.

Betrachtet man die Abbildung 6.1, so erkennt man die hierarchische Abfolge nicht unmittelbar - vor allem, wenn Kardinalitäten angegeben wurden. Für einen Simulationsentwickler ist daher die Anordnung der Container für das Verständnis essentiell - was in der jetzigen Umsetzung noch nicht automatisiert gemacht wird (die aktuelle Version ordnet die Elemente von links nach rechts an, ohne Berücksichtigung der Hierarchie, was auch noch auf die vertikale Anordnung eine Auswirkung hätte).

Ebenfalls kann bei Child-Steps die Generierung der "virtuellen"Hilfsmethoden zum erschwerten Verständnis führen (wie in der Abbildung 6.1 *runningWorld_createAnnunciators*).

[1] `http://www.umass.edu/eei/EEI%20Website%20Articles/Little-JIL%201.5%20Language%20Report.pdf`

Algorithm 6.5: Details des Root-Steps *runningWorldRoot*

```
1  Step name: runningWorldRoot of type Sequential
2    Local Data: {
3      [ agent as Resource of hADLType cad.world.Environment]
4    }
5    Child Steps: {
6      Step name: runningWorld     of type Sequential
7      has Cardinality: 0..*
8      Local Data: {
9        [ occurrenceRefTop as Subtree of hADLType cad.world.Occurrence]
10     }
11     Child Steps: {
12       // ################################################
13       // Creates a "Small Occurrence" or "Big Occurrence"
14       // ################################################
15       Step name: createOccurrence of type Leaf
16       Local Data: {
17         [ occurrenceRef as ParameterOut of hADLType cad.world.Occurrence]
18       }
19       Local to Parent Data Bindings: {
20         [ occurrenceRef > occurrenceRefTop ]
21       }
22       Step name: createAnnunciators of type Parallel
23       // In case of "Small Occurrence" break at 1, in case of "Big Occurrence" >1 and <=5
24       has Cardinality: 0..5
25       Local Data: {
26         [ agent as Resource of hADLType cad.world.Environment]
27         [ occurrenceRef as ParameterInOut of hADLType cad.world.Occurrence]
28       }
29       Local to Parent Data Bindings: {
30         [ occurrenceRef < occurrenceRefTop ]
31       }
32       Child Steps: {
33         Step name: verbalAnnunciatorObserverAndReport of type Sequential
34         Local Data: {
35           [ agent as Resource of hADLType cad.world.Annunciator]
36           [ occurrenceRef as ParameterInOut of hADLType cad.world.Occurrence]
37         }
38         Local to Parent Data Bindings: {
39           [ occurrenceRef < occurrenceRef ]
40         }
41         Child Steps: {
42           Step name: observeOccurrence of type Leaf
43           Local Data: {
44             [ useObserveAction as Subtree of actionType cad.world.Annunciator.observe]
45             [ occurrenceRef as ParameterIn of hADLType cad.world.Occurrence]
46           }
47           Local to Parent Data Bindings: {
48             [ occurrenceRef < occurrenceRef ]
49           }
50           Step name: reportOccurrence of type Leaf
51           Local Data: {
52             [ providingInformation as Subtree of actionType cad.world.EmergencyCall.
                   provideInformationToEmergencyCallCenter]
53             [ occurrenceRef as ParameterIn of hADLType cad.world.Occurrence]
54           }
55           Local to Parent Data Bindings: {
56             [ occurrenceRef < occurrenceRef ]
57           }
58         }
59       }
60     }
61   }
```

Ebenfalls ist die bedingte Ausführung (wie in der Abbildung 6.1 *runningWorld_Controller*) nicht optimal, da die Bedingungen nicht sofort ersichtlich sind. Die Bedingungen sind in den Pfeilen hinterlegt. Der Entwickler kann sich daher nur ein Gesamtbild verschaffen, wenn er mehrere Elemente separat betrachtet. Es wäre daher wünschenswert, wenn das Aufrufen von ChildSteps und Bedienungen besser darstellbar wäre (Support in der DomainPro XML Struktur erforderlich).

Um die Zeit der Occurrences eindeutig zu machen, wurde die Variable *GlobalTimeOfOccurrence* eingeführt, welche automatisch hochzählt. Ebenfalls wurde die Variable *EnvironmentRandomGenerator* eingeführt, um Zufallszahlen erstellen zu können.

Codebeispiel 6.6 zeigt den manuell eingefügten Code von *createOccurrenceContainer* → *createOccurrence*. Es wurde mit einer 25%igen Wahrscheinlichkeit entschieden, ob es sich um ein Großschadensereignis handelt oder nicht.

Algorithm 6.6: Erstellen der Vorfälle

```
1  DomainProAnalyst.Instance.Report("Environment.createOccurrenceContainer.createOccurrence");
2
3  // Default agent instances - modify if needed
4  var agentInstanceEnvironment = createOccurrenceContainer.Environment;
5  var agentInstanceCommunicationGateway = createOccurrenceContainer.Environment.cad.
       CommunicationGatewayList[0];
6  var agentInstanceDispatcher = createOccurrenceContainer.Environment.cad.DispatcherList[0];
7  var agentInstanceAnnunciator = createOccurrenceContainer.Environment.cad.AnnunciatorList[0];
8
9  // MANUAL Code appended
10 // Always create a new instance
11 createOccurrenceContainer.occurrenceRef = (Occurrence)createOccurrenceContainer.Environment.cad.
       Create("Occurrence");
12
13 var randomValue = createOccurrenceContainer.Environment.EnvironmentRandomGenerator.Next(0,100);
14
15 if(randomValue < 25)
16 {
17   createOccurrenceContainer.occurrenceRef.IsBigOccurrence = true;
18 }
19
20 createOccurrenceContainer.occurrenceRef.TimeOfOccurrence = createOccurrenceContainer.Environment.
       GlobalTimeOfOccurrence++;
```

Der Code kann in der weiterverbreiteten Sprache C# geschrieben werden. Der Vorteil ist, dass Visual Studio für das Debugging verwendet werden kann (direktes Debugging des Simulationsablaufs nicht bzw. schwer möglich - im Falle der Exception kann jedoch gut analysiert werden, wo das Problem liegt). In jeder Methode wurde der Name für Debuggingzwecke angegeben. Es ist in DomainPro sonst nicht ersichtlich, wie der Ablauf der Simulation ist - was die Fehlersuche erschwert.

Der Transfer der Daten (Bindings) ist nicht direkt ersichtlich, wenn man Variablen selektiert (e.g. occurrenceRef). Stattdessen muss man in den eingehenden Pfeilen schauen, wo die Referenz gesetzt ist. Auch ob die Variable als eingehender oder ausgehender Parameter verwendet wird, ist nicht ersichtlich. DomainPro erlaubt es zwar, dass die Farben geändert werden können (Rand- und Füllfarbe), allerdings wäre eine einheitliche Methode wünschenswert.

6.4 Analyse des Annunciators

Als nächstes werden alle Schritte betrachtet, welche die Komponente Anunciator als Agenten haben. Das ist im Fallbeispiel der Step *verbalAnnunciatorObserverAndReport* und dessen Child-Steps. Abbildung 6.2 zeigt das Ergebnis der Transformation.

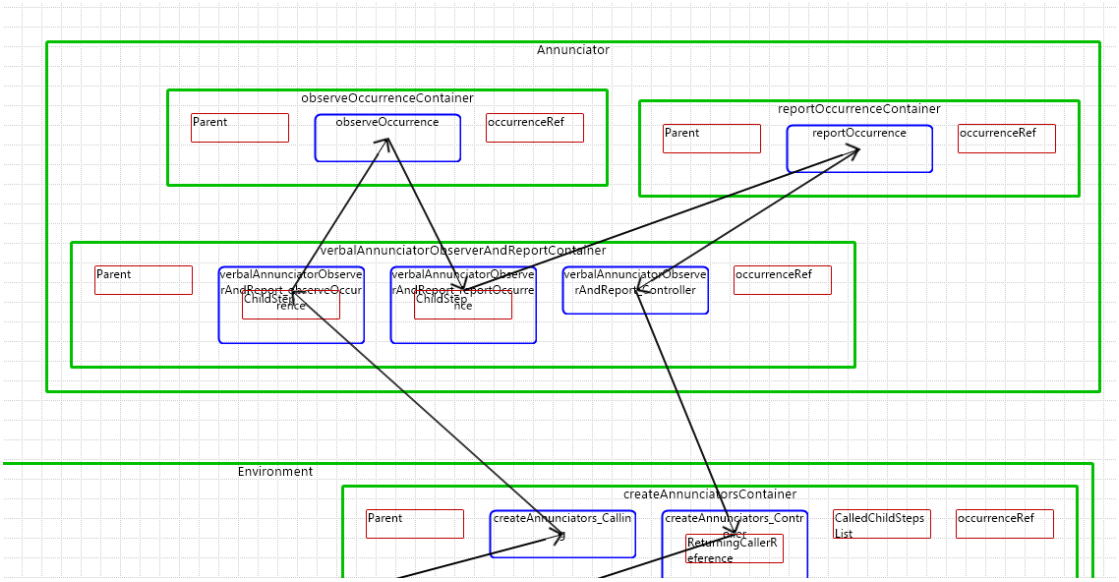

Abbildung 6.2: Annunciator Container

Wie man der Abbildung 6.2 entnehmen kann, wird *verbalAnnunciatorObserverAndReport* von *createOccurrence* aufgerufen. Im Pfeil (Transfer Eigenschaft) von *reportOccurrence* wirde die Referenz der Occurrence übergeben (Codebeispiel 6.7).

Algorithm 6.7: Setzen der Bindings

```
1   reportOccurrence.reportOccurrenceContainer.occurrenceRef =
        verbalAnnunciatorObserverAndReport_reportOccurrence.
        verbalAnnunciatorObserverAndReportContainer.occurrenceRef;
```

reportOccurrence muss den Einsatz melden. Dies passiert über die Indirektion des *EmergencyCall* und erfolgt manuell per Code (Codebeispiel 6.8).

Algorithm 6.8: Melden des Vorfalls

```
1  // MANUAL Code appended
2  var emergencyCallInstance = (Simulation.cadContext.EmergencyCall)reportOccurrenceContainer.
       Annunciator.cad.Create("EmergencyCall");
3  emergencyCallInstance.EmergencyCallToOccurrenceRef = reportOccurrenceContainer.occurrenceRef;
4  emergencyCallInstance.provideInformationToEmergencyCallCenter(false);
```

Der *EmergencyCall* wird als DataObjType umgesetzt (siehe Abbildung 6.3).

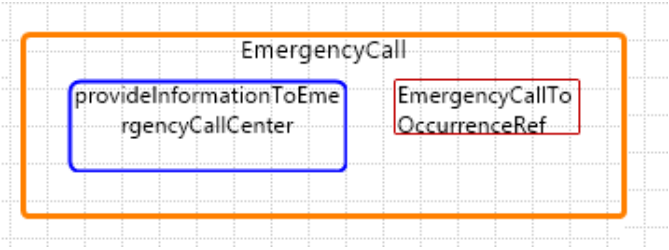

Abbildung 6.3: Emergency call data object

Die entscheidende Logik befindet sich in *provideInformationToEmergencyCallCenter* und ist im Codebeispiel 6.9 abgebildet.

Algorithm 6.9: Einreihen des Notrufs

```
1  // MANUAL Code appended
2  var communicationGateway = EmergencyCall.cad.CommunicationGatewayList[0];
3  communicationGateway.ReceivedEmergencyQueue.Enqueue(EmergencyCall);
```

Der *EmergencyCall* wird in die Queue *ReceivedEmergencyQueue* der ersten *CommunicationGateway* Agent-Instanz eingereiht (Codebeispiel 6.9).

6.4.1 Analyse

Wie man sieht, muss der Entwickler der Simulation einiges an Wissen mitbringen. DomainPro bietet keine Unterstützung für IntelliSense - d.h. der Entwickler muss sämtliche Nomenklaturen der Simulationsumgebung wissen. Ebenfalls muss er Namespaces manuell auflösen. Codebeispiel 6.10 zeigt die Zeile des letzten Beispiels nochmal.

Algorithm 6.10: Referenz des CommunicationGateway

```
1  var communicationGateway = EmergencyCall.cad.CommunicationGatewayList[0];
```

Der Entwickler muss hier die Referenzen bis zum gewünschten Agenten selbst auflösen und anschließend auch noch *<AgentName>List* dazu nutzen, um die erste Instanz zu bekommen. Eine interne DSL wäre hier wünschenswert (nur durch Änderung des DomainPro Analyzers möglich).

6.5 Analyse des CommunicationGateway

Das CommunicationGateway enthält eine Queue vom Typ *Queue<Simulation.cadContext.EmergencyCall>* (Abbildung 6.4). Das Gateway versucht, die Occurrence einem freien Dispatcher zuzuweisen.

Codebeispiel 6.11 zeigt den manuell eingefügten Code von receiveEmergency.

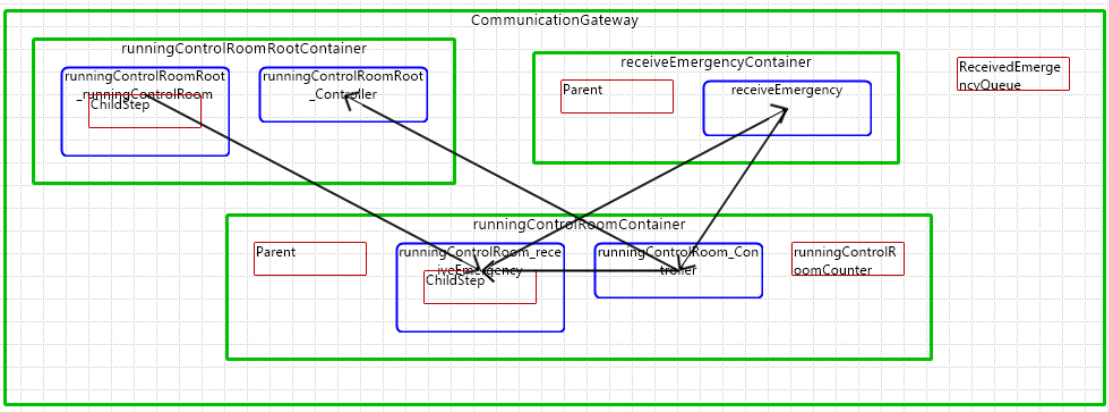

Abbildung 6.4: Communication Gateway container

Algorithm 6.11: Manuell eingefügter Code des Emergency Empfangs

```
1  if(receiveEmergencyContainer.CommunicationGateway.ReceivedEmergencyQueue.Count > 0)
2  {
3    foreach(var dispatcherInstance in receiveEmergencyContainer.CommunicationGateway.cad.
         DispatcherList)
4    {
5      if(dispatcherInstance.CurrentProcessingOccurrenceRef == null)
6      {
7        var emergencyInstance = receiveEmergencyContainer.CommunicationGateway.ReceivedEmergencyQueue
             .Dequeue();
8        dispatcherInstance.CurrentProcessingOccurrenceRef = emergencyInstance.
             EmergencyCallToOccurrenceRef;
9        break;
10     }
11   }
12 }
```

6.5.1 Analyse

Leider wurde in der Codegenerierung von DomainPro Analyzer der Namespace *System.Linq* nicht eingebunden. Daher ist die Nutzung von Extension Methoden nicht möglich. Schleifen und Abfragen können daher nicht vereinfacht werden. Der obige Code könnte mit LINQ wie im Codebeispiel 6.12 aussehen.

6.6 Analyse des Dispatcher

Der Dispatcher-Container hat die Aufgabe, die Aufgaben (Occurrences), welche er vom CommunicationGateway zugewiesen bekommen hat, zu verarbeiten (Abbildung 6.5).

Dazu muss die Referenz, welche vom CommunicationGateway gesetzt wurde, manuell in der Methode *runningDispatcher_processOccurrence* gesetzt werden (Codebeispiel 6.13).

Anschließend müssen alle anderen Disponenten gefragt werden, ob sie den Einsatz kennen. Dies würde aber nur durch einen großen Eingriff in die Container und Methode-Sequences

Algorithm 6.12: Vereinfachter Code mit LINQ

```
1  if(receiveEmergencyContainer.CommunicationGateway.ReceivedEmergencyQueue.Any())
2  {
3    var dispatcherInstance = receiveEmergencyContainer.CommunicationGateway.cad.DispatcherList.
         FirstOrDefault(di => di.CurrentProcessingOccurrenceRef == null);
4
5    if(dispatcherInstance != null)
6    {
7      var emergencyInstance = receiveEmergencyContainer.CommunicationGateway.ReceivedEmergencyQueue.
           Dequeue();
8      dispatcherInstance.CurrentProcessingOccurrenceRef = emergencyInstance.
           EmergencyCallToOccurrenceRef;
9    }
10 }
```

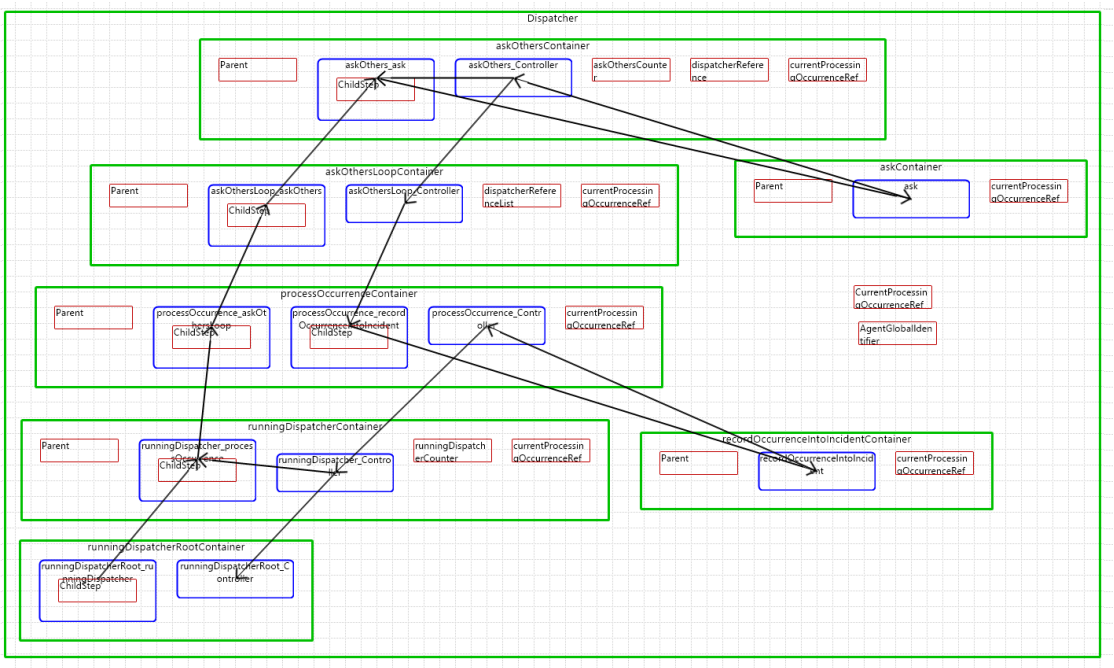

Abbildung 6.5: Dispatcher Container

möglich sein (siehe Analyse). Die Methode *askOthers_ask* führt die Befragung daher sequentiell aus (wie im Ausgangsmodell bestimmt) - siehe Codebeispiel 6.14.

Dabei muss die Agent-Instanz immer neu gesetzt werden. Es wird anschließend ein neuer Container (askContainer) auf dem Agenten erstellt und eine Methode aufgerufen. Die Bedingung, dass nur im Falle einer BigOccurrence gefragt wird, musste in die Methode *ask* gegeben werden (Codebeispiel 6.15).

6.6.1 Analyse

Conditionals sind nicht möglich. LittleJIL unterstützt diese aber:

Algorithm 6.13: Zuweisung der aktuell zu bearbeitenden Occurrence

```
1  // MANUAL Code appended
2  runningDispatcherContainer.currentProcessingOccurrenceRef = runningDispatcherContainer.Dispatcher.
       CurrentProcessingOccurrenceRef;
```

Algorithm 6.14: Das Befragen anderer Disponenten

```
1  if(askOthersContainer.askOthersCounter < askOthersContainer.Parent.dispatcherReferenceList.Count)
2  {
3    askOthersContainer.dispatcherReference = askOthersContainer.Parent.dispatcherReferenceList[
         askOthersContainer.askOthersCounter];
4  }
5  // Agent instances set per Transfer
6  agentInstanceDispatcher = askOthersContainer.dispatcherReference;
7
8  // Modify agentInstance if other Child should be called
9  ChildStep = (DispatcherContext.askContainer)agentInstanceDispatcher.Create("askContainer");
10 ChildStep.Parent = askOthersContainer;
```

Algorithm 6.15: Der Disponent wird nur im Falle einer BigOccurrence befragt

```
1  // MANUAL Code appended
2  if(askContainer.currentProcessingOccurrenceRef != null && askContainer.
       currentProcessingOccurrenceRef.IsBigOccurrence)
3  {
4    DomainProAnalyst.Instance.Report("Dispatcher " + askContainer.Dispatcher.AgentGlobalIdentifier +
         " was asked from " + askContainer.Parent.Dispatcher.AgentGlobalIdentifier + " for Occurrence
         .TimeOfOccurrence=" + askContainer.currentProcessingOccurrenceRef.TimeOfOccurrence + "
         Occurrence.IsBigOccurrence=" + askContainer.currentProcessingOccurrenceRef.IsBigOccurrence);
5  }
```

> Predicates provide a conditional mechanism to control the posting of sub-steps. A predicate appears as a parenthesized expression in the cardinality and are mutually exclusive with the agent and resource or artifact-controlled mechanisms. ... Predicates are evaluated after all of the input parameters are bound. If the predicate evaluates to true, the sub-step is posted, if false, the sub-step is not posted, and execution of the parent step continues as if the sub-step did not appear. For example, if the parent is a sequential step, the next step in the sequence is evaluated.

Dies hätte den Vorteil, dass die Dispatcher Instanz nicht erstellt werden müsste und erst recht spät entschieden wird, ob der Dispatcher überhaupt gefragt wird.

Ebenfalls wurden **parallele Child-Steps mit Agenten** nicht umgesetzt. Dadurch mussten die Disponenten sequentiell befragt werden - was im Regelfall nicht der Realität entspricht. Die unterschiedlichen Zeiten des Befragens kann der Entwickler einfach in die Duration der Methode eintragen.

6.7 Analyse von Incident und Occurrence

Beide Container sind vom Typ DataObjType (Abbildung 6.6).

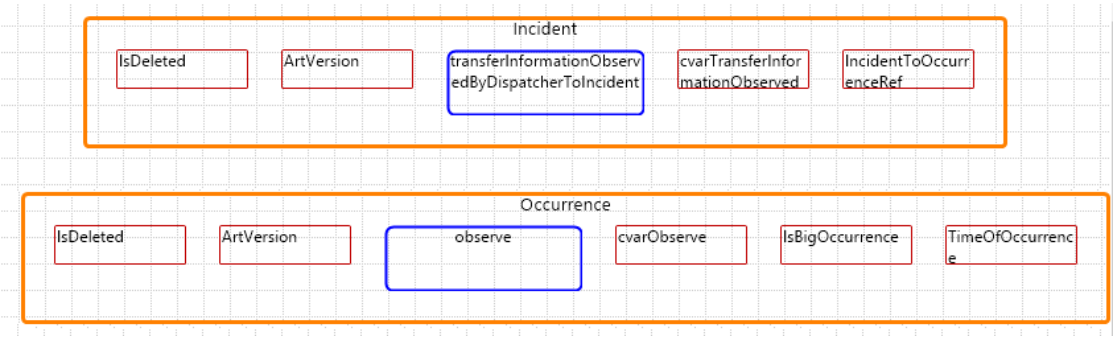

Abbildung 6.6: Dispatcher Container

Die Methode *observe* spielt dabei für die Berechnung der Dauer eine Rolle. Die Zeit (Method Duration) kann vom Entwickler frei oder zufällig gewählt werden.

Die Methode *transferInformationObservedByDispatcherToIncident* spiegelt das Melden des Ereignisses an das Incident Management System wieder. Codebeispiel 6.16 zeigt den Code der Methode.

Algorithm 6.16: Melden des Incident

```
1  // MANUAL Code appended
2  DomainProAnalyst.Instance.Report("An incident was reported! IsBigOccurrence=" + Incident.
       IncidentToOccurrenceRef.IsBigOccurrence + " TimeOfOccurrence=" + Incident.
       IncidentToOccurrenceRef.TimeOfOccurrence);
```

6.8 Manueller Code

An einigen Stellen war es erforderlich, manuellen Code einzufügen, um gewünschtes Verhalten zu erzeugen. Es mussten einige Variablen eingeführt werden:

- *ReceivedEmergencyQueue* in CommunicationGateway: Zum Empfang von Notrufen (Emergency Calls).
- *IsBigOccurrence* bzw. *TimeOfOccurrence* in Occurrence: Eigenschaften einer Occurrence.
- *GlobalTimeOfOccurrence* bzw. *EnvironmentRandomGenerator* in Environment: Die globale Zeit für Occurrences bzw. einen Random Generator für die Anzahl der Annunciators etc.

- *CurrentProcessingOccurrenceRef* bzw. *AgentGlobalIdentifier* in Dispatcher: Ersteres musste manuell eingefügt werden, da es von außen gesetzt wird (anderer Root-Prozess). Zweiteres erleichtert nur das Debugging, da jedem Dispatcher-Agenten eine eindeutige Id zugewiesen wird.

6.9 Simulation in DomainPro Analyst

Um die Simulation ausführen zu können, ist es erforderlich, die Agent-Instanzen anzugeben. Dabei wurden folgende Instanzen erzeugt:

- CommunicationGateway: 1
  - runningControlRoomRootContainer: 1
- Dispatcher: 3
  - RunningDispatcherRootContainer: 1
- Environment: 1
  - runningWorldRootContainer: 1
- Annunciator: 1 (diese Instanz wird zur Simulationszeit dynamisch erstellt - 1 Instanz sollte aber immer vorhanden sein, damit die Default-Agent Instanzen gesetzt werden können).

Ebenfalls muss angegeben werden, welche Methoden der Instanzen initial aufgerufen werden sollen:

- CommunicationGateway: *runningControlRoomRoot_runningControlRoom*
- Dispatcher: *runningDispatcherRoot_runningDispatcher*
- Environment: *runningWorldRoot_runningWorld*

DomainPro erlaubt es, die Anzahl der Aufrufe für Methoden zu protokollieren. Es wurden folgende Methoden protokolliert:

- Dispatcher: ask
- Annunciator: reportOccurrence
- Incident: transformInformationObservedByDispatcherToIncident

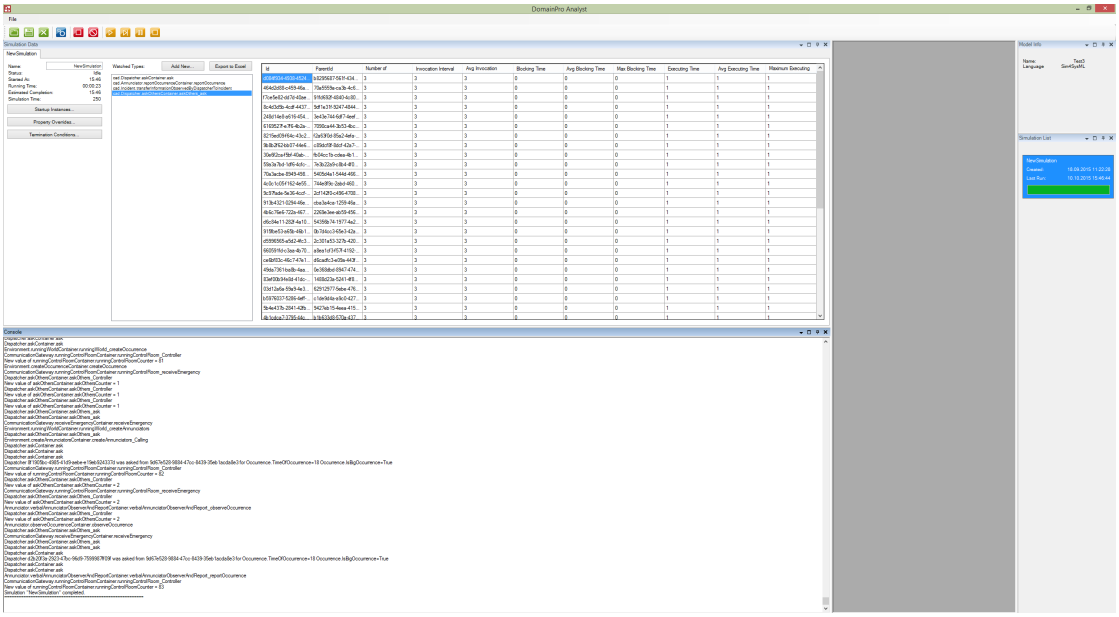

Abbildung 6.7: DomainProAnalyst

Wie man der Abbildung 6.7 entnehmen kann, ist die DomainPro Analysefunktion für die momentane Umsetzung nicht verwendbar. Das hat den Grund, dass Container jedes Mal neu initialisiert werden. Es kann daher e.g. nicht festgestellt werden, wie oft ein Dispatcher einen anderen gefragt hat. Zu diesem Zweck wurde ein Debug-Output in die Ask-Methode eingefügt (Codebeispiel 6.17).

Algorithm 6.17: Zusätzliche Debug Funktionalität

```
1  // MANUAL Code appended
2  if(askContainer.currentProcessingOccurrenceRef != null && askContainer.
       currentProcessingOccurrenceRef.IsBigOccurrence)
3  {
4    DomainProAnalyst.Instance.Report("Dispatcher " + askContainer.Dispatcher.AgentGlobalIdentifier +
       " was asked from " + askContainer.Parent.Dispatcher.AgentGlobalIdentifier + " for Occurrence
       .TimeOfOccurrence=" + askContainer.currentProcessingOccurrenceRef.TimeOfOccurrence + "
       Occurrence.IsBigOccurrence=" + askContainer.currentProcessingOccurrenceRef.IsBigOccurrence);
5  }
```

Lässt man die Simulation für 250 Zyklen laufen, so erhält man einen ähnlichen Output wie in Abbildung 6.18.

Der Hintergrund, warum Occurrence 3 (TimeOfOccurrence=3) 6 Mal vorkommt ist, dass 2 Notrufer den Vorfall gemeldet haben. Dass der Dispatcher auch sich selbst fragt, kann als Schwäche des Modells gesehen werden. Abbildung 6.19 zeigt, wie der Incident angelegt wird.

Zu erkennen ist (Codebeispiel 6.19), dass auch mehrere Occurrences mehrfach gemeldet wurden (e.g. TimeOfOccurrence=3). Es ist Aufgabe des Einsatzleitsystems, diese

Algorithm 6.18: Beispiel Output der Simulation

```
1  Dispatcher 3bb19665-8dfd-4e24-bd00-62d992dca602 was asked from 3bb19665-8dfd-4e24-bd00-62d992dca602
       for Occurrence.TimeOfOccurrence=3 Occurrence.IsBigOccurrence=True
2  Dispatcher f69bf805-cf5d-4927-9033-0d9931358e6b was asked from 3bb19665-8dfd-4e24-bd00-62d992dca602
       for Occurrence.TimeOfOccurrence=3 Occurrence.IsBigOccurrence=True
3  Dispatcher 1982126f-df56-4e57-b023-861eaf630337 was asked from 3bb19665-8dfd-4e24-bd00-62d992dca602
       for Occurrence.TimeOfOccurrence=3 Occurrence.IsBigOccurrence=True
4  Dispatcher 3bb19665-8dfd-4e24-bd00-62d992dca602 was asked from 3bb19665-8dfd-4e24-bd00-62d992dca602
       for Occurrence.TimeOfOccurrence=5 Occurrence.IsBigOccurrence=True
5  Dispatcher 3bb19665-8dfd-4e24-bd00-62d992dca602 was asked from f69bf805-cf5d-4927-9033-0d9931358e6b
       for Occurrence.TimeOfOccurrence=3 Occurrence.IsBigOccurrence=True
6  Dispatcher f69bf805-cf5d-4927-9033-0d9931358e6b was asked from f69bf805-cf5d-4927-9033-0d9931358e6b
       for Occurrence.TimeOfOccurrence=3 Occurrence.IsBigOccurrence=True
7  Dispatcher f69bf805-cf5d-4927-9033-0d9931358e6b was asked from 3bb19665-8dfd-4e24-bd00-62d992dca602
       for Occurrence.TimeOfOccurrence=5 Occurrence.IsBigOccurrence=True
8  Dispatcher 1982126f-df56-4e57-b023-861eaf630337 was asked from 3bb19665-8dfd-4e24-bd00-62d992dca602
       for Occurrence.TimeOfOccurrence=5 Occurrence.IsBigOccurrence=True
9  Dispatcher 1982126f-df56-4e57-b023-861eaf630337 was asked from f69bf805-cf5d-4927-9033-0d9931358e6b
       for Occurrence.TimeOfOccurrence=3 Occurrence.IsBigOccurrence=True
10 Dispatcher 3bb19665-8dfd-4e24-bd00-62d992dca602 was asked from 3bb19665-8dfd-4e24-bd00-62d992dca602
       for Occurrence.TimeOfOccurrence=5 Occurrence.IsBigOccurrence=True
11 Dispatcher f69bf805-cf5d-4927-9033-0d9931358e6b was asked from 3bb19665-8dfd-4e24-bd00-62d992dca602
       for Occurrence.TimeOfOccurrence=5 Occurrence.IsBigOccurrence=True
12 ...
```

Algorithm 6.19: Beispiel Output der Simulation

```
1  An incident was reported! IsBigOccurrence=False TimeOfOccurrence=0
2  An incident was reported! IsBigOccurrence=False TimeOfOccurrence=1
3  An incident was reported! IsBigOccurrence=False TimeOfOccurrence=2
4  An incident was reported! IsBigOccurrence=True TimeOfOccurrence=3
5  An incident was reported! IsBigOccurrence=True TimeOfOccurrence=5
6  An incident was reported! IsBigOccurrence=False TimeOfOccurrence=4
7  An incident was reported! IsBigOccurrence=True TimeOfOccurrence=3
8  An incident was reported! IsBigOccurrence=True TimeOfOccurrence=5
9  An incident was reported! IsBigOccurrence=True TimeOfOccurrence=5
10 An incident was reported! IsBigOccurrence=True TimeOfOccurrence=5
11 An incident was reported! IsBigOccurrence=False TimeOfOccurrence=6
12 An incident was reported! IsBigOccurrence=False TimeOfOccurrence=7
13 ...
```

Mehrfacheinträge zu mergen. Dies wurde im Zuge der Arbeit nicht mehr umgesetzt, da es keine neuen Erkentnisse bzgl. der Analyse gebracht hätte.

6.9.1 Interpretation der Simulationsergebnisse

Im untersuchten Szenario der verbalen Kommunikation zwischen Disponenten konnte eine hohe Kommunikation zwischen Disponenten festgestellt werden. Gemessen wurde die Anzahl der Anfragen von anderen Disponenten - ohne Berücksichtigung der Dauer (diese war mit dem Default-Wert 1 angegeben). Es wurde mit 3 Disponenten simuliert.

Die Simulation wurde für 250 Einheiten ausgeführt. Dabei wurden folgende Daten gemessen: Durch Instrumentieren der Variable *Environment.GlobalTimeOfOccurrence* konnte die Anzahl der Einsätze ermittelt werden. *GlobalTimeOfOccurrence* ist für jede Occurrence eindeutig und wurde pro Occurrence um 1 erhöht. Die Anzahl der Vorfälle betrug 23.

Zum Zeitpunkt des Beendens (250 Simulationszyklen) befanden sich noch 5 Emergency-Calls im CommunicationGateway. Dem Incident-Management System wurden 16 Einsätze gemeldet (bedingt durch die beschränkte Simulationszeit).

Interessant ist, dass Disponenten 33 Mal kommuniziert haben - und das für 5 Einsätze. Dies wurde durch Auswertung des Debug Outputs festgestellt (Codebeispiel 6.20). Auch werden Dispatcher mehrfach für den gleichen Vorfall befragt, wenn dieser von mehreren Notrufern gemeldet wurde. Das kann mit einer Schwäche im Modell begründet werden: Disponenten, welche für einen Vorfall mehrere Notrufe bekommen, müssen nicht nochmal alle anderen Disponenten fragen. Daher müsste sich der Disponent merken, welche Einsätze er schon kennt.

Algorithm 6.20: Disponten werden doppelt gefragt - eine Schwachstelle im Modell

```
1  ...
2  Dispatcher 37fc3633-79fb-454f-b639-2bb06438f52c was asked from 670451c4-b04a-4967-9eeb-38f099429e72
       for Occurrence.TimeOfOccurrence=9 Occurrence.IsBigOccurrence=True
3  Dispatcher d549f577-b502-4b99-acd8-0df969c98539 was asked from 670451c4-b04a-4967-9eeb-38f099429e72
       for Occurrence.TimeOfOccurrence=9 Occurrence.IsBigOccurrence=True
4  ...
```

6.10 Aufwandsanalyse und Reflektion

Eine Aufwandsanalyse im Zuge des Fallbeispiels soll den Aufwand für den Simulationsentwickler nochmals zeigen. Folgende Statistik gibt einen Überblick:

- Das Ausgangsmodell (LittleJIL) hat 17 Steps und hat 5 Kardinalitäten.
- Es wurden 32 Zeilen manueller Code eingefügt.
- Es wurden 405 Codezeilen generiert. D.h., es mussten ca. 8% der generierten Codezeilen erweitert oder geändert werden.
- Es mussten 7 Variablen eingefügt werden. Diese konnten nicht im Modell (hADL bzw. LittleJIL) abgebildet werden.
- Es wurden 32 Methode-Sequences generiert. Kardinalitäten und die daraus folgenden Schleifen erzeugen meist mehrere Pfeile. Die Anzahl der Pfeile hält sich also in Grenzen (bei 5 Kardinalitäten im Beispiel).
- Es wurden 21 Komponenten angelegt (Step Container und Agenten). Davon sind 3 Container für Agenten. Die restlichen Container wurden für Steps und deren Child-Steps erstellt. Die Anzahl der Container ist allerdings nicht verwunderlich, da sie die Anzahl der Steps und Agenten wiederspiegeln. Ein zusätzlicher Top-Level Container fasst alles zusammen.

- Es wurden 32 Methoden angelegt - die meisten Methoden sind "virtuell mussten also angelegt werden, um das Pattern der Step-Hierachie mit Rückkehrmethode umzusetzen. Die hohe Anzahl der Methoden lässt sich damit begründen, dass für Step-Hierachien immer "virtuelle"Methoden erzeugt werden mussten.
- Es wurden 62 Variablen angelegt. 14 davon waren Parent-Variablen (Zugriff auf den Parent-Step) und 13 Variablen für den Zugriff auf den Child-Step. Der Simulationsentwickler braucht diese womöglich gar nicht - allerdings sind sie in den meisten Fällen sehr hilfreich. Andere Variablen wurden aufgrund der Bindings erstellt.

Der Vorteil der automatischen Generierung liegt klar in der Konsistenz. Die Umsetzung anhand von Patterns ist die Stärke dieser Strategie - ein Auszug:

- Die Benennung der e.g. Container, Methoden oder Agenten ist immer konsistent. E.g. wird die Benennung der "virtuellen"Methoden für Step und deren Child-Steps konsistent angewandt.
- Die Benutzung eines Artifacts (und dessen Eigenschaften) ist für Simulationsentwickler immer gleich. Muss ein Simulationsentwickler eine Simulation eines anderen Entwickler warten, so wird er gewisse Grundstrukturen immer wieder finden.

Es stellt sich in vielen Bereichen der Transformation die Frage, ob man ein einfacheres Output-Model hätte, wenn man die Transformation per Hand gemacht hätte. Dies ist durchaus möglich - allerdings überwiegen - aus Sicht des Autors - auf jeden Fall die aufgezählten Vorteile. Allem voran die Konsistenz im Zielmodell. Vereinfachungen im Zielmodell sollten daher im Transformationstool beschrieben werden, um die Vorteile nicht zu verlieren.

KAPITEL 7

Future Work

Das letzte Kapitel ist eine Reflektion über etwaige zukünftige Erweiterungen. Es wird dabei nur das Transformationstool betrachtet und nicht das Ausgangsmodell.

7.1 Fehlende Features

Zu Beginn der Arbeit wurde ein Featureumfang festgelegt, welcher im Zuge der Arbeit umgesetzt wurde. Durch die Umsetzung eines Fallbeispiels konnten eine Priorisierung weiterer Features bestimmt werden.

7.1.1 Bedingte Ausführung

Dieses Feature ist sehr wichtig um e.g. einen Choice (Auswahl) zwischen zwei Sub-Steps zu haben. Dass die Ausführung von Child-Steps an Bedingungen geknüpft ist, ist ein häufiger Use-Case. Die Umsetzung würde bei sequentiellen Child-Steps über Pfeile funktionieren, bei parallelen wäre eine Generierung von entsprechendem Code erforderlich, welcher die Bedingung umsetzt.

Algorithm 7.1: Beispiele von bedingten Ausführungen

```
1  Step name: <name of step> of type Leaf
2  ...
3  when expression true: {someExpression == true}
```

Die Umsetzung des parallelen Falls ist im Codebeispiel 7.1 zu sehen.

7.1.2 Parallel Agents

Das Anwenden von Agenten aus einer Ressourceliste in Form eines parallelen Aufrufs kommt in alltäglichen kollaborativen Szenarien häufig vor. Die Umsetzung würde in

Algorithm 7.2: Parallele Agenten

```
1  foreach(var ... in ...)
2  {
3    if(someExpression == true)
4    {
5      // Call childs
6    }
7  }
```

diesem Fall wieder durch entsprechende Codegenerierung passieren. Codebeispiel 7.3 zeigt eine mögliche Implementierung.

Algorithm 7.3: Parallele Agenten

```
1  foreach(var agentInstance in agentInstanceList)
2  {
3    var containerInstance = (Context.SomeContainer)agentInstance.Create("SomeContainer");
4    ...
```

7.1.3 Referenzieren von anderen Steps

Oft unterscheiden sich Step-Hierachen nur gering, sodass eine Wiederverwendung von Steps möglich ist. Durch Referenzierung (Codebeispiel 7.4) anderer Steps erspart man sich mehrfache Wartung des Simulations-Ziel-Modells. Die Referenzierung würde über entsprechende Aufrufe passieren: im sequentiellen Fall durch Pfeile und im parallelen Fall durch Aufrufe.

Algorithm 7.4: Beispiele von bedingten Ausführungen

```
1  Step name: <name of step> of type TaskReference Local to Parent Data Bindings: {
2          [ myTaskReference < myReferencesTask ]
3  }
```

7.2 Verbesserungen im Transformationstool

Einer der wichtigsten Verbesserungen wäre eine gute Fehlerbeschreibung, falls es e.g. zu Mehrdeutigen oder Problemen bei der Interpretation des Ausgangsmodells kommt. Derzeit sind die Fehler in einem sehr technischen Kontext gehalten. Auch erfolgt keine semantische Überprüfung beim Export in Eclipse, sodass das Ausgangsmodell nicht zwangsläufig Sinn machen muss.

7.2.1 Vereinfachungen

Teile des Zielmodells weisen unter Umständen eine größere Komplexität auf, als erwünscht. Dies ist damit zu begründen, dass das Transformationstool die weiteren Absichten des

Entwicklers nicht kennt und daher nur "Best-EffortËrgebnisse liefern kann.

Daher wäre es wünschenswert, wenn der Entwickler im Ausgangsmodell "Hintsë.g. in Form von Kommentaren im hADL und LittleJIL Modell hinzufügen kann. Diese "Hints"könnten dann zur Vereinfachung des Zielmodells genutzt werden. Dabei könnte e.g. die Anzahl der "virtuellen"Methoden reduziert werden, wenn der Simulationsentwickler diese für gewisse Schritte nicht benötigt. Auch könnten unnötige Variablen entfernt werden. Codebeispiel 7.5 zeigt eine mögliche Implementierung.

Algorithm 7.5: Beispiele von Hints

```
1  // @GenerateVirtualMethods(false)
2  Step name: aSequentialStepWithLotsOfChildSteps of type Sequential
```

Codebeispiel 7.5 hätte zur Folge, dass alle Child-Steps direkt verbunden werden. Des Weiteren könnte man die Steuerung der generierten Variablen ebenfalls steuern (Codebeispiel 7.6).

Algorithm 7.6: Beispiele von Hints

```
1  // @GenerateParentVariables(false)
2  // @GenerateChildVariables(false)
3  Step name: aSequentialStepWithLotsOfChildSteps of type Sequential
```

7.2.2 Manual Code Injezierung verbessern

Auch das Einfügen von Manual Code gestaltet sich noch schwierig, da der Entwickler e.g. Container und Methode angeben muss. Das hat den Grund, weil bei jeder Generierung des Zielmodells neue GUIDs erzeugt werden und der vorhandene Code nicht korrekt eingefügt werden kann. Hier wäre auch eine Lösung wünschenswert, um die Arbeit des Entwicklers zu erleichtern.

7.2.3 Konsistenzüberprüfungen

Das Input-Modell muss nicht zwangsläufig semantisch korrekt sein. An einigen Stellen im Transformationstool kommt es unter Umständen zu einem nicht wohldefinierten Verhalten, da mehrere Lösungen aufgrund eines falschen Input-Modells möglich sind. An einigen Stellen wurde bereits eine Überprüfung eingefügt (Codebeispiel 7.7).

7.2.4 Domain Specific Simulation Language

Domain Specific Languages unterscheidet man zwischen:

- internal DSL: Nutzen die Wirtssprache

Algorithm 7.7: Feststellen des Agenten mit Konsistenüberprüfung

```
1  private static string GetAgentFromAncestor(XElement stepElement)
2  {
3    ...
4    if (parentElementWithAgent != null)
5    {
6      agentType = GetAgentFromStep(parentElementWithAgent);
7
8      if (string.IsNullOrWhiteSpace(agentType))
9      {
10       throw new Exception("a step must have a agent");
11     }
12   }
13         ...
14 }
```

- external DSL: Von Grund auf neu definiert und muss erst in eine andere Sprache umgewandelt werden

Es wäre wünschenswert, dass das Simulationstool eine DSL aufweist. Für den Entwickler ist es oft schwer, an Instanzen oder Details zu kommen ohne den generierten Code zu durchforsten oder per try-compile-error zu probieren.

7.2.5 Sämtliche XML Teile entfernen

Im Transformationstool sollten in einem Refactoring sämtliche XML Operationen entfernt werden. Die Komplexität sollte in das Parsen des Input-Models geschoben werden. Folgende Änderungen sind erforderlich:

- Elemente immutable machen, um irrtümliches Ändern während der Transformation zu verhindern.
- Elemente vergleichbar machen.
- Hilfsoperationen in die Elemente einbauen.

Ein Aufruf wie im Codebeispiel 7.8 wäre eine große Hilfe.

Algorithm 7.8: Feststellen des Agenten mit Konsistenüberprüfung

```
1  // Gets the parent step
2  currentStep.GetParentStep();
3
4  // Gets the agent of the step
5  currentStep.GetAgent();
```

An dieser Stelle sei auch noch das generische und teils komplexe Output-Modell (getrieben durch DomainPro) erwähnt. Hier wäre eine Abstraktion in den Transformationsalgorithmen hilfreich gewesen. Das betrifft e.g. "virtuelle"Methoden.

7.3 Ergänzende Betrachtung von DomainPro

Die Betrachtung geht über den Schwerpunkt der Diplomarbeit hinaus und betrachtet das Tool DomainPro, mit dem das Ziel-Modell weiter verarbeitet wird. Da Tool und Modell allerdings eng miteinander gekoppelt sind, können Einschränkungen im Tool auch als Einschränkungen im Modell gesehen werden.

7.3.1 Debugging

Der erste auffällige Punkt ist, dass sich das Debugging der Simulation äußerst schwierig gestaltet. Die Abfolge in der Simulation konnte nur durch Einfügen von Output verfolgt werden - auch hier wäre eine grafische Step-by-Step Verfolgung wünschenswert. Ebenfalls wären Breakpoints sehr hilfreich.

7.3.2 Usability

Auch das Fehlen von Auto-Complete (e.g. IntelliSense) erschwert das Schreiben des Glue-Codes. Durch das Fehlen einer DSL ist das Schreiben von Glue-Code sehr komplex.

7.3.3 Struktur erhalten

Die Struktur von LittleJIL ist im Simulationstool leider verloren gegangen. Abbildung 7.1 zeigt ein Beispiel aus der Dokumentation von LittleJIL[1]. Eine entsprechende grafische Repräsentation in DomainPro wäre wünschenswert.

7.3.4 Bessere Abstraktion

LittleJIL verfügt über domainspezifische Elemente, um Prozesshierarchien zu beschreiben. DomainPro verfügt allerdings nur über sehr generische Modell-Bausteine - die einfache Struktur geht daher verloren. Da der Entwickler allerdings auf Änderungen im Ziel-Modell angewiesen ist, erschwert dies unnötig das Verständnis.

[1] http://www.umass.edu/eei/EEI%20Website%20Articles/Little-JIL%201.5%20Language%20Report.pdf

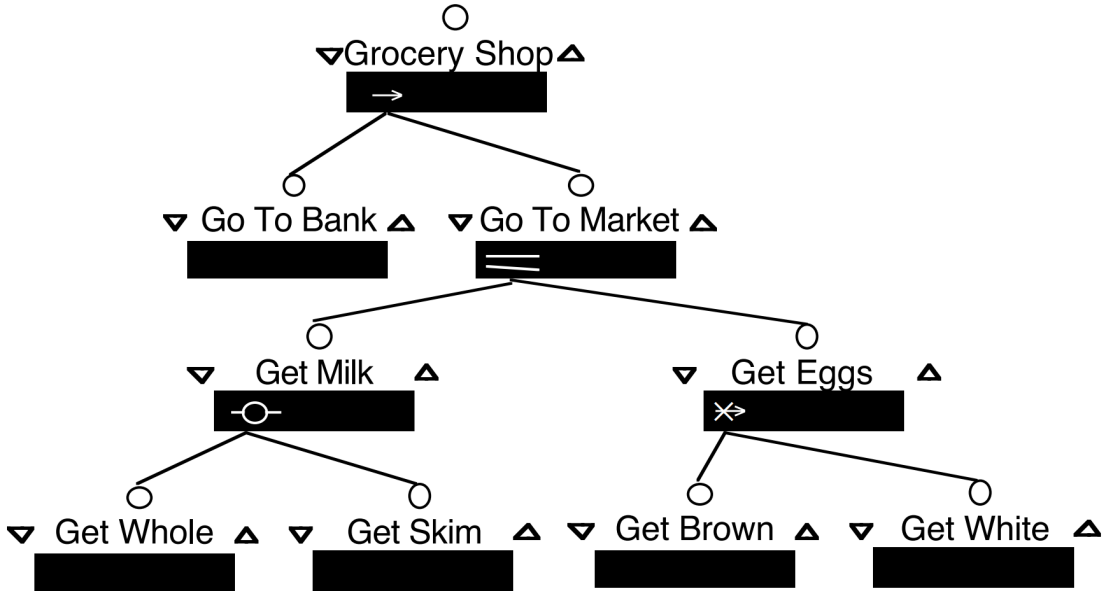

Abbildung 7.1: LittleJIL Darstellung

Abbildungsverzeichnis

Tabellenverzeichnis

List of Algorithms

Akronyme

AADL Architecture Analysis & Design Language. 15

ADL Architecture Description Language. 6, 15, 17

AMPDS Advanced Medical Priority Dispatch System. 39

BDD Behavior Driven Development. 2, 7

BDUF Big Design Up Front. 7

BPEL Business Process Execution Language. 19

BPMN Business Process Model and Notation. 19

CAD Computer Aided Dispatch System. 4, 53

CSP Communicating Sequential Processes. 15

hADL Human Architecture Description Language. vii, 7

LAN Local Area Network. 2

MBSD Model Based Software Development. 7, 11

MDE Model-Driven Engineering. 12

MDT Mobile Data Terminal. 25

OTS Off-The-Shelf. 14

PSAP Public-Safety Answering Point. 30

RPC Remote Procedure Call. 16

RTES Real-Time Embedded Systems. 22

SOA Service Orientierte Architektur. 19

SUT Softare-Under-Test. 22, 23

VCS Voice Communication System. 4

WAN Wide Area Network. 2, 55

XML Extensible Markup Language. 17

Literaturverzeichnis

[ABK10] Pekka Abrahamsson, Muhammad Ali Babar, and Philippe Kruchten. Agility and architecture: Can they coexist? *Software, IEEE*, 27(2):16–22, 2010.

[Ban98] Jerry Banks. Principles of simulation. *Handbook of simulation: Principles, methodology, advances, applications, and practice*, pages 3–30, 1998.

[BCW12] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. Model-driven software engineering in practice. *Synthesis Lectures on Software Engineering*, 1(1):1–182, 2012.

[BD87] George EP Box and Norman Richard Draper. *Empirical model-building and response surfaces*, volume 424. Wiley New York, 1987.

[CLM+00] Aaron G Cass, AS Lerner, Eric K McCall, Leon J Osterweil, Stanley M Sutton Jr, and Alexander Wise. Little-jil/juliette: a process definition language and interpreter. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 754–757. IEEE, 2000.

[Cro12] Adam Crowe. *Disasters 2.0: The application of social media systems for modern emergency management*. CRC press, 2012.

[DDO14] Christoph Dorn, Schahram Dustdar, and Leon J Osterweil. Specifying flexible human behavior in interaction-intensive process environments. In *Business Process Management*, pages 366–373. Springer, 2014.

[DEM12] Christoph Dorn, George Edwards, and Nenad Medvidovic. Analyzing design tradeoffs in large-scale socio-technical systems through simulation of dynamic collaboration patterns. In *On the Move to Meaningful Internet Systems: OTM 2012*, pages 362–379. Springer, 2012.

[DHT05] Eric M Dashofy, André van der Hoek, and Richard N Taylor. A comprehensive approach for the development of modular software architecture description languages. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(2):199–245, 2005.

[DPS00] Julie Dugdale, J Pavard, and B Soubie. A pragmatic development of a computer simulation of an emergency call center. *Designing Cooperative Systems: The Use of Theories and Models*, pages 241–256, 2000.

[DT12] Christoph Dorn and Richard N Taylor. Architecture-driven modeling of adaptive collaboration structures in large-scale social web applications. In *Web Information Systems Engineering-WISE 2012*, pages 143–156. Springer, 2012.

[DT13] Christoph Dorn and Richard N Taylor. Coupling software architecture and human architecture for collaboration-aware system adaptation. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 53–62. IEEE Press, 2013.

[DT14] Christoph Dorn and Richard N Taylor. Analyzing runtime adaptability of collaboration patterns. *Concurrency and Computation: Practice and Experience*, 2014.

[DTD12] Christoph Dorn, Richard N Taylor, and Schahram Dustdar. Flexible social workflows: Collaborations as human architecture. *IEEE Internet Computing*, (2):72–77, 2012.

[Fai10] George Fairbanks. *Just enough software architecture: a risk-driven approach*. Marshall & Brainerd, 2010.

[FG12] Peter H Feiler and David P Gluch. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Addison-Wesley, 2012.

[FR07] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In *2007 Future of Software Engineering*, pages 37–54. IEEE Computer Society, 2007.

[IAB13] Muhammad Zohaib Iqbal, Andrea Arcuri, and Lionel Briand. Environment modeling and simulation for automated testing of soft real-time embedded software. *Software & Systems Modeling*, 14(1):483–524, 2013.

[KSS+08] J Kim, Wonsang Song, Henning Schulzrinne, Anna Zacchi, Anupam Jain, Harshavardhan Chenji, Chris Magnussen, Chris Norton, Walt Magnussen, Ian Schworer, et al. The next generation 9-1-1 proof-of-concept system. *ACM SIGCOMM Demo*, 2008.

[LRS99] Neal Lesh, Charles Rich, and Candace L Sidner. *Using plan recognition in human-computer collaboration*. Springer, 1999.

[MC94] Thomas W Malone and Kevin Crowston. The interdisciplinary study of coordination. *ACM Computing Surveys (CSUR)*, 26(1):87–119, 1994.

[MOT97] Nenad Medvidovic, Peyman Oreizy, and Richard N Taylor. Reuse of off-the-shelf components in c2-style architectures. In *Proceedings of the 19th international conference on Software engineering*, pages 692–700. ACM, 1997.

[MQR95] Mark Moriconi, Xiaolei Qian, and Robert A Riemenschneider. Correct architecture refinement. *Software Engineering, IEEE Transactions on*, 21(4):356–372, 1995.

[MT00] Nenad Medvidovic and Richard N Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering, IEEE Transactions on*, 26(1):70–93, 2000.

[Nie05] Rainer Niekamp. Software component architecture. In *Gestión de Congresos-CIMNE/Institute for Scientific Computing, TU Braunschweig*, page 4, 2005.

[SCS14] Lachlan Standfield, Tracy Comans, and Paul Scuffham. Markov modeling and discrete event simulation in health care: a systematic comparison. *International journal of technology assessment in health care*, 30(02):165–172, 2014.

[SK03] Shane Sendall and Wojtek Kozaczynski. Model transformation the heart and soul of model-driven software development. Technical report, 2003.

[TMA+96] Richard N Taylor, Nenad Medvidovic, Kenneth M Anderson, E James Whitehead Jr, Jason E Robbins, Kari A Nies, Peyman Oreizy, and Deborah L Dubrow. A component-and message-based architectural style for gui software. *Software Engineering, IEEE Transactions on*, 22(6):390–406, 1996.

[UL10] Mark Utting and Bruno Legeard. *Practical model-based testing: a tools approach*. Morgan Kaufmann, 2010.

[WH05] Eoin Woods and Rich Hilliard. Architecture description languages in practice session report. In *null*, pages 243–246. IEEE, 2005.

[ZT13] Yongjie Zheng and Richard N Taylor. A classification and rationalization of model-based software development. *Software & Systems Modeling*, 12(4):669–678, 2013.