TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät Informatik Institut für Software- und Multimediatechnik, Professur für Softwaretechnologie

Bachelor-Arbeit

Verteilte Mensch-Roboter-Systeme: Ein Modellgetriebener Entwicklungsansatz

Karl Kegel
Geboren am: 06.04.1998 in Ebersbach Sa.
Matrikelnummer: 4604672

zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

Betreuer
Dr. rer. nat. Marvin Triebel
Dr.-Ing. Birgit Demuth
Betreuender Hochschullehrer
Prof. Dr. rer. nat habil. Uwe Aßmann

Eingereicht am: 01.08.2019

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit mit dem Titel *Verteilte Mensch-Roboter-Systeme: Ein Modellgetriebener Entwicklungsansatz* selbstständig und ohne unzulässige Hilfe Dritter verfasst habe. Es wurden keine anderen als die in der Arbeit angegebenen Hilfsmittel und Quellen benutzt. Die wörtlichen und sinngemäß übernommenen Zitate habe ich als solche kenntlich gemacht. Es waren keine weiteren Personen an der geistigen Herstellung der vorliegenden Arbeit beteiligt. Mir ist bekannt, dass die Nichteinhaltung dieser Erklärung zum nachträglichen Entzug des Hochschulabschlusses führen kann.

Dresden, 01.08.2019

Karl Kegel

Zusammenfassung

Durch die stets fortschreitenden technischen Möglichkeiten im Bereich des Industrial Internet of Things, sind verteilte Mensch-Roboter-Systeme zunehmend im Fokus von Forschung und Anwendung. Bei der Entwicklung von verteilten und interaktiven Systemen gibt es jedoch eine Vielzahl von Problemen und Herausforderungen. Bedingt durch die Komplexität solcher Systeme und damit einhergehender Technologien, ist eine weitreichende Expertise der beteiligten Entwickler nötig. Ebenfalls ist die klassische Entwicklung von verteilten Systemen auf Grund der Vielzahl der zu Grunde liegenden Teilanwendungen sehr komplex und fehleranfällig in Hinsicht auf Interoperabilität. Um diese Probleme für eine bestimmte Klasse von Mensch-Roboter-Systemen zu lösen, schlagen wir zu Beginn dieser Arbeit einen modellgetriebenen Entwicklungsansatz vor. Durch die Analyse verwandter Arbeiten können wir festellen, dass solche Ansätze zwar nicht neu sind, jedoch in bisherigen Anwendungsfällen zumeist auf Petrinetzen oder sehr mächtigen aber komplexen Frameworks basieren. Wir wählen hier einen weiteren Ansatz, verteilte Mensch-Roboter Systeme mit Hilfe einer DSL auf Basis von UML-Statecharts zu modellieren. Weiterhin entwerfen und implementieren wir einen Programmgenerator, der es ermöglicht, aus Modellen unserer DSL voll funktionsfähige Anwendungen zu generieren. Um die Funktionsfähigkeit unseres Vorgehens zu evaluieren, setzen wir in einer Fallstudie die Entwicklung eines beispielhaften Systems, unter Nutzung unseres zuvor elaborierten Vorgehens, vollständig modellgetrieben um. Wir kommen dadurch zu dem Schluss, dass die modellgetriebene Entwicklung von verteilten Mensch-Roboter-Systemen sehr viele Vorteile mit sich bringt. Ebenfalls können wir die Nützlichkeit einer auf Statecharts aufbauenden DSL demonstrieren.

Inhaltsverzeichnis

1 Einleitung

In der Anfangsphase der modernen Robotik, wie wir sie heute kennen und in der Industrie alltäglich erleben, waren Roboter nichts weiter, als ferngesteuerte und dabei mehr oder minder frei bewegliche elektromechanische Systeme. Diese wurden für den simplen Zweck entwickelt, Objekte ohne direkte Aktion eines Menschen von A nach B zu befördern. Heutige Anwendungen sind damit kaum noch vergleichbar. Durch kontinuierliche technische Weiterentwicklung sind Roboter inzwischen wesentlich mächtiger und daraus resultierend in ihrer Funktion um ein Vielfaches komplexer geworden. So hat sich aus der Robotik auch längst ein interdisziplinäres Entwicklungsfeld gebildet, dessen Erfolg maßgeblich der Zusammenarbeit zwischen Elektrotechnik und Informatik zu verdanken ist (vgl. [RN07]). Mit den aktuellen Möglichkeiten durch das *Industrial Internet of Things* [JE17], stets fortschreitender Entwicklung im Bereich der Sensorik sowie sich kontinuierlich verbessernder Rechenleistung, Netzwerkverfügbarkeit und Übertragungsraten, können nun völlig neuartige Anwendungen, wie verteilte Mensch-Roboter-Systeme, realisiert werden. Diese Anwendungen stellen vor allem an die Softwareentwicklung hohe Anforderungen (vgl. [JE17]).

Parallel dazu hat sich jedoch auch die Softwaretechnologie und die Art und Weise des Softwareengineering stets weiterentwickelt. Wurde in den frühen Jahren der Softwareentwicklung ein Programm meist völlig unstrukturiert und den individuellen Erfahrungen des Programmierers folgend entwickelt, haben sich seit dem zunehmend systematische Herangehensweisen durchgesetzt. Diese reichen vom Einsatz etablierter Entwurfsuster [GA93] und standardisierter Modellierungssprachen bis hin zur vollständig modellgetriebenen Entwicklung [SVEH07].

1.1 Motivation

Mit dem Aufkommen des Industrial Internet of Things und den damit verbundenen Möglichkeiten, vor allem im Bereich der vernetzten robotischen Systeme, ist das Gebiet der verteilten Mensch-Roboter-Systeme in das Zentrum der Aufmerksamkeit gerückt. Dabei ist es Ziel, die Kollaboration von Mensch und Roboter [GS08] über eine stationäre Anlage hinaus zu ermöglichen.

Solche verteilten Mensch-Roboter-Systeme (VMRS) sind Systeme, welche aus einer Vielzahl von robotischen und menschlichen Arbeitern bestehen. Abbildung 1.1 zeigt ein beispielhaftes VMRS, in welchem ein Mensch eine Aktion mit dem Werkstück ausführt und zwei Roboter diese Aktion anschließend imitieren. Dabei wird das Verhalten und die Synchronisation der Roboter über einen Server gesteuert. In Abbildung 1.2 wird anschließend die dynamische Sicht auf einen der im System beteiligten Roboter stark vereinfacht dargestellt. Der Roboter befindet sich zu Beginn im Zustand *Bereit*. Wenn er nun einen Steuerbefehl erhält, führt er für eine bestimmte Zeitdauer eine Aktion aus. Währenddessen ist der Roboter im Zustand *Beschäftigt*. Wurde die Aktion erfolgreich beendet, wechselt der Roboter wieder in den Zustand *Bereit* und wartet auf neue Steuerbefehle. Wurde während der Aktion jedoch ein Fehler festgestellt, wechselt der

Roboter in den Zustand *Fehlerhaft*, in dem das System nun gesonderte Maßnahmen ergreifen muss.

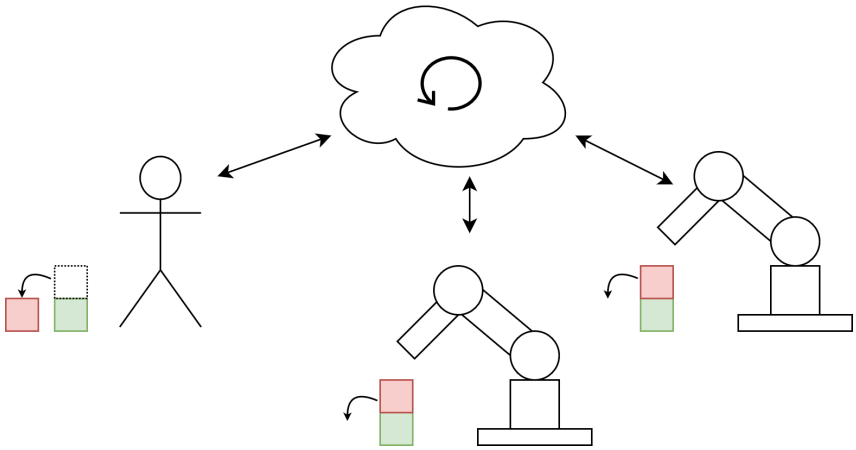


Abbildung 1.1: Beispiel verteiltes Mensch-Roboter System

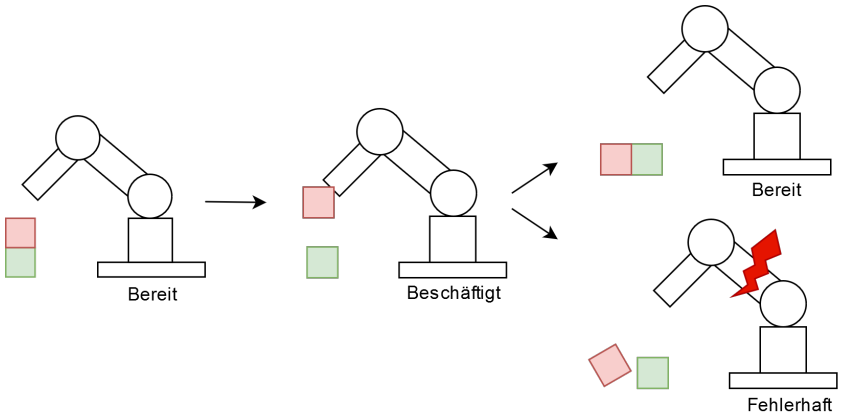


Abbildung 1.2: Dynamische Sicht auf einen beteiligten Roboter

Daraus lassen sich eine Vielzahl von möglichen Problemen ableiten. Einige ausgewählte sind:

1. Aufgrund der Komplexität in Hinsicht auf das hardwaretechnische Verhalten und die damit verbundene Ansteuerung über APIs, besteht ein hoher Bedarf an Experten mit entsprechenden Kenntnissen bezüglich solcher Systeme und Schnittstellen. Diese Experten sind während der Entwicklung jedoch oft nicht in ausreichendem Maße verfügbar. Daraus folgt, dass die Entwickler während der Erstellung eines Systems durch seine Komplexität oft überfordert sind und die resultierende Software dadurch fehlerbehaftet ist.

2. Durch das getrennte Modellieren und Implementieren der einzelnen Teilsysteme, entsteht oftmals eine Systemarchitektur, in welcher die Teilsysteme nicht erwartungsgemäß miteinander interagieren. Die Teilsysteme verhalten sich zum Zeitpunkt des Zusammenfügens nicht korrekt entsprechend ihrer Spezifikation oder aus der Spezifikation geht das Verhalten nicht vollständig hervor. Weiterhin führt eine solch problematische Softwarearchitektur ebenfalls zu schlechter Wart- und Erweiterbarkeit des Systems.

Daraus folgt das Kernproblem, dass die Entwicklung von Mensch-Roboter Systemen sehr komplex und damit fehleranfällig ist.

1.2 Zielsetzung

Nach der erfolgten Motivation und Herausstellung eines damit einhergehenden Kernproblems, wird nun die Menge der Zielsysteme genauer eingeschränkt sowie die im Rahmen dieser Arbeit angestrebten Ziele benannt.

In dieser Arbeit wird ein spezifischer modellgetriebener Ansatz gewählt, um die genannten Probleme für eine Teilmenge der zuvor beschriebenen Systeme zu lösen. Dafür wird die Menge aller verteilten Mensch-Roboter-Systeme wie folgt beschränkt:

1. Es gibt eine Menge von Arbeitsplätzen. Dabei unterscheiden sich menschzentrierte Arbeitsplätze und roboterzentrierte Arbeitsplätze.

2. An jedem Arbeitsplatz gibt es eine Menge von elektromechanischen Aktoren (z.B. Servomotor, Roboterarm, LED) und Sensoren (z.B. Näherungssensor, NFC-Reader, Drucktaster). Ebenfalls verfügt jeder Arbeitsplatz über einen Computer (Controller), welcher mit allen dazugehörigen Sensoren und Aktoren verbunden ist.

3. An einem menschzentrierten Arbeitsplatz ist ein Mensch dauerhaft anwesend, von welchem Aktionen gegen die Sensoren ausgehen und welcher Feedback von den Aktoren erhält. Menschen können über Interaktionen das Verhalten des Systems vorgeben. Dabei wird der Steuerbefehl des Menschen von Sensoren aufgenommen und über den Controller dem System zugeführt.

4. An einem roboterzentrierten Arbeitsplatz ist kein Mensch dauerhaft anwesend, sondern lediglich zur Wartung und Fehlerbehebung. Ein solcher Arbeitsplatz führt Steuerbefehle aus. Dabei richtet sich die Arbeit der verbundenen Aktoren gegen ein Werkstück. Die verbundenen Sensoren liefern Feedback zum Verhalten der Aktoren sowie dem Zustand des Werkstücks.

5. Die Controller aller Arbeitsplätze sind über ein Netzwerk miteinander verbunden und können so miteinander kommunizieren. Menschzentrierte Arbeitsplätze senden in erster Linie Steueraktionen in das Netzwerk und erhalten Feedback. Roboterzentrierte Arbeitsplätze erhalten in erster Linie Steueraktionen und senden Feedback zu ihrer Tätigkeit in das Netzwerk.

6. Ziel des Systems soll es sein, durch menschliche Interaktionen an den menschzentrierten Arbeitsplätzen, das Verhalten der Aktoren an den roboterzentrierten Arbeitsplätzen vorzugeben. Dabei muss ein Protokoll beachtet werden, welches Steuerungsregeln definiert und vom System eingehalten werden muss.

Der spezifische modellgetriebene Ansatz besteht dabei in der Programmgenerierung durch Nutzung einer domänenspezifischen Modellierung mittels UML. Im Zentrum stehen dabei Statecharts zur Modellierung der dynamischen Systemsicht sowie Klassen- bzw. Objektdiagramme zur Modellierung der statischen Systemsicht. Die Möglichkeiten von UML werden hierbei durch eine Definition domänenspezifischer Profile und Modellierungsregeln eingeschränkt, sodass die Aussagekraft der Modelle zur Codegenerierung erhöht wird.

Mit der genannten Lösungsstrategie sollen folgende Ziele erreicht werden:

1. Durch ein einheitliches und gesamtheitliches Modell besitzen die einzelnen Teilsysteme eine kompatible und aufeinander abgestimmte Architektur, da Inkompatibilitäten bereits durch die Modellierung aufgedeckt werden können. Daraus folgt auch, dass die Teilsysteme korrekt miteinander interoperieren.

2. Durch die Modellierung von Robotern und ihrem Verhalten auf verschiedenen Abstraktionsebenen entsteht ein Modell, welches ohne viel Expertenwissen direkt verwendet werden kann. Dadurch wird die Entwicklung von Robotersystemen stark vereinfacht. Daraus folgt auch, dass insgesamt nur wenig Expertenwissen zur Entwicklung solcher Systeme benötigt wird, was insbesondere aus der Weiterverwendbarkeit erstellter Modelle resultiert.

Dadurch soll das Kernziel erreicht werden, dass die Entwicklung der vorrangehend beschriebenen Systeme robust und fehlerarm erfolgt.

1.3 Methode und Aufbau

In den vorangehenden Abschnitten wurde das Problem, welches im Rahmen dieser Arbeit behandelt werden soll, motiviert sowie seine Lösungsansätze skizziert. Nachfolgend findet die Betrachtung von Grundlagen (Kapitel 2) der modellgetriebenen Entwicklung und Programmgenerierung statt. Dabei wird als erstes auf einige ausgewählte Techniken eingegangen, wie Systeme modelliert werden können. Darauf folgend, wird die modellgetriebene Entwicklung als Arbeitsweise an sich vorgestellt und anschließend werden die Techniken zur Programmgenerierung genauer untersucht. Im 3. Kapitel wird der aktuelle Einsatz von modellgetriebenen Methoden in der Robotertechnik anhand verwandter Arbeiten betrachtet und dabei existierende Beispielanwendungen aus der Forschung vorgestellt. Um eine Basis für die spätere Entwicklung zu schaffen, werden in Kapitel 4 Frameworks und Tools zur modellgetriebenen Entwicklung betrachtet und eines davon für die weitere Arbeit ausgewählt. Nach den nun erlangten Kenntnissen werden in Kapitel 5 die Initialschritte der modellgetriebenen Entwicklung ausgeführt, um unter anderem eine Zielarchitektur sowie einen prototypisch umgesetzten Codegenerator zu erhalten. In Kapitel 6 findet anschließend die Anwendung der in Kapitel 5 beschriebenen Werkzeuge, als auch einiger der in Grundlagen beschriebenen Techniken statt, um die skizzierte Lösungsstrategie anhand einer Fallstudie demonstrativ anzuwenden. In Kapitel 7 folgt daraufhin die Evaluation der Resultate dieser Arbeit. Dabei werden die Vor- und Nachteile des gewählten Ansatzes dargelegt. Ebenfalls wird aufgezeigt, wie durch mögliche Erweiterungen der Lösungsstrategie, diese auf eine größere Zielklasse angewandt werden kann.

2 Grundlagen

2.1 Modellierung

In diesem Abschnitt werden einige ausgewählte Techniken der Modellierung vorgestellt. Dabei soll ein grober Überblick über diejenigen Methoden vermittelt werden, welche im weiteren Verlauf dieser Arbeit eine zentrale Rolle einnehmen.

2.1.1 Klassendiagramme - Statische Systemsicht

In diesem Abschnitt wird auf die Verwendung und Bedeutung von Klassendiagrammen für die Modellierung eingegangen. Die detaillierte Spezifikation von Klassendiagrammen und ihrer Notation kann aus dem UML-Standard [UML17] entnommen werden.

Klassendiagramme sind eine Möglichkeit der Modellierung und Darstellung einer statischen Sicht auf ein System (vgl. [KA09]). Dabei basieren Klassendiagramme auf dem objektorientierten Konzept der Softwarearchitektur. Ein solches objektorientiertes System besteht aus dem Zusammenspiel vieler Objeke, welche mit ihren Eigenschaften und Beziehungen in Form von Klassen definiert werden, so dass innerhalb eines Programms sämtliche Objekte dynamisch aus den Klassendefinitionen erzeugt werden können (vgl. [SO07]). Bei der Verwendung von Klassendiagrammen im Prozess der modellorientierten Entwicklung, unterscheidet man zwei Arten bezüglich ihres Anwendungsgebietes. "In der Analyse liegt das Interesse auf der Darstellung der Strukturen im Anwendungsbereich und der Erfassung der Anforderungen an das System. Später verschiebt sich der Interessenschwerpunkt auf IT-spezifische Objekte"[KA09]. Üblicherweise wird dabei die Bezeichnung Analyseklassendiagramm für Spezifikationen während der Analyse der Domäne und Entwurfsklassendiagramm für Spezifikationen im Softwareentwurf verwendet.

Abb. 2.1 zeigt ein simples Beispiel eines Analyseklassendiagramms. Dabei ist zu beachten, dass weder Sichtbarkeiten, noch technische Details dargestellt werden. Jedoch ist auch ein sehr kleines Analyseklassendiagramm durch seine Struktur sehr gut geeignet, um Informationen einer Domäne aussagekräftig und übersichtlich darzustellen.

Abb. 2.2 zeigt ein simples Beispiel eines Entwurfsklassendiagramms. Dabei ist zu beachten, dass nun auch Details der Implementierung, wie Sichtbarkeiten und Rückgabetypen von Methoden, im Diagramm dargestellt werden. Dadurch entsteht ein hoher Grad an Aussagekraft über den konkreten Aufbau des Systems. Diese Eigenschaft, sowie der Vorteil, dass während der Programmgenerierung die Informationen eines Klassendiagramms sehr direkt (siehe Kapitel 2.3.1 Template Techniken) in Programmcode überführt werden können, sind der Grund, dass Modellierungs-Frameworks wie das EMF ein Klassendiagramm als Kern eines Modells vorsehen (siehe Kapitel 4).

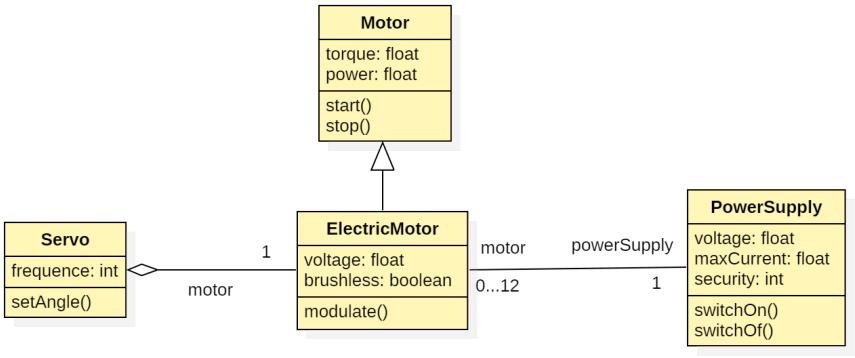


Abbildung 2.1: Beispiel Analyseklassendiagramm

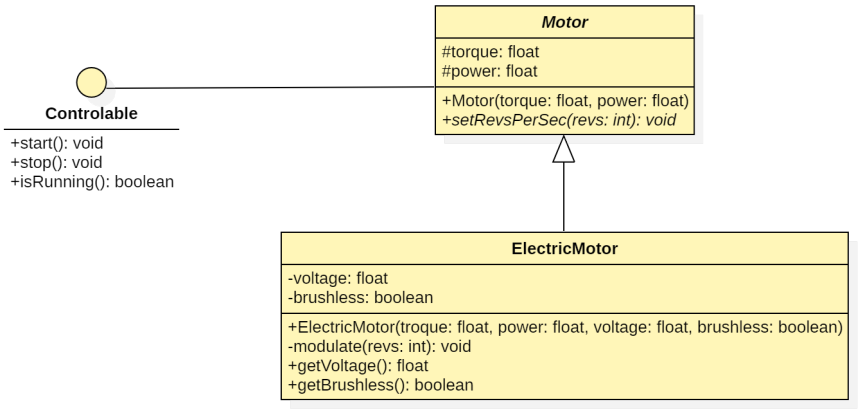


Abbildung 2.2: Beispiel Entwurfsklassendiagramm

2.1.2 Zustandsautomaten - Dynamische Systemsicht

Zustandsautomaten sind ein Formalismus um die dynamische Sicht auf ein System darzustellen. Auf Grund ihrer großen Bedeutung in den folgenden Teilen dieser Arbeit, wird ihnen in diesem Abschnitt besondere Aufmerksamkeit geschenkt. In Abb. 2.3 werden dazu die hier verwendeten Begriffe in Beziehung gesetzt. Wenn dabei in dieser Arbeit der Begriff *Zustandsautomat* verwendet wird, ist damit immer ein Automat im Sinne einer *Statechart* [HA87] gemeint. Andere zustandsbasierte Automaten wie PDAs oder Touring-Maschinen, welche formale Sprachen beschreiben, werden in dieser Arbeit nicht berücksichtigt.

Ein Zustandsautomat ist ein spezieller Automat, welcher aus einer endlichen Anzahl von Zuständen, seinen Konfigurationen, besteht. Ein Zustand trägt dabei implizit die Information aus vorrangegangenen Aktionen um spezifisch auf Folgeaktionen reagieren zu können. Die hier verwendete Notation von Zustandsautomaten entspricht der UML, wobei Zustände als abgerundete Rechtecke und Zustandsübergänge als Pfeile gekennzeichnet werden. Für detaillierte Information zur Notation, siehe im UML-Standard [UML17].

Um einen Zustandsautomat zu entwerfen, sind Kenntnisse über mögliche Eingaben (Ereignisse) in das System sowie der daraus resultierenden Reaktionen (Aktionen) des Systems erforderlich. Daraus kann abgeleitet werden, wie einzelne Ereignisse einen Zustandsübergang des Systems bewirken und welche Zustände das System besitzen muss. In der Umsetzung dieses Konzeptes können zwei Arten von klassischen Zustandsautomaten unterschieden werden: Mealy- und Moore-Automaten. (vgl. zu diesem Abschnitt [BA09])

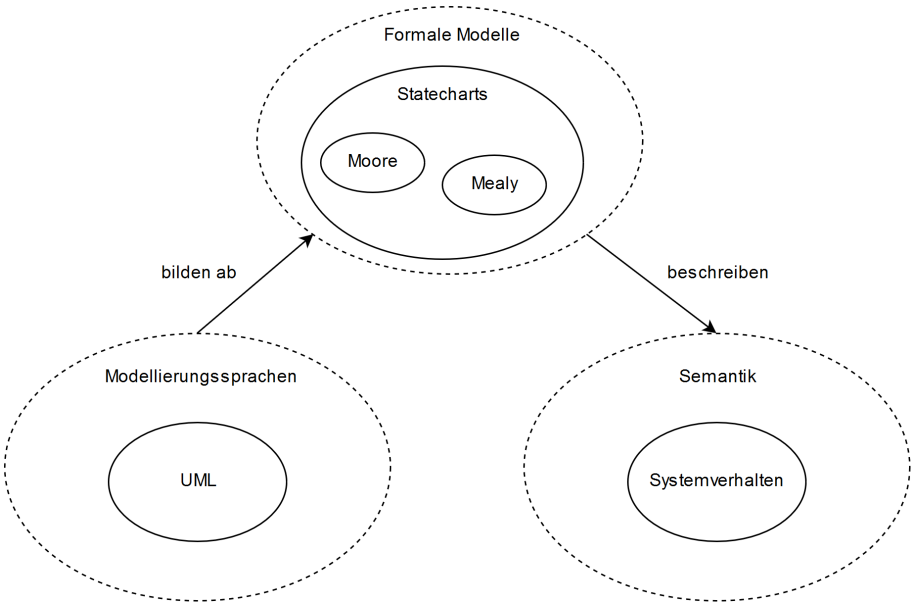


Abbildung 2.3: Ontologie Begriffe Automaten

Mealy-Automat

Im Mealy-Automat werden Zustände als Zeitintervalle konstanter Aktivität betrachtet, welche durch Zustandsübergänge auf Ereignisse mit Aktionen folgend gewechselt werden (vgl. [BA09]). In Abb. 2.4 wird dies an einem Beispiel verdeutlicht.

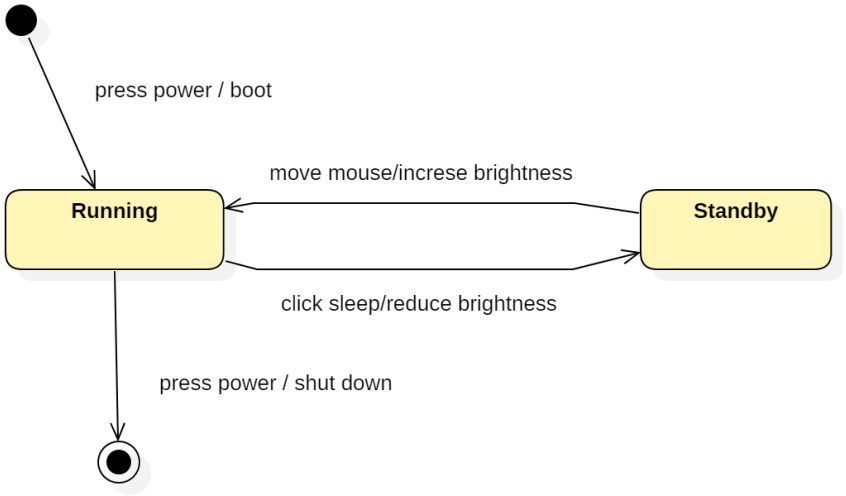


Abbildung 2.4: Beispiel Mealy-Automat

Moore-Automat

Im Moore-Automat ist die Aktion nicht an das Ereignis, sondern an den Zustand gekoppelt. D.h., ein eintretendes Ereignis bewirkt lediglich den Wechsel des Zustandes. Im Zustand selbst ist dabei definiert, welche Aktionen im jeweiligen Zustand ausgeführt werden sollen. Diese zustandsbezogenen Aktionen werden auch als Aktivitäten bezeichnet (vgl. [BA09]). In Abb. 2.5 wird dies an einem Beispiel verdeutlicht.

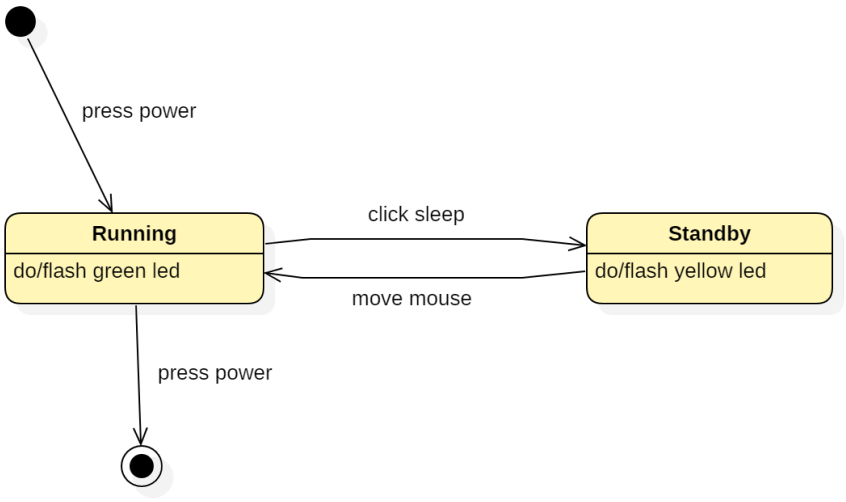


Abbildung 2.5: Beispiel Moore-Automat

Statecharts

Da die Mächtigkeit von Mealy- und Moore-Automaten zur Modellierung komplizierter Systeme oftmals nicht ausreicht, wurden von David Harel einige Erweiterungen eingeführt. Diese sogenannten *Statecharts* [HA87] ermöglichen unter anderem die Kombination von Mealy- und Moore-Automaten, an Bedingungen geknüpfte Zustandsübergänge, Gedächtniszustände, Pseudozustände sowie hierarchische Zustandsautomaten (Refined Statecharts). Auf einige ausgewählte Details wird nun genauer eingegangen (vgl. zu diesem Absatz [BA09]).

Durch die erweiterte Syntax von Statecharts, wie sie detailliert von Harel [HA87] beschrieben und in der UML [UML17] standardisiert ist, ist es möglich, Zustandsautomaten wie beispielsweise in Abb. 2.3 und 2.4 zu konstruieren:

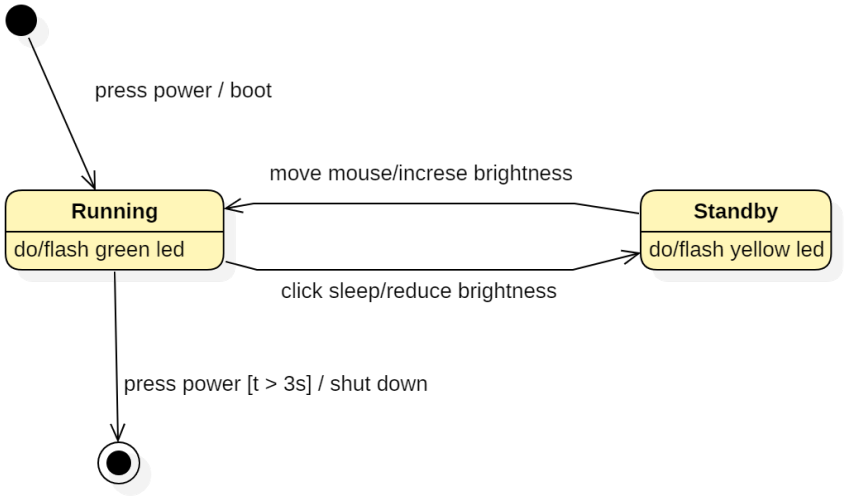


Abbildung 2.6: Beispiel Statechart 1

Das in Abb. 2.3 gezeigte Statechart zeigt an einem einfachen Beispiel die Kombination des Mealy- und Moore-Automaten (Hybridautomat) mit einem Guard an der Transition zum Endzustand. Durch die Nutzung der Eigenschaften beider Grundtypen, wird es so möglich, über-

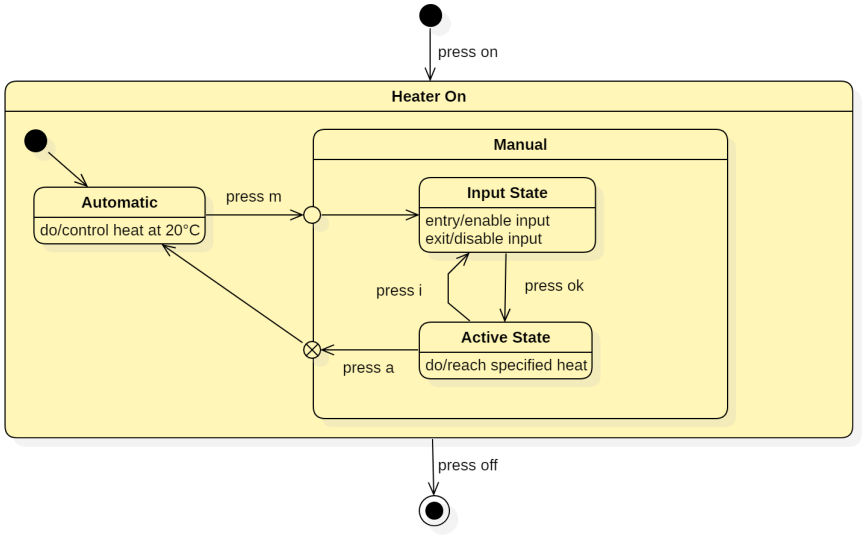


Abbildung 2.7: Beispiel Statechart 2

gangsgebundenes als auch zustandsgebundenes Verhalten zu modellieren.

Das in Abb. 2.4 gezeigte Statechart verdeutlicht an einem einfachen Beispiel die zusätzlichen Möglichkeiten durch die Einführung von Hierarchien. Dabei ist es nun möglich, Systeme auf verschiedenen Abstraktionsebenen zu modellieren.

2.1.3 Petrinetze

Petrinetze [PE62] sind eine grafische Sprache, die bereits in den 1960er Jahren von Carl Adam Petri entwickelt wurde, um das Verhalten von Computersystemen präzise zu modellieren (vgl. [WR10]). Dabei zeichnet Petrinetze unter anderem aus, dass sie es ermöglichen, auch nebenläufige und nichtdeterministische Systeme zu modellieren (vgl. [JK09]).

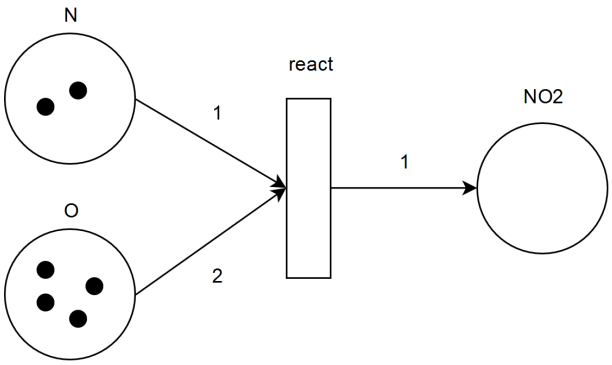


Abbildung 2.8: Beispiel Petrinetz

Die Bestandteile eines Petrinetzes sind dabei Plätze, Transitionen und Kanten, welche Plätze mit Transitionen verbinden.

Abbildung 2.7 zeigt ein einfaches Petrinetz. Feuert die Transition *react*, werden ein Token aus *N* und zwei Token aus *O* entfernt. Zur gleichen Zeit wird ein neuer Token in *NO2* abgelegt. Die Transition *react* kann dementsprechend nur höchstens zwei mal feuern.

Die Netzwerkstruktur als auch das Verhalten von Petrinetzen ist mathematisch definiert. Unter Nutzung dieser mathematischen Beschreibbarkeit von Petrinetzen ist es möglich, verschiedene Eigenschaften des Modells zu beweisen. Solche beweisbaren Eigenschaften sind z.B. Beschränktheit

der Tokenzahl, Erreichbarkeit von Markierungen, Lebendigkeit von Transitionen (vgl. [EN94]). Durch diese Möglichkeit, modellierte Systeme auf direkt mathematischem Weg zu verifizieren, entsteht ein großer Vorteil bei der Entwicklung von sicherheitskritischen (Sub-)Systemen, bei denen Fehlerfreiheit garantiert werden muss (vgl. zu diesem Absatz [WR10])

Coloured Petrinetze - CPNs

Coloured Petrinetze sind eine Erweiterung der elementaren Perinetze um die Eigenschaften von höheren Programmiersprachen. Dabei bleibt das Grundprinzip eines Netzes zwar gleich, jedoch wird es, durch das Hinzufügen von Mächtigkeit mittels der CPN ML Programmiersprache, nun möglich, selbst komplexe Systeme mit verschiedenen Datentypen und Datenmanipulationen zu modellieren (vgl. zu diesem Absatz [JK09]).

2.2 Modellgetriebene Softwareentwicklung

> "Modellgetriebene Softwareentwicklung (Model Driven Software Development, MDSD) ist ein Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugen." [SVEH07]

2.2.1 Motivation und Ziele

Modelle und vor allem ihre grafische Darstellung, sind in der Softwareentwicklung, wie in anderen Ingenieursdisziplinen auch, kein unübliches Mittel um Aspekte eines Systems zu illustrieren, zu beschreiben oder zu veranschaulichen. Oftmals geht dabei die Funktion des Modelles jedoch nicht über das Dasein als einfaches Artefakt hinaus, sei es zur Dokumentation oder Spezifikation. Die Software, also das System an sich, wird i.d.R. getrennt vom Modell programmiert. Dabei dient das Modell zwar als Vorlage, doch hat der Programmierer bei uneindeutigen Details viel Spielraum. Hinzu kommt, dass Änderungen und Fehlerbehebungen in späteren Projektphasen oftmals direkt im Quellcode umgesetzt werden. Anschließend werden die Dokumentation und Modelle getrennt angepasst, was nicht selten zu Inkonsistenzen zwischen Modell und Programm führt, welche wiederum Fehler versursachen können. Durch den Einsatz von MDSD sollen im Wesentlichen zwei Ziele erreicht werden: die Verbesserung der Softwarequalität als auch die Steigerung der Entwicklereffizienz. Dies soll dabei durch Abstraktion, eine einheitliche Architektur, erhöhte Entwicklungsgeschwindigkeit, Wiederverwendbarkeit und Plattformunabhängigkeit erreicht werden (vgl. zu diesem Abschnitt Stahl et al. [SVEH07]). Stahl et al. fanden dazu fünf Gründe, welche im Folgenden zusammengefasst werden:

Abstraktion

Bei der Entwicklung eines Softwaresystems werden üblicherweise zwei Tätigkeiten gleichzeitig durchgeführt. Zum Einen die Erstellung der eigentlichen Systemlogik (Business Logic) sowie die Konfektionierung auf eine Implementation (z.B. Java Quellcode). Diese Tätigkeiten werden bei nicht MDSD basierten Vorgehen während der Programmierung durchgeführt, selbst dann, wenn die Business Logic zuvor gesondert spezifiziert wurde.

Modelle beschreiben ein System nun als Menge von Bausteinen, die auf einer höheren Ebene als die zugrunde liegende Programmiersprache liegen. Dadurch wird die gesamte Komplexität, die vorher der Programmierung innewohnte, nun auf die Modellierung und die Programmgenerierung aufgeteilt. Dabei handelt es sich um zwei getrennt ausführbare Tätigkeiten. Durch diese Abspaltung der implementationstechnischen Aspekte, kann sich das eigentliche Modell vollständig der Komplexität der Anwendung widmen, was die Entwicklung insoweit vereinfacht, dass dem Entwickler ermöglicht wird, sich zur Zeit des Entwurfs auf die reine Business Logik zu konzentrieren und diese anschließend nicht mehr gesondert in Quellcode übertragen werden muss.

Einheitliche Architektur

Während der Entwicklung von Softwaresystemen kommt es oft dazu, dass das System uneinheitlich erstellt wird. Man bezeichnet dies auch als Verwässern. Gründe dafür sind oft ein großes System mit vielen Entwicklern. Dadurch werden eigentliche Kernkonzepte an verschiedenen Stellen verschieden implementiert, was vor allem die Wart- und Änderbarkeit erschwert. Beim MDSD gibt es jedoch von vorn herein nur ein Modell. An diesem Modell können natürlich mehrere Entwickler arbeiten, die Kernkonzepte sind jedoch zentral definiert und müssen überall im Modell gleichartig verwendet werden. Ebenfalls wird bei der abschließenden Programmgenerierung das Modell nach gleichartigen Schemen übersetzt, die im entsprechenden Toolset oder Framework definiert sind. In jedem Fall kann davon ausgegangen werden, dass ein einheitliches Modell auch zu einer einheitlichen Quellcoderepäsentation umgewandelt wird.

Erhöhte Entwicklungsgeschwindigkeit

Die erhöhte Entwicklungsgeschwindigkeit modellgetriebener Softwareentwicklung ist vor allem auf die einfache Änderbarkeit zurückzuführen. Durch ein jederzeit mit der Funktion der Software konsistenem Modell, können zum Einen Fehler schnell identifiziert und Features schnell eingebracht werden. Die Kompatibiltität lässt sich direkt aus dem erfolgreichen Einbinden in das bestehende Modell ableiten.

Wiederverwendbarkeit

Im Rahmen einer Produktlinie oder allgemein zur Umsetzung verwandter Anwendungsfälle ist das MDSD besonders gut geeignet, da sich ein Großteil der entstehenden Artefakte direkt weiterverwenden lassen. So können z.B. Programmgeneratoren ohne Anpassung für beliebige Systeme wiederverwendet werden, die sich die gleiche Zielplattform teilen. Ebenfalls können Modellteile, wie einmalig definierte Domain-Specific Languages oder UML-Profile, über eine Vielzahl von Projekten wiederverwendet werden.

Plattformunabhängigkeit

Durch die Nutzung von MDSD wird die Plattformunabhängigkeit von Softwaresystemen erhöht, was insbesondere eines der Hauptziele der Standardisierungsbemühungen der OMG in Form der Model Driven Architecture MDA [MDA14] ist. Auf Grund eines plattformunabhängigen Modells, wie z.B. auf der Basis von UML, die wird die Interoperabilität zwischen verschiedenen Plattformen deutlich im Vergleich zu statischem Quellcode verbessert. Eine Spezifikation nach der MDSD kann trotzdem nicht ohne zusätzliche Erweiterungen für verschiedene Plattformen generiert werden. Falls dies geschehen soll, wird zum Einen für jede Zielplattform ein entsprechender Generator benötigt, welcher auf die Zielsprache abbilden kann. Zum Anderen deckt ein Modell i.d.R. nicht das vollständige System ab, sondern es muss mit plattformspezifischem Quellcode angereichert werden. Jedoch werden vor allem bei der Portierung von sehr großen Softwaresystemen deutliche Zeitersparnisse erreicht, wenn diese mittels MDSD entwickelt wurden.

2.2.2 Vorgehensweise

Nachdem die Ziele der modellgetriebenen Softwareentwicklung dargestellt wurden und bekannt ist, wie diese aus dem Entwicklungskonzept hervorgehen, wird nun eine allgemeine Vorgehensweise in MDSD Projekten vorgestellt, welche sich aus den von Stahl et al. beschriebenen< Abschnitten zusammensetzt:

1. Elaboration: Zunächst wird auf Basis der bekannten Anforderungen, eine technologieunabhängige Architektur entworfen, mit welcher sich das Zielsystem am geeignetsten realisieren

lässt. Dieses Architekturmodell ist essenziell für die weiteren Schritte, sodass auf klare Definitionen und Dokumentationen zu achten ist. Auf Grundlage der Architektur wird im folgenden ein Programmiermodell definiert, welches beschreibt, wie Workflows und Usecases auf Basis der speziellen Architektur umgesetzt werden können. Ebenfalls erfolgt bereits während der Elaboration die Definition des Technologie-Mappings. Dieses Mapping legt fest, wie das zuvor entworfene Architekturmodell auf die konkrete Zielplattform abgebildet wird. Des Weiteren kann bereits auf Grundlage der vorhergehenden Spezifikationen eine Mock-Plattform entwickelt werden um Modultests in jedem Entwicklumsstadium zu ermöglichen. Zur Machbarkeitsprüfung der Konzepte ist es ebenfalls angebracht, einen aussagekräftigen Prototypen zu entwerfen und auf der Mock-Plattform auszuführen.

2. Iteration: Nachdem in der Phase der Elaboration die gundlegenden Mechanismen definiert wurden, müssen diese nun stabilisiert und in ihrer Nützlichkeit validiert werden. Dazu werden die in der Elaboration beschrieben Schritte iterativ wiederholt, bis sämtliche Definitionen und Spezifikationen dafür geeignet sind, im Produktiveinsatz genutzt zu werden.

3. Automation: In der Phase der Automationen werden aus den nun stabilen Spezifikationen alle nötigen Werkzeuge für die eigentliche modellgetriebene Arbeit erzeugt. Dabei wird unter anderen der nötige Quellcode für die Programmgenerierung entwickelt, also das Technologie-Mapping konkret umgesetzt. Dazu gehört ebenfalls die automatische Generierung von Konfigurations- und Build Dateien. Um die Arbeit mit dem Programmiermodell weiter zu vereinfachen, kann ebenfalls eine geeignete Domain-Specific Language (DSL) definiert werden. So eine DSL kann beispielsweise textuelle Form haben oder durch ein UML-Profil definiert sein.

2.3 Programmgenerierung

Einer der größten Vorteile der modellgetriebenen Softwareentwicklung ist, dass aus Modellen lauffähige Programme erzeugt werden können. Dies soll im besten Fall voll automatisiert und ohne manuelle Quellcodemodifikation geschehen. In diesem Abschnitt wird vorgestellt, mit welchen Techniken dies erreicht werden kann.

2.3.1 Techniken

Template Techniken

Template Techniken sind in der modellgetriebenen Softwareentwicklung, vor allem objektorientierter Systeme, am weitesten verbreitet. Dabei wird so viel Quellcode wie möglich in Form von Vorlagen (Templates) bereitgestellt. Diese Templates enthalten bereits in der syntaktischen Struktur des Zielsystems diejenige Funktionalität, die im Modell durch Abstraktion und Einschränkung auf ein konkretes Zeilsystem weggelassen werden konnte. Wurde beispielsweise während der Elaboration definiert, dass jede Klasse des Systems serialisierbar sein soll und ebenfalls wie diese Serialisierung erfolgt, muss dies im Modell nicht dargestellt werden. Wird aus einer Modellklasse nun Quellcode generiert, werden die Eigenschaften der Modellklasse zunächst in die Syntax des Zielsystems transformiert und anschließend in Platzhalter von vordefinierten Templates eingetragen. Dabei gibt es nun auch ein Template für die Methode zur Serialisierung, welches über eine ebenfalls zuvor spezifizierte Logik vervollständigt wird. Um Templates und Regeln zur Vervollständigung dieser zu definieren, kommen spezifische Frameworks, sogenannte Template Engines zum Einsatz. Ein solches Framework ist beispielweise Apache Velocity [VELO16]. Um zu verdeutlichen, wie innerhalb des CASE-Tools Fujaba ein Codegenerator mithilfe von Velocity Templates umgesetzt wird, kann die Arbeit von Geiger et al. [GSR05] betrachtet werden.

Formale Techniken

Automaten, wie endliche Automaten und auch verwandte Konzepte wie Petrinetze oder Statecharts implizieren auf Grund ihrer Definition bereits fast vollständig ihre Implementierung. So können unter anderem Frameworks entwickelt werden, welche die Funktionsweise des Automaten bereits so vollständig implementieren, dass die eigentliche Spezifikation nur noch die Eingabe für einen Interpreter ist, der diese vollständig simulieren kann. Ein solches Framework ist beispielsweise Bestandteil der CPN-Tools zum Simulieren von Coloured Petrinetzen [CPN19].

2.4 Nachteile der modellgetriebenen Softwareentwicklung

Das Vorgehen der modellgetriebenen Softwareentwicklung bringt auch Nachteile mit sich. Die Vorteile der MDSD liegen, wie im vorhergehenden Kapitel 2.2 beschrieben, vor allem in der Aufteilung der Komplexität des Programmierens auf verschiedene Teilprozesse. Dadurch entstehen jedoch neue Probleme, die, neben dem hohen Initialaufwand, die Durchführung der MDSD erheblich behindern können.

1. Das Spezifizieren einer plattformunabhängigen Architektur erfordert umfassende Kenntnis zu möglichen Anforderungen und Usecases. Muss die Architektur in späteren Projektphasen nur leicht verändert werden, müssen i.d.R. viele Arbeitsschritte der Elaboration, Iteration und Automation wiederholt werden, um die erfolgten Änderungen umzusetzen (vgl. [SVEH07]).
2. Ist die Abbildung von Modell auf Code oder der Codegenerator selbst fehlerbehaftet, pflanzen sich diese Fehler oft unbemerkt lange fort und zeigen erst zur Programmlaufzeit ihre Wirkung.
3. Es wird für die Umsetzung des MDSD eine hohe Expertise für Modellierung und Management des Entwicklungsprozesses benötigt.

In ihrer Studie *Relevance, benefits, and problems of software modelling and model driven techniques—A survey in the Italian industry* [TO13] untersuchen Torchianoa et al. die Vorteile und Probleme beim Einsatz von modellgetriebenen Methoden in der Italienischen Softwareindustrie. Dabei finden die Autoren drei als signifikant einzustufende Probleme:

1. Zu viel Mehraufwand im Vergleich zu herkömmlichen Vorgehensweisen
2. Zu wenig erwarteter Nutzen
3. Mangel an Expertise im Bereich des MDSD und dazugehöriger Techniken

3 Verwandte Arbeiten

3.1 Einsatz von modellgetriebener Entwicklung in der Robotertechnik

3.1.1 Modellgetriebene Entwicklung von Robotern mit Smartmars

A. Schleck und C. Schlegel beschäftigen sich in ihrer Arbeit mit dem Titel "Towards Quality of Service and Resource Aware Robotic Systems through Model-Driven Software Development" [SS10] mit den Problemen, die durch klassisches Programmieren von robotischen Systemen entstehen und wie sich diese durch einen modellgetriebenen Ansatz lösen lassen. Die Arbeit zeigt, welche Herrausforderung es bei der Entwicklung robotischer Systeme gibt, insbesondere in Hinblick auf nichtfunktionale Anforderung und Quality of Service (QoS).

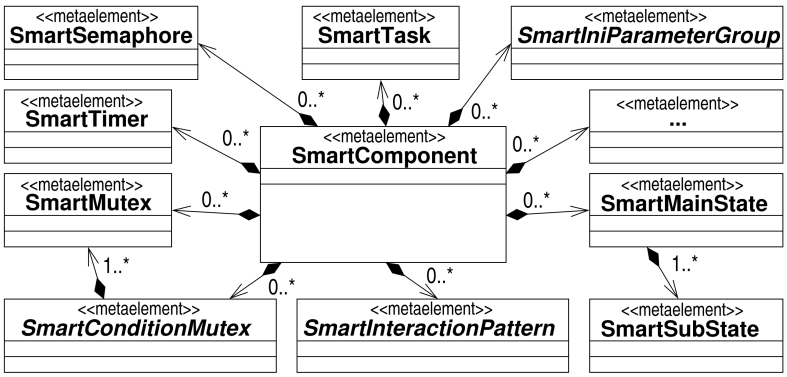


Abbildung 3.1: Auszug des SMARTMARS Metamodells [SS10]

Das behandelte Kernproblem von A. Schleck und C. Schlegel unterscheidet sich damit zwar von dem dieser Arbeit, doch wird zu dessen Lösung ebenfalls der Ansatz der modellgetriebenen Softwareentwicklung gewählt. Dabei zeigen die Autoren, wie ein beispielhafter Workflow der MDSD mit Hilfe der Eclipse Umgebung umgesetzt werden kann. Das verwendete Metamodell ist das SMARTMARS Metamodell [SM13]. Dieses Metamodell stellt eine allgemeinere Alternative des in Kapitel 5 vorgestellten Metamodells dieser Arbeit dar. Abb. 3.1. zeigt einen Ausschnitt des SMARTMARS Metamodells in UML Notation.

3.1.2 Einsatz von MDA in der Entwicklung mobiler Roboter

Einen Ansatz, der den Standardisierungen der MDA folgt, ist die Arbeit "Model-Driven Development of Intelligent Mobile Robot Using Systems Modeling Language (SysML)" [RA11] von

Rahman et al. Dabei begegnen die Autoren dem Problem, dass viele vergleichbare Ansätze der Entwicklung von Robotern mittels MDA unzureichend bezüglich Modularisierung und industriellen Ansprüchen sind. Zur Lösung dieses Problems, stellen Rahman et al. in ihrer Arbeit einen Design-Prozess vor, der die existierenden Defizite berücksichtigt. Dabei setzen die Autoren auf die Entwicklung nach der MDA mittels SysML [SYS17] "for the design and development of robot software modules in order to achieve module reusability"[RA11] sowie auf eine umfassende Anforderungsanalyse.

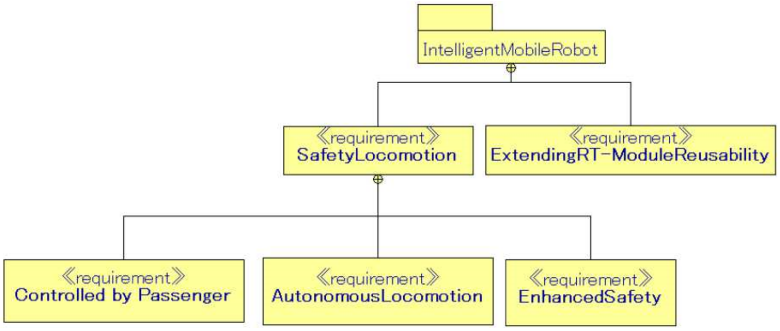


Abbildung 3.2: Top-Level Requirements der behandelten Fallstudie [RA11]

Abbildung 3.2. zeigt die Top-Level Sicht auf die Anforderungen in der Fallstudie eines mobilen Roboters von Rahman et al. Die dafür verwendete Modellierungssprache ist SysML.

3.1.3 Modellierung von Robotern mit Petrinetzen

In ihrer Arbeit "Using the Time Petri Net Formalism for Specification, Validation, and Code Generation in Robot-Control Applications" [MGV00] untersuchen L. Montano, F.J. García und J.L. Villarroel, wie sich die Steuerung von Robotern durch Petrinetze modellieren lässt. Dabei folgen Montano et al. dem Ziel, den Code für ihren Anwendungsfall vollständig aus dem Petrinetz generieren zu lassen. Dabei kamen die Autoren zu dem Schluss, dass "[The used technologies] tend to simplify the system development and, therefore, reduce its cost and the time needed significantly"[MGV00] Jedoch mussten Montano et al. auch fest stellen, dass "despite the simplicity in the implementation, the technique presents several problems"[MGV00], welche jedoch nicht weiter ausgeführt wurden.

3.2 Codegenerierung aus Statechart Modellen

Eine Herausforderung der modellgetriebenen Entwicklung mittels Statecharts, ist die finale Codegenerierung. In ihrer Arbeit "Code Generation from Uml Statecharts" [NT03] untersuchen I.A. Niaz und J. Tanaka zunächst die Probleme, welche auftreten, wenn man versucht, Statecharts entsprechend existierender State-Pattern in Code zu transformieren. Dabei wird beispielsweise der History-State genannt, welcher in üblichen State-Pattern nicht berücksichtigt wird. Zur Lösung dieser Probleme, schlagen Niaz und Tanaka eine neue Implementierungsstrategie vor, welche auf existierenden Lösungen aufbaut und diese durch weitere Konzepte der objektorientierten Programmierung erweitert, so dass es mit ihrer Strategie möglich ist, vollständige Codegenerierung zu erreichen.

4 Tools und Frameworks für modellgetriebene Softwareentwicklung

Dieses Kapitel dient der Vorstellung von Tools und Frameworks zur modellgetriebenen Softwareentwicklung. Da die Auswahl an Programmen und Frameworks in diesem Bereich groß ist, werden wir uns hier auf eine kleine Auswahl beschränken. Es wurden hier einige Tools und Frameworks aus der Eclipse Umgebung ausgewählt, da diese als freie Software verfügbar sind und im Produktiveinsatz sehr oft genutzt werden. StarUML wird als weiteres Tool präsentiert, da wir es im Rahmen dieser Arbeit einsetzen.

4.1 Eclipse Umgebung

Die folgenden Abschnitte zu Eclipse Tools und Frameworks wurden aus unserem technischen Report "Codegenerierung - Herausforderungen und Ansätze" [KK19] übernommen.

4.1.1 Eclipse Modelling Framework und Ecore

Das *Eclipse Modeling Framework* (EMF) [EMF19] ist das zentrale Framework im Kontext der Eclipse Umgebung zur modellgetriebenen Entwicklung.

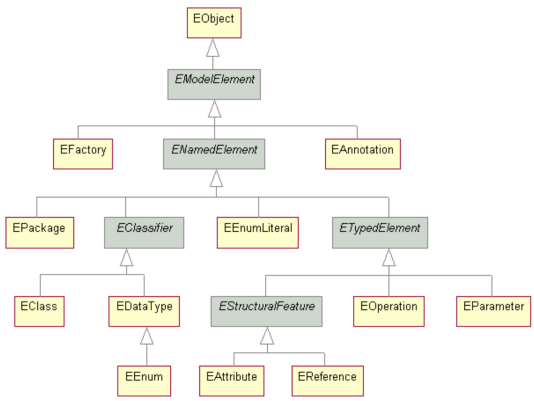


Abbildung 4.1: Übersicht Ecore Komponenten [EC19]

Im Mittelpunkt des EMF steht Ecore [EC19]. Ecore ist ein Metametamodell, welches zum Einen selbst auf dem EMF basiert, als es auch definiert. Abbildung 4.1 zeigt ein vereinfachtes

Klassendiagramm der Ecore Modellklassen. Das Diagramm dient zur Veranschaulichung der Vererbungshierarchie innerhalb von Ecore. Es kann so erkannt werden, wie Ecore aufgebaut ist und welche Möglichkeiten damit geboten werden um eine eigene Domain-Specific Language (DSL) zu entwerfen (vgl. zu diesem Absatz Steinberg et al. [EMF09]). Nach Steinberg et al. zusammengefasst, erfüllt das EMF unter anderem folgende Funktionen:

1. Das EMF implementiert das Ecore Metametamodell in Java und ermöglicht so seine Verwendung auf Programmebene.
2. Mit Hilfe des EMF können Ecore Modelle erstellt und in einem Baum-basierten Editor bearbeitet werden.
3. Ecore Modelle werden in Form von XMI Dateien serialisiert. XMI ist dabei ein standardisiertes Format, welches auf XML basiert.

DSLs nehmen im modellgetriebenen Entwicklungsprozess oftmals eine zentrale Rolle zur Modellierung von Domäneneigenschaften und Geschäftsprozessen ein. Die Eclipse Umgebung stellt auf Basis des EMF hierzu mehrere Tools bereit, von denen im Folgenden *Xtext* [XT19] und *GMF* [GMF19] vorgestellt werden.

4.1.2 Xtext

> "Xtext is a framework for development of programming languages and domain-specific languages. With Xtext you define your language using a powerful grammar language. As a result you get a full infrastructure, including parser, linker, typechecker, compiler as well as editing support for Eclipse, any editor that supports the Language Server Protocol and your favorite web browser." [XT19]

Xtext ermöglicht es dabei dem Nutzer, innerhalb der Eclipse Umgebung eigene DSLs zu definieren. Dabei baut Xtext auf dem EMF auf. Dadurch basieren auch sämtliche mit Xtext erstellten Sprachen auf dem Ecore Metametamodell. Zur Definition einer Sprache mittels Ecore, stellt Xtext dem Nutzer verschiedene Editoren, sowie Modelchecker und Verwaltungstools bereit. Wurde die Grammatik einer DSL definiert, ist Xtext in der Lage, daraus einen Parser, Linker, Typechecker und Compiler abzuleiten. Damit kann die DSL anschließend innerhalb von Eclipse direkt in textueller Form genutzt werden. Der generierte Compiler übersetzt die DSL dabei standardmäßig zu Java Quellcode. (vgl. zu diesem Abschnitt [XT19])

4.1.3 Graphical Modeling Framework

Das *Graphical Modeling Framework* ist ein auf der Eclipse Umgebung basierendes Tool, dessen zentrale Funktionalität es ist, für zuvor mittels Ecore definierte Sprachen grafische Editoren zu generieren. Dabei ist das GMF in der Lage, Metamodelle, die beispielsweise mittels Xtext erzeugt wurden, einzulesen. Anschließend ist es über das Tool möglich, Elementen des Metamodells grafische Repräsentanten zuzuordnen, sodass ein grafisches Metamodell entsteht. Ist das Metamodell grafisch spezifiziert, kann mittels des GMF ein Editor, mit welchem die jeweilige Sprache editiert werden kann, vollständig generiert werden. Dieser Editor basiert jedoch zwangsläufig auf Java. (vgl. zu diesem Abschnitt [GMF19])

4.2 Eclipse Papyrus

Eclipse Papyrus [PA19] ist eine Modellierungsumgebung, welche innerhalb der Eclipse IDE genutzt werden kann. Papyrus zeichnet sich dabei durch seinen großen Funktionsumfang aus. So

unterstützt Papyrus neben dem UML 2.x Standard in sämtlichen Diagrammarten, auch beispielsweise *SysML*, *BMM* und *BPMN*. Dadurch, dass Papyrus in die Eclipse Umgebung eingebettet ist, steht eine Vielzahl von Erweiterungen für spezifische Anwendungsfälle zur Verfügung. Beispiele dafür sind *Papyrus RT* zum Modellieren von Echtzeitsystemen und *Papyrus for Robotics*, welches für die Entwicklung von Robotersystemen optimiert ist. Weitere Informationen zu Papyrus können der offiziellen Website `www.eclipse.org/papyrus/` [PA19] entnommen werden. Eclipse Papyrus wird in der Fallstudie dieser Arbeit nicht verwendet. Das Tool besitzt zwar eine Vielzahl von Funktionen und Erweiterungen, doch ist für die korrekte Verwendung ein hoher Grad an Expertise sowie eine lange Einarbeitungszeit notwendig.

4.3 StarUML

StarUML [ST19] "[is a] sophisticated software modeler for agile and concise modeling" [ST19], welcher unabhängig der zuvor vorgestellten Eclipse Umgebung arbeitet.

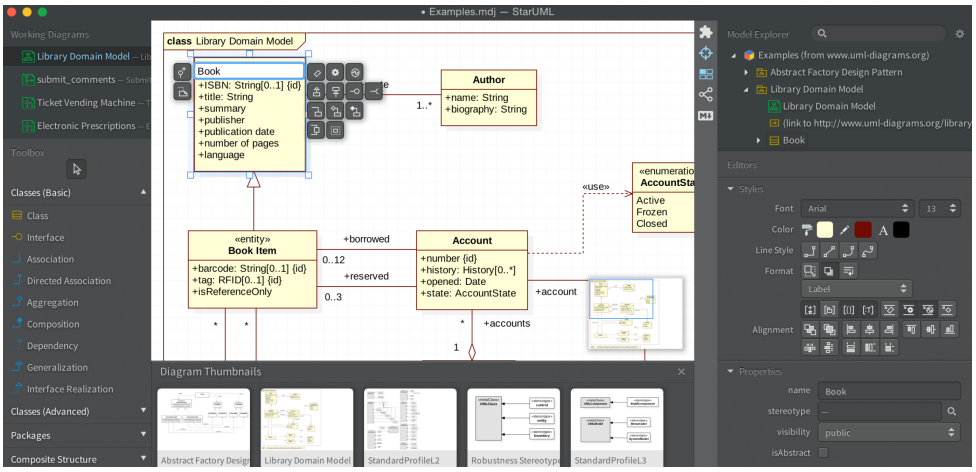


Abbildung 4.2: Screenshot des StarUML Modelliertools [ST19]

Die hier betrachtete Version 3 unterstützt den UML 2.x Standard in sämtlichen Diagrammarten. Das Modellformat, auf dem StarUML arbeitet, lautet .mdj, welches Modelle als *JSON* [JSO17] serialisiert. MDJ Modelle können jedoch auch zu XMI exportiert werden. Weitere Informationen und Dokumentationen zum Tool können der offiziellen Website `staruml.io` [ST19] entnommen werden. StarUML wird an dieser Stelle vorgestellt, da es in dieser Arbeit und insbesondere während der Fallstudie, als Tool der Wahl verwendet wird. Die entscheidenden Gründe dafür sind:

1. Das Tool besitzt eine intuitive Oberfläche, durch welche auch nach einer im Rahmen dieser Arbeit sehr kurzen Einarbeitungszeit, ein Großteil der Möglichkeiten des Tools eingesetzt werden kann.

2. Für das verwendete Format .mdj wird eine API *metadata-json* auf *npm* [NPM19] geboten, welche unter anderem in der Programmiersprache *TypeScript* [TS19] verwendet werden kann. Diese API ermöglicht einen vollständigen programmatischen Zugriff auf das Modell und bietet zusätzlich die Möglichkeit, das Modell via Queries, z.B. nach Elementen eines Stereotypes, zu durchsuchen.

5 Elaboration und Automation der Entwicklung von Mensch-Roboter Systemen

5.1 Beschreibung der Zielsysteme

1. Es gibt eine Menge von Arbeitsplätzen. Dabei unterscheiden sich menschzentrierte Arbeitsplätze und roboterzentrierte Arbeitsplätze.

2. An jedem Arbeitsplatz gibt es eine Menge von elektromechanischen Aktoren (z.B. Servomotor, Roboterarm, LED) und Sensoren (z.B. Näherungssensor, NFC-Reader, Drucktaster). Ebenfalls verfügt jeder Arbeitsplatz über einen Computer (Controller), welcher mit allen dazugehörigen Sensoren und Aktoren verbunden ist.

3. An einem menschzentrierten Arbeitsplatz ist ein Mensch dauerhaft anwesend, von welchem Aktionen gegen die Sensoren ausgehen und welcher Feedback von den Aktoren erhält. Menschen können über Interaktionen das Verhalten des Systems vorgeben. Dabei wird der Steuerbefehl des Menschen von Sensoren aufgenommen und über den Controller dem System zugeführt.

4. An einem roboterzentrierten Arbeitsplatz ist kein Mensch dauerhaft anwesend, sondern lediglich zur Wartung und Fehlerbehebung. Ein solcher Arbeitsplatz führt Steuerbefehle aus. Dabei richtet sich die Arbeit der verbundenen Aktoren gegen ein Werkstück. Die verbundenen Sensoren liefern Feedback zum Verhalten der Aktoren sowie dem Zustand des Werkstücks.

5. Die Controller aller Arbeitsplätze sind über ein Netzwerk miteinander verbunden und können so miteinander kommunizieren. Menschzentrierte Arbeitsplätze senden in erster Linie Steueraktionen in das Netzwerk und erhalten Feedback. Roboterzentrierte Arbeitsplätze erhalten in erster Linie Steueraktionen und senden Feedback zu ihrer Tätigkeit in das Netzwerk.

6. Ziel des Systems soll es sein, durch menschliche Interaktionen an den menschzentrierten Arbeitsplätzen, das Verhalten der Aktoren an den roboterzentrierten Arbeitsplätzen vorzugeben. Dabei muss ein Protokoll beachtet werden, welches Steuerungsregeln definiert und vom System eingehalten werden muss.

5.2 Entwicklung einer plattformunabhängigen Architektur und eines domänenspezifischen Programmiermodells

Nachdem die Menge der Zielsysteme im vorherigen Abschnitt spezifiziert wurde, erfolgt nun die Einführung einer plattformunabhängigen Architektur sowie einer domänen-spezifischen Sprache (DSL), um Modelle dieser Architektur zu erstellen. Die Eigenschaften der Zielsysteme können hierbei direkt als Anforderungen an die Architektur betrachtet werden. Das Beschreiben einer Architektur geht i.d.R unmittelbar mit den Entscheidungen zur Modellierung einher. So ist es auch in dieser Arbeit der Fall. Daher werden beide Konzepte in diesem Abschnitt gemeinsam eingeführt. Wir verzichten hierbei weitgehend auf das Zeigen von Beispielen, da dies in unserer Fallstudie (Kapitel 6) sehr detailliert erfolgen wird. Um den Zusammenhang der hier vorgestellten Konzepte zu verdeutlichen, folgt eine vereinfachte Übersicht in Abschnitt 5.2.2.

Die Funktionsweise der Zielsysteme basiert auf dem Erteilen von Steuerbefehlen, dem Reagieren auf Feedback und der Einhaltung eines Protokolls aus Geschäftsregeln. Auf Grund dieser Eigenschaften wird als zentrales Architekturmuster eine Event-Condition-Action (ECA) Architektur gewählt. Innerhalb der Architektur treten einzelne Arbeitsplätze gleichzeitig als Event-Quellen und Aktions-Senken auf. Als Netzwerkarchitektur des verteilten Systems der einzelnen Arbeitsplätze, eignet sich eine Client-Server Architektur. Der zentrale Server erfüllt hier zwei Funktionen. Zum Einen dient er als Broker, um Nachrichten zwischen den Arbeitsplätzen zuzustellen. Die zweite Aufgabe besteht im Synchronisieren der Zustandsinfomationen der einzelnen Arbeitsplätze, sodass Abläufe innerhalb eines Arbeitsplatzes auch Abhängigkeiten zu Fremdzuständen besitzen können.

Um die optimale Modellierung des Zielsystems zu ermöglichen, wird eine DSL benötigt. Hierzu wird, auf der UML aufbauend, ein Profil definiert. Ein solches Profil ermöglicht die Beibehaltung der bekannten UML Diagrammstruktur, als auch eine Spezialisierung und Erweiterung der UML Elemente auf die hier behandelte Domäne. An dieser Stelle wird ebenfalls von der normalsprachlichen Beschreibung zu domänenspezifischen Begriffen gewechselt: Arbeitsplatz → *Workstation*; menschzentrierter Arbeitsplatz → *HumanWorkstation*; roboterzentrierter Arbeitsplatz → *RoboticWorkstation*; Server/Broker → *Node*; Nachricht → Event|Action.

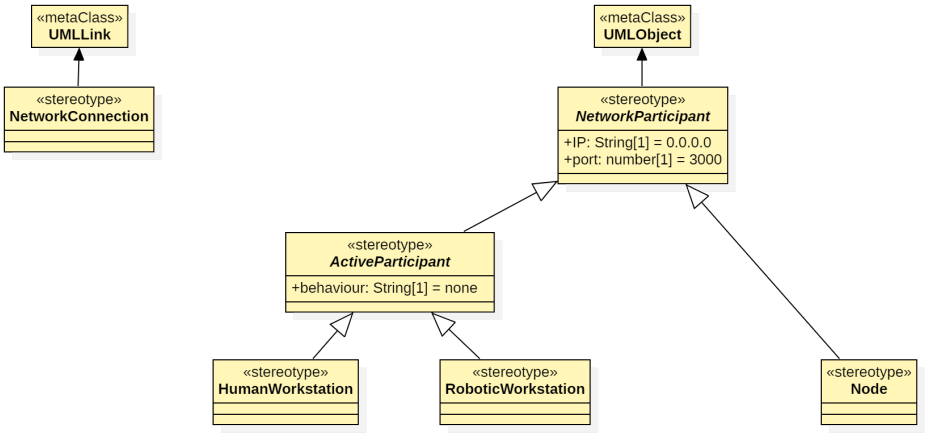
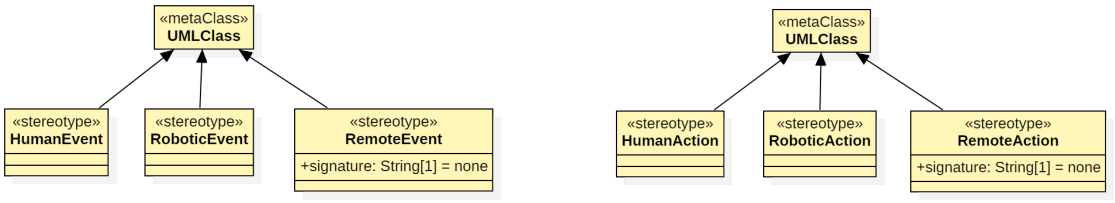


Abbildung 5.1: Profildiagramm zur Modellierung der Netzwerktopologie

Abbildung 5.1 zeigt den auf dem UML Objektdiagramm aufbauenden Teil des Profils, welchen wir nutzen, um die Netzwerktopologie des Zielsystems zu modellieren. Aus diesem Profildiagramm geht die Entscheidung hervor, dass es in der NetzwerkArchitektur drei Arten von Teilnehmern geben wird: *Nodes*, *HumanWorkstations* und *RoboticWorkstations*. Dabei besitzen alle Teilnehmer eine IP und einen Port. *HumanWorkstations* und *RoboticWorkstations* besitzen zusätzlich ein Verhalten (*Behaviour*). Die Modellierung des *Nodes* und der damit einhergehen-

den *NetworkConnections* ist auf Grund unserer Architektur redundant, da es in jedem Fall genau einen *Node* geben kann und muss. Wir wollen den *Node* jedoch trotzdem explizit in das Modell aufnehmen. Zum Einen verdeutlicht er die Semantik der Topologie, andererseits werden dadurch mögliche Erweiterungen der Architektur und DSL mit mehreren *Nodes* vereinfacht.



(a) Profildiagramm der System-Events

(b) Profildiagramm der System-Actions

Abbildung 5.2: Event und Action Profildiagramme zur Modellierung von ECA-Regeln

Zur Umsetzung der Geschäftsprozesse mittels ECA-Regeln werden, wie in den Systemanforderungen beschrieben, spezielle Events und Actions benötigt. Diese werden in Abb. 5.2.a und Abb. 5.2.b als Ausprägungen der UML-Class definiert. *HumanAction*: Nachricht an den menschlichen Bediener einer *HumanWorkstation*; *HumanEvent*: Interaktion/Eingabe eines Menschen in einer *HumanWorkstation*; *RoboticEvent*: Feedback von Sensoren innerhalb einer *RoboticWorkstation*; *RoboticAction*: Befehl an die Hardware innerhalb einer *RoboticWorkstation*; *RemoteEvent/RemoteAction*: Events und Actions, welche teilnehmerübergreifend erfolgen. Emmitiert ein Teilnehmer eine *RemoteAction*, wird diese in ein *RemoteEvent* umgewandelt und dem adressierten Teilnehmer zugestellt. Dazu müssen Event und Action ihrem Gegenüber zugeordnet werden können. Dies ermöglicht eine Erweiterung in Form einer Signatur.

Mit Hilfe der bisher spezifizierten Teile des Profils, ist es bereits möglich, die statische Systemsicht vollständig zu modellieren. Durch die entwickelte Spezialisierung eines Objektdiagramms kann die Topologie der Netzwerkteilnehmer spezifiziert werden. Durch die Verwendung der Event- und Action-Klassen können anwendungsfallspezifische Events und Actions definiert werden.

Um die dynamische Systemsicht zu modellieren, haben wir eine Spezialisierung von UML-Statecharts gewählt. Diese Entscheidung ist zum Einen durch die Eigenschaft der Teilnehmer bestimmt, da diese ein zustandsbehaftetes Verhalten besitzen. Ebenfalls ermöglichen Statecharts eine sehr gute Modellierung der ECA Architektur, da Transitionen zwischen Zuständen bereits Trigger-Events (hier Events), Guards (hier *Conditions*) und Opaque-Behaviour (hier Actions) besitzen. Unsere domänenspezifischen Statecharts unterscheiden sich von allgmeinen UML-Statecharts durch die Nutzung eines State-Profils sowie des Zwanges zu definierten Modellierungsregeln. In Abb. 5.3 ist das UML Profil der State-Erweiterung dargestellt. Dabei definiert das Profil mehrere mögliche Arten von Zuständen. *HumanState*: Zustände der *HumanWorkstation*; *BusyState*: Zustand einer *RoboticWorkstation* wenn diese gerade eine Aufgabe ausführt; *ReadyState*: Zustand einer *RoboticWorkstation*, wenn diese bereit für eine neue Aufgabe ist; *ErrorState*: Zustand einer *RoboticWorkstation*, wenn ein Fehler aufgetreten ist.

5.2.1 Regeln zur domänenspezifischen Modellierung

Um ein spezielles System mit Hilfe der zuvor definierten Profile zu modellieren, müssen die folgenden Regeln eingehalten werden. Diese Regeln schränken die Möglichkeiten der Modellierung weiter ein, ermöglichen jedoch ein hohes Maß an automatisierter Verwendung des Modells. Werden die Regeln verletzt, können bei der Prorammgenerierung Fehler auftreten oder es wird ein System mit undefiniertem Verhalten generiert.

1 Es gibt genau ein Objektdiagramm, welches die Topologie des Netzwerks definiert. Jedes verwendete UML Element muss dabei einen Stereotype, entsprechend des Topologie-

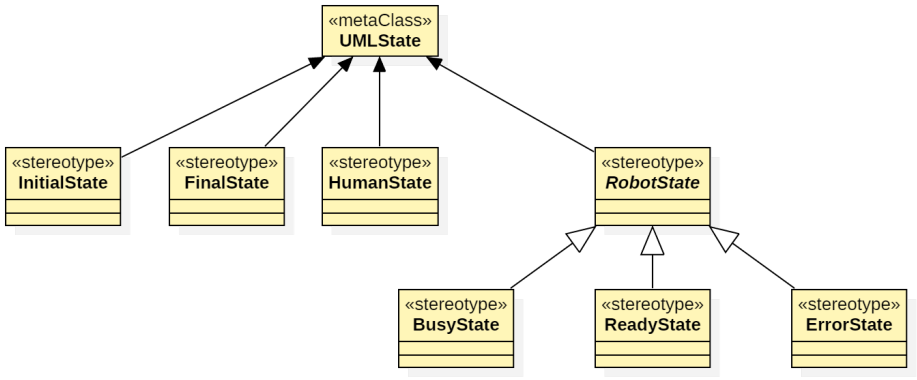


Abbildung 5.3: Profildiagramm der Zustände der Netzwerkteilnehmer

Profildiagramms, besitzen. Durch eine *NetworkConnection* dürfen hierbei nicht zwei *ActiveParticipants* miteinander verbunden werden. Es darf nur einen *Node* geben.

2 Es gibt genau ein Klassendiagramm, welches eigene Spezialisierungen der Action Klassen enthält. Jede Klasse muss dabei einen Stereotype, entsprechend des Action-Profildiagramms, besitzen. Eine Action darf hierbei keine Methoden besitzen und alle Parameter müssen public sein. Es dürfen als Datentypen nur *string*, *number*, *boolean* und *Array<string/number/boolean>* verwendet werden.

3 Es gibt genau ein Klassendiagramm, welches eigene Spezialisierungen der Event Klassen enthalten kann. Jede Klasse muss dabei einen Stereotype, entsprechend des Event-Profildiagramms, besitzen. Ein Event darf hierbei keine Methoden besitzen und alle Parameter müssen public sein. Es dürfen als Datentypen nur *string*, *number*, *boolean* und *Array<string/number/boolean>* verwendet werden.

4 Zu jeder *RemoteAction* muss es ein *RemoteEvent* geben, welches die gleichen Parameter besitzt und die gleiche Signatur hat. Dies gilt auch umgekehrt.

5 Zu jedem *Behaviour*, welches in der Topologie angegeben ist, muss es ein Statechart-Diagramm geben. Die Statecharts dürfen folgende UML Elemente beinhalten: *Simple States*, *Initial/Final States* und Transitionen. Die Regeln 6 bis 11 definieren dazu weitere Details.

6 Jeder State muss einen der im Profil definierten Stereotypen besitzen.

7 Es muss einen für die Statechart globalen *InitialState* geben. Die Transition, welche vom *InitialState* ausgeht, darf kein weiteres Verhalten (*Trigger Event*, *Opaque Behaviour*, *Guard*) besitzen.

8 Eine Transition kann ein *Trigger Event* (Regel 9) besitzen. Ebenfalls kann ein *Guard* (Regel 10) und ein *Opaque Behaviour* (Regel 11) spezifiziert werden.

9 Ein *Trigger Event* muss immer den Namen einer Event-Klasse des Modells tragen (entsprechend Regel 3).

10 Ein *Guard* kann einen booleschen Ausdruck erhalten. Dabei kann auf die Parameter des vorangegangenen Events in der Form *e.PARAMETERNAME* zugegriffen werden. Es dürfen ebenfalls die Stati anderer Workstations in der Form *NAME.READY/BUSY/ERROR* verwendet werden, sowie globale Quantoren der Form *all.READY/BUSY/ERROR* und *one.READY/BUSY/ERROR*.

11 Ein *Opaque Behaviour* muss immer den Namen einer Action-Klasse des Modells tragen (entsprechend Regel 4). Einer Action müssen die Parameter entsprechend ihres Modells mitgegeben werden. Dies erfolgt, in Abweichung zum UML-Standard, in runden Klammern textuell hinter der Action in der Form *(ACTIONNME(PARAM1, ...))*. Handelt es sich um eine *RemoteAction*, muss ebenfalls ein *target* mit dem Namen des Ziels als erster Parameter angegeben werden. Dabei kann, wie in Regel 10, auf die Parameter der Events und die Stati anderer Workstations zugegriffen werden.

12 Das Modell muss mit dem Tool *StarUML* und der dafür von uns bereitgestellten Vorlage erstellt werden.

5.2.2 Systemübersicht

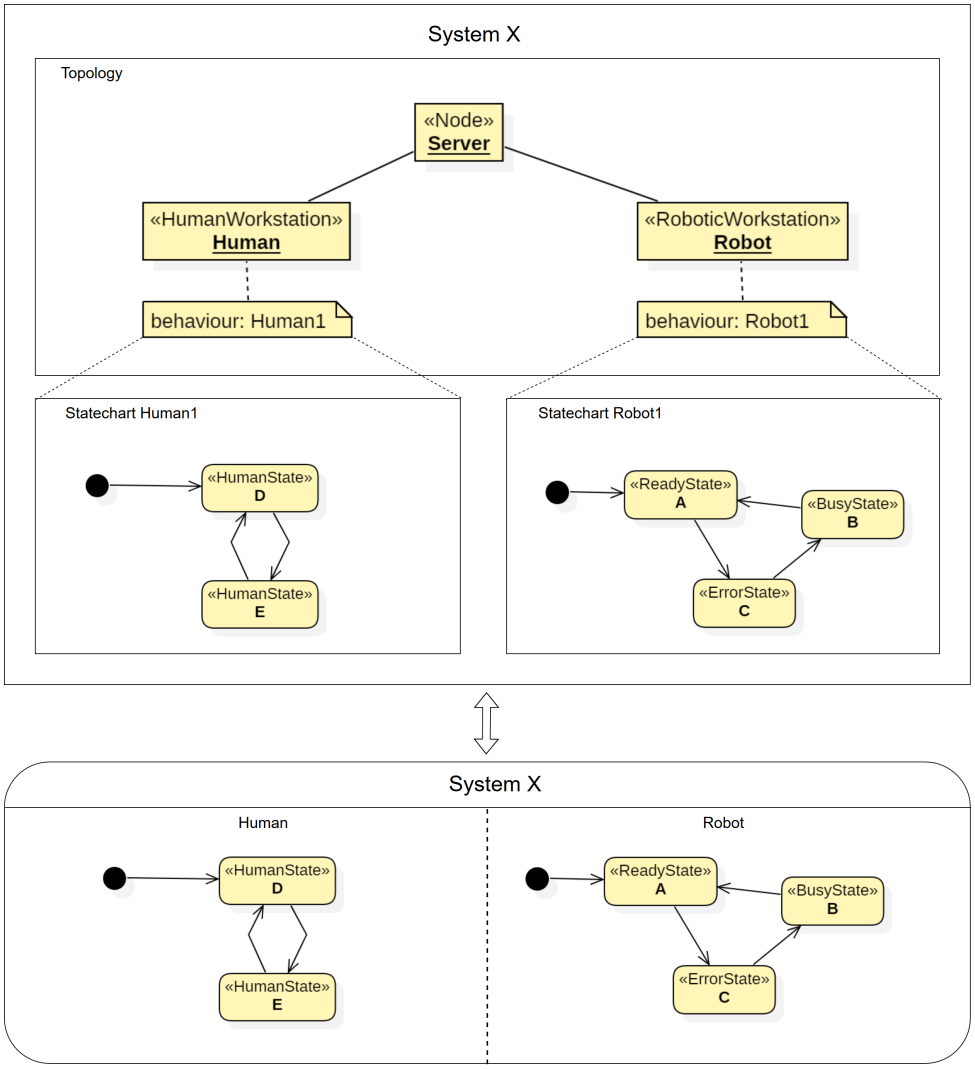


Abbildung 5.4: Vergleich eines Beispielmodells zur Darstellung mit Orthogonal-States

Abbildung 5.4 zeigt eine vereinfachte Übersicht eines Beispielsystems X. Dieses System besteht aus einer *HumanWorkstation* mit dem Namen *Human* und einer *RoboticWorkstation* mit dem Namen *Robot*. Beide *Workstations* sind mit einem zentralen *Node* verbunden. Jeder der *Workstations* besitzt ebenfalls ein spezifiziertes Verhalten. In der Darstellung wurde auf Events, Actions und Metadaten wie IP-Adressen verzichtet.

Im unteren Teil der Abb. wurde das System auf eine klassische Weise modelliert. Dabei gibt es für jedes Teilsystem eine Statechart, welche sich innerhalb eines übergeordneten Orthogonal-

State befinden. Die Teilsysteme laufen darin parallel und können miteinander wechselwirken. Diese Darstellung besitzt aus unserer Sicht jedoch einige Nachteile. So geht die Semantik eines verteilten Systems verloren, da diese Darstellung in erster Linie für parallele Subsysteme genutzt wird. Im Fall dieser Arbeit handelt es sich jedoch nicht nur um einzelne Subsysteme, sondern um eigenständige Anwendungen, welche nur indirekt miteinander kommunizieren können. Weiterhin ist diese Form ungeeignet, kollaborativ bearbeitet zu werden, da ein einzelnes Diagramm mit vielen Abhängigkeiten innerhalb des Entwicklerteams versioniert werden muss.

Der obere Teil der Abb. zeigt das System, wie wir es in dieser Arbeit modellieren wollen. Dazu ist das Verhalten jedes Teilsystems in eine eigene Statechart gekapselt. Wechselwirkungen können aber entsprechend der DSL trotzdem realisiert werden. Die so existierenden einzelnen Statecharts werden hier über eine Topologie in Beziehung gesetzt, wodurch die Semantik und Architektur des verteilten Systems verdeutlicht wird. Ebenfalls ist das Modell nun gut kollaborativ bearbeitbar, da einzelne Teilsysteme in eigenen Diagrammen getrennt modelliert werden können. Die Semantik des Systems und insbesondere der Statecharts kann dem Abschnitt 5.3.1 *Zielplattform* entnommen werden.

5.3 Spezifikation des Technologie Mappings

Im vorangegangenen Abschnitt wurde erklärt, wie unter Nutzung eines UML Profils sowie der Einhaltung von Modellierungsregeln, Modelle der Zielmenge erstellt werden können. In diesem Abschnitt beschreiben wir nun, welche Zielplattform wir gewählt haben und wie vom Modell auf ein konkretes System dieser Plattform abgebildet wird.

5.3.1 Zielplattform

Die Wahl einer Zielplattform hängt maßgeblich von den Anforderungen an diese ab. Um die optimale Wahl der Plattform treffen zu können, wird im Produktiveinsatz eine detaillierte Anforderungsanalyse durchgeführt. Da sich unsere Arbeit jedoch auf eine konzeptionelle Umsetzung konzentriert, soll an dieser Stelle darauf verzichtet werden. Unsere selbstgewählten Anforderungen an die Plattform lauten wie folgt:

1. Keine Bindung an ein bestimmtes Betriebssystem.
2. Eine einfache Bereitstellung und Inbetriebnahme.
3. Gute Unterstützung in Hinblick auf Frameworks und Treiber.
4. Möglichkeiten zur Einbindung bestehender Systemteile.

Bei der Umsetzung der beschriebenen Zielplattform nehmen wir keine Rücksicht auf Kriterien bezüglich der verfügbaren Harware. So wird von der Existenz eines üblichen Desktop-PC mit einem aktuellen Windows oder Linux Betriebssystem an jeder *Workstation* ausgegangen.

Um die von uns priorisierten Anforderungen optimal umsetzen zu können, haben wir uns für die konkrete Zielplattform TypeScript mit *NodeJS* [NJ19] entschieden. NodeJS ist eine Plattform zur serverseitigen Ausführung von JavaScript Anwendungen. Dadurch bedingt hat NodeJS einen Fokus auf ereignisorientierte Netzwerkanwendungen, welche mit einer Vielzahl von Frameworks unterstützt werden. TypeScript ist eine von Microsoft entwickelte Programmiersprache, welche zu JavaScript kompiliert wird und damit auf NodeJS ausführbar ist. Im Gegensatz zum nativen JavaScript besitzt TypeScript ein Typsystem und standardmäßig alle Voraussetzungen zur objektorientierten Programmierung, aber auch nützliche funktionale Eigenschaften. NodeJS ist für die Betriebssysteme OSX, Windows und Linux verfügbar, darum wird die freie Wahl eines Betriebssystems aus Anforderung 1 bereits erfüllt. Weil es sich bei Typescript um eine Skriptsprache handelt, können damit entwickelte Programme in textueller Form ausgeliefert werden.

Der TypeScript Compiler ist ebenfalls auf NodeJS ausführbar, wodurch die ausgelieferten TypeScript Dateien direkt auf ihrer Zielplattform vor dem Ausführen kompiliert werden können. Hiermit wird die in Anforderung 2 geforderte einfache Bereitstellung und Inbetriebnahme gewährleistet. Auf Grund der Kompatibilität von TypeScript zu JavaScript, kann eine mit Typescript entwickelte Anwendung sämtliche JavaScript Bibliotheken nutzen. Durch einen Paketmanager wie NPM steht so eine Vielzahl von Bibliotheken und Frameworks zur Verfügung. Bestehende Systemteile die in die Anwendung eingebunden werden müssen, sind oft auf hardwarenahen Sprachen entwickelte APIs und Steuerungen. Deren Anpassung soll mit möglichst geringem Aufwand erfolgen. Hierzu wird die Möglichkeit von NodeJS genutzt, auf sehr einfache Weise Sockets zur Kommunikation zu nutzen. So können Informationen mit bestehenden Systemteilen über Sockets in Form von standardisierten Nachrichtenformaten ausgetauscht werden.

5.3.2 Abbildung zwischen Modell und Zielsystem

In diesem Abschnitt wird skizziert, wie die modellierten Systemeigenschaften auf Programmcode abgebildet werden können. Auf Details der Implementation wird dabei verzichtet. Diese können im bereitgestellten Quellcode der Fallstudie und des Generators genauer betrachtet werden.

Topologie

Ein modelliertes Gesamtsystem besteht aus einer Vielzahl von *Workstations* und einem *Node*, welche in der Topologie beschrieben werden. Dabei soll es möglich sein die Teilanwendung jeder *Workstation* auf dem jeweilig dazugehörigen Computer zu installieren. Um dies zu ermöglichen, wird für jeden Teilnehmer eine eigene Anwendung erstellt, welche anschließend auf dem jeweiligen Computer gestartet wird. Um den Nachrichtenaustausch zu gewährleisten, muss zwischen jeder *Workstation* und dem verbundenen *Node* eine bidirektionale Netzwerkverbindung hergestellt werden. Dies wird über die Nutzung von Sockets erreicht. Der zentrale *Node* wird dabei als Socketserver und die einzelnen *Workstations* als Socketclients umgesetzt. Da die IP-Adressen und Ports der Teilnehmer im Modell angegeben werden müssen, verfügen alle Clients über die nötigen Informationen, mit welchem Server sie sich automatisch verbinden müssen und der Server besitzt die Information, über welche Verbindung ein bestimmter Client angesprochen werden kann.

Events und Actions

Events und Actions werden im Modell als Klassen mit bestimmten Parametern modelliert. Dies kann in das Zielsystem direkt übernommen werden, sodass die Datenstruktur jedes modellierten Events und jeder Action auf eine gleichartige Klasse im Programm abgebildet werden kann. Die Umsetzung der Semantik von Events und Actions wird im folgenden Abschnitt im Kontext der Statecharts beschrieben.

Verhalten

Jede *Workstation* des Modells besitzt eine Statechart, welche deren Verhalten modelliert. Um dieses Verhalten auf eine Anwendung abzubilden, wird das State-Pattern [GA05] verwendet. Dabei repräsentiert ein Zustand im Modell eine gleichnamige Instanz einer Zustandsvariable der Anwendung. Anhand des Wertes der Zustandsvariable, kann das System bei einem eintretenden Event entscheiden, wie es darauf reagieren muss. Erfolgt ein Zustandsübergang, wird der Zustandsvariable anschließend ihr neuer Wert zugewiesen. Die Statecharts der einzelnen Teilsysteme werden dabei asynchron ausgeführt. Sie folgen dabei jeweils der *Superstate*-Semantik [HA96]. Das heißt, eine Statechart führt so lange Transitionen aus, bis sie sich in einem Zustand befindet, der zum derzeitigen Zeitpunkt nicht verlassen werden kann. Eintretende Events können daraufhin jedoch eine neue Transitionskette auslösen (vgl. *HA96*).

Die Semantik von Events und Actions geht aus dem Modell nicht vollständig hervor. Durch Transitionen wird definiert, in welchem Systemzustand ein jeweiliges Event einen Zustandsübergang auslöst und welche Action dabei ausgeführt wird. Jedoch wird nicht modelliert, wie ein Event ausgelöst wird und welche Tätigkeit durch das Ausführen einer Action realisiert wird. Diese Eigenschaften müssen vom Entwickler außerhalb des Modells angegeben werden. Dazu gibt es für jeden *Workspace* eine eigene Schnittstelle, gegen welche Events ausgelöst werden können und Actions empfangen werden. Dies geschieht über eine Schnittstelle als TypeScript Template, welche vom Entwickler selbst spezifiziert wird.

5.4 Entwurf eines Programmgenerators

Zur automatisierten Umsetzung der Abbildung von einem Modell auf Programmcode wird ein Programmgenerator benötigt. Ein solcher Generator ist eine Anwendung, welche unter der Eingabe des Modells, Quellcode des Zielsystems erzeugt. In diesem Abschnitt wird ein solcher Generator für das hier behandelte Metamodell entworfen.

Zu Beginn benötigt der Generator drei Eingaben. Zum Einen handelt es sich dabei um das Modell als MDJ Datei. Die anderen beiden Eingaben sind Templatanwendungen. Dabei gibt es eine Templateanwendung für *Nodes* und eine weitere für *Workstations*. Hierbei wird zunächst nicht zwischen *Human-* und *RoboticWorkstations* unterschieden. Die zwei Templateprojekte enthalten bereits sämtlichen Glue-Code sowie die Klassen des Socketserver bzw. Cients, Helferklassen und Konfigurationsdateien. Alle Klassen, welche auf Grund fehlender Informationen aus dem Modell noch nicht vollständig implementiert werden können, besitzen Platzhalter. Zum Späteren ersetzen der Platzhalter durch Programmcode, kommt die *Mustache* [MU19] Template-Engine zum Einsatz. Die Templateanwendungen sollen sich bereits im Anwendungsordner des Generators befinden und von diesem aus statisch verlinkt werden, sodass der Nutzer lediglich die Datei des Modells bereitstellen muss.

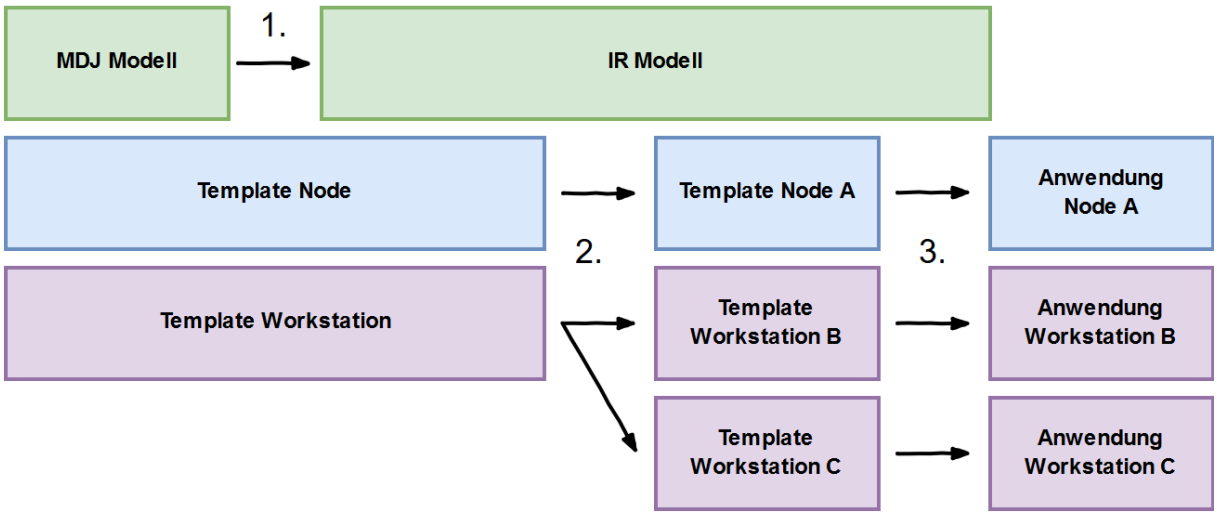


Abbildung 5.5: Veranschaulichung der Arbeitsschritte des Programmgenerators

Die Arbeitsweise des Generators wird an Abb. 5.5 veranschaulicht. Dabei handelt es sich um ein Modell mit einem *Node* A und zwei *Workstations* B und C. Die übergeordneten Arbeitsphasen 1.-3. werden im Folgenden beschrieben:

1. Die von StarUML erzeugte MDJ Datei besitzt neben den vom Generator benötigten Informationen zum Modell auch eine Vielzahl anderer Angaben wie Dokumentationen und Layout der Diagramme. Um die Logik des Auslesens der MDJ Datei von der Codegenerierung zu trennen, werden die nötigen Modellinformationen in eine Intermediate-Representation

(IR) überführt. Diese Repräsentation ist eine auf dem verwendeten UML Profil basierende objektorientierte Datenstruktur.

2. Auf Basis der Informationen zur Topologie, werden die Templateanwendungen geklont und umbenannt, sodass es anschließend für jeden Teilnehmer des Netzwerks eine eigene Templateanwendung gibt, die dessen Namen trägt. In diesem Schritt werden ebenfalls alle Variablen der Templates ersetzt, deren Werte topologieabhängig sind. Solche Variablen sind z.B. Namen anderer Teilnehmer oder IP-Adressen.

3. In diesem Schritt wird für jede erzeugte Anwendung der noch fehlende Programmcode auf Basis des Modells generiert. Dazu gehören Event-Klassen, *Event/Action*-Schnittstellen und die Logik der Statechart. Der generierte Code wird anschließend in die Variablen der Templates eingesetzt, bzw. es werden für einige Klassen neue Dateien angelegt. Nach diesem Schritt sind die einzelnen Anwendungen ausführbar. Der Entwickler muss jedoch noch die generierten Schnittstellen manuell an die Steuerung der Hardware anpassen.

Im Rahmen dieser Arbeit haben wir einen eigenen Programmgenerator entwickelt, welcher das zuvor beschriebene Vorgehen umsetzt. Der Generator selbst wurde ebenfalls in TypeScript für die NodeJS Plattform entwickelt. Für Details zur Implementierung kann der Quellcode des Generators eingesehen werden.

6 Fallstudie Roboterkollaboration "50 Jahre Informatik"

In diesem Kapitel erfolgt die Modellierung und Generierung eines Beispielsystems, welches an die *Roboterkollaboration 50 Jahre Informatik* angelehnt ist und die Anwendung der im vorherigen Kapitel eingeführten Techniken demonstriert. Der Schwerpunkt dieser Fallstudie liegt auf dem Einsatz der modellgetriebenen Methode, um diese im Anschluss zu evaluieren.

6.1 Motivation

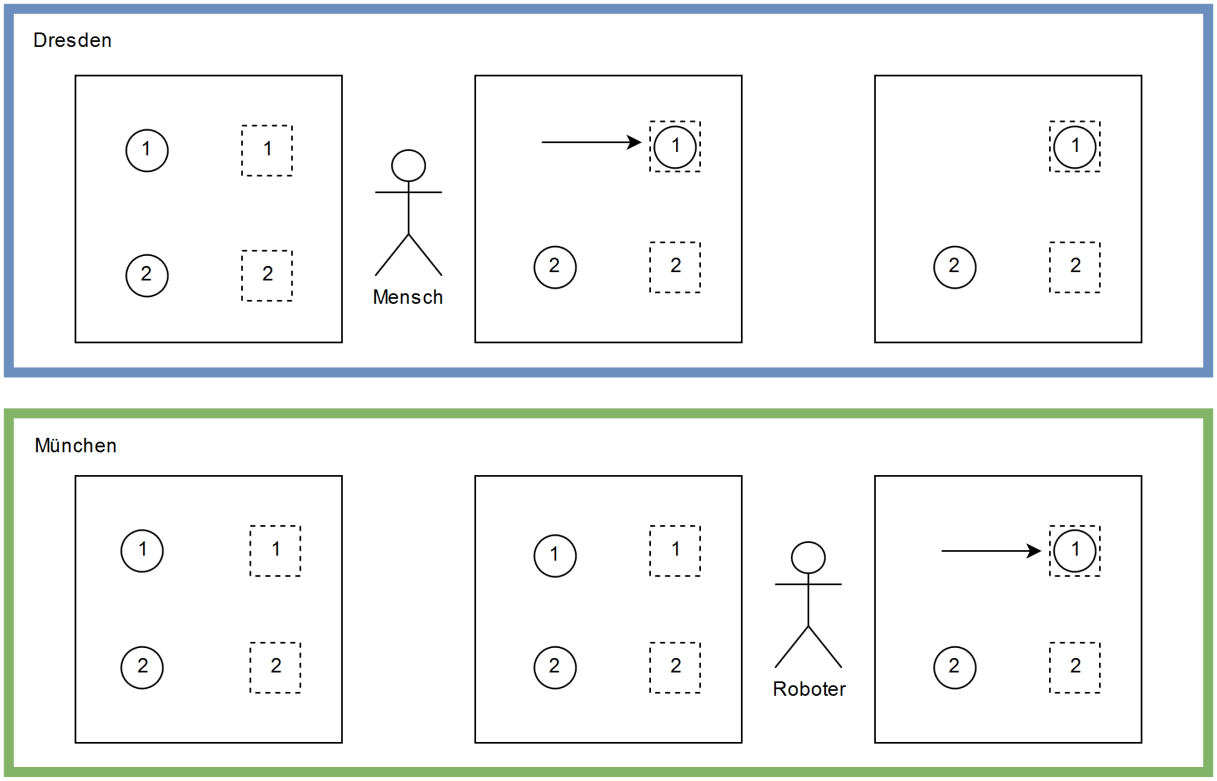


Abbildung 6.1: Darstellung eines Anwendungsfalls der Fallstudie

Die *Roboterkollaboration 50 Jahre Informatik* ist ein interaktives System zur Demonstration der deutschlandweit vernetzten Kollaboration zwischen Mensch und Maschine. Um dies zu ermög-

lichen, sind an mehreren Standorten innerhalb Deutschlands speziell ausgestattete Arbeitsplätze vorhanden. Solche Standorte befinden sich in Dresden, München, Saarbrücken, Karlsruhe, Berlin und Darmstadt. An jedem dieser Arbeitsplätze befindet sich ein Tisch mit mehreren markierten Zielpositionen und der gleichen Anzahl an Werkstücken. Jedes Werkstück liegt dabei zunächst an seiner Ausgangsposition. Unmittelbar neben jedem Tisch befindet sich ein Roboterarm, welcher ein Werkstück greifen und auf einer anderen Position absetzen kann. Nimmt nun ein Mensch ein Werkstück an einem der Arbeitsplätze in die Hand und bewegt es von seiner Ausgangsposition auf seine Zielposition, wird diese Bewegung an allen anderen Arbeitsplätzen vom jeweiligen Roboterarm imitiert, sodass die Position der Werkstücke stets synchron gehalten wird.

In Abbildung 6.1 ist ein solcher Ablauf vereinfacht an zwei Standorten dargestellt. Die Abbildung zeigt schematisch den Aufbau der Arbeitsplätze innerhalb von drei Zeitabschnitten. Im ersten Abschnitt (links) befinden sich die durch einen Kreis dargestellten Werkstücke an ihren Ausgangspositionen. Innerhalb des zweiten Abschnitts (mitte), bewegt der Mensch am Arbeitsplatz *Dresden* das Werkstück 1 auf seine Zielposition. Die Position des Werkstücks am Arbeitsplatz *München* bleibt während dieses Vorgangs unverändert. Nachdem zu Beginn des dritten Abschnitts (rechts) die Bewegung des Menschen abgeschlossen ist, führt der Roboterarm am Arbeitsplatz *München* die vom Mensch im vorherigen Schritt vorgegebene Positionsveränderung des Werkstücks selbst durch.

6.2 Spezifikation von Aufbau und Anwendung

Wie im vorherigen Abschnitt bereits beschrieben wurde, besitzt das hier betrachtete System eine Vielzahl von Standorten innerhalb Deutschlands. Um die nun folgende Modellierung des Systems übersichtlich zu gestalten, werden dabei zu Beginn nur die Standorte in Dresden und München berücksichtigt. Darauf, wie das so modellierte System auf die noch fehlenden Standorte erweitert werden kann, wird in Abschnitt 6.5 darauf folgend erläutert.

Auf Grund des gewählten Aufbaus der Fallstudie, wird auf eine umfassende Projektspezifikation in Form eines Pflichtenheftes verzichtet. Stattdessen werden alle zur Modellierung nötigen Informationen zum Aufbau des Systems und seinen Anwendungsfällen angegeben.

6.2.1 Aufbau

Der Aufbau der Arbeitsplätze in Dresden und München erfolgt identisch und wird nun anhand eines einzelnen Standortes detailliert beschrieben. Des Weiteren erfolgt die Beschreibung aus zwei verschiedenen Sichten auf das System. Zum Einen aus der Sicht des menschlichen Bedieners, als zum Anderen auch aus der Sicht der robotischen Anwendung. Jede Sicht enthält dabei nur die technischen Komponenten, welche für diese relevant sind. Dies ermöglicht eine insgesamt einfachere und aussagekräftigere Analyse des Systems.

1. (Mensch) Es gibt einen Arbeitsplatz, welcher von einem Menschen bedient wird. An diesem Arbeitsplatz befindet sich ein Tisch. Auf dem Tisch gibt es je zwei markierte Start- und Zielpositionen. An jeder Zielposition befindet sich ein NFC-Reader. An jeder Startposition liegt ein Werkstück, welches mit einem NFC-Tag versehen ist. Auf dem Tisch befindet sich zusätzlich eine rote (L1) und eine gelbe Lampe (L2), sowie ein Drucktaster (D1). Die NFC-Reader, Lampen und der Taster sind mit dem sich am Arbeitsplatz befindlichen Computer verbunden. Der Computer des Arbeitsplatzes ist mit dem Internet verbunden, es läuft ein aktuelles Linux Betriebssystem und es ist über einen Zugang möglich, Software zu installieren und auszuführen. Die IP-Adresse sowie freie Netzwerkports des Computers sind bekannt.

2. (Roboter) Es gibt einen Arbeitsplatz, an dem sich ein Roboterarm befindet. An diesem Arbeitsplatz befindet sich ein Tisch. Auf dem Tisch gibt es je zwei markierte Start- und Zielpositionen. An jeder Zielposition befindet sich ein NFC-Reader. An jeder Startposition liegt ein

Werkstück, welches mit einem NFC-Tag versehen ist. Der Roboterarm befindet sich unmittelbar neben dem Tisch, er kann ein Werkstück greifen, anheben und bewegen. Die Bewegungskurven des Roboterarms liegen für jeden Positionsübergang vor. Auf dem Tisch befindet sich eine rote Lampe (L3) und ein Drucktaster (D2). Der Roboterarm, der Taster und die Lampe sind mit dem sich am Arbeitsplatz befindlichen Computer verbunden. Der Computer des Arbeitsplatzes ist mit dem Internet verbunden, es läuft ein aktuelles Linux Betriebssystem und es ist über einen Zugang möglich, Software zu installieren und auszuführen. Die IP-Adresse sowie freie Netzwerkports des Computers sind bekannt.

Zum Aufbau des Systems stehen zusätzlich eine beliebige Zahl weiterer Computer zur Verfügung. Jeder dieser ist mit dem Internet verbunden, es läuft ein aktuelles Linux Betriebssystem und es ist über einen Zugang möglich, Software zu installieren und auszuführen. Die IP-Adresse sowie freie Netzwerkports der Computer sind bekannt.

6.2.2 Anwendungsfälle

Nachdem der statische Aufbau des Systems beschrieben wurde, folgt nun die Beschreibung der Anwendungsfälle, welche das System ermöglichen soll. Hierbei werden positive und negative Pfade als seperate Anwendungsfälle betrachtet.

ID	AH01
Sicht	Mensch
Vorbedingung	Es leuchtet keine der beiden Lampen (L1, L2). Die Roboter sind bereit, eine Aktion auszuführen.
Auslöser	Ein NFC-Reader meldet ein neues Werkstück an einer Zielposition, da es vom Mensch bewegt wurde.
Folge	Es wird ein Steuerbefehl mit der neuen Position des Werkstücks an alle Roboter gesendet.

ID	AH02
Sicht	Mensch
Vorbedingung	Es leuchtet keine der beiden Lampen (L1, L2). Einer der Roboter ist mit dem Ausführen einer Aktion beschäftigt.
Auslöser	Ein NFC-Reader meldet ein neues Werkstück an einer Zielposition, da es vom Mensch bewegt wurde.
Folge	Die gelbe Lampe (L2) beginnt zu leuchten. Neue Positionsveränderungen an diesem Arbeitsplatz werden ignoriert.

ID	AH03
Sicht	Mensch
Vorbedingung	Auf Grund von AH02 leuchtet die gelbe Lampe (L2).
Auslöser	Die Bewegung des Werkstücks wurde rückgängig gemacht und mit dem Drücken des Tasters (L1) bestätigt.
Folge	Die gelbe Lampe (L2) leuchtet nicht länger. Positionsänderungen werden nicht länger ignoriert.

ID	AH04
Sicht	Mensch
Vorbedingung	Es leuchtet keine der beiden Lampen (L1, L2). Bei einem Roboter liegt ein Fehler vor.
Auslöser	Ein NFC-Reader meldet ein neues Werkstück an einer Zielposition, da es vom Mensch bewegt wurde
Folge	Die rote Lampe (L1) beginnt zu leuchten. Neue Positionsveränderungen an diesem Arbeitsplatz werden ignoriert.

ID	AH05
Sicht	Mensch
Vorbedingung	Auf Grund von AH04 leuchtet die gelbe Lampe.
Auslöser	Die Bewegung des Werkstücks wurde rückgängig gemacht und durch Drücken des Tasters bestätigt.
Folge	Die rote Lampe leuchtet nicht länger. Positionsänderungen werden nicht länger ignoriert.

ID	AR01
Sicht	Roboter
Vorbedingung	Der Roboter führt keine Aktion aus. Die Lampe (L3) leuchtet nicht.
Auslöser	Der Arbeitsplatz erhält über das Netzwerk den Steuerbefehl, ein Werkstück an seine Zielposition zu bewegen.
Folge	Der Roboter beginnt die Aktion auszuführen.

ID	AR02
Sicht	Roboter
Vorbedingung	Der Roboter führt auf Grund von AR01 eine Aktion aus.
Auslöser	Der Roboter legt das Werkstück ab. Der NFC-Reader meldet die neue Position des Werkstücks.
Folge	Der Roboter bewegt sich zurück in seine Ausgangsposition.

ID	AR03
Sicht	Roboter
Vorbedingung	Der Roboter führt auf Grund von AR01 eine Aktion aus.
Auslöser	Das Werkstück wird fehlerhaft bewegt oder NFC-Reader erkennt das Werkstück nicht.
Folge	Der Roboter bewegt sich zurück in seine Ausgangsposition. Die rote Lampe (L3) beginnt zu leuchten. Neue Steuerbefehle aus dem Netzwerk werden von der Anwendung ignoriert.

ID	AR04
Sicht	Roboter
Vorbedingung	Der Roboter befindet sich an seiner Ausgangsposition. Die rote Lampe leuchtet.
Auslöser	Die Position des Werkstücks wurde von einem Menschen korrigiert und durch Drücken des Taster bestätigt.
Folge	Die rote Lampe (L3) leuchtet nicht länger. Steuerbefehle aus dem Netzwerk werden nicht länger ignoriert.

6.3 Modellierung

Topologie

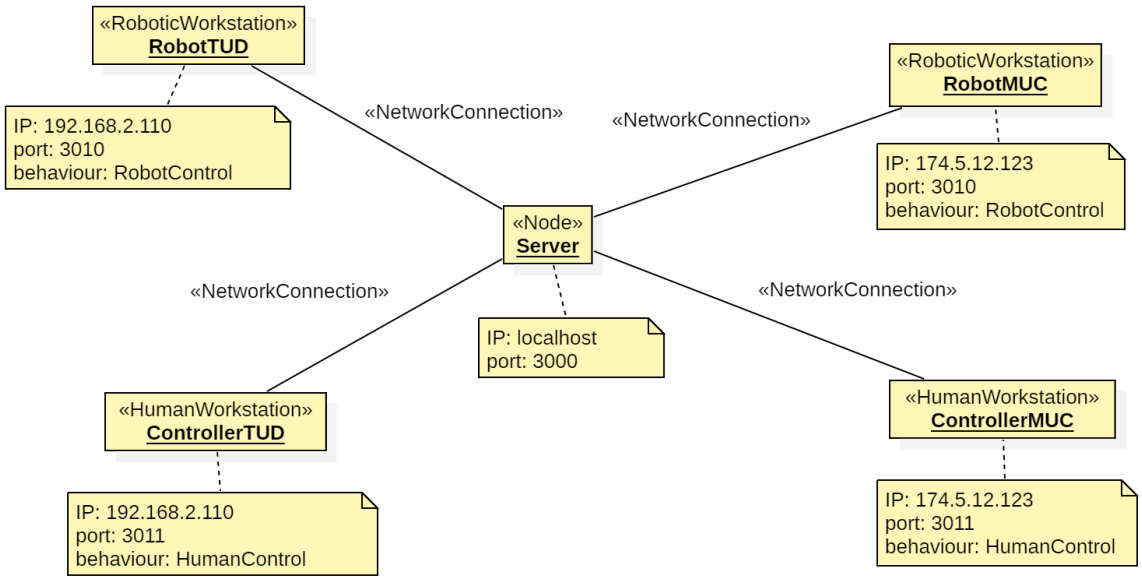


Abbildung 6.2: Topologie Diagramm des Modells, der hierzu berücksichtigten Standorte

Als erster Schritt kann aus der Beschreibung des Systems seine Netzwerktopologie abgeleitet werden. Wie im Abschnitt *Aufbau* beschrieben, gibt es im System zwei Standorte: Dresden und München. An jedem dieser Standorte gibt es einen physischen Arbeitsplatz mit einem Menschen und einem Roboter. Da es nach unserer DSL jedoch nur *HumanWorkstations* und *RoboticWorkstations* geben darf, werden die verschiedenen Sichten getrennt modelliert. So gibt es zwei Arbeitsplätze, welche von einem Menschen bedient werden und zwei Arbeitsplätze, welche in erster Linie von einem Roboter bedient werden.

Durch diese Betrachtung des Systems ist es nun möglich, die von Menschen bedienten Arbeitsplätze als *HumanWorkstations* und die von Robotern bedienten Arbeitsplätze als *RoboticWorkstations* zu modellieren. Die *HumanWorkstation* in Dresden erhält im Modell den Namen *ControllerTUD*, diejenige in München den Namen *ControllerMUC*. Die *RoboticWorkstation* in Dresden erhält im Modell den Namen *RobotTUD* und diejenige in München den Namen *RobotMUC*. Die bekannten IP-Adressen sowie Ports der Arbeitsplätze können ebenfalls direkt in das Modell übernommen werden. Dabei ist zu beachten, dass, da sich beide Sichten je einen Computer teilen, die IP-Adressen zwar gleich sind, aber verschiedene Ports gewählt werden. Zusätzlich soll jeder der Arbeitsplätze dem in den Anwendungsfällen spezifizierten Verhalten folgen. Das Verhalten der beiden *HumanWorkstations* soll gleich sein und wird als *HumanControl* bezeichnet.

Das Verhalten der *RoboticWorkstations* wird als *RobotControl* bezeichnet. Für beide Angaben des Verhaltens muss später im Modell eine gleichnamige Statechart angelegt werden. Auf Grund der gewählten Architektur, muss zusätzlich ein *Node* verwendet werden, mit dem die Arbeitsplätze über je eine *NetworkConnection* verbunden sind. Der *Node* erhält den Namen *Server*. Dieser kann hardwareseitig durch einen der zusätzlich zur Verfügung stehenden Computer realisiert werden. Die hier beschriebene Topologie ist in Abb. 6.2 als Objektdiagramm entsprechend der DSL dargestellt.

Events

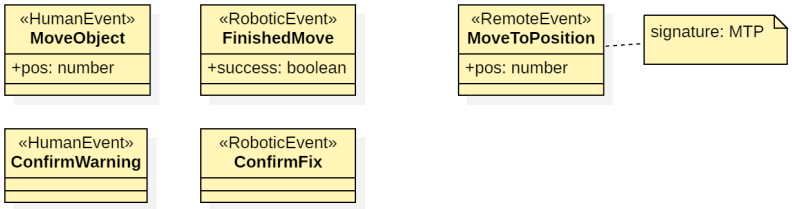


Abbildung 6.3: Event Diagramm des Modells

Die im System auftretenden Events können aus den Anwendungsfällen abgeleitet und modelliert werden. Der Ort des Auftretens eines Events entscheidet dabei, ob es als *HumanEvent*, *RoboticEvent* oder *RemoteEvent* modelliert werden muss. Events sind entsprechend unserer DSL immer Ereignisse, die einen Anwendungsfall auslösen. In Abbildung 6.3 sind die hier beschriebenen Events als Klassendiagramm, entsprechend der DSL, dargestellt. An den *HumanWorkstations* sind diese Ereignisse: das Melden der neuen Position des Werkstücks in (*MoveObject*) in AH01, AH02 und AH04; das Betätigen des Drucktasters (*ConfirmWarning*) in AH03 und AH05. Da beim Bewegen des Werkstücks dessen neue Position an die Anwendung übertragen wird, muss *MoveObject* ein Attribut *pos* vom Typ *number* erhalten, welches die Positionsdaten erhalten soll. An den *RoboticWorkstations* eintretende Ereignisse sind: die Meldung der neuen Position des Werkstücks nach der Bewegung (*FinishedMove*) in AR02 und AR03; das Betätigen des Drucktasters nach der Problembehebung (*ConfirmFix*) in AR04. Da beim Ereignis *FinishedMove* zwischen einem erfolgreichen und fehlerhaften Resultat unterschieden werden muss, erhält dieses Event ein Attribut *success* vom Typ *boolean*. Dieses Attribut ist *true*, wenn das Resultat erfolgreich war. Des Weiteren gibt es ein Ereignis in Form eines Steuerbefehls in AR01, zu welcher Position das Werkstück bewegt werden soll (*MoveToPosition*). Da dieses Ereignis aus dem Netzwerk stammt, muss es als *RemoteEvent* modelliert werden. Dieses Event trägt zusätzlich zum Attribut der Position *pos* des Typs *number*, eine Signatur, durch welche es einer entsprechenden Action zugeordnet werden kann.

Actions

Die im System auftretenden Actions sind diejenigen Aktionen oder Nachrichten, welche in den beschriebenen Anwendungsfällen als Folge eines Ereignisses das Verhalten des Systems verändern oder einen Befehl zur Verhaltensänderung auslösen. Gleichartig wie die Kategorisierung der Events, müssen auch Actions entsprechend unserer DSL in *HumanActions*, *RoboticActions* und *RemoteActions* eingeteilt werden. In Abbildung 6.4 sind die hier beschriebenen Actions als Klassendiagramm, entsprechend der DSL, dargestellt. Nach den Anwendungsfällen gibt es folgende *HumanActions*: das Anschalten der gelben Lampe (L2) mit Deakivierung der Positionserkennung (*WarnBusy*) in AH02; das Anschalten der roten Lampe (L1) mit Deaktivierung der Positionserkennung (*WarnError*) in AH04; das Ausschalten der leuchtenden Lampe mit Reaktivierung der Positionserkennung (*EndWarning*) in AH03 und AH05. Zusätzlich geht in AH01 von den

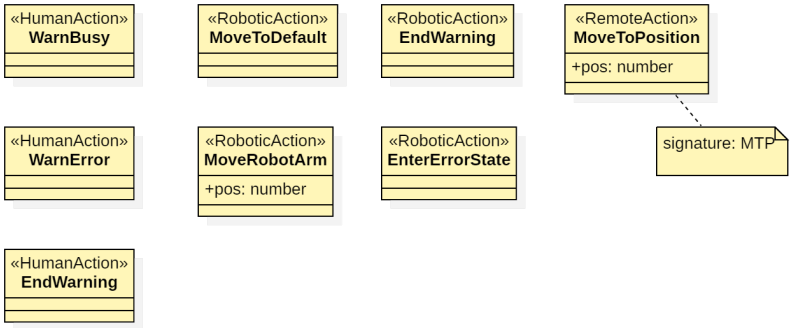


Abbildung 6.4: Action Diagramm des Modells

HumanWorkstations ein Steuerbefehl an die *RoboticWorkstations* aus, das Werkstück an eine andere Position zu bewegen. Diese Action wird deshalb als *RemoteAction* modelliert. Da sie an den *RoboticWorkstations* das gleichnamige Event *MoveToPosition* auslösen soll, wird hier die gleiche Signatur vergeben. Auf Seiten der *RoboticWorkstations* gibt es folgende *RoboticActions*: das Bewegen des Roboterarms zur Ausgangslage (*MoveToDefault*) in AR02; das Bewegen des Werkstücks zu einer Position (*MoveToPosition*) in AR01; das Anschalten der roten Lampe (L3) mit Ignorieren weiterer Steuerbefehle (*EnterErrorState*) in AR03; das Ausschalten der gelben Lampe mit Wiederaufnahme von Steuerbefehlen (*EndWarning*) in AR04. Die Actions *MoveToPosition* und *MoveRobotArm* benötigen zusätzlich noch ein Attribut des Typs *number* mit dem Namen *pos*, um die Positionsinformation übertragen zu können.

Verhalten

In der Netzwerktopologie gibt es zwei Arten von Verhalten, *HumanControl* und *RobotControl*. Im Modell werden daher auch zwei Statecharts benötigt. Eines, was das Verhalten einer *HumanWorkstation* beschreibt und eines, was das Verhalten einer *RoboticWorkstation* beschreibt. Alle für die Modellierung nötigen Informationen können hier ebenfalls den Anwendungsfällen entnommen werden. In der Statechart *HumanControl* werden entsprechend der Anwendungsfälle AH01-AH05 mehrere Zustände benötigt. Hier ist zu beachten, dass nach der DSL, jedem Zustand einer der Stereotypen *HumanState*, *InitialState* oder *FinalState* zugeordnet werden muss. Abbildung 6.5 zeigt die vollständige *HumanControl* Statechart entsprechend der DSL.

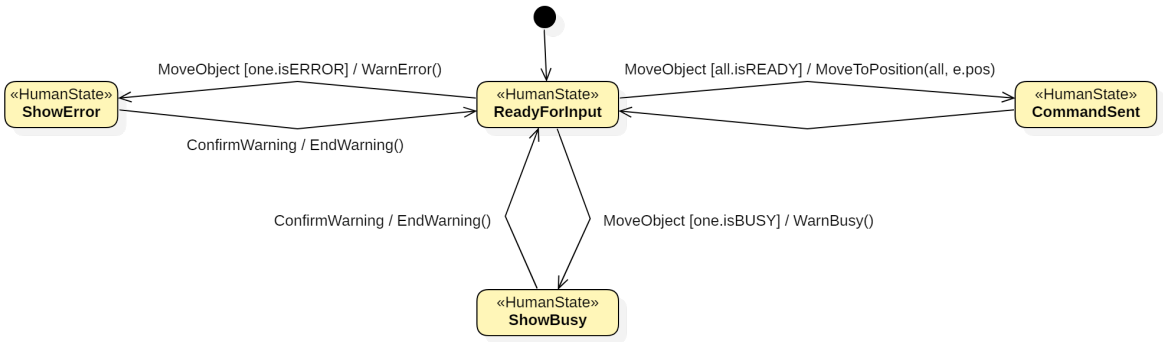


Abbildung 6.5: HumanControl Statechart des Modells

- *ReadyForInput*: es leuchtet keine der beiden Lampen (L1, L2), da bei einem Roboter ein Fehler vorliegt, als Vorraussetzung von AH01, AH02 und AH04.

- *ShowBusy*: es leuchtet die gelbe Lampe (L2) und Positionsveränderungen werden ignoriert, als Folge von AH02.
- *ShowError*: es leuchtet die rote Lampe (L1) und Positionsänderungen werden ignoriert, als Folge von AH04.
- *InitialState*: der Startzustand, von welchem aus bei Systemstart direkt in *ReadyForInput* gewechselt wird.
- *CommandSent* ein Hilfszustand. Dieser Zustand wird als Folgezustand in AH01 benötigt.

Nach Benennung der Zustände, können die Transitionen entsprechend der Anwendungsfälle AH01 bis AH05 definiert werden.

- *InitialState* → *ReadyForInput*: verhaltenslose Transition, welche nach dem Systemstart automatisch ausgeführt wird.
- *ReadyForInput* → *CommandSent*: in AH01, erfolgt bei Eintreten des Events *MoveObject* unter der Bedingung, dass alle Roboter bereit sind (*all.isREADY*). Es folgt die Action *MoveToPosition*, welche an alle Arbeitsplätze gesendet wird und den Positionsparameter überträgt.
- *CommandSent* → *ReadyForInput*: verhaltenslose Transition, welche vom Zustand *CommandSent* sofort wieder zu *ReadyForInput* wechselt. Diese zusätzliche Transition ist nötig, da es bei *UML-SimpleStates* keine Selbsttransitionen geben darf.
- *ReadyForInput* → *ShowBusy*: in AH02, erfolgt bei Eintreten des Events *MoveObject* unter der Bedingung, dass mindestens ein Roboter beschäftigt ist (*one.isBUSY*). Es folgt die Action *WarnBusy*.
- *ShowBusy* → *ReadyForInput*: in AH03, erfolgt bei Eintreten des Events *ConfirmWarning*. Es folgt die Action *EndWarning*.
- *ReadyForInput* → *ShowError*: in AH04, erfolgt bei Eintreten des Events *MoveObject* unter der Bedingung, dass mindestens ein Roboter in einem Fehlerzustand ist (*one.isERROR*). Es folgt die Action *WarnError*.
- *ShowError* → *ReadyForInput*: in AH05, erfolgt bei Eintreten des Events *ConfirmWarning*. Es folgt die Action *EndWarning*.

Die Modellierung des Verhaltens der *RoboticWorkstations* erfolgt nach gleichem Vorgehen mit Hilfe der Anwendungsfälle AR01 bis AR04. Hier ist zu beachten, dass, entsprechend der DSL, jedem Zustand einer der Stereotypen *ReadyState*, *BusyState*, *ErrorState*, *InitialState* oder *FinalState* zugeordnet werden muss. Abbildung 6.5 zeigt die vollständige *RobotControl* Statechart.

- *WaitForCommand*: Der Roboterarm führt keine Aktion aus und die Lampe (L3) leuchtet nicht, als Voraussetzung von AR01. Da der Roboterarm in diesem Zustand bereit ist Befehle auszuführen, handelt es sich um einen *ReadyState*.
- *Working*: der Roboterarm führt eine Aktion aus, als Folge von AR01. Da der Roboter beschäftigt ist, handelt es sich um einen *BusyState*.
- *WaitForHelp*: der Roboterarm befindet sich in seiner Ausgangsposition, am Arbeitsplatz liegt ein Fehler vor und die rote Lampe (L3) leuchtet, als Folge von AR03. Da ein Fehlerfall vorliegt, handelt es sich um einen *ErrorState*.

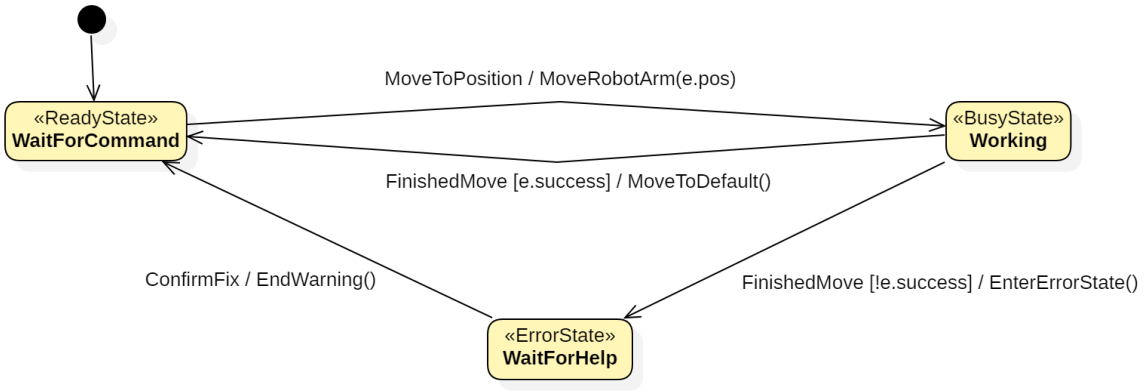


Abbildung 6.6: RobotControl Statechart des Modells

- *InitialState*: der Startzustand, von welchem aus bei Systemstart direkt in *WaitForCommand* gewechselt wird.
- *InitialState* → *WaitForCommand*: verhaltenslose Transition, welche nach dem Systemstart automatisch ausgeführt wird.
- *WaitForCommand* → *Working*: in AR01, erfolgt bei Eintreten des Events *MoveToPosition*. Es folgt die Action *MoveRobotArm*, welche den Positionsparameter des Events erhält.
- *Working* → *WaitForCommand*: in AR02, erfolgt bei Eintreten des Events *FinishedMove*, unter der Bedingung, dass dessen Parameter *success* wahr ist. Es folgt die Action *MoveToDefault*.
- *Working* → *WaitForHelp*: in AR03, erfolgt bei Eintreten des Events *FinishedMove*, unter der Bedingung, dass dessen Parameter *success* nicht wahr ist. Es folgt die Action *EnterErrorState*.
- *WaitForHelp* → *WaitForCommand*: in AR04, erfolgt bei Eintreten des Events *ConfirmFix*. Es folgt die Action *EndWarning*.

6.4 Programmgenerierung

Wenn das Modell, wie im vorhergehenden Abschnitt 6.3 beschrieben, innerhalb von StarUML erstellt wurde, kann die Modelldatei als Eingabe für den Programmgenerator genutzt werden. Der Programmgenerator produziert hierbei als Ausgabe fünf verschiedene Anwendungen: *RobotTUD*, *RobotMUC*, *ControllerTUD*, *ControllerMUC* und *Server*. Diese Anwendungen sind jedoch, mit Ausnahme des *Server* noch unvollständig, da entprechend der Spezifikation in Abschnitt 5.4 die hardwareseitige Steuerung manuell implementiert werden muss. Um dies zu ermöglichen, wurden in jeder *Workstation*-Anwendung zwei Klassen generiert, welche vervollständigt werden müssen. Am Beispiel des generierten *RobotTUD* sind diese Klassen in den folgenden Listings dargestellt. Hierbei wurde auf das Zeigen von Import- und Exportanweisungen verzichtet. Innerhalb dieser Klassen können nun Sensoren ausgelesen und Hardwareaktionen ausgeführt werden. Im Fall unserer demonstrativen Anwendung erfolgt das Auslösen von Events über die Konsole. Ausgehende Actions werden ebenfalls auf der Konsole in Textform ausgegeben. Die konkrete Umsetzung kann dem Quellcode der vollständig implementierten Endanwendungen entnommen werden.

```
class ActionProvider extends EndpointActionHandler {
  protected setup(ready: () => void): void {
    /*
     * insert your setup code here.
     * it will be executed, before the state machine starts.
     * please call ready() after everything is done.
     */
    ready();
  }
  public onMoveRobotArm(pos: number): void {
    // implement concrete action behaviour here.
  }
  public onEnterErrorState(): void {
    // implement concrete action behaviour here.
  }
  public onMoveToDefault(): void {
    // implement concrete action behaviour here.
  }
  public onEndWarning(): void {
    // implement concrete action behaviour here.
  }
}
```

```
class EventReceiver extends EndpointEventHandler {
  protected setup(ready: () => void): void {
    /*
     * insert your setup code here.
     * it will be executed, before the state machine starts.
     * please call ready() after everything is done.
     */
    ready();
  }
  /*
   * in this class, you can call super.onEvent()(event).
   * events, which can be handled, are shown in the following
   * sections.
   * in this version, please ignore default values and set all
   * parameters yourself.
   */
  // new FinishedMove(success: boolean);
  // new ConfirmFix();
}
```

6.5 Erweiterung der Fallstudie unter Nutzung des bestehenden Modells

Mit Hilfe unseres modellgetriebenen Vorgehens, ist es sehr einfach und schnell möglich, ein bestehendes Modell zu erweitern oder zu verändern, ohne dass dafür zusätzlicher Programmieraufwand nötig ist. Bei den im Folgenden gezeigten Beispielen, dient das Modell der in dieser Arbeit vorgestellten Fallstudie als Grundlage. Ebenfalls setzen wir hier voraus, dass die Implementierung der Schnittstelle für verwendete Events und Actions bereits existiert.

6.5.1 Erweiterung der Topologie

Die Topologie des Netzwerkes kann im Topologie-Diagramm verändert werden. Abbildung 6.7 zeigt die Topologie der Fallstudie, welche nun um eine *RoboticWorkstation* in Saarbrücken und eine *HumanWorkstation* in Berlin erweitert wurde. Die zusätzlichen Netzwerkteilnehmer werden hierbei mit dem existierenden *Node* verbunden. Da die beiden zusätzlichen *Workstations* einem bereits definierten Verhalten folgen, werden an dieser Stelle keine weiteren Statecharts benötigt. Wird die Anwendung nun erneut mit Hilfe des Programmgenerators generiert, finden sich in der Ausgabe die zwei zusätzlichen Teilanwendungen.

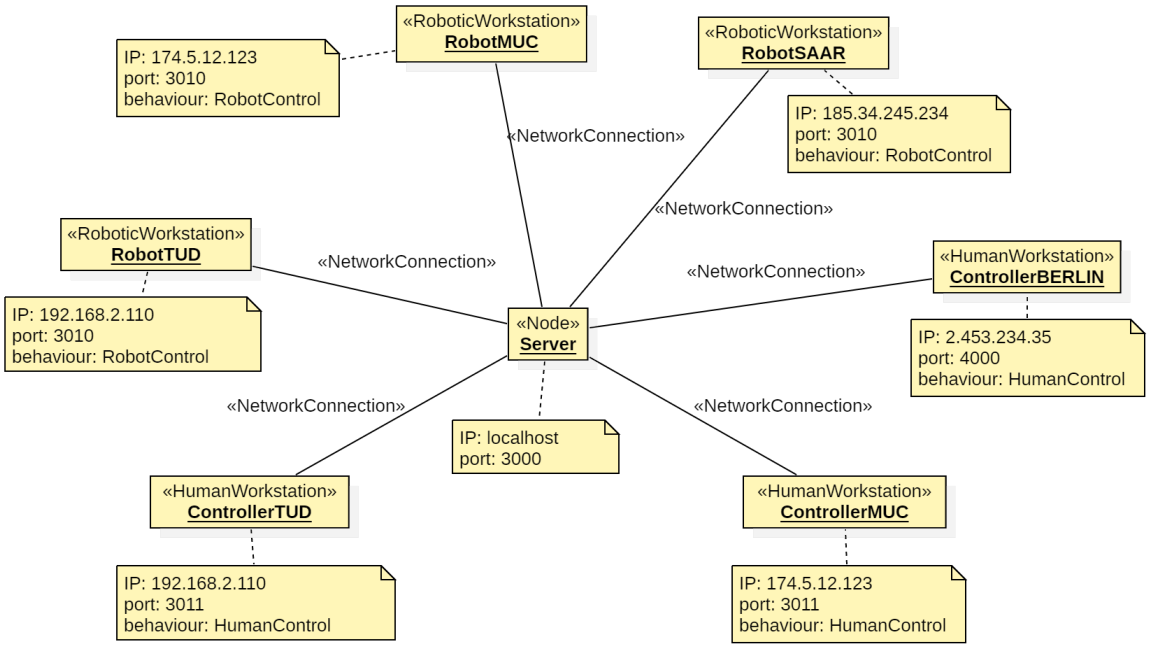


Abbildung 6.7: Erweiterte Topologie auf Basis der Fallstudie

6.5.2 Veränderung des Verhaltens

Soll das Verhalten eines oder mehrerer Netzwerkteilnehmer verändert werden, erfolgt dies innerhalb der assoziierten Statechart. In Abbildung 6.8 ist eine Variation des *RobotControl* Verhaltens dargestellt. Hierbei ist der Anwendungsfall AR01 verändert: erhält der Arbeitsplatz den Befehl aus dem Netzwerk, das Werkstück an seine Zielposition zu bewegen, nimmt der Roboterarm das Werkstück von seiner Ausgangsposition und bewegt es zu einem an seiner Zielposition befindlichen Werkzeug. Anschließend soll das Werkstück auf eine Ablage an Position 0 bewegt werden. Tritt beim ersten oder zweiten Schritt ein Fehler auf, soll sofort in den Fehlerzustand gewechselt werden. Wird die Anwendung nun erneut mit Hilfe des Programmgenerators generiert, wird

das neue Verhalten automatisch aktualisiert. Bezüglich der konkreten Ansteuerung des Roboterarms, müssen lediglich die manuell implementierten Bewegungskurven innerhalb der Action Schnittstelle angepasst werden.

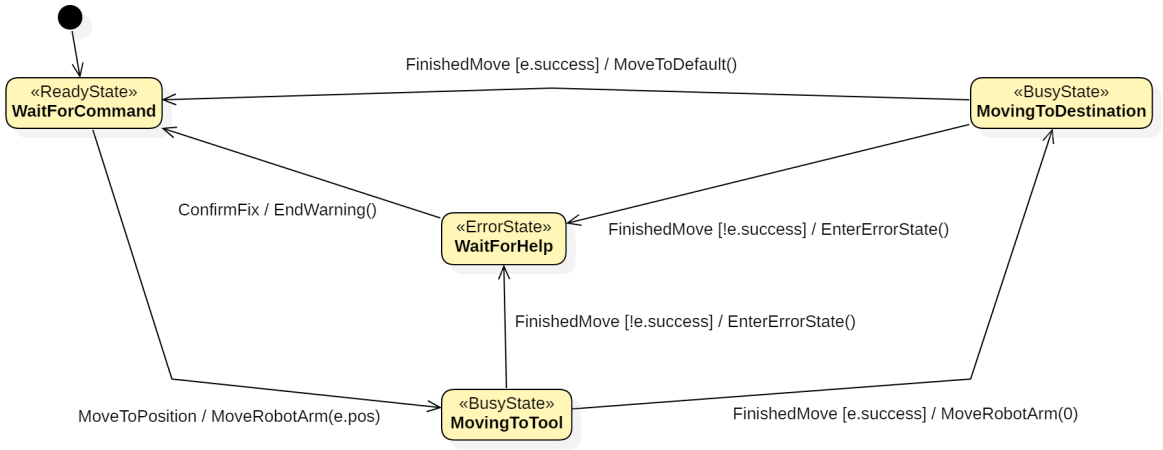


Abbildung 6.8: Erweiterte RobotControl Statechart auf Basis der Fallstudie

7 Evaluation

In Kapitel 5 habe wir ein konkretes modellgetriebenes Vorgehen beschrieben und dieses in Kapitel 6 beispielhaft angewandt. An dieser Stelle folgt nun die Evaluation unseres Lösungsansatzes. Dazu wird zunächst die aus der Fallstudie resultierende Endanwendung betrachtet. Anschließend gehen wir auf die Vorteile aber auch Herausforderungen sowie auf Möglichkeiten zur Erweiterung unseres modellgetriebenen Vorgehens ein.

7.1 Betrachtung der generierten Anwendung

Die aus dem Modell der Fallstudie generierte Anwendung funktioniert entsprechend ihrer Spezifikation. Um dies im Rahmen der Möglichkeiten dieser Arbeit zu validieren, wurden explorative Systemtests der Anwendung auf den Betriebssystemen Windows 10 und Ubuntu 18.04 durchgeführt. Dabei konnten insbesondere folgende Resultate beobachtet werden: das Verschieben der Teilanwendungen auf ihre entsprechenden Computer und die Installation von NodeJS waren die einzigen zur Bereitstellung nötigen Schritte. Nach der Bereitstellung konnten die Teilanwendungen direkt ausgeführt werden. Tests, bei welchen Systemteile zur gleichen Zeit auf dem selben oder verschiedenen Betriebssystemen ausgeführt wurden, verliefen erfolgreich. Dabei konnte festgestellt werden, dass die einzelnen Teilanwendungen fehlerfrei miteinander interoperieren. Ebenfalls ließen sich alle Anwendungsfälle mit verschiedenen Parameterwerten fehlerfrei ausführen. Die manuell implementierten Teile der Anwendung funktionierten entsprechend ihrer Vorgaben. Als Folge dieser Beobachtungen, bewerten wir die von uns gesetzten Ziele in Hinblick auf die Systemspezifikation und die Zielplattform als erreicht.

Wir weisen an dieser Stelle jedoch darauf hin, dass die Anwendung Verhalten besitzt, welches im Produktiveinsatz nicht erwünscht ist. So wurden während des gesamten Entwicklungsprozesses keine Sicherheitsmaßnahmen berücksichtigt. Dies ermöglicht z.B. bei Kenntnis der IP-Adresse des *Server*, das Einfügen von unautorisierten Events und Actions. Ebenfalls besitzt das verteilte System kein gesichertes Echtzeitverhalten. Durch unterschiedlich lange Übertragungs- und Verarbeitungszeiten von Events und Actions können Race-Conditions eintreten. Dies kann innerhalb der Statecharts zum Erreichen nicht existierender Zustände führen. Die Verbesserung dieser Kritikpunkte kann Bestandteil weiterführender Arbeiten sein.

7.2 Nachbetrachtung des gewählten Vorgehens

7.2.1 Vorteile durch das Einführen des modellgetriebenen Vorgehens

Für das Entwerfen einer plattformunabhängigen Architektur, welche sich in späteren Schritten gut automatisieren lässt, ist die Nutzung von Architektur- und Designmustern essenziell. Dies

schafft ebenfalls verbesserte Wart- und Erweiterbarkeit. Durch die nötige damit einhergehende Aufstellung von einer DSL, Modellierungsregeln und Richtlinien im Umgang mit der Architektur, existieren anschließend verbindliche und gut dokumentierte Vorgaben zur Modellierung. Während der Elaboration ist es ebenfalls wichtig, nachvollziehbare Entscheidungen zur Wahl der Zielplattform und damit einhergehenden Sprachen und Frameworks zu treffen. Dies schafft Transparenz und Verständnis in späteren Entwicklungsschritten. Bei der sich in der Automation anschließenden Entwicklung eines Programmgenerators besteht Interesse, Programmcode zu generieren, welcher einfach, kompakt und gut lesbar ist. Dadurch entstehen generierte Anwendungen, welche auch auf Codeebene gut verständlich, wart- und erweiterbar sind.

7.2.2 Vorteile durch die Nutzung der modellgetriebenen Techniken

Die Modellierung eines konkreten Systems erfolgt durch die bereitgestellten Werkzeuge sehr schnell und ohne weitere Expertise der zu Grunde liegenden Technologien. Das daraus entstehende Modell ist auf Grund seines domänenspezifischen Bezugs sehr aussagekräftig und bietet eine gute Grundlage für Diskussionen und Reviews in größeren Teams. Zu jeder Anwendung existiert genau ein aktuelles Modell, welches die Anwendung direkt beschreibt. So können keine Differenzen zwischen Modell und Anwendung existieren. Insbesondere bei verteilten Systemen, sind so alle Teilanwendungen Bestandteile des selben Modells. Dadurch wird stets Kompatibilität und Interoperabilität gewährleistet. Das enstandene Modell lässt sich ebenfalls sehr einfach erweitern und verändern, ohne, dass zu Grunde liegende technische Aspekte berührt werden müssen, da ein Großteil der sich ändernden Details durch das verwendete Technologie-Mapping von der eigentlichen Modellierung getrennt sind. Ist die korrekte Funktionsweise des Programmgenerators sichergestellt, folgen daraus stets Anwendungen, welche korrekte Bilder ihres Modells sind.

7.2.3 Herausforderungen

Während der Elaboration und Automation unseres modellgetriebenen Vorgehens, als auch während dessen Anwendung innerhalb der Fallstudie, sind mehrere Schwierigkeiten und Herausforderungen aufgetreten, welche an dieser Stelle hervorgehoben werden sollen. Zum Einen forderte der Entwurf einer plattformunabhängigen Architektur und domänenspezifischen Darstellung sehr detaillierte Kenntnisse der Zielsysteme. Ebenfalls verlangte das Finden einer geeigneten Zielplattform ein sehr breites Wissen zu verschiedenen Technologien. Es war dazu erforderlich, die Expertise und Erfahrungen anderer Entwickler in den Entscheidungsprozess mit einzubeziehen. Die Entwicklung eines Programmgenerators ist ein sehr komplexes Projekt, für welches ein hohes Maß an Kenntnis und Erfahrung mit den eingesetzten Technologien nötig war. Es erwies sich ebenfalls nicht als zielführend, den Programmgenerator losgelöst vom eigentlichen Prozess der Modellierung zu entwickeln. Wir haben daraufhin einen iterativen Ansatz gewählt, bei welchem Erweiterung eines Beispielmodells und Erweiterung des Generators abwechselnd stattfanden. Insgesamt nahm das Durchführen der initalen Schritte (Elaboration, Automation) deutlich mehr Zeit in Anspruch, als für die Entwicklung einer einzelnen Zielanwendung nötig gewesen wäre.

7.3 Erweiterungsmöglichkeiten des Ansatzes

Im Rahmen weiterführender Arbeiten kann das von uns beschriebene modellgetriebene Vorgehen erweitert werden. Zum Einen können durch Erweiterungen innerhalb der Modellierungsregeln und des Programmgenerators zusätzliche Funktionalitäten der UML nutzbar gemacht werden. Konkrete zusätzliche Funktionen wären zum Beispiel *Composite-States*, *Submachine-States* und *History-States* innerhalb der verwendeten Statecharts. Ein weiterer Ansatz ist die Vergößerung der Menge der Zielsysteme. Dabei wäre es zum Einen möglich, weitere Arten von Netzwerkteilnehmern einzuführen. Ebenfalls können Einschränkungen bezüglich der Netzwerktopologie

aufgehoben werden. Ein Beispiel hierfür wäre es Netzwerke zu modellieren, welche aus sich überlagernden Teilnetzwerken mit einer beliebigen Anzahl von *Workstations* und *Nodes* bestehen. Ein ebenfalls in dieser Arbeit nicht berücksichtigtes Gebiet ist das Echtzeitverhalten des verteilten Systems. Dieses kann im Rahmen weiterer Arbeiten auf Fehlerwirkungen, wie Race-Conditions, untersucht und dahingehend erweitert werden.

8 Fazit

Zu Beginn dieser Arbeit haben wir uns das Kernziel gesetzt, die Entwicklung der in dieser Arbeit behandelten Systeme robust und fehlerarm erfolgen zu lassen. Dies sollte durch die Umsetzung der beiden folgenden Teilziele erreicht werden:

1. Durch ein einheitliches und gesamtheitliches Modell besitzen die einzelnen Teilsysteme eine kompatible und aufeinander abgestimmte Architektur, da Inkompatibilitäten bereits durch die Modellierung aufgedeckt werden können. Daraus folgt auch, dass die Teilsysteme sehr gut miteinander interoperieren.

2. Durch die Modellierung von Robotern und ihrem Verhalten auf verschiedenen Abstraktionsebenen, entsteht ein Modell, welches ohne viel Expertenwissen direkt verwendet werden kann. Dadurch wird die Entwicklung von Robotersystemen stark vereinfacht. Daraus folgt auch, dass insgesamt nur wenig Expertenwissen zur Entwicklung solcher Systeme benötigt wird, was insbesondere aus der Weiterverwendbarkeit erstellter Modelle resultiert.

Wir können an dieser Stelle feststellen, dass wir die gesetzten Ziele erreicht haben. Die von uns spezifizierte DSL führt stets zu einem gesamtheitlichen Modell pro System und durch die automatische Programmgenerierung entspricht das resultierende System immer exakt dem Modell. Wir erhalten dadurch stets interoperable Teilsysteme. Teilziel 1 wurde somit erfüllt. Inwiefern das von uns beschriebenen Vorgehen und die damit einhergehende DSL die Entwicklung für Nicht-Experten vereinfacht, konnten wir im Rahmen dieser Arbeit nicht evaluieren. Jedoch konnten wir feststellen, dass die Modellierung mit Hilfe des eingeführten Vorgehens innerhalb der Fallstudie für uns sehr schnell und ohne Probleme erfolgte. Wir konnten auch zeigen, dass Modelle unserer DSL sehr einfach erweitert und verändert werden können, ohne Programmcode manuell ändern zu müssen. Als Folge betrachten wir Teilziel 2 ebenfalls als erfüllt.

Literaturverzeichnis

[RN07] Reza N. Jazar, *Theory of Applied Robotics: Kinematics, Dynamics, and Control*, Springer Science+Business Media, 2007, LCCN 2006939285, ISBN 978-0-387-32475-3 (siehe S. 1-2)

[JE17] Sabina Jeschke, Christian Brecher, Tobias Meisen, Denis Özdemir, Tim Eschert, *Industrial Internet of Things and Cyber Manufacturing Systems*, Springer International Publishing Switzerland, 2017, DOI 10.1007/978-3-319-42559-7 (siehe S. 3-10)

[GS08] Michael A. Goodrich, Alan C. Schultz, *Human–Robot Interaction: A Survey*, Foundations and Trends in Human–Computer Interaction: Vol. 1: No. 3 pp 203-275, 2008, DOI 10.1561/1100000005 (siehe S. 1-9)

[GA93] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Abstraction and Reuse of Object-Oriented Design* (Paper), in ECOOP 93 - Object-Oriented Programming pp 406-431, 1993, DOI 10.1007/3-540-47910-4_21 (siehe S. 1 ff.)

[GA05] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design patterns: elements of reusable object-oriented software*, Addison-Wesley, Boston, München, 32. Auflage, 2005 ISBN 978-0-201-63361-0 (siehe S. 305-310)

[UML17] OMG, *OMG® Unified Modeling Language® (OMG UML®)*, Object Management Group, Version 2.5.1, December 2017, OMG Document Number: formal/2017-12-05, `https://www.omg.org/spec/UML/2.5.1` (zuletzt zugegriffen am 24.07.2019)

[HA87] David Harel *Statecharts: A Visual Formalism for Complex Systems*, (S. 231-274), Science of Computer Programming, Elsevier, June 1987, `https://www.sciencedirect.com/science/article/pii/0167642387900359` (siehe S. 1 ff.)

[HA96] David Harel, Amnon Naamad, *The STATEMATE Semantics of Statecharts*, in *ACM Transactions on Software Engineering and Methodology* Vol. 5 Nr. 4, 1996, DOI 10.1145/235321.235322 (siehe S. 24, 25)

[BA09] Helmut Balzert *Lehrbuch der Softwaretechnik: Basiskonzepte und Requirements Engineering*, Spektrum Akademischer Verlag, 3. Auflage, Heidelberg 2009, ISBN 978-3-8274-1705-3 (siehe S. 269-292)

[KA09] Bernd Kahlbrandt *Software-Engineering mit der Unified Modeling Language*, Springer-Verlag, 2. Auflage, Berlin Heidelberg New York 2001, ISBN 3-540-41600-5 (siehe Kapitel 6)

[SO07] Ian Sommerville *Software Engineering* Pearson Studium, 8. aktualisierte Auflage, München 2007, ISBN 978-3-8273-7257-4 (siehe Kapitel 14.1)

[WR10] Wolfgang Reisig *Petrinetze Modellierungstechnik, Analysemethoden, Fallstudien*, Vieweg+Teubner Verlag, Wiesbaden 2010, ISBN 978-3-8348-1290-2 (Siehe S. 1-6, 21-34)

[JK09] Kurt Jensen, Lars M. Kristensen *Coloured Petri Nets Modelling and Validation of Concurrent Systems*, Springer-Verlag, Berlin Heidelberg 2009, ISBN 978-3-642-00283-0 DOI 10.1007/b95112 (siehe S. 1-12, 43, 44)

[EN94] Javier Esparza, Mogens Nielsen *Decidability Issues for Petri Nets*, BRICS, Department of Computer Science University of Aarhus, Aarhus 1994, ISSN 0909-0878 (siehe S. 5-11)

[PE62] C.A.Petri, *Kommunikation mit Automaten*, Institut für Instrumentelle Mathematik, Schriften des IIM Nr. 2, 1962, urn:nbn:de:gb:18-228-7-1602 (siehe S. 1 ff.)

[SVEH07] Thomas Stahl, Markus Völter, Sven Efftinge, Arno Haase *Modellgetriebene Softwareentwicklung Techniken, Engineering, Management*, dpunkt.verlag, 2.Auflage, Heidelberg 2007, ISBN 978-3-89864-448-8 (siehe S. 1 ff., Definition Kapitel 2.1)

[MDA14] OMG, *Model Driven Architecture (MDA) MDA Guide rev. 2.0*, Object Management Group, 2014, `https://www.omg.org/mda/` (zuletzt zugegriffen am 24.07.2019)

[EMF09] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, Ed Merks, *EMF - Eclipse Modeling Framework*, Pearson Education Inc., Boston 2009, ISBN 978-0-321-33188-5 (siehe S. 11-23)

[EMF19] Eclipse Foundation *Eclipse Modeling Framework (EMF)*, 2019, `https://www.eclipse.org/modeling/emf/` (zuletzt zugegriffen am 24.07.2019)

[VELO16] The Apache Software Foundation *The Apache Velocity Project*, `http://velocity.apache.org/` (zuletzt zugegriffen am 24.07.2019)

[GSR05] Live Geiger, Christian Schneider, Carsten Reckord, *Template- and modelbased code generation for MDA-Tools* in Proceedings Fujaba Days 2005, Paderborn 2005 (siehe S. 1 ff.)

[CPN19] Eindhoven University of Technology *CPN Tools* `http://cpntools.org/` (zuletzt zugegriffen am 24.07.2019)

[SS10] Andreas Steck, Christian Schlegel, *Towards Quality of Service and Resource Aware Robotic Systems through Model-Driven Software Development*, University of Applied Sciences Ulm, 24 Sep 2010 arXiv:1009.4877v1 [cs.RO], (siehe S. 1 ff.)

[SM13] *SmartSoft Components and Toolchain for Robotics* University of Applied Sciences Ulm, 2013, `http://smart-robotics.sourceforge.net/index.php` (zuletzt zugegriffen am 24.07.2019)

[MGV00] Luis Montano, Francisco José García, José Luis Villarroel, *Using the Time Petri Net Formalism for Specification, Validation, and Code Generation in Robot-Control Applications*, in The International Journal of Robotics Research, 2000 Volume: 19 issue: 1, Seiten: 59-76, `https://doi.org/10.1177/02783640022066743` (siehe S. 1 ff.)

[RA11] Mohd Azizi Abdul Rahman, Katsuhiro Mayama, Takahiro Takasu, Akira Yasuda, Makoto Mizukawa, *Model-Driven Development of Intelligent Mobile Robot Using Systems Modeling Language (SysML)*, 2011, DOI: 10.5772/26906 (siehe S 1. ff)

[NT03] Iftikhar Niaz, Jiro Tanaka, *Code Generation From Uml Statecharts*, in Proceedings of the 7th IASTED International Conference on Software Engineering and Applications, 2003 (siehe S 1. ff.)

[SYS17] SysML.org *SysML Specifications*, 2017, `https://sysml.org/sysml-specifications/` (zuletzt zugegriffen am 24.07.2019)

[ST19] MKLabs Co.,Ltd *StarUML 3*, 2019, `http://staruml.io/` (zuletzt zugegriffen am 24.07.2019)

[NPM19] npm, Inc., *Node Package Manager*, 2019, `https://www.npmjs.com/` (zuletzt zugegriffen am 24.07.2019)

[TS19] Microsoft, *TypeScript* 2019, `https://www.typescriptlang.org/` (zuletzt zugegriffen am 24.07.2019)

[NJ19] Node.js Foundation, *NodeJs*, 2019, `https://nodejs.org` (zuletzt zugegriffen am 24.07.2019)

[JSO17] Ecma International *The JSON Data Interchange Syntax*, in ECMA-404 2nd Edition, Dezember 2017, `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf` (zuletzt zugegriffen am 24.07.2019)

[PA19] The Eclipse Foundation *Eclipse Papyrus*, 2019, `https://www.eclipse.org/papyrus/` (zuletzt zugegriffen am 24.07.2019)

[JA19] Oracle, *Java*, 2019, `https://www.oracle.com/de/java/` (zuletzt zugegriffen am 24.07.2019)

[EC19] Eclipse Foundation, *Package org.eclipse.emf.ecore*, 2019, `https://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/` (zuletzt zugegriffen am 24.07.2019)

[XT19] Eclipse Foundation, *Xtext*, 2019, `https://www.eclipse.org/Xtext/index.html` (zuletzt zugegriffen am 24.07.2019)

[GMF19] Eclipse Foundation, *Graphical Modeling Framework GMF*, 2019, `https://wiki.eclipse.org/index.php?title=Graphical_Modeling_Framework&redirect=no#Get_started` (zuletzt zugegriffen am 24.07.2019)

[KK19] Karl Kegel, *Codegenerierung - Herausforderungen und Ansätzte*, Seminararbeit *Service and Cloud Computing*, Technische Universität Dresden, 2019

[TO13] Marco Torchianoa, Federico Tomassetti, Filippo Ricca, Alessandro Tiso, Gianna Reggio, *Relevance, benefits, and problems of software modelling and model driven techniques—A survey in the Italian industry*, in *Journal of Systems and Software* (S. 2110-2126), Volume 86 Issue 8, Elsevier, 2013 (siehe S. 2119 ff.)

[MU19] Mustache Github Organization *Mustache*, 2019, `http://mustache.github.io/` (zuletzt zugegriffen am 24.07.2019)