Gottfried Wilhelm
Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Praktische Informatik
Fachgebiet Software Engineering

Simulationsumgebung für die Analyse von probabilistischer szenariobasierten Spezifikationen zuverlässiger ubiquitärer Systeme

Bachelorarbeit

im Studiengang Informatik

von

Dennis Freund

Prüfer: Prof. Dr. Joel Greenyer
Zweitprüfer: Prof. Dr. Kurt Schneider
Betreuer: Prof. Dr. Joel Greenyer

Hannover, 10.03.2015

Erklärung der Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 10.03.2015

Dennis Freund

Zusammenfassung

Ubiquitäre Systeme bestehen aus vielen einzelnen Komponenten, welche wiederum eigenständige Systeme sein können. Um den Entwicklungsprozess solcher Systeme zu verbessern, wurde ein Tool entwickelt, welches unter anderem die Möglichkeit der Durchführung einer frühzeitigen Zuverlässigkeitsanalyse, basierend auf Szenarien, ermöglicht. Innerhalb des Tools wurden bereits Möglichkeiten implementiert Szenarien in Form von MSD, eine Erweiterung der UML Sequenzdiagramme, zu erstellen und, mit Hilfe eines Play-Out Algorithmus, von Computern interpretieren und durchspielen zu lassen.

In dieser Arbeit geht es darum, dieses Tool zu erweitern. Hier wird eine mögliche Lösung beschrieben, mit deren Hilfe die Darstellung von Wahrscheinlichkeiten innerhalb dieses Tools ermöglicht wird. Hierzu musste die Spezifikationssprache angepasst werden, nachdem ein passendes Beispiel gefunden und implementiert wurde. Des Weiteren wurde ein vorhandener Simulationsalgorithmus angepasst, damit dieser die eingegebenen Wahrscheinlichkeiten korrekt interpretieren und ihnen entsprechend agieren kann.

Abstract

Ubiquitous systems are composed of many individual components, which can be stand-alone systems on their own. With the goal to improve the development process, a tool got developed, which provides a possibility to analyze the reliability, based on scenarios, of such systems. Within this tool, an option got implemented, which allows the user to create MSDs. MSDs are an extension of the widely known sequence diagrams from UML. With the help of a play-out algorithm, the MSDs can be interpreted and played through by computers.

The goal of this thesis is to create an extension. I present a possible solution, which helps to represent possibilities within this tool. To achieve this, I adjusted the specification language, right after I choose a fitting example and implemented called example. Also I changed an existing simulation algorithm in such manner that given possibilities would be interpreted the correct way, so that the system would act accordingly.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Problemstellung

In einer Zeit, in der immer mehr computergesteuerte Systeme zum Einsatz kommen, rückt die Entwicklung von besseren und innovativeren Systemen immer mehr in den Vordergrund. Insbesondere sogenannte ubiquitäre Systeme werden immer präsenter. In diesem Zusammenhang sind ubiquitäre Systeme zu verstehen als vernetzte, eingebettete Systeme, welche den Menschen in vielfältiger Form umgeben.

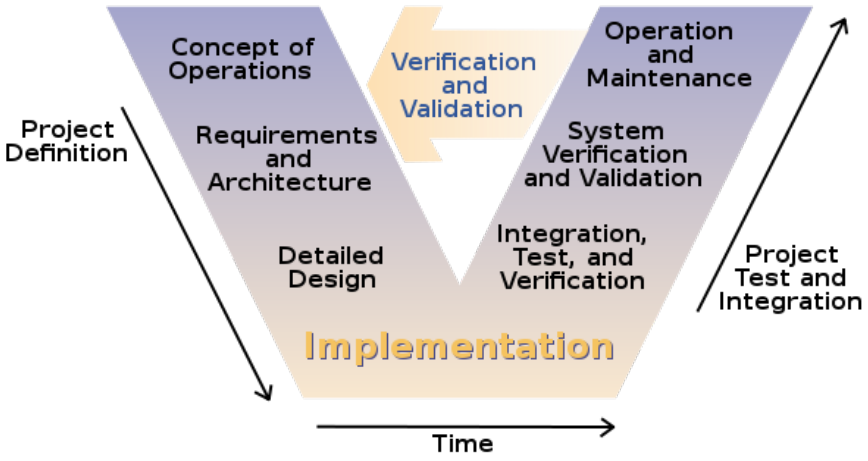

Abbildung 1.1: V-Model der Softwareentwicklung

Diese Systeme werden nach einer gewissen Vorgehensweise entworfen, die man grob in vier Schritte einteilen kann. Wie in Abbildung 1.1 zu erkennen, werden im ersten Schritt Anforderungen von Entwicklern an das System, in Form von Use-Cases und Szenarien, erhoben. Danach fertigen die Entwickler einen Entwurf, in welchem funktionale und nichtfunktionale Anforderungen

als Eingabe dienen, mit deren Hilfe ein Modell, in Zustandsbasierter Form, des Softwaresystems erstellt werden kann. Daraufhin wird das System implementiert und im letzten Schritt getestet. Dieses Vorgehen wird V-Model genannt und ist in Abbildung 1.1 dargestellt. Aufgrund der Komplexität dieser Systeme ist es sehr wahrscheinlich, dass spätestens beim Testen fehlerhafte Anforderungen entdeckt werden, welche gegebenenfalls eine weitere Iteration oder Neuentwicklung des Systems zur Folge haben.

Bei der Entwicklung von modernen Maschinen und Systemen wird sehr stark Wert darauf gelegt, dass diese möglichst zuverlässig arbeiten. Die Gründe dafür sind vor allem, dass bei unzuverlässigen Produkten der Absatz und somit auch die Einnahmen erheblich sinken und die Produktion sich somit als unwirtschaftlich erweisen könnte. Um dem entgegenzuwirken werden Verlässlichkeitstests durchgeführt. Meistens werden diese Tests erst mit Hilfe der fertigen Systeme durchgeführt. Sollte sich bei dem Testen jedoch zeigen, dass das Produkt nicht den Erwartungen entspricht müssen die Anforderungen oder der Entwurf entsprechend angepasst werden. Solch eine Iteration ist äußerst aufwendig und sollte nach Möglichkeit vermieden werden. Aus diesem Grund agieren moderne Verfahren auf Basis Zustandsbasierten Modellen. Die können bereits in der späten Entwurfsphase auf deren Verlässlichkeit überprüft werden. Trotz des relativ frühen Einstiegs, sind diese Verfahren immer noch mit hohen Kosten verbunden, da Testumgebung, Testtreiber sowie Testfälle bereitgestellt werden müssen. Um dem entgegenzuwirken müsste noch früher in der Entwicklung eingestiegen werden.

Zu diesem Zweck wurde ScenarioTools 2.6 entworfen. ScenarioTools ist ein Programm, welches mit modifizierten Sequenzdiagrammen und somit szenariobasiert arbeitet. Diese sind schon aus UML bekannt. Es wird das Ziel verfolgt, Widersprüche innerhalb der Anforderungen frühzeitig aufzudecken und möglichst früh im Entwicklungsprozess eine erste Verlässlichkeitsanalyse durchzuführen. Hierfür werden MSD (Modal Sequence Diagrams) verwendet, welche Sequenzdiagrammen der UML stark ähnlich sind, aber eine erweiterte Palette an Möglichkeiten bereitstellen. MSD stellen unter anderem eine Möglichkeit bereit, das Verhalten des Systems mit dem Austausch von Nachrichten innerhalb des Systems und mit anderen Systemen für die Entwickler leicht interpretierbar und auch für Computer verständlich wiederzugeben. Innerhalb der letzten Jahre wurde ein Algorithmus entwickelt, der Szenarien, welche in Form von MSD erstellt wurden, interpretieren kann. Mit Hilfe dieses Algorithmus können Softwareentwickler schon während des Entwurfs das Verhalten, welches sich durch das Zusammenspiel von Szenarien ergibt, durchspielen und eventuelle Widersprüche frühzeitig aufdecken. Hierbei handelt es sich um eine erste automatisierte Validierung.

In dem momentanen Implementierungszustand können noch nicht alle Einzelheiten realer Systeme modelliert und einbezogen werden, so soll bei kritischen Systemen ein gewisser Zuverlässigkeitsgrad 2.3 erreicht werden. Aus ebendiesem Grund müsste eine Möglichkeit implementiert werden um

die Fehleranfälligkeit von Systemen nachstellen zu können. Hierzu muss die Spezifikationssprache um eine Möglichkeit erweitert werden, (Ausfall-) Wahrscheinlichkeiten darzustellen. Außerdem muss der Simulationsalgorithmus dementsprechend angepasst werden, sodass dieser implementierte Wahrscheinlichkeiten als solche wahrnehmen und verarbeiten kann. Dies geschieht um sicherzustellen, dass Ausfallwahrscheinlichkeiten, wie eine Fehleranfälligkeit von 1 zu 1000 beim Schwenken eines Roboterarms in der Verlässlichkeitsanalyse beachtet werden und das Ergebnis dementsprechend an die Gegebenheiten der realen Welt angepasst werden kann.

1.2 Lösungsansatz

In dieser Arbeit habe ich zuerst ein geeignetes Beispiel ausgewählt, an welchem ich nach der Implementierung die Funktionalität der von mir vollführten Änderungen testen kann. Zur Auswahl standen zwei Möglichkeiten, ein sensorgesteuerter Bahnübergang und eine vereinfachte Produktionsmaschine. Zwecks der leichteren Überprüfbarkeit habe ich die Produktionsmaschine ausgewählt. Diese Maschine besteht aus zwei Laufbändern und einem Roboterarm, welcher Teller von Laufband 1 auf Laufband 2 überführt. Ich habe zwei Fehlerquellen eingebaut, jedes Mal wenn der Roboterarm zwischen den Laufbändern hin und herschwenkt, besteht die Möglichkeit, dass ein Fehler passiert und der Arm nicht ankommt.

Als nächsten Schritt habe ich das ausgewählte Beispiel in ScenarioTools implementiert.

Danach ging es darum, eine möglichst benutzerfreundliche und sinnvolle Lösung zu finden Wahrscheinlichkeiten innerhalb von ScenarioTools darzustellen und den relevanten MSD zuzuordnen. Dies ließ sich dadurch bewältigen, dass ich bei dem Austausch der Nachrichten die Möglichkeit hinzufügte, diesen einen Anhang beizufügen, welcher die zwei relevanten Eigenschaften enthielt. Erstens, es wurde klar gemacht, dass diese Nachricht eine Wahrscheinlichkeit enthält. Zweitens, es wird eine Wahrscheinlichkeit in Form von zwei Integern übergeben, welche einen Bruch formen.

Nach der Einführung der Wahrscheinlichkeiten habe ich mich damit beschäftigt, einen Algorithmus zu entwerfen, welcher die Möglichkeit eines automatischen Durchlaufs bereitstellt, bzw. den bereits bereitgestellten Algorithmus weiterzuentwickeln. Dieser automatische Durchlauf stellt eine Methode bereit, sodass ScenarioTools ohne weitere Eingabe / Interaktion, die bereits erstellten MSD durchgeht und automatisch die nächste Nachricht auswählt. Der bereitgestellt Algorithmus hatte eine Methode implementiert, die es erlaubte die nächste Nachricht zufällig auszuwählen. Auf Basis dieser Methodik habe ich ein Verfahren entwickelt, welches die vorhandenen Wahrscheinlichkeiten korrekt interpretiert und bei der Auswahl der nächsten Nachricht entsprechend berücksichtigt.

1.3 Struktur der Arbeit

Hier die Struktur der Arbeit kurz zusammenfassen: Diese Arbeit ist wie folgt strukturiert. Diese Arbeit ist in sechs Kapitel unterteilt. In Kapitel 2 erläutere ich auf die technischen Grundlagen. Wichtige Konzepte werden in Kapitel 3 vorgestellt. Kapitel 4 handelt von der Implementierung dieser Konzepte in ScenarioTool. Verwandte Arbeiten vergleiche ich anschließend in Kapitel 5 und liefere anschließend in Kapitel 6 eine Zusammenfassung meiner Arbeit.

Kapitel 2

Grundlagen

2.1 Ubiquitäre Systeme

Übersetzt man Ubiquität, erhält man die Eigenschaft, überall vorkommen zu können. In dieser Arbeit sind Ubiquitäre Systeme jedoch anders definiert. Als Ubiquitäre Systeme sehen wir Systeme an, welche aus mehreren eigenständigen Systemen, evtl. auch ubiquitäre, bestehen und im Zusammenspiel mit ihrer Umwelt sicher und zuverlässig funktionieren. Diese Systeme sind vernetzt, eingebettet und umgeben den Menschen in vielfältiger Form, wie moderne Automobile oder *Smart Homes*. Damit das Ganze verständlicher wird, erläutere ich die Definition anhand von zwei Beispielen.

2.1.1 Beispiel 1

Als erstes Beispiel nehmen wir eine automatisierte Schrankensteuerung für einen Autoübergang einer Bahnstrecke. Im Basiszustand oder auch „Default-Zustand“ sind die Schranken geöffnet und somit wird der Übergang für Fahrzeuge ermöglicht. Um es zu ermöglichen, dass die Schranken automatisch geschlossen werden, sobald sich ein Zug nähert, müssen mehrere Systeme miteinander arbeiten und Informationen austauschen. Als erstes gibt es Sensoren, die in ausreichendem Abstand zu dem Übergang angebracht sind und Informationen weitergeben, wenn sich ein Zug in Richtung Bahnübergang bewegt. Weiterhin sind Sensoren angebracht, die den Übergang der Fahrzeuge beobachten und Informationen weitergeben, ob sich ein Fahrzeug auf dem Übergang befindet. Dies dient dem Zweck, dass keine Fahrzeuge auf dem Übergang eingeschlossen werden, wenn sich die Schranken senken. Des Weiteren gibt es zwei Verkehrsampeln, welche den Zügen signalisieren, ob sie weiterfahren dürfen, zwei Verkehrsampeln, welche den Autofahrern signalisieren, dass sich ein Zug nähert und zwei Systeme, die die Schranken anheben und senken. Im Normalfall, ohne Ausfälle oder Fehlverhalten der Systeme, nähert sich ein Zug dem Übergang und wird von einem Sensor als solcher erkannt. Daraufhin überprüft der Sensor am Übergang ob sich

gerade ein Fahrzeug auf diesem Übergang befindet. Ist dies nicht der Fall, so signalisiert die Ampel dem Zug, dass er weiterfahren kann, die Verkehrsampeln für die Fahrzeuge signalisieren, dass sich ein Zug nähert und die Schranken senken sich. Nachdem der Zug den Übergang hinter sich gelassen hat, signalisiert ein weiterer Sensor -der gleiche Sensor der misst ob sich ein Zug nähert- jedoch auf der anderen Seite des Übergangs, dass der Zug das Gebiet verlassen hat. Sobald das Signal dafür gekommen ist, wird die Verkehrsampel für Straßenfahrzeuge ausgeschaltet, die Schranken heben sich und die Ampel für die Züge wird so eingestellt, dass ein ankommender Zug halten muss, sollte sie nicht wieder umgestellt werden.

2.1.2 Beispiel 2

Das zweite Beispiel ist eine Produktionsmaschine, wie sie in Fabriken anzutreffen ist. Der Einfachheit halber wählen wir eine Maschine kleineren Ausmaßes. Diese Maschine besteht aus zwei Laufbändern, zwei Roboterarmen und einer Presse. Der Ablauf dieser Maschine ist folgender, ein Produkt bewegt sich entlang des ersten Laufbandes, sobald es das Ende dieses Laufbandes erreicht, signalisiert ein Sensor, dass das Produkt bereitliegt. Daraufhin nimmt der erste der Roboterarme das Produkt auf, legt es in die Presse und geht wieder zum ersten Laufband zurück um weitere Produkte in Empfang zu nehmen. Im folgenden Schritt presst die Presse das Produkt und signalisiert dem zweiten Roboterarm, dass dieser das Produkt in Empfang nehmen und auf dem zweiten Laufband platzieren kann.

Wie in beiden Beispielen beschrieben arbeiten bei ubiquitären Systemen eigenständige Systeme (Sensoren, Roboterarme, Laufbänder, etc.) zusammen um einen gewünschten Zweck zu erfüllen.

2.2 Entwicklung von Systemen

Die in 2.1 beschriebenen Systeme können mitunter äußerst komplexe Formen annehmen und somit ist es zweckdienlich, sich festgelegter Vorgehensweisen zu bedienen um solche Systeme zu entwickeln. Wie eine solche Vorgehensweise aussehen kann, möchte ich anhand des sogenannten V-Modells erklären, welches seinen Namen der graphischen Visualisierung dieser Arbeitsabfolge verdankt. Wie in Abbildung 1.1 zu sehen ist, geht man bei dem V-Modell in 4 Schritten vor.

Als erstes werden sogenannte Anforderungen erhoben. Beispiel für eine solche Anforderung könnte bei der Entwicklung eines Fahrassistenten für PKWs sein, dass gesagt wird: „Wenn hinter dem Fahrzeug ein LKW fährt, nicht abrupt und scharf bremsen.“ So eine Anforderung ergibt Sinn, da falls der führende PKW zu scharf und abrupt bremst, durch den längeren Bremsweg des LKWs, leicht ein Unfall zustande kommen

kann. Eine zweite Anforderung an den Fahrassistenten könnte sein: „Bei einem Fußgängerüberweg bremsen, falls dieser gerade genutzt wird, im Zweifelsfall auch scharf.“ Auch diese Anforderung ist durchaus stimmig mit dem Fahrverhalten von PKWs im Straßenverkehr, da man ansonsten einen schweren Unfall mit enormen Verletzungen riskiert.

Der zweite Schritt bei der Entwicklung von Systemen ist, wenn man dem V-Model folgt, die Erstellung eines Entwurfs. Nehmen wir hier als Beispiel einen Kaffeeautomaten. In der Entwurfsphase wird festgelegt, wie sich das System des Automaten zu verhalten hat. Wenn „Kaffee“ gedrückt wird gefolgt von „Milch“, soll natürlich ein Kaffee mit Milch ausgegeben werden.

Nach der Erstellung eines Entwurfs, folgt die Implementierung. Hierbei wird das System erstellt. Dies bedeutet die notwendigen Programme werden geschrieben oder abgeändert, Bauteile werden eventuell hergestellt und verarbeitet.

Im letzten Schritt folgt das Testen. Beim Testen wird das im dritten Schritt erstellte System systematisch überprüft, ob es die im ersten Schritt erstellten Anforderungen erfüllt.

2.3 Reliability

Reliability R(t) eines Systems zum Zeitpunk t ist die Wahrscheinlichkeit, dass dieses System ohne Fehler im Interval [0, t] arbeitet, wenn man annimmt, dass das System zum Zeitpunkt 0 korrekt funktioniert hat (Verweis: Fault Tolerant Design, S6). Hier wird Reliability als Ausfallsicherheit übersetzt, welche ein Maß für das konstante Liefern des korrekten Services ist. Besonders hohe Ausfallsicherheit ist insbesondere wichtig, wenn ein System ohne Unterbrechung arbeiten muss, oder wenn eine Wartung nicht durchgeführt werden kann [5].

Bei unserem Beispiel ist sowohl eine Unterbrechung als auch die Wartung des Systems möglich, trotzdem wird ein hohes Maß an Ausfallsicherheit angestrebt. Dies gilt bei Produktionsmaschinen zumindest solange, bis die Verbesserung der Ausfallsicherheit eine ungleichmäßig höhere Investionen nach sich ziehen. In diesem Fall müsste berechnet werden, ob das derzeitige System nicht ausreicht und man Fehler in Kauf nimmt, aus dem Grund, dass ein 99,9% ausfallsicheres System reicht, welches weitere Investitionen nicht rechtfertigen würde.

2.4 Unified Modeling Language

UML ist eine graphische Modellierungssprache, welche von der Object Modeling Group (OMG) entwickelt wurde. Zum grundlegenden Verständnis dieser Arbeit, ist das Wissen um UML-Sequenzdiagramme von Nöten,

sowie das Wissen um Möglichkeit UML mit Hilfe von Stereotypen zu erweitern. Von der Kenntnis der Funktionsweise der Sequenzdiagramme wird in weiteren ausgegangen. Als Stereotype der UML ist hier zu verstehen, eine Erweiterung bereits vorhandener Modellelemente [8] [9]. Diese Stereotypen werden vor allem in sogenannten UML-Profilen genutzt, bei welchen sie wichtige Bestandteile sind um das UML-Metamodell zu erweitern. Ein gutes Beispiel für solche Stereotypen ist in Abbildung 2.1 zu erkennen.

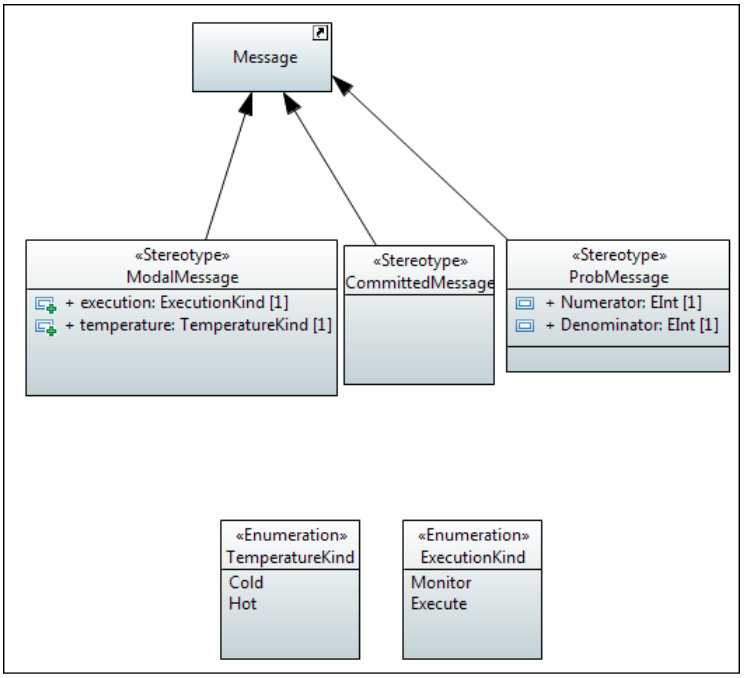

Abbildung 2.1: Ein Beispiel für Stereotypen in UML

Hierbei wird der Message, der Nachricht in einem Sequenzdiagramm, durch Stereotypen die Möglichkeit bereitgestellt, diese zu erweitern. In diesem Fall ist es zum Beispiel möglich einer Nachricht den Stereotype ModalMessage beizufügen, durch welchen man der Nachricht einen TemperatureKind sowie einen ExecutionKind anhängen kann.

2.5 Modal Sequence Diagrams

MSD sind eine szenariobasierte Sprache, welche Ingenieuren und Entwicklern die Möglichkeit bietet, intuitiv und präzise die Interaktionen von Systemkomponenten untereinander zu spezifizieren [2]. Sie sind eine Erweiterung der sogenannten Live Sequence Charts (LSC), welche in dem Buch „Come Let's Play“ vorgestellt wurden [4].

2.5.1 Beispiel

Zur Verdeutlichung ist ein geeignetes Beispiel eine vereinfachte Version einer Produktionsanlage. Dieses Beispiel wurde ScenarioTools entnommen und leicht von mir verändert.

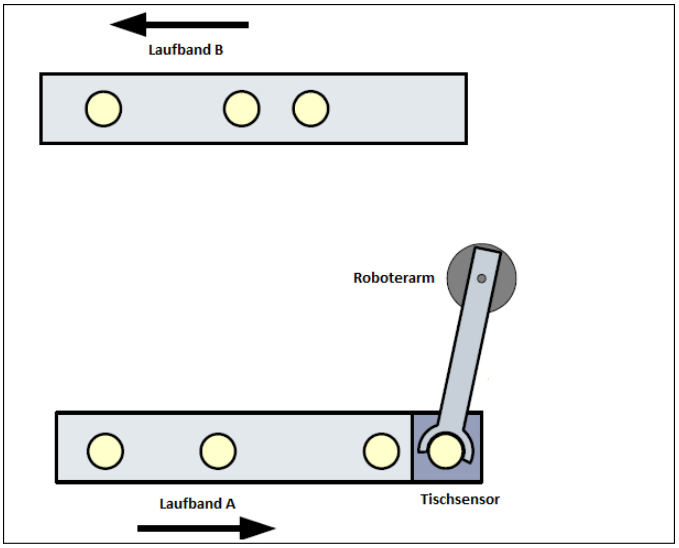

Abbildung 2.2: Der Roboterarm nimmt einen Teller von Laufband A auf.

Bei dieser Anlage existieren zwei Laufbänder, ein Roboterarm und ein Tischsensor. Bei dieser Anlage, werden auf Laufband A Teller in Richtung eines Tischsensors befördert. Sobald die Teller den Sensor erreichen nimmt ein Roboterarm die Teller auf, wie in Abbildung 2.2.

Nachdem der Roboterarm den Teller aufgenommen hat, schwenkt der Arm in Richtung von Laufband B und platziert den Teller auf ebendiesen 2.3. Nach der Platzierung des Tellers auf Laufband B schwenkt der Roboterarm wieder in die Ausgangsposition über Laufband A, wo er auf das Erreichen eines weiteren Tellers wartet, um diesen wieder aufzunehmen.

Wenn man dieses Beispiel in Anforderungen und Annahmen übersetzt, ergeben sich eine Anforderung sowie drei Annahmen:

Anforderung 1: Sobald ein Teller vom Tischsensor registriert wird, muss der Roboterarm ihn aufnehmen, zu Laufband B bringen und ihn darauf ablegen. Daraufhin hat der Arm zu seiner Grundposition über dem Tischsensor zurückzukehren.

Annahme 1: Wenn sich der Arm in Richtung Laufband B bewegt, kommt er dort an.

Annahme 2: Wenn sich der Arm in Richtung Laufband A bewegt, kommt er dort an.

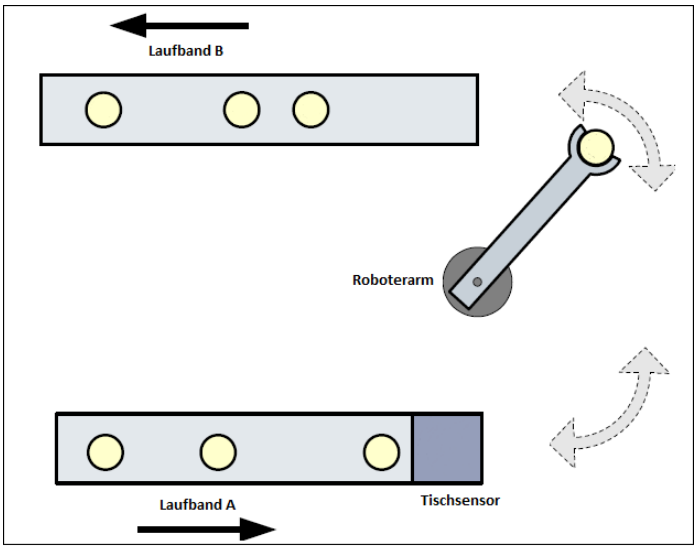

Abbildung 2.3: Der Roboterarm bewegt sich mit dem Teller zu Laufband B, lässt ihn dort ab und bewegt sich wieder zu Laufband A.

Annahme 3: Wenn ein Teller vom Sensor registriert wurde, darf der nächste Teller erst ankommen, nachdem der Arm wieder in seiner Grundposition ist.

2.5.2 Modalitäten

Ein großer Unterschied zwischen MSD und bisherigen Sequenzdiagramme ist auch graphisch zu erkennen. So haben die Nachrichten hier vier mögliche Zustände, welche über den bereits vorgestellten Stereotypen *ModalMessage* einstellbar sind [2].

Abbildung 2.4: Die zwei Attribute mit ihren Einstellmöglichkeiten.

Eine der zwei einstellbaren Attribute für die Nachricht enthält die Auskunft ob diese *hot (rot)* oder *cold(blau)* ist. Hierbei steht *hot* für einen Vermerk, dass diese Mitteilung auf keinen Fall verletzt werden darf, wohingegen *cold* dafür steht, dass eine Verletzung akzeptabel ist. Das zweite Attribut, welches sich dadurch kennzeichnet ob ein Pfeil durchgezogen ist, gibt die Einstellung *executed (durchgezogen)* oder *monitored (nicht durchgezogen)* an. *Executed* steht hierbei für die Tatsache, dass diese Nachricht eine

Lebendigkeitsanforderung ist, sie muss passieren. Bei *monitored* hingegen, wäre die Tatsache, dass eine Nachricht nicht passiert hinnehmbar [2].

2.5.3 Übersetzung des Beispiel in MSD

Wenn dieses Beispiel in MSD übersetzt wird, muss man den Modalitäten besondere Beachtung schenken, um ein korrektes Diagramm zu erstellen.

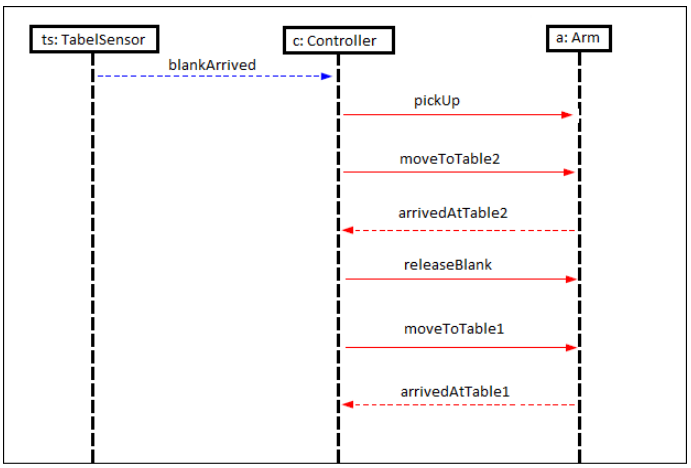

Abbildung 2.5: MSD der Produktionsmaschine ohne Wahrscheinlichkeiten.

Wie in Bild 3 zu erkennen, kommt im ersten Schritt ein Teller *(Blank)* beim Tischsensor an, welcher die Nachricht *blankArrived* an den Controller weiterleitet. Dieser wiederrum sendet die Nachricht *pickUp* an den Roboterarm, welcher daraufhin den Teller aufnimmt. Dann bekommt der Arm vom Controller den Befehl *moveToTable2*, woraufhin der Arm sich Richtung Laufband B bewegt und eine Rückmeldungsnachricht sendet *arrivedAtTable2* sobald er beim Laufband B angekommen ist. Im Anschluss folgen die Befehle des Controllers an den Arm *releaseBlank*, der Teller wird losgelassen, und *moveToTable1*, welcher dem Arm das Signal gibt, dass er wieder zu Laufband A schwenken soll. Wie nach der ersten Schwenkbewegung sendet der Arm nach seiner Ankunft bei Laufband A die Nachricht *arrivedAtTable1* an den Controller. Nach Erhalt dieser Rückmeldung, ist der Empfang eines weiteren Tellers wieder möglich, sodass die Maschine weiter zyklisch arbeiten kann.

2.5.4 Aufteilung der MSD

MSD lassen sich noch weiter kategorisieren, je nachdem was konkret modelliert wurde. So werden Anforderungen an das System in sogenannten *Requirement MSDs* gespeichert, wohingegen Annahmen über die Umwelt in sogenannten *Assumption MSDs* hinterlegt werden. (Verweis: The Scenario-

Tools Play-Out of Modal Sequence Diagram Specifications with Environment Assumptions)

2.6 ScenarioTools

ScenarioTools [1]ist ein Projekt, welches es über durch Nutzung eines UML-Editors ermöglicht MSDs und deren Spezifikationen zu erstellen und im Rahmen der Dissertation von Prof. Dr. Joel Greenyer [7] entstand. Es eignet sich somit zur Modellierung und Analyse szenariobasierter Spezifikationen und bietet die Möglichkeit zur Simulation der Spezifikationen. Diese Simulation findet mit Hilfe eines Play-Out Algorithmus (Verweis: ScenarioTools Real-Time Play-Out for Test Sequence Validation in an Automotive Case Study) statt, welcher auf der Idee des in dem Buch *„Come Let's play“* [4] geschriebenen play-out für ein GUI fundiert.

2.6.1 Play-Out

Dieser Algorithmus wird verwendet, um zu überprüfen, ob sich ein modelliertes System so verhält wie es das Ziel war. Sollte dem nicht so sein, liegen Wiedersprüche, die Anforderungen sind nicht korrekt, oder die Spezifikation wurde fehlerhaft implementiert [6] [3].

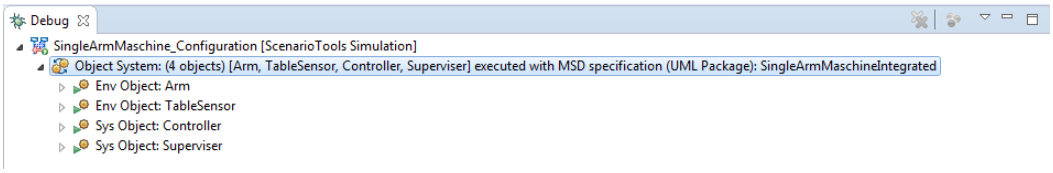

Abbildung 2.6: Debug Anzeige beim Start eines neuen Play-Outs.

Der Algorithmus funktioniert, indem in jedem Schritt eine Nachricht gesendet wird, wobei davon ausgegangen wird, dass das System schneller agiert als die Umwelt.

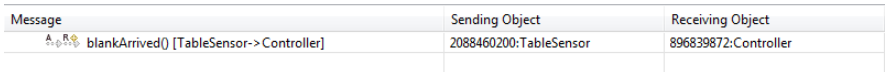

Abbildung 2.7: Nachrichtenauswahl.

Der Benutzer muss aus denen zur Auswahl gestellt Nachrichten eine auswählen, welche dem System übergeben wird. Mit dieser Nachricht als Eingabe fährt der Algorithmus dann fort und gibt im nächsten Schritt weitere Nachrichten zur Wahl an. Mitunter können auch mehrere MSDs gleichzeitig aktiv sein, welche mögliche Eingaben bereitstellen.

[1]http://scenartiotools.org

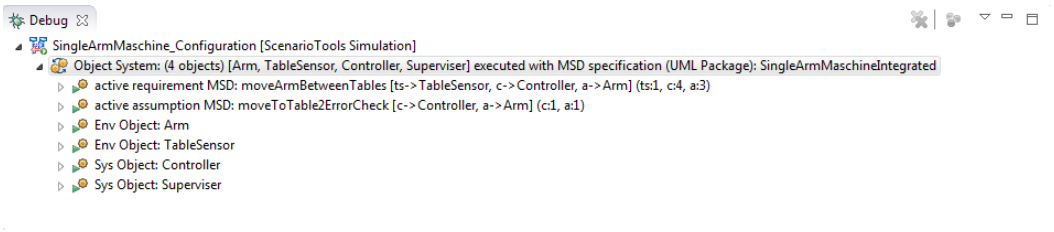

Abbildung 2.8: Debug Anzeige mit mehreren aktiven MSDs.

Während der Simulation können nur Events gewählt werden, welche in aktiven Diagrammen enabled sind. Dies dient dazu violations zu vermeiden.

Message	Sending Object	Receiving Object	Parameter	enabled in Active MSD
blankArrived() [TableSensor->Controller]	2088460200:TableSensor	896839872:Controller		
arrivedAtTable1() [Arm->Controller]	1673626372:Arm	896839872:Controller		
arrivedAtTable2() [Arm->Controller]	1673626372:Arm	896839872:Controller	moveArmBetweenTables...	moveArmBetweenTables...
failureDetected() [Arm->Controller]	1673626372:Arm	896839872:Controller	moveToTable2ErrorCheck	moveToTable2ErrorCheck

Abbildung 2.9: Nachrichtenauswahl mit mehreren Auswahlmöglichkeiten.

Kapitel 3

Konzepte

3.1 Entwicklung eines geeigneten Beispiels

Im ersten Arbeitsschritt haben ich nach einem geeigneten Beispiel gesucht, welches ich zu modellieren und implementieren vorgehabt habe. Dieses Beispiel musste einige Eigenschaften erfüllen um sich als zweckmäßig zu erweisen. Zum einen durfte es nicht zu komplex sein, um spätere Datenausgaben einfacher auf ihre Korrektheit überprüfen zu können. Zum anderen, musste es Möglichkeiten bereitstellen, die den Einsatz von Wahrscheinlichkeiten rechtfertigen würden, da die Implementierung dieser eine der Hauptaufgaben dieser Bachelorarbeit ist.

Im Endeffekt, haben wir uns darauf geeinigt zwei Beispiele genauer zu untersuchen, die als vielversprechend erschienen. Das erste Beispiel, auf welches ich schon in 2.1.1 eingegangen bin, ist ein Ampelgesteuerter Bahnübergang für Fahrzeuge. Die Idee hinter diesem Übergang war, dass Sensoren zu beiden Seiten des Übergangs positioniert sind, die einen heranfahrenden Zug melden würden. Daraufhin würde durch einen dritten Sensor überprüft werden ob sich gerade Fahrzeuge auf dem Übergang befinden. Sollte dies der Fall sein, gerät das System in eine Schleife bei der der Sensor solange überprüft wird, bis sich keine Fahrzeuge mehr auf dem Übergang befinden. In der Zwischenzeit hätte eine Ampel dem Zug signalisiert, dass dieser zu halten habe. Sobald der Sensor signalisieren würde, dass sich kein Fahrzeug mehr auf dem Übergang befindet, wären die Bahnschranken heruntergelassen worden, die Ampel des Zuges hätte die Erlaubnis zur Weiterfahrt gegeben und eine Ampel am Übergang hätte den Fahrzeugen, die vor den Schranken stehen ein Durchfahrverbot angezeigt.

Bei diesem Beispiel hätte es eine Anzahl von Möglichkeiten gegeben, Wahrscheinlichkeiten zu implementieren. Unter anderem wäre es möglich gewesen zu sagen, dass Sensoren nur in 99,99% der Fälle das richtige Ergebnis messen, oder die Schranke sich nur in 99,8% der Fälle tatsächlich senkt, sei es aus mechanischen oder elektronischen Versagen.

Als zweites Beispiel haben wir die vereinfachte Produktionsanlage gewählt, welche bereits in 2.1.2 und den Abbildungen 2.2 und 2.3 genauer erläutert wurde.

Die Implementierung von Wahrscheinlichkeiten, wäre bei diesem Beispiel gegeben, sobald man die Möglichkeit des Fehlverhaltens des Roboterarms in Betracht zieht. So erscheint es durchaus realistisch, dass der Roboterarm, beim Schwenken nur zu 99,9% sein Vorgegebenes Ziel erreicht und in 0,1% einen Fehler produziert.

Wie eben dargestellt bieten beide Beispiele die Gelegenheit für Möglichkeit Wahrscheinlichkeiten zu implementieren und sind in diesem Sinne beide geeignet, um als Evaluationsobjekte angesehen zu werden. Durch die Tatsache, dass das erste Beispiel wesentlich komplexer ist, ohne jedoch mehr oder bessere Ergebnisse zu erzielen, haben wir beschlossen, unser Hauptaugenmerk auf die vereinfachte Produktionsmaschine zu lenken, die ich im Folgenden weiter bearbeitet habe.

3.2 Ausarbeitung des Beispiels

Bei der Implementierung des Beispiels mit der vereinfachten Produktionsmaschine bin ich in zwei einzelnen Schritten vorgegangen. Zuerst haben wir die nötigen MSD aufgestellt und nach deren Fertigstellung und Überprüfung auf Korrektheit habe ich diese als Vorlage benutzt, um die MSD in ScenarioTools korrekt zu implementieren. Die bereits in den Grundlagen vorgestellte Produktionsmaschine besteht aus einem einzelnen MSD (Abbildung 2.5), welches alle erforderlichen Daten des Produktionszykklus bereitstellt. Sobald jedoch bei diesem Beispiel die Möglichkeit von Wahrscheinlichkeiten betrachtet wird, verkomplizieren sich die MSD dementsprechend und an Stelle von einem einzelnen MSD werden vier benötigt, um den Anforderungen gerecht zu werden. Diese sind in den Abbildungen A.1, A.2, A.3 und A.4 des Anhangs zu sehen.

Es gibt es eine starke Ähnlichkeit zwischen dem MSD ohne Wahrscheinlichkeiten 2.5 und dem mit Wahrscheinlichkeiten A.1. Dies liegt daran, dass sich an der Grundfunktion der Produktionsmaschine nichts geändert hat, sondern nur neue, bisher nicht erfasste, Eigenschaften implementiert werden. Eine der grundlegenden Änderungen, die ich hier vollführt habe, ist eine Abfrage vor der *pickup*-Nachricht, welche überprüft ob die *failureOccured*-Variable der *Controller* Klasse auf *false* gesetzt ist. Sollte dem so sein, wird der Produktionszyklus wie bisher weiter durchgeführt. Sollte die *failureOccured*-Variable jedoch auf *true* stehen, hieße dies, dass ein Fehler vorgefallen ist und die Maschine keine weiteren Teller verarbeiten soll, bis der Fehler behoben wurde. Der Ablauf wäre somit an dieser Stelle unterbrochen. Die zweite Änderung ist die Einführung eines *forbidden*-Operators, welcher eine *failureDetected*-Nachricht enthält. Diese bedeutet lediglich, dass in

diesem MSD keine *failureDetected*-Nachricht verwendet werden darf.

Unschwer zu erkennen ist, dass sich Abbildung A.2und A.3 in großen Teilen gleichen. Dies rührt daher, dass diese MSD dieselbe Funktion erfüllen, lediglich an unterschiedlichen Stellen des Produktionszyklus. In beiden MSD kommt zuerst eine *moveToTable1*- bzw. *moveToTable2*-Nachricht, welche den Befehl des Controllers an den Roboterarm zum Schwenken darstellt. Danach kommt ein *alternate*-Operator, welcher die Nachrichten *moveToTable1* / *moveToTable2* sowie eine*failureDetected*-Nachricht enthält. Hierbei handelt es sich um die Repräsentation der Implementierung der Fehleranfälligkeit des Systems. Sollte das System die *failureDetected*-Nachricht erhalten, zeigt dies an, dass beim Schwenken des Arms ein Fehler unterlaufen ist, welcher dafür sorgt, dass der Arm nicht in der gewünschten Position ankommt. Anschließend gibt es noch einen *forbidden*-Operator, in welchem verbotene Nachrichten, wie in Bild 2.5, angezeigt werden.

Das vierte und somit letzte MSD A.4, das wir erstellt haben, um den Anforderungen gerecht zu werden, drückt die Abfolge aus, welche gewählt werden muss, sollte in Abbildung A.2 oder A.3 der Roboterarm die *failureDetected*-Nachricht verschicken. Wie hier zu erkennen ist, wird nach der Ankunft der Nachricht eine weitere Mitteilung vom Controller an sich selbst geschickt, mit deren Hilfe der interne Boolean *FailureOccured* auf *true* gesetzt wird, sodass wie in Abbildung A.2 nach dem Ankommen von weiteren Tellern nicht weitergearbeitet wird, bis der wieder zurückgesetzt wurde. Des Weiteren wird eine *failureOccured*-Meldung an den zuständigen Supervisor gesendet, sodass dieser die Maschine entsprechen zurücksetzen oder reparieren kann.

3.2.1 Implementierung in ScenarioTools

Diese MSD konnten, so wie hier dargestellt, eins zu eins in ScenarioTools übernommen werden. Es folgten noch die Erweiterung um ein Klassendiagramm, sowie ein CompositeDiagram, in welchem ich die vier Klassen (*TableSensor, Arm, Controller, Supervisor*) zusammengefasst habe. Außerdem wurde noch in „Overview"-Diagramm von mir erstellt, in welchem die MSD, welche in verschiedene Packages aufgeteilt wurde, durch mergen zu verbinden.

3.3 Implementierung der Wahrscheinlichkeiten

Meine grundlegende Idee bei der Implementierung der Wahrscheinlichkeiten war, dass ich mich dem Stereotypen-System bediene. Das Ziel hierbei war es, eine Möglichkeit zu schaffen, einer Message einen neuen Stereotypen zuzuweisen, welcher eine Wahrscheinlichkeit enthält. Der Genauigkeit halber habe ich beschlossen an Stelle von einem *Double*, welcher mir die Wahrscheinlichkeit als Kommazahl angegeben hätte, zwei *Integer* zu verwenden,

welche einen Bruch formen. Auf diese Idee kam ich bei der Entwicklung des Algorithmus, welcher die Wahrscheinlichkeiten interpretieren sollte. Da diese Bachelorarbeit als Teil einer Lösung dienen soll, welche später eine Zuverlässigkeitsanalyse abgeben soll, bin ich zu dem Schluss gekommen, dass es besonders bei den Wahrscheinlichkeiten auf Genauigkeit ankommt. Ein Beispiel hierfür ist, wenn der Roboterarm zu $\frac{1}{9}$ einen Fehler verursacht, so wäre das äquivalente Double hierfür ca. 0,111. Da bei $\frac{1}{9}$ die Einsen jedoch periodisch weitergeführt werden, ist 0,111 keine ausreichende Annäherung, welche der Genauigkeit von Brüchen gerecht wird.

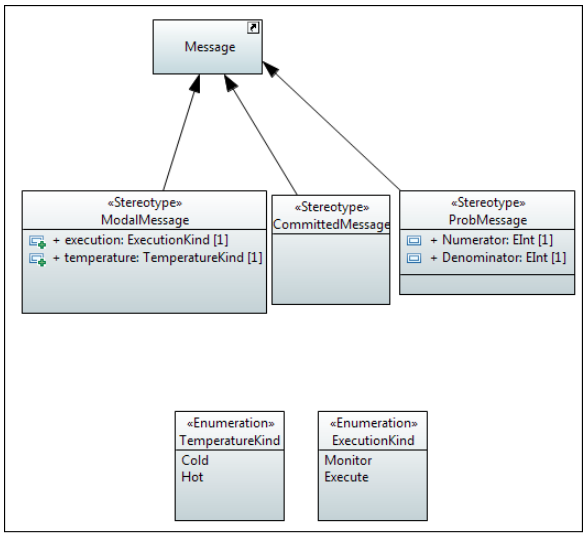

Abbildung 3.1: Auf Message anwendbare Stereotypen.

Das Ergebnis dieser Überlegung ist in Bild 3.1 dargestellt. Es wurde von mir ein neuer Stereotyp *ProbMessage*, Kurzform für probabilistic Message, erstellt, welcher *Message* hinzugefügt wurde. Dieser Stereotyp enthält die beiden Integer *Numerator* und *Denominator*, die Zähler und Nenner eins Bruches darstellen. Nach dem verbinden des Stereotypen mit *Message*, lässt sich durch einfaches hinzufügen des Stereotypens zu einer Nachricht, dieser Message ein Bruch zuordnen.

3.4 Entwicklung des Algorithmus

Als Konzept hatten wir bei der Interpretation der Wahrscheinlichkeiten, welche ich an den drei Beispielen in Bild A.5 verdeutlichen möchte.

Bei Beispiel A handelt es sich um eine Entscheidung wie sie in Bild A.2 und A.3 zu sehen ist. Eine Entscheidung zwischen zwei Alternativen wie: „kommt der Roboterarm an“ und „passiert ein Fehler“. Bei dem ersten Konzept hätten wir gesagt wir lassen uns eine zufällige Zahl zwischen null

und eins geben, ist diese Zahl größer als 0,5 so soll C gewählt werden, ansonsten B.

Bei Beispiel B ist die Entscheidung minimal anders, es gibt eine dritte Alternative zu B und C, somit hätte der Algorithmus angepasst werden müssen, sodass er eine zufällige Zahl zwischen null und eins wählt und dann überprüft wie viele verschiedene Auswahlmöglichkeiten es gibt. Bei drei müsste dann geprüft werden ob die Zahl unter 1/3 ist, dann soll B gewählt werden, zwischen 1/3 und 2/3 liegt, dann soll C gewählt werden oder über 2/3, in welchen Fall C gewählt werden soll. Wie man sieht, wird eine solche Überprüfung wesentlich komplexer.

Das dritte Beispiel, zeigt die Schwächen dieses Konzepts sehr deutlich auf. Hier wird davon ausgegangen, dass es - wie in Beispiel B - drei Alternativen gibt: B, D und E. Um zu verdeutlichen, wie das Beispiel zu verstehen ist, kann man sich vorstellen, dass eine Münze geworfen wird. Es besteht eine 50% Chance, dass die Münze nicht auf dem Boden landet, welches zu Endpunkt B führen würde. Sollte die Münze auf dem Boden landen, Punkt C, besteht eine 50:50 Chance, welche Seite oben liegt, Punkt D und E. Wie also zu sehen ist, sind die Chancen: B = 50%, D = 25% und E = 25%. Bei der Implementierung müsste also beachtet werden, ob Nachrichten in irgendeiner Form zusammen hängen und sollte dies so sein, so müssten diese gruppiert werden um korrekte Ergebnisse zu erzielen. Da ich der Meinung war, dass es eine einfachere Methode geben muss habe ich zwei weitere Konzepte für den Algorithmus entworfen.

Zur besseren Erläuterung der ersten Idee habe ich in Bild A.6 einen Beispielpfad ausgewählt. Das ursprüngliche Beispiel A (Beispiel C in Bild A.5) wird modifiziert, sodass keine Gruppierung mehr nötig ist, sondern alle Endzustände im ersten Schritt dieselbe Auswahlwahrscheinlichkeit von 33% haben. Der signifikante Unterschied ist im Folgenden zu erkennen. Von Zustand C und D gibt es jeweils zwei weitere Zustände: A und D. A steht hierbei für *accept* und D für *decline*. Die Idee ist, dass sobald ein Zustand akzeptiert wird, dieser ausgewählt und sobald ein Zustand abgewiesen wird, dieser aus dem Graphen gelöscht wird. In diesem Beispiel wird zufällig Zustand C angewählt, welche 90%ige Chance hat, akzeptiert zu werden. Danach wird wiederum zufällig entschieden, dass der Zustand abgewiesen und gelöscht wird. Im folgenden Schritt wird dann Zustand B angewählt. Da dieser keine Wahrscheinlichkeiten enthält, wird er direkt akzeptiert. Die Idee hinter diesem Konzept war, die Gruppierung überflüssig zu machen, um Arbeitsschritte zu sparen, sowie die Implementierung zu vereinfachen.

Die zweite Konzeptidee grundiert auf der eben erklärten Idee, geht aber bei der Ausführung in eine andere Richtung. Wie in Bild A.7 zu erkennen, nehmen wir dasselbe Beispiel, welches wir auch zur Erläuterung des letzten Algorithmus verwendet haben. Graue Wege sind nach rechts begehbar, blaue nach links. Der Unterschied ist, dass es von den Zuständen C und D jeweils einen Weg zu einem „A“-Zustand gibt. welcher, wie im letzten Fall auch,

für das akzeptieren des vorherigen Zustandes nötig ist. Was jedoch neu ist, ist die Tatsache, dass an Stelle von einem Weg zu einem ablehnenden Zustand, welcher die Löschung des vorherigen Zustandes im letzten Beispiel zu Folge hätte, ein Pfad zurück zum Ausgangszustand A vorhanden. Um das Ganze zu verdeutlichen, wird in Bild A.7 Zustand C zufällig ausgewählt. Daraufhin wird per Zufall entschieden, dass der Zustand abgelehnt wird und wir uns wieder im Anfangszustand befinden. Im letzten Schritt, wird dann Zustand B ausgewählt und, da dieser keine Wahrscheinlichkeit enthält, direkt akzeptiert. Der Vorteil hierbei ist, dass nach der Ablehnung die Menge der möglichen Zustände nicht mehr verändert werden muss, sondern die Funktion, welche die nachfolgende Nachricht auswählt, nur noch einmal aufgerufen werden muss.

Aus eben genannten Gründen habe ich mich entschlossen, das zuletzt vorgestellte Konzept weiter zu verfolgen und zu implementieren. Zuallererst habe ich jedoch überprüft, ob die Wahrscheinlichkeiten der Endzustände übereinstimmen. Dies geschah um sicher zu gehen, dass sich die Veränderung der Abarbeitung keineswegs auf das Ergebnis auswirkt. Um dies zu berechnen wurde von mir aus dem gewählten Beispiel eine Markov-Kette A.8 erstellt, für welche ich im Anschluss eine statistische Verteilung berechnet habe.

Zur Berechnung der statistischen Verteilung muss eine 6x6 Übergangsmatrix sowie eine 1x6 Matrix erstellt werden, in welcher die Wahrscheinlichkeiten mitgeführt werden. Diese ist in Abbildung A.9 aufgeführt.

Anschließend wird die Wahrscheinlichkeitsmatrix mit der Übergangsmatrix wiederholt multipliziert, um eine Annäherung zu bestimmen. Die Startwahrscheinlichkeiten der Übergangsmatrix können Abbildung A.10 entnommen werden.

Wie man deutlich sieht, sind nach 8 Schritten die Wahrscheinlichkeiten denen von Beispiel A aus Bild A.7 sehr ähnlich. B = 50%, D = 45% und E = 5% im Vergleich zu denen Zuständen hier B = 49,38%, D = 44,4% und E = 4,9%. Man kann also erkennen, dass bei weiteren Durchläufen der Markov-Kette, sich die Wahrscheinlichkeiten immer mehr denen annähenern, wodurch sich dieses Konzept als funktionsfähig erwiesen hat. In Abbildung A.11 werden die Wahrscheinlichkeiten nach 1-8 Schritten angegeben.

Kapitel 4

Implementierung

Hierbei handelt es sich um den wichtigsten Schritt, der Umsetzung von Theorie in Praxis. Am Anfang beschreibe ich auf welchen Grundlagen ich arbeiten und mein Konzept verfolgen kann. Es existierte bereits eine erste Version eines automatischen Durchlaufs. Dieser funktionierte, indem eine *ArrayList<ModalMessageEvent>* erstellt wurde, in welcher erlaubte *ModalMesssageEvents* gespeichert werden. Von diesen ausgewählten Nachrichten wird dann mit Hilfe der Funktion *Math.random()* ein zufälliges Event ausgewählt und als nächstes ausgeführt. Dies geschieht, indem der *Double* der *Math.random()* Funktion mit der Länge der *Numerator*ArrayList multipliziert wird. Diese Funktionalität wurde zweifach implementiert, in Form von zwei *Agenten*, die die Nachrichten verwalten und auswählen. Zum einen gibt es den *RandomSystemSimulationAgent*, welcher, wie der Name schon vermuten lässt,*MessageEvents* auswählt, sollte eine Eingabe des Systems gefragt sein. Im anderen Fall ist der*RandomEnviromentAgent* zuständig. Dieser ist für Auswahl der *MessageEvents* der Umwelt verantwortlich.

Mein Ursprüngliches Konzept, die zufällig ausgewählte Nachricht darauf zu prüfen, ob sie Wahrscheinlichkeiten enthält, hat sich jedoch nicht ohne Probleme durchführen lassen. Dies lag an der Tatsache, dass die Agenten, so wie sie derzeit implementiert sind, keinen direkten Zugriff auf die UML die Stereotypen als UML Konstrukt haben. Die Messages, auf welche die Agenten Zugriff haben, gehören zum Runtime Modell.

Um dieses Problem zu umgehen, habe ich eine *EList<ActiveProcess>* erstellt, in welcher aktive Prozesse gespeichert werden. Diese Liste wird erstellt, indem die *ArrayList<ModalMessageEvent>* an der zufällig ausgewählten Stelle daraufhin überprüft wird, in welchen aktiven Prozessen das *ModalMessageEvent* aktiv ist. Die *Elist<ActiveProcess>* wird daraufhin durchlaufen und jeder aktive Prozess wird daraufhin kontrolliert, ob es sich um ein *ActiveMSD* handelt. Ist dies der Fall, so werden alle erlaubten Nachrichten in einer weiteren *for*-Schleife durchlaufen. Innerhalb dieser Schleife habe ich dann das eigentliche Prinzip des Algorithmus implementiert, da

an dieser Stelle Zugriff auf die übergebenen *Integer* und somit auf die Wahrscheinlichkeiten gegeben ist. Im ersten Schritt wird verifiziert, dass es sich bei dem zufällig in der *ArrayList* ausgewählten Element und dem derzeitig in der *EList* betrachteten um dasselbe handelt, ansonsten wird dieses Element nicht weiter betrachtet. Konnte dies bestätigt werden, wird abgefragt ob die Message eine Wahrscheinlichkeit in Form eines Bruches enthält. Sollte dem nicht so sein, wird das Element als Eingabe für den nächsten Schritt des automatischen Durchlaufs gewählt. Für den Fall, dass diese Nachricht eine Wahrscheinlichkeit enthält, wird der Nenner mit einer zufälligen Zahl zwischen 0-1 multipliziert und mit dem Zähler abgeglichen. Wenn der Zähler den höheren Wert enthält, ist das derzeitige Element wiederrum als Eingabe zulässig. Hat jedoch der Zähler den geringeren Wert, wird die Funktion rekursiv aufgerufen, um die Funktionsweise des Algorithmus zu gewährleisten.

Kapitel 5

Verwandte Arbeiten

PRISM[1] ist eine *free* und *open source* Software. Im Jahre 2002 wurde PRISM als Werkzeug vorgestellt, mit welchem Wahrscheinlichkeitssysteme analysiert werden können. Zu der Zeit wurden drei Wahrscheinlichkeitsmodelle unterstützt: Markov-Ketten diskreter Zeit, Markov-Entscheidungsprozesse und Markov-Ketten kontinuierlicher Zeit. PRISM hatte drei Möglichkeiten zur Modelüberprüfung: symbolisch, dünnbesetzte Matrizen und ein Mix aus symbolisch und dünnbesetzten Matrizen [10].

Seitdem wurde PRISM stätig erweitert [11] [1] und es ist inzwischen eine sehr gute Alternative um Wahrscheinlichkeitsmodelle zu analysieren.

Eine große Schwäche von PRISM gegenüber ScenarioTools ist jedoch, dass bei der Entwicklung eines Systems generell Sequenzdiagramme entworfen werden. Diese können mit wenig Aufwand in MSDs umgewandelt werden, welche wiederrum als Eingabe für ScenarioTools genutzt werden können. Mit dieser Eingabe, welche mit sehr geringer Arbeit erstellt werden kann, kann anschließend eine Verlässlichkeitsanalyse durchgeführt werden, welche zu einem aussagekräftigen Ergebnis führt. Für PRISM hingegen müssen erst eigenen Modelle erstellt werden, welche eher abstrakt wirken, im Vergleich zu der Abfolge an Ereignissen eines Sequenzdiagrammes. Somit müsste für die Umsetzung in PRISM ein Mehraufwand betrieben werden welcher sich nicht rechtfertigen lässt.

Des Weiteren stellt sich die Frage warum etwas in zwei arbeitsreichen Schritten erledigt werden sollte, wenn es auch in einem möglich ist. Zumal dieser eine Schritt, die Umsetzung und das play-out in ScenarioTools, auch aus anderen Gründen gemacht, nämlich der Überprüfung auf Widersprüche.

Schlussendlich lässt sich sagen, dass PRISM in diesem Zusammenhang höchstens über eine direkte Einbindung in ScenarioTools, z.B. mit Hilfe einer automatischen Codegenerierung, als nützlich erweisen würde.

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

Kapitel 6

Zusammenfassung und Ausblick

6.1 Zusammenfassung

Hauptziel dieser Bachelorarbeit war es die Darstellung von Wahrscheinlichkeiten innerhalb von ScenarioTools zu ermöglichen, sowie eine Anpassung des Simulationsalgorithmus, sodass dieser die angegebenen Wahrscheinlichkeiten interpretieren kann. Gelöst wurde das Problem durch hinzufügen eines neuen Stereotypen, welcher es einer Nachricht erlaubt zwei *Integer* hinzuzufügen, welche einen Bruch ergeben. Anschließend habe ich ein Konzept für einen Algorithmus entworfen, welcher den Wahrscheinlichkeiten entsprechend handelt. Dieses Konzept habe ich danach umgesetzt.

Nach Fertigstellung der Umsetzung sind ScenarioTools nun so gut wie alle Werkzeuge gegeben um eine aussagekräftige Verlässlichkeitsanalyse durchzuführen.

6.2 Ausblick

Die bereits von mir implementierten Module lassen sich ohne großen Aufwand erweitern. Besonders interessant wäre es, bei dem automatischen Durchlauf ein System einzubauen, welches Rückmeldungen über den derzeitigen Zustand des Systems gibt. Wenn wir bei dem Beispiel der Produktionsmaschine bleiben, so wäre eine Rückmeldung von Nutzen, die mitteilen würde sobald ein Teller fertiggestellt wurde. Des Weiteren hätte eine auch eine negative Rückmeldung großen Nutzen, welche ausgelöst wird, sobald ein Fehler innerhalb des Systems festgestellt wird. Durch die Implementierung solch eines Feedbacks und den daraus folgenden Informationen könnte eine erste Analyse der Ausfallsicherheit durchgeführt werden. In diesem Fall kann der Durchlauf mehrfach gestartet oder eine weitere, dies durchführende Erweiterung implementiert und die daraus

gewonnenen Daten ausgewertet werden. Bei diesem Beispiel wäre es von Interesse wie viele Teller produziert wurden, bis ein Fehler auftrat.

Anhang A

Anhang

A.1 Modal Sequence Diagrams Bilder

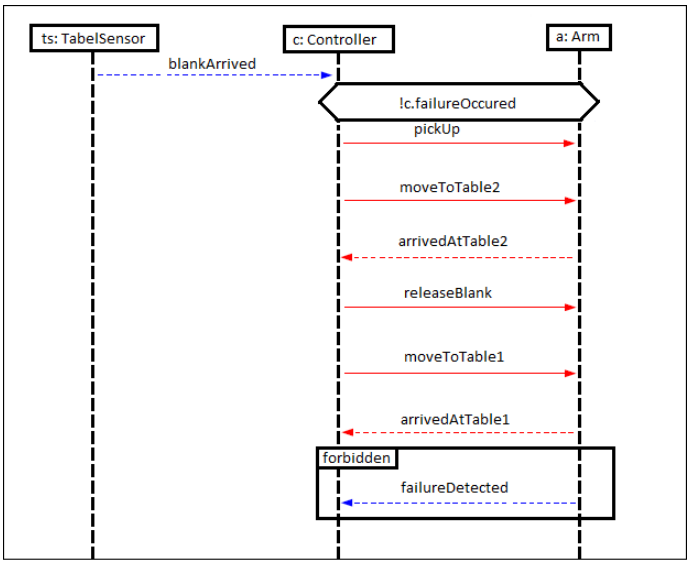

Abbildung A.1: MSD: MoveArmBetweenTables

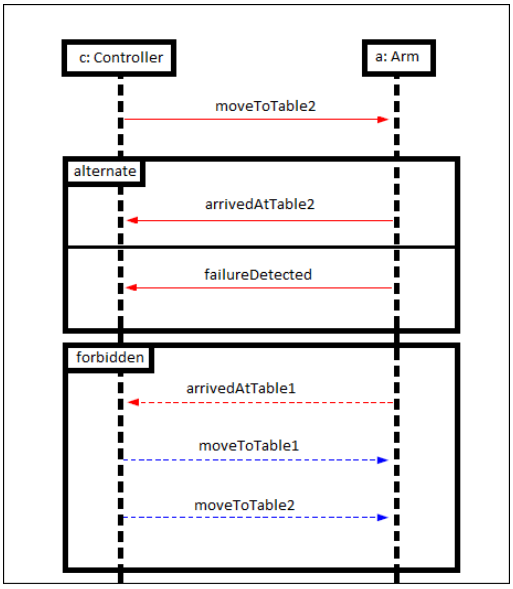

Abbildung A.2: MSD: MoveToTable2ErrorCheck

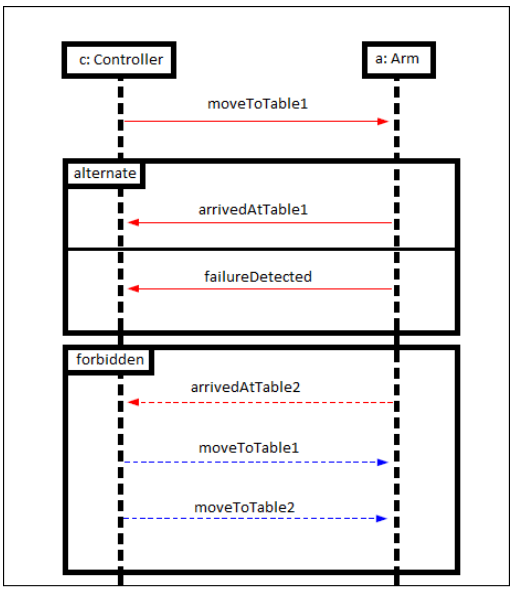

Abbildung A.3: MSD: MoveToTable1ErrorCheck

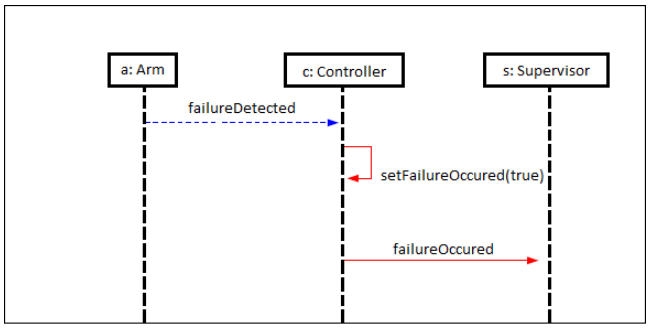

Abbildung A.4: MSD: FailureDetected

A.2 Wahrscheinlichkeitsbäume und Tabellen

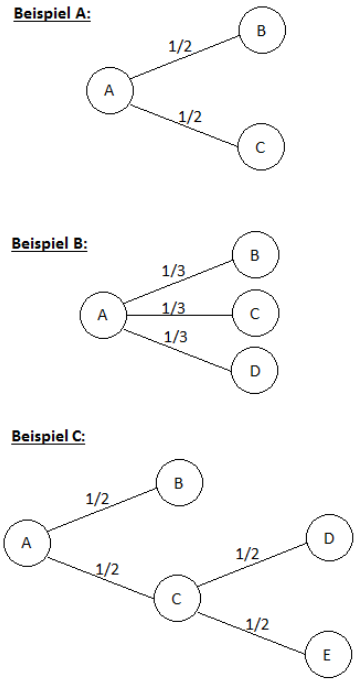

Abbildung A.5: 3 Beispiele an Wahrscheinlichkeitsbäumen.

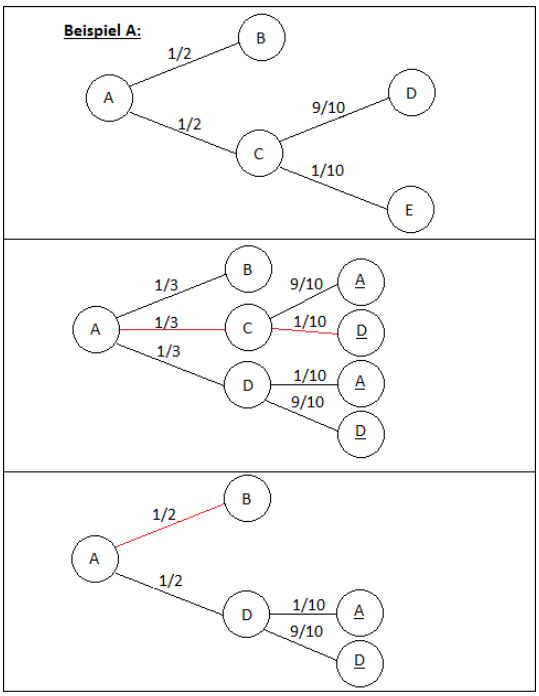

Abbildung A.6: Beispieldurchlauf eines Wahrscheinlichkeitsbaum mit Konzept 1.

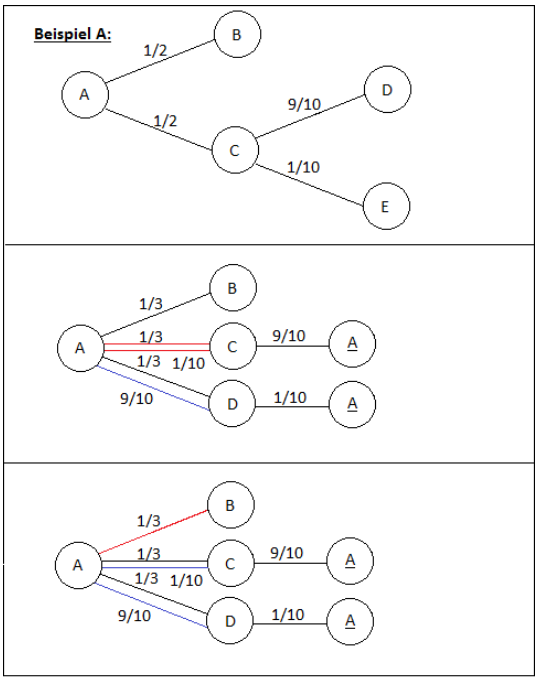

Abbildung A.7: Beispieldurchlauf eines Wahrscheinlichkeitsbaum mit Konzept 2.

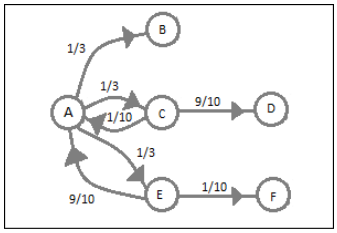

Abbildung A.8: Markov-Kette der Wahrscheinlichkeitsverteilung in Abbildung A.7

0	1/3	1/3	0	1/3	0
0	1	0	0	0	0
1/10	0	0	9/10	0	0
0	0	0	1	0	0
9/10	0	0	0	0	1/10
0	0	0	0	0	1

Abbildung A.9: Übergangsmatrix der Markov-Kette von Abbildung A.8

1	0	0	0	0	0

Abbildung A.10: Startvektor zur Berechnung der Wahrscheinlichkeiten.

1.	0	1/3	1/3	0	1/3	0
2.	1/3	1/3	0	3/10	0	1/30
3.	0	4/9	1/9	3/10	1/9	1/30
4.	1/9	4/9	0	2/5	0	2/45
5.	0	13/27	1/27	2/5	1/27	2/45
6.	1/27	13/27	0	13/30	0	13/270
7.	0	40/81	1/81	13/30	1/81	13/270
8.	1/81	40/81	0	4/9	0	4/81

Abbildung A.11: Ergebnis der Berechnung der statistischen Verteilung der Markov-Kette in Bild A.8 nach 1-8 Schritten .

Literaturverzeichnis

[1] G. N. D. P. Andrew Hinton, Marta Kwiatkowska. Prism: A tool for automatic verification of probabilistic systems. In *Tools and Algorithms for the Construction and Analysis of Systems*, 2006.

[2] J. G. Christian Brenner and V. P. L. Manna. The scenariotools play-out of modal sequence diagram specifications with enviroment assumptions. In *Proceedings of the 12th International Workshop on Graph Transformation and Visual Modeling Techniques (GTVMT 2013)*, volume 58. EASST, 2013.

[3] J. H. G. L. G. S. M. T. Christian Brenner, Joel Greenyer. Scenariotools real-time play-out for test sequence in an automotive case study. In *Proceedings of the 13th International Workshop on Graph Transformation and Visual Modeling Techniques (GTVMT 2014)*, volume 67. EASST, 2014.

[4] R. M. David Harel. *Come, Let's Play.* Springer Verlag, 2003.

[5] E. Dubrova. *Fault-Tolerant Design.* Springer Verlag, 2013.

[6] J. Greenyer. Synthesizing modal sequence diagram specifications with uppaal-tiga. Technical report, Software Engineering Group, Department of Computer Science, University of Paderborn, February 2010.

[7] J. Greenyer. *Scenario-based Design of Mechatronic Systems.* PhD thesis, Universität Paderborn, October 2011.

[8] O. M. Group. Uml 2.4.1 infrastructure, August 2011.

[9] O. M. Group. Uml 2.4.1 superstructure, August 2011.

[10] D. P. Marta Kwiatkowska, Gethin Norman. Prism: Probabilistic symbolic model checker. In *Computer Performance Evaluation*, 2002.

[11] D. P. Marta Kwiatkowska, Gethin Norman. Probabilistic symbolic model checking with prism: a hybrid approach. *International Journal on Software Tools for Technology Transfer*, 2004.