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

IAV GmbH
Ingenieurgesellschaft Auto und Verkehr
VD-D1 Embedded System Software

Erprobung einer formalen Methode zur Anforderungsanalyse in der Automobilsoftwareentwicklung am Beispiel einer Schnittstelle für Hochvolt-Nebenaggregate

Bachelorarbeit

im Studiengang Mechatronik

von

Maximilian Haase

Prüfer: Prof. Dr. Joel Greenyer
Zweitprüfer: Prof. Dr. Kurt Schneider
Betreuer: Prof. Dr. Joel Greenyer
Dipl.-Ing. René Bellmer

Gifhorn, 24. März 2015

Inhalt

Sperrvermerk i

Inhalt i

Abbildungsverzeichnis v

Tabellenverzeichnis vi

Abkürzungsverzeichnis vii

1 Einleitung und Motivation 1

1.1 Struktur der Arbeit 4

2 Grundlagen 7

2.1 Anforderungen 7

2.1.1 Funktionale Anforderungen 7

2.1.2 Qualitätsanforderungen 7

2.1.3 Randbedingungen 7

2.1.4 Template 7

2.2 Sprachformen 8

2.2.1 Natürliche Sprache 8

2.2.2 Formale Sprache 8

2.2.3 Halbformale Sprache / Semiformal Sprache 8

2.3 Szenariobasierte Anforderungsanalyse 8

2.4 Unified Modeling Language 9

2.4.1 Stereotypen 9

2.4.2 Klassendiagramm 9

2.4.3 Kompositionsstrukturdiagramm 10

2.4.4 Sequenzdiagramme und Lebenslinien 10

2.4.5 Kombinierte Fragmente 11

2.4.6 Object Constrained Language 11

2.5 Modale Sequenz Diagramme 12

2.5.1 Objektsystem 12
2.5.2 Nachrichten, Events und der Cut 13
2.5.3 Bedingungen 15
2.6 ScenarioTools 15
2.6.1 Play-Out 15
2.6.2 Synthese 16
3 Die *flex*^{Pi}-Schnittstelle 17
3.1 Konzept 17
3.2 Anwendungsbereiche 17
3.3 Besondere Anforderungen 18
3.4 Komponenten der Schnittstelle 19
4 Anforderungserstellung 21
4.1 Gefahren- und Risikoanalyse 22
4.2 Anforderungsliste 22
4.3 Formulierung einzelner Anforderungen 23
4.4 Beispiel 23
5 Modellierung 27
5.1 Arten von Anforderungen 27
5.2 Modellierung des Systems 28
5.3 Modellierung der Anforderungen 30
5.3.1 Aktionen als Auslöser 30
5.3.2 Attributsänderungen als Auslöser 31
5.3.3 Aktion und Attributswert als Auslöser 32
5.3.4 Mehrere Attribute als Auslöser 33
5.3.5 Alternative Aktionen als Auslöser 34
5.3.6 Quasiparallele Aktionen als Auslöser 35
5.3.7 Aktion als Reaktion 36
5.3.8 Änderung eines Attributs als Reaktion 36

5.3.9 Antiszenarien ... 37
5.3.10 Verhalten einschränken ... 38
5.4 Modellierung der Annahmen ... 39
5.4.1 Umweltannahmen ... 40
5.4.2 Systemannahmen ... 40
5.5 Gefundene Lücken durch Modellierung ... 41
5.6 Zusätzliche Sprachelemente ... 43
5.6.1 Schleifen ... 43
5.6.2 Parallele Abläufe ... 43
6 Synthese eines Automaten und Anforderungsanalyse ... 45
6.1 Synthese ... 45
6.1.1 Zustände ... 47
6.1.2 Dynamisches Objektsystem ... 47
6.1.3 Export ... 48
6.1.4 Counter-Strategie ... 48
6.1.5 Nicht erreichte MSDs ... 49
6.2 Analyse und Korrekturen ... 49
6.2.1 Fehlerhafte Modellierung ... 49
6.2.2 Untereinschränkung des Environments ... 50
6.2.3 Fehlende Anforderung führt zu Antiszenario ... 51
6.2.4 Widersprüchliche Anforderungen ... 51
6.2.5 Fehlende Anforderung führt zu nicht verwendeten Anforderungen ... 51
6.2.6 Übereinschränkung des Environments ... 52
6.2.7 Reaktivierung von Diagramm ... 52
6.2.8 Fehlende Annahme führt zu Anforderungsverletzung ... 53
7 Fazit und Ausblick ... 55
7.1 Useability ... 55
7.1.1 Papyrus ... 55

7.1.2 ScenarioTools 56
7.1.3 Modellierung durch MSDs 57
7.2 Grenzen und notwendige Erweiterungen 57
7.2.1 Anforderungen 58
7.2.2 Anforderungsformulierungen 58
7.3 Vorteile der Methode 59
7.4 Aufwand 60
7.5 Textuelle MSDs 60
7.6 Testfallableitung 60
Literaturverzeichnis 61
Eigenständigkeitserklärung 63
Anhang 65
A. Fehlerkorrektur 65
B. Codeerweiterungen 67
B.1. Korrigiertes Prüfen der Bedingungen 67
B.2. Violation Bedingung 68
B.3. Zusätzliche Ausgabe des Objektsystems 68
B.4. Zusätzliche Ausgabe der violated MSDs 69
B.5. Umwelteinflussdiagramm und Superstepdiagramme 70
B.6. Cold verbotene Nachrichten 73
B.7. Nicht aktivierte MSDs ausgeben 73
C. Anforderungsliste 74
C.1. Umgebungsannahmen 74
C.2. High Level Requirements 74
C.3. Isolationswächter 74
C.4. Schützansteuerung 74
C.5. Schützklebererkennung 74
C.6. Verriegelung 74

C.7. LEDs 74

C.8. Kommunikation / Interlock 74

C.9. Strom- und Spannungsmessung 74

C.10. Fehlerspeicher 74

C.11. Glosar 74

C.12. Parameter 74

D. Korrekturen 74

D.1. Umweltannahmen 74

Abbildungsverzeichnis

Abbildung 1-1: V-Modell nach VDI 2206 [1] 1

Abbildung 1-2: *flex*^{Pi} in einem E-Fahrzeug 4

Abbildung 2-1: Klasse in UML-Klassendiagramm 9

Abbildung 2-2: Beispiel eines Kompostionsstrukturdiagramms 10

Abbildung 2-3: UML-Sequenzdiagramm 10

Abbildung 2-4: Der Cut in einem MSD 13

Abbildung 2-5: Arten von Nachrichten in MSDs [12] 14

Abbildung 3-1: Übersicht über alle Komponenten der *flex*^{Pi} [5] 19

Abbildung 5-1: oberste Ebene der Modellierung 28

Abbildung 5-2: Klassendiagramm 29

Abbildung 5-3: Collaboration Diagramm 29

Abbildung 5-4: Aufbau einer als MSD dargestellten Anforderung 30

Abbildung 5-5: Aktion als Auslöser 31

Abbildung 5-6: Beispiel für Aktion als Auslöser 31

Abbildung 5-7: Beispiel für Änderung einer Objekteigenschaft als Auslöser 32

Abbildung 5-8: Aktion und Bedingung als Auslöser 32

Abbildung 5-9: Beispiel für Auslösersystem aus Aktion und Objekteigenschaft 33

Abbildung 5-10: "Und"-Verknüpfung zweier Bedingungen als Auslöser 34

Abbildung 5-11: Alternative Auslöser 35

Abbildung 5-12: Parallele Aktionen mit "par" kombiniertem Fragment 35

Abbildung 5-13: Beispiel für Aktion als Reaktion 36

Abbildung 5-14: Beispiel für Änderung einer Objekteigenschaft 37
Abbildung 5-15: Beispiel für Antiszenario 38
Abbildung 5-16: Beispiel für verbotene Nachricht 39
Abbildung 5-17: Umweltannahme, Stecker stecken/ausstecken 40
Abbildung 5-18: HLR10, HLR12, HLR13 nach Korrektur 42
Abbildung 5-19: Paralleler Ablauf mit kombinierten Fragment "par" 43
Abbildung 6-1: MSDs für Synthesebeispiel 46
Abbildung 6-2: Zustandsautomat aus Synthesebeispiel 47
Abbildung A-1: Vorgehen zur Analyse und Korrektur von Fehlern 65
Abbildung B-2: Ursprüngliche Darstellung eines Zustands in exportiertem Zustandsautomaten ... 68
Abbildung B-3: Zustand aus exportiertem Zustandsautomaten mit Objektattributen 69
Abbildung B-4: violated MSD in exportiertem Zustand 70
Abbildung B-5: zusammenlaufende Supersteps 71
Abbildung B-6: Superstepgraph 72
Abbildung D-7: A-MSDs Klemme 15 alternierend 74
Abbildung D-8: Einstecken und Austecken streng abwechselnd 74
Abbildung D-9: Ausstecken nicht möglich, wenn verriegelt 74

Tabellenverzeichnis

Tabelle 4-1: Beispielanforderungen HLR 24
Tabelle 4-2: Beispielanforderungen Kommunikation / Interlockfunktion 25
Tabelle 4-3: Beispielanforderung Schützansteuerung 26
Tabelle 5-1: HLR10, HLR12, HLR13 inkonsistent 41
Tabelle 5-2: HLR09, HLR11, HLR12 inkonsistent und abstrakt 41
Tabelle 5-3: HLR11 korrigiert und abstrakt 42
Tabelle 5-4: HLR11 korrigiert und konkret 42

Abkürzungsverzeichnis

AC	Alternating current – Wechselspannung
BMS	Batterie-Management-System
DC	Direct Current – Gleichspannung
FC	Function-Controller – Funktionsprozessor
$flex^{Pi}$	Flexible Power interface - flexible Hochvoltschnittstelle
GuR	Gefahren- und Risikoanalyse
HLR	High Level Requirements - Anforderungen auf oberster Ebene
HV	Hochvolt
ID	identifier – Kennung
IT	Isolé Terre – vom Erdpotential isoliert
MSD	Modal Sequence Diagram – Modales Sequenz Diagramm
MSS	Modal State Structure – Modaler Zustandsautomat
OCL	Object Constraint Language – Objekteinschränkungssprache
OMG	Object Management Group
PDU	Power distribution unit – Leistungsverteiler
SC	Safety-Controller – Sicherheitprozessor
UML	Unified Modeling Language – Vereinheitlichte Modellierungssprache

1 Einleitung und Motivation

Grundlage für die Entwicklung von Systemen, Teilsystemen, Komponenten und Funktionen in der Automobilindustrie bilden Anforderungen, die zu Beginn aufgestellt und während des Projektes gepflegt werden müssen. Diese Anforderungen beschreiben, wie sich das entwickelnde Produkt verhalten und wie dieses Verhalten realisiert werden soll. Sie bilden Vorgaben für die Umsetzung und werden beim Test als Grundlage verwendet. Der Entwicklungsprozess lässt sich durch das sog. *V-Modell* darstellen.

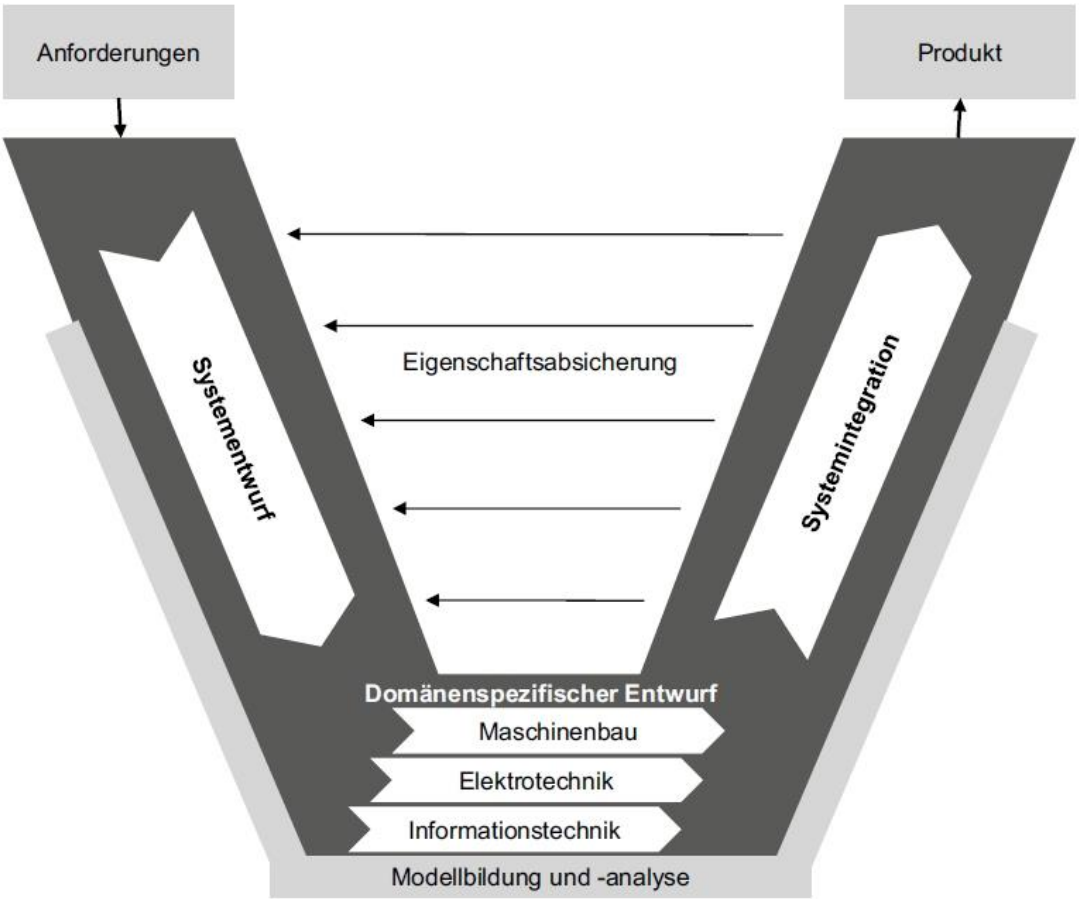

Abbildung 1-1: V-Modell nach VDI 2206 [1]

Dieses Modell wird auch in der Praxis angewendet. Dabei werden zu Beginn die Fahrzeugfunktionsanforderungen aufgestellt. Diese beschreiben, wie die Funktion dem Benutzer gegenüber erscheint. Dieser Schritt befindet sich links oben im V. In Abbildung 1-1 sind diese Fahrzeugfunktionsanforderungen als „Anforderungen“ bezeichnet. Der „Systementwurf“ gestaltet das System aus und ergänzt somit weitere detailliertere Anforderungen an die Teilsysteme und Komponenten. In der Praxis werden die Anforderungen nicht nur in Domänen sondern auch in Teilsysteme gegliedert. Diese Auffächerung findet im unteren Teil des Systementwurfs statt. Durch diese Auffächerung entstehen im unteren Teil des V-Modells mehrere parallele Entwicklungsstränge. Diese

haben jeweils die Struktur, die durch das V-Modell vorgegeben ist. Bereits bei der Aufstellung von Anforderungen für die Komponenten und Teilsysteme und der Umsetzung als Software können sich Lücken und Widersprüche in den Anforderungen zeigen.

Um die Qualität der Software zu erhöhen und die Entwicklung effizienter und kontrollierbarer durchzuführen, wird die Entwicklung durch Prozesse gelenkt. In der Automobilindustrie hat sich zur Gestaltung und Bewertung dieser Prozesse das Reifegradmodell für Softwareprojekte *Automotive SPICE* durchgesetzt [2]. Es wird zunehmend versucht, dieses Modell auch auf andere Bereiche in der Automobilentwicklung anzuwenden. Es gibt Richtlinien, welche Teilprozesse der gesamte Entwicklungsprozess beinhalten soll und welche Produkte (meist Dokumente) diese haben sollen. Sie werden durch *Basispraktiken* („base practices“) ergänzt, die für jeden vorgesehenen Teilprozess eine Vorlage darstellen, wie dieser gestaltet sein kann. In diesen Basispraktiken ist immer auch eine Verifikation der Prozessprodukte enthalten. Diese erfolgen meist durch *Reviews* oder *Simulationen*. Bei Reviews werden die Anforderungen von einem Prüfer manuell auf Mängel untersucht. Dabei spielt dieser *Use-Cases* (Anwendungsfälle) und *Misuse-Cases* (Missbrauchsfälle) durch und versucht anhand dieser, das Verhalten des Systems zu untersuchen. Bei Simulationen werden ebenfalls Use- und Misuse-Cases in einem Simulator durchgespielt. Dieser Simulator stellt das System vereinfacht dar. Aufgrund der Menge der Anforderungen, die in einem Automobilprojekt vorhanden sind, ist es nicht möglich alle möglichen Use- und Misuse-Cases zu testen. Da diese Methoden zur Verifikationen i.d.R. nicht alle Lücken und Wiedersprüche finden, werden diese oft erst bei den Tests, die die Systemintegration begleiten, entdeckt. Für jeden bei der Systemintegration entdeckten Fehler muss der Systementwurf und die Umsetzung im unteren Teil des V-Modells erneut durchlaufen werden. Diese erneuten Durchläufe nehmen den größten Teil des Entwicklungsaufwands in Anspruch und generieren somit den Hauptanteil der Entwicklungskosten.

Um die Erkennung von Lücken und Widersprüche bei der Verifikation zu optimieren, sollten Methoden verwendet werden, die eine automatisierte Analyse möglich machen. In der Automobilindustrie sind derartige Methoden bislang nicht vorhanden bzw. nicht etabliert. Daher teste ich in dieser Arbeit eine Methode, die Anforderungen als *Modale Sequenzdiagramme (MSD)* darstellt, und ein Tool, mit dem diese modelliert und analysiert werden können, an einem Automobilsoftwareprojekt. Mit MSDs kann das Verhalten von Objekten anschaulich und formal dargestellt werden. Die Herausforderung besteht an dieser Stelle die informell festgehaltenen Anforderungen zu formalisieren. Hierfür entwickle ich in dieser Arbeit, basierend auf üblichen Qualtitätsrichtlinien für Anforderungen, ein Schema, das den Entwickler, der die Anforderungen prüfen soll, bei diesem Schritt unterstützt. Auf Basis dieser formalen Darstellung wird das Verhalten des spezifizier-

ten Systems in allen möglichen Situationen mit dem Tool *ScenarioTools* untersucht [3]. Aus dem daraus resultierenden Ergebnis können Anforderungslücken und Widersprüche erkannt werden.

In MSDs werden die Interaktionen und die Kommunikation zwischen Objekten dargestellt. Diese Objekte können Systemkomponenten und Akteure aus der Umgebung des Systems sein. Die szenariobasierte formale Sprache MSD hat einige Vorzüge gegenüber anderen Darstellungsformen wie Zustandsautomaten oder herkömmlichen Sequenzdiagrammen. Im Gegensatz zu herkömmlichen Sequenzdiagrammen können MSDs darstellen, ob eine Aktion auftreten „kann“ oder „muss“. Diese Unterscheidung wird in Anforderungen oft angestellt und sollte somit auch in einer formalen Darstellung wieder zu finden sein. Sequenzdiagramme und damit auch MSDs stellen das Verhalten von Systemen als Szenarien dar. Diese äußere szenarioorientierte Sichtweise ist für Entwickler, wie Haral und Marelly in ihrem Buch „Come Let'sPlay“ [4] ausführen, die „natürlichere Sichtweise“ auf ein System. So würde ein Mensch über das Starten des Motors eher die Aussage machen: „Wenn ich den Zündschlüssel drehe, dann wird der Motor gestartet“, als zu beschreiben, „Der Motor befindet sich im Zustand Betriebsbereit und wenn ein Ereignis vom Zündschloss kommt, wird der Motor gestartet.“ Zudem legt diese äußere Sichtweise eine Top-Down-Entwicklung nahe, d.h. ausgehend von dem nach außen hin sichtbaren Verhalten, wird das Verhalten immer feinerer Komponenten spezifiziert. Dieses Vorgehen entspricht i.d.R. auch dem Vorgehen bei der Erstellung von Anforderungen. Erst bei einem sehr hohen Datailgrad wird die Sichtweise manchmal gewechselt und eine zustandsbasierte innere Sichtweise verwendet.

Zum Schluss bewerte ich den Nutzen dieser Methode, der verwendeten Sprache und beteiligten Tools und benenne Punkte, die Verbesserungspotential haben.

Als Beispielprojekt zur Erprobung der Methode werden die Anforderungen für die *flex*^{Pi}-Schnittstelle verwendet. Diese Schnittstelle soll es ermöglichen, Geräte mit hohem Leistungsbedarf oder –angebot an das HV-Netz eines elektrifizierten Fahrzeuges anzuschließen [5]. Zu diesen Geräten zählen Klimakompressoren für gekühlte Auflieger für Sattelschlepper oder flexibel einsetzbare *Range Extender*[1]. Für eine solche Schnittstelle ist aufgrund der erheblichen Sicherheitsanforderungen (siehe Kapitel 3) eine Steuerung erforderlich, die die Schnittstelle und das Nebenaggregat kontrolliert. Da das System vom Aufbau mit HV-Batterien und Automobil-Ladesystemen verwandt ist und ähnliche Anforderungen an das System gestellt werden, kann diese Software exemplarisch verwendet werden, um diese Anforderungsanalysemethode für die Automobilindustrie zu beleuchten.

[1] Reichweitenerweiterer: Zusätzliche Batterie, Brennstoffzelle oder Verbrennungsmotor mit angeschlossenem Generator, welche das HV-System mit zusätzlicher elektrischer Energie versorgen.

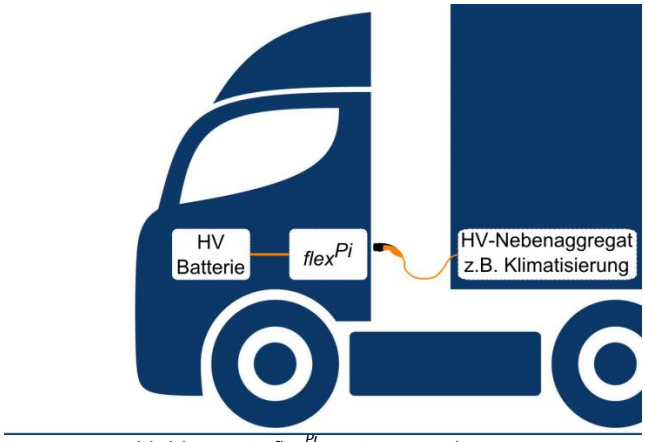

Abbildung 1-2: *flex*^{Pi} in einem E-Fahrzeug

Beim Test an den realen Anforderungen hat sich gezeigt, dass nahezu alle funktionalen Anforderungen durch MSDs darstellbar sind. Jedoch müssen zusätzliche Konstrukte in die formale Sprache eingeführt bzw. in ScenarioTools implementiert werden. Um alle verwendeten Sprachelemente der MSDs nutzen zu können, musste ich einige Bugs von ScenarioTools beheben. Es hat sich gezeigt, dass funktionale Anforderungen, die nicht mit dem Schema, das ich in dieser Arbeit eingeführt habe, modelliert werden können, i.d.R. nicht eindeutig und nicht atomar formuliert sind. Widersprüche werden durch die automatisierte Analyse von ScenarioTools schnell gefunden, jedoch gestaltet sich die Identifikation des Fehlers oft schwierig, weshalb ich ein neues Vorgehen entwickelt habe, diese Fehler zu finden und zu beheben. Um dieses Vorgehen zu unterstützen, habe ich für ScenarioTools Erweiterungen implementiert, die das Ergebnis so darstellen, dass dieses analysierbar ist.

Die wichtigsten Punkte dieser Arbeit sind:

- Erprobung der Methode an einem realen Projekt aus der Automobilindustrie
- Entwurf eines Schemas zur Formalisierung von Anforderungen als MSDs
- Vorgehen zur Analyse von Widersprüchen und anderen Mängeln in den Anforderungen
- Bewertung der Methode und Konzeption von notwendigen Erweiterungen

1.1 Struktur der Arbeit

In Kapitel 2 werden die Grundlagen für diese Arbeit erklärt. Das Projekt aus der Automobilindustrie, das ich in dieser Arbeit verwende, wird in Kapitel 3 ausführlich vorgestellt. Kapitel 4 befasst sich mit der Erstellung und Darstellung von Anforderungen, wie sie in der Automobilindustrie zu finden sind. Die Formalisierung von Anforderungen als MSDs stelle ich in Kapitel 5 dar. In Kapitel 6 befasse ich mich mit der Synthese des Systemverhaltens aus den, als MSD dargestellten, Anforderungen und mit der Analyse von Fehlern. Das Fazit ziehe ich in Kapitel 7. Dabei bewerte ich das verwendete Tool, gebe Verbesserungsvorschläge und stelle dar, wie sich die Methode mit den

Verbesserungen in den Entwicklungsprozess integrieren lässt. Im Anhang B stelle ich die Erweiterungen und Korrekturen an ScenarioTools, die ich im Rahmen dieser Arbeit vorgenommen habe, dar.

2 Grundlagen

2.1 Anforderungen

Eine Anforderung beschreibt die Erwartung des Kunden bzw. Benutzers an ein Produkt [6]. Nach [7] sind Anforderungen wie folgt definiert:

- Eine Bedingung oder Fähigkeit, die von einem Benutzer benötigt wird, um ein Problem zu lösen oder ein Ziel zu erreichen.
- Eine Bedingung oder Fähigkeit, die durch das System oder eine Systemkomponente erfüllt werden muss oder zur Verfügung gestellt werden muss, um einem Vertrag, einem Standard, einer Spezifikation oder einem anderen formal vorgegebenen Dokument gerecht zu werden.
- Eine dokumentierte Darstellung einer der oben genannten Bedingungen oder Fähigkeiten.

2.1.1 Funktionale Anforderungen

Nach C. Ebert [6] beschreiben funktionale Anforderungen die Funktion, die ein System oder eine Systemkomponente zur Verfügung stellt. Dabei beschreiben diese Anforderungen, wie die Eingänge des Systems mit den Ausgängen verknüpft sind bzw. wie das System auf bestimmte Anregungen reagiert. Diese Art von Anforderungen macht den größten Teil der Anforderungen in der Automobilindustrie aus. Im Rahmen dieser Arbeit werden hauptsächlich funktionale Anforderungen behandelt.

2.1.2 Qualitätsanforderungen

Qualitätsanforderungen beschreiben qualitative Anforderungen, die ein System oder eine Funktion erfüllen muss [6]. Sie ergänzen die funktionalen Anforderungen um Anforderungen wie z.B. Zuverlässigkeit, Wartbarkeit, Bedienbarkeit und funktionale Sicherheit. Sie sind meist nicht testbar, da es sich z.T. um subjektive Merkmale, wie eine intuitive Bedienbarkeit oder Merkmale, die statistisch ausgewertet werden müssen, wie die Langlebigkeit eines Produkts, handelt. Daher wird in der Praxis oft auch die Testbarkeit als Kriterium verwendet.

2.1.3 Randbedingungen

Randbedingungen enthalten Einschränkungen und Bedürfnisse der im Geschäftsprozess Beteiligten [6]. Diese beinhalten Gesetze, Kosten und Geschäftsprozesse beim Lieferanten und Kunden.

2.1.4 Template

Es ist üblich für die Formulierung von Anforderungen Vorlagen (*Template*) zu verwenden [6]. Diese helfen die Anforderungen eindeutig und verständlich zu formulieren und erleichtern die teil-

weise automatisierte Weiterverarbeitung. Sie geben einen strukturierten Satz vor, der Bedingungen und Reaktionen durch eine begrenzte Menge von Verknüpfungswörtern in Zusammenhang setzt. Durch diese Templates können semiformale Formulierungen erzielt werden.

2.2 Sprachformen

Es gibt verschiedene Sprachformen, dazu zählen in diesem Kontext neben den natürlichen Sprachen (z.B. Deutsch, Englisch) und textuellen Programmiersprachen (z.B. C, Python, Java) auch grafische Darstellungen (z.B. UML, Simulink).

2.2.1 Natürliche Sprache

Eine Sprache, deren Regeln eher auf dem Sprachgebrauch beruhen als auf vor Verwendung festgelegten Regeln [7], wird als natürliche Sprache bezeichnet. Da sich der Sprachgebrauch bei verschiedenen Personen unterscheidet, ist hier stets Interpretation notwendig, um den Inhalt zu erfassen. Beispiele sind Sprachen wie Deutsch und Englisch.

2.2.2 Formale Sprache

Eine Sprache, deren Regeln vor der Verwendung explizit festgelegt sind [7], ist eine formale Sprache. Beispiele sind die verschiedenen Programmiersprachen.

2.2.3 Halbformale Sprache / Semiformal Sprache

Eine halbformale bzw. semiformale Sprache besteht aus formalen und nicht formalen (aus natürlicher Sprache) Elementen. Hierzu zählen auch die UML und Anforderungssprachen, bei denen eine natürliche Sprache durch Templates eingeschränkt wird.

2.3 Szenariobasierte Anforderungsanalyse

Mit der Anforderungsanalyse soll ermittelt werden, ob gegebene Anforderungen das zu entwickelnde System vollständig beschreiben und ob es Anforderungen gibt, die sich widersprechen. Da in vielen technischen Systemen die Menge der Anforderungen nicht einfach überblickt werden kann, ist es erstrebenswert, diese automatisiert oder zumindest teilweise automatisiert zu untersuchen.

Ein möglicher Ansatz ist, die Anforderungen als Szenarien darzustellen und automatisiert in einen Zustandsautomaten zu überführen. Aus diesem können Betriebszustände ermittelt werden, die auf Grund von fehlenden, widersprüchlichen oder nicht eindeutigen und deshalb interpretierten Anforderungen keine weitere Ausführung zulassen oder nicht der erwarteten Funktion entsprechen.

2.4 Unified Modeling Language

Mit der *Unified Modeling Language (UML)* hat die *Objekt-Managements-Group (OMG)* eine visuelle Sprache entwickelt, die in möglichst allen Bereichen der Softwareentwicklung, von Desktopanwendungen bis zu Software in der Luftfahrt oder Medizintechnik angewendet werden kann. Um die verschiedensten Problemstellungen lösen zu können, sind in der UML eine Reihe von verschiedenen Diagrammtypen definiert. Zudem sind die Beziehungen zwischen verschiedenen Diagrammtypen und den darin enthaltenden Elementen definiert. Unter den Diagrammtypen sind Diagramme, die Strukturen zwischen Objekten darstellen können, aber auch Diagramme, die das Verhalten eines Systems oder einer Komponente abbilden können [8]. Da für die verschiedenen Einsatzbereiche der Sprache nicht immer alle Diagrammtypen notwendig sind, werden im Folgenden die im Rahmen dieser Arbeit verwendeten Typen vorgestellt.

2.4.1 Stereotypen

Stereotypen ermöglichen es, bestehende UML-Elemente weiter zu spezifizieren. D.h. es können ihnen weitere Eigenschaften gegeben werden. Oft wird durch die Stereotypen der Kontext der Klassen definiert.

2.4.2 Klassendiagramm

Eine Klasse ist eine Menge von Objekten, die durch dieselben Eigenschaften und Verhaltensweisen beschrieben werden können [9]. In Klassendiagrammen können Klassen mit ihren Eigenschaften und Verhaltensweisen definiert werden. Hier können auch Beziehungen zwischen den Klassen wie z.B. das Erben von Eigenschaften einer anderen Klasse festgelegt werden. Die Eigenschaften einer Klasse werden als Attribute und die Verhaltensweisen als Operationen[2] bezeichnet.

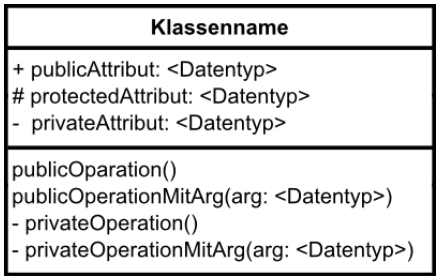

Abbildung 2-1: Klasse in UML-Klassendiagramm

In einem Klassendiagramm werden die Klassen als Rechtecke dargestellt (siehe Abbildung 2-1). Jede Klasse ist in mehrere Rechtecke unterteilt. Das oberste beinhaltet den Klassennamen, das zweite alle Attribute und das dritte alle Operationen. Die Datentypen der Attribute werden mit einem „:“ getrennt hinter das Attribut geschrieben. Mit den Zeichen „+“, „#“ und „-“ wird die

[2] In anderen objektorientierten Sprachen werden Operationen als Methoden bezeichnet.

Sichtbarkeit dargestellt, d.h. ob das Attribut *public*, *protected* oder *private* ist. Bei Operationen wird das optionale Argument in die Klammen geschrieben, dabei wird wie bei den Attributen der Datentyp durch einen „:“ getrennt hinter den Attributsnamen geschrieben. Operationen mit der Sichtbarkeit *private* werden durch ein „-“ gekennzeichnet.

2.4.3 Kompositionsstrukturdiagramm

Im Kompositionsstrukturdiagramm werden in einer sog. *Collaboration* die Objekte dargestellt, die miteinander kooperieren müssen, um die Aufgabe des Systems zu erfüllen [9]. Hier lassen sich auch die Signale, die zwischen den Objekten ausgetauscht werden, darstellen. Jedem Objekt muss eine Klasse zugeordnet werden, die zuvor z.B. in einem Klassendiagramm definiert wurde.

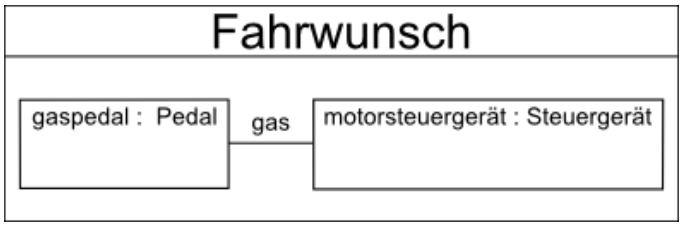

Abbildung 2-2: Beispiel eines Kompostionsstrukturdiagramms

Das in Abbildung 2-2 dargestellte Kompositionsstrukturdiagramm stell die Komponenten dar, die den Fahrwunsch des Fahrers verarbeiten. Umgangssprachlich bedeutet das, wie viel Gas gegeben werden soll. Die beteiligten Komponenten sind das Gaspedal (*gaspedal)* und das Motorsteuergerät (*motorsteuergerät*). Das Objekt *gaspedal* ist von der Klasse *Pedal* und *motorsteuergerät* von der Klasse *Steuergerät* abgeleitet. Dies ist durch den Klassennamen hinter dem „:“ dargestellt. Diese Beiden Objekte tauschen ein Signal, das in diesem Beispiel *gas* genannt ist, aus. Die kann in der späteren Realisierung entweder durch einen Bautenzug oder ein CAN-Signal erfolgen.

2.4.4 Sequenzdiagramme und Lebenslinien

In der Sprache UML sind Sequenzdiagramme als eine Darstellungsform für Interaktionen definiert [9].

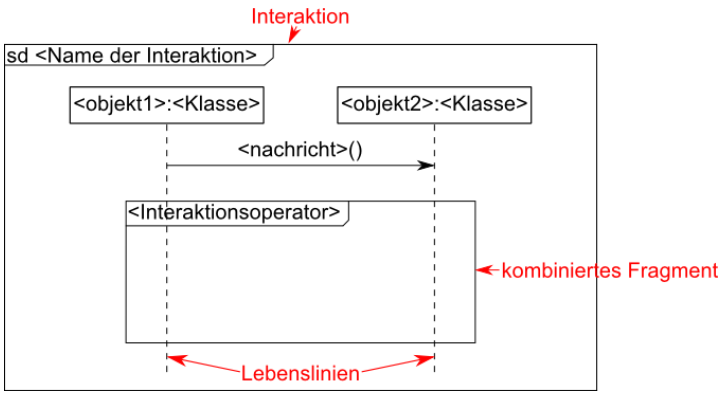

Abbildung 2-3: UML-Sequenzdiagramm

Interaktionen sind Abläufe von Nachrichten zwischen Objekten. Die Objekte werden durch *Lebenslinien* dargestellt, welche im Sequenzdiagramm durch gestrichelte vertikale Linien mit einem Rechteck als Kopf dargestellt werden (siehe Abbildung 2-3). Der Kopf beinhaltet den Namen des Objekts und die Klasse von der das Objekt abgeleitet ist. Nachrichten werden durch Pfeile zwischen den Lebenslinien dargestellt. Die Reihenfolge der Elemente auf den Lebenslinien ist für die Abfolge entscheiden. Der Abstand zwischen den Elementen hat jedoch keine Bedeutung. Damit werden nicht die genauen Zeitpunkte, sondern lediglich die Reihenfolge der Nachrichten dargestellt. Zudem gibt es das Sprachelement der sog. *kombinierten Fragmente*. Diese können wiederum verschiedene Kontrollstrukturen wie Entscheidungen, Schleifen oder Negation des Inhalts darstellen (siehe Kapitel 2.4.5).

2.4.5 Kombinierte Fragmente

Kombinierte Fragmente sind Elemente in Sequenzdiagrammen, mit denen die Abfolge von Nachrichten oder die Interpretation der Abfolge manipuliert werden kann. Durch den *Interaktionsoperator* wird festgelegt, um welchen Typ von kombiniertem Fragment es sich handelt. Die für diese Arbeit interessanten Typen sind die „Negation“ (neg), „Alternative“ (alt), „Parallele“ (par), „Wiederholung“ (loop) und „Zusicherung“ (assert). Innerhalb des kombinierten Fragments können sich je nach Interaktionsoperator ein oder mehrere *Interaktionsoperanden* befinden. Abhängig vom Interaktionsoperator können die Operanden mit Wächtern versehen sein, die, je nach Rückgabewert eines OCL-Ausdrucks (siehe Kapitel 2.4.6) und des Interaktionsoperators, die Ausführung des Operanden zulassen. Der Inhalt des Operanden kann wie eine Unterinteraktion betrachtet werden und somit Nachrichten und kombinierte Fragmente enthalten[3].

2.4.6 Object Constrained Language

Die *Object Constrained Language (OCL)* ist eine formale textuelle Sprache, die im Rahmen der UML definiert ist [10]. Sie ist keine Programmiersprache, d.h. sie kann nicht verwendet werde, um Programmabläufe zu beschreiben. Ein OCL-Ausdruck hat auch keine Seiteneffekte, d.h. es können keine Werte von Attributen geändert werden.

In dieser Arbeit werden OCL-Ausdrücke als Wächter für kombinierte Fragmente verwendet. Zudem können mit ihnen Argumente von Nachrichten von Objektattributen abhängig gemacht werden. Um den Wert eines Objektattributs darzustellen, wird in der OCL folgender Syntax verwendet: <objektname>.<attributsname>. Für Vergleiche mit Konstanten oder anderen Attributen können die Vergleichsoperatoren „=“, „<“, „>“, „>=“, „<=“ verwendet werden. Diese Vergleiche

[3] Die weiteren Sprachelemente von Sequenzdiagrammen werden an dieser Stelle ausgelassen, da sie für diese Arbeit irrelevant sind.

geben *true* oder *false* zurück. Diese boolschen Ausdrücke können mit den boolschen Operatoren „and“, „or“, „xor“ und „not“ verknüpft werden.

Beispiel:

Der Ausdruck „auto.anzahlRäder=4“ ergibt true, wenn das Attribut „anzahlRäder“ des Objekts „auto“ den Wert 4 hat. Dies kann auch mit anderen Vergleichen verknüpft werden. So ergibt der Ausdruck „auto.anzahlRäder=4 and auto.reifen='Winterreifen'“ dann true, wenn die Anzahl der Räder 4 ist und das Objekt „auto“ ein Attribut „reifen“ hat, das den Wert „'Winterreifen'“ hat.

Es können auch boolsche Variablen verwendet werden. So ergibt der Ausdruck „not auto.kannFahren“ dann true, wenn das Attribut „kannFahren“ des Objekts „auto“ false ist, also wenn das Auto nicht fahren kann.

2.5 Modale Sequenz Diagramme

Modale Sequenz Diagramme (MSD) sind, wie die allgemeinen UML-Sequenzdiagramme, eine Darstellungsform für Folgen von Aktionen [11]. MSDs lassen sich in *universelle* und *existentielle* MSDs einteilen. Ein existentielles MSD muss bei wenigsten einer Folge von Aktionen in einem System ausgeführt werden können. Ein universelles MSD darf bei keiner Folge von Aktionen verletzt werden. Da Anforderungen immer gelten sollen und ScenarioTools nur universelle MSDs betrachtet, beschränke ich mich in dieser Arbeit ebenfalls auf diese. Mit MSDs lässt sich zudem darstellen, ob eine Aktion erfolgen muss oder kann. Die Elemente der UML-Sequenzdiagramme werden durch ein Profil, das verschiedene Stereotype enthält, erweitert, um die Erweiterungen der MSDs darzustellen. Eine dieser Erweiterungen ist der Stereotyp „EnvironmentAssumption“, der eine Interaktion als Umweltannahme kennzeichnet. Da jedes MSD genau eine Interaktion enthält, wird damit das MSD als *Umweltannahme* gekennzeichnet. Diese werden als *assumption-MSDs* bezeichnet. Die anderen MSDs, die das Verhalten des Systems beschreiben, sind sog. *requirement-MSDs* Die Interpretation von Umweltannahmen wird in 2.6.2 erklärt. Im Folgenden werden die Elemente der MSDs vorgestellt.

2.5.1 Objektsystem

Die Objekte, die in MSDs miteinander interagieren können, müssen im sog. Objektsystem enthalten sind. Dieses kann mit einem Kompositionsstrukturdiagramm beschrieben werden. Hier kann angegeben werden, welche Objekte des Objektsystems zum System und welche zur Umgebung gehören. Dies wird mit einem Stereotyp umgesetzt, der eine Eigenschaft hat, die auf einen der Werte „Environment“ oder „System“ gesetzt werden kann. Environmentobjekte gehören nicht zum System, können dieses aber beeinflussen. Ihr Verhalten kann nicht definiert, jedoch durch

assumption-MSDs eingeschränkt werden. Systemobjekte sind Teil des Systems und verhalten sich den modellierten MSDs entsprechend. In MSDs werden die Objekte als Lebenslinien dargestellt. Das Objektsystem enthält mit den Objekten, die Instanzen der Klassen aus einem Klassendiagramm sind, auch deren Attribute und Operationen.

2.5.2 Nachrichten, Events und der Cut

Sowohl die Environmentobjekte als auch die Systemobjekte senden und empfangen Nachrichten an bzw. von andere Objekte oder sich selbst. Wenn ein Objekt eine Nachricht sendet oder empfängt, ist dies ein [3]. Da in dieser Arbeit das Sende- und das Empfangsevent immer gleichzeitig stattfinden, also nur synchrone Nachrichten vorkommen, werden diese beiden Events als ein Event bezeichnet. Jedes Event im Objektsystem ist mit einer Operation des empfangenden Objekts verknüpft. Zudem werden die Events im Objektsystem durch Diagramm-Nachrichten in den MSDs repräsentiert. Dabei müssen die Operation, der Sender und der Empfänger, sowie der optionale Parameterwert übereinstimmen. Zudem wird zwischen System-Events bzw. System-Nachrichten und Environment-Events bzw. Environment-Nachrichten unterschieden. System-Nachrichten werden von Objekten des Systems und Environment-Nachrichten von Objekten aus der Umwelt gesendet.

Der *Cut* beinhaltet alle Stellen eines aktiven MSDs, die durchlaufen wurden [3]. Damit charakterisiert er die aktuelle Position in einem MSD. In Abbildung 2-4 ist der Cut grün dargestellt.

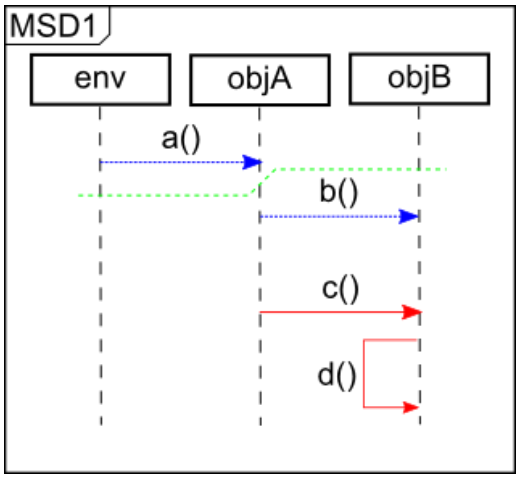

Abbildung 2-4: Der Cut in einem MSD

Der Cut bewegt sich bei jeder Nachricht, die aus diesem Diagramm ausgeführt wird, nach unten, bis er auf allen Lebenslinien das Ende des MSDs erreicht hat. Wenn sich der Cut vor der ersten Nachricht befindet, wird das MSD als *inaktiv* bezeichnet. Sobald der Cut die erste Nachricht übersprungen hat, ist das MSD *aktiv*, bis er am Ende des MSDs angekommen ist oder das Diagramm

abgebrochen wird. Wenn das MSD wieder inaktiv wird, wird der Cut an den Anfang des Diagramms zurückgesetzt. Er erstreckt sich immer über alle Lebenslinien des MSDs.

In MSDs werden die Nachrichten durch einen Stereotyp erweitert, der die weiteren Eigenschaften *Temperatur* und *Ausführungsart* definiert. Bei der Temperatur gibt es die Optionen *hot* (rot dargestellt) und *cold* (blau dargestellt), bei der Ausführungsart *monitored* (gestrichelt dargestellt) und *executed* (durchgezogen dargestellt) (siehe Abbildung 2-5).

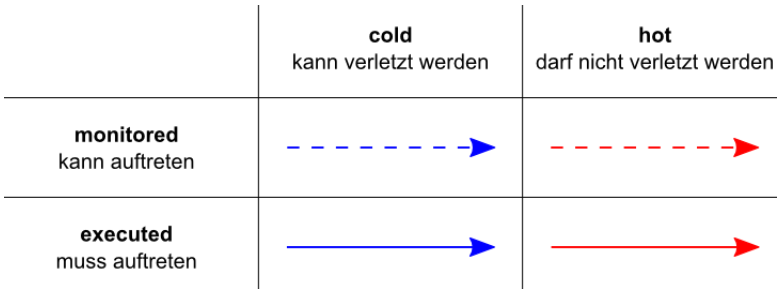

Abbildung 2-5: Arten von Nachrichten in MSDs [12]

Eine Nachricht ist dann *aktiviert*, wenn sich der Cut vor der Nachricht befindet [12, 3]. Eine aktivierte Nachricht mit der Temperatur *hot* bedeutet, dass keine andere Nachricht, die in diesem MSD enthalten ist, als nächstes auftreten darf. Wenn dies doch geschieht, tritt eine sog. *safety-Violation* auf. Wenn die Temperatur der aktivierten Nachricht *cold* ist, dürfen auch andere in diesem MSD enthaltene Nachrichten auftreten, jedoch wird dann das MSD inaktiv und der Cut zurückgesetzt. Diese Verletzung wird als *cold-Violation* bezeichnet. Nachrichten die *executed* sind, müssen irgendwann ausgeführt werden. Wenn der Cut eines MSDs in alle Ewigkeit vor einer executed Nachricht hängt, wird dies als *liveness-Violation* bezeichnet. Liveness-Violations dürfen nicht auftreten. Wenn der Cut sich vor einer Nachricht befindet, die hot ist, ist der Cut ebenfalls hot. Ansonsten ist der Cut cold. Ähnliches gilt für den Fall, dass der Cut vor einer executed Nachricht ist. In diesem Fall ist der Cut ebenfalls executed, andernfalls monitored.

Zudem gibt es verbotene Nachrichten [3]. Diese werden am Ende des Diagramms aufgeführt und dürfen während das Diagramm aktiviert ist, nicht auftreten. Wenn eine verbotene Nachricht mit der Temperatur cold ausgeführt wird, dann wird eine cold-Violation ausgelöst und damit das Diagramm abgebrochen. Bei der Temperatur hot wird eine safety-Violation ausgelöst. Diese verbotenen Nachrichten werden je nach verwendetem Tool entweder durch einen eigenen Stereotyp festgelegt oder sie werden in ein kombiniertes Fragment mit dem Operator „neg“ eingebettet.

Allen Arten von Nachrichten können Argumente übergeben werden [3]. Wenn eine Nachricht ein Attribut des Zielobjekts als Argument hat, wird mit dieser Nachricht der Wert des Attributs geändert. Für diese Nachrichten wird die Konvention verwendet, dass sie den Namen des Attributs mit

dem Präfix „set“ haben. Diese Nachrichten werden auch als Nachrichten mit Seiteneffekt bezeichnet. Zudem können als Argument auch OCL-Ausdrücke verwendet werden, um das Argument von Objektattributen abhängig zu machen.

2.5.3 Bedingungen

Mit Bedingungen kann die Ausführung eines MSDs abhängig von einem OCL-Ausdruck (siehe Kapitel 2.4.6) beeinflusst werden [3]. Dabei kann diese abgebrochen oder verzögert werden. Diese Bedingungen werden als Rechtecke[4] dargestellt. Wenn sich der Cut des MSDs auf allen Lebenslinien direkt vor der Bedingung befindet, wird der OCL-Ausdruck der Bedingung ausgewertet. Wenn dieser *true* ergibt, wird der Cut hinter die Bedingung verschoben. Andernfalls wird die Ausführung je nach Temperatur der Bedingung angehalten oder abgebrochen. Bei der Temperatur hot bleibt der Cut vor der Bedingung stehen, bis die Auswertung *true* ergibt. Wenn eine cold Bedingung *false* ergibt, wird eine cold-Violation ausgelöst und damit das MSD abgebrochen.
Zudem kann durch eine Bedingung sofort eine safety-Violation ausgelöst werden. Hierfür muss als Bedingung das Wort „violation“ gesetzt werden.

2.6 ScenarioTools

Hierbei handelt es sich um ein Tool zur Analyse von MSDs. Es basiert auf Eclipse, dem Eclipse-Modeling-Framework und dem UML-Editor Papyrus. Die Entwicklung findet an den Universitäten Leibniz Universität Hannover, Politecnico di Milano, Universität Paderborn und Göteborgs universitet statt [13].

2.6.1 Play-Out

Der *Play-Out-Algorithmus* wartet zunächst auf eine Umweltnachricht und aktiviert alle MSDs, deren erste Nachricht der Umweltnachricht entsprechen [3]. D.h. die Operation, das Sendeobjekt und das Empfangsobjekt sowie ggf. der Wert des Arguments müssen mit der Nachricht im MSD übereinstimmen. Anschließend wird eine der aktivierten Nachrichten, die executed sind, ausgeführt und damit der Cut in allen aktiven MSDs, die diese Nachricht als aktuell aktivierte Nachricht haben, fortgesetzt. Zudem werden alle MSDs aktiviert, deren erste Nachricht der ausgeführten Nachricht entspricht. Somit ergibt sich eine neue Menge von executed Nachrichten, von der wiederum eine Nachricht ausgewählt wird, die ausgeführt wird, bis keine executed Nachricht mehr aktiviert ist. Dies ist auch der Fall, wenn kein MSD mehr aktiviert ist. Dann wird auf eine neue Umweltnachricht gewartet.

[4] Die Darstellung lehnt sich an die Darstellung des in dieser Arbeit verwendeten Tools an. In anderen Ausführungen über MSDs werden diese als Sechsecke mit gestrichelter Linie dargestellt.

Im *Play-Out* von ScenarioTools wird die Entscheidung, welche Umweltnachricht bzw. welche executed Nachricht ausgeführt wird, dem Benutzer überlassen. So kann man mit dem spezifizierten System spielen und ausprobieren, wie es auf bestimmte Stimulationen reagiert. Damit können kleinere Systeme untersucht werden oder in größeren Systemen bestimmte Stimulationsfolgen gezielt ausprobiert werden.

2.6.2 Synthese

Bei der *Synthese* wird eine Strategie erstellt, mit der das System auf jede mögliche Folge von Umwelteinflüssen reagieren kann, ohne eine Anforderung zu verletzen. Diese Strategie kann als Zustandsdiagramm dargestellt werden. Diese Strategie funktioniert dann, wenn sie die folgenden Bedingungen erfüllt [3]:

1. Die Strategie verhindert, dass in irgendeinem Anforderungs-MSD eine safety-Violation auftritt.
2. Die Strategie sorgt dafür, dass alle aktivierten Events irgendwann ausgeführt werden.
3. Die Strategie sorgt dafür, dass das System unendlich oft auf Umweltereignisse wartet.

Außer wenn…

4. Eine safety-Violation in einem Umweltannahmen MSD ausgelöst wird.
5. Eine liveness-Violation in einem Umweltannahmen MSD auftritt.

Wenn einer der Punkte 4 und 5 zutrifft, dann bedeutet dies, dass die Folge von Umwelteinflüssen nicht auftreten kann und somit eine Verletzung gegen die Bedingungen 1 bis 3 für das System unkritisch ist. Zur Erstellung dieser Strategie werden automatisiert alle möglichen Nachrichtenfolgen des Play-Outs verfolgt. Der Benutzer muss und kann kein Nachrichten mehr auswählen.

3 Die *flex*^{Pi}-Schnittstelle

In diesem Abschnitt beleuchte ich das Beispielprojekt, das ich in dieser Arbeit verwende, näher. Die Konzeption und Spezifikation der Schnittstelle wurde in anderen studentischen Arbeiten bei IAV GmbH ausgeführt und in der IAV Kundenzeitschrift „automotion“ präsentiert [5].

3.1 Konzept

In modernen elektrifizierten Fahrzeugen ist ein *Hochvolt-Board-Netz (HV-Netz)* vorhanden, das eine hohe elektrische Leistung zur Verfügung stellen kann. Somit wäre es attraktiv diese Leistung für zusätzliche Anbaugeräte, für die der 12V-Zigarettenanzünder-Anschluss nicht ausreicht, verwenden zu können. Da das Anschließen und Abtrennen von Geräten an das HV-Netz eines Fahrzeuges bei falscher Vorgehensweise gefährlich ist, dürfen bislang nur ausgebildeten Personen diese Tätigkeit vornehmen und es ist auf Grund der Kapselung des HV-Netzes mit erheblichem Aufwand verbunden. Die Schnittstelle *flex*^{Pi} (flexible Power interface) soll es ermöglichen Geräte an das Hochvoltnetz einfach und flexibel anzuschließen und wieder abzutrennen. Dabei soll die Schnittstelle die notwendigen Vorkehrungen treffen, sodass die Tätigkeit ungefährlich ist und auch von Laien durchgeführt werden kann.

3.2 Anwendungsbereiche

Ein solches System findet hauptsächlich im Nutzfahrzeugbereich viele Anwendungen [5], aber auch Geräte für den PKW-Bereich sind denkbar.

Im Nutzfahrzeugbereich sind Klimakompressoren für Kühlanhänger eine sinnvolle Anwendung. Bisher werden für diese LKWs, wenn die Kühlleistung groß sein soll, zusätzliche Generatoren [5] an den Verbrennungsmotor angeflanscht oder eigene Aggregate für das Kühlsystem verbaut. Bei elektrifizierten Zugmaschinen wäre eine Anbindung an das Hochvoltnetz sinnvoll. Um die Flexibilität eines Anhängers zu bewahren, müsste der LKW-Fahrer den Kühlanhänger an das Hochvoltnetz anschließen können. An dieser Stelle ist eine Schnittstelle wie die *flex*^{Pi} notwendig.

Eine weitere Anwendung ist die elektrische Versorgung von Geräten auf einer Baustelle oder an einem Unfallort. Hier wird bislang für Scheinwerfer und elektrische Werkzeuge ein zusätzlicher Generator zur Versorgung betrieben. Mit der *flex*^{Pi}-Schnittstelle ließe sich der leistungsstarke Generator eines Hybrid-LKWs nutzen. An die Schnittstelle könnte bei Bedarf auch ein Wechselrichter angeschlossen werden, der eine dreiphasige Drehstromversorgung bereitstellt.

Für rein elektrisch betriebene PKWs ist die Möglichkeit, einen Range-Extender anzuschließen, interessant. Dieser Range-Extender stellt aus einer zusätzlichen Batterie, einer Brennstoffzelle oder einem Verbrennungsmotor mit Generator zusätzliche Energie zur Verfügung. Damit erhöht

sich die Reichweite des Fahrzeugs. Dieser Range-Extender kann bei Bedarf mitgenommen und vom Benutzer an das HV-Netz angeschlossen werden.

3.3 Besondere Anforderungen

Um den Benutzer der Schnittstelle vor den bereits erwähnten Gefahren zu schützen, muss die Schnittstelle zuverlässig spannungsfrei geschaltet werden können und das Herausziehen des Steckers unter Last durch eine Verriegelung verhindert werden. Um dies zu realisieren, muss ein System aus Mechanik, Hard- und Software umgesetzt werden.

Es muss über die Steuerung sichergestellt werden, dass die HV-Anschlüsse der Schnittstelle zum Nebenaggregat nur dann mit dem HV-Netz des Fahrzeugs verbunden sein dürfen, wenn der Stecker gesteckt und verriegelt ist und weder in der Schnittstelle noch im Anbaugerät ein Fehler vorliegt.

Damit wird auch verhindert, dass der Stecker unter Last, d.h. mit Stromfluss, herausgezogen wird. Dies würde zu einem stehenden Lichtbogen führen. Dieser reißt nicht, wie bei einem Wechselstromsystem, beim Nulldurchgang selbst ab und kann zu Verblitzung der Augen (vgl. Schweißen ohne Schirm) und schweren Verbrennungen führen.

Außerdem muss die Verbindung zum Anbaugerät ständig überprüft werden, um im Falle eines abgerissenen Kabels die Schnittstelle schnellstmöglich spannungsfrei zu schalten. Ein Berühren von zwei unter Spannung stehenden Kontakten kann schwere Verbrennungen, Nervenschädigungen und Gasfreisetzung im Blut durch Hämolyse zur Folge haben. Um derartige Unfälle zu vermeiden wird vor dem Einschalten und während des Betriebs der Isolationswiderstand des Systems gemessen. Wenn dieser zu stark abfällt, kann man daraus schließen, dass ein ungewollter Kontakt z.B. zum Gehäuse besteht.

Um die Funktionsfähigkeit der Sicherheitseinrichtungen zu überwachen, werden der Strom, den das Anbaugerät aufnimmt und die Spannung an den Anschlüssen gemessen. Zudem wird ein Mechanismus vorgesehen, um sog. *Schützkleber* zu erkennen. Hierbei handelt es sich um einen Defekt der Relais, bei dem durch Lichtbogenbildung im Inneren, beim Schließen die Kontakte miteinander verschweißt werden. Wenn ein Schützkleber vorliegt, kann der entsprechende Pol des HV-Netzes nicht mehr von der Schnittstelle getrennt werden. Wenn dies der Fall ist, muss sichergestellt werden, dass der andere Pol nicht mehr geschlossen wird, um dort den gleichen Defekt zu verhindern, denn wenn durch den Schützkleber die kontrollierte Vorladung verhindert wird, kommt es zwangsläufig zu einem weiteren Schützkleber. So lange nur ein Schütz klebt ist der Zu-

stand nicht gefährlich, da es sich bei einem Fahrzeug-HV-Netz um ein sog. IT-Netz[5] handelt, das vom Erdpotential galvanisch getrennt ist. Aufgrund der Tatsache, dass der Stromkreis bei der Berührung eines Potentials nicht geschlossen ist, kann es zu keinem Stromfluss und damit auch zu keinem schädlichen Körperstrom kommen.

3.4 Komponenten der Schnittstelle

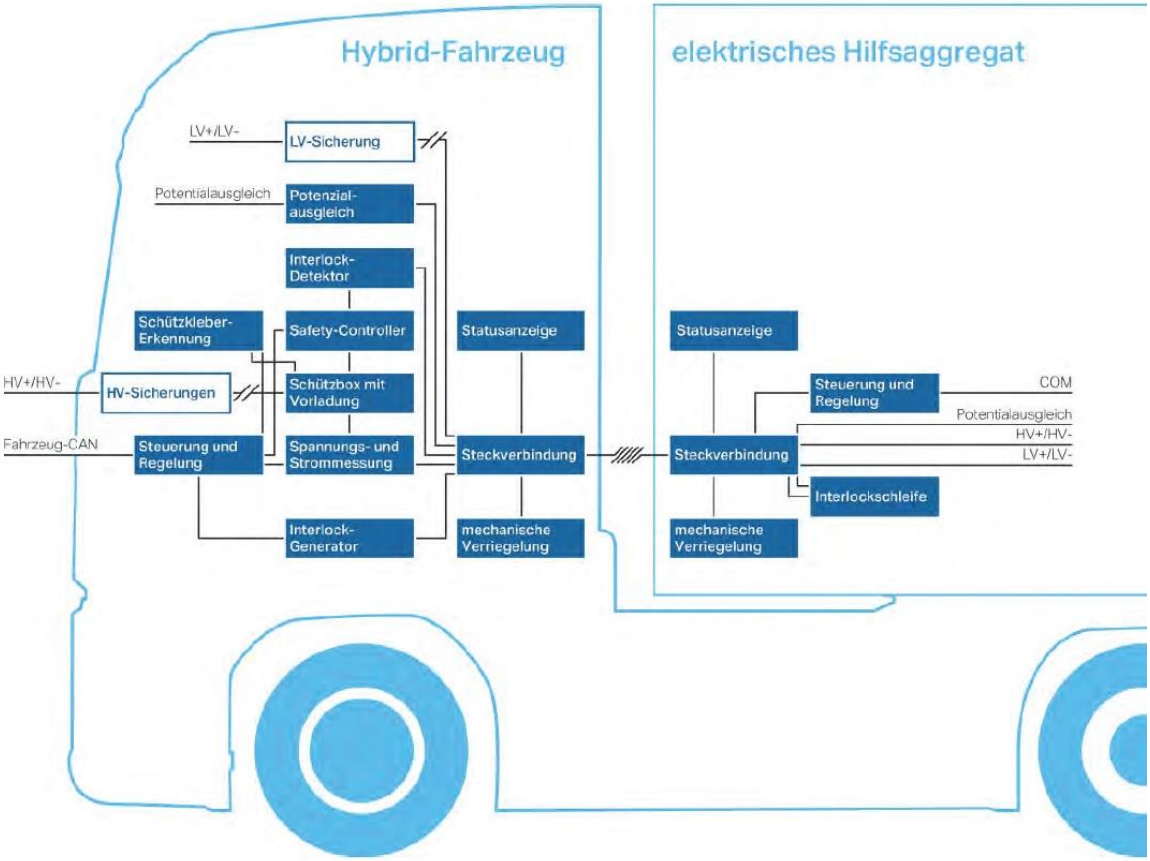

Abbildung 3-1: Übersicht über alle Komponenten der *flex*^{Pi} [5]

In Abbildung 3-1 sind die Komponenten der *flex*^{Pi}-Schnittstelle dargestellt. Die *LV-* und *HV-Sicherung* sind im LV- bzw. HV-System bereits vorhanden und damit eigentlich nicht Bestandteil der Schnittstelle. Die *Steckverbindung* besteht aus einem Stecker und einer Buchse. Die *mechanische Verriegelung* ist ein Bolzen in der Buchse der Schnittstelle, die den Stecker in der Buchse fixiert. Der Bolzen kann über ein Steuersignal ein- und ausgefahren werden. Die *Statusanzeige* wird durch zwei RGB-LEDs am Stecker realisiert. Hier sind jedoch auch andere Konzepte denkbar. Der *Potentialausgleich* ist eine elektrische Leitung zur Fahrzeugmasse, die elektrostatisch aufgeladene Kapazitäten, z.B. am Gehäuse, entlädt. Die *Schützbox mit Vorladung* enthält die Hauptschütze, die die elektrische Verbindung zwischen dem HV-System und der Steckverbindung herstellen

[5] Issolé Terre

bzw. trennen können. Diese Schütze sind in einer separaten Box untergebracht, da an den Schraubkontakten der Schütze sonst offen HV-Kontakte zugänglich wären. Die Vorladung besteht aus einem elektrischen Widerstand, der über ein eigenes Schütz parallel zu einem der Hauptschütze geschaltet werden kann, um eine Kapazität im Nebenaggregat kontrolliert aufladen zu können. Diese Vorladung befindet sich ebenfalls in der Schützbox. Die *Steuerung und Regelung* ist die Software zur Steuerung der Schnittstelle und befindet sich auf einem *Function-Controller (FC)* eines Steuergeräts[6]. Der *Safety-Controller (SC)* ist ein zusätzlicher Prozessor im Steuergerät, der die korrekte Funktion des FCs und der darauf laufenden Software überwacht. Der *Interlockgenerator* und der *Interlockdetektor* sind Softwarekomponenten, im FC. Im SC ist zusätzlich ein Interlockdetektor vorhanden. Der Interlockgenerator im FC ist eine Softwarekomponente, die einem Ausgang mit einem bestimmten Signal ansteuert, das die Interlockdetektoren im FC und SC an einem Eingang erwarten. Diese sind Ebenfalls Softwarekomponenten. Dieses Signal wird vom Interlockgenerator über die *Interlockschleife*[7], die im einfachsten Fall eine Leitung ist, zu den Interlockdetektoren geleitet. Die *Spannungs- und Strommessung* misst die Spannung, die an der Buchse anliegt, und den Strom, der durch die Schnittstelle fließt. Beide Messungen müssen eine galvanische Trennung von HV- und LV-System sicherstellen. Die *Schützkleber-Erkennung* ist eine Softwarekomponente, die auf Basis der Ansteuersignale der Schütze und der Spannungs- und Stromwerte analysiert, ob ein Schützkleber vorliegt.

[6]Steuergeräte auf denen sicherheitskritische Funktionen laufen haben oft zwei Prozessoren. Diese werden Function-Controller und Safety-Controller genannt.

[7] Die Interlockschleife war in der Ursprünglichen Konzeption vorgesehen. In der Zwischenzeit wird die Funktion des Interlocks durch Kommunikation mit den eingesteckten Komponenten realisiert.

4 Anforderungserstellung

In diesem Kapitel stelle ich das in der Automobilindustrie übliche Vorgehen zur Erhebung und Pflege von Anforderungen dar. Die Methode zur Darstellung und Analyse von Anforderungen, die ich in den folgenden Kapiteln darstelle, muss sich in das übliche Vorgehen integrieren lassen. Zudem zeige ich an einem Beispiel aus dem Beispielprojekt, wie ich die Anforderungen erstellt habe.

Der Ausgangspunkt für Entwicklungen sind immer die Anforderungen an das Produkt. Diese sind zu Beginn des Projekts mehr oder weniger detailliert ausformuliert. In der Automobilindustrie werden diese Anforderungen in Tabellen oder speziellen Anforderungsdatenbanken festgehalten. Diese Formulierungen sollen möglichst eindeutig und atomar, d.h. nicht in mehrere Anforderungen unterteilbar, sein. Für die Formulierungen gibt es keine Vorgaben und so ist von ausformulierten Sätzen, über Stichworte bis zu codeähnlichen Formulierungen und grafischen Darstellungen in der praktischen Umsetzung alles zu finden. Bei schlechten Anforderungen können diese Formulierungen mehrdeutig sein und müssen vom Entwickler interpretiert werden. Wenn Anforderungen als MSDs darstellbar sind, dann können diese damit eindeutig formal festgelegt werden, jedoch können nicht alle Anforderungen als MSDs dargestellt werden. So muss z.B. bei Hardwareanforderungen und nicht-funktionale Anforderungen auf die herkömmlichen Formulierungen zurückgegriffen werden.

Um die Integrierbarkeit von MSDs in den Automobilentwicklungsprozess zu untersuchen, muss ich für die Analyse einen Ausgangspunkt verwenden, der im üblichen Entwicklungsprozess bereits vorhanden ist. Daher erstelle ich zunächst auf klassischem Wege eine Anforderungsliste. Da das Beispiel relativ überschaubar ist, verzichte ich auf die Verwendung einer Anforderungsdatenbank und verwende stattdessen einige Tabellen mit IDs zur Referenzierung der Anforderungen. Diese Anforderungsliste soll möglichst vollständig sein, um später mit der Methode, die ich in dieser Arbeit untersuche, vergleichen zu können.

Bevor die Anforderungen aufgestellt werden können, muss zunächst eine Beschreibung der Funktion formuliert werden. Dabei können bereits Use-Cases formuliert werden. Bevor die Aufstellung der Anforderungen abgeschlossen ist, muss eine *Gefahren- und Risikoanalyse (GuR)* nach ISO 26262 [14] durchgeführt werden. Diese ermittelt die Komponenten des Systems, bei deren Versagen besonders hohe Gefahren für Menschen in der Umgebung des Systems ausgehen. Diese Analyse ist im Serienentwicklungsprozess vorgeschrieben und ergibt je nach Ergebnis zusätzliche Anforderungen an bestimmte Teilsysteme. Die in der GuR verwendeten IDs referenziere ich in der

Anforderungsliste, um Bezüge zwischen den Anforderungen und den Analyseergebnissen herstellen zu können.

4.1 Gefahren- und Risikoanalyse

In der GuR werden zunächst Annahmen über das Verhalten des Systems aufgestellt. Diese Annahmen sollten möglichst vollständig sein, damit selbst ohne Vorkenntnis über das System die Analyse durchgeführt und nachvollzogen werden kann. Anschließend werden Szenarien aufgestellt, in denen angenommen wird, dass ein Teil oder eine Funktion des Systems ausfällt. Anschließend wird abgeschätzt, welche Auswirkungen dies direkt oder indirekt auf Personen in der Umgebung des Systems hat. Das Szenario und dessen Auswirkungen werden dann unter den Aspekten Auftretenswahrscheinlichkeit, Schwere und Beherrschbarkeit in je eine von vier Stufen eingeteilt, die dann an Hand einer Tabelle aus ISO 26262 [14] in eine der fünf Kategorien QM, ASIL A bis ASIL D eingestuft werden. Wobei für QM keine und für ASIL A bis ASIL D aufsteigend strengere und weiter führende Maßnahmen hinsichtlich Entwicklungsprozess und funktionaler Sicherheit ergriffen werden müssen. Diese haben das Ziel die Ausfallsicherheit zu erhöhen. Ein Beispiel für solche Maßnahmen ist das Schaffen von Redundanzen z.B. durch einen zusätzlichen Mikrokontroller, der die korrekte Funktion überwacht.

4.2 Anforderungsliste

Die Anforderungsliste formuliere ich in mehreren Ebenen aus[8]. Auf der obersten Ebene befinden sich die sog. *High Level Requirements* (HLR). Diese Anforderungen beschrieben das Zusammenwirken der Komponenten. Zudem definiere ich dort Fehlerfälle, auf die das System reagieren soll und lege die Reaktionen auf diese fest. Auf der darunter liegenden Ebene spezifiziere ich die verschiedenen Teilsysteme detaillierter, teile in die Bereiche Mechanik, Hardware und Software auf und beschreibe ihre Schnittstellen. Die Anforderungen jedes Teilsystems pflege ich in einer eigenen Tabelle. Dort wird auch festgelegt, wie die einzelnen Komponenten die in den HLR geforderten Anforderungen umzusetzen haben. Dabei referenziere ich die IDs der übergeordneten Anforderungen. Damit der Softwareentwickler keine Kenntnis über das technische System selbst mitbringen muss, müssen Annahmen über das Verhalten der Komponenten festgehalten werden. Wenn das Dokument aus Sicht der obersten Systemebene geschrieben wird, können diese Annahmen auch als Anforderungen an Teilsysteme formuliert werden.

[8] Hierbei handelt es sich um eine mögliche Form, nicht um eine eindeutig festgelegte.

4.3 Formulierung einzelner Anforderungen

Um möglichst atomare und eindeutige Anforderungen zu schreiben, ist wenn möglich eine Formulierung in der Form: Wenn…, dann… zu wählen. Hiermit lässt sich jedoch nur das Verhalten eines Systems und dessen Zustände beschreiben (z.B. Wenn die Schnittstelle spannungsfrei ist, dann liegt der sichere Zustand vor.). Zudem werden Begriffe, die in anderen Anforderungen verwendet werden, definiert (z.B. Wenn die Spannung an den Ausgängen kleiner als 60 V ist, dann ist die Schnittstelle spannungsfrei.).

4.4 Beispiel

Im Folgenden stelle ich das Erstellen der Anforderungen exemplarisch dar. Bei der GuR des Beispielprojekts hat sich gezeigt, dass der kritischste Fall der ist, in dem das Kabel zwischen Schnittstelle und Nebenaggregat reißt bzw. aus dem Nebenaggregat heraus gerissen wird. Dies ist z.B. bei einem sich außerhalb des Fahrzeugs befindenden Wechselrichter zur Versorgung einer Baustelle leicht möglich, wenn das Fahrzeug weg gefahren wird, ohne die Verbindung zum Wechselrichter zu trennen. Für dieses Szenario ermittele ich den ASIL-Level ASIL B. Als Sicherheitsziel formuliere ich, dass die Verbindung zum Nebenaggregat permanent überwacht werden muss und dass im Falle einer abgebrochenen Verbindung schnellst möglich die Schütze geöffnet werden müssen, um die Schnittstelle spannungsfrei zu schalten. Um dieses Sicherheitsziel zu erreichen, müssen die Schützansteuerung und die Kommunikation mit dem Nebenaggregat besondere Anforderungen erfüllen. Die Präfixe der entsprechenden IDs und der davon abhängigen IDs werden mit der Zeichenfolge „ASILB“ erweitert. Somit ist auf jeder Ebene zu erkennen, dass für diese Anforderung der ASIL-Level zu beachten ist. In diesem Projekt habe ich mich dafür entschieden, einen zusätzlichen Mikrokontroller zu verwenden, der diese Funktion überwacht, um die Redundanz, die durch den ASIL-Level gefordert wird, zu erzielen.

Auf der HLR-Ebene der Anforderungen wird das Sicherheitsziel aus der GuR durch die Anforderungen „Interlockfunktion“ (HLR_ASILB_39) und durch die „entladen bei Interlockfehler“ (HLR_ASILB_90) erfüllt.

Der *Interlock* ist eine Leitung, die durch alle zu überwachenden Stecker gelegt wird und somit deren Zustand überwachen kann. Mit Endschaltern, die in den Interlockpfad geschaltet sind, kann überprüft werden, ob alle Gehäusedeckel geschlossen sind. Elektrische Geräte können über Relais, wenn sie über ein solches Verfügen, bei Bedarf ebenfalls den Interlockpfad unterbrechen. Um die zusätzlichen Leitungen einzusparen, wird in jüngster Zeit die Funktion der Leitung durch die Kommunikation mit den Geräten realisiert. Auch hier gilt, dass die Verbindung nicht besteht, wenn der Stecker nicht gesteckt ist. In der Software müssen für die Interlockfunktion alle Verbin-

dungen „Und“-Verknüpft werden. In diesem Beispiel realisiere ich die Interlockfunktion über die CAN-Kommunikation[9] mit dem Nebenaggregat. Zum Entladen des Systems werden alle Schütze getrennt, sodass keine elektrische Verbindung zwischen HV-System und der Schnittstelle mehr besteht. Das Entladen muss, wenn das Nebenaggregat kapazitiv ist, von diesem automatisch ausgeführt werden. Induktive Lasten müssen geeignete Maßnahmen, z.B. eine Freilaufdiode, für einen Freilauf haben.

Name der Anforderung	ID	abhängige ID	Anforderung
HV-aktiv	HLR_01		Wenn mindestens ein Schütz geschlossen ist oder die Spannung an der Buchse über V_SPANNUNGSFREI ist, dann ist die flexPi-Schnittstelle HV-aktiv.
HV-inaktiv	HLR_02		Wenn alle Schütze geöffnet sind und die Spannung an der Buchse unter V_SPANNUNGSFREI ist, dann ist die flexPi-Schnittstelle HV-inaktiv.
vorladen	HLR_03		Wenn die Spannung am Ausgang über den Vorladewiderstand aufgebaut wird, wird das System vorgeladen.
entladen	HLR_04		Wenn alle Schütze geöffnet sind und sich die Spannung an der Buchse abbaut, dann wird das System entladen.
betriebsbereit	HLR_05		Wenn die beiden Hauptschütze geschlossen sind, dann ist die Schnittstelle betriebsbereit.
Interlockfunktion	HLR_ASILB_39	GUR_ASILA_9, GUR_ASILA_17, GUR_ASILB_18, GUR_ASILA_20	Wenn das Nebenaggregat eingesteckt ist, dann muss die Verbindung ständig überprüft werden.
Interlockfehler	HLR_ASILB_89	GUR_ASILA_9, GUR_ASILA_17, GUR_ASILB_18, GUR_ASILA_20	Wenn das Nebenaggregat nicht verbunden ist und die flexPi-Schnittstelle nicht HV-inaktiv ist, dann liegt ein Interlockfehler vor.
Entladen bei Interlockfehler	HLR_ASILB_90	GUR_ASILA_9, GUR_ASILA_17, GUR_ASILB_18, GUR_ASILA_20	Wenn ein Interlockfehler vorliegt, müssen alle Schütze geöffnet werden.

Tabelle 4-1: Beispielanforderungen HLR

Zunächst definiere ich mit den Anforderungen HLR_01 bis HLR_05 die Zustände des Systems, damit ich im Folgenden diese Begriffe verwenden kann. Der sichere Zustand ist „HV-inaktiv“, hier

[9] Hier wird nur der Fall eines aktiven Nebenaggregats betrachtet. D.h. das Nebenaggregat beinhaltet ein Steuergerät mit dem eine CAN-Kommunikation aufgebaut werden kann. Für die Erprobung der formalen Methode ist diese Fallunterscheidung nicht erheblich.

ist sichergestellt, dass an der Schnittstelle keine Spannung anliegt. Im Zustand „HV-aktiv“ liegt Spannung an der Schnittstelle an oder es ist mindestens ein Schütz geschlossen. Das System ist immer dann in diesem Zustand, wenn es nicht im sicheren Zustand ist. Das System ist dann im Zustand „Betriebsbereit“, wenn beide Hauptschütze geschlossen sind. Nur in diesem Zustand darf das Nabenaggregat betrieben werden. Die beiden Zustände „Vorladen“ und „Entladen“ sind die Übergänge zwischen „HV-inaktiv“ und „Betriebsbereit“. In beiden Fällen sollte das Nebenaggregat noch nicht betrieben werden. Es werden lediglich puffernde Energiespeicher, wie z.B. Glättungskondensatoren, auf- bzw. entladen.

Die Anforderung „Interlockfunktion“ fordert, dass das die Verbindung zum Nebenaggregat überprüft wird, wenn es eingesteckt ist. Damit wird schon vor dem Einschalten der HV-Versorgung geprüft, ob das Nebenaggregat richtig angeschlossen und fehlerfrei ist.

Die Anforderung „Entladen bei Interlockfehler“ definiert die Reaktion auf das Ereignis „Interlockfehler“, das durch die Anforderung „Interlockfehler“ definiert wird. Dieser Fehler liegt vor, wenn das Nebenaggregat nicht verbunden ist und die Schütze geschlossen sind bzw. das System vorgeladen oder entladen wird.

Name der Anforderung	ID	abhängige ID	Anforderung
Interlock, CAN	COM_ASILB_3	LOCK_10, HLR_42, HLR_ASILB_39	Wenn die CAN-Kommunikation aktiv ist, dann muss die Kommunikation durch eine regelmäßig wiederkehrende Nachricht überwacht werden.
Interlockfunktion, aktiv	COM_ASILB_22	COM_ASILB_3	Wenn die Statusnachricht länger als T_INTERLOCK_CAN nicht mehr empfangen wurde, dann ist der Stecker nicht gesteckt.

Tabelle 4-2: Beispielanforderungen Kommunikation / Interlockfunktion

Die in Tabelle 4-2 aufgeführten Anforderungen spezifizieren die Interlockfunktion näher. „Interlock, CAN“ fordert, dass, wenn eine CAN-Kommunikation besteht, diese durch regelmäßig wiederkehrende Nachrichten überwacht werden muss. D.h. das Nebenaggregat muss regelmäßig eine Nachricht, z.B. seinen eigenen Status, senden. In „Interlockfunktion, aktiv“ wird die maximale Zeit zwischen zwei solchen Nachrichten durch den Parameter T_INTERLOCK_CAN festgelegt. Die Parameter werden in einer eigenen Tabelle definiert und sollen auch bei der Implementierung wenn möglich verwendet werden. Wenn diese Zeit überschritten wird, wird angenommen, dass der Stecker nicht gesteckt ist. Wenn die Schnittstelle gleichzeitig nicht HV-inaktiv ist, wird die Bedingung der Anforderung „Interlockfehler“ erfüllt.

Name der Anforderung	ID	abhängige ID	Anforderung
Fehler, schütz öffnen	Cc_ASILB_6	HLR_ASILB_33	Wenn ein Fehler erkannt wird, müssen die Schütze geöffnet werden.

Tabelle 4-3: Beispielanforderung Schützansteuerung

Die Anforderung „Fehlerfall“ legt die Reaktion auf Fehlerfälle fest. Hierunter zählt auch der Interlockfehler. Im Fehlerfall werden alle Schütze geöffnet. Damit wird die Anforderung „entladen bei Interlockfehler“ erfüllt.

5 Modellierung

In diesem Kapitel erläutere ich, das von mir entwickelte Vorgehen zur Darstellung von Anforderungen als MSDs. Zudem wäge ich einige alternative Ansätze ab.

Für das Vorgehen gibt es zunächst zwei Grundsätzliche Ansätze. Entweder können für möglichst viele zusammenhängende Anforderungen ein Diagramm oder für jede Anforderung ein Diagramm verwendet werden. Der erste Ansatz hat den Vorteil, dass das Modell übersichtlicher wird und das Verhalten leichter nachzuvollziehen ist, jedoch lässt sich dann kein Bezug zu den modellierten Anforderungen herstellen. Für Jede Anforderung ein Diagramm zu verwenden, hat den Vorteil, dass der Zusammenhang zwischen Modell und den Anforderungen in der Anforderungsliste sehr einfach über den Namen des Diagramms hergestellt werden kann. Das Modell wird hierdurch jedoch unübersichtlicher, da Abhängigkeiten zwischen den Anforderungen für den Entwickler nicht offensichtlich sind. Da bestehende Anforderungen mit dieser Methode verifiziert werden sollen und die Rückverfolgbarkeit auf die Anforderungen in der Anforderungsliste unerlässlich ist, habe ich mich für den zweiten Ansatz entschieden. Diese Rückverfolgbarkeit wird in automotive Spice als *Traceability* bezeichnet und mit hoher Wichtigkeit bemessen. Hierbei verwende ich die IDs der Anforderungen in den Bezeichnungen der Diagramme als Präfix.

Das in dieser Arbeit dargestellte Vorgehen zur Modellierung bezieht sich auf das Tool ScenarioTools, ist jedoch weitgehend auch allgemein für MSDs anwendbar. Zudem schränke ich die Sprache der MSDs ein, damit diese dem Aufbau einer einzelnen Anforderung entsprechen.

5.1 Arten von Anforderungen

Die Anforderungen können, unabhängig von den Abhängigkeiten und der Ebene in der sie auftreten, in verschiedenen Gruppen eingeteilt werden. Es gibt Anforderungen, die Begriffe definieren (z.B. HLR_01 „HV-aktiv“). Diese lassen sich offensichtlich nicht als Sequenzdiagramm darstellen. Dies ist nicht notwendig, da sie nur zur einheitlichen Definition anderer Anforderungen dienen und nicht das System beschreiben. Außerdem gibt es Anforderungen, die Hardwareeigenschaften definieren (z.B. Cc_ASILB_16 Beidseitig Ansteuerung: Die Ansteuerung der Schütze muss über die High-Side und die Low-Side erfolgen.). Diese lassen sich ebenfalls nicht mit Sequenzdiagrammen darstellen, da es in diesen die entsprechenden Elemente nicht gibt. Diese Anforderungen sind für die Hardwareentwicklung bzw. für die Auswahl der Hardwarekomponenten unerlässlich und müssen somit ebenfalls neben den Anforderungen, die das Verhalten beschreiben, festgelegt sein. Schließlich gibt es noch Anforderungen, die das Verhalten des Systems beschreiben. Da, wie schon zu Beginn angenommen, nicht alle Anforderungen mit MSDs modelliert werden können, muss für die Darstellung aller Anforderungen eine Mischform aus MSDs und textuellen Formulie-

rungen gewählt werden. Im Folgenden betrachte ich nur noch Anforderungen, die das Verhalten des Systems beschreiben. Diese werden als MSDs modelliert, um daraus eine Aussage über deren Vollständigkeit und Widerspruchsfreiheit zu erhalten.

Die Elemente aus denen die Anforderungen bestehen, lassen sich wiederum in Aktionen und Änderungen und Auswertungen der Objekteigenschaften einteilen. Aktionen lassen sich sehr einfach durch Nachrichten modellieren. Objekteigenschaften werden als Attribut der Klasse dargestellt. Die Änderung eines Zustands wird durch eine Nachrichte mit Seiteneffekten modelliert. Mit Bedingungen kann der Zustand von Eigenschaften ausgewertet werden.

5.2 Modellierung des Systems

Zunächst lege ich eine Paketstruktur an [13]. Im obersten Paket werden die Komponenten des Systems definiert. Dieses Paket wird von mehreren Paketen mit dem „merge“-Befehl eingebunden, die das Verhalten des Systems spezifizieren. Diese können auch in mehrere Pakete aufgeteilt werden. Durch diese Aufteilungen lassen sich die Ebenen der Anforderungsliste darstellen und es können übergeordnete Ebenen unabhängig von den darunter liegenden Ebenen getestet werden. Diese Pakete werden wiederum mit dem „merge“-Befehl in ein Gesamtpaket integriert.

Hier habe ich die zweite Ebene in zwei Pakete aufgeteilt, eines, das die Anforderungen enthält, und eines, das das Verhalten der Umgebung und der einzelnen Systemkomponenten beschreibt.

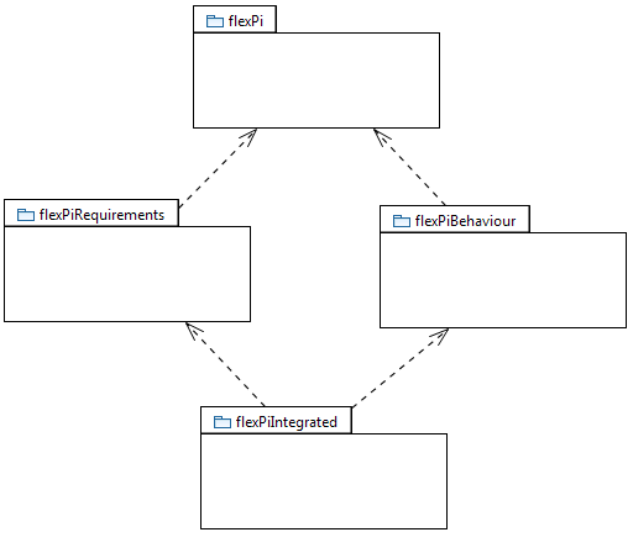

Abbildung 5-1: oberste Ebene der Modellierung[10]

[10] Die in diesem Abschnitt gezeigten Diagramme stellen nicht die finale Version der Modellierung dar. Es soll nur gezeigt werden, wie solche Diagramme aussehen.

Nun können im Paket „flexPi“ in einem Klassendiagramm die Eigenschaften der Komponenten definiert werden. Dabei sind die Operationen, die später mit den Nachrichten verknüpft werden, immer in der Klasse des Empfängerobjekts platziert. Die Operationen werden hier mit Zahnrädern dargestellt und können mit Parametern versehen werden. Wenn der Name des Parameters mit dem Namen eines Attributs übereinstimmt, kann über diese Operation das Attribut gesetzt werden.

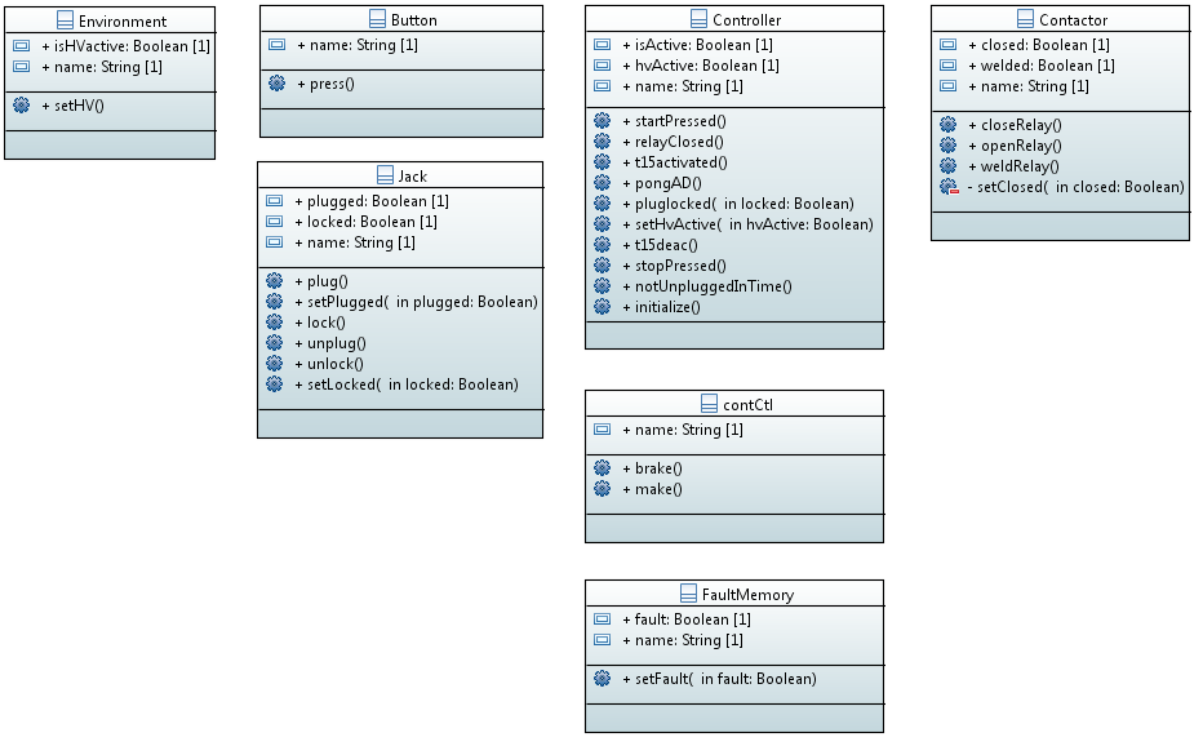

Abbildung 5-2: Klassendiagramm[10]

Die Objekte des Systems können dann als Instanzen der zuvor definierten Klassen in einem Kompositionsstrukturdiagramm (siehe Abbildung 5-3) erstellt werden. Alle Objekte müssen sich innerhalb einer sog. „Collaboration“ befinden. Die einzelnen Objekte werden als „Property“ angelegt. Außerdem wird jedem Objekt ein Name und eine Klasse zugewiesen. Dabei kann eine Klasse auch mehreren Objekten zugeordnet werden (siehe „start“ und „stop“).

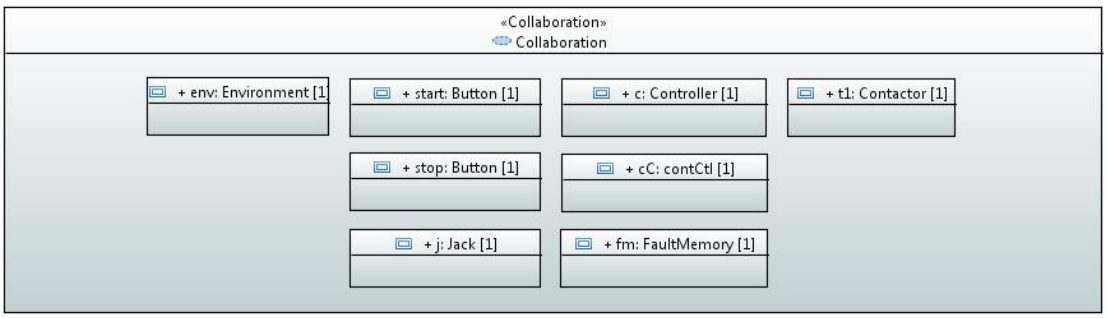

Abbildung 5-3: Collaboration Diagramm[10]

5.3 Modellierung der Anforderungen

Die einzelnen Teile einer Anforderung, die Verhalten modelliert, können in zwei Gruppen eingeordnet werden (siehe Kapitel 5.1). Zum einen sind das Aktionen, zum anderen Änderungen und Auswertungen von Attributen. Außerdem besteht eine Anforderung, die sich an die Vorgabe „Wenn…, dann…“ hält, immer aus einem Auslöser und einer Reaktion. Durch Kombination von Aktionen und der Verwendung von Attributen im Auslöser und in der Reaktion ergeben sich mehrere Arten von Verhaltensanforderungen. Da der Auslöser und die Reaktion strukturell unabhängig voneinander sind, betrachte ich diese nun getrennt. Die Darstellung der verschiedenen Auslöser und Reaktionen, wie ich sie im Rahmen dieser Arbeit entwickle, erleutere ich in den folgenden Unterkapiteln näher. Abbildung 5-4 zeigt den allgemeinen Aufbau eines MSDs, das eine Anforderung darstellt[11]. Dieser Aufbau ähnelt den von Harel und Marely beschriebenen *life sequence charts* (LSCs) [4], die mit den MSDs verwandt sind. Bei LSCs wird explizit zwischen einem *prechart*, das dem Auslöser entspricht, und einem *main chart*, das der Reaktion entspricht, unterschieden

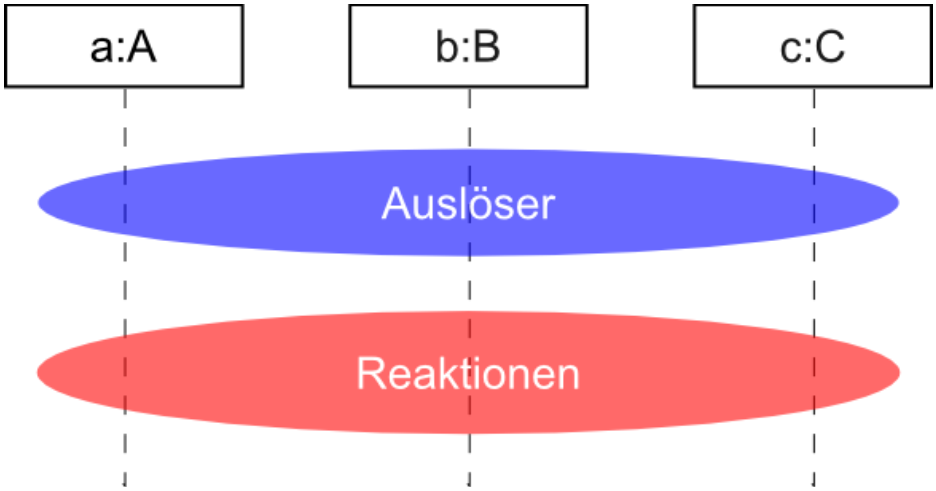

Abbildung 5-4: Aufbau einer als MSD dargestellten Anforderung

Grundsätzlich können MSDs nur mit Nachrichten aktiviert werden. Diese aktivierende Nachricht ist die erste Nachricht des Diagramms und somit Teil des Auslösers, jedoch nicht zwingend der gesamte Auslöser. Außerdem ist diese Nachricht *cold* und *monitored*.

5.3.1 Aktionen als Auslöser

Der einfachste Auslöser ist eine oder mehrere Aktionen mit festgelegter Reihenfolge. Die Nachrichten im Auslöser sind immer *cold*, da sie ansonsten eine Aktion darstellen, die bei Nicht-Erfüllung eine *Safety-Violation* auslöst. Die Ausführungsart ist stets *monitored*, da die Aktion die Anforderung auslöst und nicht von ihr ausgelöst werden soll. In Abbildung 5-5 ist ein MSD mit

[11] Der hier dargestellte Ansatz, ist nicht der einzige mögliche Ansatz, ein System mit MSDs zu modellieren. Er orientiert sich an der Empfehlung für die Struktur einer Anforderung: „Wenn…, dann…“

einer Aktion als Auslöser dargestellt. Wenn der Auslöser aus mehreren Nachrichten bestehen soll, sind diese in der richtigen Reihenfolge aufzulisten. Die erste Nachricht des Auslösers aktiviert das MSD.

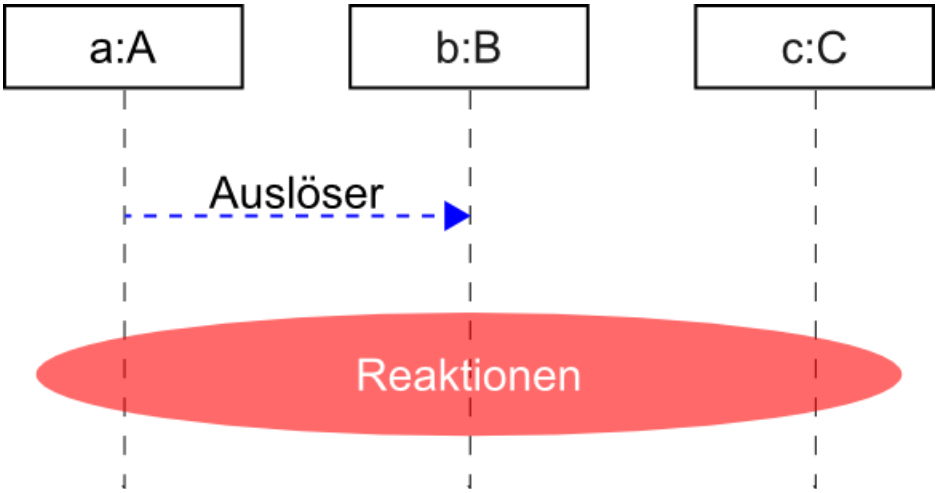

Abbildung 5-5: Aktion als Auslöser

Im in Abbildung 5-6 dargestellten Beispiel wird die Aktion „Schütz öffnen“ (*break()*) als Auslöser des Diagramms verwendet. Als Reaktion wird gefordert, dass das System irgendwann entladen ist. Dies wird durch das Setzen von *c.hvActive=false* dargestellt.

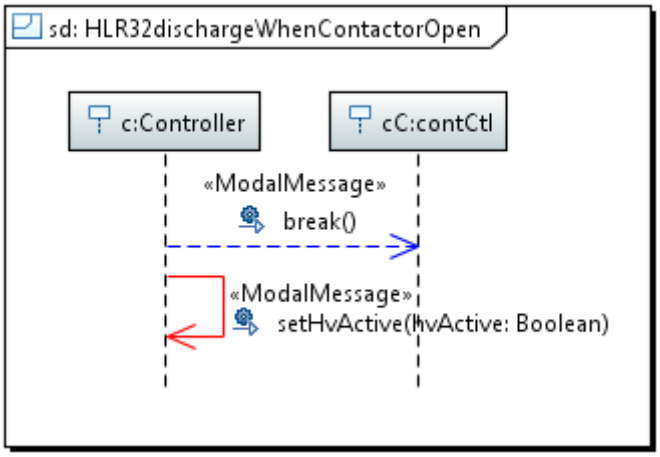

Abbildung 5-6: Beispiel für Aktion als Auslöser

5.3.2 Attributsänderungen als Auslöser

Da es MSDs nicht zulassen, durch Bedingungen ausgelöst zu werden, muss die Nachricht, die die Änderung des Attributs herbeiführt, als Auslöser verwendet werden. Diese Nachrichten habe ich meist als Nachrichten zu sich selbst modelliert. Sie bekommen den neuen Wert des Attributs als Argument übergeben. Dieser Wert ist im MSD in den Eigenschaften der Nachricht festzulegen. Somit werden Änderungen von Attributen wie Aktionen modelliert. Das Abfragen der Attribute erfolgt über Bedingungen (siehe Kapitel 5.3.3).

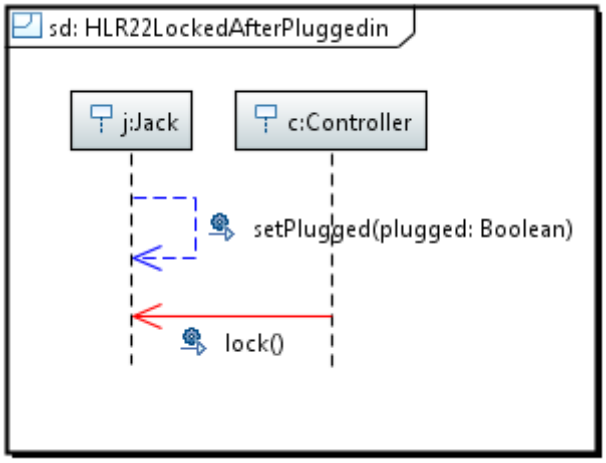

Abbildung 5-7: Beispiel für Änderung einer Objekteigenschaft als Auslöser

Im Beispiel, das in Abbildung 5-7 dargestellt ist, wird das Ändern des Attributs *j.plugged* durch die Funktion *setPlugged(plugged: Boolean)* als Auslöser verwendet. Die Reaktion, die darauf sofort erfolgen muss, ist das Verriegeln des Steckers (*lock()*).

5.3.3 Aktion und Attributswert als Auslöser

Wenn eine Aktion ausgeführt wird und abhängig vom Wert eines Attributs darauf eine Reaktion erfolgen soll, wird dies wie in Abbildung 5-8 dargestellt modelliert. Das Diagramm wird beim Auftreten der Nachricht a**ktivator** aktiviert. Wenn **bedA** nun *true* ist, dann wird die Reaktion ausgeführt. Andernfalls wird das Diagramm durch eine cold-Violation abgebrochen. Somit wird die Reaktion, auch wenn zu einem späteren Zeitpunkt **bedA** *true* wird, erst nach einem erneuten Auftreten der Nachricht **aktivator** ausgeführt. **bedA** ist hier ein OCL-Ausdruck der den Zustand eines Attributs prüft.

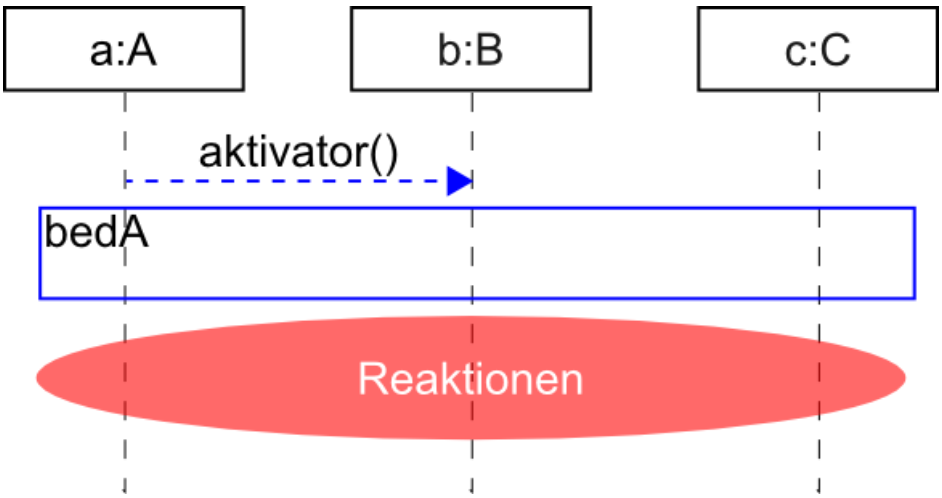

Abbildung 5-8: Aktion und Bedingung als Auslöser

In Abbildung 5-9 ist ein Beispiel dargestellt, bei dem der Auslöser aus einer Aktion (*stopPressed()*), die die Meldung des stop-Tasters an das Steuergerät darstellt, und einer Bedingung (*c.hvActive=false*) besteht.

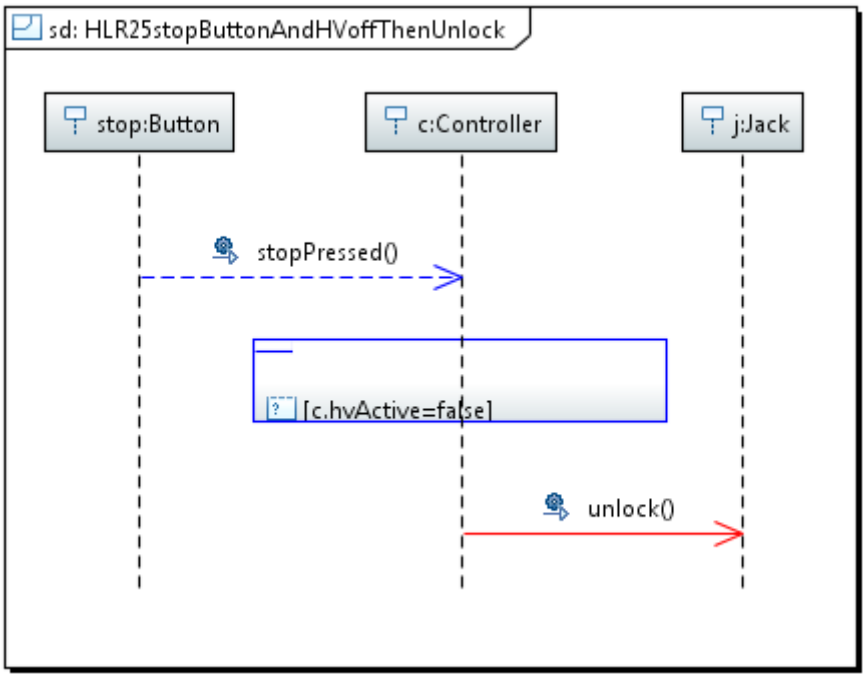

Abbildung 5-9: Beispiel für Auslösersystem aus Aktion und Objekteigenschaft

Es kann auch dargestellt werden, das nachdem eine Aktion ausgeführt wurde auf einen Zustand gewartet wird. Dies wird durch eine hot Bedingung dargestellt. Das Diagramm bleibt dann so lange vor der Bedingung stehen, bis diese *true* wird. Wenn während des Wartens vor der Bedingung die Nachricht **aktivator** erneut ausgeführt wird, wird eine *Safety*-Violation ausgelöst. Wenn ein erneutes Ausführen von **aktivator** zulässig sein soll, muss durch eine verbotene Nachricht (siehe Kapitel 5.3.10) das MSD mit einer cold-Violation beendet werden.

5.3.4 Mehrere Attribute als Auslöser

Anforderungen können nicht nur von Aktionen, sondern auch von Zuständen des Systems abhängen. Wenn gewünscht ist, dass gewartet wird, bis die Bedingung *true* wird, wie es z.B. bei einer Und-Verknüpfung zweier Attributswerte der Fall ist, muss die Bedingung, wie in Abbildung 5-10 dargestellt, hot sein. Das in Abbildung 5-10 dargestellte MSD wird aktiviert, sobald **eigA** auf *true* gesetzt wird. Dann wird gewartet, bis **bedB** *true* ist. Dabei ist unerheblich, ob der Wert vor oder nach dem Setzen von **eigA** *true* wird. Wenn während des Wartens **eigA** erneut auf *true* gesetzt wird, kommt es zu einer safety-Violation, da ein MSD nicht mehrfach aktiviert sein darf. Dies kann umgangen werden, indem das Setzen von **eigA** als *cold* verbotene Nachricht in das Diagramm

eingefügt wird (siehe 5.3.10). Dann wird das Diagramm durch diese Nachricht beendet und somit ist das Diagramm nicht mehrfach aktiviert.

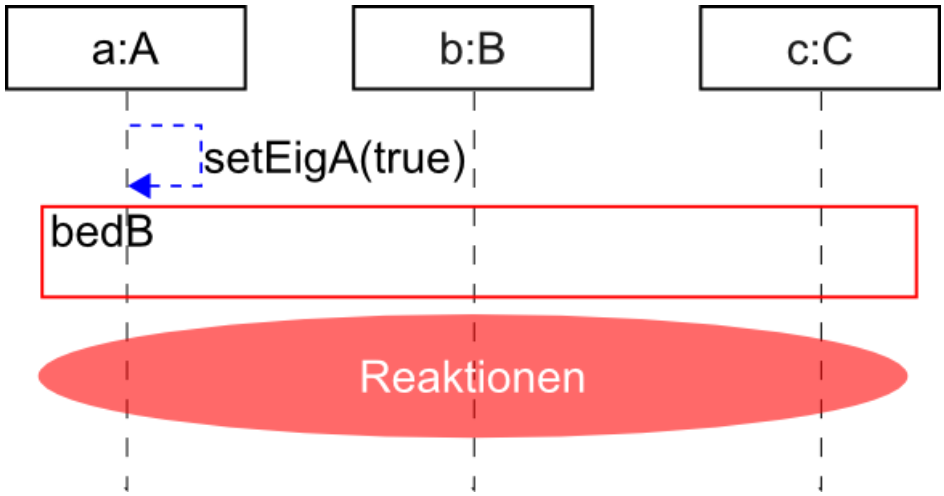

Abbildung 5-10: "Und"-Verknüpfung zweier Bedingungen als Auslöser

Wenn im Gegensatz dazu gefordert wird, dass **bedB** bereits *true* ist, wenn **eigA** gesetzt wird, dann muss **bedB** *cold* sein. Dann wird, wenn **bedB** *false* ist, nicht gewartet, sondern der Cut des MSDs zurückgesetzt. Somit muss, um die Reaktion zu erreichen, **eigA** erneut gesetzt werden.

5.3.5 Alternative Aktionen als Auslöser

Es kann vorkommen, dass eine Anforderung mehrere Auslöser hat, die Oder-Verknüpft sind. Dies widerspricht der Forderung nach atomaren Anforderungen kann bei nicht optimalen Anforderungsformulierungen jedoch vorkommen. Dies wird durch ein Kombiniertes Fragment mit dem Operator „alt“ (alternativ) dargestellt. Das Kombinierte Fragment besteht aus mehreren Operanden, von denen bei der Ausführung des MSDs der Operand gewählt wird, der mit der ausgelösten Nachricht übereinstimmt. Der andere Operand kann dann bei diesem Durchlauf des MSDs nicht mehr ausgeführt werden.

In Abbildung 5-11 ist dies exemplarisch dargestellt. Wenn Auslöser 1 ausgeführt wird, erfolgt die Reaktion ebenso, wie wenn Auslöser 2 ausgeführt wird. Beide Auslöser können auch aus mehreren Nachrichten bestehen. Sogar Bedingungen innerhalb der Operanden sind möglich. Da die dieser Modellierung zugrunde liegende Anforderungsformulierung nicht dem Forderung nach atomaren Anforderungen entspricht gibt es aus dem Beispielprojekt kein Beispiel, jedoch wird im Folgenden das Sprachelement verwendet.

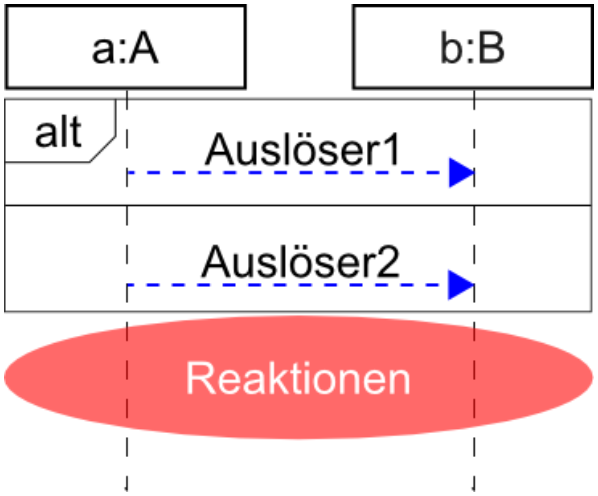

Abbildung 5-11: Alternative Auslöser

5.3.6 Quasiparallele Aktionen als Auslöser

In vielen Fällen ist es unerheblich, in welcher Reihenfolge Elemente eines Auslösers auftreten. Deshalb wird i.d.R. in einer Anforderung eine Reihenfolge nur festgelegt, wenn dies explizit gefordert werden muss. Um dies darzustellen ist das in Kapitel 5.6.2 beschrieben Sprachelement notwendig. Da dieses nicht in ScenarioTools implementiert ist, stelle ich hier eine alternative Modellierungsweise vor, die jedoch nur für wenige Nachrichten praktikabel ist.

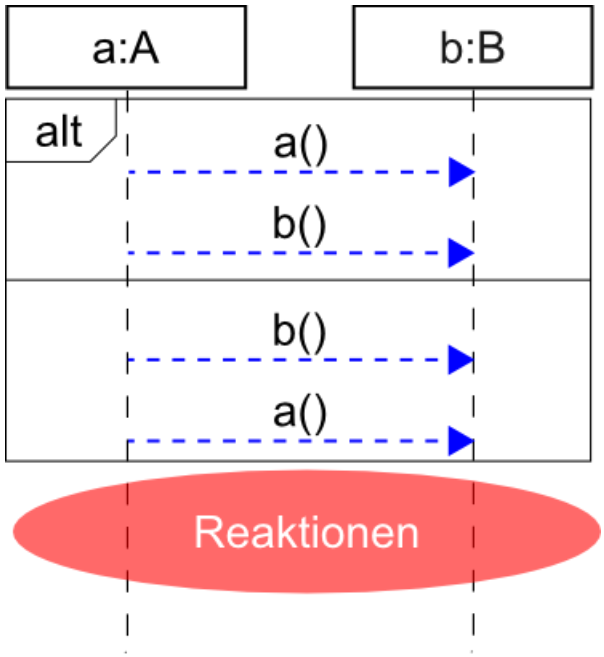

Abbildung 5-12: Parallele Aktionen mit "par" kombiniertem Fragment

Hierfür müssen alle möglichen Reihenfolgen der Auslösenden Nachrichten in einem kombinierten Fragment vom Typ „alt“ eingefügt werden. Bei zwei Nachrichten a() und b() müssen in diesem kombinierten Fragment zwei Interaktionsoperanden eingefügt werden, in die dann jeweils die Nachrichten a() und b() bzw. b() und a() eingefügt werden (siehe Abbildung 5-12).

5.3.7 Aktion als Reaktion

Die Reaktion kann eine Folge von Aktionen sein. Dabei ist die Reihenfolge festgelegt. Durch die Ausführungsart kann festgelegt werden, ob die Nachricht in Folge des Auslösers sofort ausgeführt werden muss (*executed*) oder ob gefordert wird, dass sie durch eine andere Anforderung ausgeführt wird (*monitored*). In dieser anderen Anforderung muss die Nachricht dann *executed* sein.

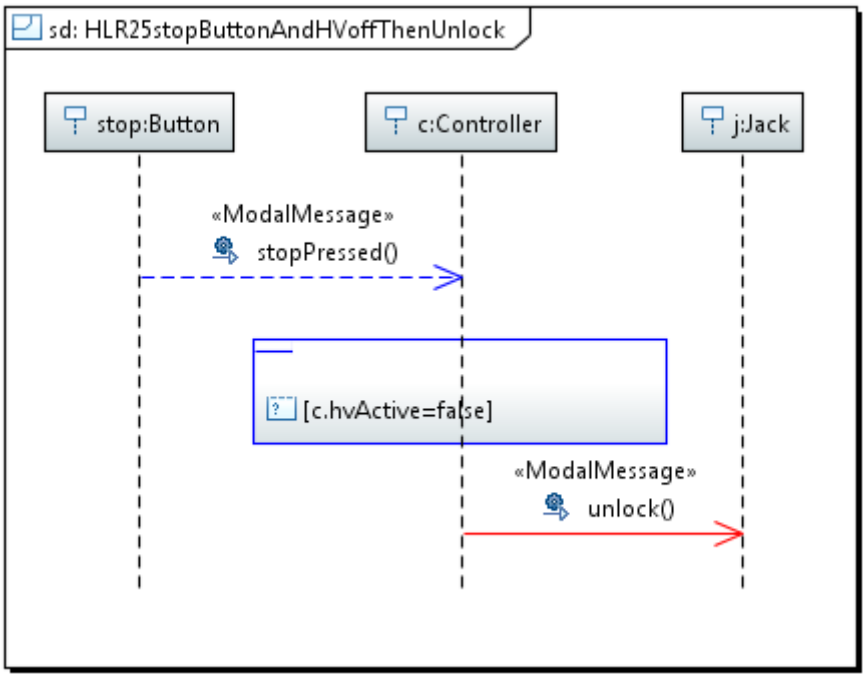

Abbildung 5-13: Beispiel für Aktion als Reaktion

Im in Abbildung 5-13 dargestellten Beispiel, wird die Aktion „Stecker entriegeln“ (*unlock()*) ausgeführt, in Folge des Auslösers, dass der stop-Taster betätigt wurde (*stopPressed()*) und das System HV-inaktiv[12] (*c.hvActive=false*) ist. Das Entriegeln muss hier erfolgen.

5.3.8 Änderung eines Attributs als Reaktion

Wenn sich als Reaktion eine oder mehrere Attribute ändern sollen, dann sind Nachrichten mit Seiteneffekt zu verwenden, denen als Argument der gewünschte Wert übergeben werden muss (vgl. 5.3.2). Da es sich hierbei ebenfalls um Nachrichten handelt, können diese beliebig mit den Nachrichten von Aktionen kombiniert werden.

In Abbildung 5-14 ist ein Beispiel für das Ändern eines Attributs dargestellt. Wenn die Aktion, die den Auslöser darstellt (*unplug()*). Ausgeführt wird, wird als Reaktion der Wert von *j.plugged* geän-

[12] Alle Schütze sind geöffnet und die Spannung am Nebenaggregat ist abgebaut. Das System befindet sich also im sicheren Zustand.

dert (*setPlugged(plugged: Boolean)*). In diesem Fall wird der Wert auf *false* gesetzt, was in den Diagrammen nicht dargestellt wird, jedoch bei den Einstellungen der Nachricht angegeben ist.

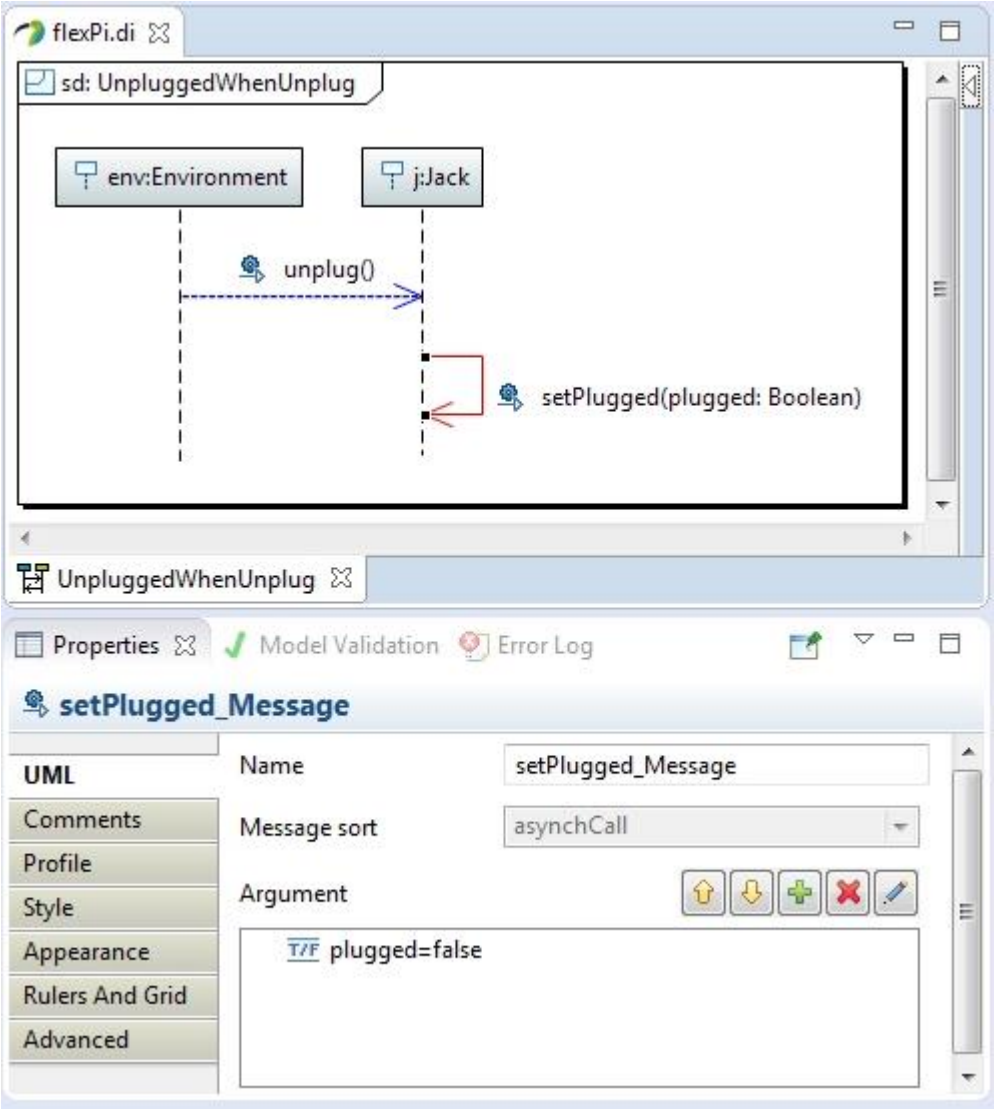

Abbildung 5-14: Beispiel für Änderung einer Objekteigenschaft

5.3.9 Antiszenarien

Antiszenarien haben als Reaktion eine violation-Bedingung. Wenn in einem solchen Diagramm die violation-Bedingung erreicht wird, wird eine safety-Violation ausgelöst, d.h. es ist gegen eine Anforderung verstoßen worden. Es ist dabei darauf zu achten, dass eine Abweichung von dem Antiszenario eine cold-Violation verursacht. D.h. alle Nachrichten und Bedingungen des Auslösers sollten mit der Temperatur cold versehen sein.

Ein Beispiel für ein Antiszenario ist Anforderung HLR_27 „Nur wenn gesteckt“. Hier wird das Schließen der Schütze (*make()*) als erster Auslöser verwendet. Dieser wird von einer Bedingung gefolgt, die den Zustand der Verriegelung (*j.locked*) und den gesteckt-Zustand (*j.plugged*) abfragt. Wenn einer dieser Zustände *false* ist, dann wird als Reaktion die violation-Bedingung ausgeführt.

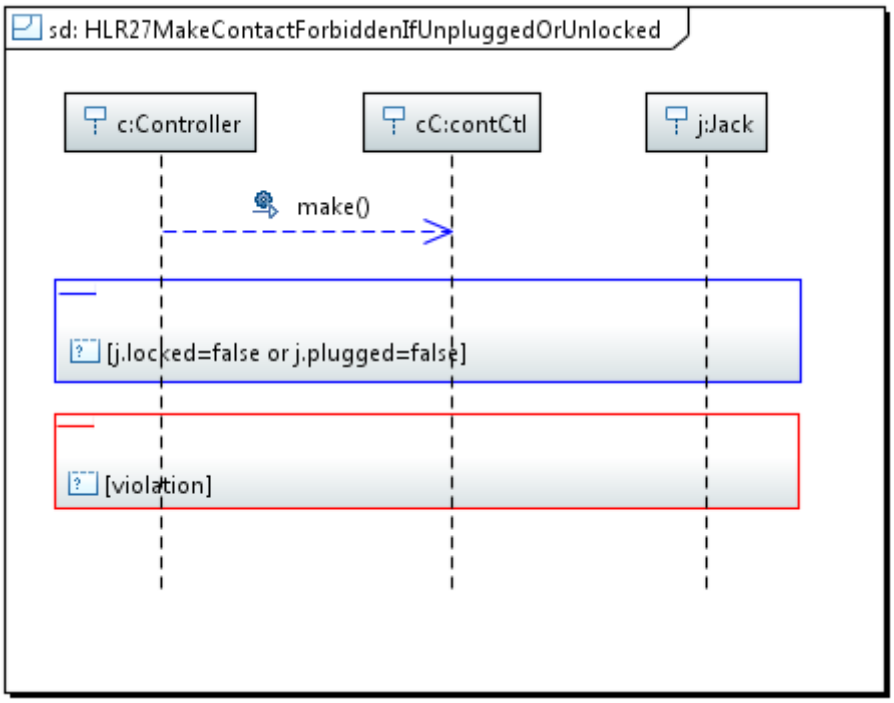

Abbildung 5-15: Beispiel für Antiszenario

5.3.10 Verhalten einschränken

Mit MSDs kann auch das Verhalten eingeschränkt werden. D.h. während ein Diagramm ausgeführt wird, werden bestimmt Nachrichten verboten. Dies stellt eine Abbruchbedingung dar, die wenn sie in Kraft tritt je nach Temperatur der verbotenen Nachricht eine cold- oder safety-violation auslöst. Diese Art von Nachrichten kann bei assumption-MSDs vorkommen, um darzustellen, dass eine Aktion keinen Sinn macht, wie z.B. das erneute Stecken bevor der Stecker wieder ausgesteckt wurde. Es können aber auch in requirement-MSDs Aktionen verboten werden, wie das Schließen der Schütze, wenn ein Fehler vorliegt (siehe Abbildung 5-16). Hier wird mit der Änderung der Objekteigenschaft (*setFault(true)*) das MSD aktiviert und an der Bedingung so lange gewartet, bis die Objekteigenschaft wieder auf *false* gesetzt wird. In der Zwischenzeit würde ein Aufruf der verbotenen Nachricht, die die Aktion *make()* darstellt, zu einer *Safety-Violation* führen. Eine Nachricht ist in ScenarioTools dann verboten, wenn sie sich in einem kombinierten Fragment mit dem Schlüsselwort „neg“ befindet. Dieses Schlüsselwort wird nicht im Diagramm angezeigt, kann jedoch bei den Einstellungen des kombinierten Fragments angegeben werden (siehe Abbildung 5-16).

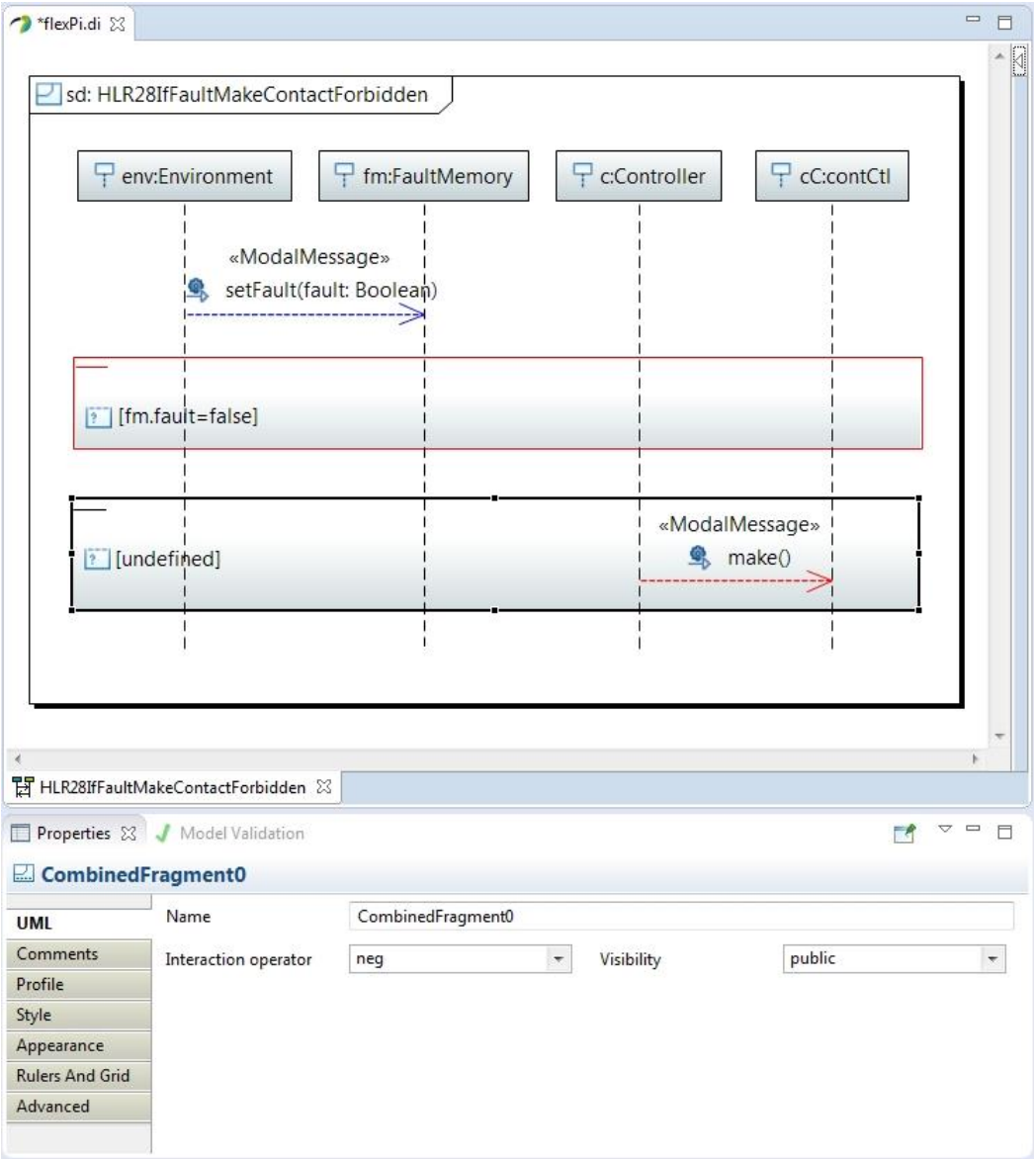

Abbildung 5-16: Beispiel für verbotene Nachricht

5.4 Modellierung der Annahmen

Die schon zuvor bei der GuR und der Anforderungsliste aufgestellten Annahmen über das System müssen für die Analyse ebenso wie die Anforderungen modelliert werden. Hierbei werden Annahmen über die Umgebung des Systems als assumption-MSDs modelliert (siehe 5.4.1). Annahmen, wie sich eine einzelne Systemkomponente verhält (z.B. ein Relais schließt, wenn es angesteuert wird) können als requirement-MSD formuliert werden (siehe 5.4.2). Wenn das Gesamtsystem betrachtet wird, sollten diese als Anforderungen an die Komponenten formuliert werden. Aus Sicht des Softwareentwicklers, der nur die Softwarekomponente entwickeln soll sind dies Annahmen. In der bisherigen Praxis wurde die Formulierung dieser Annahmen oft vernachlässigt, durch die Methode, die ich in dieser Arbeit teste, werden sie erzwungen.

5.4.1 Umweltannahmen

Annahmen über die Umwelt werden in assumption-MSDs dargestellt. Diese Diagramme werden in ScenarioTools durch den Stereotyp *EnvironmentAssumption* gekennzeichnet.

Mit Umweltannahmen kann die Reihenfolge von Nachrichten aus der Umwelt erzwungen werden. Es macht z.B. keinen Sinn den Stecker erneut einzustecken, bevor er wieder ausgesteckt wurde. Dies wird durch zwei Diagramme dargestellt.

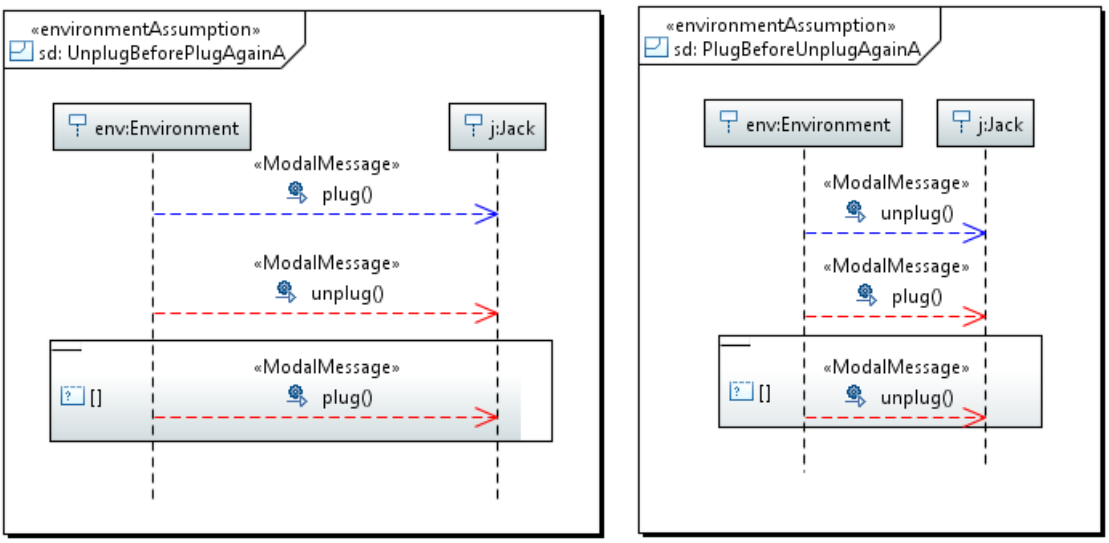

Abbildung 5-17: Umweltannahme, Stecker stecken/ausstecken

Die in Abbildung 5-17 abgebildeten assumption-MSDs erzwingen, dass das Ein- und Ausstecken des Steckers abwechseln erfolgen muss. Sobald das Diagramm „UnplugBeforePlugAgainA“ aktiviert wurde, werden diese beiden Diagramme immer abwechselnd aktiviert, da die letzte Nachricht des einen Diagramms die erste Nachricht des anderen ist. Die Nachrichten, die jeweils ganz unten im kombinierten Fragment enthalten sind, sind verbotene Nachrichten. Somit ist das erneute Ausführen von *plug()* bevor *unplug()* ausgeführt werden darf, bzw. durch das andere Diagramm anders herum, untersagt und würde zu eine hot-Violation führen.

5.4.2 Systemannahmen

Neben den Umweltannahmen kann es Systemannahmen geben. Diese sind in ScenarioTools nicht explizit vorgesehen, können aber modelliert werden. Systemannahmen stellen die Annahmen aus der GuR (siehe Kapitel 4.1) und der Anforderungsliste (siehe Kapitel 4.2) dar. Die dazugehörigen Diagramme lassen sich je nach Ebene der Betrachtung, wie auch die Annahmen in den Listen, auch in Anforderungen überführen. Bei der Modellierung ist wie bei den Anforderungen vorzugehen.

5.5 Gefundene Lücken durch Modellierung

Durch eine nähere Kenntnis über das System können gewisse Zusammenhänge als selbstverständlich angenommen werden und somit bei der Aufstellung der Anforderungen vergessen werden. Solche Fehler können durch die Modellierung bereits auffallen.

Dies sei am folgenden Beispiel näher dargestellt.

Ausschalten des Steuergeräts	HLR_10		FlexPi-Funktionen dürfen erst abgeschaltet werden, wenn sicher Zustand (HV-inaktiv) erreicht ist.
Ausschalten des Steuergeräts	HLR_12	EA_1, EA_2	Wenn Klemme 15 deaktiviert wird, muss die flexPi-Schnittstelle in den sicheren Zustand versetzen werden.
Ausschalten des Steuergeräts, sicherer Zustand	HLR_13	HLR_12, HLR_10	Wenn Klemme 15 deaktiviert und die flexPi-Schnittstelle in sicherem Zustand ist, dann sollen die flexPi-Funktionen abgeschaltet werden.

Tabelle 5-1: HLR10, HLR12, HLR13 inkonsistent

Die Intension dieser Anforderung ist die, dass wenn Klemme 15**Fehler! Textmarke nicht definiert.** deaktiviert wird, die Schnittstelle in einen sicheren Zustand versetzt wird, d.h. die Schütze werden geöffnet. Sobald dieser Zustand erreicht ist, wird die *flex*^{Pi}-Funktion, d.h. das Steuergerät, abgeschaltet. Im konkreten Beispiel ist dies schlüssig. Wenn sich die Schnittstelle bereits im sicheren Zustand befindet, muss sie natürlich nicht mehr in diesen versetzt werden.

Im Folgenden werden die konkreten Elemente durch Platzhalter ersetzt. Das Deaktivieren der Klemme 15 sei Bedingung A, der sichere Zustand sei Bedingung B, das deaktivieren der flexPi-Funktionen sei Aktion C und das Herstellen des sicheren Zustands sei Aktion D. Dann lauten die Anforderungen wie folgt.

HLR_10		C darf erst ausgeführt werden, wenn B erfüllt ist.
HLR_12		Wenn A, muss Aktion D ausgeführt werden.
HLR_13	HLR_10, HLR_12	Wenn A und B, dann C.

Tabelle 5-2: HLR09, HLR11, HLR12 inkonsistent und abstrakt

Es fällt auf, das HLR_12 von der Bedingung B unabhängig ist. Das würde im konkreten Beispiel bedeuten, dass der sichere Zustand auch hergestellt wird, wenn er bereits vorliegt. Das macht keinen Sinn. Korrekt ist in diesem Fall, den Auslöser von HLR_12 mit der negierten Bedingung B zu erweitern.

HLR_12		Wenn A und nicht B, muss Aktion D ausgeführt werden.

Tabelle 5-3: HLR11 korrigiert und abstrakt

Die konkrete HLR_11 lautet somit

Ausschalten des Steuergeräts	HLR_11		Wenn Klemme 15 deaktiviert wird und die Schnittstelle sich nicht im sicheren Zustand befindet, muss die flexPi-Schnittstelle in den sicheren Zustand versetzen werden.

Tabelle 5-4: HLR11 korrigiert und konkret

Die Anforderungen sehen als MSDs dann wie folgt aus. Durch die Erweiterung ist die Bedingung in „HLR11WhenT15DeacThenSafeState“ hinzugekommen.

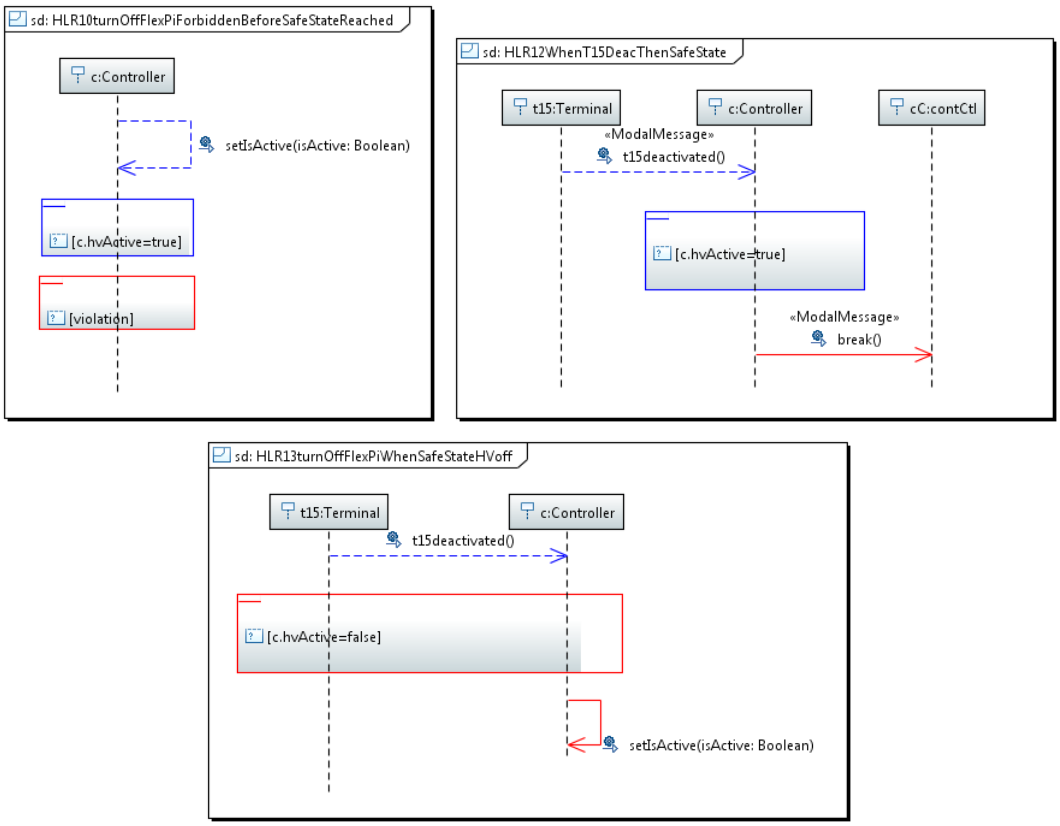

Abbildung 5-18: HLR10, HLR12, HLR13 nach Korrektur

In diesem Fall wäre ein solches Vorgehen nicht schlimm, da durch die Aktion D die Versorgung der Schütze abgeschaltet wird und ein doppeltes Abschalten keine Auswirkung hätte, jedoch sind auch Fälle denkbar, in denen eine doppelte Ansteuerung zu Schäden des Systems führen könnte.

5.6 Zusätzliche Sprachelemente

5.6.1 Schleifen

Hierfür kann ein kombiniertes Fragment mit dem Interaktionsoperator „loop“ verwendet werden. Im Wächter des Interaktionsoperanden muss eine Bedingung gesetzt werden, die den Durchlauf der Schleife kontrolliert. Dies kann entweder eine Anzahl von Durchläufen oder so lange, bis eine Bedingung *true* ist, sein.

5.6.2 Parallele Abläufe

Um parallele Abläufe darzustellen werden in UML-Sequenzdiagrammen kombiniert Fragmente mit dem Interaktionsoperanden „par“ verwendet [15]. Als Sprachelement von Sequenzdiagrammen sind auch diese kombinierten Fragmente in MSDs denkbar.

Dabei können die einzelnen Operanden des kombinierten Fragments parallel auftreten. Vom Interpreter des MSDs muss gefordert werden, dass die Operanden in beliebiger Reihenfolge abgearbeitet werden, jedoch das kombinierte Fragment erst verlassen wird, sobald alle Operanden durchlaufen sind. Wenn ein Operand mehrere Nachrichten beinhaltet, dann können zwischen diesen Nachrichten auch Nachrichten aus anderen Operanden erfolgen. Die Nachrichten innerhalb eines Operanden müssen, wie auch in anderen kombinierten Fragmenten und außerhalb von kombinierten Fragmenten, in exakt dieser Reihenfolge erfolgen.

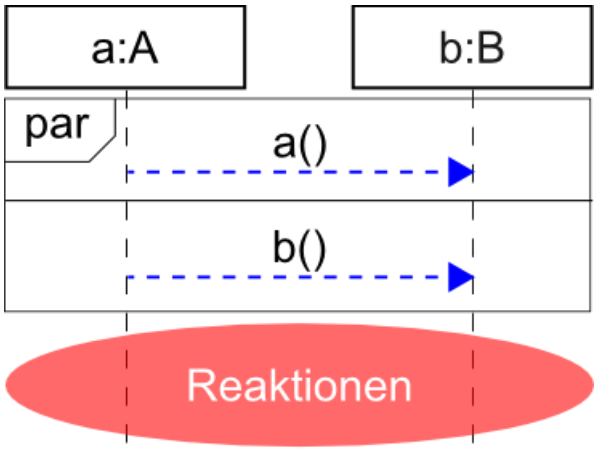

Abbildung 5-19: Paralleler Ablauf mit kombinierten Fragment "par"

In Abbildung 5-19 ist eine Anforderung mit dem kombinierten Fragment mit dem Operator „par“ dargestellt. Dieses fordert, dass wenn die Nachrichten a() und b() erfolgen eine Reaktion folgt. Dabei ist es unerheblich in welcher Reihenfolge diese Nachrichten erfolgen. Sie könnten auch gleichzeitig erfolgen. Das Verhalten entspricht dem Verhalten des MSDs, das in Abbildung 5-12 dargestellt ist. Bei der in Kapitel 5.3.6 dargestellten Modellierungsform sind bei n Nachrichten $n!$ Interaktionsoperanden notewendig. Bei drei Nachrichten, die gleichzeitig ausgeführt werden

sollen, sind dies bereits sechs Operanden. Mit dem Operator „par“ ist pro Nachricht nur ein Interaktionoperand notwendig. Durch Einführung dieses Sprachelements werden Anforderungen möglich, bei denen eine Reihe von Vorbedingungen, wie Zustandsprüfungen, erfolgen müssen, deren Reihenfolge jedoch nicht festgelegt ist.

Die Forderungen an das Sprachelement würde ich wie folgt umsetzen: Sobald das kombinierte Fragment aktiviert ist, müssen die einzelnen Interaktionsoperanden als einzelne Unterinteraktionen betrachtet werden und jeweils einen eigenen Cut haben. Jede aktivierte Nachricht einer Unterinteraktion ist dann auch eine mögliche nächste Nachricht für das MSD. Erst wenn alle Interaktionsoperanden durchlaufen sind, wird der Cut des MSDs hinter das kombinierte Fragment gesetzt.

6 Synthese eines Automaten und Anforderungsanalyse

Aus den als MSDs modellierten Anforderungen kann mit Tools wie ScenarioTools ein Zustandsautomat synthetisiert werden. In diesem Kapitel stelle ich, das von mir entworfene, Schema vor, mit dem, von diesem Zustandsautomaten ausgehend, die Mängel analysiert und behoben werden können. In Kapitel 6.1 erkläre ich, wie die Synthese von ScenarioTools durchgeführt und das Ergebnis dargestellt wird. Die Analyse des Automaten und die daraus resultierenden Korrekturen an den Anforderungen bzw. der Modellierung stelle ich in Kapitel 6.2 dar.

6.1 Synthese

Bei der Synthese werden ausgehend von einem Ausgangszustand, in dem kein MSD aktiviert ist, alle Environment-Events gesendet. Da, wie in Kapitel 2.5.1 beschrieben, das Empfangsevent unmittelbar auf das Sendeevent folgt bezeichne ich dies, ebenso wie in der Ausarbeitung von Joel Greenyer [3], als ein Event. Jedes Event wird durch eine Transition zu einem Zustand dargestellt. Wenn in diesem Zustand andere Diagramme aktiviert sind bzw. die MSDs sich an einer anderen Stelle befinden, als beim Ausgangszustand führt diese Transition zu einem neuen Zustand. Wenn sich an den aktivierten MSDs nichts geändert hat, weil das durch das Event aktivierte MSD sofort durch eine cold-Violation beendet wird, wird das Event durch eine Selbsttransition dargestellt. Anschließend werden alle in den MSDs vorgeschriebenen executed Nachrichten als Events ausgeführt. Diese führen von dem jeweiligen Zustand zu neuen Zuständen. Die Events aus diesen Nachrichten werden als System-Events bezeichnet. Es werden alle System-Events ausgeführt, bis keine executed-Nachricht übrig ist. Von den daraus resultierenden Endzuständen werden jeweils alle Environment-Events ausgelöst, die nicht durch ein assumption-MSD verboten sind. Wenn ein Zustand erreicht wird, der bereits existiert, wird die Transition zu dem bereits existierenden Zustand gezogen. Die Synthese wird so lange durchgeführt, bis eine Anforderung verletzt wurde, oder kein Zustand mehr übrig ist, von dem ausgehende Transitionen existieren, die zu noch nicht existierenden Zuständen führen.

Im folgenden abstrakten Beispiel demonstriere ich diesen Vorgang. Die verwendeten MSDs sind in Abbildung 6-1 dargestellt.

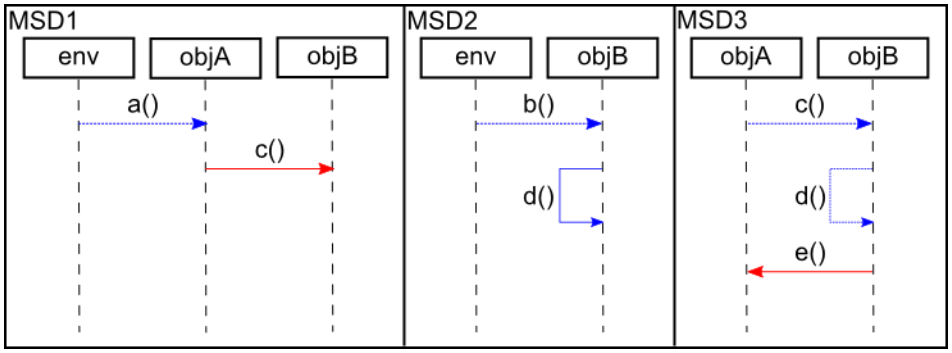

Abbildung 6-1: MSDs für Synthesebeispiel

Der Ausgangszustand ist in Abbildung 6-2, in Anlehnung an die Darstellung aus ScenarioTools, rot dargestellt und mit der Nummer 1 versehen. Die Environment-Events aus den MSDs sind a() und b() – sie gehen von der Lebenslinie des Environment-Objekts (env) aus. Zunächst wird eines dieser Events – hier a() – ausgeführt. Hiermit wird der Zustand 2 erreicht, in dem das MSD1 aktiviert ist und sich auf beiden Lebenslinien nach dem ersten Element, also zwischen den beiden Nachrichten befindet. Die aktiven MSDs werden im Zustand aufgelistet und die Position im MSD durch die Zahlen in Klammern dargestellt, wobei (1,1) bedeutet, dass die Stelle auf beiden Lebenslinien nach der ersten Nachricht ist. Anschließend muss das Event c() erfolgen, da es in MSD1 die folgende executed Nachricht ist. Diese Transition führt zu Zustand 3, in dem es keine weitere Systemnachricht gibt, die executed ist. Dann wird vom Ausgangszustand aus das zweite Environment-Event (b()) ausgeführt. Dieses aktiviert MSD2, das das Event d() auslöst und wieder zurück zu Zustand 1 führt. Zustand 3 wird dann als neuer Ausgangszustand betrachtet und von dort aus werden die Environment-Events ausgelöst. a() führt über Zustand 5, in dem MSD1 aktiviert wird, zu einer Anforderungsverletzung, da es durch c() das aktive MSD3 erneut aktivieren würde und ein MSD nur einmal aktiviert sein darf. Mit Nachricht b() wird Zustand 7 erreicht in dem durch MSD2 das Event d() ausgelöst wird. Nun kann in Zustand 8, der mit d() erreicht wird, das MSD3 durch den Event e() erfolgreich beendet werden. Da nun keine MSDs mehr aktiviert sind, wird der ursprüngliche Ausgangszustand 1 erreicht.

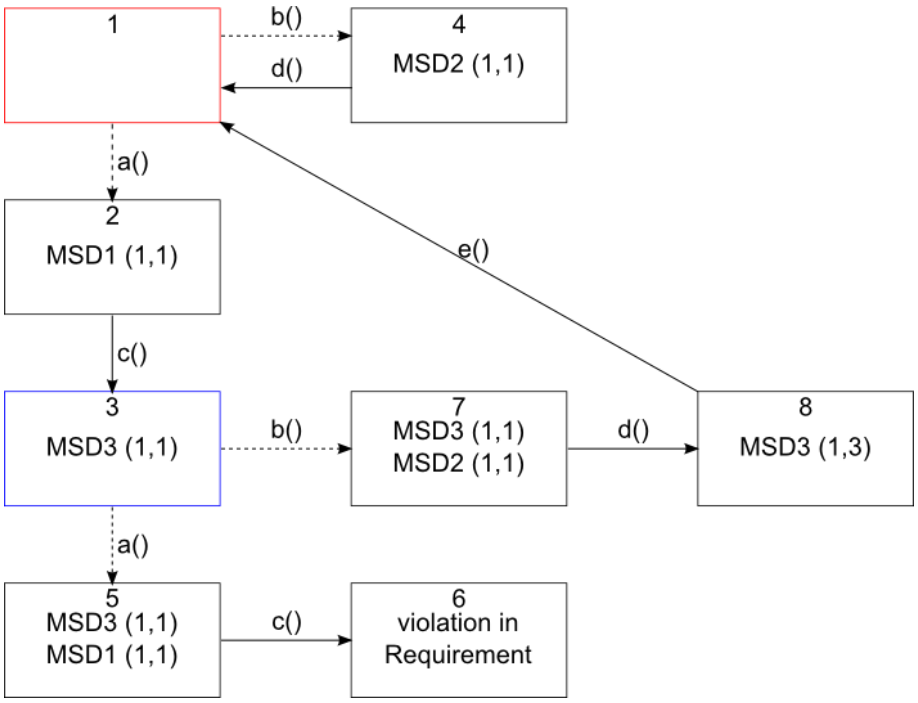

Abbildung 6-2: Zustandsautomat aus Synthesebeispiel

6.1.1 Zustände

Die Zustände werden durch die aktivierten MSDs und die erreichten Positionen in den Diagrammen identifiziert. Des Weiteren werden der Ausgangszustand (rot umrahmt) und Wartezustände (blau umrahmt) herausgehoben. Der Ausgangszustand ist ein spezieller Wartezustand, in dem keine MSDs aktiviert sind. Wartezustände, sind die Zustände, in denen das System auf ein Ereignis von außen wartet. Es wird davon ausgegangen, dass die Zustände zwischen den Wartezuständen ausreichend schnell durchlaufen werden, sodass kein Ereignis aus der Umgebung erfolgt, bevor ein Wartezustand erreicht wurde. Dabei sind die Zustände entweder winning oder losing. Ein Zustand ist dann *winning*, wenn von ihm aus keine Anforderungsverletzung erreicht werden kann und das System unendlich oft einen Wartezustand erreicht, ohne das ein assumption-MSD verletzt wird [3]. Alle anderen Zustände sind *losing*. In, dem in Abbildung 6-2 dargestelltem Zustandsautomat, sind durch die Anforderungsverletzung im Zustand 6 alle Zustände losing. In ScenarioTools wird dies durch eine grüne (winning) bzw. rote (losing) Füllung der Zustände dargestellt.

6.1.2 Dynamisches Objektsystem

Um das Verhalten von technischen Systemen darzustellen, ist es oft erforderlich Zustände von Systemkomponenten darzustellen. In Kapitel 5 wurden diese Zustände als Attribute modelliert. Die Objekte und ihre Attribute sind im Objektsystem enthalten. Da auch bei der Synthese des Zustandsautomaten die Attribute der Objekte berücksichtigt werden müssen, bietet ScenarioTools die Option dynamische Objektsysteme zu verwenden. Diese Option muss in der *scenari-*

orunconfiguration-Datei mit der Option *dynamicObjectSystem="true"* aktiviert werden. Hierbei wird zur Identifikation der Zustände, zu den oben genannten Merkmalen, das Objektsystem hinzugefügt und über Nachrichten mit Seiteneffekten die Attribute im Objektsystem manipuliert. Damit werden die Zustände nun auch an den Werten der Attribute unterschieden. Mit diesem zusätzlichen Unterscheidungsmerkmal wächst die Zahl der synthetisierten Zustände erheblich, jedoch ist diese Option in Systemen mit Zustandscharakter, wie sie in der Automobilindustrie sehr oft vorkommen, notwendig.

6.1.3 Export

Der synthetisierte Graph liegt zunächst in einer XML-Struktur vor und ist somit für den Entwickler nicht übersichtlich dargestellt. Um eine grafische Darstellung zu erhalten bietet „ScenarioTools" zwei PDF-Exportfunktionen. „Export PDF…" produziert eine einzelnen Grafen, in dem alle Zustände dargestellt sind. Da schon bei überschaubaren Systemen, wie dem hier verwendeten Beispiel, eine sehr große Zahl von Zuständen entsteht, werden diese Diagramme schnell unübersichtlich. Deshalb habe ich eine zweite Exportfunktion implementiert (siehe B.5), die in einem Umweltereignisgraphen nur die Wartezustände und die Umweltereignisse darstellt. Die Informationen über die System-Nachrichten, die zwischen den Wartezuständen liegen, werden in zusätzlichen Superstepgraphen[13] dargestellt, wobei diese Graphen mit der Nummer des Ziel-Wartezustands beschriftet werden. Der Graph zu einem Ziel-Wartezustand enthält alle Zustände der Supersteps, die zu diesem Wartezustand führen. Anhand des Umweltereignisgraphen lassen sich jene Stellen ausmachen, in denen Anforderungen verletzt werden. Mit dem dazugehörenden Superstepgraphen kann die Anforderungsverletzung analysiert werden. Des Weiteren habe ich die Ausgabe in den generierten Graphen derart erweitert, dass die Werte der Attribute der Objekte in jedem Zustand ausgegeben werden (siehe Anhang B.3). Diese Information hilft, um das Verhalten des Systems nachvollziehen zu können, ohne die gesamte Historie bis zu dem betrachteten Zustand verfolgen zu müssen. Zudem habe ich bei Zuständen, in denen eine Anforderungsverletzung vorliegt, den Namen des verletzten MSDs hinzugefügt (siehe Anhang B.4).

6.1.4 Counter-Strategie

Ein weiterer Ansatz zur Analyse von Anforderungsverletzungen ist die von Gutjahr in ScenarioTools integrierte Methode [16]. Hierbei wird nach der Synthese des Zustandsgraphen eine Counter-Strategie erstellt. Diese ist eine Strategie, die das System schnellst möglich in eine Anforderungsverletzung führt. Dabei wird nur die Verletzung berücksichtigt, die am schnellsten erreicht werden kann. Diese Strategie kann zur Simulation als zusätzliche Umweltannahme hinzugeführt

[13] Superstep: Ein Superstep beinhaltet alle Events, die von einem Wartezustand zum nächsten führen.

werden, sodass der Entwickler in der interaktiven Simulation gegen diese Strategie spielen kann. Diese Zusätzliche Annahme lässt dann nur noch Umwelteinflüsse zu, die in Richtung der Anforderungsverletzung führen. Somit kann er jene Stelle identifizieren an der sich die Umwelt und das System anders als erwartet verhalten. An dieser Stelle lässt sich auch ablesen, welche MSDs und Nachrichten zu diesem Verhalten führen. Alternativ kann die Strategie auch als Graph in eine PDF-Datei exportiert und somit betrachtet werden.

6.1.5 Nicht erreichte MSDs

Bei der Synthese erstellt ScenarioTools eine Liste, die alle nicht aktivierten MSDs enthält. Diese Feature habe ich im Rahmen dieser Arbeit in das Tool integriert (siehe Anhang B.7). Vor der Synthese wird eine Liste mit allen Vorhanden Interaktionen[14] erstellt. Sobald ein Prozess zu einer Interaktion erstellt wird, d.h. sobald das MSD aktiviert wird, wird die Interaktion aus der Liste entfernt. Nach der Synthese werden die in der Liste verbleibenden Interaktionen in der Konsole ausgegeben.

6.2 Analyse und Korrekturen

Es hat sich gezeigt, dass es bei der Korrektur der Anforderungen sinnvoll ist, eine Anforderungsverletzung nach der andern zu bearbeiten, da ein Fehler in den Anforderungen i.d.R. zu Verletzungen an mehreren Stellen im Zustandsautomaten führt, da es Attribute im Objektsystem und MSDs gibt die keinen Einfluss auf die Verletzung haben jedoch den Zustandsraum vergrößern. Die Anforderungsverletzungen teile ich im Folgenden in verschiedenen Kategorien ein, um darauf basierend verschiedene Lösungsstrategien zu entwickeln. Sollte eine Verletzung nicht in das Schema passen ist die Kreativität des Entwicklers gefragt.

Die Arten von Anforderungsverletzungen schließen sich nicht aus. Welche Kategorie im konkreten Fall das Problem herbeiführt muss der Entwickler entscheiden. In Anhang A ist ein mögliches Vorgehen zum Auflösen von Fehlern basierend auf den folgenden Kategorien dargestellt.

6.2.1 Fehlerhafte Modellierung

Bei der Modellierung können aus verschiedenen Gründen Fehler entstehen. Hierzu zählen nicht eindeutig formulierte textuelle Anforderungen, falsches Verständnis der Anforderungen vom modellierenden Entwickler oder schlicht Flüchtigkeitsfehler. Dabei treten die folgenden Fehler leicht auf.

[14] Jedes MSD beinhaltet genau eine Interaktion die den kompletten Inhalt des MSDs beinhaltet.

Bei der Modellierung von Antiszenarien kann leicht eine **Abbruchbedingung vergessen oder falsch formuliert** werden. Dies zeigt sich meist darin, dass dieses Antiszenario bei der Synthese zu einer Anforderungsverletzung führt, obwohl es bei dieser Stimulationsfolge nicht erfolgen sollte.

Wenn eine Systemnachricht nicht ausgelöst wird, obwohl es eine Anforderung gibt, die diese auslösen soll, kann es daran liegen, dass diese Nachricht mit der Eigenschaft **monitored anstatt executed** versehen ist.

Wenn ein MSD auf Grund einer Reaktivierung verletzt wird, kann dies an einer **fehlenden verbotenen Nachricht** liegen, die das Diagramm durch eine cold-Violation abbrechen würde. Dies passiert, wenn das Diagramm vor einer hot Bedingung wartet.

Bei nicht eindeutig formulierten Anforderungen passiert es leicht, dass die **Reihenfolge von Aktionen in mehreren MSDs widersprüchlich** ist. Z.b., wenn ein Diagramm fordert, das a() vor b() eine anders aber b() vor a() fordert und beide durch einen gleichen Auslöser aktiviert sind. Wenn dieser Fehler auftritt, ist jedoch bereits die Qualität der zu Grunde liegenden Anforderung in Frage zu stellen.

6.2.2 Untereinschränkung des Environments

Durch assumption-MSDs wird das Environment eingeschränkt. Es kann sein, dass es Environment Events gibt, die nur in bestimmten Reihenfolgen vorkommen können. Dabei sind zwei Nachrichten, die nur abwechseln auftreten können, der häufigste Fall. Hierzu zählt auch das in Abbildung 5-17 dargestellt Beispiel. Ohne diese Diagramme könnte der Stecker ausgesteckt werden, obwohl er nicht eingesteckt ist, was in einem anderen Diagramm zu einer Anforderungsverletzung führen kann. Dieser Mangel bei den Annahmen lässt sich sehr gut im Wartezustandsdiagramm erkennen. Dabei muss der Entwickler den Weg vom Ausgangszustand bis zur angeblichen Anforderungsverletzung verfolgen, ob alle Environment-Events in dieser Reihenfolge möglich sind. Im Graphen der Counter-Strategie oder im Play-Out gegen die Counter-Strategie lassen sich solche Mängel ebenfalls einfach erkennen, wenn die Anzahl der Systemnachrichten zwischen den Wartezuständen überschaubar bleibt.

Um diesen Fehler zu beheben, muss ermittelt werden, welche Annahme fehlt. Diese Annahme sollte dann zunächst in der Anforderungsliste ergänzt werden. Meist handelt es sich dabei um Umweltannahmen, die durch den „gesunden Menschenverstand“ selbstverständlich sind, jedoch nicht weg gelassen werden dürfen, um eine konsistente Anforderungsliste zu erhalten. Es muss dabei darauf geachtet werden, dass das Verhalten, das durch die neue Annahme ausgeschlossen wird, tatsächlich nicht möglich ist. Wenn das Verhalten möglich ist, auch wenn es nicht dem nor-

malen Gebrauch entspricht, darf es nicht durch eine neue Annahme ausgeschlossen werden und muss durch zusätzliche Anforderungen vom System behandelt werden.

6.2.3 Fehlende Anforderung führt zu Antiszenario

Immer wenn gefordert wird, dass ein bestimmtes Verhalten nicht auftritt, wird dies durch ein Antiszenario dargestellt. Der Fall, dass ein Antiszenario aufgrund einer fehlenden Anforderung stattfindet, kann dann vorliegen, wenn das verletzte MSD ein Antiszenario ist. Wenn ein Antiszenario als verletztes MSD angezeigt wird, muss zunächst geprüft werden, ob das Antiszenario und die anderen Anforderungen, die in diesem und den vorhergehenden Zuständen aktiviert sind, korrekt modelliert wurden. Häufig auftretende Fehler, wie eine falsch formulierte Abbruchbedingung, sind in Kapitel 6.2.1 dargestellt. Anschließend ist der Wartezustandsgraph zu untersuchen, ob die zu dieser Verletzung führende Folge von Stimulationen möglich ist. Wenn das Environment eine nicht mögliche Folge auslöst, liegt der in Kapitel 6.2.2 beschriebene Fall vor. Wenn ein Antiszenario die violation-Bedingung erreicht, obwohl eine zulässige Folge von Umwelteinflüssen erfolgt ist, fehlt i.d.R. eine Anforderung, die eine Nachricht auslöst, die das Antiszenario abbricht oder die Nachricht ist in einer Anforderung monitored, obwohl sie executed sein sollte (siehe Kapitel 6.2.1). Sollte die Anforderung fehlen muss sie ergänzt werden bzw. korrigiert werden, wenn sie fehlerhaft modelliert wurde.

6.2.4 Widersprüchliche Anforderungen

Wenn eine Anforderung ein Event fordert, das in einem anderen aktiven MSD verboten ist. Dies kann zum einen daran liegen, dass dieses Event als verbotene Nachricht mit der Temperatur hot in einem aktivierten MSD enthalten ist. Zum anderen kann dieses Event in einem aktivierten MSD vorkommen, dessen Cut in diesem Moment nicht vor dieser Nachricht, hot und executed ist. Dies kann z.B. vorkommen, wenn zwei verschiedene Anforderungen die gleichzeitig aktiviert sein können, je eine Reihe von Reaktionen fordern, die teilweise identisch sind. Dies führt dann zu einem Problem, wenn die Reihenfolge der Reaktionsnachrichten sich widerspricht. Dies liegt entweder daran, dass die Reihenfolge in den Anforderungen nicht festgelegt ist und bei der Modellierung unterschiedlich festgelegt wurde oder dass sich die Anforderungen tatsächlich widersprechen.

Die Reihenfolge ist in der Anforderungsliste festzulegen. Wenn diese geschehen ist, ist die Modellierung entsprechend zu korrigieren.

6.2.5 Fehlende Anforderung führt zu nicht verwendeten Anforderungen

Wenn MSDs bei einer angeblich erfolgreichen Synthese nicht aktiviert werden, liegt das daran, dass das Event der ersten Nachricht nie ausgelöst wird. Unter der Annahme, dass der Ersteller der Anforderungen keine vom System unabhängigen Annahmen formuliert hat, die in die Modellie-

rung eingeflossen sind, heißt das, das Anforderungen fehlen, die durch eine Nachricht in ihre Aktion, die nicht aktivierte Anforderung auslösen. Welche Anforderung genau fehlt lässt sich mit dem interaktiven Play-Out herausfinden. Der Entwickler muss hierfür versuchen an die Stelle zu kommen, an der die nicht verwendete Anforderung sein sollte. An dem Punkt, an dem das Play-Out den Weg zu dieser Anforderung nicht zulässt liegt einer der folgenden Fehler vor. Zum einen kann eine Anforderung fehlen, die eine executed Nachricht enthält, die die nicht aktivierte Anforderung aktiviert. Zum anderen kann diese Nachricht in einer Anforderung existieren, jedoch fälschlicher Weise monitored statt executed sein. Wenn die, das MSD aktivierende, Nachricht ein Environment-Event ist, dann liegt der im Kapitel 6.2.6 beschrieben Fall, eines zu sehr eingeschränkten Environments, vor.

Entsprechend dem erwarteten Verhalten ist die fehlende bzw. sind die fehlenden Anforderungen zu ergänzen.

6.2.6 Übereinschränkung des Environments

Dieser Mangel am Modell ist relativ schwierig zu erkennen, wenn nicht der Fall voeliegt, den ich in Kapitel 6.2.5 beschreibe. Durch Übereinschränkung der Umwelt kann die Synthese erfolgreich verlaufen, obwohl noch weitere Fehler vorhanden sind. Das liegt daran, dass Verstöße gegen die requirement-MSDs zulässig sind, wenn gleichzeitig ein assumption-MSD verletzt wird, da dann davon ausgegangen wird, dass diese Stimulation gar nicht möglich ist. Um diese Übereinschränkung zu verhindern müssen die assumption-MSDs auf Plausibilität überprüft werden. Der synthetisierte Graph kann an dieser Stelle nicht unterstützen.

6.2.7 Reaktivierung von Diagramm

Wenn die Transition, die die Anforderungsverletzung herbeiführt, dasselbe Event darstellt, wie die erste Nachricht eines bereits aktivierten MSDs führt dies zu einer Violation. Dies ist zu erkennen, wenn das Event, das die Verletzung herbeiführt, das erste Event des verletzten MSDs ist und das MSD bereits im vorhergehenden Zustand aktiviert ist. Beides wird in dem Zustand, in dem die Verletzung auftritt angezeigt. Dies geschieht, wenn der Auslöser aus mehreren Nachrichten (monitored) besteht, die nicht vollständig abgearbeitet werden und dann mit diesen wieder von vorne begonnen wird.

Dieser Verletzung kann behoben werden, indem die Nachrichten, die zu einer Reaktivierung führen, dem Diagramm als cold verbotene Nachrichten hinzugefügt werden. Dann ist jedoch zunächst zu prüfen, ob das erneute Auftreten der Auslösenden Nachrichten überhaupt zulässig ist. Andernfalls ist zu untersuchen, warum nicht alle Nachrichten des Auslösers ausgeführt werden.

6.2.8 Fehlende Annahme führt zu Anforderungsverletzung

Hierbei kann es sich um eine Untereinschränkung des Environments (siehe Kapitel 6.2.2) handeln, die oft zu einer Reaktivierung von Diagrammen führt (siehe Kapitel 6.2.7). Fehlende Systemannahmen äußern sich darin, dass z.B. ein Attribut nicht gesetzt wird, oder eine Antwort von einer Systemkomponente nicht ausgelöst wird. Der daraus resultierenden Fehler führt zu einem Durchlauf eines Antiszenarios und ist damit analog zum Fehler in Kapitel 6.2.3 oder führt zur Nichtaktivierung eines MSDs und ist somit analog zum Mangel in Kapitel 6.2.5.

7 Fazit und Ausblick

In diesem Kapitel stelle ich meine Erfahrungen mit der Darstellung und Analyse von Anforderungen durch MSDs mit Hilfe von ScenarioTools zusammen und bewertet diese. Zudem gebe ich einen Ausblick auf Ansätze, die in der Entwicklung von ScenarioTools vorhanden sind oder nahe liegen, die sehr wahrscheinlich einen weiteren Nutzen für die Verwendung in der Automobilindustrie mit sich bringen.

7.1 Useability

Für die industrielle Verwendung ist die Verwendbarkeit oder auch *Useability* ein entscheidendes Kriterium, da durch eine schlechte Useablitiy mit einem erhöhten Zeitaufwand zu rechnen ist bzw. die Bearbeitungszeit unkalkulierbar wird. Beide Erscheinungen produzieren Kosten die durch die Verwendung dieser Methode eingespart werden sollen. Dabei gilt für den Nutzen im Unternehmen, dass die Methode nur so gut sein kann wie die Tools, die man für die Umsetzung benötigt.

7.1.1 Papyrus

ScenarioTools basiert auf dem UML-Editor Papyrus. Dieser ist ausreichend übersichtlich gestaltet. Durch die Papyrus-Erweiterung von ScenarioTools sollten nicht verwendete Sprachelemente ausgeblendet werden. Hierdurch würde die Modellierung von MSDs etwas erleichtert.

Der *Model Explorer*[15] von Papyrus muss bei der Bearbeitung der MSDs immer geöffnet sein und kontrolliert werden, da dessen Inhalt dem Inhalt der UML-Datei entspricht, die von ScenarioTools zur Simulation und Synthese verwendet wird. Die Beziehungen zwischen den Elementen können sich zwischen der grafischen Darstellung und dem Inhalt des Model Explorers unterscheiden. Jede Nachricht bekommt als zusätzliche Elemente ein Sendeevent und ein Empfangsevent. Diese werden mit den Lebenslinien verknüpft. Für die Reihenfolge der Nachrichten in einem Diagramm sind die Positionen der Sende- und Empfangsevents in der uml-Datei bzw. im Model Explorer entscheidend. Da bei MSDs davon ausgegangen wird, dass die Nachrichten eine unendlich kurze Dauer haben müssen das Sendeevent und das Empfangsevent direkt hintereinander auftreten. Dies wird im Normalfall durch einen waagrechten Pfeil dargestellt. In einigen Fällen kann es vorkommen, dass im Model Explorer bei mehreren Nachrichten das Empfangsevent der letzten Nachricht vor dem Sendeevent der ersten Nachricht positioniert wird. Dies hat zur Folge, dass das MSD nie aktiviert wird, weil die erste Botschaft durch ScenarioTools nicht mehr richtig erkannt werden kann oder erwartet wird, dass das erste Event ein Sendeevent ist und dadurch in Scenari-

[15] Model Explorer: ein Eclipse-View von Papyrus, in dem die UML-Struktur als Baum dargestellt wird.

oTools eine Liste, die das Sendeevent enthalten soll, leer ist und in Folge dessen eine NullPointer-Exception[16] ausgelöst wird. Im grafischen Sequenzdiagramm ist dieser Mangel nicht zu erkennen.

Zudem müssen, um ein korrektes Modell zu erhalten, bei kombinierten Fragmenten und Lebenslinien die Verknüpfungen *CoveredBy bzw.* bei den Events die Verknüpfung *covered* in den Einstellungen des Elements geprüft werden. Wenn diese fehlen, kann ScenarioTools die entsprechenden Verbindungen nicht erkennen. Die Folge sind auch hier leere Listen, die zu NullPointer-Exceptions führen.

Durch das notwendige Prüfen der Abhängigkeiten zwischen den Modellierungselementen und dem unvorhersehbaren Verhalten von Papyrus ist der industrielle Einsatz bereits durch Verwendung dieser Papyrus-Version nicht denkbar. Dennoch bewerte ich im Folgenden auch die anderen verwendeten Teile der Methode.

7.1.2 ScenarioTools

Die zusätzlichen Eigenschaften der Elemente, wie die Temperatur und Ausführungsart von Nachrichten oder die Temperatur von Bedingungen, müssen im „Properties“-View unter „Profile“ eingestellt werden. Diese Einstellmöglichkeit ist etwas versteckt, jedoch für den mit ScenarioTools vertrauten Entwickler schnell zu erreichen. Dies könnte durch einen zusätzlichen „ScenarioTools“-View für Papyrus oder durch zusätzlich Einträge im Kontextmenü der Modellierungselemente etwas verbessert werden. Dies würde auch die Erstelldauer eines MSDs (siehe Kapitel 7.3) etwas reduzieren.

Durch die in Kapitel 7.1.1 beschriebenen Effekte von Papyrus und durch Modellierungsfehler kann es vorkommen, dass das resultierende UML-Modell nicht konsistent ist. D.h. UML-Elemente fehlen oder an einer falschen Stelle stehen. Dies führt im Play-Out und bei der Synthese zu unverständlichen Fehlermeldungen. In den meisten Fällen handelt es sich hierbei um eine NullPointer-Exception oder eine Assertion-Exception[17]. Der Fehler ist zunächst unbekannt. Durch eine erneute Ausführung und das Setzen von Wartepunkten im Quelltext von ScenarioTools, lässt sich oft herausfinden, in welchem Diagramm oder an welchem Objekt der Fehler liegt. Diese Informationen müssten über Fehlermeldungen dem Benutzer angezeigt werden. Eventuell ist es sinnvoll, vor dem eigentlichen Play-Out bzw. der Synthese eine Konsistenzprüfung des UML-Modells durchzuführen.

[16] Fehler in einem Programm, bei dem z.B. auf ein Element einer leeren Liste zugegriffen wird. Dieser führt i.d.R. zum Abbruch des Programms.
[17] Fehler in einem Programm, bei dem ein Ausdruck explizit geprüft wird und einen nicht zulässigen Wert hat.

Der Dialog zur Erstellung der TGG Interpreter Konfiguration muss um alle vorhandenen Optionen erweitert werden. Die Option „DynamicObjectSystem“, die ich in dieser Arbeit verwendet habe, muss durch eine Check-Box in diesem Dialog ausgewählt werden können. Ansonsten ist dieser Dialog verständlich und übersichtlich gestaltet.

Die Funktionen von ScenarioTools sind über Kontextmenüs der Dateien im Projektexplorer leicht zu erreichen. Das Kontextmenü der Scenariorunconfiguration-Datei könnte um einen Schaltfläche für das interaktive Play-Out erweitert werden. Somit könnte die Konfiguration über die „Debug Configuration“ von Eclipse umgangen werden. In dieser muss festgelegt werden, welche Konfiguration ausgeführt wird. Eine Auswahl über das Kontextmenü wäre für den Benutzer flexibler und weniger versteckt.

Die zur Modellierung in ScenarioTools verwendete Paketstruktur lässt ein Testen der einzelnen Ebenen sehr leicht zu. So können z.B. zunächst nur die HLR getestet werden, bevor die darunter liegenden Subsysteme und Unterfunktionen spezifiziert werden. Durch den einfachen Merge-Operator lässt sich das zuvor getestet HLR-Paket sehr einfach in die anderen Pakete integrieren. Dies ist vor allem dann von Vorteil, wenn durch verschiedene scenariorunconfiguration-Dateien verschiedene Ebenen bzw. Teile der Anforderungen geprüft werden.

7.1.3 Modellierung durch MSDs

Ein MSD stellt eine einzelne Anforderung sehr übersichtlich da. Es lässt sich sehr einfach erkennen, welche Komponenten von dieser Anforderung betroffen sind. Zudem wird die Anforderung in einem Szenario einfach dargestellt.

Grundsätzlich ist das Modellieren von MSDs leicht zu erlernen, da die Regeln einfach sind und das Verhalten des spezifizierten Systems sehr intuitiv abgebildet wird.

Dennoch wäre eine zusammengefasst Dokumentation über alle in MSDs zur Verfügung stehenden Sprachelemente und ein Tutorial, das an die Verwendung dieser Diagramme heran führt wünschenswert. Die bisher vorhandene Dokumentation ist verteilt auf mehrere Bücher und Papiere aus unterschiedlichen Quellen.

7.2 Grenzen und notwendige Erweiterungen

In diesem Abschnitt möchte ich die Grenzen, der in dieser Arbeit ausgearbeiteten Methode und die für die Automobilsoftwareentwicklung notwendigen Erweiterungen darstellen.

7.2.1 Anforderungen

Durch Sequenzdiagramme und damit durch MSDs können nur Interaktionen zwischen Objekten dargestellt werden. Mit diesen Interaktionen kann das funktionale Verhalten der Objekte dargestellt werden. Qualitätsanforderungen wie z.B. die Lebensdauer eines Bauteils oder Softwarequalitätsanforderungen können nicht dargestellt werden. Somit können nur funktionale Anforderungen modelliert und damit auch nur diese geprüft werden. Die funktionalen Anforderungen des Beispielprojekts konnte ich fast vollständig als MSDs darstellen. Jene Anforderungen, die ich nicht umsetzen konnte, waren schlecht verständlich bzw. nicht eindeutig. Wenn die Anforderungen mit dem „Wenn…, dann…"-Template formuliert sind, können sie mit dem Schema, das ich in Kapitel 5.3 vorgestellt habe, sehr einfach umgesetzt werden.

Da auf einer sehr detaillierten Softwareanforderungsebene oft die Sichtweise gewechselt wird und die Anforderungen als Zustandsautomaten formuliert werden, müssen die in ScenarioTools bereit vorhanden *Modal State Structures (MSS)* verwendet werden. Aufgrund der noch nicht implementierten Unterstützung von dynamischen Objektsystemen konnte ich MSSs nicht in dieser Arbeit verwenden. Diese stellen Abläufe als Zustandsautomat dar. Dabei wird den Zuständen, ähnlich wie den Cuts der MSDs, eine Temperatur und Ausführungsart zugeordnet [16]. Jede den Zustand verlassende Nachricht ist dabei zulässig, alle anderen Nachrichten, die ebenfalls im MSS enthalten ist nicht. Wenn eine nicht zulässige Nachricht ausgeführt wird, wird je nach Temperatur des aktuellen Zustands eine safety-Violation oder eine cold-Violation ausgelöst. Diese MSS sollten mit der vollen Funktionalität in ScenarioTools zur Verfügung stehen.

Um wiederkehrende Abläufe darstellen zu können ist ein Schleifenkonstrukt, wie ich in Kapitel 5.6.1 notwendig. Dieser Ansatz wird bereits für ScenarioTools entwickelt, ist aber noch nicht im Hauptstrang integriert. Aus diesem Grund habe ich dieses Element in dieser Arbeit ebenfalls nicht testen können.

Da ein System oft auf mehrere Steuergeräte verteilt ist, sich auch im Automobilbereich Multicore-Prozessoren durchsetzen und auf den Steuergeräten Multitasking-Echtzeit-Betriebssysteme wie OSEK-OS [17] oder AUTOSAR-OS verwendet werden, müssen auch parallele Strukturen dargestellt werden können. Hierfür kann die Unterstützung des kombinierten Fragments mit dem Interaktionsoperator „par" umgesetzt werden, wie ich sie in Kapitel 5.6.2 bereits konzipiert habe.

7.2.2 Anforderungsformulierungen

Da es für die Formulierungen für Anforderungen keine festgelegten Regeln gibt, können funktionale Anforderungen auch als Prosatext vorliegen.

Beispiel: *Der Ausgang der Schnittstelle darf erst dann mit der HV-Batteriespannung verbunden werden, nachdem das Nebenaggregat und die Schnittstelle auf Isolationsfehler, Schützkleber, durchgebrannte Sicherungen, gesteckte und verriegelte Stecker, korrekte Kommunikation und korrekte Funktion geprüft wurden. Dabei ist bei einem Fehlerfund dieser in den Fehlerspeicher einzutragen. Zudem muss der Zustand „Fehlerfrei" bzw. der gefundene Fehler dem Benutzer signalisiert werden.*

Bei einer solchen Formulierung ist das darstellen als MSD nur dann möglich, wenn man das System so gut kennt, dass man diese Anforderungen in mehrere Anforderungen zerlegen kann, ohne dass sich die Funktion ändert. Anforderungen, wie in diesem Beispiel, sind zwar selten geworden, können aber vereinzelt vorkommen. Nicht atomare Anforderungen kommen jedoch relativ häufig vor und bringen ähnliche Probleme mit sich, wie das hier aufgeführt Beispiel.

7.3 Vorteile der Methode

Automatisierte Anforderungsanalysen haben für ein Unternehmen viele Vorteile. Die wichtigsten sind eine schnelle Analyse einer sehr großen Menge von Anforderungen und ein zuverlässigeres Ergebnis, als bei einer manuellen Analyse.

Die schnelle Analyse ist bei der Methode, die ich in dieser Arbeit erarbeitet habe, auch gegeben, jedoch ist die Modellierung zeitaufwändig (siehe Kapitel 7.4). Ein besonders hoher Zuwachs an Geschwindigkeit ist deshalb vor allem dann zu erwarten, wenn die meisten Anforderungen bereits modelliert sind und nur einzelne Anforderungen verändert oder hinzugefügt werden. Wie viel Zeit die Methode einspart müsste in einem Benchmark untersucht werden, in dem zwei Entwicklungsteams das gleiche Produkt entwickelt, wobei das eine den herkömmlichen Prozess zur Anforderungsanalyse und das andere die von ScenarioTools unterstütze Methode verwendet.

Das Ergebnis ist in jedem Fall zuverlässiger, da es nicht von der begrenzte Aufnahmefähigkeit der Anforderungsprüfer abhängt. Bei Entwicklern hat ein automatisiert geprüftes Ergebnis zudem eine höheren Akzeptanz und bietet somit eine bessere Argumentationsgrundlage wenn die Anforderungsliste bzw. das Lastenheft nachgebessert werden muss. Zudem werden die Ersteller der Anforderungen gezwungen das Verhalten der Komponenten und der Umwelt zu beschreiben. Dies wird oft vernachlässigt, da der Ersteller der Anforderungen gewisse Sachverhalte als Grundlage voraussetzt. So weiß ein Hochvolt-Sicherheits-Spezialist, was ein Schützkleber ist, für einen Softwareentwickler muss jedoch definiert werden, wie sich das Schütz verhält, wenn dieser Fehlerfall vorliegt.

Um Anforderungen als MSD darstellen zu können, müssen diese eindeutig formuliert sein. Somit werden nicht eindeutig formulierte Anforderungen bereits bei der Anforderungsprüfung erkannt. Mit der bisher üblichen Methode werden diese oft erst bei der Umsetzung entdeckt.

7.4 Aufwand

Der Modellierungsaufwand um Anforderungen als MSD darzustellen ist relativ hoch. Die Erstellung eines Diagramms mit zwei Objekten und zwei Nachrichten, von denen eine hot und executed ist und die andere cold und monitored ist dauert über 1,5 Minuten. Bei ca. 1000 Anforderungen ergibt das eine Dauer von 25 Stunden oder 3 Arbeitstagen. Wenn zudem kombinierte Fragment notwendig sind erhöht sich die Zeit für ein Diagramm erheblich. Durch eine verbesserte Useability könnte diese Zeit etwas verbessert werden. Ein weiterer Ansatzpunkt zur schnelleren Modellierung sind textuelle MSDs (siehe Kapitel 7.5). Um den Aufwand möglichst stark zu reduzieren müssten die Anforderungen bereits als MSDs oder als textuelle MSDs formuliert werden. Dies durchzusetzen ist jedoch eine erhebliche Herausforderung.

7.5 Textuelle MSDs

Florian König hat in seiner Bachelorarbeit eine Sprache zur textuelle Darstellung von MSDs entwickelt [18]. Eine solche Sprache kann die Modellierung eines MSDs und damit die Modellierung aller Anforderungen erheblich beschleunigen. Dies gilt vor allem dann, wenn für die Erstellung der MSDs ein Editor mit Autovervollständigung zur Verfügung steht.

Zudem lassen sich textuelle Darstellungen leichter in die Anforderungslisten und Anforderungsdatenbanken integrieren, als grafische Darstellungen. Wenn textuelle MSDs in einer Anforderungsdatenbank integriert sind, können diese automatisiert extrahiert und analysiert werden. Somit würde sich eine deutlich bessere Integrierbarkeit in den Entwicklungsprozess ergeben, als mit grafischen MSDs, die in einer UML-Struktur in einer UML-Datei abgelegt sind.

7.6 Testfallableitung

Für die vorgeschriebenen Tests müssen im Softwareentwicklungsprozess Testfälle generiert werden. Diese spezifizieren Folgen von Stimulationen, mit denen die Software im SIL-Test oder das gesamte System im HIL-Test angeregt wird. Bei der Erstellung der Testfälle ist, wie bei den Szenarien zur Anforderungsanalyse, die Kreativität der Entwickler gefragt. Auch hier ist es wahrscheinlich, dass nicht alle möglichen Testfälle geprüft werden. Hierfür ist es interessant, ob und wie aus den, als MSDs dargestellten, Anforderungen automatisiert Testfälle generiert werden können. Wenn diese Testfälle in die Testumgebungen importiert werden können, kann hiermit ebenfalls eine erhebliche Effizienz- und Qualitätssteigerung erzielt werden.

Literaturverzeichnis

[1] J. M. u. M. R. Günther Schuh, „Gestaltung von Produktentwicklungsprozessen," in *Innovationsmanagement, Handbuch Produktion und Management 3*, Berlin Heidelberg, Springer-Verlag , 2012, pp. 161 - 247.

[2] M. Müller, K. Hörmann, L. Dittmann und J. Zimmer, Automotive SPICE in der Praxi, Heidelberg: dpunkt.verlag GmbH, 2007.

[3] J. Greenyer, Scenario-based Design of Mechatronic Systems, Paderborn: Universität Paderborn, 2011.

[4] D. Harel und R. Marelly, Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine, Springer-Verlag, 2003.

[5] IAV GmbH, „Die 400-Volt-Steckdose fürs Fahrzeug," *automotion - 2/2014 Nutzfahrzeuge,* pp. 36-37, Juni 2014.

[6] C. Ebert, Systematisches Requirements Engeneering, Heidelberg: dpunkt.verlag, 2012.

[7] IEEE Computer Society, „IEEE Standard Glossary of Software Engeneering Glosary," IEEE, 1990.

[8] Object Management Group, *ISO/IEC 19505-1: Information technology - Object Management Group Unified Modeling Language (OMG UML), Infrastructure,* ISO, 2012.

[9] Object Management Group, *ISO/IEC 19505-2: Information technology - Object Management Group Unified Modeling Language (OMG UML), Superstructure,* ISO, 2012.

[10] Object Management Group, *Object Constraint Language, v2.4,* OMG, 2014.

[11] D. Harel und S. Maoz, „Asset and Negate Revisited: Modal Semantics for UML Sequence Diagrams," *Software & System Modeling,* pp. 237-252, 2008.

[12] J. Greenyer, C. Brenner, M. Cordy, P. Heymans und E. Gressi, „Incrementally Synthesizing Controllers from Scenario-Based Product Line Specifications," in *ESEC/FSE'13*, St. Petersburg,

2013.

[13] J. Greenyer, „Modeling | ScenarioTools," ScenarioTools, 2014. [Online]. Available: http://scenariotools.org/modeling/. [Zugriff am 30 10 2014].

[14] ISO, *ISO 26262 - Straßenfahrzeuge - Funktionale Sicherheit,* 2011.

[15] F. Stacheki, „Concurrency in UML," 8 April 2014. [Online]. Available: http://www.omg.org/ocup-2/documents/concurrency_in_uml_version_2.6.pdf. [Zugriff am 2 02 2015].

[16] T. Gutjahr, „ScenarioTools Counter-Play-Out Simulation zur Analyse von unrealisierbaren szenariobasierten Spezifikationen," Gottfried Wilhelm Leibniz Universität Hannover, Hannover, 2014.

[17] K. Reif, Automobilelektronik, Eine Einführung für Ingenieure, Wiesbaden: Springer Fachmedien, 2014.

[18] F. König, „MSpec -- Eine Technik zur textuellen Modellierung und Simulation multimodaler szenariobasierter Spezifikationen," Gottfried Wilhelm Leibniz Universität hannover, Hannover, 2014.

Eigenständigkeitserklärung

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.

Gifhorn, den 31. März 2015

_____

Maximilian Haase

Anhang

A. Fehlerkorrektur

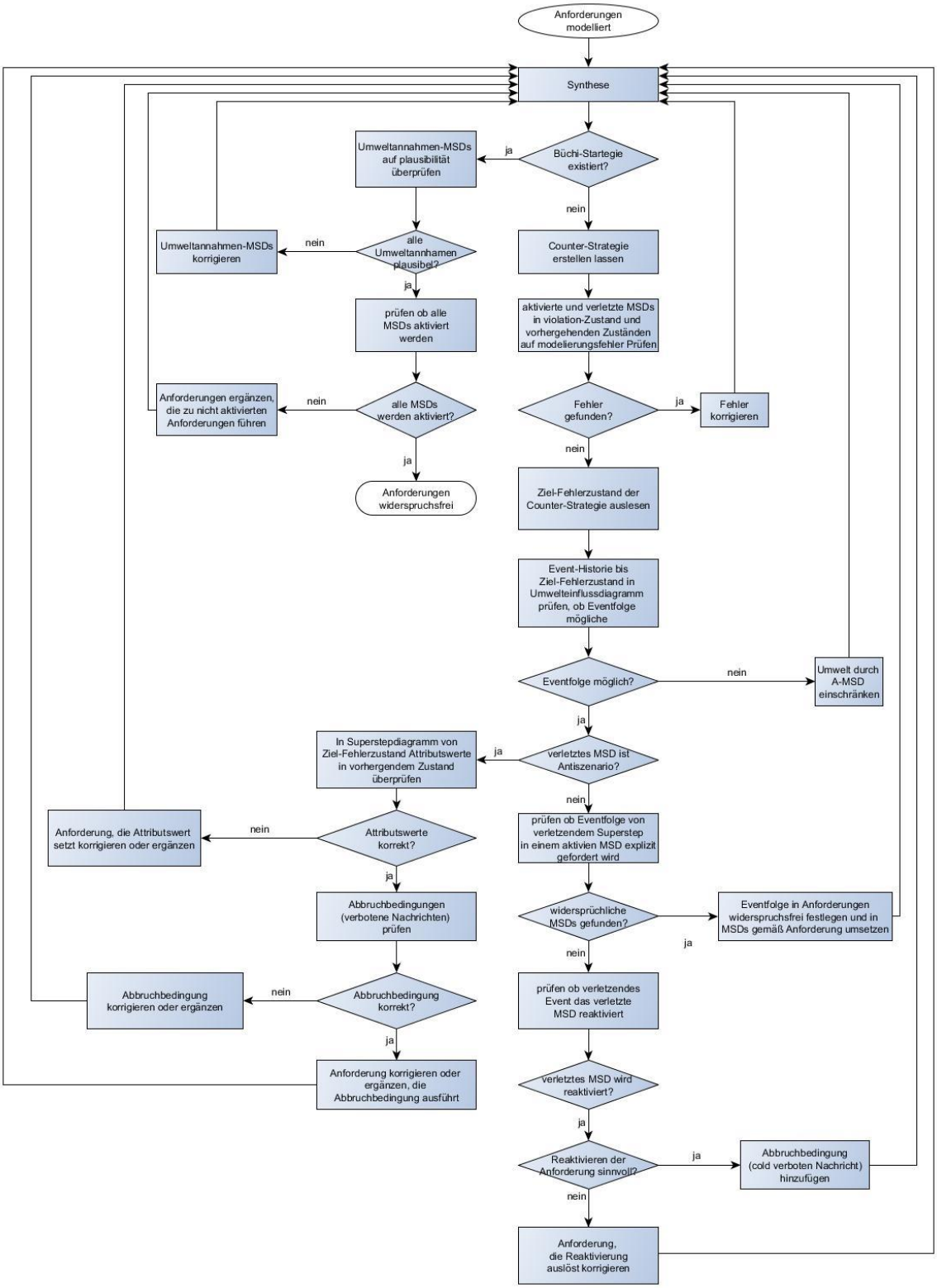

Abbildung A-1: Vorgehen zur Analyse und Korrektur von Fehlern

B. Codeerweiterungen

In diesem Abschnitt stelle ich die Erweiterungen und Korrekturen an ScenarioTools dar, die ich für die Verwendung zur Anforderunsanalyse mit dem, in dieser Arbeit beschriebenen, Vorgehen umgesetzt habe. Für die Methoden, die ich in dieser Arbeit entwickelt habe, habe ich diese Erweiterungen und Korrekturen als Bestandteil von ScenarioTools angenommen.

B.1. Korrigiertes Prüfen der Bedingungen

Es hat sich gezeigt, dass ein MSD, dessen Cut vor einer hot Bedingung steht, die zunächst *false* ergibt, an dieser Stelle für immer hängen bleibt. Das lag daran, dass die Bedingung nur einmal bei Erreichen der Bedingung ausgewertet wurde.

Die Methode *performStepOnActiveProcesses*[18] führt den Cut in allen aktiven MSDs, deren aktivierte Nachricht mit einem gegeben Event übereinstimmt, mit der Methode progressActiveMSD[18], fort. Des Weiteren behandelt diese Funktion Violations und aktiviert MSDs, deren erste Nachricht mit dem Event übereinstimmt. Zum Ende der Funktion ruft sie die Methode *performPostProcessingForStepOnActiveMSDs*[18] mit einer Liste aller MSDs auf, deren Cut sich geändert hat. Innerhalb der Methode *performPostProcessingForStepOnActiveMSDs* werden in der Methode *progressEnabledHiddenEvents*[19] u.a. Bedingungen ausgewertet und verarbeitet. Da *performPostProcessingForActiveMSDs* nur die MSDs verarbeitet, bei denen sich durch das aktuelle Event etwas geändert hat, werden Bedingungen, die bei der ersten Auswertung *false* ergeben haben, nicht mehr ausgewertet und somit bleibt der Cut an dieser Stelle hängen, auch wenn die Bedingung durch Seiteneffekte in anderen MSDs *true* wird.

Um dieses Fehlverhalten zu korrigieren, habe ich in der Methode *performStepOnActiveProcesses,* bevor *progressActiveMSD*[18] aufgerufen wird, einen Aufruf der Methode *progressEnabledHiddenEvents* eingefügt. Da die Methode *progressEnabledHiddenEvents* keinen Effekt hat, wenn keine Bedingung vorliegt, ist dies möglich.

Diese Änderung bewirkt, dass eine Bedingung, deren erste Auswertung *false* ergeben hat und die am Ende eines MSDs steht, dazu führt, dass das MSD nicht beendet wird, auch wenn der Cut über die Bedingung am Ende des MSDs angekommen ist. Dies liegt wiederum daran, dass das Beenden eines MSDs ebenfalls in der Methode *performPostProcessingForStepOnActiveMSDs* stattfindet. Auch hier wird das MSD nicht in die Liste, die dieser Methode übergeben wird eingetragen.

[18] Diese Methode ist aus der Klasse *MSDRuntimeStateImpl.*
[19] Diese Methode ist Teil der Klasse *ActiveMSDImpl.*

Hierfür prüfe ich in der Methode *progressEnabledHiddenEvents*, mit *terminalCutReached*[20], ob das Ende des MSDs erreicht ist. Wenn dies der Fall ist, wird ein entsprechender Wert zurückgegeben, sodass in der Methode *progressActiveMSD* das MSD mit der Methode *disposeState*[18] von der Liste der Aktiven MSDs entfernt wird.

B.2. Violation Bedingung

Für Antiszenarien ist es sinnvoll, eine Bedingung zu verwenden, die eine Violation auslöst. In ScenarioTools wurden Antiszenarien derart modelliert, dass eine Bedingung, die hot ist und den OCL-Ausdruck *false* als Wächter hat erreicht wurde. Dies führt zu eine liveness-Violation [3]. Wenn das betreffende Diagramm nun auch ein cold verbotene Nachricht beinhaltet, kann das Diagramm, trotz der Bedingung, vor der der Cut ewig stehen bleiben würde, beendet werden.

Um dennoch sicher eine Violation erzielen zu können, wenn der Cut eine bestimmte Stelle im Diagramm erreicht, habe ich in ScenarioTools violation-Bedingungen integriert. Diese sollen als Wächter nicht einen OCL-Ausdruck beinhalten, sondern die Zeichenkette „violation“. Da dies kein gültiger OCL-Ausdruck ist, muss eine solche Bedingung abgefangen werden, bevor die Auswertung eines OCL-Ausdrucks stattfindet. Hierfür habe ich in der Methode *progressCombinedFragment*[21] vor dem Aufruf von *isConditionTrue*[21] den Ausdruck des Wächters mit der Zeichenkette „violation“ verglichen. Wenn dieser Vergleich *true* ergibt, also wenn der Ausdruck gleich „violation“ ist, dann wird die Methode *progressCombinedFragment*[21] mit dem Rückgabewert *ActiveMSDProgress.SAFETY_VIOLATION* beendet. Somit wird zurückgegeben, dass bei der Verarbeitung der Bedingung und damit bei der Verarbeitung des MSDs an dieser Stelle eine Safety-Violation aufgetreten ist.

B.3. Zusätzliche Ausgabe des Objektsystems

Bei der Fehlersuche hat sich gezeigt, dass es sehr wichtig ist, im Zustandsgraphen die Wert der Objektattribute zu kennen, um entstehende Violations nachvollziehen zu können. Da dies in den Zuständen der Exportierten Zusstandsautomaten nicht mit angezeigt wird (siehe Abbildung B-2), habe ich diese Feature für die PDF-Exportfunktion implementiert.

Abbildung B-2: Ursprüngliche Darstellung eines Zustands in exportiertem Zustandsautomaten

[20] Aus der Klasse *ActiveMSDCutImpl*
[21] Teil der Klasse *ActiveMSDImpl*

Die Zustände des Zustandsgraphens, der bei der Synthese erstellt wird, enthalten die Werte der Attribute im Objektsystem. Aus diesem Zustandsgraphen wird eine Graphviz-Datei erstellt, aus der Grphviz dann wiederum eine PDF-Datei exportiert. In der Graphviz-Datei sind alle Zustände und die Transitionen aufgelistet. Dabei hat jedes Element auch einige Beschriftungen. Hierfür habe ich im Exporter die Methode *getObjectSystemState*[22] implementiert. Diese liest aus einem gegeben Zustand das Objektsystem aus, schreibt in eine Zeichenkette die Namen der Attribute und ihre Werte. Der Name des Objekts wird aus dem Attribut *name* ausgelesen und vor jedem anderen Attribut mit einem „." getrennt platziert. Nachdem alle Attribute eines Objekts in die Zeichenkette geschrieben sind, wird ein „
" in die Zeichenkette eingefügt, um Graphiz anzuweisen, an dieser Stelle einen Zeilenumbruch auszugeben. Somit stehen in jeder Zeile die Attribute eines Objekts (siehe Abbildung B-3). Diese Zeichenkette wird in der Methode *exportState*[22] in die Ausgabe integriert.

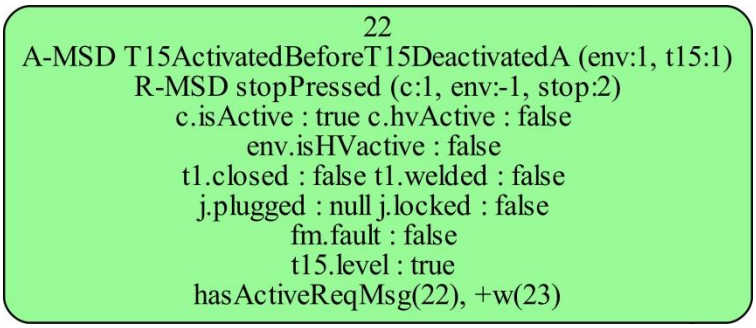

Abbildung B-3: Zustand aus exportiertem Zustandsautomaten mit Objektattributen

B.4. Zusätzliche Ausgabe der violated MSDs

Für das Vorgehen, das ich in Kapitel 6.2 beschrieben habe, ist es wichtig zu wissen, welches MSD verletzt wurde. Diese Information war bisher nicht im exportierten Graphen enthalten. Deshalb habe ich die Ausgabe, bei Zuständen bei denen eine Violation vorliegt, so erweitert, dass das verletzte Diagramm ausgegeben wird. Da diese Information im generierten Graphen nicht enthalten ist, habe ich bei der Synthese, an die betreffenden Zustände, eine *EAnnotation* angehängt, die den Namen des Diagramms enthält. EAnnotations sind Notizen, die an die meisten Objekte, die innerhalb von ScenarioTools verwendet werden, angehängt werden können. Die EAnnotations sind in einer Map[23] des Objekts angelegt.

Zur Umsetzung dieser Funktionalität habe ich in der Methode *handleViolation*[24] das Eintragen des MSD-Namens in die Notiz des synthetisierten Zustands hinzugefügt. Diese EAnnotations werden in den synthetisierten Zuständen abgelegt. Beim Export lassen sie sich somit wieder auslesen. Im

[22] Teil der Klasse *RuntimeStateGraphExporter*
[23] Menge, bei der die Werte (*value*) über einen *key* adressiert werden können
[24] Aus der Klasse *MSDRuntimeStateImpl*

Exporter habe ich ähnlich wie in B.3 beschrieben die Ausgabe des Namens hinzugefügt (siehe Abbildung B-4). Dabei wird einem verletzten requirement-MSD „v R-MSD:“ und einem assumption-MSD „v AMSD-:“ vorangestellt. Wenn die Notiz nicht vorhanden ist, also in diesem Zustand keine Verletzung vorliegt, wird diese zusätzliche Zeile nicht ausgegeben.

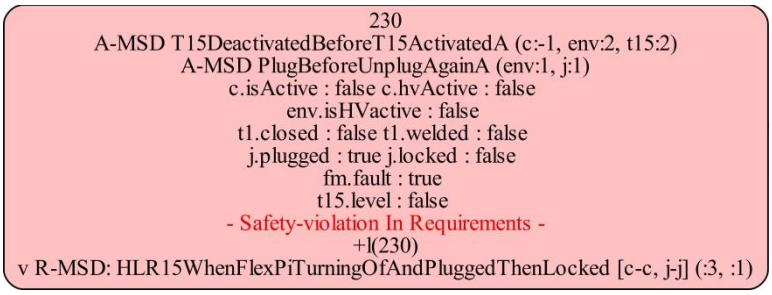

Abbildung B-4: violated MSD in exportiertem Zustand

B.5. Umwelteinflussdiagramm und Superstepdiagramme

Es hat sich gezeigt, dass bereits bei der Synthese der HLR (ca. 40 Anforderungen) Zustandsautomaten von mehreren hundert Zuständen entstehen. Bei dieser Anzahl von Zuständen werden die Zustandsautomaten zum einen so unübersichtlich, dass eine Analyse nur schwer möglich ist, und zum anderen kann Graphiz derart große Graphen nicht als PDF-Dateie exportieren.

Um dies übersichtlicher zu gestalten, habe ich einen weiteren Exporter implementiert, der den gesamten Graphen in einen Übergeordneten und viele untergeordnete Graphen zerlegt. In den meisten Fällen ist zunächst die Folge von Environment-Events Interessant, um die Verletzung zu analysieren. Daher liegt es nahe, in einem Graphen nur die Wartezustände und die Environment-Events darzustellen, die eine Kette von Ereignissen auslösen, um zu den nächsten Wartezuständen zu gelangen. Hierfür nehme ich an, dass sich ein System bei einer Folge von Umweltereignissen immer gleich verhält. Das bedeutet, dass ein System aus einem Wartezustand, in dem es durch ein bestimmtes Environment-Event angeregt wird, über einige System-Events, immer zum selben Wartezustand gelangt. Dabei kann die Folge von System-Events mit den System-Events eines anderen Supersteps, wie in Abbildung B-5 dargestellt, zusammenlaufen. Die Wartezustände sind hier blau und die Environment-Events gestrichelt dargestellt.

Zudem ist es durch parallel ablaufende Vorgänge möglich, dass sich der Pfad in einem Superstep aufspaltet, jedoch muss dieser dann auch wieder zusammengeführt werden. Eine Aufspaltung zu zwei verschiedenen Wartezuständen kann nicht vorkommen, da das System auf eine Folge von Environment-Events immer gleich reagieren soll und durch so eine Aufspaltung dies nicht mehr gegeben wäre.

Somit habe ich mich dafür entschieden neben dem Umwelteinflussdiagramm, das allen Environment-Events und die Wartezustände enthält, die Systemzustände in Superstepgraphen darzustellen, die alle Zustände der Supersteps enthalten, die zu einem Wartezustand führen.

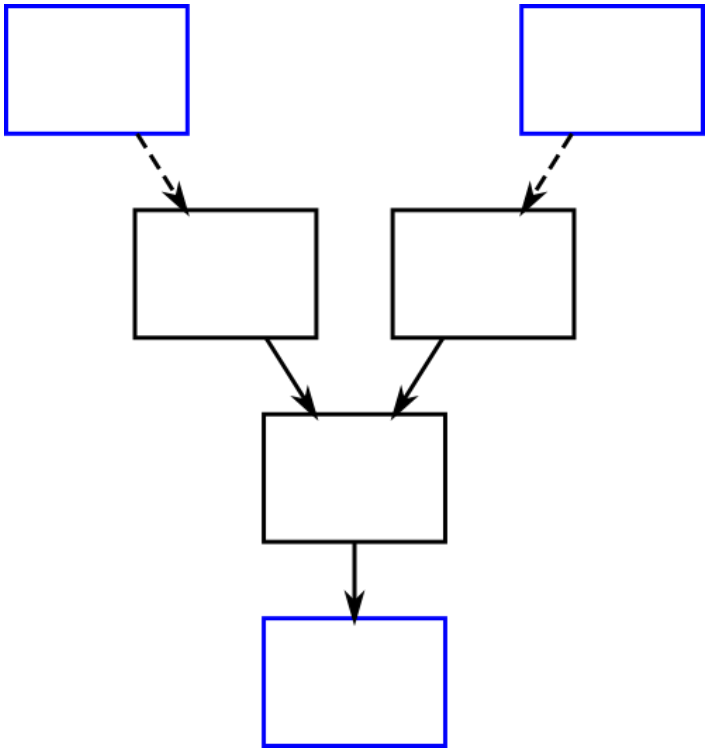

Abbildung B-5: zusammenlaufende Supersteps

Zur Implementierung dieses Exporters habe ich den anderen Exporter kopiert und die Methode *exportState* derart modifiziert, dass Zustände, die keine Wartezustände sind zunächst übersprungen werden und die Transitionen der Environment-Events zum nächsten Wartezustand umgeleitet werden. Das Vorgehen stelle ich in folgendem Beispiel dar:

Ausgehend von Zustand 1 (siehe Abbildung B-6) wird zunächst das Ziel von Transition a betrachtet. Da Zustand 3 kein Wartezustand ist, wird dann der Zielzustand der ausgehenden Transition des Zustandes betrachtet, also der Zielzustand der Transition c. Dieses Vorgehen wird so lange wiederholt, bis ein Wartezustand erreicht ist. Wenn die Transition eine Selbsttransition ist, wird eine andere Transition ausgewählt. Wenn es keine andere Transition gibt, die ggf. über weitere Transition zu einem Wartezustand führt, wird der Zustand als Wartezustand betrachtet und ebenfalls in den Wartezustandsgraphen aufgenommen.

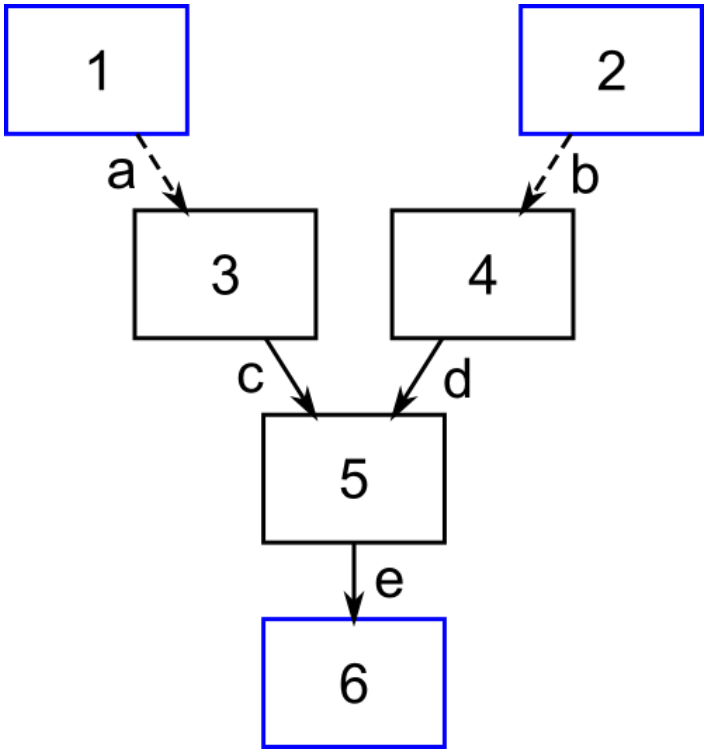

Abbildung B-6: Superstepgraph

Für jeden Wartezustand (z.B. Zustand 6 in Abbildung B-6), wird automatisch ein eigener Exporter *subGraphExporter* aufgerufen. Es sollen alle Transitionen und Zustände, die von den vorhergehenden Wartezuständen (Zustände 1 und 2) zu dem betrachteten Wartezustand führen dargestellt werden. Hierfür ermittelt der Exporter in der Methode alle Zustände die dargestellt werden sollen. Hierfür wird eine Menge *statesToExport* angelegt, in die zunächst der betrachtete Wartezustand eingefügt wird. Dann werden die Startzustände aller eingehenden Transitionen hinzugefügt. Anschließend wird über *statesToExport* iteriert und bei allen Zuständen, die keine Wartezustände sind, werden die Startzustände der eingehenden Nachrichten hinzugefügt, wenn diese nicht bereits in *statesToExport* enthalten sind. Dies wird so lange wiederholt, bis keine Zustände mehr hinzugefügt werden, also bis nur noch bei Wartezuständen eingehende Nachrichten vorhanden sind, deren Startzustände nicht in *statesToExport* vorkommen. Diesen Vorgang stelle ich hier nochmals exemplarisch am Beispiel aus Abbildung B-6 dar. Zunächst wird Zustand 6 zu *statesToExport* hinzugefügt. Dessen eingehende Transition (e) hat den Startzustand 5, der ebenfalls hinzugefügt wird. Dann wird über *statesToExport* iteriert, also es wird ein Programmteil für jedes Element ausgeführt. Zustand 6 ist ein Wartezustand, also wird kein weiterer Zustand hinzugefügt. Zustand 5 hat zwei eingehende Transitionen (c und d). Deren Startzustände (3 und 4) werden hinzugefügt. Dann wird erneut über die Menge iteriert, da in dem vorhergehenden Durchlauf Zustände hinzugefügt wurden. Bei Zustand 6 ergibt sich dasselbe Verhalten, wie beim ersten Durchlauf. Die vorhergehenden Zustände von Zustand 5 sind bereits vorhanden, also werden sie nicht hinzugefügt. Die Zustände 3 und 4 haben jeweils eine eingehende Nachricht, deren Startzu-

stände noch nicht enthalten sind, daher werden diese Zustände hinzugefügt. Da wiederum Zustände hinzugefügt wurden, findet ein weiterer Durchlauf statt. Bei den Zuständen 5 und 6 ist das Verhalten wie beim vorhergehenden Durchlauf. Ebenso wird bei den Zuständen 3 und 4 kein vorhergehender Zustand hinzugefügt, da diese bereits vorhanden sind. Die Zustände 1 und 2 sind Wartezustände, daher wird auch hier kein weiterer Zustand hinzugefügt. Da in diesem Durchlauf keine Zustände hinzugefügt wurden, sind jetzt alle notwendigen Zustände in *statesToExport* enthalten. Nun wird erneut über *statesToExport* iteriert. Dabei wird jeder Zustand *subGraphExporter* zum Exportieren übergeben. Zudem werden von jedem Zustand alle ausgehenden Transitionen exportiert, deren Zielzustand ebenfalls in *statesToExport* enthalten ist. Somit werden alle Transitionen zwischen den exportierten Zuständen ebenfalls exportiert. Nun wird *subGraphExporter* angewiesen die Graphviz-Datei zu erzeugen und den Graphiz-PDF-Export zu starten. In den Dateinamen wird die Nummer des Zielwartezustands – im hier Dargestellten Beispiel die Nummer 6 – hinzugefügt.

B.6. Cold verbotene Nachrichten

Es hat sich gezeigt, dass ein Event, das mit einer cold verbotenen Nachricht vereinbar ist gleichzeitig aber eine safety-Violation auslöst auch eine safety-Violation auslöst. Damit ist es nicht möglich das Reaktivieren von MSDs zulässig zu machen, indem, durch eine cold verbotene Nachricht, die der ersten Nachricht entspricht, der alte Durchlauf beendet wird und gleichzeitig ein neuer Durchlauf gestartet wird. Das liegt daran, dass zuerst überprüft wird, ob eine safety-Violation vorliegt, bevor die verbotenen Nachrichten behandelt werden.

Um das Reaktivieren von MSDs, wie beschrieben, möglich zu machen, habe ich in der Methode *handleViolations* vor dem abprüfen, ob eine safety-Violation vorliegt (*isSafetyViolating*[25]), einen Block eingefügt, der prüft, ob das Event mit eine cold verbotenen Nachricht vereinbar ist. Hierfür prüfe ich, ob die Nachricht in *coldForbiddenEvents*[26] enthalten ist. Wenn dies der Fall ist, dann wird die Methode *handleColdViolations*[27] aufgerufen, in der das MSD durch eine cold-Violation beendet wird.

B.7. Nicht aktivierte MSDs ausgeben

Wie in Kapitel 6.2.5 beschrieben, kann es vorkommen, dass einzelne MSDs bei der Synthese nicht aktiviert werden. Dies ist ein Indiz dafür, dass die Anforderungen nicht vollständig sind oder fehlerhaft modelliert wurden. ScenarioTools hatte bislang hierfür keinen Mechanismus, um dies dem

[25] Aus *ModalityImpl*
[26] in der Klasse *ActiveProcessImpl* definiert
[27] Aus der Klasse *MSDRuntimeStateImpl*

Entwickler anzuzeigen. Manuell zu prüfen, ob alle Diagramme aktiviert wurden, also ob jedes Diagramm in einem der Zustände aktiviert ist, ist aufgrund der Vielzahl von Anforderungen zu aufwändig.

Hierfür habe ich ScenarioTools derart erweitert, dass nach der Synthese in der Eclipse-Konsole eine Liste aller nicht aktivierten MSDs aufgeführt wird. Diese Liste muss für eine gute Useability in den Dialog, der nach Abschluss der Synthese angezeigt wird, integriert werden oder in den Exportierten Graphen ausgegeben werden. Zur Erstellung der Liste habe ich die Klasse *MSDRuntimeStateGraphImpl* mit einer unsortierten Liste *untouchedInteractions* erweitert. Am Ende der *init*-Methode der Klasse *MSDRuntimeStateGraphImpl* werden alle Interaktionen in *untouchedInteractions* geschrieben. In der Methode *createActiveMSD*[27], die MSDs aktiviert, wird das zu aktivierende MSD bzw. dessen Interaktion aus *untouchedInteractions* entfernt. Zuvor wird überprüft, ob die Interaktion noch in der Liste vorhanden ist, wenn nicht, wird die Interaktion nicht nochmals entfernt. Alle Interaktionen, die nach Abschluss der Synthese noch in der Liste sind, sind nicht aktiviert worden. Am Ende wird die Liste in der Eclipse-Konsole ausgegeben.