Masterarbeit

Daniel Lorenz

Modellgetriebenes Testen von Simulationsmodellen unter Verwendung von SysML

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Daniel Lorenz

Modellgetriebenes Testen von Simulationsmodellen unter Verwendung von SysML

Masterarbeit eingereicht im Rahmen der Masterprüfung
im Studiengang Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Bettina Buth
Zweitgutachter : Prof. Dr. Franz Korf

Abgegeben am 25. Februar 2011

Daniel Lorenz

Thema der Masterarbeit

Modellgetriebenes Testen von Simulationsmodellen unter Verwendung von SysML

Stichworte

SysML, MiL, modellgetriebenes Testen, Simulink, Windkraftanlage, Topcased, Messina, CTE

Kurzzusammenfassung

Diese Arbeit ist eine Machbarkeitsstudie über das modellgetriebene Testen mit SysML. Die Studie diskutiert zum einen eine machbare und praktische modellgetriebene Test-Methodologie mit SysML und zum anderen eine konkrete Realisierung einer Werkzeugkette zur Unterstützung dieser Methodologie. Für den Nachweis der Machbarkeit, wird eine Simulation einer Windkraftanlage als Testobjekt für das Fallbeispiel verwendet.

Daniel Lorenz

Title of the paper

Model-driven testing of simulation models using SysML

Keywords

SysML, MiL, model-driven testing, Simulink, wind energy converter, Topcased, Messina, CTE

Abstract

This paper is a feasibility study on model-driven testing with SysML. The study discuss a feasible and practical model-driven testing methodology with SysML on the one hand and a concrete realization of a tool chain supporting the methodology on the other. To provide proof about feasibility, a simulation of a wind energy converter is used as the test object for the case study.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

1. Einführung

Informations- und Steuerungs-Funktionen von eingebetteten Systemen werden mit wachsenden technischen Möglichkeiten zunehmend komplexer - und somit in gleichem Maße fehlerträchtiger. Gerade in sicherheitskritischen Anwendungen, wie sie im Automobil- oder Avionik- Bereich vorkommen, müssen diese Systeme jedoch auch bei gestiegener Funktionalität nachweislich korrekt und zuverlässig sein.

Um die Entwicklung solcher Systemen bewältigen zu können, wurden mit zunehmendem Erfolg modellbasierte Entwicklungsmethoden eingeführt. Dies gilt zuerst für den Bereich der Softwareentwicklung, in welchem sich die modellbasierte Entwicklung mit der Modellierungssprache UML etabliert hat. Später wurden diese Ansätze auf den Bereich der Systementwicklung mit der Systems Modeling Language (SysML) übertragen, welche aus der UML abgeleitet wurde.

Als eine Qualitätssicherungsmaßnahme ist das systematische Testen solcher Systeme unabdingbar, wobei jedoch die Komplexität auch der zu erfüllenden Testaufgaben entsprechend zunimmt. Es liegt nahe, entsprechende modellbasierte Vorgehensweisen auch für den Bereich des Testens konsequent anzuwenden. Dies wird seit Jahren erforscht, und viele Methodiken wurden schon dazu entwickelt [22]. In dieser Arbeit werden einige dieser Ansätze unter dem Aspekt der Durchgängigkeit ausgewählt und die so insgesamt definierte Methodologie hinsichtlich ihrer praktischen Anwendbarkeit untersucht.

1.1. Motivation

In der Entwicklung von Systemen jeglicher Art bedarf es systematischen Vorgehensmethoden. Das Systems Engineering hält solche multidisziplinäre Methoden für die Entwicklung von Systemen bereit. Die aus diesem Kontext herangewachsene und noch relativ junge Modellierungssprache SysML soll diese Vorgehensmethoden in Modellen unterstützen. Besonders bei der Entwicklung von eingebetteten Systemen, wo das zu entwickelnde und evaluierende System ein System in einem größeren System ist (System of Systems), sind die Methoden aus dem Systems Engineering und die Fähigkeiten von Modellen wie der SysML, Systeme hierarchisch zu modellieren, wichtige Faktoren für ein erfolgreiches Ergebnis. Sie sollen den Entwickler unterstützen, durch geeignete Wahl des Abstraktionsgrades in den Modellen sowohl den System-Kontext zu überblicken als auch auf die jeweils interessierenden konkreten Gesichtspunkte in der Entwicklung fokussieren zu können.

Zahlreiche Publikationen und wissenschaftliche Arbeiten haben sich mit dem Thema Entwicklung mit SysML auseinandergesetzt. Immer mehr Publikationen beschäftigen sich ebenfalls mit dem Thema Testdesign und Testfallgewinnung mithilfe der SysML (siehe [2] und [34]). Bisher beschreiben die meisten Arbeiten noch einzelne Methoden oder Aspekte für das modellbasierte Testen, aber wenige betrachten das ganzheitliche methodische Vorgehen beim Testen der Systeme mit SysML. Eine Machbarkeitsstudie über das durchgängige

modellbasierte Testen mit der SysML kann identifizieren, welche Probleme und neue Themenstellungen bei der Durchführung der Methodiken aufkommen können, und wie mit der Komplexität einhergehenden Menge an Testfällen umgegangen werden kann.

Durch die Demonstration der Machbarkeit lassen sich die Stärken und Schwächen der SysML für das modellbasierte Testen verdeutlichen. Ferner wird bei der Studie ein Beispiel für eine Methodologie und Vergehensmethodik für das modellbasierte Testen mit SysML entwickelt. Dieses Beispiel kann hinterher im Rahmen weiterer Studien oder Arbeiten für Argumentationen oder zur Ideenfindungen herangezogen werden.

1.2. State of the Art

Mit der wachsenden Komplexität in der Entwicklung von Systemen, sind auch die Testmethoden mit gewachsen. Immer ausgeklügeltere Testmethoden wurden entwickelt, um systematisch an das Testen dieser Systeme heran zu gehen und die Vielfalt der Möglichkeiten ein System zu testen in den Griff zu bekommen. Methoden wie die Grenzwertanalyse oder die Äquivalenzklassenbildung für eine systematische Herangehensweise zur Erschließung von Eingabewerten, oder Methoden für das zustandsgetriebene Testen wurden entwickelt, um bestimmte Testkriterien zu erfüllen.

Zunehmend werden Tests in der Industrie vorab auf Simulationsmodellen durchgeführt, um frühzeitig Fehler in den Designentscheidungen zu entdecken und um sicherheitskritische oder kostenintensive Tests an realen Prototypen weitestgehend zu vermeiden.

Ein oft verwendetes Werkzeug für die Modellierung und Simulation von dynamischen Systemen ist das Werkzeug Matlab/Simulink. Mithilfe von mathematischen Blöcken können simulierfähige Modelle erstellt und über die Zeit ausgeführt werden. Diese Modelle können mit Eingabewerten stimuliert und auf ihr Reaktionsverhalten an den Ausgängen ausgewertet werden. Entspricht das Modell in den zu untersuchenden Aspekten näherungsweise der Realität, so erhält man Ergebnisse, mit denen man weitere Designentscheidungen treffen kann.

Allerdings nicht nur die Testobjekte werden mittlerweile in Modelle überführt, sondern auch das Testdesign und die Testfälle werden beim modellgetriebenen Entwickeln immer häufiger in Modellen entwickelt. Dabei werden die Informationen über das Verhalten und die Struktur des Testobjektes zusammen mit diversen test-relevanten Informationen in ein Modell abgebildet. Dieses wird von einem Codegenerator oder Simulator interpretiert und in ausführbare Testfälle gewandelt. Mit diesen Testfällen können die Testobjekte stimuliert und auf ihr korrektes Verhalten getestet werden.

Etliche Werkzeuge unterstützen bereits dieses Vorgehen des modellbasierten Testens (siehe IX-Studie [18]). Diese Werkzeuge verfügen entweder über eigens entwickelte Modellierspachen für das modellbasierte Testen oder bedienen sich weitverbreiteten Modelliersprachen, wie dem UTP (UML Testing Profile) [28]. Das UTP beispielsweise ist ein erweiterndes

Profil für die von der OMG veröffentlichten Modellierungssprache UML (Unified Modeling Language) um einige Modellelemente und Stereotypen zur Modellierung von test-relevanten Informationen. Der "Testconductor" von IBM [19] wäre beispielsweise ein Werkzeug, dass das UTP verwendet.

Laut der Studie [23] gibt es wenige Publikationen im Bereich “Model-Based Testing” (MBT) die eine Analyse und Bewertung im realen Einsatz oder Fallstudien mit Analysen zu Effizienz und Effektivität in einer kontrollierten Praxisumgebung beinhalten. In dieser Studie wurden 406 Publikationen betrachtet und gerade mal 202 wurden als “gehaltvoll” angesehen und zur Analyse herangezogen. Die restlichen Publikationen beinhalteten unwesentliche oder replizierende Informationen. Von den 202 Publikationen wurden 85 ausgewählt, die sich entweder auf Modellierung mit UML oder auf andere aktuelle Modellierungssprachen bezogen und davon machten nur 15 % eine Machbarkeitsstudie für den Nachweis der grundsätzlichen Anwendbarkeit und nur 13 % eine Evaluierung mit einer praktischen Anwendung. Von den insgesamt 13 Publikationen mit einer Machbarkeitsstudie oder einer Evaluierung sind nur 3 auf der UML basierend. Bis heute sind zudem nur wenige Machbarkeitsstudien über MBT mit SysML gemacht wurden. Im Competence Center MOTION des Frauenhofer-Instituts FOKUS werden unter Anderem Studien über das MBT mit SysML gemacht. Jene Fallstudien beschäftigen sich beispielsweise mit der Überführung der Testmodelle in TTCN-3 und anderen gängigen Testnotationssprachen [10].

Die vorliegende Arbeit beschäftigt sich mit dem modellbasierten Testen von Simulationen. Für die Modellierung der Testmodelle wird die SysML als Modellierungssprache verwendet. Die meisten Ansätze des modellbasierten Testens sind im Bereich der Softwareentwicklung angesiedelt. Mit der SysML soll das "System under Test" (SuT) und der System-Kontext im Testmodell abgebildet werden, unabhängig ob es sich um Software oder Hardware in jeglicher Art handelt. Die Matrix in Abbildung 1 zeigt die Einordnung dieser Arbeit in den Modellierungssprachen. Nach dem Kenntnisstand dieser Arbeit, gibt es noch keinen offiziellen Standard für die Modellierung von Tests für Systeme mit der SysML. Ähnlich wie das UTP bei der UML soll eine Menge an Stereotypen die SysML für die Notation von test-relevanten Informationen erarbeitet werden.

1.3. Zielsetzung

Die Machbarkeitsstudie setzt sich mit der Testfallgewinnung und Testfallgenerierung von funktionalen Tests aus SysML-Modellen und dem systematischen daten- sowie zustandsgetriebenen Testen auseinander. Eine Untersuchung soll zeigen, wie und von welchen Modellelementen die testrelevanten Daten bezogen werden können und welcher Grad der Vollständigkeit an Informationen in so einem Modell benötigt wird, damit aus ihnen Testfälle generiert werden können.

Die Machbarkeitsstudie soll verdeutlichen, welche Möglichkeiten und Probleme in bestimmten Ansätzen des MBT im Allgemeinen existieren und wie eine mögliche Umsetzung die-

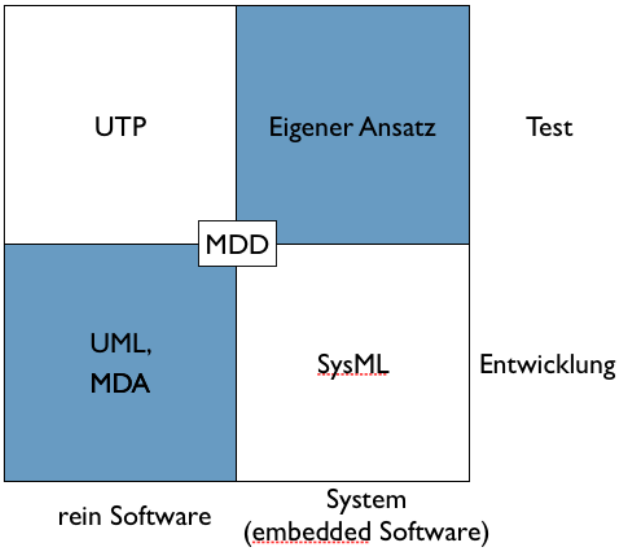


Abbildung 1: Einordnung der Machbarkeitsstudie

ser Ansätze in der SysML im Konkreten aussehen könnte. Ein Ziel ist die Integration von modellbasierten Testansätzen in der Modellierung mit SysML zu elaborieren. In diesem Zusammenhang ist eine durchgängige und praktikable Methodologie für das MBT mit SysML erarbeitet worden.

Als Fallbeispiel für die Studie dient die Steuerung und Regelung eines Windnachführungssystems einer Windkraftanlage. Das Windnachführungssystem wurde als reales Bespiel für die Untersuchung herangezogen, um möglichst realistische Ergebnisse zu erzielen, hinsichtlich Effektivität und Effizienz der Methodologie.

Für die Kodierung und Vereinfachung der testrelevanten Informationen im Modell, wird eine formale Testnotation entwickelt. Diese soll genutzt werden, um das Reaktionsverhalten des SuT in Form von Rückgabewerten und Eingaben im Modell zu kodieren.

Ferner wird untersucht, wie die SysML-Modelle für die Definition der Testkonfiguration verwendet werden können. Hierbei geht es vor allem um die Schnittstellen-Definition zwischen der Testumgebung und dem SuT und unter anderem um die Evaluierung von Methoden, Testszenarien hinsichtlich der Zustände des Systems abzuleiten.

1.4. Abgrenzung der Arbeit

Die Betrachtung dieser Arbeit beschränkt sich ausschließlich auf das Testen von elektronischen eingebetteten Steuerungssystemen. Die erarbeitete Methodologie erzeugt Testfälle

für funktionale Tests im Black-Box-Verfahren. Es werden keine nicht-funktionalen Tests oder White-Box-Verfahren behandelt.

Diese Arbeit diskutiert Methoden für das modellbasierte Testen. Der Anspruch dieser Arbeit ist jedoch nicht alle Methoden für das modellbasierte Testen anzuführen, sondern ein Beispiel einer Methodologie für das Vorgehen beim modellbasierten Testen zu erarbeiten.

Anhand eines Fallbeispiels soll die Machbarkeit der Methodologie gezeigt werden. Das Fallbeispiel, ein Steuerungssystem in einer Windkraftanlage, ist ein idealisiertes System, ausschließlich für die Machbarkeitsstudie entwickelt. Das Steuerungssystem ist einem realen System nachempfunden wurden, aber realisiert einige Funktionen in gewissen Gesichtspunkten abweichend von den realen Systemen.

2. Grundlagen

In diesem Kapitel werden die benötigten Grundlagen für das Verständnis dieser Arbeit erarbeitet.

2.1. Systematisches Testen

In dieser Arbeit werden laut [35] folgende Begrifflichkeiten definiert:

- Testkontrolle - gibt die Ablaufreihenfolge der Testfälle oder Testszenarien innerhalb eines Testkontexts an, steuert die Eingabe bei parametrisierten Testfällen und leitet Zustandswechsel ein
- Testdaten - Umfasst Eingabe-, Zustands- und Sollwerte für ein Testobjekt
- Testfall - Ein Testfall ist eine Menge von Eingabewerten, den für die Ausführung notwendigen Vorbedingungen und Randbedingungen, der Menge der erwarteten Ergebnisse (Sollwerten) und den erwarteten Nachbedingungen, entwickelt mit dem Ziel, ein oder mehrere Testkriterien abzudecken.
- Testszenario - Umfasst eine Menge von Testfällen mit einer definierten Reihenfolge
- Anforderung - Aussage über eine zu erfüllende qualitative und/oder quantitative Eigenschaft eines Systems
- Testorakel (Arbiter) - Entität, die über das erfolgreiche Abschließen eines Tests entscheidet.
- Testkriterium - Qualitätsmerkmal oder strukturelles Element des Testobjektes, welches verifiziert werden kann

Diese Informationen werden in der Regel für ein systematisches Testen benötigt. Beim Testdesign müssen diese Informationen parat sein, um automatisch Tests ableiten und fahren zu können. In der Vergangenheit wurden diverse Methoden und Techniken entwickelt Tests systematisch zu erschließen.

Eine Technik zum Testen von reaktiven Systemen ist die X-in-the-Loop Methode. Hierbei gibt es eine oder mehrere Instanzen, welche das SuT mit kontrollierten Stimuli anregen und die Reaktion des Systems aufzeichnen oder weiterverarbeiten. Man unterscheidet in der Regel zwischen "open loop" und "closed loop" XiL.

In einer open loop XiL-Simulation wird die Zustandsänderung des zu testenden Systems oder dessen ausgehenden Signale entweder aufgezeichnet oder für frühzeitige Warnungen genutzt. Anders als bei der open loop, haben die Zustandsänderungen oder Ausgangssignale bei closed loop Simulationen einen direkten Einfluss auf die Simulation. Es gibt eine Rückkopplung auf die Simulation, was einen anderen Simulationsablauf zufolge haben kann,

indem die Simulation auf den Ausgang des zu testenden SuT reagiert [33]. Besonders beim Testen von Regelsystemen ist eine Rückkopplung der Ausgänge auf die Eingänge des SuT wichtig.

Das X in XiL steht stellvertretend für eine Reihe von gleichen Methoden auf verschiedenen Klassen von SuT-Objekten. Die gängigen Bezeichnungen sind:

Hardware-in-the-Loop (HIL) testet Hardware in einer Simulation. Mittels Simulator werden Stimuli an die Schnittstellen der Hardware geschickt und ausgelesen.

Model-in-the-Loop (MIL) ist eine Technik, bei der statt Hardware oder Software ein Simulationsmodell des SuT zum Testen genutzt wird. Das Modell steht stellvertretend für eine Hardware oder Software und kann gleichermaßen über entsprechenden Schnittstellen, wie bei der HIL, stimuliert und ausgelesen werden.

Software-in-the-Loop (SIL) testet Software in einer Simulation. Das SuT ist eine Software, die während einer Simulation in einer geeigneten Umgebung ausgeführt wird.

Die Abbildung 2 zeigt skizziert einen klassischen Aufbau einer Model-in-the-Loop Simulation mit einer Mensch-Maschinen-Schnittstelle (HMI). Ein MiL-Testaufbau bietet viele Vorteile. Dadurch, dass man einige Komponenten eines Komplexen Systems in einem Simulator simulieren kann, kann schon in der frühen Phase der Entwicklung getestet werden, ohne alle Komponenten des Systems bereits fertig vorliegen zu haben. Auf diese Art und Weise lassen sich sehr komplexe Systeme leichter in den Griff bekommen, weil man iterativ eine Komponente nach der Nächsten in einem Testdurchlauf testet. Durch den kontrollierten Testablauf einer XiL-Simulation ist eine Reproduzierbarkeit der Testergebnisse möglich.

Die definierten Testfälle in einem XiL-Testaufbau sind wiederverwendbar, sodass verschiedene Varianten des SuT getestet und verglichen werden können, weil die Testergebnisse einheitlich sind. Ein XiL Testaufbau bietet demnach eine Menge an Vorteilen:

- komplexe Systeme in Griff bekommen
- Vorabtests anhand von Modellen, dadurch mögliche schnelle Fehlererkennung
- Testen in verschiedenen Detailtiefen und Integrationsstufen
- Reproduzierbarkeit der Testergebnisse
- Wiederverwendbarkeit und somit Vergleichbarkeit zwischen verschiedenen SuTs
- einheitliche automatische Testergebnisse für Dokumentation

Im Zuge der Gestaltung von Testdesign werden immer effizientere Methoden evaluiert, große komplexe Systeme systematisch zu testen. Modellbasierte Entwicklungsmethoden können einen großen Beitrag dazu leisten.

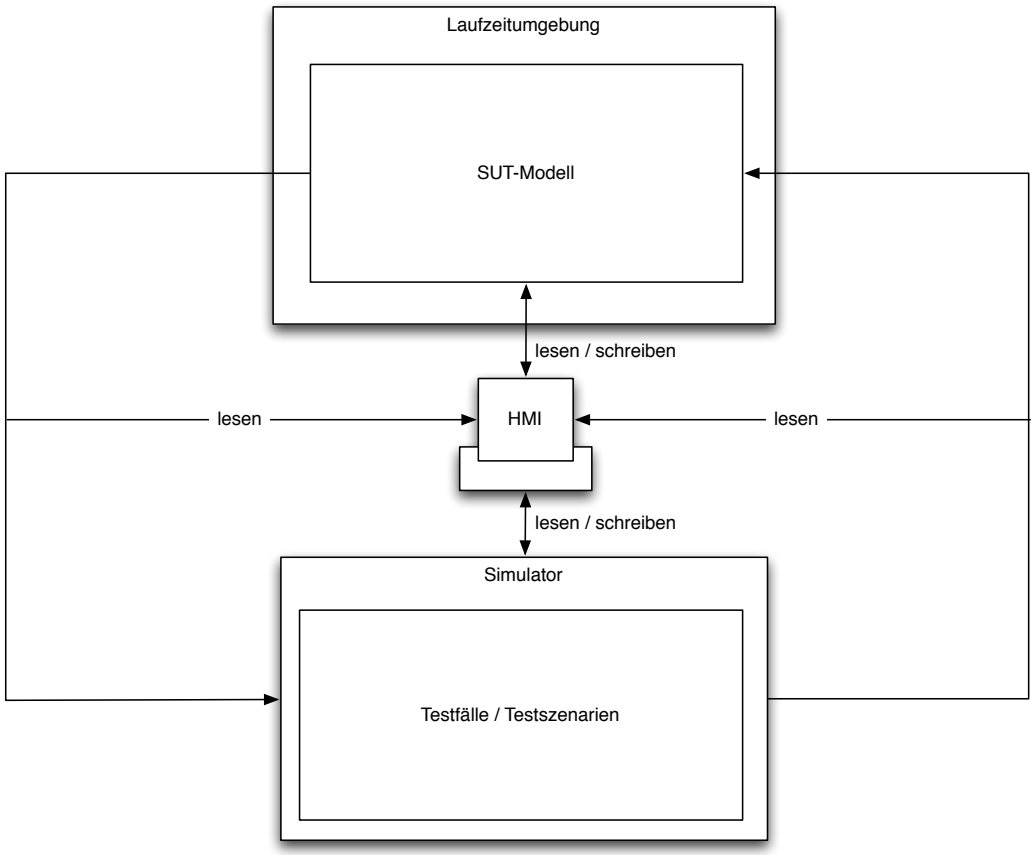


Abbildung 2: MIL-Testaufbau (closed-loop)

2.2. Modellbasiertes Testen

Modellbasiertes Testen ist mittlerweile kein fremder Begriff mehr in der Entwicklung von Systemen. Der Begriff MBT(model-based testing) wird von etlichen Gruppen verschieden definiert. Nach [4] werden aus Modellen, die das Verhalten eines Systems beschreiben, Testfälle abgeleitet. Gegenwärtig gibt es für die UML verschiedenste Profile für MBT, unter anderen das UTP (UML Testing Profile). Laut der Studie [23] gibt es nur sehr wenige Machbarkeitsstudien, für den Nachweis der grundsätzlichen Anwendbarkeit, und Evaluierungen mit einer praktischen Anwendung im Bereich MBT. Dennoch wird MBT schon erfolgreich in größeren Projekten eingesetzt und Publikationen über MBT geschrieben. Inzwischen gibt es auch schon einige Publikationen über MBT Ansätze mit SysML, wie [2] oder [34].

Die Unterstützung von Werkzeugen für MBT ist für einige Modellierungssprachen gegeben. So unterstützen beispielsweise viele Entwicklungsumgebungen (bsp. IBM TestConductor

[19]) das UTP. Aber auch Modellierungswerkzeuge mit domänenspezifischen Modellierungssprachen besitzen Funktionen und Modellierungselemente für das MBT [9].

Im Gegensatz dazu, wird MBT auch als analytische Qualitätssicherung für ausführbare Modelle (Simulation) verstanden [18]. Hierbei wird das Modell selbst zum Testobjekt und direkt getestet [31], noch bevor die Software implementiert wird (Modelltest). Beide Ansichten schließen sich aber nicht gegenseitig aus.

In dieser Ausarbeitung wird der Begriff MBT als Methode für die Gewinnung von Testkonfigurationen und Testfällen aus Modellen verstanden. Für die Ableitung von Tests aus Modellen bedarf es mehr an Diagrammtypen als nur die Verhaltensdiagramme.

Für dieser Arbeit wird zwischen folgenden Modellen unterschieden:

Simulationsmodell Ein ausführbares Modell. Der Begriff Simulation ist equivalent dazu.

SuT-Modell Das SuT-Modell ist ein Modell des SuT. In dieser Arbeit liegt das SuT-Modell als ein Matlab/Simulink-Modell vor und somit als ein Simulationsmodell

Testmodell Ein Modell, dass die test-relevanten Informationen enthält, um Testfälle zu generieren oder Testkonfigurationen zu beziehen

Java-Modell ein Modell, dass in Java vorliegt

2.2.1. Methodologie des MBT

Modellbasiertes Testen ist kein festgeschriebenes Vorgehen, sondern ein Begriff für das Testen mit einem systematischen Vorgehen mittels Modellen. Modelle sind eine abstrakte Abbildung der Wirklichkeit. Ein Modell ist eine Approximation, eine Repräsentation oder eine Idealisierung eines ausgewählten Aspektes einer Struktur, eines Verhaltens, einer Operation oder einer Implementierung eines realen Prozesses, Konzeptes oder Systems [20]. Dieser Abstraktionsgrad soll helfen, die wichtigen Aspekte des Testens besser fokussieren zu können. In der Regel werden aus formlosen und unvollständigen Anforderungen mentale Testmodelle gebildet. Die Idee des modellbasierten Testens ist, explizite Verhaltensmodelle zur Kodierung des erwarteten Verhaltens zu verwenden [40]. Für die Modellierung von vollständigen Tests, reichen aber die Verhaltensmodelle nicht aus. Es gilt zu untersuchen, welche Informationen für das Testen benötigt werden und wie sie in Modellen notiert werden können.

Häufig wird das modellbasierte Vorgehen, sei es MBSE (model-based system engineering), MBD (model-based design), MBA (model-based architecture) oder MBT, als ein Arbeitsmehraufwand gesehen. In der Tat ist der Entwurf eines Modells ein gewisser Mehraufwand. Dieser Mehraufwand ist jedoch gerechtfertigt, wenn man auch einen Mehrwert dadurch erzielt. Die zu erzielenden Mehrwerte sind:

- strukturiertes Vorgehen beim Testen über Modelle auf einem hohen Abstraktionsgrad

- bessere Übersicht des Testes anhand des Modells
- Testen auf Modellen
- automatische Testgenerieren und Testdurchführung aus Testmodellen
- Wiederverwendbarkeit von Modellen und leichte Abänderbarkeit
- Generierung von Testdokumentation

Einige Studien darüber haben gezeigt, dass die Qualität der Tests mit MBT, durch das systematische Vorgehen mit Modellen, besser wird [29]. Das erzeugte Modell ist abstrakter als das SuT (System under Test). Das Validieren, Warten und das Verständnis für die Tests in einem Modell ist einfacher. Modelle sind in der Regel zugänglicher für die Generierung von Testfällen.

Um den Verwaltungs- und Modellieraufwand zu reduzieren ist es wünschenswert, Programmcode und Testfälle aus einem einzelnen Modell zu generieren. Es würde dem Tester viel Zeit sparen, kein neues Modell erzeugen zu müssen. In [30] wird dieses Thema behandelt. Daraus geht hervor, dass eine gewisse Redundanz in Modellen unabdingbar ist, falls aus einem Modell sowohl Programmcode, für das zu entwickelnde System, als auch ausführbarer Code, zum Testen des Systems, generiert werden soll. Für den Fall, man leitet den Code für das System und die Testfälle aus nur einem Verhaltensmodell ab, würde man letztlich nicht das korrekte Verhalten des Systems testen, sondern eine qualitative Aussage darüber gewinnen, wie gut der Programmcode-Generator das Modell umgesetzt hat und ob die Voraussetzungen der umgebenen Komponenten eingehalten wurden. Der Programmcode-Generator wird sozusagen einer Qualitätssicherung unterzogen. Als Redundanz in Modellen sind in [30] die Verhaltensmodelle gemeint, ferner die Ausgaben des SuT. Unter anderem sollte berücksichtigt werden, dass die Verhaltensmodelle zum Testen eines Systems, nicht zwingender weise sämtliches Verhalten des SuT beinhalten müssen. Meist reicht es aus, nur die für den Testfall relevanten Verhaltensaspekte in einem Modell abzubilden.

Nach [36] gibt es drei fundamentale Charakteristiken für die Verwendung von Modellen:

- Abbildungen von der konkreten zur abstrakten Welt
- Modelle, die einer bestimmten Zielsetzung dienen
- Modelle die vereinfachen. Sie lassen bestimmte Eigenschaften aus und reflektieren nicht die Realität

Für die Vereinfachung der Modelle gibt es zwei Ansätze:

Das Weglassen von Informationen In der Literatur gibt es zahlreiche Beispiele für das Abstrahieren von Modellen und das Ergänzen von Informationen für das MBT in Form von Treiber-Komponenten. Bei diesem Ansatz der Vereinfachung von Modellen wird

nur eine bestimmte Sicht der Eigenschaften der Welt behandelt. Benötigte Informationen können inkrementell von Menschenhand hinzugefügt werden.

Das Kapseln von Informationen Informationen können hierarchisch gekapselt werden, um die Übersicht zu verbessern. Weiterhin kann ohne Wissen über die Implementierung modelliert werden. Die Implementierung kann zu einem späteren Zeitpunkt nachgefügt werden, indem man in eine Hierarchie-Stufe tiefer die Modellelemente genauer spezifiziert.

Über Treiber-Komponenten werden Funktionen auf den Modell-Input und -Output Daten ausgeführt, die die abstrakten Daten in konkrete Testdaten und zurück konvertieren. Die Funktionen geben über die Implementierung des SuT Aufschluss [30].

In der Literatur gibt es zahlreiche verschiedene Methodiken, MBT zu betreiben. Folgend sind einige verschiedene Beispiel-Szenarien aufgelistet, wie man MBT betreiben kann:

- Die Code Generierung und das Generieren der Testfälle erfolgt aus dem gleichen Modell.

  Beim Testen besteht eine gewisse Redundanz der Testdaten: die beabsichtigten/erwarteten und die tatsächlichen Daten. Diese Redundanz fehlt, falls der Code des SuT und die Testfälle sich ein Modell, aus dem sie generiert werden, teilen. Man würde den Code sozusagen gegen sich selber testen. Daher sind an dieser Stelle beim Testen keine automatischen Vergleiche der Ergebnisse sinnvoll.

- Automatische Generierung des Modells aus dem Code (Reverse Engineering). Das Modell wird genutzt, um Testfälle zu generieren.

  Beim Generieren des Modells aus dem SuT Code gerät man wieder in die gleiche Situation, dass man keine redundanten Daten für den automatischen Vergleich erhält. Weiterhin wird das Modell hinsichtlich einer bestimmten Absicht erstellt. Die Informationen sind in der Regel unvollständig und müssen nachträglich manuell in das Modell integriert werden.

- Manuelle Generierung des MBT Modells aus der Spezifikation des SuT. Wie im vorigen Punkt wird das Modell genutzt, um Testfälle automatisch zu generieren.

  In diesem Szenario gibt es erstmals eine Trennung zwischen der Implementierung des SuT – also der Informationen im Code - und dem Modell, aus dem die Testfälle generiert werden. Automatische Vergleiche der Ergebnisse beim Testen sind sinnvoll.

- Die letzte Variante ist die Generierung des Codes des SuT und der Testfälle aus zwei seperaten Modellen heraus.

  Der automatische Vergleich der Ergebnisse beim Testen ist sinnvoll, weil es zwei individuelle Modelle mit redundanten Testdaten gibt.

In dieser Arbeit wird die letzte Methodik für das MBT verwendet, damit die redundanten Daten für den automatischen Vergleich gegeben sind.

2.3. System Modeling Language

Die SysML (System Modeling Language) [26] ist eine noch relativ junge Modellierungsprache die ihren Ursprung in der UML (Unified Modeling Language) hat und von der OMG spezifiziert und veröffentlicht wird. Mit der SysML lassen sich Systeme modellieren. Man kann die Strukturen, das Verhalten und Bedingungen eines Systems in einem Modell abbilden. Durch Beziehungen der Modellelemente untereinander können komplexe Systemstrukturen modelliert werden. Systeme in Systemen und ganze Lebenszyklen eines Systems können ausgearbeitet werden. Die SysML hat verschiedene Diagrammtypen für die grafische Darstellung dieser Modelle.

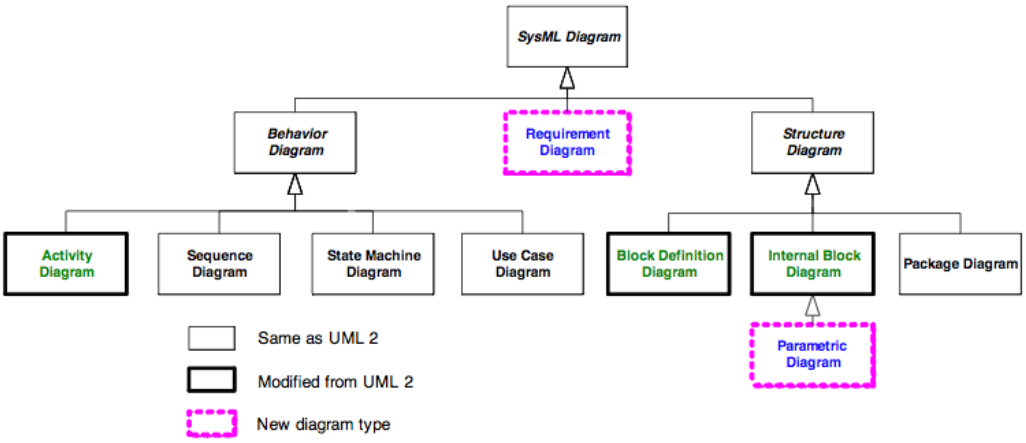


Abbildung 3: Diagrammtypen aus der SysML aus [26]

Die Diagrammtypen sind im Groben in Struktur/Daten-Diagrammen, Verhaltensdiagrammen und in Anforderungsdiagrammen aufgeteilt (siehe Abb. 3). Durch Referenzen auf den Eigenschaften der Modellelemente können Beziehungen zwischen den verschiedenen Typen erstellt werden, sodass ein großes hierarchisches Modell aus verschiedenen Digrammtypen entsteht.

Die SysML soll das MBSE (model-based system engineering) zum Entwickeln von Systemlösungen in Bezug auf wiederkehrende und oft auftretenden technologischen Aufgaben unterstützen.

Systems Engineering ist ein multidisziplinäres Vorgehenskonzept, um ausgewogene Systemlösungen in Bezug zu den Bedürfnissen verschiedener Interessengruppen zu entwickeln

[11]. Das Systems Engineering beinhaltet sowohl Management-Prozesse als auch technische Prozesse zum minimieren von Risiken, welche dem Erfolg des Projektes beitragen sollen. Die Systems-Engineering-Management-Prozesse befassen sich mit den Themen wie die Planung des technischen Aufwandes, das Risikomanagement, das Controlling der technischen Richtlinien und das Überwachen der technischen Ausführung im Projekt. Die technischen Prozesse hingegen beinhalten das Spezifizieren, den Entwurf und das Verifizieren der Systeme. Die Praxis von Systems Engineering ist nicht statisch, sondern entwickelt sich ständig weiter, um mit steigenden Anforderungen umzugehen.

Im Groben kann man die technischen Prozesse in drei Prozesse unterteilen, wie in Abbildung 4 zu sehen. Es gibt Prozesse für die Systemspezifikation und das Design, wo die Anforderungen an das System und die an die einzelnen Komponenten zur Erfüllung der Bedürfnisse der Interessengruppen erschlossen werden. Der Entwurf der Komponenten und dessen Test geben Rückschlüsse auf die Spezifikation und Anforderungen der Komponenten. Mit den fertig implementierten und positiv getesteten Komponenten kann das System zusammengeführt und mit den Systemanforderungen Integrations- und Systemtests am System durchgeführt werden. Die Ergebnisse geben wiederum Rückschlüsse für die Anforderungen und die Spezifikation.

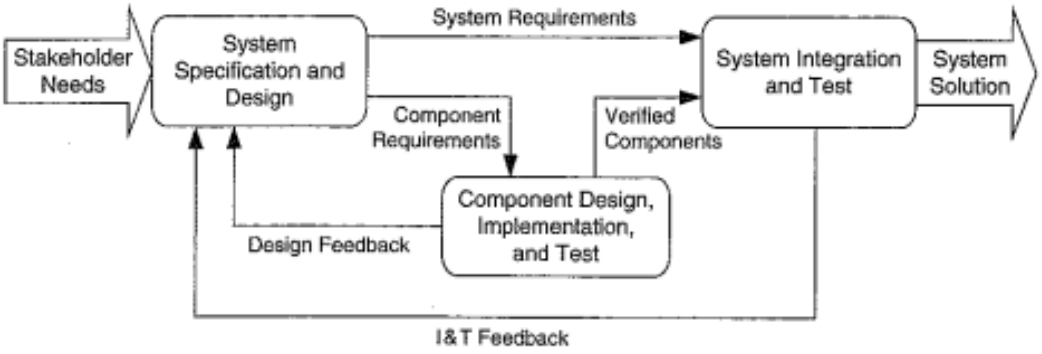


Abbildung 4: Technische Prozesse des Systems Engineering im Groben aus [11]

Die Rückverfolgbarkeit der Systemkomponten und den Testergebnissen zu den Anforderungen und der Spezifikation ist ein wichtiger Faktor beim System Engineering. Somit lassen sich Rückschlüsse auf Designentscheidungen und den Anforderungen ziehen. In Kapitel 4.1.3 wird gezeigt, wie solche Verbindungen für Rückschlüsse zwischen Komponenten, Anforderungen und Tests in einem Modell hergestellt werden können.

2.4. Tools und Frameworks

In dieser Arbeit werden verschiedene Werkzeuge für die Modellierung und das Generierung von test-relevanten Informationen genutzt. Die folgenden Werkzeuge sind Hauptbestandteil der in dieser Arbeit ausgearbeiteten Methodologie für das MBT mit SysML und Simulationsmodellen.

2.4.1. Matlab/Simulink

Matlab/Simulink ist ein Softwarewerkzeug des Herstellers The Mathworks [38] zum Modellieren von dynamischen Systemen. Simulink baut auf der Software MATLAB auf, die für die Berechnung von mathematischen Gleichungen genutzt wird. Mit den grafischen Blöcken in Matlab/Simulink lassen sich hierarchische Modelle erstellen, die intern in mathematische Gleichungen überführt werden. Es können zeit-diskrete als auch zeit-kontinuierliche Modelle mit Simulink konstruiert werden. Der Vorteil von Matlab/Simulink ist die Fähigkeit, die Modelle zu simulieren. Matlab/Simulink hat verschiedene Solver (Löser für numerische Gleichungen) zum Lösen der Gleichungen und Simulieren der Modelle.

Mit dem zusätzlichen Werkzeug Real-Time Workshop lässt sich aus den Modellen Programmcode generieren. Die Zielsprache kann variabel über Konfigurationsdateien angepasst werden. In Verbindung mit dem Windriver Compiler [43] kann man aus den Modellen real-time fähige Simulationsmodelle für Messina generieren. Genauere Informationen zu dem Export von Matlab/Simulink Modellen nach Messina sind in Kapitel 5.1 zu finden.

2.4.2. Topcased

Topcased ist ein Plugin für die Entwicklungsumgebung Eclipse und basiert auf dem EMF (Eclipse Modeling Framework). Gedacht zur Modellierung von sicherheitskritischen Systemen, bündelt es weitere Plugins von Drittanbietern für eine geschlossene Toolkette. Es beinhaltet ein Requirements-Management-Tool zur Anbindung von Anforderungen. Das Importieren von Anforderungen aus Dokumenten (Excel, Word, u.s.w) ist gegeben.

Mit diesem Werkzeug lassen sich verschiedenste Modelle erstellen, unter anderem auch SysML-Modelle. Mit dem integrierten Codegenerator Acceleo lassen sich Codegeneratoren zum Generieren von Dokumenten und Programmcode aus Modellen erstellen. Topcased ist ein open-source Projekt und wird ständig weiterentwickelt. Das Werkzeug ist frei verfügbar und kann unter der Eclipse Public License verwendet werden. Mithilfe der Metamodellierungssprache Ecore von Eclipse können weitere Metamodelle für die Erstellung von eigenen Modellierungssprachen erzeugt oder für die Erweiterung anderer Metamodelle verwendet werden.

2.4.3. CTE XL

Der Classification Tree Editor (CTE) XL ist ein Produkt der Berner & Mattner Systemtechnik GmbH. Mit diesem Editor können Klassen-Bäume von Äquivalenzklassen modelliert werden. Der Editor implementiert die Klassifikationsbaummethode. Anhand von Regeln auf den Klassifikationen und/oder den Äquivalenklassen, können Kombinationen zwischen den Elementen hergestellt werden. Die Regeln können verwendet werden, um Testfälle in Form von Wertekombinationen zu erstellen. Die Wertekombinationen lassen sich in verschiedene Werkzeuge exportieren, wie der Export nach Messina (2.4.4). Jede Wertekombination stellt einen Testfall dar. Um die Anzahl der zu exportierenden Testfälle weiter zu beschränken, können Filter-Regeln auf den Kombinationen mit logischen Ausdrücken angewendet werden.

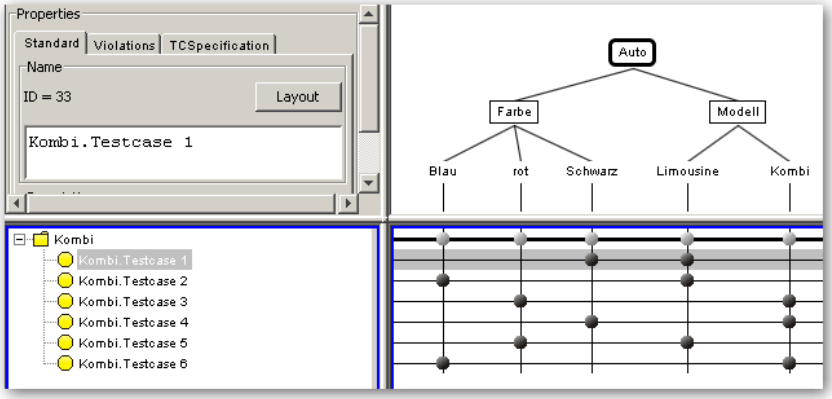


Abbildung 5: Beispiel eines CTE Baumes

2.4.4. Messina

Messina ist wiederum ein Produkt der Berner & Mattner Systemtechnik GmbH. Die Software basiert auf der Eclipse Umgebung und ist eine Testplattform für eingebettete Systeme. Messina unterstützt das Testdesign für XiL Testmethoden (MiL, SiL, HiL). Die Kommunikation der Testkomponenten untereinander wird über einen sogenannten Signalpool realisiert. Für jeden Signaltypen wird ein eigener Nachrichtenkanal aufgebaut, auf dem mehrere Teilnehmer Lesen und Schreiben können. Die Kommunikation findet wahlweise entweder über eine TCP/IP Socket-Verbindung oder über POSIX Message Queues statt.

Die Testkonfiguration eines Tests wird in Messina durchgeführt. Messina als Testplattform definiert das Mapping zwischen den Testkomponenten und den Testfällen, die Testreihenfolge und kann zur Teststeuerung verwendet werden. Über eigene Messina-Bibliotheken lassen sich Testfälle in Java definieren und zusammen in einem Testszenario bündeln. Messina

besitzt verschiedene Schablonen für die Erstellung von Live-Anzeigen und kann innerhalb der Testdurchläufe über die Live-Anzeigen Eingaben manipulieren. Über die Import-Funktion lassen sich verschiedene Simulationsmodelle in die Testumgebung einbinden, unter anderem Matlab/Simulink-Modelle.

3. Beschreibung des Fallbeispiels

Eine Fallstudie, die die Effektivität der in dieser Arbeit zugrunde liegenden Technik zum systematischen Testen untersuchen soll, benötigt ein Fallbeispiel für den Nachweis der Machbarkeit. Anhand des Fallbeispiels soll nachvollziehbar gezeigt werden, welche Menge und mit welcher Herangehensweise Testfälle erschlossen werden können. Ferner soll das Beispiel dem Leser helfen, die Techniken zum generieren der Testfälle leichter zu verstehen.

Für das Fallbeispiel wurde ein in sich geschlossenes System gesucht, welches nicht all zu komplex aufgebaut ist. Das System sollte schnell zu erfassen sein und übersichtlich viele Testszenarien benötigen um ausreichend getestet zu werden. Zudem sollte das System zustandsbasiert regeln und zeitliche Anforderungen haben.

In dieser Arbeit dient ein Teilsystem einer Windkraftanlage (WKA) als Fallbeispiel. Die Wahl des Fallbeispiels fiel auf die Windnachführung einer Windkraftanlage, weil das System in die oben genannten Kriterien passt. Zudem sind die WKA mit der regenerativen Energiegewinnung eine höchstaktuelle Technologie, die in der Zukunft noch weiterentwickelt werden wird.

Das Windnachführungssystem, oder auch Gierregelung gennant, steuert die Ausrichtung der Gondel einer WKA. Je nach Windstärke, Windrichtung oder anderen Zuständen verhält sich die Regelung verschieden. Die Tests für das Windnachführungssystem sollen hauptsächlich funktionale Tests beinhalten. Die Tests sollen mögliche Fehler in der Funktionalität der Regelung in den verschiedenen Zuständen aufdecken können. Hierzu gehören lediglich die Positivtests.

3.1. Windnachfürhungssystem einer Windkraftanlage

Die Hauptaufgabe des Windnachführungssystem einer Windkraftanlage ist das Drehen der Gondel in die Position bzw. den Winkel in dem die Rotorblätter optimal für die Stromerzeugung laufen können. Aktive Nachführungs-Systeme von großen WKA (> 1 MW) sind in der Regel mit Azimutantrieben auf Azimutlagern ausgestattet. Um starke Schwankungen der Gondel bei Starkwindbetrieb zu vermeiden, werden Azimutbremsen eingesetzt. Der Antrieb befindet sich normalerweise zwischen Gondel und Turm auf einem Drehkranz mit Stirnradverzahnung (siehe Abb. 6).

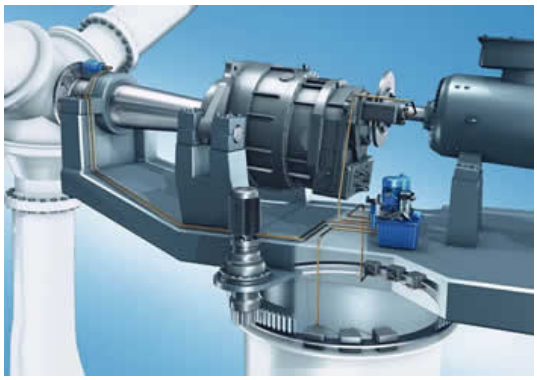


Abbildung 6: Schnitt eines Windnachfuehrungssystems aus [5]

Das hier modellierte Windnachführungssystem (WNS) kann in verschiedene Zustände versetzt werden. Diese Zustände bilden sich aus den aktuell anliegenden Signalen an den Eingängen des Systems. Das System ist nicht Fehlertolerant ausgelegt und wechselt bei einem Fehler in einen Fehlerzustand aus dem es nur noch mit einem Reset rauskommt. Weiterhin muss ein WNS berücksichtigen, dass die Kabel, die von der Gondel in den Trägerturm führen, sich nicht zu sehr verwinden. In der Regel haben WKA dafür einen Verwindungs-Sensor die die Drehungen um die eigene Achse in die gleiche Richtung zählen. Bei einer gewissen Anzahl der Umdrehungen in die gleiche Richtung dreht das WNS die Gondel in die Zählerposition null zurück. In unseren Beispiel aber wird grundsätzlich vermieden, dass die Gondel sich um 360 Grad drehen kann. Zwischen 135 und 225 Grad darf die Gondel nicht eindrehen (siehe Abbild. 8 (verbotener Bereich)). Somit ist die Steuerungslogik des Fallbeispiel-Modells etwas anders als bei den regulären WKA-WNS.

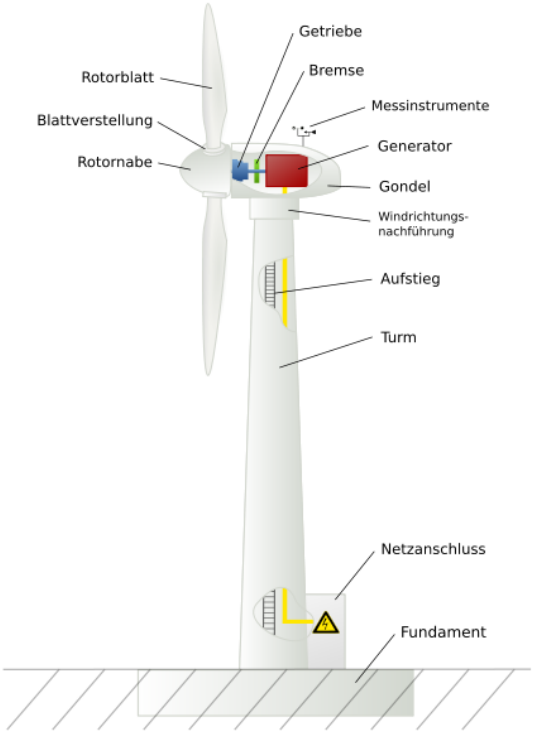


Abbildung 7: Bild aus [37]

Das Verhalten des WNS definiert sich über seinen aktuellen Zustand. Das WNS kann in folgende Zustände wechseln:

- **Normaler Betrieb** - in diesem Zustand regelt das WNS die Gondel in den Wind

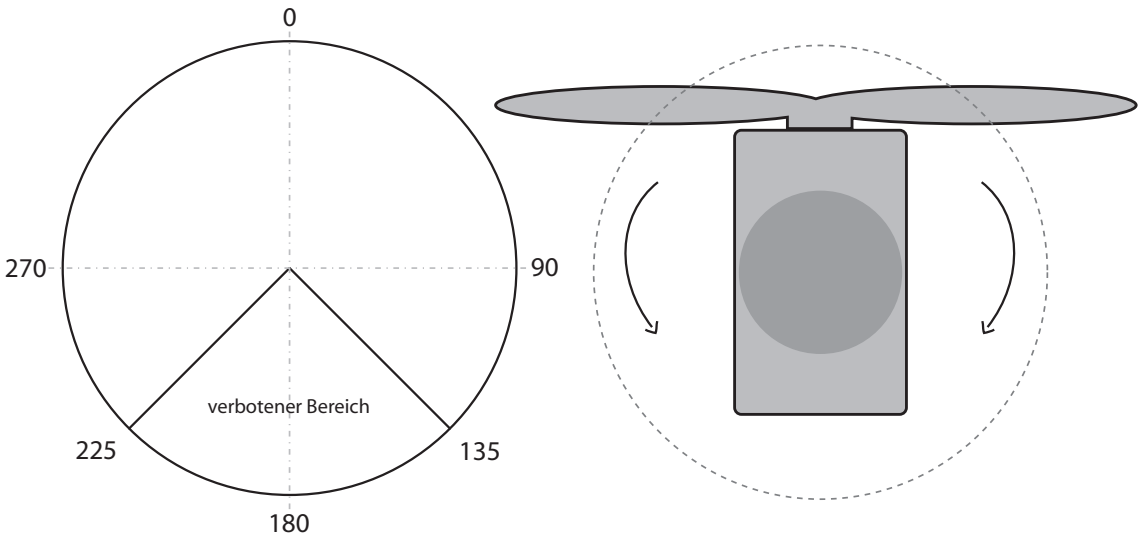


Abbildung 8: Betrachtung der Winkel der WKA aus dem Fallbeispiel

- **Manueller Betrieb** - der manuelle Betrieb ist für Wartungsarbeiten an der WKA gedacht. In diesem Zustand kann das Wartungs-Personal die Gondel in einen gewünschten Winkel drehen
- **Sicherheitsabschaltung** - die Sicherheitsabschaltung ist für Notfälle gedacht. In diesem Zustand gibt das WNS den Befehl die Motoren auszuschalten und die Bremsen anzuziehen.
- **Sturmwarnung** - Bei zu starken Wind dreht sich die Gondel mit den Rotorblättern einmal um 90 Grad zur Windrichtung und verharrt an dieser Stelle
- **WindgeschwindigkeitZuLangsam** - Falls sich das System im normalen Betriebsmodus befindet und die Windgeschwindigkeit unter 2 m/s liegt, werden die Bremsen angezogen und keine Gondelbewegung wird initiiert.
- **Fehlermodus** - das WNS wechselt in diesen Zustand, wenn ein Fehler aufgetreten ist. Gegibt sich das WNS einmal in diesen Zustand kann es nicht mehr diesen Zustand verlassen es sei denn es wird ein Neustart mit einem Reset vorgenommen.

Für die Regelung der Bremsen und des Azimut-Antriebs benötigt das WNS Informationen über die aktuellen Zustände einiger Umgebungskomponenten. Zu den Umgebungskomponenten eines WNS gehören:

- Azimutmotoren (im Modell des Fallbeispiels existiert nur ein Motor)
- Bremsen

- Wind
- Sicherheitsschalter
- Schaltpult für manuelle Steuerung
- ggf. Fernsteuerung (nicht im Fallbeispiel berücksichtigt)

Die Informationen über die aktuellen Zustände der Umgebungskomponenten erhält das WNS über Sensoren oder direkt von den Umgebungskomponenten. Die Sensoren oder Umgebungskomponenten geben die Informationen in Form von analogen Signalen an das WNS weiter. Das WNS verarbeiten die Kombination der Eingänge und fängt dementsprechend an zu regeln. Die Informationen die das WNS von den Sensoren über die Umgebungskomponenten erhält sind folgende:

- Drehzahl des Azimutmotors
- aktuelle Position (Drehwinkel) der Gondel
- Windrichtung
- Windgeschwindigkeit
- Position des Sicherheitsschalters
- Position des Schalters zur manuellen Steuerung
- Sollposition (Drehwinkel) der manuellen Steuerung
- Zustand der Bremsen (angezogen oder gelöst)

3.2. Definition der Anforderungen für eine WKA-WNS

Das Fallbeispiel eines Windnachführungssystems wurde im Rahmen dieser Arbeit entworfen. Vor dem Entwurf wurden Anforderungen definiert. Das Fallbeispiel-Modell wurde anhand der Anforderungen an das System konstruiert. Die Anforderungen beinhalten weiterhin die Informationen für unsere Tests. Daher stellen sie, zumal sie noch nicht formalisiert sind, ein mentales Modell unserer Anforderungen dar, gegen die das SuT getestet werden soll. Nicht gegen alle Anforderungen kann getestet werden. Im Kapitel 3.4 wird erörtert, gegen welche Anforderungen in dieser Arbeit getestet wird.

In den Anforderungen werden die Schnittstellen und ihre Randbedingungen definiert. Die Schnittstellen sind aufgeteilt in Eingängen und Ausgängen. Weil alle Daten, die über die Schnittstellen laufen, analoge Signale sind, wird allen Schnittstellen der Datentyp double im Modell zugeordnet. Die Schnittstellen sind wie folgt definiert:

3.2.1. Eingänge

WindrichtungEin Ein Windrichtungssensor auf der Gondel ermittelt die Richtung aus der der Wind aktuell weht. Die eingehenden Werte sind über einen Spannungsbereich von 1 bis 2 V definiert. Dieser Wertebereich ist linear auf den Bereich von 0 bis 360 Grad abgebildet.
Definitionsbereich: [1 2] V auf [0 360] Grad

WindgeschwindigkeitEin Ein Anemometer misst die aktuelle Windgeschwindigkeit. Der Sensor misst Windgeschwindigkeiten von 0 bis 50 m/s. Diese Werte werden linear auf den Bereich zwischen 1 und 3,5 V abgebildet.
Difinitionsbereich: [1 3,5] V auf [0 50] m/s

SicherheitsabschaltungEin Die WKA verfügt über eine Sicherheitsabschaltung. Die Sicherheitsabschaltung muss permanent 3 bis 5 V aufweisen, sonst ist sie aktiviert. So wechselt die Anlage in einen sicheren Zustand, falls einen Kabelbruch oder ähnliches entsteht. Bei 0 bis 1 V ist wechselt das System in einen sicheren Zustand. Dazwischen ist eine Hysterese eingebaut, die plötzliche Spannungsschwankungen abfängt, um ungewollte Zustandswechsel zu vermeiden.
Definitionsbereich: [0 5] V auf [AN AUS]

ManuellerBetriebEin Das WNS kann über diese Schnittstelle in den Zustand für den manuellen Betrieb versetzt werden. Zwischen 4 und 6 V bleibt das WNS im normalen Betrieb, sofern die Sicherheitsabschaltung nicht aktiviert ist. Wie bei der Sicherheitsabschaltung ist eine Hysterese zwischen 2 bis 4 V. Liegt auf diesem Eingang aber ein Wert zwischen 1 bis 2 V und die Sicherheitsabschaltung ist nicht aktiviert (AUS), wechselt das WNS in den manuellen Betrieb.
Definitionsbereich: [1 6] V auf [AN AUS]

ManuellePositionEin In dem Fall, dass das WNS im manuellen Betrieb ist, gibt dieser Eingang den Wert für die Sollposition an, in die die Gondel regeln soll. Der Eingang erhält Werte im Spannungsbereich zwischen 1 und 2 Volt. Dieser ist wie bei dem Eingang "WindrichtungEin" auf den Wertebereich von 0 bis 360 linear abgebildet.
Definitionsbereich: [1 2] V auf [0 360] Grad

AktuellePositionEin Ein Sensor am Drehkranz der Gondel ermittelt die aktuelle Drehwinkelposition der Gondel. Somit erhält das WNS mit diesem Wert und einem Sollwert die Differenz, die es auszuregeln gilt. Dieser Wert ist für das Regeln sehr wichtig. Der Eingang erhält Werte im Spannungsbereich zwischen 1 und 6 Volt. Diese sind linear auf den Wertebereich von 0 bis 360 Grad abgebildet.
Definitionsbereich: [1 6] V auf [0 360] Grad

BremsenAngezogenEin Über diesen Eingang bekommt das WNS die Information, ob die Bremsen angezogen oder gelöst sind. Die Werte geben nur Aufschluss über die Endzustände, aber nicht über den Vorgang, während die Bremsen lösen bzw. anziehen.

Die eingehenden Werte liegen im Spannungsbereich von 0 bis 1 Volt. Der Wertebereich zwischen 0 und 0.8 Volt bedeutet, die Bremsen sind angezogen. Wenn die Spannung höher als 0.8 Volt ist, sind die Bremsen gelöst.
Definitionsbereich: [0 1] V auf [Angezogen Gelöst]

DrehzahlEin Die aktuelle Drehzahl des Azimutmotors erhält das WNS über diesen Eingang. Der Eingang hat einen Spannungsbereich von 0 bis 1 V. Die eingehenden Werte sind auf den Wertebereich von 0 bis 20 u/min linear abgebildet.
Definitionsbereich: [0 1] V auf [0 20] u/min

ResetEin Das WNS hat einen Fehlermodus, der nur verlassen werden kann, wenn an diesem Eingang eine Spannung zwischen 0.5 und 1 V angelegt wird.
Definitionsbereich: [0 1] V auf [Nichts Zurücksetzen]

3.2.2. Ausgänge

MotorAnsteuerungAus Das WNS steuert mit diesem Ausgang den Azimutmotor. Der Ausgang kann von -12 bis +12 Volt Spannung liefern. Legt das WNS genau 0 V Spannung an, soll der Motor nichts machen. Zwischen 0 bis 12 Volt Spannung soll der Motor rechts drehen (positiv). Je größer der Betrag der Spannung an den Motor ist, desto mehr Drehmoment soll der Motor entwickeln. Ist die Spannung negativ, soll der Motor nach links drehen. Auch hier gilt, je größer der Betrag der Spannung ist, desto mehr Drehmoment soll der Motor entwickeln. So kann das WNS fein regeln.
Definitionsbereich: [-12 0 +12] V auf [max_links stopp max_rechts]

BremsenAnsteuerungAus Über diesen Ausgang steuert das WNS die Bremsen. Der Ausgang hat einen Spannungsbereich von 0 bis 12 Volt. Von 0 bis 2 V sollen die Bremsen anziehen. Bei einem Kabelbruch würde das System die Bremsen anziehen, also in einen sicheren Zustand übergehen. Bei Werten über 2 V sollen die Bremsen gelöst werden.
Definitionsbereich: [0 12] V auf [Anziehen Lösen]

FehlerAus An diesem Ausgang ist ein Spannungsbereich von 0 bis 1 Volt definiert. Wenn das WNS eine Spannung größer 0 anlegt, befindet das WNS sich im Fehlermodus.
Definitionsbereich: [0 1] V auf [AUS AN]

In den Anforderungen sind weiterhin auch die Funktionen und Randbedingungen definiert, die das System erbringen muss. So soll das WNS zum Beispiel mit einem konkreten Motor zusammen in einer gewissen Zeit die Gondel ausregeln können. Eine weitere Anforderung ist die Fähigkeit in verschiedene Modi bzw. Zustände wechseln zu können. Das WNS soll bei plötzlichen Windböen oder anderen kurzzeitigen Schwankungen der Windrichtung oder Windstärke das Regeln vermeiden, wenn zu die Gondel zu dem Zeitpunkt ruht. Das System muss Fehlfunktionen (Kabelbruch, Spannungsschwankung) oder nicht relevante Daten erkennen und interpretieren können.

Eine detaillierte Auflistung der Anforderungen in Tabellenformat ist im Anhang A zu finden.

3.3. Modell des Fallbeispiels

Das Fallbeispiel ist ein Simulationsmodell, welches in Matlab/Simulink beschrieben ist. Ein Simulationsmodell besitzt die Möglichkeit ausgeführt zu werden. Nach [20] ist ein Simulationsmodell ein Modell, welches sich nach einer kontrollierten Menge an Eingaben in den spezifizierten Aspekten wie ein vorgegebenes System verhält. In der Industrie werden Simulationsmodelle meist in der Entwicklungsphase eines Produktes benutzt, um vorab Ergebnisse zu erzielen, verschiedene Strategien von Verhalten vorab zu evaluieren oder wenig bis keine Ressourcen zu verbrauchen. Weiterhin beleuchten die Modelle nur bestimmte Aspekte der Wirklichkeit, sodass man durch die Abstrahierung gewisse Eigenschaften eines Systems leichter entwickeln oder testen kann, ohne unter Umständen andere Aspekte oder Eigenschaften berücksichtigen zu müssen.

Das Simulationsmodell in Matlab/Simulink ist ein grafisches Modell, welches ein mathematisches Modell beschreibt. Ein grafisches Modell ist laut [20] ein symbolisches Modell. Ein symbolisches Modell beschreibt seine Eigenschaften mit Symbolen. In einem grafischen Modell werden die Symbole in Diagrammen notiert. Das mathematische Modell hinter dem grafischen Modell kann mit Formeln und Beschreibungen der Eigenschaften der Modellelemente im grafischen Modell angereichert werden. Ein sehr positiver Aspekt der Simulationsmodelle von Matlab/Simulink ist die zeitliche Ausführbarkeit. Mithilfe des mathematischen Modells von Matlab/Simulink kann zu jedem Zeitpunkt x die Werte der Eigenschaften im Modell berechnet werden. Somit sind beispielsweise zeit-kontinuierliche Systeme einfach modellier- und ausführbar.

Im Kontext dieser Arbeit wird der Begriff Simulation als Nachahmung der Operationen und des Verhaltens eines realen Systems über der Zeit verstanden. Als Nachahmung ist das Ausführen des Simulationsmodells gemeint. Das Verhalten des Fallbeispiel-Systems ist dementsprechend nicht nur von der Menge an Eingaben abhängig, sondern berücksichtigt eine weitere Dimension, nämlich die Zeit. Somit sollte bei einem Test das erwartete Verhalten in Abhängigkeit des Zeitpunktes der Überprüfung stehen.

Beim Testen eines solchen zeit-kontinuierlichen Modells bedarf es einer Umgebung, die die Zeit simuliert. Matlab/Simulink hat eine Simulationsumgebung, die das übernimmt und die Werte der Eigenschaften des Modells entsprechend einer Zeit berechnet. In dieser Arbeit wird zwischen drei Zeiten unterschieden.

physikalische Zeit diese Zeit entspricht der realen Zeit eines Verhaltens, welches das Simulationsmodell darstellt

Simulierzeit die Simulierzeit ist die Abbildung der physikalischen Zeit im Simulationsmodell

Simulationszeit die Simulationszeit ist die reale Zeit, die ein Rechner für eine Simulation braucht

Beim Testen eines reaktiven Systems, wie das Fallbeispiel eines ist, bedarf es spezieller Testmethoden. Reaktive Systeme observieren bestimmte Umgebungssysteme und geben entsprechend der Beobachtungen Befehle an Umgebungssysteme weiter. Sie interagieren mit ihrer Umgebung. Daher sollte beim Testen darauf geachtet werden, dass die Umgebung beim Testen in irgendeiner Art und Weise mit emuliert wird, damit das SuT sich richtig verhalten kann. Übertragen auf unser Fallbeispiel heißt das, dass eine Rückkopplung des Verhaltens einiger Umgebungssysteme, wie das des Azimutmotors oder der Bremsen, mit im Test eingebunden werden müssen. Zum besseren Verständnis sei folgendes Beispiel gegeben:

Befindet sich das Windnachtführungssystem im Normal Betrieb und erhält eine neue Windrichtung, muss das WNS, falls die Azimutbremsen im angezogenen Zustand sind, vorerst die Bremsen lösen, bevor es den Azimutmotor regeln darf, um die Gondel in die neue Windrichtung zu drehen. In einem Testfall für diese recht häufig vorkommende Situation muss auf das Signal "Bremsen lösen" des SuT ein Signal "Bremsen gelöst" an den Eingang BremsenAngezogenEin des SuT folgen, damit das SuT anfängt, den Motor zu regeln. Ohne diese Rückkopplung sollte das SuT nicht regeln und damit wäre ein Test auf das Regler-Verhalten nicht möglich. Weiterhin braucht das SuT eine Rückkopplung der aktuellen Position des Drehwinkels der Gondel um richtig Regeln zu können. Das heißt, dass die Motorsteuerbefehle am Ausgang MotorAnsteuerungAus des SuT verarbeitet und daraus generierte Signale als Rückkopplung an den Eingang AktuellePostitionEin geschickt werden müssen, welche als Wert die neu berechnete Position der Gondel enthalten.

Das Simulationsmodell des SuT verwendet Matlab/Simulink als Laufzeitumgebung. Diese Umgebung kann auch als Umgebung für die Umgebungssysteme verwendet werden. Es gibt mehrere Möglichkeiten das Verhalten der Umgebungssysteme in den Test einzubinden. Zum einem kann ein explizites Modell der Umgebungskomponenten/systeme erstellt werden. Dafür würde sich beispielsweise ein zusätzliches Matlab/Simulink-Modell anbieten. Dieses Modell muss nachher in die Testumgebung in geeigneter Art und Weise eingebunden werden. In der Testumgebung müssen demnach Möglichkeiten bestehen, die Ein- und Ausgänge der Matlab/Simulink-Modelle miteinander logisch zu verbinden, um einen MiL-Testaufbau zu gewährleisten. Eine übergeordnete Laufzeitumgebung für die Testumgebung muss hierfür gewährleisten, dass alle Komponenten im Test (SuT-Modell, Modelle der Umgebungskomponenten, Testtreiber) zeitlich gleich ablaufen und keine Verzögerungen in der Kommunikation zwischen den Komponenten besteht. Bei einer Echtzeit-Simulation würde das heißen, dass die Laufzeitumgebung der Testumgebung echtzeitfähig ist und ausreichend schnell ist, um die Simulationszeiten der einzelnen Komponenten im Test nicht zu verletzen und die Echtzeit-Kommunikation zu unterstützen.

Eine zweite Variante ist, die Umgebungskomponenten in das Modell des SuT mit einzubauen. Der Vorteil dieser Variante ist, dass die Kommunikation des SuT mit den Umgebungskomponenten schon innerhalb des Modells festgelegt ist. Möchte man aber bei dieser Variante die Umgebungskomponenten in bestimmte Zustände versetzen oder manipulieren, sollte darauf geachtet werden, dass die Eigenschaft der Beobachtbarkeit und Reproduzier-

barkeit nicht verletzt wird. Die Komponenten des SuT und der Umgebungskomponenten sollten in dem Modell strikt getrennt sein. In der Abb. 9 sieht man die hierarchische Trennung der Komponenten zueinander. Die Komponente “MotorMockup” ist das Teilmodell für die Umgebungskomponten und die Komponente “WKA_Gondelnachführungssystem” das Teilmodell des SuT. Diese Komponenten sind voneinander getrennt. Das Teilmodell für das SuT beispielsweise darf nicht direkt manipuliert werden, sondern nur über die Eingänge stimuliert werden, sodass eine indirekte Manipulation über eine Manipulation an der Komponente “MotorMockup” möglich ist.

In dieser Arbeit wurde die zweite Variante gewählt, weil der Vorteil, dass die Laufzeitumgebung die Simulation der Umgebungskomponenten übernimmt, überwiegt. Es muss sich nicht um das Timing gekümmert werden und die Schnittstellen zwischen dem SuT und den Umgebungskomponenten sind schon im Matlab/Simulink-Modell definiert.

Wie man schon aus dem obigen Beispiel eines Testfalls entnehmen kann, werden für die Tests vier Umgebungskomponenten für das reaktive Verhalten im Modell gebraucht. Zum einen die Bremsen und dessen Sensorik, um das SuT darauf zu testen, ob es nicht versucht bei angezogenen Bremsen den Motor anzusteuern, und zum anderen den Motor mit dem Drehwinkelsensor, für die Positionsbestimmung der Gondel. Bei der Modellierung des Motors wurde darauf geachtet, dass das Modell des Azimutmotors wesentliche Merkmale eines Elektromotors mit einbezieht. Auf der einen Seite haben wir die mechanischen Merkmale, wie Reibungskraft und Massenträgheit, und auf der anderen Seite die elektrischen Merkmale, wie anliegende Spannung und fließender Strom.

Der modellierte Motor ist ein idealer DC-Motor. Das Modell wird gebraucht, um zu errechnen, welche Drehzahl der Motor fährt, wenn eine bestimmte Spannung anliegt. Hierbei wird beachtet, dass der Motor eine Massenträgheit besitzt und gegen eine Last zu fahren hat. Die Massenträgheit soll das Verhalten simulieren, dass der Motor nicht sofort die Soll-Drehzahl erreicht, nachdem eine Spannung angelegt wird, sondern eine gewisse Zeit braucht, die Masse zu bewegen. Weiterhin bewirkt die Massenträgheit nach Entzug der Spannung, dass der Motor noch ein Stück weit weiter dreht, was das Regeln der Gondel realistischer machen soll. Hinzu kommt noch eine Last, die der Motor zu überwinden hat. In diesem Beispiel soll die Last alle möglichen Gegenkräfte (z.B. Reibungskraft) darstellen. In dem Fallbeispielmodell wird eine konstante Last angelegt. Man könnte aber auch eine variable Last modellieren, die Abhängigkeiten mit anderen Umgebungskomponenten besitzt. Eine Möglichkeit wäre die Last davon abhängig zu machen, wie schnell sich das Windrad der Windkraftanlage dreht. Die Schwingungen der Windräder einer Windkraftanlage erzeugen auch eine gewisse Kraft, die sich auf das Nachführungssystem auswirkt.

Um das Modell des Motors besser verstehen zu können, werden folgend einige Gleichungen aufgestellt, die das Modell abbildet.

Größen:

$c\phi$ Maschinenkonstante

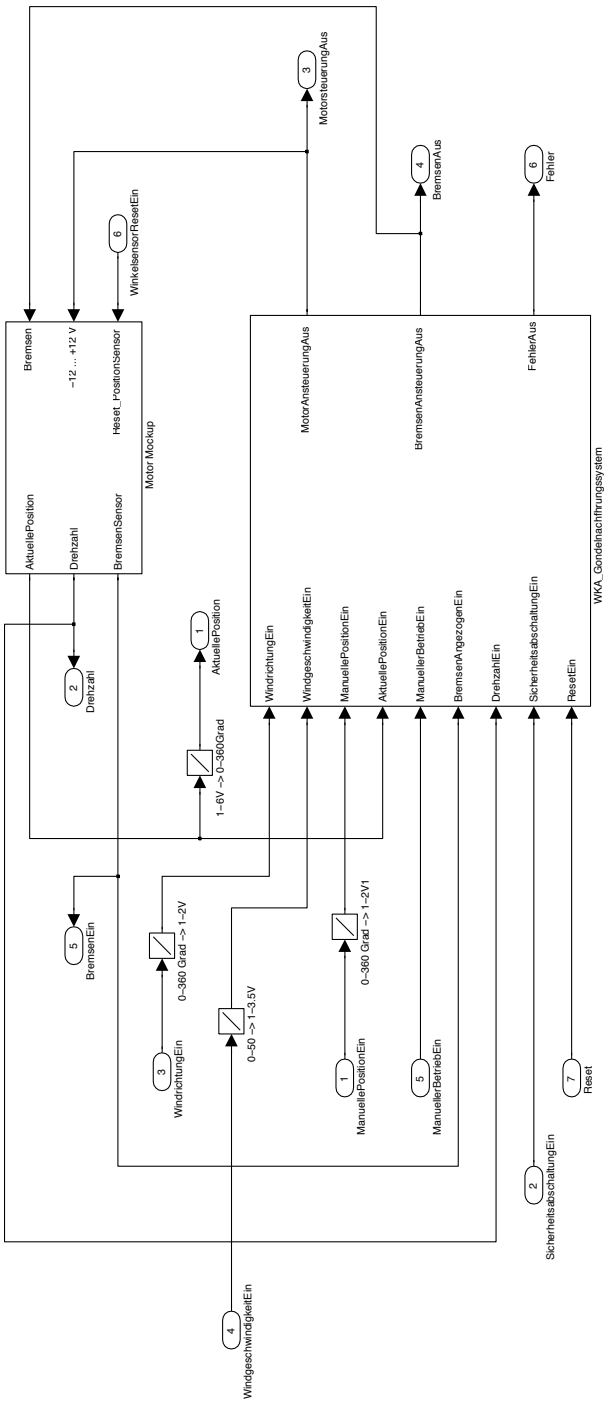


Abbildung 9: MIL Aufbau für den Test (Oberste Ebene im Simulink Modell)

M_L Lastmoment

M_{EL} aufgebrachtes Moment des Motors

I_a Massenträgheitsmoment der Welle des Motors

L_a Ankerinduktivität

R_a Ankerwiderstand

U anliegende Klemmspannung

U_i Spannung über den Motor

I fließender Strom

ω Drehzahl des Motors

Nach Kirchhoff gilt folgende elektrische Gleichung:

$$U = I * R + L_a * \frac{dI}{dt} + U_i \tag{1}$$

nach Umstellung erhält man:

$$\frac{dI}{dt} = \frac{U}{L_a} - \frac{U_i}{L_a} - \frac{I * R}{L_a} \tag{2}$$

Die Spannung über den Motor ist abhängig von der Maschinenkonstante $c\phi$ und der Drehzahl ω. Somit kann die Spannung U_i in Gleichung 2 durch die Gleichung 3 ersetzt werden.

$$U_i = c\phi * \omega \tag{3}$$

$$\frac{dI}{dt} = \frac{U}{L_a} - \frac{c\phi * \omega}{L_a} - \frac{I * R}{L_a} \tag{4}$$

Die mechanische Gleichung, die hier verwendet wurde ist in Gleichung 5 zu sehen.

$$I_a * \frac{d\omega}{dt} = \sum_n M_n = M_{EL} - M_L \tag{5}$$

$$M_{EL} = c\phi * I \tag{6}$$

Nach Umstellung der Gleichung 5 und Ersetzung des Moments des Motors M_{EL}, erhält man die Gleichung 7.

$$\frac{d\omega}{dt} = \frac{c\phi * I}{I_a} - \frac{M_L}{I_a} \tag{7}$$

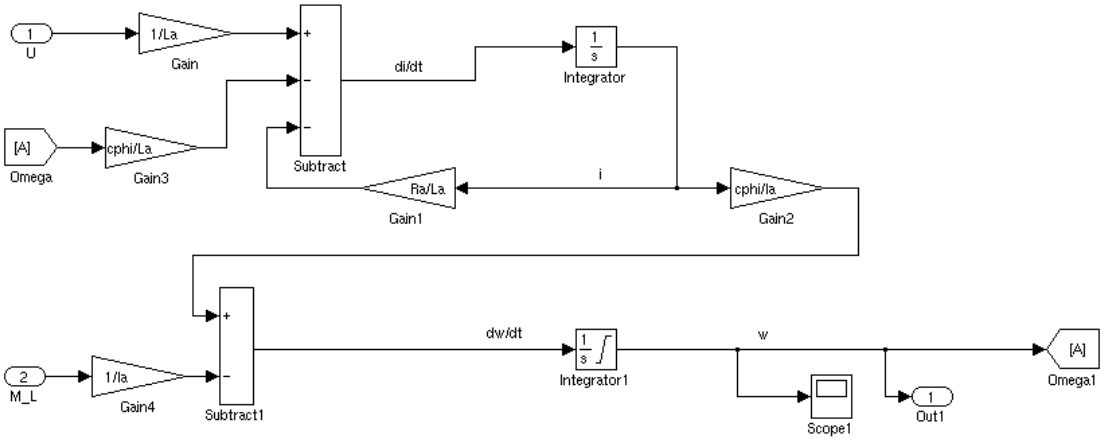


Abbildung 10: Simulink Modell der Komponente “MotorSimulation”

Die Umsetzung der Differenzialgleichungen 4 und 7 in Simulink ist in Abbildung 10 zu sehen. Das Modell stellt das Verhalten des idealen DC-Motors dar, welcher für die Tests verwendet wird.

In Abbildung 11 ist zu sehen, dass das Signal für den Befehl der Bremsen auf ein Totzeitglied läuft und von dort aus einfach wieder zurück an das SuT geschickt wird. Dieser Teil soll die Dauer der Umsetzung des Bremsenbefehls simulieren. Nebst einer konstanten Last an die MotorSimulation wird zusätzlich das Signal der Bremsen hinzu addiert, falls die Bremsen den Befehl zum Anziehen erhalten.

Die Komponente MotorSimulation errechnet die Drehzahl. Die Drehzahl wird einerseits zurück an das SuT und andererseits auf einen Verstärker geführt, der die Übersetzung des Motors auf den Drehkranz der Gondel simulieren soll. Dieser Wert wird über die Zeit integriert. Der integrierte Wert soll den aktuellen Wert des Winkelsensors darstellen. Dieser wird in das vom SuT erwartete Signal gewandelt und an das SuT in den Eingang für die aktuelle Position gegeben.

Die Mehrheit der Testfälle verlangt als Vorbedingung vordefinierte Zustände der Umgebungskomponenten, damit der Testfall korrekt durchgeführt werden kann. Daher besteht die Notwendigkeit von Funktionen, die vor, nach und mitten in der Simulation des SuT-Modells Manipulationen an den Umgebungskomponenten vornehmen können. Die Manipulationen dürfen ausschließlich auf den Umgebungskomponenten im Modell vorgenommen werden, damit

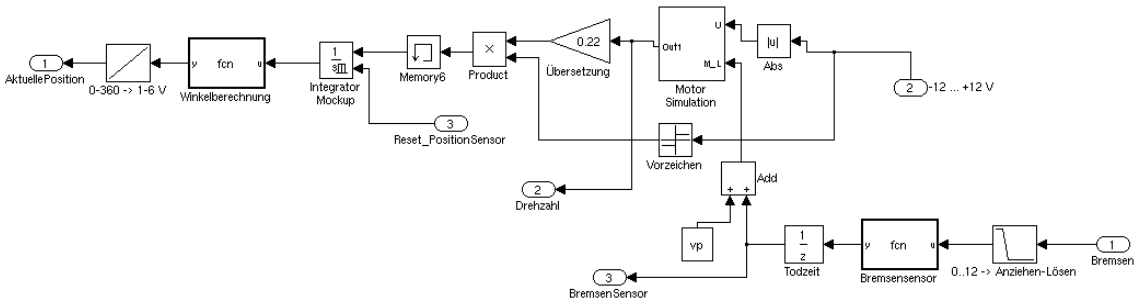


Abbildung 11: Die Umgebungskomponenten

nicht das Kriterium der "Beobachtbarkeit" verletzt wird. In dem Fallbeispiel besteht beispielsweise die Notwendigkeit, den Drehwinkelsensor auf einen bestimmten Wert zu setzen. Der Drehwinkelsensor ist in dem Mockup als Integrator realisiert. Leider erlaubt es die Laufzeitumgebung nicht, den Wert des Integrators während der Laufzeit direkt zu manipulieren. Es gibt aber zwei Methoden den Wert des Integrators auf den gewünschten Wert zu setzen.

Eine Methode wäre, den Wert des Integrators auszulesen und am Eingang des Integrators die Differenz zum gewünschten Wert anzulegen. Über einen Schalter vor dem Integrator kann dann ein Wert von außen gesetzt werden. Diese Methode braucht mindestens drei Operationen, die Leseoperation, das Schalten des Schalters und das Setzen der Differenz. Eine zweite Methode wäre das Setzen des initialen Wertes des Integrators. Jeder Integrator in Matlab/Simulink hat einen initialen Wert, den der Integrator zum Zeitpunkt null annimmt. Über eine Reset-Funktion am Integrator kann der aktuelle Wert des Integrators zur Laufzeit auf diesen gesetzt werden. Verändert man den initialen Wert des Integrators und setzt von außen den Reset, verändert man so dessen aktuellen Wert. Diese Methode braucht nur zwei Operationen und muss keine Berechnungen für die Wertedifferenz durchführen. Für das Setzen des Wertes des Drehwinkelsensors wurde in dieser Arbeit die zweite Methode angewendet, unter anderem, weil sie durch die weniger auszuführenden Operationen schneller und einfacher zu implementieren ist (siehe Abbildung 11).

3.4. Durchzuführende Tests

Die durchzuführenden Tests lassen sich an den Anforderungen im Anhang A ableiten. Die Anforderungen beziehen sich auf verschiedene Eigenschaften des SuT. Einige Anforderungen beziehen sich auf die funktionalen Eigenschaften und andere auf nicht-funktionale Eigenschaften, wie zeitliche Bedingungen. Manche Anforderungen beziehen sich auf Wechselwirkungen mit der Umgebung, sodass bei der Validierung dieser Anforderung, ein Verhalten der Umgebung vorhanden sein muss. Anforderungen, die sich zugleich auf funktionale

und nicht-funktionale Eigenschaften beziehen, sollten verfeinert werden, damit zwischen den funktionalen und nicht-funktionalen Tests klar getrennt werden kann.

Es können prinzipiell alle Anforderungen getestet werden, soweit die Eigenschaften zum Testen der Anforderung im Simulationsmodell implementiert sind. Viele nicht-funktionale Anforderungen sind aber sehr umständlich zu testen, weil der Aufwand für die Modellierung der zu testenden Eigenschaften zu groß ist. Hier stellt sich die Frage, wie sinnvoll es ist, diese Anforderung in einem Simulationsmodell zu testen. Als Beispiel sei eine Anforderung auf die Benutzerfreundlichkeit des Systems gegeben. Wie effektiv testet man die Erlernbarkeit oder Angemessenheit der Bedienung in einem Simulationsmodell des SuT?

Die Tests sollen sich auf die funktionalen Anforderungen beschränken. Die Anforderungen werden im Blackbox-Tests validiert. Dementsprechend wird von außen beobachtet und Eingaben vorgenommen. Die Schnittstellen des SuT-Modells sind die Beobachtungs- beziehungsweise Eingabepunkte. Es werden Methoden benötigt, die die Werte der ausgehenden Signale des SuT-Modells interpretieren können, um das Verhalten des SuT-Modells gegen das erwartete Verhalten zu testen.

Die Anforderungen im Anhang A beschreiben viele Eigenschaften des Systems. Sie fordern des Vorhandenseins von bestimmten Schnittstellen, das Vorhandensein von verschiedenen Modi und beschreiben den Wechsel zwischen den Modi. Weitere Anforderungen beschreiben das Verhalten des Systems bezüglich des momentanen Modus, in dem sich das System befindet, und andere Anforderungen an das Systems Modus übergreifend.

Die Anforderungen mit den IDs A-CTRL-014, A-CTRL-016, A-CTRL-017, A-CTRL-045, A-CTRL-047, A-CTRL-048, A-CTRL-49 und A-CTRL-036 beschreiben den Wechsel zwischen den Modi. In diesen Anforderungen kann entnommen werden, welche Signalwerte an den Eingängen des SuT-Modells angelegt werden müssen, um das SuT-Modell in einen bestimmten Modus zu versetzen. Davon ausgenommen wurde der Fehlermodus, weil dieser Modus einen speziellen internen Zustand des SuT darstellt. Anders als die übrigen Modi kann der Fehlermodus nur verlassen werden, wenn am Reset-Eingang ein Reset Wert angelegt wird. Außerdem gibt es mehr als eine Signalwert-Kombination, mit der man das SuT in den Fehlermodus versetzen kann. Diese stehen des Weiteren in Abhängigkeit mit zeitlichen Bedingungen. Die Anforderungen A-CTRL-034, A-CTRL-035 und A-CTRL-039 beschreiben den Wechsel in oder aus dem Fehlermodus.

Das Verhalten des SuT, in den verschiedenen Modi oder Modus übergreifend, wird in den Anforderungen A-CTRL-015, A-CTRL-018, A-CTRL-019, A-CTRL-021, A-CTRL-022, A-CTRL-024, A-CTRL-027, A-CTRL-028, A-CTRL-029, A-CTRL-039, A-CTRL-039, A-CTRL-040, A-CTRL-046 und A-CTRL-050 beschrieben. Es sind funktionale Anforderungen an das System. Einige der Anforderungen besitzen zeitliche Bedingungen. Diese Bedingungen beschreiben zeitliche Perioden in denen das System bestimmte Eingänge beobachtet und entsprechend der Beobachtungen reagiert. Diese Anforderungen sind nicht zu verwechseln mit nicht-funktionalen Anforderungen, wie Performanz-Anforderungen. Aus [35] geht hervor,

dass nicht-funktionale Anforderungen Attribute des funktionalen Verhaltens beschreiben, also "wie gut" bzw. mit welcher Qualität das System seine Funktionen erbringen soll. Beispiele für nicht-funktionale Anforderungen währen A-CTRL-030, A-CTRL-031 und A-CTRL-037.

Um das korrekte Verhalten, gemäß den Anforderungen, am System überprüfen zu können, sind Tests von Nöten. Die Tests können Fehlverhalten aufdecken, indem gegen die Spezifikation der Anforderungen getestet wird. Die beschriebenen Anforderungen im Anhang A genügen diesem Anspruch nicht, weil die natürliche Sprache zu komplex ist, um die Anforderungen entsprechend der Tests interpretieren zu können. Sie müssen formalisiert werden, damit gegen sie getestet werden kann.

Ein Testfall kann auf eine oder mehrere Anforderungen testen. Auf eine Anforderung kann in mehreren Testfällen getestet werden. Sind demnach alle Testfälle für eine bestimmte Anforderung erfolgreich durchlaufen (oder zumindest die definierte Mindestanzahl laut Testspezifikation), gilt diese Anforderung bzw. Funktion als validiert [35].

Die in dieser Arbeit zugrundeliegende Herangehensweise zur Ermittlung der Testfälle bedient sich einer Mischung dreier Testmethoden, dem anforderungsbasierten, dem anwendungsfallbasierten und dem zustandsbezogenen Testen. Gemäß den Anforderungen im Anhang A werden die Zustände des Systems identifiziert. Die Zustände sind in diesem Fall die Betriebsmodi. Daraufhin wird untersucht, wie diese Zustände aktiviert werden und welche Zustände mögliche Folgezustände sein können. Für jeden Zustand wird mindestens ein Testfall generiert, der einen Anwendungsfall abbildet. Durch die Parametrisierung der Testfälle und mithilfe der Äquivalenzklassenmethode zur Ermittlung der Parameterdaten werden die Testfälle verfeinert und ergänzt.

Für die Äquivalenzklassenmethode werden die Eingänge des SuT herangezogen. Alle Werte innerhalb der Wertebereiche eines Eingangs, die das gleiche Verhalten des SuT zur Folge haben, werden in eine Äquivalenzklasse getan. In der Tabelle 1 wird die Aufteilung der Äquivalenzklassen bezüglich der Eingänge des SuT gezeigt. Die Äquivalenzklassen beruhen auf den Wertebereichen, die zu den Schnittstellen definiert wurden. Aus [35] geht hervor, dass aus zusammenhängenden Wertebereichen eine oder mehrere gültige und zwei ungültige Äquivalenzklassen gebildet werden. Die ungültigen Äquivalenzklassen beschreiben Wertebereiche mit ungültigen Eingaben. Diese Eingabewerte sind für Robustheitstest interessant, weil sie im undefinierten Bereich der Schnittstelle liegen. Für unsere Testfälle reichen uns die gültigen Äquivalenzklassen, weil die Anforderungen weder eine Fehlerbehandlung noch ein spezifisches Verhalten nach ungültigen Eingaben definieren.

In weiteren Anforderungen werden zusätzliche Informationen wie Bedingungen und Abhängigkeiten zwischen den Schnittstellen beschrieben, die zu einer Verfeinerung der Äquivalenzklassen herangezogen werden können. Die Tabelle 2 zeigt die Verfeinerung der Äquivalenzklassen bezüglich der Schnittstellen. Außerdem wurden in dieser Tabelle auch Vereinfachungen vorgenommen. Im folgenden Kapitel 4 werden noch weitere Äquivalenzklassen aus den Anforderungen gebildet. Zum Beispiel wurden die Wertebereiche der Eingänge "WindrichtungEin", "AktuellePositionEin" und "ManuellePositionEin" durch die logischen

Eingänge	Äquivalenzklasse	Repräsentant
WindrichtungEin	gÄK$_{11}$: [1 , . . . , 2]	1.5
	uÄK$_{11}$: [MIN_DOUBLE , . . . , 1 [	0
	uÄK$_{12}$:] 2 , . . . , MAX_DOUBLE]	3
WindgeschwindigkeitEin	gÄK$_{21}$: [1 , . . . , 3.5]	1.5
	uÄK$_{21}$: [MIN_DOUBLE , . . . , 1 [	0
	uÄK$_{22}$:] 3.5 , . . . , MAX_DOUBLE]	3
SicherheitsabschaltungEin	gÄK$_{31}$: [0 , . . . , 1]	0
	gÄK$_{32}$:] 1 , . . . , 3 [	2
	gÄK$_{33}$: [3 , . . . , 5]	0
	uÄK$_{31}$: [MIN_DOUBLE , . . . , 0 [	-1
	uÄK$_{32}$:] 5 , . . . , MAX_DOUBLE]	6
ManuellerBetriebEin	gÄK$_{41}$: [1 , . . . , 2]	1.5
	gÄK$_{42}$:] 2 , . . . , 4 [	3
	gÄK$_{43}$: [4 , . . . , 6]	5
	uÄK$_{41}$: [MIN_DOUBLE , . . . , 1 [	0
	uÄK$_{42}$:] 6 , . . . , MAX_DOUBLE]	7
ManuellePositionEin	gÄK$_{51}$: [1 , . . . , 2]	1.5
	uÄK$_{51}$: [MIN_DOUBLE , . . . , 1 [	0
	uÄK$_{52}$:] 2 , . . . , MAX_DOUBLE]	3
AktuellePositionEin	gÄK$_{61}$: [1 , . . . , 6]	4
	uÄK$_{61}$: [MIN_DOUBLE , . . . , 1 [	0
	uÄK$_{62}$:] 6 , . . . , MAX_DOUBLE]	7
BremsenAngezogenEin	gÄK$_{71}$: [0 , . . . , 0.8]	0.2
	gÄK$_{72}$:] 0.8 , . . . , 1]	0.9
	uÄK$_{71}$: [MIN_DOUBLE , . . . , 0 [	-1
	uÄK$_{72}$:] 1 , . . . , MAX_DOUBLE]	1.2
DrehzahlEin	gÄK$_{81}$: [0 , . . . , 1]	4
	uÄK$_{81}$: [MIN_DOUBLE , . . . , 0 [	-1
	uÄK$_{82}$:] 1 , . . . , MAX_DOUBLE]	2
ResetEin	gÄK$_{91}$: [0 , . . . , 1]	4
	uÄK$_{91}$: [MIN_DOUBLE , . . . , 0 [	-1
	uÄK$_{92}$:] 1 , . . . , MAX_DOUBLE]	2

Tabelle 1: Äquivalenzklassen nach der Spezifikation der Eingänge

Eingänge	Äquivalenzklasse	Repräsentant
WindrichtungEin	gÄK$_{11}$: [0 , . . . , 135 [	90
	gÄK$_{12}$: [135 , . . . , 225]	180
	gÄK$_{13}$:] 225 , . . . , 360 [	270
WindgeschwindigkeitEin	gÄK$_{21}$: [1 , . . . , 3.5]	1.5
SicherheitsabschaltungEin	gÄK$_{31}$: [0 , . . . , 1]	0
	gÄK$_{32}$:] 1 , . . . , 3 [	2
	gÄK$_{33}$: [3 , . . . , 5]	0
ManuellerBetriebEin	gÄK$_{41}$: [1 , . . . , 2]	1.5
	gÄK$_{42}$:] 2 , . . . , 4 [	3
	gÄK$_{43}$: [4 , . . . , 6]	5
ManuellePositionEin	gÄK$_{51}$: [0 , . . . , 135 [	90
	gÄK$_{52}$: [135 , . . . , 225]	180
	gÄK$_{53}$:] 225 , . . . , 360 [	270
AktuellePositionEin	gÄK$_{61}$: [0 , . . . , 135 [	90
	gÄK$_{62}$: [135 , . . . , 225]	180
	gÄK$_{63}$:] 225 , . . . , 360 [	270
BremsenAngezogenEin	gÄK$_{71}$: [0 , . . . , 0.8]	0.2
	gÄK$_{72}$:] 0.8 , . . . , 1]	0.9
DrehzahlEin	gÄK$_{81}$: [0]	0
	gÄK$_{82}$:] 0 , . . . , 1]	0.5
ResetEin	gÄK$_{91}$: [0 , . . . , 0.5 [	0
	gÄK$_{92}$: [0.5 , . . . , 1]	1

Tabelle 2: Verfeinerung der Äquivalenzklassen aus der Tabelle 1

Werte ersetzt (von Volt auf Grad), um die Berechnungen und Nachvollziehbarkeit der Testfälle zu vereinfachen. An dieser Stelle darf das gemacht werden, weil die Robustheitstests wegfallen. Die logischen Werte dürfen nicht hergenommen werden, wenn die ungültigen Eingaben ebenfalls getestet werden sollen, weil diese nicht in den logischen Werten enthalten sind und dadurch nicht in einem Robustheitstest berücksichtigt werden. Für jeden dieser Eingänge wurden weiterhin neue Äquivalenzklassen aus den Anforderungen bezogen. Aus den Anforderungen A-CTRL-022 und A-CTRL-024 geht beispielsweise hervor, dass es einen verbotenen Bereich für den Gondelwinkel gibt (zu sehen in Abb. 8). Der numerische Wertebereich beginnt mit der 0 und endet mit 360. Dazwischen befindet sich der verbotene Bereich, sodass drei Äquivalenzklassen daraus abgeleitet werden, der Wertebereich vor dem verbotenen Bereich, der Wertebereich nach dem verbotenen Bereich und der verbotene Wertebereich.

Die Testfälle ergeben sich aus den Kombinationen der Eingabewerte. Erzeugt man nun eine Kombination der Eingänge über die Äquivalenzklassen, erhält man gemäß der Tabelle 2 3*1*3*3*3*3*2*2*2 = 1944 Testfälle. Angenommen die Dauer des Durchlaufs eines Testfalls beträgt im Durchschnitt 30 Sekunden, wäre folgende Rechnung gegeben:

$$\frac{1944 * 30}{60 * 60} = 162 \tag{8}$$

Der Testdurchlauf aller Testfälle würde dementsprechend 162 Stunden (ca. 7 Tage) lang laufen. Durchaus lässt sich solch ein Test mit dieser Dauer durchführen, aber es wäre zum größten Teil verschwendete Zeit, zumal es Möglichkeiten zur Optimierung und Filterung der Testfälle gibt, die die Gesamtdauer stark reduzieren würden.

In den verschiedenen Modi reagiert das SuT nur auf bestimmte Eingänge. Dadurch erhält man, entsprechend des Modus, Filterungsoptionen auf den Äquivalenzklassen. Nur die relevanten Daten bzw. Eingänge werden in der Kombinatorik der Äquivalenzklassen berücksichtigt. Die Berücksichtigung von nicht relevanten Eingangswerten wäre nur sinnvoll, wenn man Negativtests wie Robustheitstests durchführen möchte, um zu untersuchen, ob das SuT auch wirklich nicht auf nicht-relevante Daten bzw. Eingangswerte reagiert. So ist beispielsweise der Eingang "ManuellePositionEin" in einem Testfall, der das Verhalten im normalen Betriebsmodus testet, nicht von Relevanz, weil das SuT in diesem Modus nicht auf die eingehenden Signale an diesem Eingang reagieren sollte.

Über Regeln auf den Testfällen lässt sich die Anzahl der Testfälle zusätzlich reduzieren. Eine sinnvolle Regel wäre die Priorisierung der Testfälle nach bestimmten Kriterien, wie Häufigkeit des Vorkommens eines Anwendungsfalls, nach Signifikanz der Anforderung oder Sicherheitsrelevanz. Diese Maßnahme reduziert noch nicht die Anzahl der Testfälle, aber hilft bei der Auswahl nach Prioritäten. In Verbindung mit einem Testsendekriterium, welches so definiert ist, dass der Äquivalenzklassen-Überdeckungsgrad kleiner als 100% beträgt, würden vorrangig die höher priorisierten Testfälle durchgeführt und die nieder-prioren Testfälle ausgelassen.

Bisher wurde für die Gewinnung der Testfälle das Kreuzprodukt der Parameter bzw. Eingänge über die Äquivalenzklassen berechnet. Die paarweise Kombination der Parameter über die Äquivalenzklassen würde die Anzahl der Testfälle stark reduzieren. Ein Minimalkriterium für die Kombinatorik hätte in der Regel eine noch stärkere Reduzierung zu Folge. Ein Minimalkriterium könnte lauten, dass eine Äquivalenzklasse oder die Äquivalenzklassen eines Parameters in mindestens einem Testfall vertreten sein müssen.

Vor allem sollten aber Testfälle mit Grenzwerten bevorzugt werden. Nach [35] sind Grenzwerte von Äquivalenzklassen fehlerträchtige Fallunterscheidungen im SuT. Daher ist es sinnvoll, für die Auswahl der Repräsentanten einer Äquivalenzklasse, einer Untersuchung auf Grenzwerte vorzunehmen.

Die Grenzwertanalyse ergänzt sich gut mit der Äquivalenzklassenmethode. Innerhalb einer Äquivalenzklasse lassen sich Grenzwerte entnehmen. Die Äquivalenzklassen in Tabelle 2 definieren einen Wertebereich. Mittels der Grenzwertanalyse erhält man fünf Repräsentanten für solche Äquivalenzklassen, jeweils 2 Repräsentanten für die Bereichsgrenzen (einen Wert innerhalb und einen außerhalb des Wertebereichs) und einen in der Mitte. Wie schon weiter oben angeführt, interessieren die ungültigen Werte nicht. Somit fallen die 2 Repräsentanten jeweils außerhalb des Wertebereichs weg. Auch bei zwei angrenzenden gültigen Äquivalenzklassen fallen diese weg, weil diese Werte schon als innerer Grenzwert-Repräsentant der jeweils anliegenden Äquivalenzklasse definiert sind.

Alle Schnittstellen des SuT sind als Datentyp DOUBLE definiert. Bei Fließkommazahlen wird für die Grenzwertanalyse eine Schrittweite bzw. Rechengenauigkeit benötigt, um die Grenzwerte zu ermitteln. An dieser Stelle wird die Rechengenauigkeit δ (an vielen Stellen auch StepSize genannt) eingeführt. Sie gibt den Abstand zwischen Grenzwert und Grenze an. Die Rechengenauigkeit wird auch an den Stellen verwendet, wo Wertebereiche definiert werden die aneinander grenzen. Die oben angeführten Tabellen mit den Äquivalenzklassen wären ein Beispiel dafür. Ist ein Wertebereich mit den eckigen Klammern nach außen definiert, so heißt das für die untere Grenze Wert-δ und für die obere Grenze Wert+δ.

3.4.1. Funktionale Tests

Zur Übersichtlichkeit dieses Dokuments wird die Anzahl der zu diskutierenden Testfälle beschränkt. Für jeden Betriebsmodus wird mindestens ein Testfall benötigt, um zu verifizieren, dass das SuT sich in diesem Modus gemäß den Anforderungen verhält. Ein Testfall, der für dieses Dokument für die Beispiele herangezogen wird, ist der Test des normalen Betriebsmodus.

Der normale Betriebsmodus umfasst die Ansteuerung und Regelung der Bremsen und des Motors. Dabei werden die Eingangs-Informationen für die Windrichtung, die Gondelausrichtung (Drehwinkel), die Drehzahl des Motors und der Zustand der Bremsen benötigt. Alle anderen Eingänge sind laut der Anforderungen für diesen Testfall nicht von Belangen oder

als Vorbedingung definiert. In der Vorbedingung der Testfälle für den normalen Betriebsmodus ist die Windgeschwindigkeit, die Position des Schalters für den manuellen Betrieb und die Position des Sicherheitsschalter von Bedeutung.

Die Testfälle für den normalen Betriebsmodus müssen auf die Anforderung A-CTRL-022, A-CTRL-024, A-CTRL-027, A-CTRL-028, A-CTRL-029, A-CTRL-031, A-CTRL-032 und A-CTRL-038 testen. Ausgenommen sind an dieser Stelle die Anforderungen für das Vorhandensein einer Schnittstelle. Diese Anforderungen werden als gegeben für diese Tests angesehen und als Deklaration der Schnittstellen hergenommen. Die wichtigen Aspekte sind die Regelung der Gondel in den Wind und die Ansteuerung der Bremsen.

Die Regelung der Gondel-Ausrichtung ist in den Tests ein wichtiger Aspekt. Die Regelung wird insofern getestet, dass mit verschiedenen Eingangswerten der Windrichtung und der aktuellen Gier-Position stimuliert wird. Das erwartete Verhalten wird in den Testfällen neu aus den Anforderungen geschlossen, um zwei Verhalten gegeneinander testen zu können, das erwartete Verhalten und das Verhalten des SuT. Wie das erwartete Verhalten in den Modellen kodiert wird, ist im nächsten Kapitel beschrieben.

Die Fehlerbehandlung des SuT ist in den Testfällen direkt eingebunden und wird nicht als gesonderter Betriebsmodus behandelt, um die Auswirkungen in den internen sicherheitsrelevanten Zuständen des SuT zu überprüfen. Für die Testfälle des normalen Betriebsmodus wurden daher die Anforderungen A-CTRL-034, A-CTRL-035 und A-CTRL-039 des Fehlermodus herangezogen. Um die Fehlerereignisse zu provozieren, werden die Umgebungskomponenten während des Tests manipuliert. Das erwartete Verhalten des Fehlermodus wird dementsprechend im Testfall für den normalen Betriebsmodus überprüft.

4. Methodische Aspekte des modellbasierten Testens

In diesem Kapitel wird eine Methodologie für das modellbasierte Testem mit SysML vorgestellt. Dabei wird das Konzept des Aufbaus des Testmodells und Algorithmen zum Erschließen von Informationen für das systematische Testen diskutiert und bewertet. Anschließend wird anhand eines Beispiels gezeigt, wie ein Testmodell aufgebaut und genutzt werden kann, um Testfälle und Testkonfigurationen daraus zu gewinnen.

4.1. Konzeption

Für die Spezifikation eines vollständigen Tests werden gewisse Informationen stets benötigt. Falls ausschließlich Black-Box Tests durchgeführt werden, ist die Menge an benötigten Informationen eingeschränkt. Informationen über die Implementierung und die interne Architektur des SuT sind nicht von Bedeutung. Für die Black-Box Tests wird lediglich die System-Spezifikation des SuT für die Informationsgewinnung herangezogen.

Ein Testfall beinhaltet laut [35] folgende Informationen: "die für die Ausführung notwendigen Vorbedingungen, die Menge der Eingabewerte (ein Eingabewert je Parameter des Testobjektes) und die Menge der erwarteten Sollwerte, die Prüfanweisungen (wie Eingaben an das Testobjekt übergeben und Sollwerte abzulesen sind) sowie die erwarteten Nachbedingungen". Dabei sollten die Testfälle so gewählt werden, dass eine gewisse Wahrscheinlichkeit zur Aufdeckung einer noch nicht bekannten Fehlerwirkung besteht.

Für das Erschließen der Menge an Eingabewerten werden zuallererst die Schnittstellen des SuT untersucht. Welche Schnittstellen sind nach außen vorhanden und welche Flussrichtung haben sie? Die Schnittstellen können Input-Schnittstellen, Output-Schnittstellen und sogar beides sein. Zusätzlich ist die Beziehung der Schnittstellen zu den Umgebungskomponenten wesentlich. Welche Beziehung und Anbindung haben sie zueinander? Für die Interpretation der Kommunikation wird außerdem eine Datenspezifikation der Schnittstellen benötigt.

Die Datenspezifikation gibt an, welche Art von Daten oder Signalen kommuniziert werden, um welche Datentypen oder Datenstrukturen es sich handelt und welche Wertebereiche sie definieren. Sofern es möglich ist, ist es sinnvoll die Daten- oder Signalwerte in Äquivalenzklassen zu unterteilen (siehe Kapitel 3.4).

Für das Erschließen eines Testfalls bezüglich ein oder mehrerer Testkriterien, erfordert es einer Funktion, die das Sollverhalten des SuT hinsichtlich der Eingabewerte berechnet. Diese Funktion wird in dieser Arbeit als Soll-Funktion bezeichnet. Das Sollverhalten wird mit dem Ist-Verhalten des SuT verglichen, um eine Aussage hinsichtlich des Ergebnisses des Tests treffen zu können.

Die Funktion des Sollverhaltens wird üblicherweise beim MBT in Verhaltensmodellen abgebildet. Unter Umständen ist es brauchbar, das Verhalten der Umgebungskomponenten in den Verhaltensmodellen hinein zu kodieren, damit das Generieren der Stimuli, also der

Eingabewerte, weitestgehend der realen Welt entspricht. Eingabewerte, die die Umgebungskomponenten gewöhnlich nicht erzeugen, sind nur für Negativtests nützlich und sollten gesondert in datengetriebenen Tests erzeugt werden.

Die Modellelemente in SysML können mit sogenannten Stereotypen um spezielle Marker und weitere Eigenschaften-Felder erweitert werden. Somit ist es auch möglich, die Modellelemente mit Daten und Werten für das MBT zu erweitern. In der Abbildung 12 sind die für diese Arbeit elaborierten Stereotypen für das MBT gezeigt. Die Stereotypen sind insoweit wichtig, damit das MBT-Modell hinreichend viele Informationen für die Generierung von Testfällen beinhaltet. In den folgenden Abschnitten werden die einzelnen Stereotypen erklärend aufgegriffen.

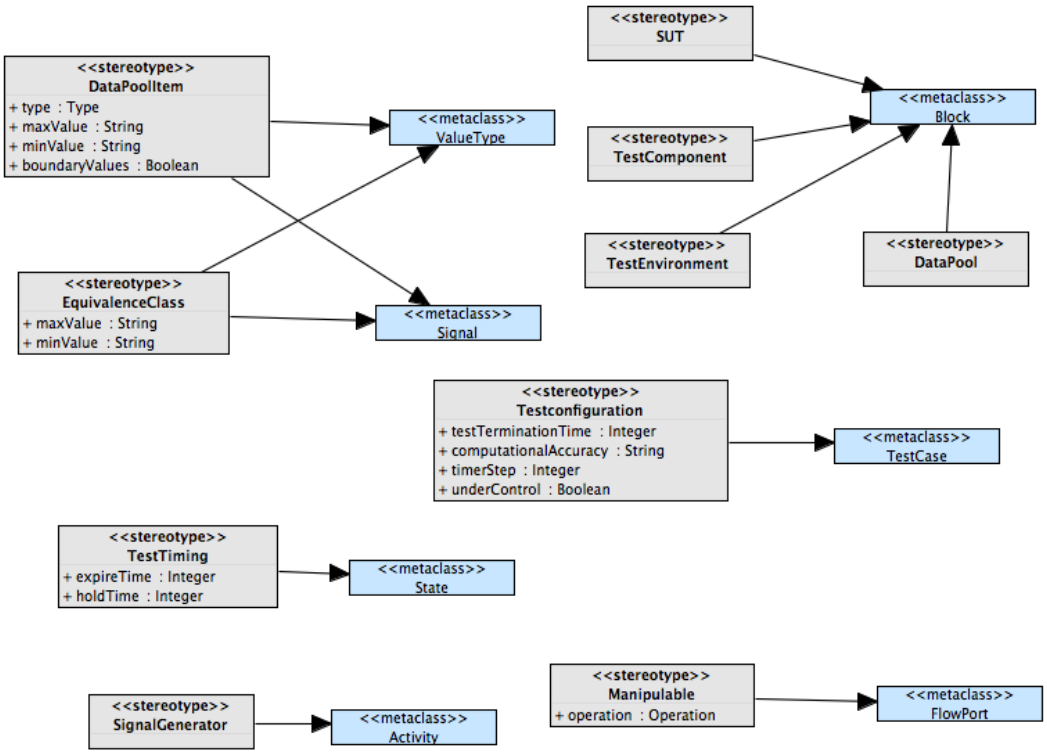


Abbildung 12: Stereotypen Erweiterung für das MBT

4.1.1. Strukturen und Testdaten mit Strukturdiagrammen

Die Testumgebung mit dem SuT und den Umgebungskomponenten lässt sich mit den Block-Definitions-Diagrammen modellieren. Durch Blöcke können die Komponenten einer Testum-

gebung dargestellt werden. Die Blöcke können Eigenschaften, Referenzen, Bestandteile, Bedingungen, Werte, Schnittstellen und Funktionen besitzen (siehe Abb. 13).

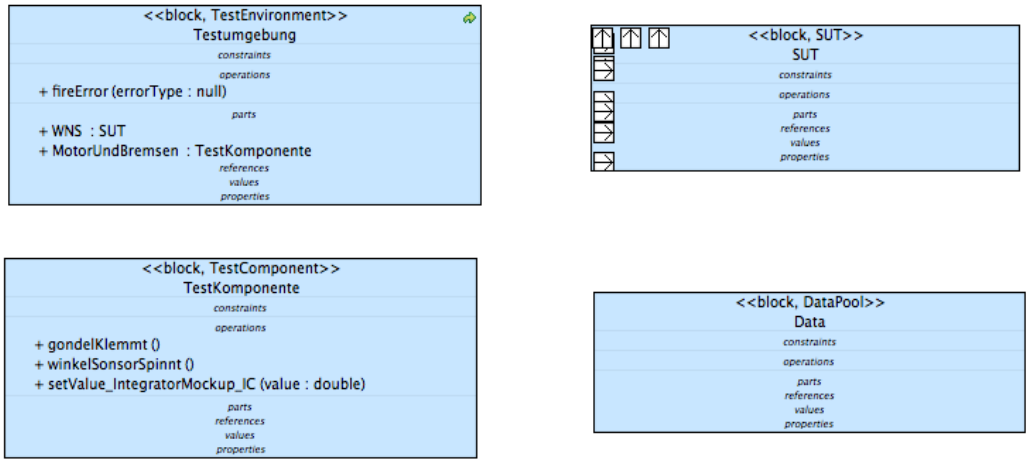


Abbildung 13: BDD der Testumgebung

Für das SuT wird ein Block mit dem Stereotypen "SuT" versehen. Dieser Block steht stellvertretend für das SuT. Für diesen Block werden die benötigten Schnittstellen definiert. Für die Modellierung der Umgebungskomponenten gibt es den Stereotypen "TestComponent". Auch für diese Blöcke werden die Schnittstellen und Funktionen, die von der Testumgebung ausgeführt werden können, definiert. Weiterhin gibt es noch einen Block mit dem Stereotypen "TestEnvironment" der die Beziehungen der Implementierungen zwischen dem SuT und den Testkomponenten darstellt. Eine Hierarchiestufe unter dem Block mit dem Stereotypen "TestEnvironment" gibt es ein Internal-Block-Diagramm, welches den Testaufbau abbildet. Hier sind die Verbindungen zwischen den Schnittstellen der Testkomponenten und dem SuT abgebildet (siehe Abb. 14).

Die zuvor definierten Blöcke sind nun als "Part"-Instanzen in der Testumgebung vertreten. Es wäre beispielsweise möglich, mehrere Instanzen von Testkomponenten desselben Blocks in den Testaufbau zu definieren, aber mehr als eine SuT-Instanz wäre wiederum nicht sinnvoll. Für die Definition der Schnittstellen können in der SysML die Ports verwendet werden. Das Dokument [14] der GFSE (Gesellschaft für Systems Engineering) stellt bewährte Modellierungsparadigmen zu SysML vor. Unter anderem sind dort auch Vorschläge zu der Modellierung von Schnittstellen gemacht worden. Im Allgemeinen wird die Art der Modellierung von der Art der Schnittstelle abhängig gemacht. Falls die Schnittstelle ein Service anbietet oder benötigt, so können für den Port Interfaces definiert werden. Weiterhin können auch die FlowSpecification Elemente als eine Erweiterung von den Interfaces verwendet werden, falls man für einen Port genau spezifizieren möchte welche Art von Daten oder Strukturen über diesen Port fließen dürfen und wie die Flussrichtung der verschiedenen Daten aussehen

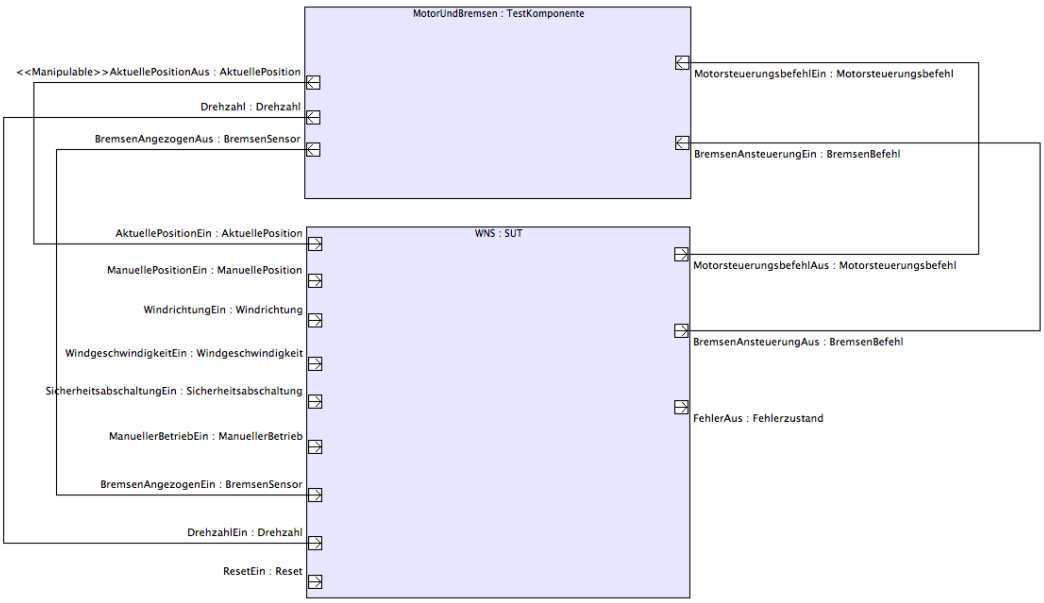


Abbildung 14: IBD der Testumgebung

darf. Die FlowSpecification Elemente werden aber zumeist verwendet, wenn es komplexere oder mehrere Daten oder Strukturen für die Schnittstelle zu definieren gilt.

In der Regel und so auch in dem Fallbeispiel dieser Arbeit reicht es die spezialisierte Form der Ports, die FlowPorts, zu verwenden. Wie in Abbildung 16 zu sehen, kann man die Flussrichtung (isAtomic muss wahr sein) direkt auf dem FlowPort setzen und ablesen. Da ein Port von dem Element Property erbt, besitzt er auch die Eigenschaft "Type", über den man nun den Datentypen definieren kann. Die Datentypen werden in einem Datenpool definiert. Der Datenpool ist eine Menge von Daten, die für das Treiben der Tests verwendet werden.

Die im Modell definierten Testkomponenten sind Stubs der Testumgebungskomponenten. Stubs sind laut [35] Platzhalter. In dieser Studie werden die Stubs als Ersatz der Umgebungskomponenten verwendet. Sie haben ein programmiertes Verhalten, das unabhängig von dem Testcontroller läuft. Somit können die Input-Schnittstellen des SuT, die mit den Testkomponenten verbunden sind, nicht seitens der Tests kontrolliert beeinflusst werden. Möchte man dennoch kontrolliert Einfluss auf die mit dem Stubs verbundenen Schnittstellen nehmen, weil beispielsweise die Werte an diesen Schnittstellen als Vorbedingung eines Testfalls gesetzt werden müssen, kann man den Stubs Funktionen zuweisen, mit denen man das Stub während der Simulation manipulieren kann. Um eine Verbindung zwischen der manipulierenden Funktion und dem zugehörigen Port bekannt zu machen, wurde ein Stereotyp "Manipulable" in der Arbeit eingeführt (siehe Abb. 12 und 14). Dieser Stereotyp wird einem Flowport zugewiesen. Er besitzt eine Eigenschaft "operation", welche eine Referenz auf eine Funktion ist.

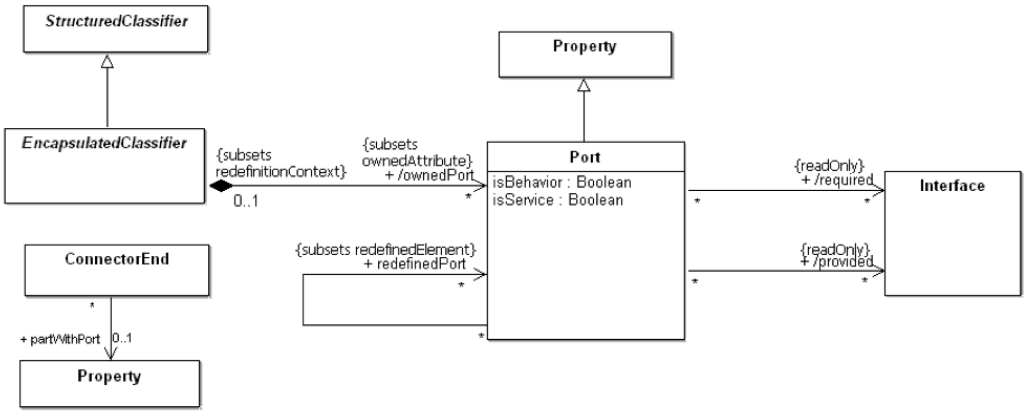


Abbildung 15: Metamodell der Ports aus der UML Spezifikation aus [27]

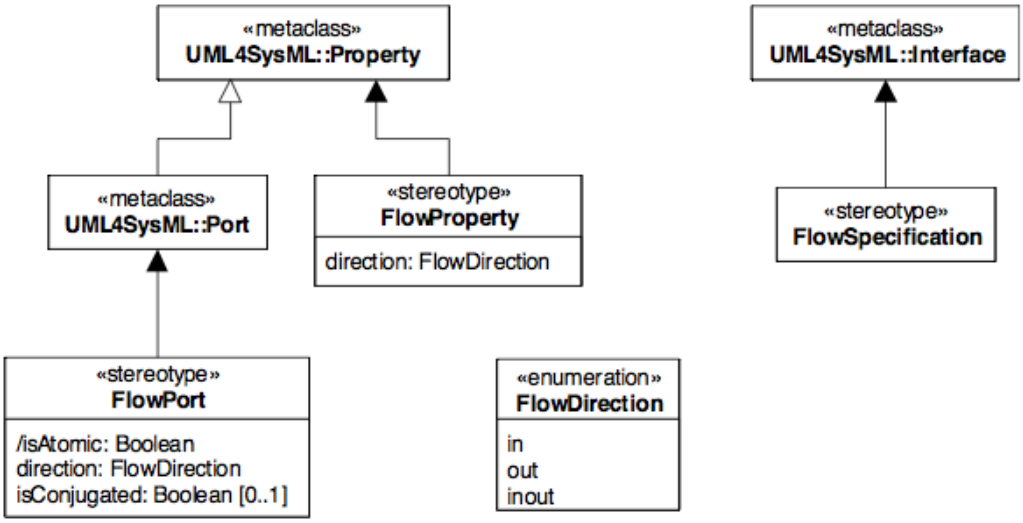


Abbildung 16: Metamodell der FlowPorts aus der SysML Spezifikation aus [26]

Die Namen der Ports des SuT im IBD (Internal-Block-Diagramm) sind die Identifikatoren für das Mapping zwischen den Schnittstellen im Testmodell und den Schnittstellen des realen SuTs. Sie müssen mit den Namen der Ports in den Simulink-Modellen übereinstimmen. Jeder Port hat einen eigenen Datentyp im Datenmodell. Somit erhält man eine Zuweisung der Ports über die Datentypen des Datenmodells (Portname des SuT-Modells <=> DatenTyp des Ports).

Für die Abbildung der Datenstrukturen gibt es den Block mit dem Stereotypen "DataPool" (siehe 13). In einer Hierarchiestufe unter diesem Block ist ein weiteres Block-Definitions-Diagramm enthalten, welches die Datenstruktur beherbergt. In diesem Datenmodell werden die speziellen Datentypen für die Ports und weitere Parameter-Variablen definiert. Die Datentypen werden mit dem "Signal" Element definiert und werden mit dem Stereotypen "DataPoolItem" versehen. Für die Definition der Datentypen können auch die Elemente "ValueType" oder "DataType" verwendet werden. Der Grund für die Wahl der Signal-Elemente ist die einfache Handhabung von Signalen in den Verhaltensdiagrammen. Für das Triggern von Ereignissen und das Senden von Signalen oder Nachrichten in den Verhaltensdiagrammen, lassen sich die Signal-Elemente sehr gut einsetzen, ohne zusätzliche Modell-Kontrukte als Container für die Datentypen einzusetzen.

Für einen Zustandsautomaten, welcher das Verhalten über Ereignisse definiert, können Trigger mit einem "SignalEvent" verwendet werden. Über ein SignalEvent kann eine Referenz über die Eigenschaft "signal" zu einem Signal geschaffen werden. Im Übrigen sind die Signal Elemente auch in Aktivitäts- und Sequenzdiagrammen unkompliziert anwendbar. In Aktivitätsdiagrammen lassen sie sich über "SendSignalAction" Elemente direkt und über "SignalEvent" Elemente von "AcceptEventAction" Elementen über einen Trigger indirekt referenzieren. In den Sequenzdiagrammen können die Signal Elemente als Inhalt einer Nachrichten kodiert werden. Der Vorteil der unkomplizierten Einbindung der Signal Elemente, in allen Verhaltensdiagrammen der SysML, wird in Kapitel 5.3 noch mehr offensichtlich.

Jedes Signal Element, das in einer Composite-Beziehung als Komponente zu den Signalen mit den Stereotypen "DataPoolItem" steht, ist eine Äquivalenzklasse dieses Wertebereichs. Die Äquivalenzklasse kann ein Wert oder ein Wertebereich sein. Diese Elemente werden wiederum mit dem Stereotypen "EquivalenceClass" versehen, um über die Eigenschaften dieses Stereotyps den Wert oder Wertebereich für die Äquivalenzklasse zu definieren (siehe Abb. 17).

Im UTP (UML Testing Profile) [4] wird eine Äquivalenzklasse von Testdaten über eine Composite-Assoziation hergestellt. Ein Testdatum wird über eine Klasse beschrieben und anhand mehrerer Assoziationen zu einer Datenpool-Klasse werden die Äquivalenzklassen erschaffen. Die Namen der Äquivalenzklassen werden über den Rollennamen der Assoziation definiert. Hierbei ist noch völlig unklar, welche Werte über die Äquivalenzklassen beschrieben werden. Das Prinzip mit den Composite-Assoziationen wurde zum Teil in dieser Arbeit übernommen.

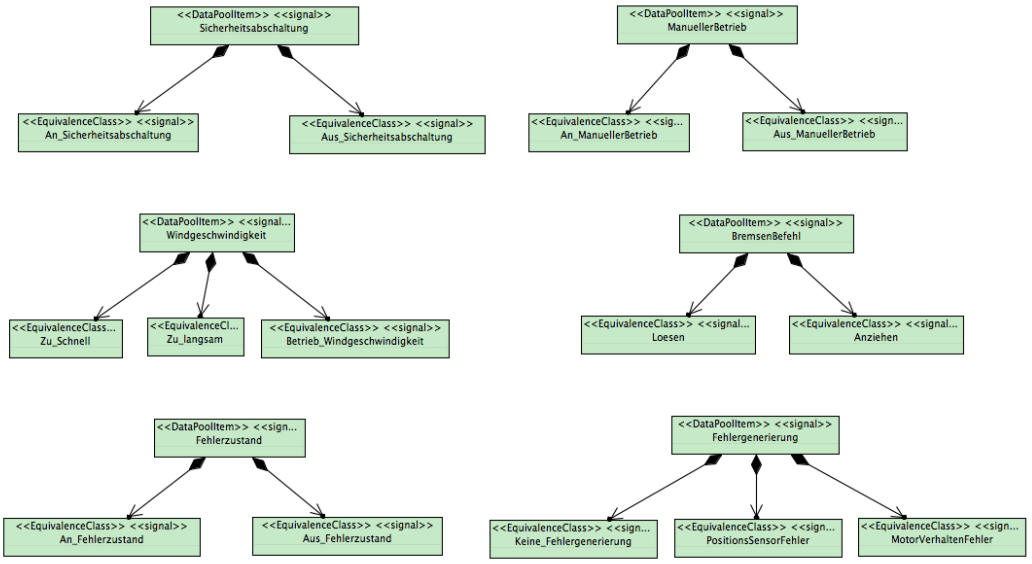


Abbildung 17: Ein Teil des Datenmodells für das MBT

Anders als in [4], wird in dieser Arbeit jeder Äquivalenzklasse ein eigenes Signal-Element zugewiesen. Somit können die Äquivalenzklassen, die stellvertretend für eine Menge von Eingabewerten stehen, als Signale für die Verhaltensdiagramme verwendet werden.

Abbildung 18: Beispiel für die Belegung der Eigenschaften des Stereotypen “DataPoolItem”

Die Stereotypen "DataPoolItem" und "EquivalenceClass" haben die Eigenschaften "min" und "max", welche für die Definition der Wertebereiche verwendet werden. Möchte man statt eines Wertebereichs nur einen Wert angeben, setzt man beide Eigenschaften auf den gleichen Wert. Der Stereotyp "DataPoolItem" hat noch zwei weitere Eigenschaften, "type" und "boundaryValues". Die Eigenschaft "type" ist ein "Type"-Datentyp. Das bedeutet, es werden für diese Eigenschaft Datentypen angegeben. Die bisher in dieser Arbeit unterstützten Datentypen sind Boolean, Integer, Double und String. Die Eigenschaft "boundaryValues" ist in einer

späteren Phase für den Codegenerator wichtig. Diese Eigenschaft kann die Werte "false" und "true" annehmen. Ist der Wert dieser Eigenschaft auf "true" werden in der Codegenerierung die Grenzwerte der Wertebereiche beim datengetriebenen Testen mit berücksichtigt. Wie das funktioniert wird in Kapitel 5.3 gezeigt.

Die definierten Signale werden größtenteils als Parameter für die später erzeugten Testfälle benutzt. Möchte man demnach noch weitere Parameter zur systematischen Steuerung der Test angeben, definiert man einfach Signale wie oben beschrieben und setzt den Typ auf String.

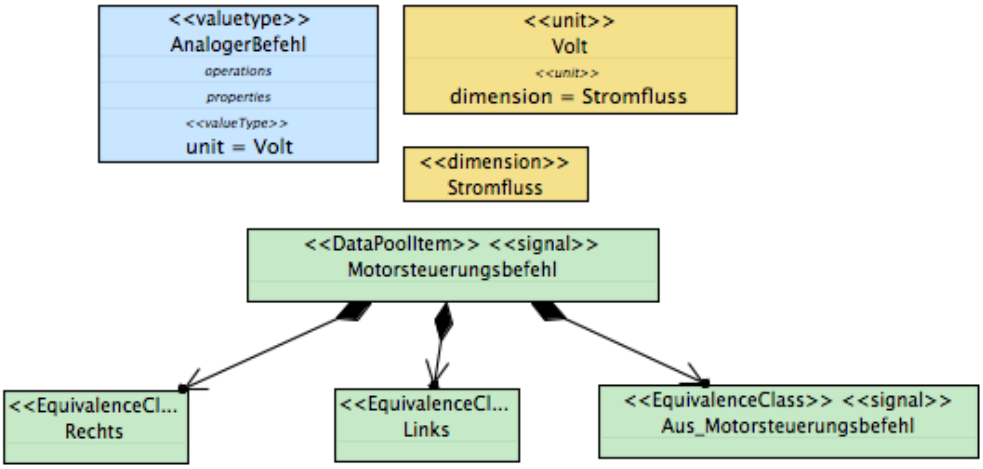


Abbildung 19: Ein Beispiel, die Testdaten mit mehr Informationen anzureichern

Idealerweise sind die Testdaten mit mehr Informationen versehen. Möchte man beispielsweise Informationen wie die Dimension und die Einheit der Datumsbeschreibung hinzufügen, kann dies durch die "RedefinedClassifier" Referenz geschehen. Man definiert wie in Abbildung 19 zu sehen ein Valuetype, welches eine Einheit (Unit) referenziert, welche wiederum auf eine Dimension verweist. Dieser Valuetype wird in der Eigenschaft "RedefinedClassifier" referenziert, womit man eine Vererbungs-Beziehung hergestellt hat.

4.1.2. Testfallgewinnung aus Verhaltensmodellen

Ein Verhalten eines Systems kann in SysML in verschiedener Art und Weise und aus verschiedenen Kontexten modelliert werden. Wie schon erwähnt, sollen aus den Verhaltensmodellen Testfälle ableitbar sein. Die Aufgabe, die sich hier stellt, ist das Auffinden einer geeigneten formalen Notation mit den Verhaltensmodellen. Aus diesen Verhaltensmodellen

sollten mindestens eine Menge an Eingabewerten und die Sollwerte für den Vergleich zu beziehen sein. Für viele Testfälle wäre eine temporal logische Reihenfolge der Eingabewerte sinnvoll.

Weiterhin stellt sich nicht nur die Frage wie etwas modelliert, sondern auch was und wie gut etwas modelliert werden kann und wie effizient Testfälle daraus bezogen werden können. Bei der Frage um die Effizienz ist der Bezug zwischen den Nutzen der Informationen im Modell und den Aufwand für die Modellierung gemeint.

In der Modellierung der Informationen für einen Testfall kann zwischen zwei Fällen unterschieden werden.

- die explizite Modellierung eines Testfalls
- die Gewinnung an Informationen aus Modellen, die das SuT-Verhalten auf bestimmte Eingaben hin beschreiben

Besonders bei der expliziten Modellierung eines Testfalls sollte geklärt werden, wie hoch der Aufwand ist, den Testfall zu modellieren, und wie viele Testfälle daraus zu gewinnen sind. Im Grunde kann man behaupten, dass das Modellieren nur eines Testfalls nicht sehr effizient ist. Um die Modellierung effizienter zu gestalten, kann man die Testfälle parametrisieren. Die Parameter können für die Eingabedaten im Sinne des datengetriebenen Testens oder zum Steuern verschiedener Testfallunterscheidungen eingesetzt werden.

Die Menge der Eingabewerte sollte bei der expliziten Modellierung von Testfällen so gewählt werden, dass die Ausgabewerte des SuT, nach der Stimulierung mit den Eingabewerten, sich möglichst ändern. Ein Test mit Eingaben, die voraussichtlich den Status des SuT nicht ändern oder das SuT zum Reagieren auffordern, ist auf Relevanz zu überprüfen. Vorherige Analysen, wie Risikoanalysen, bestimmen die Relevanz solcher Tests. Bei einer Risikoanalyse wird üblicherweise eine Wahrscheinlichkeit für das Auftreten einer Fehlerwirkung berechnet. Diese Wahrscheinlichkeit wird daraufhin mit der Einschätzung des Ausmaßes dieser Fehlerwirkung multipliziert, womit man eine relative Einschätzung des Risikos erhält und somit die Größe der Relevanz.

Die zu erwartenden Ausgabewerte können in den explizit modellierten Testfällen sowohl als statische Werte als auch dynamisch per Parameter in das Modell kodiert werden. Ferner ist auch eine Modellierung einer Soll-Funktion, zur Berechnung der Ausgabewerte, möglich. Die explizite Modellierung von Testfällen eignet sich gut für die anwendungsfallbasierten Tests, weil in den Modellen eine kontrollierte Eingabe von Stimuli modelliert werden kann.

Ein Vorteil der explizit modellierten Testfälle ist die in der Regel einfache Informationsgewinnung durch den Aufbau der Modelle. Diese Modelle sind oftmals gerichtet und trivial aufgebaut, sodass ein Codegenerator über einfache Algorithmen das Modell traversieren kann, um die nötigen Informationen für die Tests zu erhalten. Der Testarchitekt bestimmt die Komplexität der Modelle für die Testfälle.

Hingegen ist die Generierung von Testfällen aus Verhaltensdiagrammen, die das SuT-Verhalten beschreiben, gewöhnlich anspruchsvoller. Durch die Verhaltensmodelle werden einige Aspekte des Verhaltens des SuT abgebildet. Dadurch kann man eine Soll-Funktion ableiten. Für den Erhalt von sinnvollen Mengen von Eingaben sind Algorithmen erforderlich, die hinsichtlich eines oder mehrerer Testkriterien die Informationen aus den Modellen beziehen.

Ein geeigneter Algorithmus kann, anhand von Bedingungen aus Verhaltensdiagrammen und Testkriterien, Kombinationen von Eingabewerten erschließen. Durch Zyklen in den Verhaltensdiagrammen der Modelle kann der Raum der Mengen-Kombinationen unendlich groß werden. Man nennt dieses Phänomen Testfallexplosion. Um dem entgegen zu wirken, sollte für das Erschließen von Testfällen ein Testendekriterium festgelegt werden. Durch die Gewinnung von Testfällen aus den Verhaltensmodellen des SuT, erhält man systematischen hinsichtlich der Verhaltensstruktur und des angewandten Testkriteriums eine Menge an Testfällen. Soweit das Testkriterium keine Rückschlüsse darauf ziehen lässt, ist die Systematik der Gewinnung der Testfälle noch kein Garant für eine qualitative oder relevante Aussage über diesen Test. Unter umständen müssen Filterungen auf den Mengen oder weitere Testkriterien hinzugezogen werden, um nur die realevanten und qualitativ aussagekräftigen Testfälle zu erhalten.

Eine dritte Möglichkeit, Informationen über Testfälle in einem Modell zu kodieren, wäre die Verknüpfung der beiden oben genannten Fälle. Die explizite Modellierung von Stimuli in einem Verhaltensdiagramm und das Beziehen von Soll-Funktionen aus den Verhaltensdiagrammen des SuT. Die Informationen für einen Testfall wären dementsprechend in mehreren Verhaltensdiagrammen enthalten.

Die Konstrukte in den Modellen, aus denen die Informationen für die Tests bezogen werden, sollten auf jeden Fall an die Möglichkeiten der Extrahierung von Informationen angepasst werden. Dabei spielen die Wahl der Modellelemente und die Verbindungsmöglichkeiten eine große Rolle. Welche Elemente in welchen SysML Modell dürfen zum Beispiel genutzt werden und wie dürfen sie zueinander in Verbindung gesetzt werden? Welche graphentheoretische Restriktionen sind bei der Modellierung der Modelle gegeben, um bestimmten Algorithmen und Codegeneratoren zu verwenden? Die Syntax wird zum großen Teil schon von dem SysML Metamodell vorgegeben, aber nicht alle Elemente und Konstrukte sind in dieser Arbeit davon verwendet worden und in den in Kapitel 5.3 beschriebenen "Model-to-Text"-Tranformation eingeflossen. Welche Elemente und Restriktionen in dieser Arbeit verwendet wurden, wird in den folgenden Kapiteln erklärend aufgegriffen. Zudem muss auch an dieser Stelle erwähnt werden, dass auch Restriktionen bezüglich des Modellierwerkzeuges entstehen können. Auch in dieser Arbeit konnte das Problem des Fehlens von Modellelementen oder Funktionen zum akkuraten Modellieren mit SysML nicht umgangen werden.

Daten- und aktionsflussorientiertes Modellieren des Verhaltens

Die SysML hat für das Modellieren des Verhaltens mit Kontroll- und Datenflüssen die Aktivitätsdiagramme. Sie beschreiben mittels Kontrollflüssen eine temporallogische Reihenfolge

von Aktionen. Zudem können Datenflüsse hineingenommen werden. Die Steuerung eines Verhaltens funktioniert bei den Aktivitätsdiagrammen mittels Kontrollflusselementen. Sie geben den Anfang (InitialNode), das Ende (FlowFinal) und weitere Kontrollflussvorgänge an. Unter anderen wären da auch die Entscheidungsknoten (DecisionNodes). Sie entscheiden anhand einer Bedingung, wie der Kontrollfluss weiter verläuft.

Die Aktivitätsdiagramme eignen sich gut für die Modellierung von Stimuli. Die Aktionen können ein Senden von Stimuli oder das Warten auf Ereignisse darstellen. Dazwischen wären Entscheidungsknoten mit den Parametern für die Steuerung der Testfälle. Die Abbildung 20 zeigt ein Beispiel für die Abbildung von Stimuli-Kombinationen. Das in der Abbildung 20 gezeigte Diagramm wird verwendet, um das WNS in seinem normalen Betrieb zu stimulieren. Dabei bestimmen die Werte der Bedingungen an den DecisionNodes den Kontrollfluss.

Parallelität und spezielle Schleifen-Konstrukte wurden in dieser Arbeit nicht weiter behandelt. Für einen einfachen Zyklus können aber die DecisionNodes verwendet werden. Diese Zyklen sollten aber so geschickt eingesetzt werden, dass die Ausführung solcher Zyklen begrenzt möglich ist, indem man beispielsweise einen Zähler verwendet, der bei der Bedingung des DecisionNodes abgefragt wird, und damit die Anzahl der Ausführung der Zyklen limitiert.

In [4] werden die Aktivitätsdiagramme für die Testfälle verwendet. So werden sie für die explizite Modellierung von Testfällen verwendet, mit den Informationen zu den Eingaben und den zu erwarteten Ausgaben. In diesen Modellen wird die Übergabe der Testergebnisse an den Arbiter (Testorakel) zusätzlich mitmodelliert. Für den Leser wird dadurch deutlich, dass es sich um einen Testfall handelt. Falls die Berechnung des Ergebnisses aber komplexer ist, muss vor der Ergebnis-Übergabe eine oder mehrere Aktionen für die Berechnung des Ergebnisses stattfinden.

Möchte man aus einem Aktivitätsdiagramm, welches ein Verhalten des SuT beschreibt, Testfälle ableiten, müssen die Aktionen hinreichend gut spezifiziert sein. Die Mengen an Eingabewerten könnten aus den Bedingungen im Diagramm erschlossen werden. Für die Soll-Werte, die für den Vergleich mit den Ist-Werten gebraucht werden, müssen Soll-Funktionen ableitbar sein. Einige Arbeiten (siehe [32]) beschäftigen sich mit der Generierung von Simulationsmodellen aus den Aktivitätsdiagrammen. Diese Simulationsmodelle kann man für die Berechnung der Sollwerte verwenden. Die Generierung von Simulationsmodellen aus Aktivitätsdiagrammen ist ein Thema, welches in dieser Arbeit nicht weiter behandelt wird.

Eventbasierte Verhaltensmodelle

Mit den Zustandsautomaten in der SysML lässt sich ein ereignisbasiertes Verhalten modellieren. Viele Begebenheiten eines Systems lassen sich in Ereignisse ausdrücken, sodass man das Verhalten eines Systems oftmals in einem Zustandsautomaten abbilden kann. Sehr viele Arbeiten (siehe [24] und [16]) haben sich schon mit dem Thema der Generierung von Testfällen aus Zustandsautomaten beschäftigt.

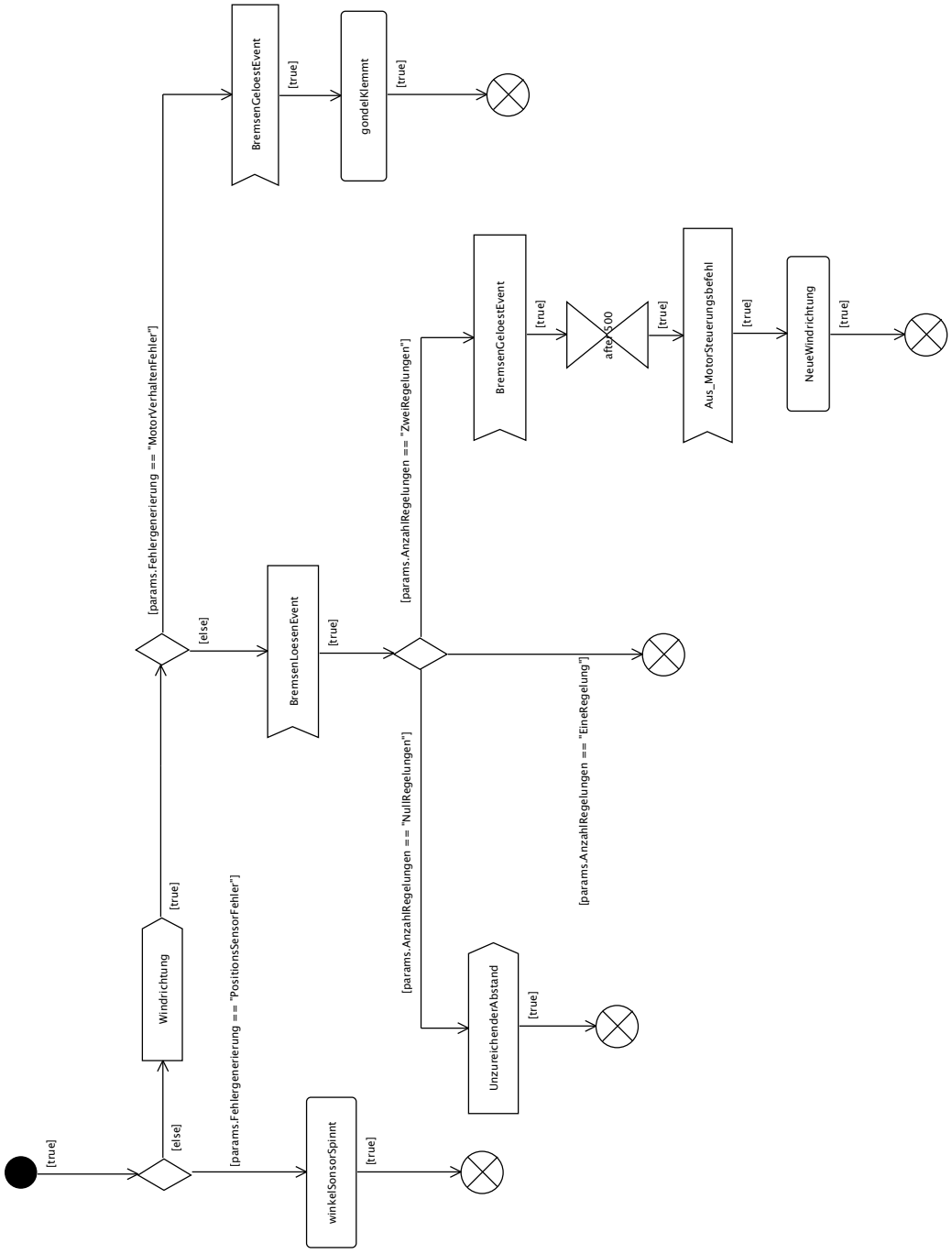


Abbildung 20: Beispiel für das Verhalten eines parametrisierten Testtreibers (Stimuli Generator)

Über die Zustände lassen sich die Ausgaben eines Systems beschreiben. Besonders praktisch sind die drei Verhaltensspezifikationen eines Zustandes. So kann man das Verhalten für den Eintritt, den Austritt und der Aktivität des Zustandes definieren. Mindestens die Verhaltensspezifikationen über die Aktivität des Zustandes sollte eine Ausgabe definieren, um eine Soll-Funktion aus einem Zustandsdiagramm ableiten zu können. Die Transitionen beherbergen die Informationen für den Wechsel in weitere Zustände. Über einen Trigger wird das Eintreffen eines Ereignisses überprüft. Eine Wächterbedingung kann zusätzlich die Benutzung der Transition einschränken. Trifft nun das getriggerte Ereignis ein und ist die Wächterbedingung auf der Transition wahr, wird das Austritts-Verhalten des aktuellen Zustandes ausgeführt. Falls die Transition als Effekt ein Verhalten definiert, wird daraufhin dieses Verhalten ausgeführt. Danach wird der Zustand, auf den die Transition verweist, aktiviert. Falls der neu aktivierte Zustand ein Eintritts-Verhalten definiert, wird vorerst dieses vor dem Aktivitäts-Verhalten des neuen Zustandes ausgeführt.

Überführt man nun diese Information eines Zustandsdiagramms in ein ausführbares Modell, kann dieses Modell als Soll-Funktion dienen. In den in dieser Arbeit benutzten Zustandsdiagrammen wurde das Aktivitäts-Verhalten der Zustände für die Definierung der Soll-Werte verwendet. Die Trigger und Wächter an den Transitionen, sind abfragen auf Signale. Somit kann zu jeder Zeit eines Testfalls eine Aussage über den erwarteten internen Zustand des SuT gemacht werden und damit auch über die erwarteten Ausgabewerte. Für das Erschließen von systematisch sinnvollen Mengen an Eingabewerten, müssen geeignete Testendekriterien definiert werden.

Für einen Zustandsautomaten wäre ein Überdeckungsgrad als Testendekriterium eine brauchbare Lösung. Ein Zustandsüberdeckungsgrad von 100 % würde beispielsweise heißen, dass in einem Test jeder Zustand mindestens einmal aktiviert wurde. Ein Zustandsübergangsüberdeckungskriterium würde eine Aussage über die verwendeten Transitionen im Test machen. Ein Zustandsübergangsüberdeckungsgrad von 100 % würde dementsprechend verlangen, dass jede Transition mindestens einmal in einem Test verwendet wurde. In dem Kapitel 4.2 werden die Algorithmen für die Gewinnung von Mengen an Eingabewerten, zur Erfüllung der zwei genannten Testendekriterien, erörtert.

Die Stubs in den Tests dienen zur Simulation der Umgebungskomponenten. Sie stimulieren Eingänge des SuT. Diese Eingänge sind von einem Testcontroller nicht stimulierbar, weil sie schon von den Stubs bedient werden und daher in Konkurrenz stehen würden. Für die Generierung von Eingangswerte-Mengen aus Zustandsautomaten heißt das wiederum, dass keine Ereignisse oder Bedingungen im Automaten Abhängigkeiten auf diese Eingänge haben dürfen, sonst erhält man Eingangswerte-Mengen, die zu einem unkontrolliert oder ungewollten Testverlauf führen. Um diesem Problem aus dem Weg zu gehen, muss entweder das Verhalten der Stubs in den Zustandsautomaten einfließen oder man bezieht die Eingangswerte-Mengen aus einem separaten Modell, welches die Eingangswerte-Kombinationen explizit abbildet wie in Abbildung 20.

Wenn man das Verhalten der Stubs in die Zustandsautomaten einfließen lässt, können Aufrufe der Funktionen auf den Stubs, die die Stimuli des Stubs beeinflussen können, als Ersatz

stellvertretend für die Eingangswerte, welche die mit dem Stubs verbundenen Eingänge stimulieren, stehen. Somit erhält man Mengen von Eingangswerten und Funktionsaufrufen auf den Stubs.

In dieser Arbeit wurde entschieden, dass die Stimuli aus gesonderten expliziten Verhaltensmodellen bezogen werden, um den Testverlauf besser kontrollieren zu können. Die Zustandsautomaten, die das Verhalten des SuT darstellen, werden für das Generieren von Soll-Funktionen herangezogen.

Ausgenommen davon sind die Zustandsautomaten, die die eigentlichen Zustände (die Betriebsmodi) darstellen. Für den Wechsel der verschiedenen Modi des Fallbeispiels ist es wichtig, die Eingangswerte für die Ereignisse zum auslösen der Zustandswechsel beziehen zu können. Der Zustandsautomat, welcher in Abbildung 21 zu sehen ist, beschreibt die Modi des WNS mit seinen Zuständen und welche Signale einen Übergang zu einem anderen Modus bewirken. Alle Zustände, welche den gleichen Namen mit einer Zahl am Ende tragen, sind der gleiche Modus. Für den Test der WNS-Modi müssen Testfälle überprüfen, ob das SuT sich auch wirklich in diesem Modus befindet, indem auf das Verhalten des SuT in diesem Modus getestet wird. Für das Verhalten eines Modus können wiederum Modelle herangezogen werden, wie in Abb. 33 zu sehen, welche das Verhalten des SuT im Modus "NormalerBetrieb" zeigt.

Das Wechseln und das Testen der einzelnen Modi kann in ein Testszenario zusammengelegt werden. Ein Testszenario nach [35] ist eine Aneinanderreihung von Testfällen. Hier muss darauf geachtet werden, dass die Nachbedingung eines Testfalls der Vorbedingung des nächsten Testfalls entspricht. Da die Menge der Signale für den Wechsel der Modi disjunkt von der Menge der Signale in den Testfällen ist, kann das Signal für den Wechsel des Modus in die Nachbedingung des jeweiligen Testfalls hinzugegeben werden. Die temporallogische Reihenfolge der Signale für die Wechsel der Modi bleibt insofern in einem Testlauf erhalten, dass die Testfälle mit diesen Signalen in den Nachbedingungen die gleiche Reihenfolge in einem Testszenario einnehmen. Gegeben sei folgendes Beispiel:

Die Abb. 22 zeigt einen vereinfachten Zustandsautomaten von Betriebsmodi, welcher auf bestimmte Signale den Modus wechselt. Für eine vollständige Zustandsübergangsüberdeckung erhält man folgende Reihenfolge an Signalen:

“Zu_Schnell” → “An_Sicherheitsabschaltung” → “Aus_Sicherheitsabschaltung”

Um zu überprüfen ob das SuT nach den senden der Signale sich auch im richtigen Modus befindet, müssen die Testfälle für die einzelnen Modi durchgeführt werden. Die Reihenfolge der Testfälle wäre folgende:

Testfall:NormalerBetrieb → Testfall:Sturmwarnung → Testfall:Sicherheitsabschaltung → Testfall:NormalerBetrieb

Für die ersten drei Testfälle müssen die Signale für die Zustandswechsel in den Nachbedingungen integriert werden, damit die Vorbedingung des nächsten Testfalls eingehalten und dieser Test korrekt ablaufen kann. Dieser Test setzt einen Zustandswechsel des Systems

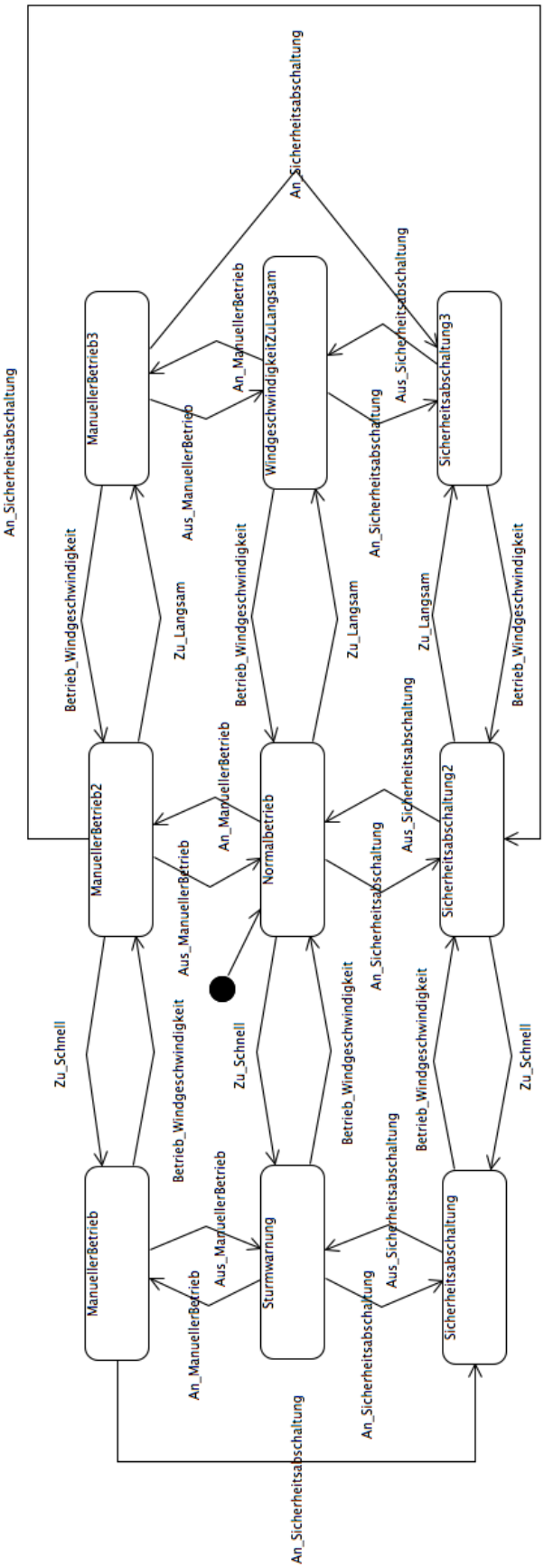


Abbildung 21: Zustandsautomat für die Betriebsmodi des WNS

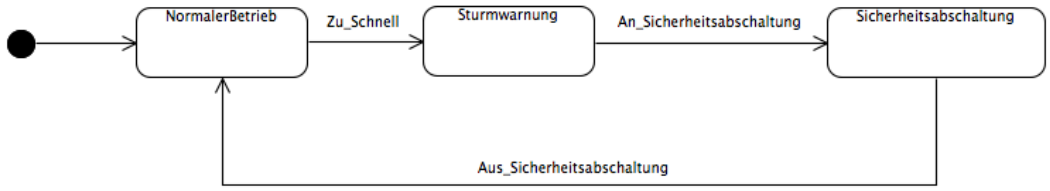


Abbildung 22: Vereinfachtes Beispiel für einen Zustandsautomaten

aus kontrollierten internen Zuständen voraus. Der Wechsel des Modus des SuT wird erst vorgenommen, wenn der Testfall beendet wurde. So ein Testfall erwartet in der Regel einen internen oder system-globalen Zustand des SuT um den Testfall zu beenden. Im Fallbeispiel WNS würde ein Testfall im normalen Betriebsmodus darauf warten, dass das WNS die Gondel beispielsweise in die aktuelle Windrichtung geregelt hat, um ein Ergebnis berechnen zu können. Die Betriebsmodi können aber jederzeit wechseln. Für sicherheitskritisches Verhalten sollten auch Wechsel aus verschiedenen Zeitpunkten oder internen Zuständen der Betriebsmodi druchgeführt werden. Dieser Sachverhalt sollte besonders für den Wechsel aus irgendeinem Betriebsmodus in den Sicherheitsabschaltung-Modus in den Tests berücksichtigt werden, weil dieser Wechsel sicherheitskritisch ist.

Eine Möglichkeit wäre einen Zustandswechsel zufällig während eines Tests wechseln zu lassen. Das würde aber die Eigenschaft der Reproduzierbarkeit eines Tests verletzen. Weiterhin hätte man den Effekt, dass die noch laufenden Testfälle Fehler im System attestieren würden, was an dieser Stelle nicht wahr wäre. Für diesen Sachverhalt müssen daher explizite Testfälle für einen gewollten Wechsel des Betriebsmodus erstellt werden. Mit vordefinierten Zeitpunkten für den Wechsel der Modi können solche Testfälle nur die nötigen sicherheitskritischen Merkmale am SuT überprüfen.

Nachrichtenbasierte Verhaltensmodelle

Das Modellieren von nachrichten-basierten Verhalten ist mit den Sequenzdiagrammen möglich. Sie zeigen bestimmte Kommunikation von Nachrichten zwischen Entitäten auf. Diese Entitäten können das SuT, Testkomponenten/Umgebungskomponenten oder Testtreiber sein. Durch die Lebenslinien der einzelnen Entitäten lässt sich recht einfach eine temporallogische Reihenfolge für die Kommunikation der Nachrichten beziehen. Die "CombinedFragments" können den Verlauf von Nachrichten verschachteln und somit den Verlauf der Nachrichten an Bedingungen knüpfen oder das Senden oder Empfangen von Nachrichten parallelisieren.

Mit den Sequenzdiagrammen lassen sich Zeitpunkte oder Zeitdauer zwischen dem Eintreffen oder Absenden von Nachrichten besonders gut veranschaulichen. Leider lassen sich aber Bedingungen, Zustände oder Signalwertkombinationen, sei es global oder auf einzelnen Komponenten, nicht sehr gut in den Sequenzdiagrammen modellieren. Diese Bege-

benheiten können in Nachrichten mit Signalen verpackt werden. Dabei werden komplexe Signale in die Nachrichten kodiert und die Bedingungen auf den Nachrichten gemacht. Die Abbildung 23 zeigt ein Beispiel für einen expliziten Testfall mit einem Sequenzdiagramm. Eine Vielfalt von Beispielen von explizit modellierten Testfällen mit Sequenzdiagrammen findet man in [4]. Es werden Beispiele gezeigt, wie man die Diagramme verschachteln oder die Nachrichten für datengetriebenes Testen parametrisieren kann.

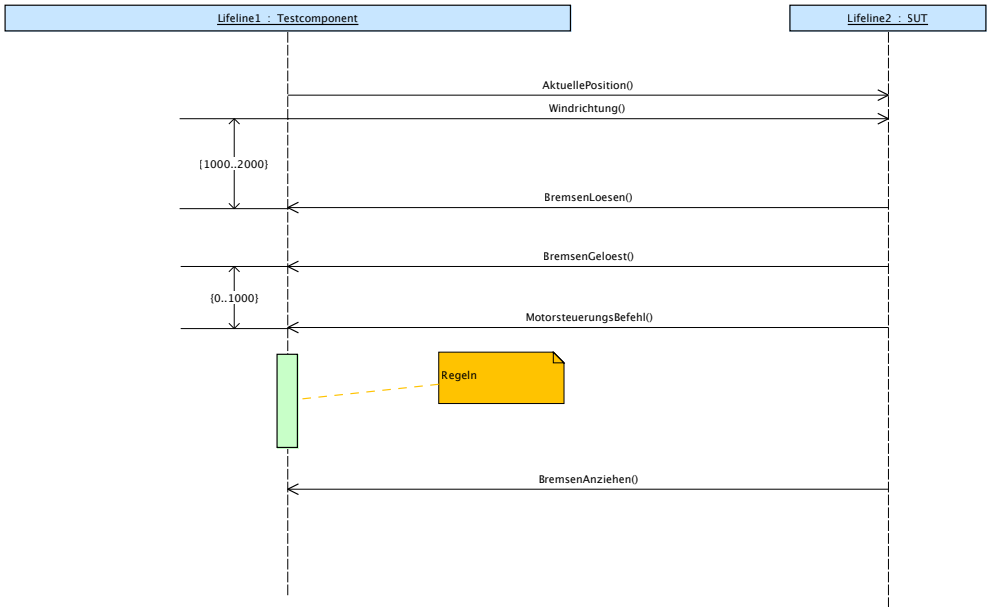


Abbildung 23: Ein Beispiel eines explizit modellierten Testfalls mit einem Sequenzdiagramm

Mit der Modellierung von mehreren Entitäten lassen sich komplexe Zusammenhänge in der Kommunikation zwischen den Entitäten darstellen. Beispielsweise kann man die Umgebungskomponenten als Testkomponenten in das Diagramm einfügen und durch die CombinedFragments mit dem "InteractionOperator" "Optional" verschiedene Verhaltensweisen der Testkomponente in Form von Nachrichten an das SuT (oder sogar das Ausbleiben solcher) modellieren.

Die Sequenzdiagramme sind in dieser Arbeit nicht weiter behandelt worden, weil das in dieser Arbeit verwendete Modellierwerkzeug die Reihenfolge der Nachrichten-Ereignisse auf den Lebenslinien der Entitäten und die CombinedFragments aus den Diagrammen nicht korrekt in das dahinterstehende Modell überführen konnte.

Bedingungen in Modellen und mathematische Gleichungen

Manche Anforderungen definieren Bedingungen an das SuT, die jederzeit gelten. Diese Bedingungen können demnach in jedem Testfall überprüft werden. Solche Bedingungen kön-

nen heißen, dass ein Wert an einem Port nicht gleichzeitig mit einem anderen Wert an einem anderen Port auftreten darf. Solche Constraints (engl. Beschränkung) können in den Strukturdiagrammen beschrieben werden. Die Abbildung 24 zeigt, wie ein Constraint auf zwei Ports referenziert und eine Beschränkung definiert. Dieses Constraint wird in jedem Testfall als Bedingung für ein positives Testergebnis herangezogen. Ist diese Bedingung zu irgendeiner Zeit nicht erfüllt, ist das Testergebnis negativ. In der Abbildung 24 darf, solange die Bremsen angezogen sind, kein Motorsteuerungsbefehl zum Fahren des Motors vom WNS gegeben werden.

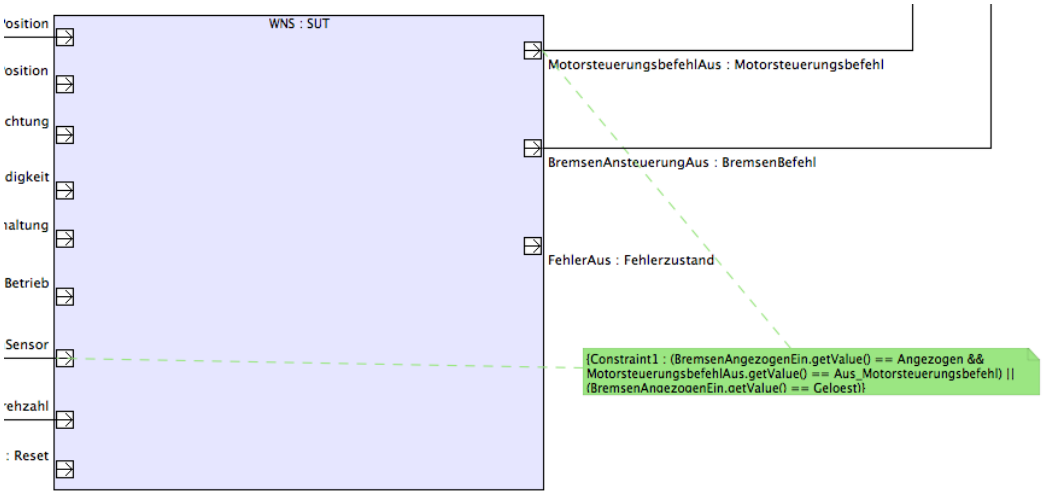


Abbildung 24: System-Globale Anforderung als Constraint in einem BDD definiert

Diese Art der Bedingungen kann noch weiter präzisiert werden. In der SysML gibt es die Parametrischen-Diagramme. Sie definieren über die Eigenschaften von Blöcken Zusammenhänge in mathematischen Formeln. Entwickelt man demzufolge Zusammenhänge zwischen den Eingangs-Ports und den Ausgangsports eines SuT-Blocks, hätte man dadurch eine Soll-Funktion. Leider waren die Funktionalitäten des Modellierungswerkzeuges für die Parametrischen-Diagramme zur Zeit dieser Arbeit unzureichend implementiert, sodass man diese nicht nutzen konnte. Eine Untersuchung auf diesen Diagrammen, als Definition von Soll-Funktionen für das MBT, steht dementsprechend noch aus.

Wenn es anschließend um die Frage geht, wie und mit welchen Verhaltensdiagrammen beschreibt und modelliert man das Verhalten richtig, kann keine exakte Antwort gegeben werden. Je nach Kontext, der modelliert werden soll, haben die einzelnen Diagrammtypen ihre Vor- und Nachteile. Die meisten der MBT-Studien aber beschäftigen sich mit den Zustandsdiagrammen (siehe [22] Kapitel 3.1.4), weil sie unter anderem von vielen Untersuchungen und Studien vollständig erarbeitet wurden und man auf diesen Studien leicht aufbauen kann.

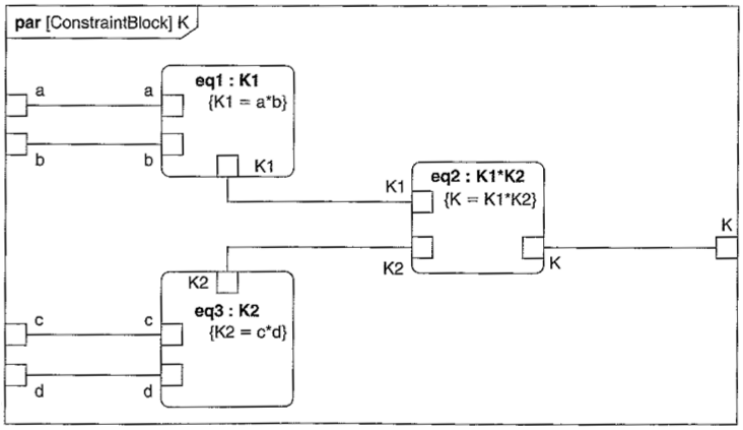


Abbildung 25: Ein Beispiel für ein parametrisches Diagramm aus dem Buch [11]

4.1.3. Verfolgbarkeit der Anforderungen im Modell

In der SysML gibt es das Anforderungsdiagramm zum Modellieren der Anforderungen an ein System. In diesen Diagrammen können Anforderungen und ihre Beziehung zueinander dargestellt werden. Ferner lassen sich Beziehungen zwischen den Anforderungen und anderen Elementen aus dem Modell herstellen. Mit den Anforderungen im Modell werden die Beziehungen zwischen den Modellelementen deutlich. Somit kann verfolgt werden, welche Komponenten eines Systems welche Anforderungen erfüllen. Auch die Verfeinerungs-Beziehungen der Anforderungen untereinander lassen darauf schließen, welche übergeordneten Anforderungen erfüllt sind, indem man nämlich überprüft, ob alle verfeinerten Anforderungen erfüllt sind.

Für das MBT ist die Verifizierungs-Beziehung (verifies bzw. verifiedBy) wichtig. Sie gibt an, welche Modellelemente die Anforderung an das SuT überprüfen. Diese Beziehung wird meist mit einem "TestCase" Element und einer Anforderung hergestellt (siehe 26). Manchmal ist es aber sinnvoller, diese Beziehung mit einem Teilelement des TestCase Elementes herzustellen, um Rückschlüsse auf einzelne bestimmte Anforderungen bei einem negativen Testergebnis ziehen zu können.

Die Beziehungen in Anforderungsdiagrammen haben den Vorteil, dass sie eine Verfolgbarkeit der Anforderungen und deren Umsetzung im Modell bieten. Speziell für das MBT heißt das, das man über die Verifizierungs-Beziehungen herausfinden kann, für welche Anforderungen noch kein Testfall vorhanden ist, indem man sich alle Testfälle einer Anforderung anzeigen lässt. Weil diese Beziehungen bidirektional sind, kann man sich auch alle Anforderung für einen Testfall anzeigen lassen. Das hat den Vorteil, dass man nach dem Testlauf automatisch einen Testbericht erstellen kann. Dieser listet auf, welche Testfälle erfolgreich und welche nicht erfolgreich durchgeführt wurden. Sind alle Testfälle für eine Anforderung erfolgreich abgeschlossen, gilt diese Anforderung als erfolgreich getestet. Für alle Testfälle,

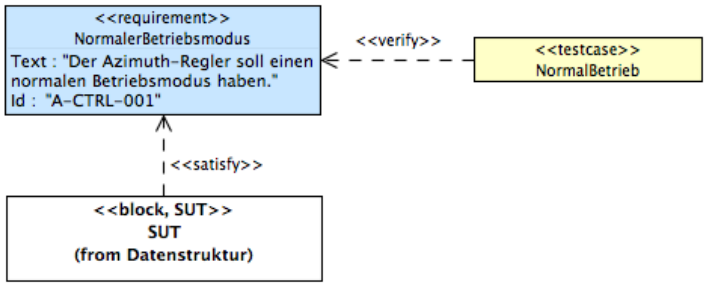


Abbildung 26: Beispiel von einer Beziehung zwischen Anforderung, Testfall und SuT Komonente

die nicht erfolgreich durchgelaufen sind, gilt es den Grund und gegebenenfalls die Anforderungen zu überprüfen.

Ein weiterer Vorteil ist wenn man eine Anforderung im Modell ändert, kann man über die Verifizierungs-Beziehungen die abzuändernden Testfälle oder Teilkomponenten feststellen.

Die Anforderungen werden vorerst mit in das Modell genommen. Es besteht keine Notwendigkeit, jede Anforderung in ein Anforderungsdiagramm darzustellen. Für bestimmte komplexere Beziehungs-Darstellungen ist das Anforderungsdiagramm eine gute Lösung. Es kann also für das MBT von Bedeutung sein, zu wissen, welche Komponenten die Anforderungen erfüllen und von welchen Testfällen sie getestet werden müssen. Häufig reicht aber eine Herstellung der Beziehung über sogenannte Callout-Notizen aus (siehe Abb. 27). Callout-Notizen sind Notizen-Elemente in Diagrammen, die eine Beziehung zwischen einer Anforderung und einem Modellelement herstellt. Dabei spielt es keine Rolle, in welchem Diagrammtyp man sich befindet.

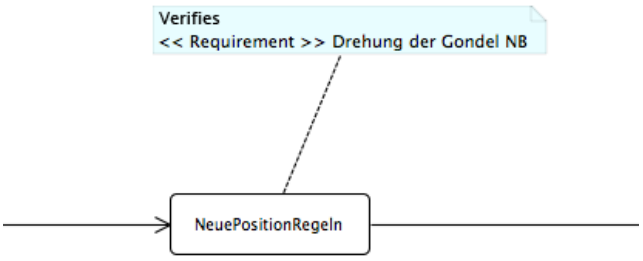


Abbildung 27: Ein Beispiel für eine Callout-Notiz

4.2. Algorithmen

Für das modellbasierte Testen müssen aus Modellen Informationen ausgelesen und korrekt interpretiert werden. Geeignete Algorithmen helfen, automatisch Informationen auszulesen und diese in andere Modelle zu überführen. Dabei werden verschiedene Algorithmen für diverse Problemstellungen verwendet.

Im Kontext dieser Arbeit werden Algorithmen verwendet, um Dateninformationen aus den Modellen auszulesen und zu interpretieren, und mittels Testendekriterien systematisch Testfälle aus den Modellen zu beziehen.

Die Algorithmen müssen teilweise durch das Modell traversieren und die Informationen in ein Datenmodell überführen können. Besonders für die Gewinnung von systematischen Testfällen mittels Testendekriterien, müssen die Algorithmen Aufgaben wie das Erkennen von Strukturen oder für die Optimierung der Testfälle Minimalwert-Probleme lösen. Für die Lösung der Aufgaben gibt es verschiedenste Lösungswege und Algorithmen, die verwendet werden können, wie Algorithmen aus der Graphentheorie zum Finden kürzester Wege, Algorithmen zur Berechnung von Minimalwerten oder Algorithmen zur Interpretation und Auswertung von kombinatorischen Logiken.

Die Algorithmen finden anhand geeigneter Testendekriterien minimale Mengen von Testfällen zur Erfüllung von bestimmten Testkriterien. Die Aussagekraft der gebildeten Menge von Testfällen ist größtenteils von der Signifikanz der Testkriterien mit ihren Metriken abhängig. Ferner werden von den Algorithmen keine Analysen auf den Informationen im Modell gemacht, sodass Unstimmigkeiten, wie eine falsche Definierung eines Ports im Modell, zwischen dem MBT-Modell und dem SuT nicht gefunden werden.

Um Algorithmen auf Modellen ausführen zu können, müssen diese formal spezifiziert sein, damit die Algorithmen die Informationen richtig interpretieren können. Wenn demnach der Formalismus nicht vom Algorithmus interpretiert werden kann oder formale Informationen auf bestimmten Modellelementen fehlen, bricht der Algorithmus an dieser Stelle ab. Zusätzlich geben die Algorithmen bestimmte strukturelle Beschränkungen auf den Modellen vor, damit sie ausgeführt werden können.

In der Regel kann aus den Verhaltensmodellen eine unendliche Menge von Testfällen bezogen werden. Welche Mengen von Testfällen sinnvoll aus diesen Modellen bezogen werden können, wird über die Testendekritrien definiert. In [35] wird ein Testendekriterium als, eine "Menge abgestimmter generischer und spezifischer Bedingungen, die von allen Beteiligten für den Abschluss des Tests akzeptiert wurden" beschrieben. "Durch Festlegung von Ausgangsbedingungen wird vermieden, das der Test als abgeschlossen betrachtet wird, auch wenn Teile des Tests noch nicht abgeschlossen sind".

Überdeckungskriterien sind ein häufig verwendetes Testendekriterium für die Gewinnung von endlich vielen Testfällen aus Modellen. Die Überdeckungskriterien können in folgende Aspekte klassifiziert werden:

Funktionale Kriterien die abzudeckenden Aspekte im Test sind die funktionalen Eigenschaften des SuT. Oft auftretende Funktions-Szenarien werden hergenommen, um einzelne Testfälle in Testszenarien zu bündeln. Die Abdeckung der Funktionen des SuT stehen in den Tests im Vordergrund

Strukturelle Kriterien die Strukturen eines Systems oder einer Systembeschreibung werden für das abzudeckende Kriterium verwendet. Dabei könnte ein Kriterium lauten, dass mindestens jeder Zustand des SuT in den Tests aktiviert und getestet wurde.

Stochastische Kriterien die Testspezifikation beinhaltet stochastische Informationen über verschiedene Aspekte des SuT. Informationen beispielsweise darüber, wie oft eine Funktion in Zukunft benutzt wird, gibt einen Faktor für die Relevanz und die Priorisierung für den Test dieser Funktion an.

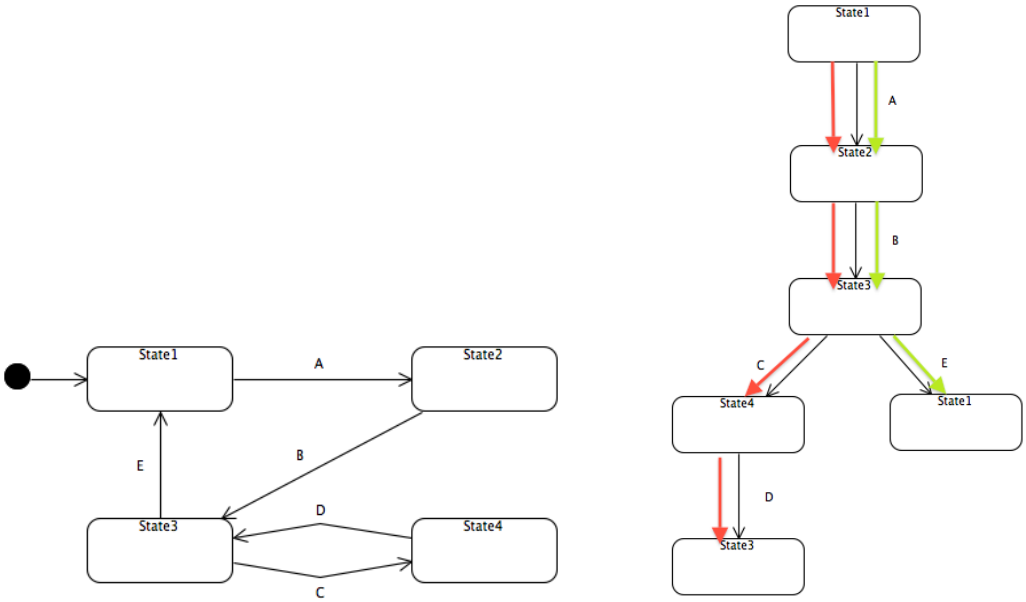
In [13] wird evaluiert, wie die verschiedenen Überdeckungskriterien in den Tests angewendet werden können. In dieser Arbeit werden die funktionalen und die strukturellen Überdeckungskriterien verwendet. Der Grad der Überdeckung soll in dieser Arbeit immer 100 % betragen. Als funktionales Überdeckungskriterium werden die verschiedenen Betriebsmodi des WNS auf ihre korrektes Verhalten geprüft. Im normalen Betriebsmodus wird beispielsweise die Nachführung der Gondel und das Ansteuern der Bremsen getestet. Für den Nachweis der Abdeckung aller Funktionen wird die in dem Kapitel 4.1.3 beschriebene Methodik der Verfolgbarkeit von Anforderungen zunutze gemacht.

Die strukturellen Überdeckungskriterien werden in dieser Arbeit auf den Zustandsdiagrammen und den Daten angewendet. Für die Daten werden kombinatorische Logiken aufgestellt, um die für das datengetriebene Testen relevanten Mengen von Eingabewerten zu erhalten. In Kapitel 5.3.1 wird gezeigt, wie die Daten-Kombinationen in Form von Parametern für die Testfälle erzeugt werden.

Für die Zustandsdiagramme können verschiedene Testendekriterien für die Testfälle definiert werden. Eine Methodik für Erschließung von Mengen von Eingabewerten für Testfälle ist das Aufspannen eines Übergangsbaums. Dieser Spannbaum definiert über seine Zweige die Pfade durch einen Zustandsautomaten ausgehend von einem Startzustand. Dabei werden die Ereignisse und die Wächter an den Transitionen als Eingabewerte formuliert. Damit der Zustandsbaum nicht unendlich groß wird, werden Testendekriterien darauf angewendet. Ein Testendekriterium könnte folgendermaßen lautet: Wenn ein Endzustand erreicht wurde, führt kein weiterer Zweig davon ab. Ein weiteres Endekriterium beim Aufbau des Baumes wäre, wenn ein Zustand oder eine Transition eine gewisse Anzahl oft aktiviert bzw. ausgelöst wurde.

Die Abbildung 28 (a) zeigt einen Zustandsautomaten, der in ein Übergangsbaum (b) überführt wird. Das Endekriterium ist die zweite Aktivierung eines Zustandes, also wenn ein Zustand schon oben im Baum einmal aktiviert war, wird bei der zweiten Aktivierung dieses Zustandes der Baum an diesem Zweig unterbrochen.

Algorithmus:



(a) Beispiel Zustandsautomat (b) Übergangsbaum

Abbildung 28: Aufspannen eines Übergangsbaums

1. Nehme den Anfangszustand als Wurzel des Baumes.
2. Für jede ausgehende Transition wird ein Zweig mit dem Ereignis und dem Wächter zu dem Zustand, zu dem die Transition hinführt, erzeugt.
3. Falls der nächste Zustand weder ein Endzustand oder das Testendekriterium erfüllt, führe den Baum mit jedem aktuellen neuen Zustand mit Punkt 2 fort.

Wenn der Baum nun aufgespannt ist, können Pfade, wie in Abbildung 28 (b) mit den farblichen Pfeilen illustriert, von der Wurzel aus bis in die Blätter gezogen werden. Jeder dieser Pfade (in unserem Beispiel sind es zwei) ist ein Testfall. Anhand der Reihenfolge der Ereignisse und Wächter können die Eingabewerte ermittelt werden.

Ein Nachteil an dieser Methodik ist, dass sehr schnell viele Testfälle erzeugt werden, wo womöglich weniger Testfälle ausreichen würden. Möchte man unter Umständen nur alle Zustände getestet haben, würde in Abbildung 28 (b) der Testfall mit den roten Pfad ausreichen.

Eine Möglichkeit für das Minimieren der Testfälle wäre eine geschickte Wahl von Testfällen, die in der Summe alle Zustände abdecken. Andererseits könnte man auch andere Algorithmen zur Findung von Testfällen für das erfüllen der Testendekriterien heranziehen. Der

dieser Arbeit zugrundeliegende Codegenerator, für die Erzeugung von Testfällen, implementiert folgende Testendekriterien:

1. die Menge der Testfälle deckt jeden Zustand ab
2. jede Transition im Zustandsautomaten wurde mindestens einmal ausgelöst

Für die Erfüllung der Testendekriterien, mussten verschieden Algorithmen verwendet werden. Wenn das Testendekriterium aus Punkt 2 erfüllt ist, ist automatisch auch das Kriterium in Punkt 1 erfüllt. Einen optimalen Algorithmus für den Punkt 1 zu evaluieren ist sehr aufwendig. Das Problem ist NP-Vollständig und lässt sich daher nicht effizient lösen. Für den Punkt 2 wiederum gibt es verschiedene Lösungsansätze. Ein Algorithmus zur Lösung des "Chinese Postman Problems" wäre ein möglicher Lösungsansatz für Punkt 2. In 4.2.1 wird eine Java-Implementierung von Algorithmen zur Lösung des Chinese Postman Problems vorgestellt. Dieser wurde in den Codegenerator implementiert.

Die Lösung des Chinese Postman Problems hat den Vorteil, dass man nur einen Testfall zur Erfüllung der beiden oben genannten Testendekriterien erhält. Der Nachteil ist, dass die Algorithmen zum Lösen des Problems nicht für jeden erstellten Graphen der Verhaltensdiagramme anwendbar ist.

4.2.1. Chinese Postman Problem

Das Chinese Postman Problem geht zurück auf einen chinesischen Mathematiker, der einen Algorithmus suchte, mit dem man die kürzeste Tour für einen Postboten findet. Die Aufgabe bestand darin, eine Tour zu finden, bei der jede zu besuchende Straße mindestens einmal durchlaufen wird und die Tour an der Post, wo sie begonnen hatte, auch endet. Dabei kann es auch vorkommen, dass eine Straße mehrere Male durchlaufen wird. Zudem sollte noch berücksichtigt werden, dass Straßen verschiedene Längen haben können. Das Lösen des CPP's (Chinese Postman Problem) für einen Graphen hat den Vorteil, einen kurzen Pfad in einem Graphen zu finden, bei dem jede Kante mindestens einmal durchlaufen wurde.

In dieser Arbeit wird ein Algorithmus besprochen, der das CPP in gerichtete Graphen löst (Directed CPP). Bezogen auf das Problem des Postboten, würden nur Einbahnstraßen existieren. Weiterhin wird darauf eingegangen, wie man aus dem CPP ein „open CPP“ macht. Bei einem offenen CPP ist es egal, ob der Startknoten und der Endknoten derselbe ist. In vielen Fällen erhält man mit dem offenen CPP einen kürzeren Pfad, weil der Pfad nicht zwingenderweise auf dem Startknoten enden muss.

Eine optimale Chinese Postman Tour, welches die Lösung des CPP verlangt, ist durch einen Eulerkreis gegeben. Der Begriff Eulerkreis kommt aus der Graphentheorie und ist ein Zyklus in einem Graphen, der jede Kante genau einmal enthält. Der Eulerkreis beginnt und endet auf demselben Knoten. Eine Spezialisierung dessen ist der offene Eulerzug, bei dem End- und Anfangsknoten nicht von Bedeutung sind, aber trotzdem jede Kante in dem Pfad enthalten ist.

Ist ein Eulerkreis im Graphen gegeben, ist das Auffinden des Zyklus trivial. Das Auffinden eines Eulerkreises wäre durch den Algorithmus von Fleury gelöst. Man baut einen Spannbaum des Graphen auf, bestimmt irgendeinen Knoten des Graphen als Start- und Endknoten und wählt irgendeine Kante zu einem Folgeknoten, wobei diese Kante aus dem Spannbaum gestrichen wird. Bei der Wahl der Kante beachtet man, dass die Kanten, die zu einem Brücken-Knoten (ein Knoten der zum Anfangsknoten führt) führen, als letztes gewählt werden, genau dann, wenn keine anderen Kanten von dem aktuellen Knoten mehr wegführen.

In einem gerichteten Graphen, wie Zustandsautomaten oder Aktivitäten in der SysML, ist ein Eulerkreis vorhanden, wenn der Graph stark zusammenhängend ist und für jeden Knoten des Graphen gilt, dass der Eingangsgrad und der Ausgangsgrad des Knoten gleich ist. Bei einem stark zusammenhängenden Graphen gibt es von jedem Knoten aus einen Pfad zu jeden anderen Knoten. Der Ausgangsgrad eines Knotens ist die Anzahl der vom Knoten wegführenden Kanten. Analog dazu ist der Eingangsgrad die Anzahl der Kanten eines Knotens, die zu ihm hinführen. Sind Eingangsgrad und Ausgangsgrad gleich, also führen genauso viele Kanten in den Knoten wie aus den Knoten, nennt man den Knoten ausbalanciert.

Eine Voraussetzung zum Verwenden des hier verwendeten CPP Algorithmus ist, dass der Graph stark zusammenhängend ist. Bei Zustandsautomaten beispielsweise ist es in der Regel ein gutes Design , wenn von jedem Zustand aus jeder andere Zustand erreichbar ist. Gegeben sei ein Beispiel einer Menüführung. Ist die Menüführung in einem Zustandsautomaten abgebildet und gibt es einen oder mehrere Zustände, aus denen die anderen Zustände nicht mehr aktivierbar sind, kann das ein möglicher Indikator für ein schlechtes Design sein. Allerdings gibt es auch Begebenheiten, in denen Zustandsautomaten nicht stark zusammenhängend sein müssen, wo der hier verwendete Algorithmus nicht anwendbar ist. Für solche Begebenheiten müssen andere Algorithmen für dieses Testsendekriterium verwendet werden. Eine andere Vorgehensweise für die Erschließung von Testfällen mit dem Zustandsübergangsüberdeckungskriteriums wäre der Aufbau eines Übergangsbaums, wie im vorigen Kapitel beschrieben. Dadurch erhält man aber auch statt nur einem Testfall mehrere Testfälle.

Ist ein Eulerkreis im Graphen gegeben, ist das CPP schnell gelöst. Zur Überprüfung, ob ein Eulerkreis im Graphen existiert, reicht es zu überprüfen, ob die beiden oben beschriebenen Eigenschaften im Graphen gegeben sind. Allgemein wird die Eigenschaft des stark zusammenhängenden Graphen vorausgesetzt, um das CPP zu lösen. Die zweite Eigenschaft, dass alle Knoten ausbalanciert sind, muss überprüft werden. Gegeben seien folgende formale Eigenschaften für gerichtete Multigraphen:

$v \in$ V, wobei V die Menge aller Knoten ist

$d^-(v)$ Eingangsgrad des Knoten v. Anzahl der Kanten, die in den Knoten v führen

$d^+(v)$ Ausgangsgrad des Knoten v. Anzahl der Kanten, die aus den Knoten v führen

$\delta(v)$ Differenz zwischen d^+ und d^- $(\delta(v) = d^+(v) - d^-(v))$

D^+ Die Menge der Knoten v für die gilt, dass $\delta(v) > 0$ ist. $D^+ = \{ \text{v} \mid \delta(v) > 0\}$

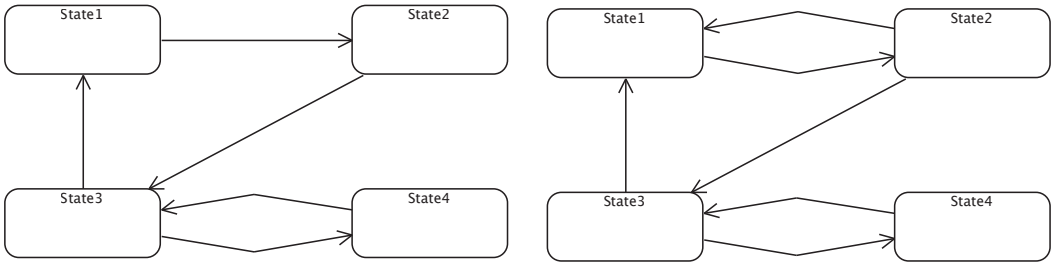
D^- Die Menge der Knoten v für die gilt, dass $\delta(v) < 0$ ist. $D^- = \{ \text{v} \mid \delta(v) < 0\}$

$i \rightsquigarrow j$ Der Pfad mit den kleinsten Kosten von Knoten i nach Knoten j

c_{ij} Die Kosten für einen Pfad von Knoten i nach Knoten j

$k = \sum_{v \in D^+} \delta(v)$ Die Anzahl der extra Pfaden für einen Euler-Kreis

f_{ij} Anzahl der Traversierungen des Pfades $i \rightsquigarrow j$



(a) Zustandsautomat mit Eulerkreis (b) Zustandsautomat ohne Eulerkreis

Abbildung 29: Beispiel für zusammenhängende Zustandsautomaten

Für jeden Knoten v wird der Ein- und Ausgangsgrad ($d^-(v) und d^+(v)$) bestimmt und das $\delta(v)$ berechnet. Wenn $\delta(v) = 0$ ist bedeutet das, dass der Knoten v ausbalanciert ist. Sind die Mengen D^+ und D^- leer, liegen nur ausbalancierte Knoten vor und es existiert ein Eulerkreis im vorliegenden Graphen. Ist dies aber nicht der Fall, müssen weitere Anstrengungen vorgenommen werden, um einen Eulerkreis aus dem Graphen zu erzeugen.

Zum Ausbalancieren der Knoten nimmt man sich die Knoten in den Mengen D^+ und D^- vor. Kanten zwischen den Knoten der beiden Mengen würden die Delta der Knoten in beiden Mengen gegen Null laufen lassen. Zieht man also eine Kante von einem Knoten i $\in D^-$ nach einen Knoten j $\in D^+$, so werden oder nähern sich die Werte der Delta $\delta(i)$ und $\delta(j)$ der Null. Zunächst zieht man virtuelle Kanten zwischen den Knoten der beiden Mengen. Alle Kanten des Kreuzproduktes zwischen D^+ und D^- sind möglich. Grundsätzlich müssen $k = \sum_{v \in D^+} \delta(v)$ virtuelle Kanten erzeugt werden, denn die Summe der $\delta(i)$ aller Knoten i $\in D^-$ ist im Betrag immer die gleiche wie die Summe der $\delta(j)$ aller Knoten j $\in D^+$.

Die Kombinationen der virtuellen Kanten aus dem Kreuzprodukt sind durch die Delta der Kanten in den beiden Mengen D^+ und D^- beschränkt. Im schlimmsten Fall können bis k! Kombinationen existieren, die es anhand der Kosten ihrer Pfade zu evaluieren gilt. Das Finden einer Kombination von Paaren beider Mengen ist trivial:

1. Man nimmt sich irgendeinen Knoten i $\in D^-$ vor, erhöht den Ausgangsgrad $d^+(i)$ des jeweiligen Knotens i um eins, zieht eine virtuelle Kante zu irgendeinen Knoten j $\in D^+$ und erhöht den Eingangsgrad $d^-(j)$ des zweiten Knotens j um eins.
2. Die beiden zuvor gewählten Knoten i und j werden hinsichtlich ihrer Delta $\delta(i)$ und $\delta(j)$ untersucht. Wenn ein Knoten ausbalanciert ist, wird er aus der jeweiligen Menge entfernt.
3. Falls noch Knoten in der Menge D^- existieren, wird mit Schritt 1 weiter gemacht, bis keine Knoten mehr vorhanden sind.
4. Zuletzt werden die virtuellen Kanten anhand ihrer Pfade mit den geringsten Kosten aufgelöst. Das heißt, dass der Pfad $i \rightsquigarrow j$ gefunden werden muss.

Mit diesem Algorithmus lässt sich eine Kombination finden, mit der man die Knoten des Graphen einfach ausbalanciert. Hierbei wurden aber nicht die Kosten und die Längen der Pfade, die sich hinter den virtuellen Kanten verstecken, berücksichtigt. Somit erhält man nach der Auflösung der virtuellen Kanten mit hoher Wahrscheinlichkeit zunächst nur ein lokales Minimum des CPP. Das Auffinden aller möglichen Kombinationen wiederum ist nicht mehr so trivial und die Herangehensweise und die Algorithmen zum Erschließen der Kombinationen sollen an dieser Stelle nicht erörtert werden, weil sie sehr komplex sind und über den Rahmen dieser Arbeit hinausgehen. In [39] wird aufgegriffen, welche Algorithmen dieses Problem lösen können.

Nachdem man alle virtuellen Kanten hat, löst man die virtuellen Kanten auf, indem man über den zuvor gebildeten Spannbaum die kürzesten Pfade zwischen diesen Knoten berechnet. Die durch die Pfade verursachten extra Traversierungen der Kanten, werden in k vermerkt. An dieser Stelle soll davor gewarnt werden, negative Kosten in den Graphen zu verwenden. Sie können negative Zyklen verursachen, was dazu führen würde, dass der kürzeste Pfad negativ unendlich wäre. Daher wird nach einem Auffinden eines negativen Zyklus der Algorithmus abgebrochen.

Für die Berechnung der kostengünstigsten Pfade werden die Kosten c_{ij} einer Kante und die Anzahl der Kanten in einem Pfad herangezogen. Die kostengünstigsten Pfade können durch den Floyd-Warshall-Algorithmus berechnet werden. In diesem Greedy-Algorithmus wird eine Matrix aller Kanten erstellt. Die Matrix beherbergt die günstigsten Kosten für Pfade zwischen allen Knoten. Der Algorithmus durchläuft alle Knoten und notiert sich die Kanten und ihre Kosten. Mit diesen Informationen werden die Pfade zwischen allen Knoten berechnet. Die Berechnung beruht auf der Erkenntnis, dass wenn eine Kante von Knoten a nach Knoten b und eine Kante von Knoten b nach Knoten c führt, existiert auch ein Pfad von Knoten a nach Knoten c. So berechnet der Algorithmus alle Pfade von allen Knoten der Knotenmenge zu allen Knoten derselben Knotenmenge.

Der Floyed-Warshall-Algorithmus, der weiter unten zu sehen ist, überprüft bei der Erstellung der Pfade, ob der Pfad schon in der Matrix definiert ist und, falls ein Pfad schon existiert, ob

der aktuelle Pfad unter Umständen kostengünstigster ist. Auf diese Art und Weise werden die kostengünstigsten Pfade $i \rightsquigarrow j$ erschlossen.

```
1  void leastCostPaths(){
2    for( int k = 0; k < N; k++ )
3      for( int i = 0; i < N; i++ )
4        if( defined[i][k] )
5        for( int j = 0; j < N; j++ )
6          if( defined[k][j] && (!defined[i][j] || c[i][j] > c[i][k]+c[k
               ][j]) ){
7            path[i][j] = path[i][k];
8            c[i][j] = c[i][k]+c[k][j];
9            defined[i][j] = true;
10           if( i == j && c[i][j] < 0) return; // stop on negative cycle
11         }
12 }
```

Listing 1: Algorithmus von Floyd und Warshall

Nachdem nun die virtuellen Kanten in die kostengünstigsten Pfade aufgelöst werden können, gilt es zu untersuchen, ob der zuvor erstellte Spannbaum des Graphen mit den extra Kanten in k (der nun ein Eulerkreis ist) wirklich optimal, also der kostengünstigste Zyklus ist. Dahinter steckt das Optimierungsproblem, das kleinste $\phi = \sum c_{ij} f_{ij}$ zu finden. An den Kosten c kann nichts geändert werden, die sind von Anfang an gegeben. Die Anzahl f der Traversierungen der Kanten ist gegeben durch die Kombination der virtuellen Kanten. Die Kombinationen der virtuellen Kanten sind daher zu optimieren.

Der in dieser Arbeit verwendete Algorithmus löst das obige Optimierungsproblem mit dem Cycle Canceling Algorithmus. Statt bei der Belegung der Kombination der virtuellen Kanten eine Berechnung für den kleinsten Zyklus durchzuführen, wird wie oben beschrieben vorerst ein Eulerkreis gebildet und anhand eines Restbaumes überprüft, ob der Spannbaum der günstigste ist oder nicht. Der Restbaum ist ein Spannbaum aus den Knoten der Mengen D^+ und D^-. Anhand dieses Teilbaumes werden kürzere Pfade gesucht und die Anzahl f der Traversierungen der Kanten so verändert, dass die kürzeren Pfade verwendet werden. Wie genau das Cycle Canceling funktioniert ist in [39] beschrieben.

Gegeben seien die zwei Beispiele für Zustandsautomaten in Abbildung 29. Beide Zustandsautomaten zeigen einen zusammenhängenden Graphen. Das Beispiel (a) zeigt einen ausbalancierten Zustandsautomaten, sodass beide Eigenschaften für einen Eulerkreis gegeben sind. Das Beispiel (b) zeigt den gleichen Zustandsautomaten mit einer weiteren Transition, sodass der vorliegende Graph nicht mehr ausbalanciert ist und keinen Eulerkreis beinhaltet. Für die Berechnung eines Eulerkreises in beiden Beispielen sind folgende Schritte nötig:

1. Berechne die kürzesten Strecken zwischen allen Knoten im Graphen in einer Knoten-zu-Knoten-Matrix(siehe 1 Floyd-Warshall Algorithmus)
2. Überprüfe ob der Graph stark zusammenhängend ist. Wenn nein, breche den Algorithmus ab.

3. Suche alle nicht ausbalancierten Zustände und unterteile sie in die Mengen D^+ und D^-. Falls alle Zustände schon ausbalanciert sind, fahre mit dem Schritt 8 fort.
4. Berechne eine paarweise Kombination zwischen den Knoten aus den Mengen D^+ und D^- mit virtuellen Kanten, wie oben beschrieben. Dabei reicht es vorerst ein lokales Minimum zu berechnen.
5. Die paarweise Kombination werden in einen Spannbaum eingefügt. Dieser Spannbaum wird Restbaum genannt und beinhaltet nur die zusätzlichen virtuellen Kanten zwischen den Knoten, um einen Eulerkreis zu erhalten.
6. Die virtuellen Kanten werden in Kanten bzw. Pfaden (folgen von Kanten) anhand der Knoten-zu-Knoten-Matrix aufgelöst.
7. Auf dem Restbaumes mit den aufgelösten virtuellen Kanten, wird der Cycle Canceling Algorithmus angewendet, um eine optimale Lösung zu erhalten. Sind die optimalen Kanten-Kombinationen gefunden, werden die Kanten in den Spannbaum des Graphen für den Zustandsautomaten hinzugefügt.
8. Führe den Algorithmus von Fleury aus, um den Eulerkreis im Graphen des Zustandsautomaten zu finden.

Das Beispiel (a) aus Abbildung 29 beinhaltet bereits einen Eulerkreis und würde dazu führen, dass der Algorithmus die Schritte 4 bis 7 komplett auslassen würde. Der Eulerkreis ist folgendermaßen definiert: State1 $\rightarrow$ State2 $\rightarrow$ State3 $\rightarrow$ State4 $\rightarrow$ State5 $\rightarrow$ State1

An dieser Stelle ist für die Notation nicht wichtig, welche Transition zwischen den Zuständen aktiviert wird, weil zwischen den Zuständen nur eine Transition in eine Richtung vorhanden ist. Falls aber mehr Transitionen zwischen den Zuständen in ein Richtung führen, sollte die die Information vorhanden sein, um welche Transition es sich handelt, denn es könnten Transitionen mit verschiedenen Kantengewichten vorliegen.

Für den Beispiel (b) sieht es anders aus. Das Beispiel ist nicht von vornherein ausbalanciert und muss noch ausbalanciert werden. Die Transition vom Zustand State2 zum Zustand State1 bewirkt, das die Zustände State1 und State2 nicht ausbalanciert sind. Das Ausbalancieren dieser beiden Knoten ist in diesem Fall einfach, weil es nur die Möglichkeit gibt eine virtuelle Kante von State1 zu State2 zu ziehen, die in der Auflösung zu einer Kante und nicht zu mehreren Kanten wird. Die Abbildung 30 zeigt den Eulerkreis beginnend vom Zustand1. Hier sollte beachtet werden, dass die Transition von Zustand1 nach Zustand2 zweimal genommen wird.

Somit erhält man für Beispiel (b) den folgenden Eulerkreis: State1 $\rightarrow$ State2 $\rightarrow$ State3 $\rightarrow$ State4 $\rightarrow$ State3 $\rightarrow$ State1 $\rightarrow$ State2 $\rightarrow$ State1

Die Kantengewichte (Kosten der Kanten) werden in dem Algorithmus, falls keine angegeben wurden, mit 1 belegt. Für das Belegen der Kantengewichte könnte man in den Zustandsautomaten die Transitionen mit Stereotypen belegen, die einen Integer Wert in einer Eigenschaft für das Kantengewicht annehmen können.

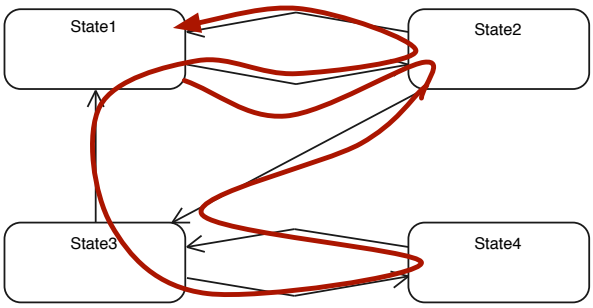


Abbildung 30: Eulerkreis in Beispiel (b) aus Abbildung 29

Obwohl in dieser Arbeit die Kantengewichte raus gelassen wurden, gäbe es schon sinnvolle Anwendungsbereiche. Man kann sie zum Beispiel dafür nutzen, um im Zustandsautomaten für den Algorithmus anzugeben, welche Kanten für die Bildung des Eulerkreises priorisiert werden sollen. Andererseits entstehen dadurch unter Umständen längere Pfade durch den Graphen und dadurch längere Testfälle. Indem man aber im Zustandsautomaten diejenigen Transitionen mit großen Kantengewichten belegt, die zu Sonderfällen oder in der Realität selten vorkommenden Ereignissen führen, erhält man Pfade durch den Graphen, bei denen es vermieden wurde unter Umständen Abkürzungen durch diese Transitionen zu nehmen. Somit erhält der Modellierer eine Möglichkeit den Fluss für die generierten Pfade ein wenig zu lenken.

4.2.2. Eigene Ansätze

Ein Algorithmus für das "open CPP" müsste den erzeugten Pfad nicht auf den Startknoten enden lassen. Für die Lösung des Problems reicht ein Pfad aus, der alle Kanten mindestens einmal verwendet. Zur Berechnung eines "open CPP" Pfades durch den Graphen, kann der Algorithmus für das Lösen des CPP genommen werden. In der Regel sind die Pfade des "open CPP" kürzer als die des CPP, außer man hat einen Graphen mit einen Eulerkreis vorliegen, dann ist die optimale Lösung für ein "open CPP" die optimale Lösung des CPP. Falls der Graph nicht ausbalanciert ist, muss dieser ausbalanciert werden. Vor dem Ausbalancieren allerdings, sollte eine virtuelle Kante von einem Knoten aus D^- auf einen virtuellen Knoten geführt werden, der den Beginn des CPP Algorithmus definiert. Der Knoten aus D^- definiert den Endknoten. Von dem virtuellen Knoten aus, wird eine virtuelle Kante auf den Knoten gezogen, der den Startknoten definiert. Die Abbildung 31 zeigt skizziert den virtuellen Knoten und die virtuellen Kanten.

Diesen Vorgang wiederholt man mit allen Knoten in D^- und untersucht die Gesamtkosten aller Kantengewichte für den jeweils berechneten Eulerkreise. Der Graph mit den kleinsten Gesamtkosten für einen Eulerkreis, endend auf einen Knoten aus D^-, wird für den optimalen

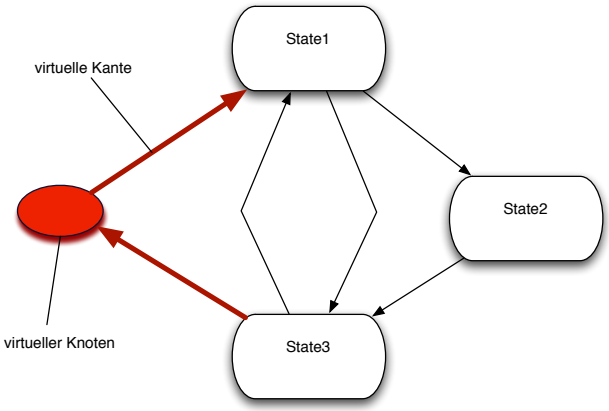


Abbildung 31: Virtueller Knoten und virtuelle Kanten für die Definition eines Start- und Endknoten

Pfad genommen. Ein Nachteil, der bei diesem Verfahren auftritt, ist das für alle v $\in$ D^- ein neuer Graph mit virtuellem Knoten erzeugt wird, welcher jeweils den CPP Algorithmus durchlaufen muss. Damit ruft man mit dem "open CPP" Algorithmus x = $\mid D^- \mid$ mal den CPP Algorithmus auf, um zu überprüfen, welche offene Eulertour die kostengünstigste ist.

Der virtuelle Knoten dient als Start- und Endknoten für den CPP Algorithmus. Anders als die vorrigen virtuellen Kanten, können diese virtuellen Kanten nicht aufgelöst werden, weil sie zu oder von einem virtuellen Knoten führen. Diese Kanten werden aber im Ergebnis-Pfad wieder weggeschmissen.

Möchte man den Algorithmus für die Zustandsüberdeckung optimieren, kann man im Fleury Algorithmus Bedingung setzen, dass der Algorithmus beendet, wenn jeder Zustand einmal aktiviert wurde. Somit hätte man einen Pfad, der alle Zustände mindestens einmal aktiviert und ein lokales Minimum darstellt. Dieser Pfad kann noch kürzer sein, als der Pfad für die Zustandsübergangsabdeckung des "open CPP".

Algorithmus zum Auslesen der Stimuli-Generatoren

Die Aktivitäten im Modell mit dem Stereotypen «StimuliActivity» beschreiben Möglichkeiten der Stimulierung des SuTs mihilfe der Pfade. Diese Pfade werden von einem Algorithmus ausgelesen und in einer If-Else-Struktur implementiert. Der Algorithmus, zum Auslesen der Pfade und zur Generierung der If-Else-Struktur ist wie folgt definiert:

Erzeuge eine Liste und füge das Aktion, auf dem das “InitialNode”-Element zeigt, zur Liste hinzu. Rufe eine Funktion auf, die diese Liste als Parameter fordert und eine Liste wieder als Rückgabewert ausgibt. In dieser Funktion wird das nächste Element auf dem Kontroll-Pfad betrachtet. Ist das Element weder ein "DecisionNode"-Element oder ein "FinitNode"-Element, dann füge das Element der Liste hinten an. Ist das Element ein "FinitNode"-

Element, so beende die Funktion und gebe die Liste als Rückgabewert zurück. Für den Fall, dass das Element ein "DecisionNode"-Element ist, rufe für jeden vom "DecisionNode"-Element ausgehenden Pfad dieselbe Funktion in einer Rekursion auf und füge die Rückgabewerte der Aufrufe an das Ende der Liste. Vor jedem rekursiven Aufruf wird der Index des letzten Elements der Liste gemerkt und in einer If-Else-Struktur für das "DecisionNode"-Element in der Liste notiert.

Am Ende erhält man eine Liste, die in einer If-Else-Struktur abgebildet werden kann. In dem Testfall aus Anhang B ist so eine generierte If-Else-Struktur abgebildet. Diese Struktur entstammt der Stimuli-Generator aus Abbildung 20.

4.3. Beispiel

Die in den vorrigen Kapiteln beschriebenen Gestaltungsoptionen für das Modellieren des Soll-Verhaltens und der Aufbau des Testmodells, wie es in dieser Arbeit verwendet wird, ist in der Abbildung 32 skizziert.

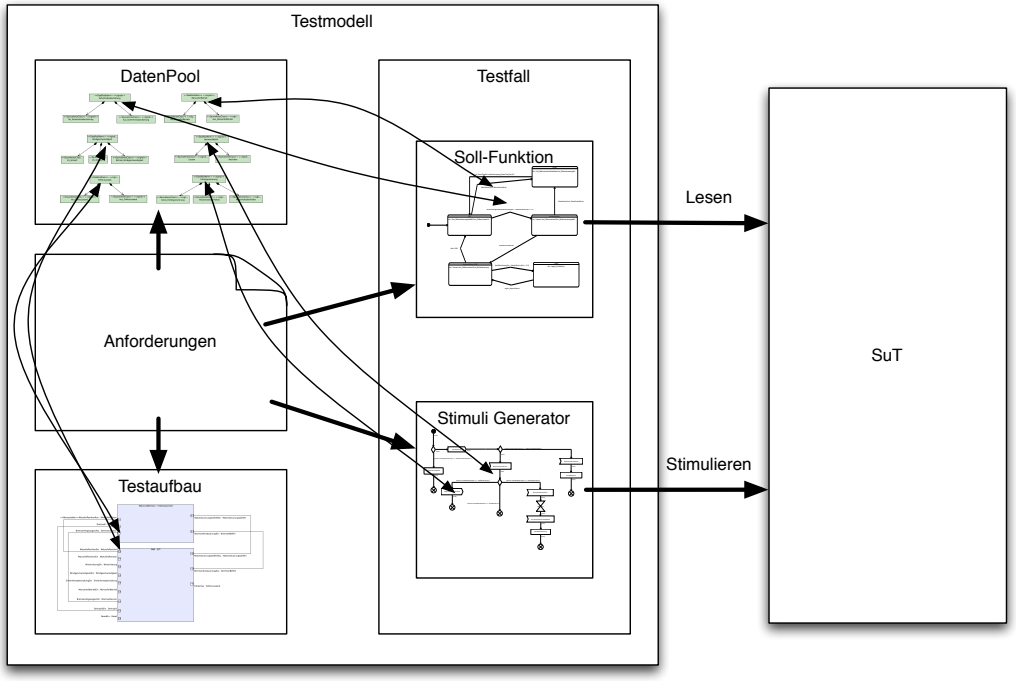


Abbildung 32: Skizze des Aufbaus des Testmodells und der Beziehungen des Testmodells mit dem SuT

Nach dem Import der Anforderungen in das Modell, können anhand der Informationen aus den Anforderungen die Daten für die entsprechenden validen Signale an den Eingängen und den Ausgängen des SuT im Datenpool definiert werden. Weiterhin werden über die Anforderungen die Beziehungen des SuT mit den Umgebungskomponenten in einer Testaufbau-Struktur für den Testaufbau abgebildet. Wie in 32 zu sehen, werden insoweit Beziehungen zwischen dem Datenpool und der Struktur aufgestellt, dass jeder Port des SuT in der Testaufbau-Struktur einen eigenen Datentypen im Datenpool besitzt.

Nachdem die Struktur- und Datenmodelle definiert sind, können nun die anwendungsbasierten Testfälle modelliert werden. Jeder Testfall in dieser Arbeit enthält zwei Teilmodelle. Das Modell für die Beschreibung der Soll-Funktion (zu sehen in Abbildung 33) und das Modell für die Beschreibung der gültigen Eingangsfolgen von Stimuli an das SuT.

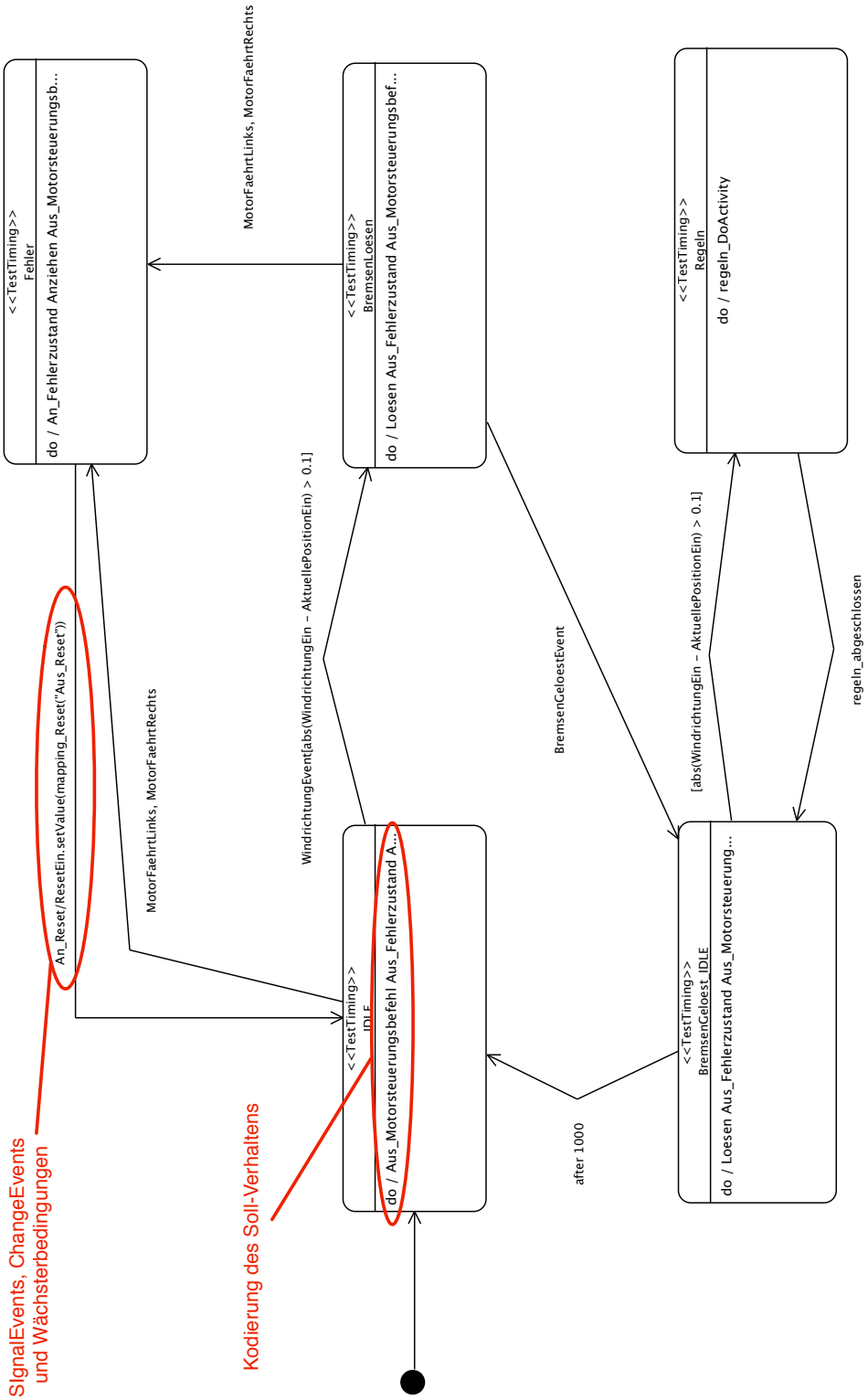


Abbildung 33: Ein Beispiel für das Verhalten des WNS im normalen Betriebsmodus

Für die Beschreibung der Soll-Funktion werden die Zustandsautomaten aus der SysML verwendet. Dieses Verhaltensmodell wird während der Testdurchführung für den jeweiligen Testfall nebenläufig zum SuT ausgeführt. Es beinhaltet die Informationen, in welchen internen Verhaltenszustand (soweit die Modellierung möglich ist) das SuT sich befinden und anhand des Zustandes, welche Werte an den Ausgängen des SuT anliegen sollten. Daraus berechnet man Vergleichsprüfungen, die für das Testergebnis herangezogen werden. Über die definierten Events weiß das Modell der Soll-Funktion, wann und in welchen Zustand gewechselt werden muss. In der Abbildung 33 ist der test-relevante Teil des Verhaltens des SuT modelliert. Es beschreibt folgendes Verhalten, welches zunächst eine ruhenden Position der Gondel vorraussetzt:

- Wenn eine große Differenz zwischen der aktuellen Position der Gondel und der aktuellen Windrichtung erkannt wurde, möchte das WNS die Gondel Nachregeln und gibt zunächst den Befehl, die Bremsen zu lösen.
- Daraufhin wartet das WNS auf das Signal “BremsenGeloest”.
- Ist die Differenz nach dem Lösen der Bremsen immer noch ausreichend groß, wird die Gondelposition nachgeregelt.
- Nach dem Regeln der Gondel, wird eine Sekunde lang gewartet, bis der Befehl vom WNS gegeben wird, die Bremsen anzuziehen.

In dem Modell wurden auch Übergänge in den Fehlermodus eingebaut. Dadurch kann die Fehlerdiagnose des SuT auf mindestens zwei Zeitpunkten in dem internen Verhalten getestet werden, nämlich der Wechsel des SuT in den Fehlermodus, nachdem erkannt wurde, dass die Gondel sich bewegt, obwohl die Bremsen angezogen sind und kein Befehl für den Motor zum Drehen gegeben wird.

Die Eingaben an das SuT werden durch eine Aktivität beschrieben. Die Aktivität mit dem Stereotypen "StimuliGenerator" enthält die Informationen für die gültigen test-relevanten Eingangsfolgen von Stimuli. In der Abbildung 34 ist so eine Aktivität zu sehen. Sie hat verschiedene Pfade, die durch Parameter bestimmt werden. In der Reihenfolge dieser Pfade nach, werden nach und nach die Aktionen ausgeführt. Es gibt Aktion zum Senden von Signalen, zum Warten auf bestimmte Signale des SuT oder der Umgebungskomponenten (in diesem Fall Stubs), zum Warten einer gewissen Zeit und zum Ausführen von Funktionen auf den Stubs oder von direkt eingebettetem Programmcode. Die Pfade werden mit dem Algorithmus aus Kapitel 4.2.2 aus der Aktivität extrahiert und mittels Parameter ausgeführt. Welches Signal gesendet wird oder auf welches Signal gewartet wird, wird von den angegebenen Parametern bestimmt.

Bei der Modellierung der beiden Verhaltensmodelle ist darauf zu achten, dass die Aktionen, die vom Stimuli-Generator angestoßen werden, in dem Modell der Soll-Funktion auch berücksichtigt werden. In den beiden Modellen aus den Abbildungen 34 und 33 ist absichtlich eine Aktion im Stimuli-Generator eingebaut, die nicht in der Soll-Funktion berücksichtigt wird, sodass das SuT korrekterweise ein Verhalten anstößt, welches die Soll-Funktion als Fehler

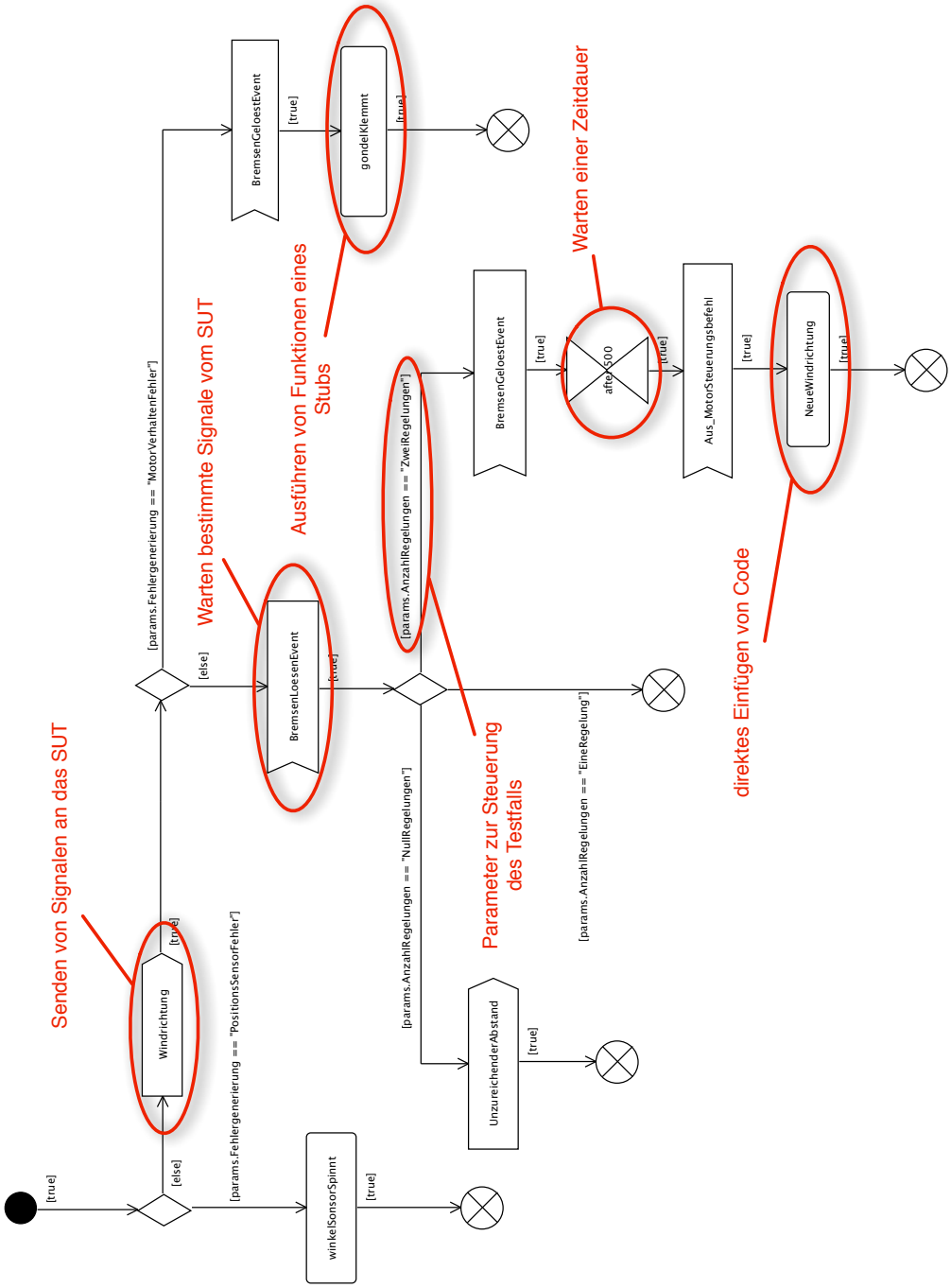


Abbildung 34: Ein Beispiel für einen Stimuli-Generator

detektiert. Der “Fehler” liegt genaugenommen im SuT-Verhalten. Die betroffene Aktion ist in diesem Falle der Aufruf der Stub-Funktion “GondelKlemmt”, eine klemmende Gondel zu simulieren.

Bereits an diesem einfachen Beispiel wird erkennbar, wie vielfältig die Gestaltungsmöglichkeiten von Testfällen mit den modellbasierten Testänsätzen sind.

5. Technologische Aspekte der Umsetzung

In diesem Kapitel wird der Aufbau der Testumgebung, die Testdurchführung und die daraus resultierenden Ergebnisse diskutiert. Die Konzeption aus dem vorherigen Kapitel wird für die Testdurchführung angewandt. Im ersten Abschnitt dieses Kapitels wird der Test-Aufbau und die Gewinnung der Testdaten erläutert. Dabei stehen konzeptionelle Fragen über die Techniken zum Beobachten und Manipulieren der Tests und Designentscheidungen beim Aufbau im Vordergrund. Der zweite Abschnitt ist ein kurzer Abriss über die systematische Vorgehensweise der Tests und die Sammlung von Resultaten. Die Interpretation dieser Ergebnisse und die Zurückführung der Resultate auf die Tests wird im letzten Drittel dieses Kapitels aufgegriffen.

5.1. Testdesign

Für einen MiL-Testaufbau werden einige Komponenten benötigt. In Kapitel 2.1 wird ein klassischer MiL-Testaufbau eingeführt. Für den Aufbau wird eine Laufzeitumgebung für das SuT, ein Simulator und ein HMI benötigt. Das HMI ist zwar für die Ausführung der Tests nicht notwendig, es ist aber hilfreich, den Test während der Durchführung zu beobachten und gegebenenfalls manipulieren oder abbrechen zu können. Die Aufgabe bei solch einem Aufbau ist es, eine geeignete Laufzeitumgebung für das Simulationsmodell und einen passenden Simulator zu erschließen.

Matlab/Simulink-Modelle fordern einige besondere Eigenschaften an eine Laufzeitumgebung. Die Umgebung muss mathematische Berechnungen durchführen können. Für die Simulation dieser Modelle müssen die Berechnungen zyklisch zu vordefinierten zeitdiskreten Simulationsschritten durchgeführt werden, um die Eigenschaft eines zeit-kontinuierlichen Systems zu simulieren. Die Größe der zeitdiskreten Simulationsschritte sollte so gewählt werden, dass möglichst Tendenzen von Werteänderungen in der Simulation enthalten sind. Umso kleiner die Simulationsschrittweite ist, desto kleiner ist die Wahrscheinlichkeit, eine möglicherweise gewichtige Werteänderung in der Zeit zu versäumen. Für das Fallbeispiel-Modell wurde beispielsweise eine feste Simulationsschrittweite von 0,1 s verwendet. Das heißt, dass in einer Simulation alle 100 ms, gemessen an der physikalischen Zeit, eine Berechnung aller Werte des Simulationsmodells durchgeführt wird. Angenommen die Anforderungen an das zu simulierende System beschreiben Reaktionszeiten deutlich unter 100ms, wäre diese Simulationsschrittweite viel zu groß gewählt.

HIL-Simulatoren sollten ca. um den Faktor 10 schneller sein als die zu testende Hardware. Das gilt auch für die Simulationsschrittweite von MiL-Simulatoren. Dabei ergeben sich beispielsweise für moderne Steuergeräte im Automotive-Bereich Simulationsschrittweiten von ca. 500 μs [7]. Für das eher träge System in dem Fallbeispiel reichen Simulationsschrittweiten im Millisekunden-Bereich völlig aus.

An dieser Stelle liegt es nahe, für das Simulationsmodell die eigene Laufzeitumgebung von Matlab/Simulink zu verwenden. Diese Laufzeitumgebung wurde speziell für Matlab/Simulink-Modelle entwickelt und funktioniert ohne viel Zutun auf Anhieb. Ein Test-Aufbau könnte wie in Abbildung 35 aussehen.

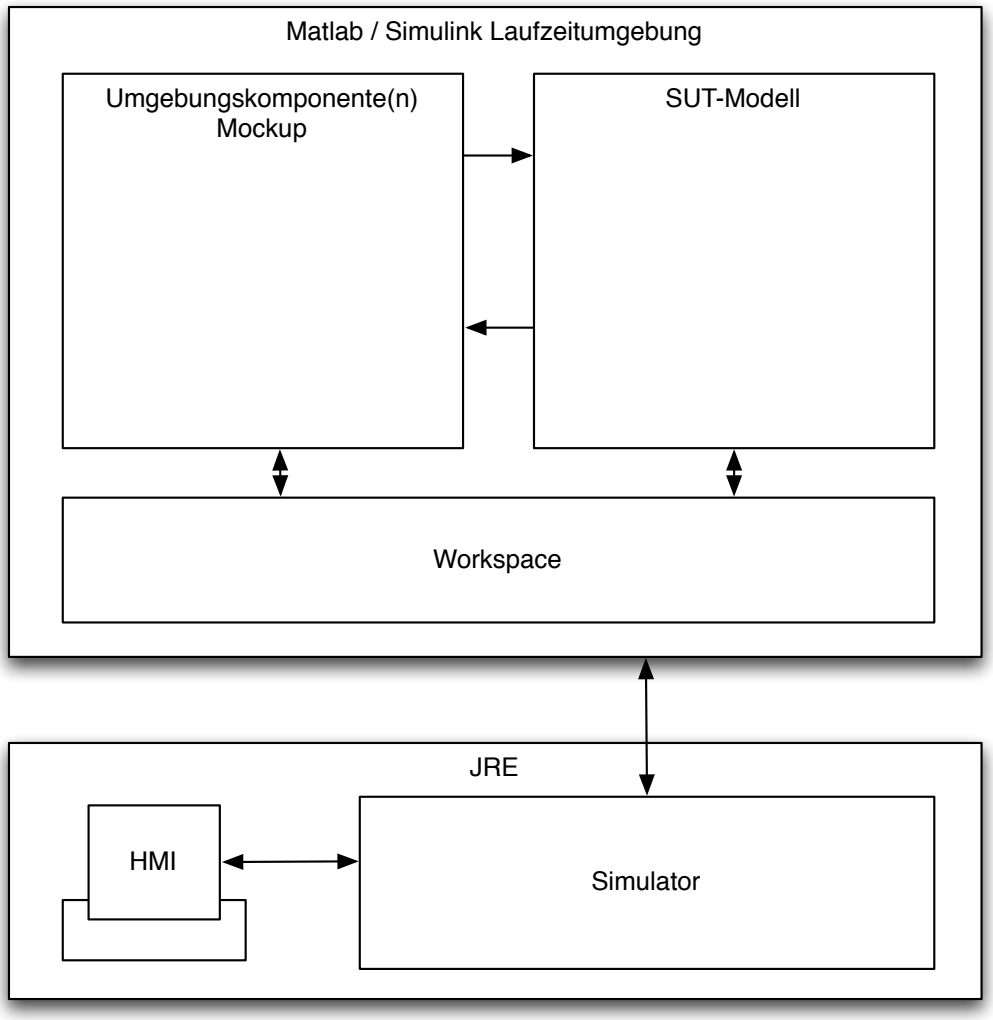


Abbildung 35: MiL Test-Aufbau mit Matlab/Simulink Laufzeitumgebung

Die Arbeitsumgebung von Matlab/Simulink besitzt Funktionen zum Definieren von Signalen. Signale für die Eingänge oder die Werte Variablen im Simulationsmodell können über die Matlab-Arbeitsumgebung definiert werden. Es können Werteänderungen über die Zeit gesetzt werden, um das Modell während der Simulation zu stimulieren. Weitere Funktionen ermöglichen das Beobachten und Loggen des Reaktionsverhaltens.

Mit den angeführten Funktionen ist es möglich, Testfälle manuell einzugeben und ausführen zu lassen. Die Ergebnisse können anhand des aufgezeichneten Reaktionsverhaltens offline erschlossen werden. Dadurch, dass nicht während der Simulation auf das Reaktionsverhalten eingegangen werden kann, ist diese Methodik für reaktives Testen ungeeignet. Es wird ein Prozess benötigt, der den Test reaktiv treibt. Die Abbildung 35 zeigt einen Aufbau mit einem Simulator zum Treiben des Tests.

Eine Möglichkeit zur Steuerung des Testverlaufs wäre das Aufrufen von Matlab/Simulink Funktionen aus einem weiteren Programm. Mithilfe des Java-to-Matlab-Interface (JMI) Adapters sind Aufrufe aus Java-Programmen heraus möglich. Die Adapter Bibliothek JMI (siehe [42] und [17]) ist ein OpenSource Projekt, welches schon in verschiedenen industriellen wie universitären Projekten Verwendung fand. Der Adapter ist jedoch kein anerkannter Standard und wird weder seitens Matlab offiziell unterstützt noch gibt es eine ausführliche Dokumentation dazu. Es gibt einige Projekte, die sich mit dieser Problematik auseinander setzen. Ein von dem Bundesministerium für Bildung und Forschung gefördertes Projekt für eine Standardisierung einer Java zu Matlab Schnittstelle wäre beispielsweise MAJA [8].

Die Verwendung des JMI Adapters hat allerdings den Nachteil, dass man einen Java-Simulator selber entwicklen muss. Dazu gehört die Signalaufbereitung, die Ausführung der definierten Testfälle, die Interkommunikation mit dem Simulationsmodell sowie die Einhaltung der zeitlichen Bedingungen während der Simulation. Die Erstellung solch einer Testumgebung wäre mit sehr viel Aufwand verbunden und würde den Rahmen dieser Arbeit deutlich überschreiten. Daher wurde nach einer Testumgebung gesucht, die die Anforderungen für reaktives Testen mit Simulationsmodellen erfüllt.

Für die Tests in dieser Ausarbeitung wurde die Testumgebung Messina ausgewählt, weil sie unter anderem eine Laufzeitumgebung verwendet, die auch Matlab/Simulink-Modelle ausführen kann. Bei der Laufzeitumgebung handelt es sich um das Echtzeitbetriebssystem VxWorks von Windriver [44]. Für die Ausführung der Matlab/Simulink-Modelle bedarf es einer Aufbereitung der Modelle. Die Modelle werden nicht direkt ausgeführt, sondern in Programmcode transformiert, der schließlich vom Windriver Compiler [43] in für VxWorks ausführbare Dateien umgewandelt wird. An dieser Stelle soll darauf hingewiesen werden, dass die Transformation zu Programmcode nur durchgeführt wird, damit die Modelle auch in der VxWorks Umgebung zum Laufen gebracht und letztendlich getestet werden können. Wird der erzeugte Programmcode aber auch für das spätere Produkt verwendet, haben wir laut Definition kein MiL- sondern einen SiL-Testaufbau. Solch eine Situation wirft die Interpretationsfrage auf, ob es sich um einen MiL- oder SiL-Testaufbau handelt. Im weiteren Verlauf dieser Arbeit wird dieser Test-Aufbau als ein MiL-Testaufbau deklariert, weil der Focus dieser Arbeit auf dem Testen von Simulationsmodellen liegt und weniger auf den verschiedenen Integrationsstufen bei der Entwicklung eines Produktes.

Messina verwendet das VxWorks Betriebssystem als ein Targetsystem, auf dem das Testobjekt ausgeführt wird. Das Betriebssystem kann entweder auf einem dedizierten System, wie einer eingebetteten Hardware, oder virtualisiert auf dem lokalen Rechner, auf dem auch Messina ausgeführt wird, laufen. Messina kommuniziert über eine Socket-Verbindung mit

dem VxWorks. Das VxWorks ist so aufgebaut, dass ein oder mehrere Clients sich über eine Socket-Verbindung an das VxWorks verbinden und die Ausführung des Modells beobachten oder manipulieren können.

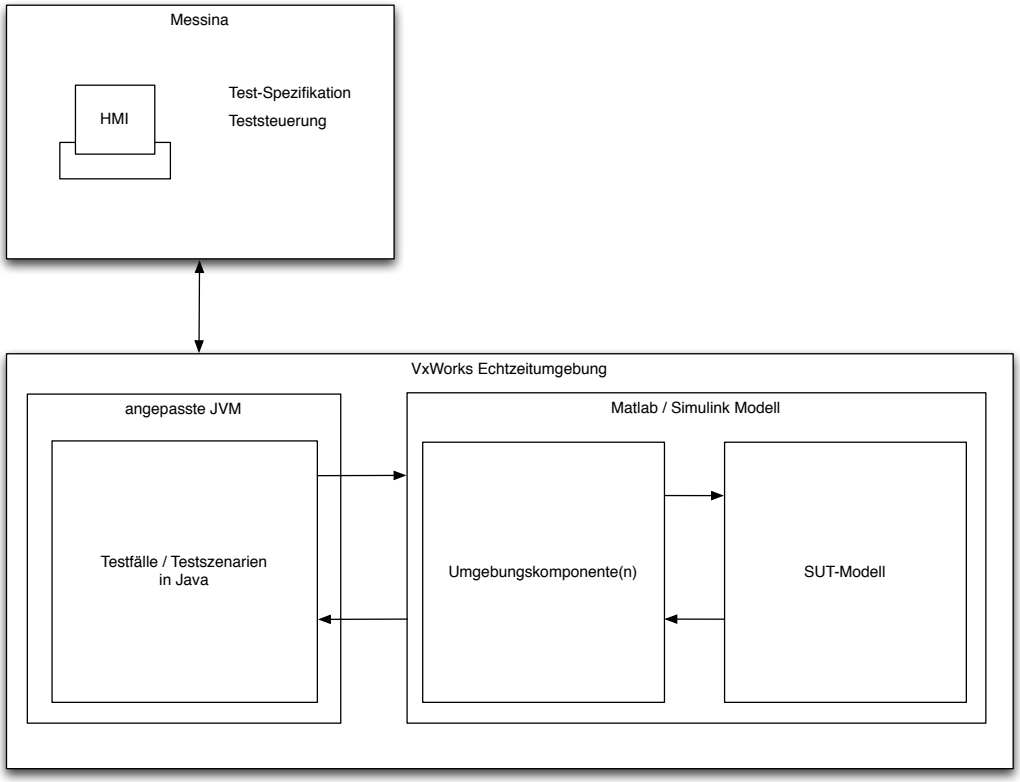


Abbildung 36: MiL-Testaufbau mit Messina und VxWorks

Auf dem Targetsystem ist eine für Test-Zwecke angepasste Java Virtual Machine installiert. Die JVM dient der Ausführung der Testfälle. Die Testfälle werden in Messina in der Programmiersprache Java definiert. Die Programmiersprache ist um einige Bibliotheken für das Testen mit VxWorks von Berner & Mattner erweitert worden.

Die Ausführung der Testfälle auf dem VxWorks System bietet einige Vorteile. Programme welche auf dem VxWorks laufen können über POSIX Message Queues kommunizieren. Das Treiben des Simulationsmodells über POSIX Message Queues ist schneller als eine TCP/IP Socket-Verbindung von außen. Man kann mit reaktiven Testfällen das Verhalten in Echtzeit testen. Des weiteren greifen der Simulator und das SuT auf die gleiche Zeit zu. Dadurch, dass das VxWorks in Echtzeit abläuft, ist die physikalische Zeit, die Simulierzeit und die Simulationszeit die gleiche. Es Bedarf keinen großen Aufwand der Synchronisation zwischen parallel laufenden Prozessen. Die Durchführung der Tests in Echtzeit zieht aber

auch den Nachteil mit sich, dass die Dauer der Tests sehr lang werden kann. Dieser Nachteil wird an dieser Stelle in Kauf genommen.

Messina übernimmt verschiedene Aufgaben in diesem Aufbau, die folgend aufgelistet sind:

- Testspezifikation
- Teststeuerung
- HMI
- Live-Anzeige
- Festhalten der Testergebnisse

Während der Testdurchführung nimmt Messina die einzelnen Ergebnisse der Testfälle auf. Die Testfälle beinhalten die Überprüfung des Soll-Verhaltens. Die Überprüfung des Soll-Verhaltens wird in den Testfällen online durchgeführt. Das heißt, dass der Vergleich zwischen dem reaktiven Verhalten des SuT und dem Soll-Verhalten während der Testdurchführung vorgenommen wird. Aus diesem Grund ist die Funktion zur Berechnung des Soll-Verhaltens in den Testfällen integriert.

Mittels der Reihenfolge der Testfälle beziehungsweise Testszenarien in Messina wird die Abfolge der Testfälle für die Testdurchführung bestimmt. In Messina gibt es nicht nur die Möglichkeit den Testdurchlauf zu starten, sondern auch zu stoppen. Über verschiedene Live-Anzeigen (Listen-Ansicht, grafische Ansichten, Zwei-Achsen-Graphen), lässt sich der Test an den Eingängen, Ausgängen und sonstige Eigenschaften des Simulationsmodells beobachten. Messina kann aber auch als HMI genutzt werden. Die Eingänge und die Variablen des Simulationsmodells lassen sich während der Simulation aus Messina heraus manuell manipulieren. Zusätzlich wird Messina genutzt, um die Testfälle zu spezifizieren und die Zuordnung der Schnittstellen zwischen den Testfällen und dem SuT herzustellen.

5.2. Vorgehen

In diesem Kapitel wird das Vorgehen zum modellbasierten Testen erörtert. Im Kapitel 3 wurden die Anforderungen des Fallbeispiels diskutiert und welche Informationen man aus ihnen gewinnen kann. Diese Informationen werden sowohl für die Entwicklung als auch für das Testen des Simulationsmodells gebraucht. In dieser Arbeit wird das SuT-Modell strikt vom Testmodell getrennt (siehe Kapitel 2.2.1). Das SuT-Modell ist ein Simulationsmodell basierend auf Matlab/Simulink. Die Informationen für die Testfallgewinnung werden aus einem separaten Modell bezogen, dem Testmodell. Das Testmodell ist in SysML modelliert und beinhaltet die Informationen für die systematische Testfallgenerierung. Die Abbildung 37 zeigt das systematische Vorgehen zum modellbasierten Testen von Simulationsmodellen in dieser Arbeit.

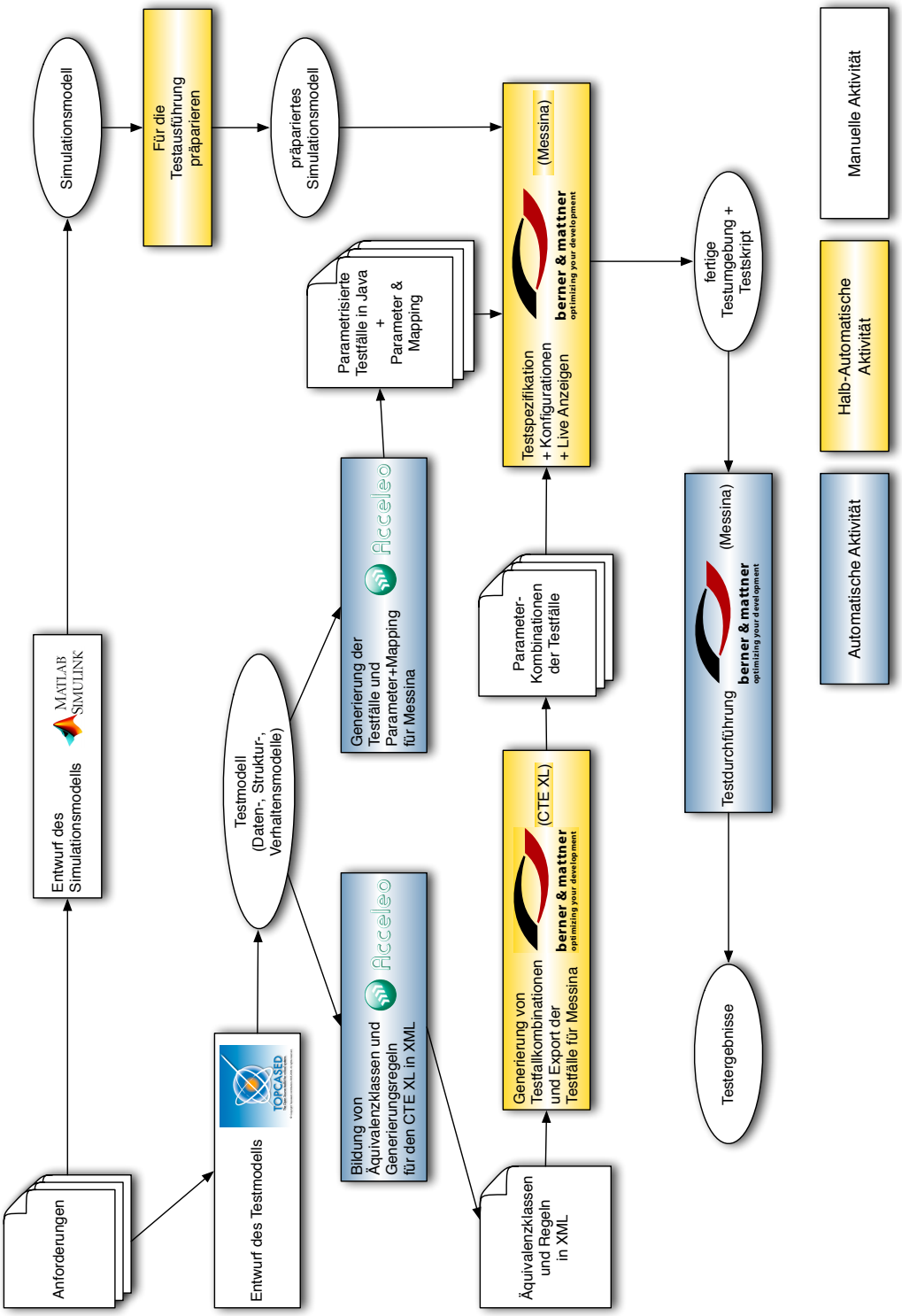


Abbildung 37: Das Vorgehensmodell mit Klassifikation des Automatisierungsgrades

Für die Modellierung des Testmodells wird Topcased verwendet. Über eine Import-Funktion werden die Anforderungen in SysML modelliert. Die modellierten Anforderungen werden als «requirement»-Blöcke definiert. Diese Blöcke werden im Testmodell genutzt, um Beziehungen zwischen den Anforderungen und einigen Modell-Elementen abzubilden (siehe Kapitel 4.1.3). Anhand der Informationen aus den Anforderungen werden die für die Testfallermittlung relevanten Daten und Strukturen modelliert. Zu diesen Informationen zählen Datenmodelle, Strukturmodelle und Verhaltensmodelle. Welche Informationen und auf welcher Art und Weise die Informationen aus den einzelnen Modellen bezogen werden können, wurde in Kapitel 4 gezeigt.

Aus dem Testmodell werden zum einen Äquivalenzklassen und Generierungsregeln für den CTE XL und zum anderen parametrisierte Testfälle und Konfigurationen für Messina automatisch generiert. Für die Generierung der Daten wird das Model-zu-Text-Werkzeug Acceleo benutzt. Dabei dient Acceleo nur für das Aufrufen von bestimmten Java-Funktionen und der Generierung von Dateien mit den Rückgabewerten der aufgerufenen Funktionen als Inhalt. Die Java-Funktionen traversieren das Modell mittels der in Kapitel 4.2 beschriebenen Algorithmen und erstellen eigene Modelle aus denen der Inhalt für die Dateien entstehen. Die Informationen aus den internen Modellen werden in Templates (Text-Schablonen mit zu füllenden Lücken) eingefügt. Die Templates und die Funktionen zum Befüllen der Templates bilden die Abbildung der Informationen aus dem Testmodell auf die Testfälle ab. Mehr zu den Abbildungen zwischen Testmodell und Testfall in Kapitel 5.3.

Die Äquivalenzklassen und die Generierungsregeln werden in eine für CTE XL auslesbare XML-Datei geschrieben. Im CTE XL werden die Generierungsregeln auf den Äquivalenzklassen angewendet. Die daraus entstehenden Testfallparameterkombinationen werden zudem mit Abhängigkeitsregeln auf die relevanten Kombinationen hin gefiltert. Die gefilterten Testfallparameterkombinationen werden letztendlich über eine Export-Funktion an Messina exportiert und einem parametrisierten Testfall zugeordnet. Jede einzelne Kombination ergibt in Zusammenhang mit einem parametrisierten Testfall einen Testfall, der in Messina ausgeführt werden kann. Genauere Informationen über den Vorgang im CTE XL sind im Kapitel 2.4.3 beschrieben.

Aus den Struktur- und Verhaltensmodellen werden die parametrisierten Testfälle generiert. Die Testfälle sind in Java beschrieben und benutzen Bibliotheken aus der Testumgebung Messina. Weiterhin werden Konfigurationen in XML-Dateien für den Signalpool in Messina erzeugt. Der Signalpool in Messina ist eine Abbildung der Schnittstellen und anderen Eigenschaften des Simulationsmodells auf interne Signale in Messina. Diese Signale werden in den Testfällen genutzt, um mit dem Simulationsmodell zu kommunizieren.

In Messina können die Testfälle in Test-Campagnen gruppiert werden. Die Reihenfolge der Testfälle in einer Campagne und die der Campagnen untereinander ergibt den Ablauf der Tests. Sobald diese Reihenfolge steht, kann die Testdurchführung gestartet werden. Vordefinierte Live-Anzeigen und interaktive Anzeigen der einzelnen Testergebnisse in Messina werten die Testdurchführung auf. Am Ende eines Testdurchlaufes hat man eine Liste an Testergebnissen der einzelnen Testfälle.

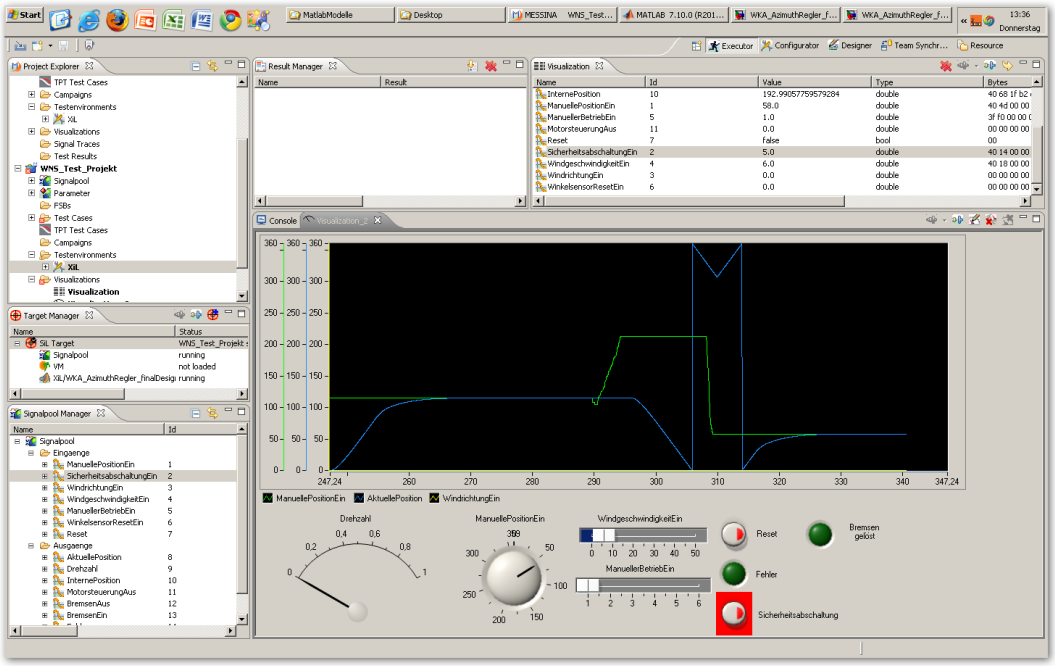


Abbildung 38: Live-Anzeige in Messina

5.3. Modell zu Text Abbildung

Das Modellierungswerkzeug Topcased verwendet das EMF (Eclipse Modeling Framework) als Grundlage für den Aufbau eines Modells. Das EMF hat Funktionen zum Manipulieren von Modellen in XML-Schemata. Ein Metamodell in XML wird von dem EMF in Java-Klassen überführt, die bei der Modellierung in Objekte instanziiert werden. So ein Modell kann mittels Java ausgelesen oder manipuliert werden. Topcased bringt sein eigenes Metamodell der SysML in XML mit, welches das vom EMF schon mitgelieferte UML-Metamodell erweitert.

Für die M2T-Transformation ("Model to Text"-Transformation) kann das Werkzeug Acceleo verwendet werden. Es besitzt eine Implementierung der "MOF Model To Text Transformation Language" der OMG [25]. Anhand dieser Sprache können MOF (Meta Object Facility) konforme Modelle, wie die EMF-Modelle, in Text transformiert werden. Da diese Sprache aber nur bedingt eine Traversierung zwischen den Modellelementen erlaubt, wird sie dafür verwendet, um bestimmte Modellelemente im Modell zu lokalisieren und an Java-Funktionen zu übergeben. Die Java-Funktionen verwenden Templates, die sie mit Modellinformationen und Programmcode befüllen. Die befüllten Templates werden zuletzt als Rückgabewert an Acceleo zurückgegeben und in Dateien geschrieben. Diese Transformations-Regeln werden in Acceleo in ein Eclipse-Plugin gewandelt, welches als Codegenerator verwendet werden kann.

Während der Traversierung über das Modell, analysiert der Codegenrator das Modell. Ähnlich wie bei einem Model-Checker, wird das Modell auf Konstistenz und Vollständigkeit überprüft. Diese Überprüfungen finden statt, um eine eindeutige Interpretation der Semantik des Modells und eine vollständige Codegenerierung zu gewährleisten. Ist das Modell nicht konsistent oder die Informationen nicht vollständig, bricht der Codegenerator ab und gibt eine Fehlermeldung gemäß der unvollständigen Informationen aus. Die Ausgabe der Fehlermeldung erfolgt am Anfang der Ausgabe-Datei.

5.3.1. Datenmodell für CTE XL

Die Daten aus dem MBT-Modell über die Signale für den Test werden in einem Java-Datenmodell überführt. Die Überführung der Daten in ein spezielles Java-Datenmodell hat den Vorteil, dass die Daten im SysML-Modell nicht ständig mit großen Aufwand ausgelesen werden müssen, um an anderer Stelle im Modell eine Überprüfung auf Konsistenz vorzunehmen. Weiterhin vereinfacht das Datenmodell in Java die Generierung in Code, weil es vereinfachte und effiziente Funktionen für das Setzen von Zusammenhängen der Signale und das Auslesen der Daten beherbergt.

Datenmodell

Das Java-Datenmodell wird aus den Daten (siehe Kapitel 4.1.1) des MBT-Modells erstellt. Es beinhaltet die Signale aus dem Block-Definitions-Diagramm, welches den Daten-Pool beschreibt. Die Abbildung 39 zeigt ein UML-Klassendiagramm über das Java-Datenmodell.

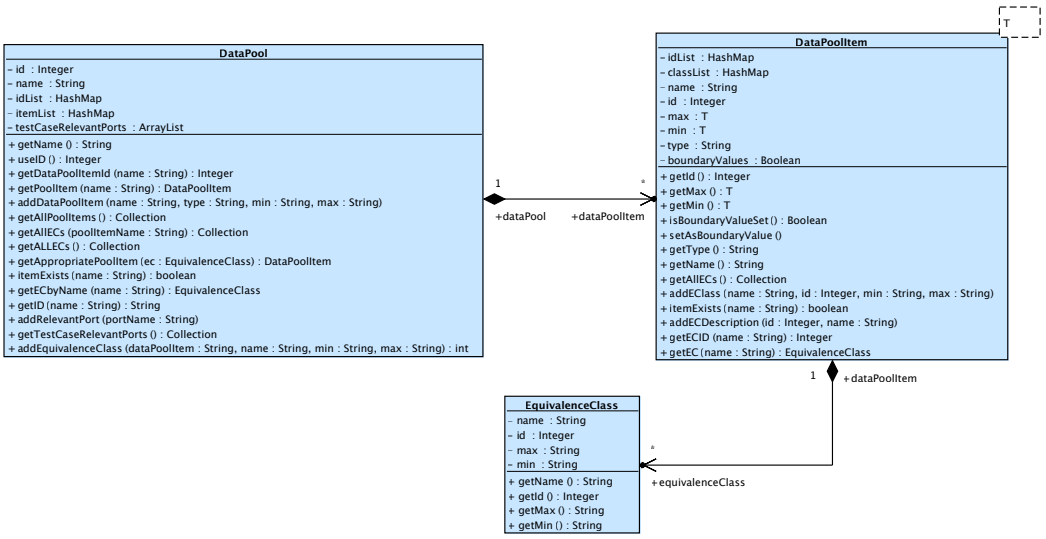


Abbildung 39: UML Klassendiagramm des Java-Datenmodells

Das Java-Datenmodell vergibt allen DataPoolItem-Objekten und EquivalenceClass-Objekten eine eindeutige ID und eine zweite ID als Platzhalter für ein Beschreibungselement. Die Vergabe von ID's ist in vielerlei Hinsicht wichtig. Zum Beispiel benötigt der CTE XL für jedes Datenelement eine eindeutige ID, die nicht noch einmal vorkommen darf. Anhand der ID's werden auch die Generierungsregeln für das CTE XL aufgestellt.

Der Algorithmus zum Auslesen des Datenmodells und zur Erstellung des Java-Datenmodells ist einfach aufgebaut:

1. Finde alle Signale mit dem Stereotypen «DataPoolItem» aus dem BDD, welches sich hierarchisch unter dem Block mit dem Stereotypen «DataPool» befindet.
2. Für jedes dieser Signale, erstelle ein DataPoolItem-Objekt, fülle es mit den Informationen aus den Stereotypen-Attributen und suche alle Signale, die eine Kompositions-Beziehung mit dem aktuellen Signal und den Stereotypen «EquivalenceClass» haben.
3. Wiederum für jedes dieser Signale, erstelle ein EquivalenceClass-Objekt, lese alle Datenfelder aus den Stereotypen-Attributen und erstelle eine Relation zu dem DataPoolItem-Objekt.

Nachdem alle Signale übernommen wurden, stehen im Java-Datenmodell alle Definitionen von Eingangs- und Ausgangswerten. Zusätzlich können auch Parameter-Variablen aus dem Datenmodell bezogen werden.

Die Daten in dem Java-Datenmodell müssen nun an das CTE XL übergeben werden. Das CTE XL liest und speichert Daten in einer XML-Datei. Diese Datei beinhaltet Informationen über die Datenstruktur, die graphische Anordnung im CTE XL, die Generierungsregeln für die Kombinationen und die Filterungsregeln auf den Kombinationen. Für den Export wird eine Template-Datei (siehe Abb. 2) mit den Informationen befüllt. Im Anhang 9 ist eine generierte XML-Datei zu sehen. Der Codegenerator in dieser Arbeit schreibt die Datenstruktur und die Generierungsregeln in die CTE XL Datei.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE cteobject SYSTEM "ctexl-1.0.dtd">
3  <cteobject nextpid="8490" pid="p0">
4  <tree name="Azimutregler" pid="p2" type="root">
5    <nodelayout xpos="489" pagepid="p1" ypos="16">
6      <layoutstyle fontstyle="plain" fgcolor="#000000" fontsize="10"
           fontfamily="Arial" bgcolor="#ffffff"/>
7    </nodelayout>
8    <activetag tagtype="Description" pid="p3"/>
9    <activetag tagtype="Autolayout" pid="p21">
10     <AUTOLAYOUT alignment="0.0" distParent="35" distY="20" distX="20"
           type="4" fixed="false"/>
11   </activetag>

13 ${classes}

15 </tree>
16 <page pid="p1"/>
17 ...
```

```
18  ...  Konfigration (statischer Text)
19  ...
20  <activetag tagtype="TestCaseGeneratorTag" pid="p121">

22  ${generationrules}

24  </activetag>
25  </cteobject>
```

Listing 2: Ausschnitt der CTE XL Template-Datei

Das CTE XL unterscheidet zwischen vier Elementen. Das "root"-Element ist die Wurzel des Baumes. Die "classification"- und "composition"-Elemente sind die Zweigelemente und die "class"-Elemente sind die Blätter des Baumes. Die DataPoolItem-Objekte werden als Zweigelemente mit dem Typen "classification" in das Template geschrieben. Innerhalb dieses Elementes werden die "class"-Elemente als Blätter dieses Zweiges definiert. Die Blätter repräsentieren schließlich die Äquivalenzklassen der DataPoolItems und werden mit den Daten aus den zugehörigen EquivalenceClass-Objekten gefüllt. Die Abbildung 40 zeigt eine Darstellung des Zweiges im CTE XL und in 3 die XML Zeilen für die Definition dieses Zweiges.

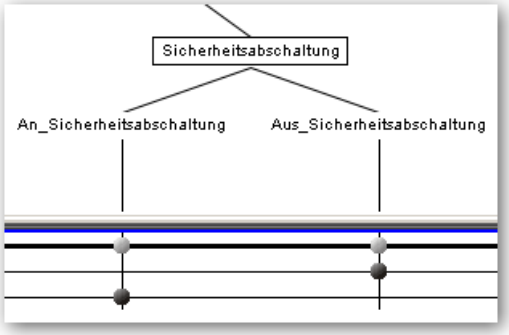


Abbildung 40: Beispiel einer CTE XL Baumstruktur

```
1  <tree name="Sicherheitsabschaltung" pid="p1045" type="classification">
2    <nodelayout xpos="0" pagepid="p1" ypos="0">
3    <layoutstyle fontstyle="plain" fgcolor="#000000" fontsize="10"
         fontfamily="Arial" bgcolor="#ffffff"/>
4    </nodelayout>
5    <activetag tagtype="Description" pid="p1046"/>
6    <tree name="An_Sicherheitsabschaltung" pid="p1047" type="class">
7      <nodelayout xpos="0" pagepid="p1" ypos="0">
8        <layoutstyle fontstyle="plain" fgcolor="#000000" fontsize="10"
             fontfamily="Arial" bgcolor="#ffffff"/>
9      </nodelayout>
10     <activetag tagtype="Description" pid="p1048"/>
11   </tree>
12   <tree name="Aus_Sicherheitsabschaltung" pid="p1049" type="class">
13     <nodelayout xpos="0" pagepid="p1" ypos="0">
```

```
14          <layoutstyle fontstyle="plain" fgcolor="#000000" fontsize="10"
                fontfamily="Arial" bgcolor="#ffffff"/>
15        </nodelayout>
16        <activetag tagtype="Description" pid="p1050"/>
17      </tree>
18    </tree>
```

Listing 3: XML-Aufbau eines Zweiges

Möchte man für die Eingangswerte Grenzwerte verwenden, so setzt man das Stereotypen-Attribut "boundaryValues" der DataPoolItem Elemente auf "true". Mit dieser Fallunterscheidung erzeugt der Codegenerator eine andere Struktur im Klassifikationsbaum des CTE XL. Gemäß einer Grenzwertanalyse, werden innerhalb der Äquivalenzklassen das Minimum, das Maximum und der Median des Wertebereichs verwendet. Somit wird die Struktur um eine Hierarchieebene erweitert. Das DataPoolItem ist nun ein "composition"-Element, die Äquivalenzklasse ein "classification"-Element und die Grenzwerte der Äquivalenzklasse (immer drei) die "class"-Elemente und somit die Blätter. Die Struktur sieht nun wie in Abbildung 41 aus.

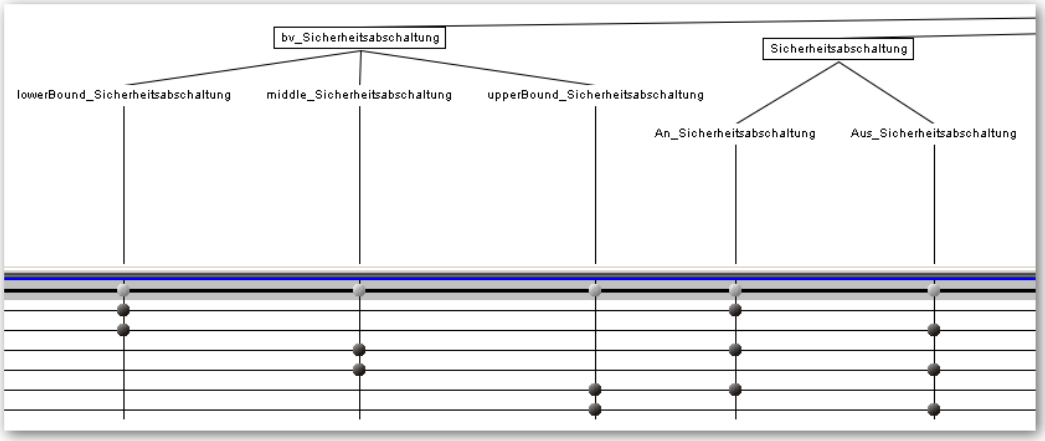


Abbildung 41: Beispiel einer CTE XL Baumstruktur mit Grenzwerten

In einigen Fällen ist es wichtig, die Eingangswerte auf konkrete Werte setzen zu können. Daher kann man im SysML-Modell die beiden Stereotypen-Attribute für den Wertebereich des Stereotypen «EquivalenceClass» auf denselben Wert setzen. Der Codegenerator erkennt das und deaktiviert die Möglichkeit zur Generierung der Grenzwerte.

Die Generierung von Eingangswerte-Kombinationen ist im CTE XL von den Generierungsregeln abhängig. Die Generierungsregeln geben durch logische Ausdrücke Kombinationsregeln zwischen den Zweigelementen oder Blättern an. Die Regeln lassen sich durch "UND" und "ODER" Operatoren und einer Klammerung formulieren. Ein logischer Ausdruck mit

zwei Klassifikationen und einem "UND"-Operator dazwischen würde beispielsweise bedeuten, dass jede Klasse der einen Klassifikation mit jeder Klasse der anderen Klassifikation eine Kombination der daraus entstehenden Menge an Eingabewerte-Kombinationen bildet.

Um zu definieren, welche Eingangswerte für welche Testfälle relevant sind, wurde ein Aktivitätsdiagramm mit eigener formaler Sprache für die Bedingungen verwendet (siehe Abb. 42). Das Aktivitätsdiagramm beschreibt mittels DecisionNode Elementen und dem auf den Pfad liegenden Wächtern die Vorbedingungen eines Testfalls. Die Bedingungen der Wächter werden mit der eigenen formalen Sprache "MBT" definiert. Die Beschreibungssprache ist sehr einfach gehalten.

Die Grammatik der Sprache ist wie folgt definiert: "DataPoolItem:Name" = "EquivalenceClass:Name" Aus den Kapitel 4.1.1 ging hervor, dass man über den Namen eines DataPoolItem Elements den Port der Schnittstelle zu dem SuT-Modell adressieren kann, weil jeder Port seinen eigenes DataPoolItem als Datentypen besitzt. Die Semantik dieses formalen Ausdrucks sagt aus, dass der vom DataPoolItem adressierte Port einen Wert aus dem Wertebereich der Äquivalenzklasse anliegen hat. Diese Bedingung kann wahr oder falsch sein.

Prinzipiell kann man die Adressierung zu einem Port auch über ein EquivalenceClass Element herstellen, weil es zu einem DataPoolItem Element gehört. Für diesen Zweck wurde eine zweite Formale Beschreibungssprache "MBT_Signals" entwickelt, die in den Verhaltensdiagrammen als Kurzschreibweise Verwendung fand. Somit sieht die Grammatik stark verkürzt aus: "EquivalenceClass:Name"

Adressiert eine Äquivalenzklasse in einem Bedingungs-Ausdruck der Sprache "MBT_Signals" einen Input-Port, so wird dieser Port gelesen und mit dem Wertebereich der Äquivalenzklasse verglichen. Falls es ein Output-Port ist, wird der Wert von diesem Port ebenso gelesen und verglichen.

Das Modell hinter dem Diagramm in 42 beinhaltet noch weitere Informationen, die nicht im Diagramm zu sehen sind. Jedes CallBehaviorAction Element, welches auf ein TestCase Element verweist, hat eine Vorbedingung in Form eines Constaints. Über die Eigenschaft "localPrecondition" wird ein Verweis auf das Constraint Element gemacht. Das Constraint Element hat keine Spezifikation, sondern referenziert auf alle für den Test wichtigen Signale als Liste in der Eigenschaft "constrainedElement".

Für den Testfall "NormalerBetrieb" erhält man beispielsweise aus dem Aktivitätsdiagramm in Abbildung 42 folgende Generierungsregel:

```
1  Sicherheitsabschaltung/Aus_Sicherheitsabschaltung *
       Windgeschwindigkeit/Betrieb_Windgeschwindigkeit * ManuellerBetrieb/
       Aus_ManuellerBetrieb * Windrichtung * AktuellePosition *
       AnzahlRegelungen * Fehlergenerierung;
```

Listing 4: Generierungsregel

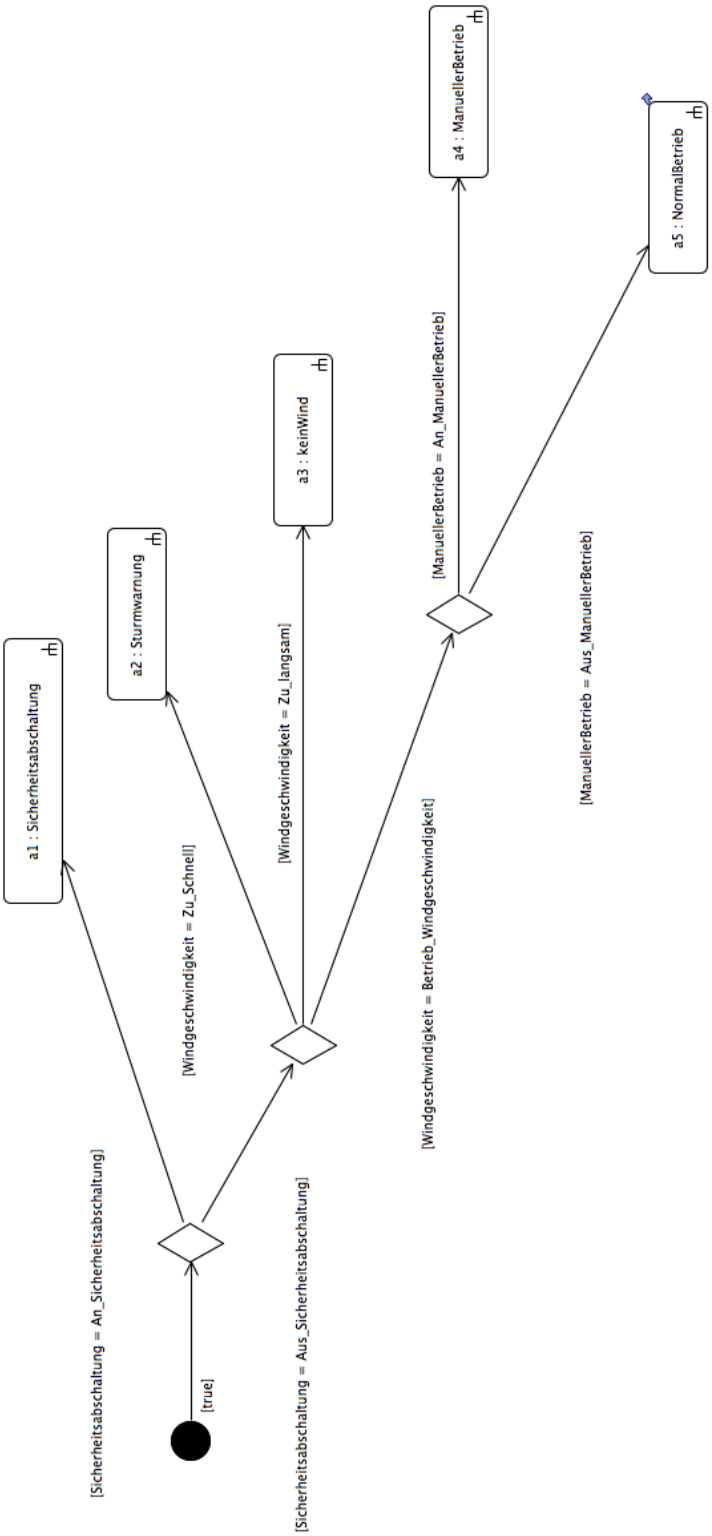


Abbildung 42: Aktiviätsdiagramm mit den Vorbedingungen für die einzelnen Testfälle

In den Generierungsregeln der XML-Datei werden die Namen der Elemente mit ihrer ID ersetzt und man erhält folgende Zeile:

```
<generationrule term="1029 * 1026 * 1037 * 1043 * 1008 * 1069 * 1000 *
    1091;" name="NormalBetrieb" pid="p1103"/>
```

Listing 5: Beispiel für eine Generierungsregel für CTE

Die aus den Generierungsregeln entstehenden Kombinationen von Klassen, können im nachhinein als Parameter für die Testfälle übergeben werden. Üblicherweise sind unter der Menge der Kombinationen einige dabei, die in der realen Welt so nicht auftreten würden oder sollen. Um diese Kombinationen aus der Menge zu filtern, gibt es im CTE die Möglichkeit Beziehungsregeln aufzustellen. Die Beziehungsregeln filtern die Mengen von Kombinationen wiederum auf der Grundlage von logischen Ausdrücken. Die Anzahl der Operatoren ist größer als die bei den Generierungsregeln. Alle Versuche, diese Beziehungsregeln in Modelle zu kodieren scheiterten bisher, sodass man die Regeln im CTE immer noch selber aufstellen muss, falls man Filter auf den Kombinationen legen möchte.

5.3.2. Konfiguration der Testumgebung

Die Strukturdiagramme im Test-Modell beinhalten Informationen, die für die Konfiguration der Testumgebung verwendet werden können. Man kann Informationen über die Signal-Zuordnung, über die Parameter für die Testfälle, die Reihenfolge bzw. Steuerung der Testfälle in Testszenarien aus ihnen gewinnen.

Die Spezifikation über die Signale wird in Messina in einer XML-Datei "SignalPool" gehalten. Die dort definierten Signale werden über eine Signal-ID an den Port des importierten Matlab/Simulink Modell zugewiesen. Die Signal-ID wird in einer Konfigurationsdatei für das Modell beim Import festgelegt. Die Konfigurationsdatei ist auch in XML gehalten und kann nach dem Import eingelesen werden. Nach einem Import sind alle Signal-IDs der Ports oder Variablen in der Konfigurationsdatei vorerst auf den Wert -1 gesetzt. Über den Namen kann der jeweilige Ein- bzw. Ausgangs oder die Variable gefunden werden. Für die Zuweisung an den Signalpool in Messina, wird diesem Port oder der Variable eine eindeutige Signal-ID zugewiesen. Das zugehörige Signal in der Signalpool-Datei erhält folglich die gleiche Signal-ID.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<signals>
    <Name>Project_3</Name>
  <Group>
      <Name>Eingang</Name>
     <signal>
          <Name>ManuellePositionEin</Name>
          <Type>double</Type>
          <signalLength>8</signalLength>
          <Default>0</Default>
          <Unit>fr</Unit>
          <signalId>1</signalId>
```

```
13          </signal>
14          <signal>
15              <Name>SicherheitsabschaltungEin</Name>
16              <Type>double</Type>
17              <signalLength>8</signalLength>
18              <Default>0</Default>
19              <Unit/>
20              <signalId>2</signalId>
21          </signal>
22      </Group>
23      <Group>
24          <Name>Ausgang</Name>
25          <signal>
26              <Name>AktuellePosition</Name>
27              <Type>double</Type>
28              <signalLength>8</signalLength>
29              <Default>0</Default>
30              <Unit/>
31              <signalId>7</signalId>
32          </signal>
33          <signal>
34              <Name>Drehzahl</Name>
35              <Type>double</Type>
36              <signalLength>8</signalLength>
37              <Default>0</Default>
38              <Unit/>
39              <signalId>8</signalId>
40          </signal>
41      </Group>
42  </signals>
```

Listing 6: Beispiel für die Definition des Datenpools

Die Parameter für die Testfälle sind wiederum in einer XML-Datei festgelegt. Diese können eins zu eins aus dem Datenmodell des Test-Modells übernommen werden. Jedes DataPoolItem wird in dieser Datei mit dem Namen und dem Typen String hinterlegt. Die Parameter erhalten den Typen String, weil die Parameter aus dem CTE XL als Strings übergeben werden und die Auflösung in Werte anderer Typen erst innerhalb der Testfallbeschreibung stattfindet.

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <params>
3      <Name>Project_3</Name>
4      <param>
5          <Name>WindgeschwindigkeitEin</Name>
6          <Type>string</Type>
7          <paramLength>0</paramLength>
8          <Default/>
9      </param>
10     <param>
11         <Name>BremsenAus</Name>
12         <Type>string</Type>
13         <paramLength>0</paramLength>
14         <Default/>
15     </param>
16 </params>
```

Listing 7: Beispiel für eine Parameter-Konfiguration

Anknüpfend an das Kapitel 4.1.2 wird hier noch einmal das Thema mit den Testszenarien aus den Zustandsdiagrammen des SuT aufgegriffen. Die Testszenarien werden anhand des Zustandsdiagramms für die Betriebsmodi des WNS definiert. Jeder Zustand bzw. Modus in dem Diagramm (siehe Abb. 21) definiert einen Testfall. Die Ereignisse zum Aktivieren dieser Zustände sind die Vorbedingungen der Testfälle. Daher kann über die in Kapitel 4.2 beschriebenen Algorithmen ein Pfad bezüglich eines Abdeckungskriteriums aus dem Zustandsdiagramm bezogen werden. Dieser Pfad gibt eine Reihenfolge für die Testfälle an. Diese Reihenfolge kann in einem Testszenario gebündelt werden. Das Auftreten der Ereignisse, für den Zustandswechsel, wird in den Nachbedingungen der Testfälle als Parameter definiert.

In Messina können Testszenarien definiert werden. Die Testszenarien werden in XML-Dateien beschrieben. In den XML-Dateien stehen die Reihenfolgen der Testfälle. Zusätzlich können dort die Parameter Variablen für die einzelnen Testfälle angegeben werden. Macht man einen Export von CTE XL nach Messina, wird bespielsweise genau so eine XML-Datei erstellt, die aber nicht für diesen Kontext verwendet werden kann. Ein aus Messina exportiertes Testszenario stellt einen datengetriebenen Test auf einem Testfall dar.

Die XML-Datei für die Definition der Testszenarien ist wie in Beispiel 8 aufgebaut. Zwischen den Zeilen 4 bis 20 wird der Aufruf eines Testfalls “NormalBetrieb.java” mit der Parameterübergabe definiert.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <Campaign><Name>NormalBetrieb</Name>
3      <elements>
4      <test>
5        <Name>NormalBetrieb.java</Name>
6        <package/>
7        <testFolder>TestCases</testFolder>
8        <paramValues>
9          <param>
10           <Name>ManuellerBetriebEin</Name>
11           <Type>string</Type>
12           <Value>Aus</Value>
13         </param>
14         <param>
15           <Name>BremsenAus</Name>
16           <Type>string</Type>
17           <Value>Anziehen</Value>
18         </param>
19       </paramValues>
20     </test>
21     </elements>
22 </Campaign>
```

Listing 8: Beispiel eines Testszenarios mit einem Testfall

Lautet das Testendekriterium, dass jede Transition mindestens einmal ausgelöst wurde, so würde der Algorithmus in Kapitel 4.2.1 für das Diagramm in Abbildung 21 folgende Testfall Reihenfolge ausgeben:

Für das Verständnis der Notation:

$$\text{Testfall: “Zustand:Name”} \xrightarrow{\text{“}Trigger:SignalEvent:Signal:Name\text{”}} \rightarrow$$

Die Notation beschreibt die Ausführung eines Testfalls, der den Zustand mit den Namen “Zustand:Name” testet. Nach der Ausführung wird das Signal mit den Namen “Trigger:SignalEvent:Signal:Name” ausgeführt, um in den nächsten Zustand zu wechseln. Das Senden des Signals wird in der Nachbedingung des Testfalls durchgeführt. Das Senden des Signals ist eine Vorbedingung des nächsten Testfalls.

Testfall: Normalbetrieb $\xrightarrow{An\_Sicherheitsabschaltung}$

Testfall: Sicherheitsabschaltung $\xrightarrow{Aus\_Sicherheitsabschaltung}$

Testfall: Normalbetrieb $\xrightarrow{An\_ManuellerBetrieb}$

Testfall: ManuellerBetrieb $\xrightarrow{Zu\_Schnell}$

Testfall: ManuellerBetrieb $\xrightarrow{An\_Sicherheitsabschaltung}$

Testfall: Sicherheitsabschaltung $\xrightarrow{Aus\_Sicherheitsabschaltung}$

Testfall: Sturmwarnung $\xrightarrow{An\_ManuellerBetrieb}$

Testfall: ManuellerBetrieb $\xrightarrow{Betrieb\_Windgeschwindigkeit}$

Testfall: ManuellerBetrieb $\xrightarrow{An\_Sicherheitsabschaltung}$

Testfall: Sicherheitsabschaltung $\xrightarrow{Zu\_Schnell}$

Testfall: Sicherheitsabschaltung $\xrightarrow{Betrieb\_Windgeschwindigkeit}$

Testfall: Sicherheitsabschaltung $\xrightarrow{Zu\_Langsam}$

Testfall: Sicherheitsabschaltung $\xrightarrow{Aus\_Sicherheitsabschaltung}$

Testfall: WindgeschwindigkeitZuLangsam $\xrightarrow{An\_ManuellerBetrieb}$

Testfall: ManuellerBetrieb $\xrightarrow{An\_Sicherheitsabschaltung}$

Testfall: Sicherheitsabschaltung $\xrightarrow{Aus\_Sicherheitsabschaltung}$

Testfall: WindgeschwindigkeitZuLangsam $\xrightarrow{An\_Sicherheitsabschaltung}$

Testfall: Sicherheitsabschaltung $\xrightarrow{Betrieb\_Windgeschwindigkeit}$

Testfall: Sicherheitsabschaltung $\xrightarrow{Aus\_Sicherheitsabschaltung}$

Testfall: Normalbetrieb $\xrightarrow{Zu\_Langsam}$

Testfall: WindgeschwindigkeitZuLangsam $\xrightarrow{An\_ManuellerBetrieb}$

Testfall: ManuellerBetrieb $\xrightarrow{Betrieb\_Windgeschwindigkeit}$

Testfall: ManuellerBetrieb $\xrightarrow{Zu\_Langsam}$

Testfall: ManuellerBetrieb $\xrightarrow{Aus\_ManuellerBetrieb}$

Testfall: WindgeschwindigkeitZuLangsam $\xrightarrow{Betrieb\_Windgeschwindigkeit}$

Testfall: Normalbetrieb $\xrightarrow{An\_ManuellerBetrieb}$

Testfall: ManuellerBetrieb $\xrightarrow{Aus\_ManuellerBetrieb}$

Testfall: Normalbetrieb $\xrightarrow{Zu\_Schnell}$

Testfall: Sturmwarnung $\xrightarrow{An\_ManuellerBetrieb}$

Testfall: ManuellerBetrieb $\xrightarrow{Aus\_ManuellerBetrieb}$

Testfall: Sturmwarnung $\xrightarrow{An\_Sicherheitsabschaltung}$

Testfall: Sicherheitsabschaltung $\xrightarrow{Aus\_Sicherheitsabschaltung}$

Testfall: Sturmwarnung $\xrightarrow{Betrieb\_Windgeschwindigkeit}$

5.3.3. Testfallgenerierung

Das Erschließen von Testfällen aus Verhaltensmodellen wird in Kapitel 4.1.2 diskutiert. In dem Kapitel geht hervor, dass die Modelle bestimmte Informationen als Vorraussetzung enthalten müssen, um Testfälle aus ihnen gewinnen zu können. Die notwendigen Informationen für einen Testfall sind:

- die Datentypen der Eingaben
- die Eingabewerte (u.U. auch die logische Aussage der Werte)
- welche Eingänge des SuT mit welchen Eingaben stimuliert werden
- sinnvolle Reihenfolgen der Eingaben
- die Datentypen der Ausgaben des SuT
- das reaktive Soll-Verhalten des SuT in Ausgabewerten oder als Soll-Funktion kodiert
- die Randbedingungen für den Testfall, wie Vor- und Nachbedingungen des SuT oder der Umgebungskomponenten

Die Informationen über die Datentypen von Ein- und Ausgaben, sowie die Werte mit den logischen Aussagen und die zugehörigen Ein- bzw. Ausgänge des SuT, sind im Java-Datenmodell aus Kapitel 5.3.1 bereits enthalten. Die sinnvollen Reihenfolgen von Eingabewerten und das reaktive Verhalten das SuT in Form von Ausgaben, wird in Verhaltensdiagrammen abgebildet. In dieser Arbeit wird grundsätzlich zwischen drei Modelliermethoden mit Verhaltensdiagrammen für das MBT unterschieden.

1. Verhaltensmodelle, die einen Testfall explizit beschreiben
2. Verhaltensmodelle, die ein Verhalten des SuT beschreiben
3. mehrere Verhaltensmodelle, die separat Eingaben und Sollverhalten beschreiben

Verhaltensmodelle, die Testfälle explizit beschreiben werden in dieser Arbeit nicht behandelt. Beispiele für solche Verhaltensmodelle sind in [4] zu finden. Häufig werden Sequenzdiagramme für diese Verhaltensmodelle verwendet.

In einem Verhaltensmodell, in dem ein Verhalten des SuT beschrieben ist, sind die Informationen, wie die möglichen Eingaben, implizit enthalten. Für diese Art der Modellierungsmethode wird in dieser Arbeit das Zustandsdiagramm verwendet. Die Zustände geben Aufschluss über die aktuellen Ausgangswerte des SuT und die Transitionen, die aus dem aktuellen Zustand führen, über weitere mögliche Eingaben. Dabei können die Ausgabewerte als ein Datum, ein Intervall von Daten oder als eine von Eingangswerten abhängige Funktion beschrieben werden. Diese Ausgangswerte werden im Aktivitäts-Verhalten des Zustandes kodiert.

Mithilfe der Algorithmen aus Kapitel 4.2 werden sinnvolle Reihenfolgen, für das Auslösen der Transitionen, erschlossen. Für das Auslösen der einzelnen Transitionen, müssen die Trigger und Wächterbedingungen auf den Transitionen analysiert werden. Wenn alle Transitionen nur einen Trigger und keine Wächterbedingung haben, so werden die Eingaben aus dem Event, auf dem getriggert wird, interpretiert. Falls eine Transition mehr als einen Trigger besitzt, muss zwingenderweise ein getriggertes Event ausgewählt werden. Um diesem Problem aus dem Weg zu gehen, sollte für jeden Trigger eine eigene Transition erstellt werden, damit der CPP Algorithmus in 4.2 beispielsweise, jeden dieser getriggerten Events in dem Testfall berücksichtigt.

Das Auslösen einer Transition mit einem Trigger und einem Wächter, gestaltet sich schwieriger. Auch in diesem Fall muss aus dem getriggerten Event die Eingabe interpretiert werden. Zudem muss aber auch die Wächterbedingung wahr sein, um die Transition auszulösen. Der Codegenerator für die Generierung der Testfälle muss die Wächterbedingung korrekt interpretieren und gegebenenfalls Maßnahmen für die Erfüllung der Bedingung berechnen können. Die Maßnahmen müssen in der Reihenfolge vor der Eingabe für das Event durchgeführt werden.

In einem Test in dem Stubs verwendet werden die autonom vom Testtreiber laufen, muss darauf geachtet werden, dass das Stub nicht Eingaben tätigen kann, die die im Zustandsautomaten beschriebenen Events auslösen oder die Wächterbedingungen ändern können. Andernfalls sind die aus dem Zustandsautomaten generierten Testfälle gefährdet, unkontrolliert abzulaufen, weil die Eingaben des Stubs dazwischen auftreten könnten.

Die Tests in dieser Arbeit sind MiL-Simulationen. In diesen Simulationen wird der Motor und die Bremsen als autonomer Stub realisiert. Aus diesem Grund können keine, wie oben beschrieben, kontrollierten Tests aus Zustandsautomaten abgeleitet werden. Dennoch können die Zustandsautomaten für die Generierung von Testszenarien wie in Kapitel 5.3.1 oder zur Gewinnung von Soll-Funktion für die Berechnung von Sollwerten verwendet werden.

Die aus den Zustandsautomaten generierten Soll-Funktion sind ausführbare Java-Modelle. Für die Berechnung der Sollwerte werden die Zustandsautomaten in Java abgebildet. In den

jeweiligen Zuständen sind Abfragen auf das in den Rümpfen des Zustandes beschriebenem Ausführungs-Verhalten. In einem weiteren Diagramm einer Aktivität sind die Generierungsregeln für die Eingaben definiert. Das Aktivitätsdiagramm in Abbildung 20 zeigt einen Testtreiber für die Generierung von Stimuli an das SuT. Die Eingaben und die erwarteten Ausgaben sind demnach in zwei Verhaltensdiagrammen aufgeteilt.

Zusätzlich werden Vorbedingungen für einen Testfall benötigt. Diese werden in Aktivitätsdiagrammen modelliert. Die Abbildung 42 zeigt ein Aktivitätsdiagramm, dass für die Testfälle die Vorbedingungen enthält. Die Vorbedingungen sind entlang des Pfades vom Startknoten zur jeweiligen Aktivität, die einen Testfall startet, kodiert. Weitere Vorbedingungen sind mit Constraint Elementen auf den Aktivitäten definiert. Diese können parametrisiert werden, um dynamische Tests in verschiedenen Situationen zu erhalten.

Die Parametrisierung der Testfälle kann verschieden angewendet werden. In der Implementierung des Codegenerators sind zwei Anwendungen eingeflossen:

1. Parametrisierte Testfälle zum datangetriebenen Testen und Steuerung des Tests.
2. Parametrisierte Testfälle für Testszenarien.

In der ersten Anwendung werden die Eingaben dynamisch über die Parameter bestimmt. Ferner steuern die Parameter den Ablauf des Testfalls. In dem Aktivitätsdiagramm in Abbildung 20 sind mehrere Testfälle in Form von verschiedenen Mengen von Eingaben enthalten. Mithilfe der DecisionNode Elementen sind mehrere Flüsse in der Aktivität möglich. Die Parameter bestimmen sozusagen den Pfad in dem Diagramm.

Die zweite Anwendung ist für die Bildung von Testszenarien aus mehreren Testfällen wichtig. Die Nachbedingungen der Testfälle sind insoweit parametrisiert, dass über die Parameter bestimmte Aktionen durchgeführt werden, um die Vorbedingung des nächsten Testfalls zu entsprechen. Bisher beschränken sich die Aktionen auf Eingaben an das SuT, die das System in einen anderen Betriebsmodus versetzen soll, und das Manipulieren der Stub Komponenten über deren Operationen für die Ports. Man könnte diese Aktionen aber insofern erweitern, dass Prüfungen auf Bedingungen und komplexere Aktionen möglich wären.

Für die Abfragen der Ausgänge und Bedingungen und das Senden der Eingaben wird im Programmcode der Testfälle eine Funktion periodisch aufgerufen. Die Funktion ist eine überladene Funktion der Messina Programm-Bibliothek. Gestartet wird sie mit dem Aufruf startTimer(int Millisekunden). Daraufhin wird die überladene Funktion timer(int tick, long late) periodisch in den angegebenen Millisekunden aufgerufen. Solange die "timer"-Funktion kein "false" als Rückgabewert liefert, wird sie immer wieder aufgerufen. Die Programm-Bibliothek von Messina stellt sicher, dass der Programmcode im Rumpf der "timer"-Funktion nicht die Dauer der in der Startfunktion angegebenen Millisekunden überschreitet. Trifft der Fall ein, dass der Programmcode die Dauer vor dem nächsten Aufruf überschreitet, wird dementsprechend von der Messina Programm-Bibliothek eine Fehlermeldung ausgegeben.

Listing der Programm-Bibliothek von Messina zu der timer-Funktion

Die Programm-Bibliothek von Messina stellt für die Definition der Testfälle drei Funktionen bereit, preCondition(), postCondition() und run(). Diese werden in der logischen Reihenfolge preCondition() → run() → postCondition() ausgeführt. Weiterhin gibt es eine globale Variable namens "resultString", welche den Text für einen nicht erfolgreichen Test enthält und in der Test-Ergebnis-Liste von Messina stehen wird.

Um zeitintensive Lese-Operationen auf den Signalen im Datenpool von Messina zu vermeiden, gibt es die Möglichkeit über eine Funktion register() vom Datenpool nur die Wertänderungen zu erhalten. Der Datenpool schreibt bei Wertänderungen des Signals den Wert in eine lokale Variable, die mit der Funktion getLokalValue() ausgelesen werden kann. Diese Funktionalität ist besonders für die Lese-Operationen in der "timer"-Funktion nützlich.

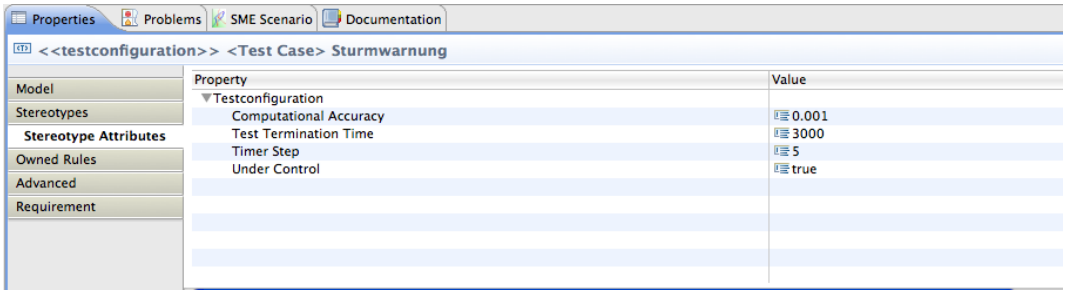


Abbildung 43: Beispiel einer Testkonfiguration auf dem Testfall “Sturmwarnung”

Jeder Testfall im MBT-Modell wird mit dem Stereotypen «Testconfiguration» versehen. In den Attributen des Stereotypen werden einige wichtige Informationen für die Testkonfiguration des Testfalls festgelegt. Diese Konfigurations-Informationen werden in den Testfällen benötigt. Die Abbildung 12 zeigt die Stereotypen-Attribute. Die Attribute haben folgende Bedeutungen:

testTerminationTime: Dieses Attribut gibt einen Zeitwert an, wie lang der Testfall maximal laufen darf. Überschreitet der Testfall diesen Wert, wird er abgebrochen.

computationalAccuracy: Für das Berechnen von und mit Werten wird eine Rechengenauigkeit benötigt. Diese wird beispielsweise für die Berechnung der Grenzwerte aus den Wertebereichen der Signal-Difinitionen benötigt. Dieser Wert muss ein numerischer Wert sein.

timerStep: Der Wert dieses Attributs wird in Millisekunden aufgefasst und wird für die "timer"-Funktion verwendet. Weiterhin wird er für die Zeitberechnungen im Rumpf der "timer"-Funktion verwendet. In Kombination mit dem Parameter tick der "timer"-Funktion lässt sich beispielweise die verstrichene Zeit berechnen, durch das Produkt der beiden Werte (tick*timerStep).

underControl: Dieses Attribut wurde für den Fall aufgenommen, dass die Testumgebung keine autonom laufenden Stubs beinhaltet. Ist dieser Wert auf "true", werden die Testfälle

allein aus dem Zustandsautomaten hergeleitet, ohne eine Aktivität mit der Definition der Stumili zu verwenden.

Wie weiter oben schon erwähnt, werden für die Generierung der Testfälle Templates verwendet. Das Template für die Testfälle sieht wie folgt aus:

```
1  import signalBase.AllSignals;
2  import signalBase.params;
3  import java.io.IOException;

5  import com.berner_mattner.messina.executer.vmaccess.ITargetLogger;
6  import com.berner_mattner.messina.vm.testbase.TestCase.
       TestFailedException;

8  ${error}
9  ${enumDefinition}

11 public class ${TestCaseName} extends AllSignals {

13   private boolean errorOccured;
14   private int stateactiveTime;
15   ${testConfiguration}

17   State state = ${initialState};


20 ${signalValueMapping}


23 ${testComponentFunctions}

25   @Override
26   public boolean preCondition() throws IOException,
         TestFailedException,
27        InterruptedException {


30     ${precondition}

32     // register for lokal values
33     ${signalRegistering}

35     return super.preCondition();
36   }

38   public int run() throws IOException, TestFailedException,
39       InterruptedException {

41     startTimer(timerStep);

43     return errorOccured?1:0; //0 means PASSED
44   }

46   @Override
47   protected boolean timer(int tick, long late) throws IOException,
48       TestFailedException, InterruptedException {

50     ${initiateLokalValues}
```

```
52      // ---------------- Stimuli generated from the Signal Generator
            Activity Diagram -----------

54      ${stimuli}

56      // ---------------- State specific asserts
            --------------------------

58      ${switchCase}

60      // ---------------- Global asserts
            ----------------------------------

62      ${globalConstraints}


65      if (errorOccured){
66        resultString += " - Error occured at Tick="+tick;
67        return false;
68      }

70      ${end}

72      return true;
73    }

75    @Override
76    public boolean postCondition() throws IOException,
          TestFailedException,
77          InterruptedException {

79      ${postConditions}

81      return super.postCondition();
82    }

84  }
```

Listing 9: Template für die Testfälle

Die Platzhalter – zu erkennen an der Notation ${"name"} – werden vom Codegenerator durch Programmcode ersetzt. Die verschiedenen Platzhalter haben folgende Bedeutungen:

error an dieser Stelle werden Fehlermeldungen geschrieben, falls der Codegenerator Inkonsistenzen oder fehlende Informationen im Modell erkennt

enumDefinition an diesen Platzhalter werden die Zustände für die Soll-Funktion als Enumeration definiert.

TestCaseName definiert den Namen der Klasse. Der Name der Klasse ist der Name des Testfalls im Modell.

testConfiguration hier werden die Testkonfigurationen wie die "timerStep"-Variable geschrieben.

initialState der Enumerations-Wert für den Zustand, mit dem die Soll-Funktion beginnt.

signalValueMapping hier werden die Funktionen für das Abbilden der String-Parameter in die zugehörigen Werte definiert. Ein Beispiel für eine Funktion ist in dem Template Beispiel 9 oder im Anhang B gegeben.

testComponentFunktions hier stehen die Funktionen der Stubs, die vom Testfall für die Manipulierung der Stubs verwendet werden können.

precondition an dieser Stelle werden die Vorbedingungen des Testfalls geprüft und gegebenenfalls Funktionen ausgeführt, um die Bedingungen zu erfüllen.

signalRegistering auf den testrelevanten Signalen, die in der "timer"-Funktion gelesen werden, werden an dieser Stelle die "register"-Funktionen aufgerufen.

initiateLokalValues die lokalen Variablen der zuvor registrierten Signale, werden hier ausgelesen und in Variablen geschrieben.

stimuli dieser Teil des Testfalls stellt die Abbildung der Aktivität mit den Stimuli dar. Wie die einzelnen Elemente aus der Aktivität abgebildet werden, wird weiter unten beschrieben.

stateSpecificAsserts hier steht die Soll-Funktion zur Berechnung der Sollwerte. Der für den Testfall zugehörige Zustandsautomat wird an dieser Stelle in Java abgebildet. Wie die Abbildung aussieht und welche Informationen hierfür noch benötigt werden wird im folgenden beschrieben.

globalConstraints die globalen Bedingungen, die in Kapitel 4.1.2 diskutiert wurden, werden hier als Bedingung für den erfolgreichen Testdurchlauf verwendet.

end dieser Platzhalter ist für Testendekriterien gedacht, die nicht in den Soll-Funktionen definiert sind.

postConditions hier können Nachbedingungen eines Testfalls stehen. Speziell bei der Erzeugung von Testszenarien, wird an dieser Stelle eine Funktion mit Parametern generiert, die die Vorbedingung des nächsten Testfalls erfüllt.

Im Anhang B ist ein Beispiel für einen generierten Testfall. Für die Abbildung des Zustandsautomaten in eine Java Soll-Funktion werden noch einige Informationen benötigt. Jeder Zustand im Zustandsautomaten erhält den Stereotypen «TestTiming» wie in Abbildung 33 zu sehen. Das Stereotyp hat die Attribute "expireTime" und "holdTime". Besonders der Wert in "expireTime" ist wichtig, um ein Resultat in dem Testfall bilden zu können. Wenn ein Zustand aktiv wird, wird eine Variable "stateActiveTime" hochgezählt. Die Ausgänge des SuT werden mit den erwarteten Ausgaben verglichen. Da das Zustandsautomaten-Modell in Java in der Regel schneller ist als das SuT, gibt es eine Zeit "expireTime", in der auf das SuT gewartet wird. Nimmt das SuT innerhalb dieser Zeit nicht die erwarteten Werte an den Ausgängen an, so wird ein Fehler vom Testfall attestiert.

Die Zeit in "holdTime" wird vom Codegenerator nur verwendet, wenn die Test-Konfiguration "underControl" auf "true" gesetzt ist. Das bedeutet aber auch wiederum, dass bei der Verwendung der Stubs darauf geachtet werden muss, dass diese nicht Events auslösen können, die zu einem Zustandswechsel führen könnten. Die Zeit "holdTime" gibt an, wie lange ein Zustand aktiv bleibt, nachdem vom Testtreiber der Zustand am SuT erkannt wurde und bevor das nächste Event ausgelöst wird, um in den nächsten Zustand zu wechseln. Somit erhält man eine zeitliche Kontrolle über den Ablauf des Testfalls.

Im Folgenden ist ein Beispiel für die Abbildung eines Zustandes im Testfall:

```
1  case IDLE :
2    if(firstEntry){
3      firstEntry = false;
4      stateActiveTime = 0;
5      stateHoldTime = 0;
6      entry();
7    }
8    if(!(lokalMotorsteuerungsbefehlAus > 0 &&
         lokalMotorsteuerungsbefehlAus < 10
9    && lokalFehlerAus > 0 && lokalFehlerAus < 1 &&
         lokalBremsenAnsteuerungAus >
10   0 && lokalBremsenAnsteuerungAus < 5) && stateActiveTime > 20){
11     errorOccured = true; // hier kann noch eine returnMsg kommen
12   }

14   if (lokalMotorsteuerungsbefehlAus > 0 &&
         lokalMotorsteuerungsbefehlAus < 10
15   && lokalFehlerAus > 0 && lokalFehlerAus < 1 &&
         lokalBremsenAnsteuerungAus >
16   0 && lokalBremsenAnsteuerungAus < 5) {
17     if(stateActiveTime > 3000)
18       return false;
19   }
20   if((lokalWindrichtungEin > 0 && lokalWindrichtungEin < 6) &&
21   abs(lokalWindrichtungEin - lokalAktuellePositionEin) > 0.1){
22     nextState = BREMSENLOESEN;
23     firstEntry = true;
24     exit();
25   }
26   break;
```

Listing 10: Abbildung eines Zustandes in Java

Die Zeilen 2 bis 7 definieren das Eintritts-Verhalten des Zustandes. Die Zeile 6 steht stellvertretend für eine Eintritts-Funktion, die in dem Eintritts-Verhalten des Zustandes definiert ist. Die Zeilen 8 bis 12 vergleichen die Ausgänge des SuT mit den erwarteten Werten. Die Zeilen 14 bis 19 definieren die Zeit für das Beenden des Testfalls. Falls in der Zeit "testTerminationTime" (in dem Beispiel der Wert 3000) keine Signale für einen Zustandswechsel kommen, wird der Testfall beendet. Die Zeilen 20 bis 25 beschreiben den Zustandswechsel und das Austritts-Verhalten des Zustandes. Falls der Zustand mehrere ausgehende Transitionen hat, werden dementsprechend auch mehrere Fälle abgeprüft.

Die Abbildung der Modellelemente aus den Aktivitäten mit dem Stereotypen «SignalGenerator» nach Java Programmcode sind in der Tabelle 3 aufgeführt. Das Aktivitätsdiagramm in Abbildung 20 zeigt so eine Aktivität. Der Algorithmus, der diese Aktivitäten ausliest, sucht nach möglichen Pfaden. Diese werden in einer if-else-Struktur in Java abgebildet. Ein Beispiel für die Abbildung der Aktivität in Java ist in Anhang B zu sehen.

Die automatisierte Testfallerzeugung hat aber auch ihre Grenzen. Besonders bei der Modellierung des erwarteten Verhaltens müssen entweder umständliche Konstrukte modelliert oder Programmcode mit OpaqueExpression Elementen in den Modellelementen eingebettet werden. Die Beschreibung des Verlaufs eines Signals beispielsweise lässt sich nur sehr schwer in ein SysML Diagramm modellieren. An solchen Stellen lässt es sich aus Gründen der Effizienz nicht vermeiden, Programmcode in den Elementen einzubetten.

Name des Modell-elements	Modellelement	Abbildung
SendSignalAction		Das Modellelement SendSignalAction stellt das Senden eines Signals dar. Das Signal in der Eigenschaft "signal" muss eines der Signale aus dem Datenpool sein, damit der Codegenerator weiß, welches Datenpool-Element er setzen soll. Dadurch, dass es eine bidirektionale Beziehung zwischen den Signalen und den Ports des SuT gibt, muss die "target"-Eigenschaft des SendSignalAction Elements nicht gesetzt werden. Der Codegenerator kann diese Beziehung ermitteln. Das Signal Element muss den Stereotypen «EquivalenceClass» oder «DataPoolItem» haben, damit der zu setzende Wert ermittelt werden kann. Der Wert eines Signals mit dem Stereotypen «EquivalenceClass» ist der Median des Wertebereichs welches das Signal definiert. Wird ein Signal mit dem Stereotypen «DataPoolItem» gesetzt, so wird der zu setzende Wert aus den Testfall-Parametern erschlossen.
AcceptEventAction		Das AcceptEventAction Element wird in einem Testfall als Abfrage definiert. Solange diese Abfrage nicht wahr ist, wird keine weitere Aktion ausgeführt. Dieses Verhalten soll eine blockierende Lese-Operation imitieren. Der Codegenerator liest das Event aus dem Element zugehörigen Trigger. Der Codegenerator kann SignalEvents und TimeEvents interpretieren. Ein SignalEvent wird als Wert eines Wertebereichs auf einem Port des SuT interpretiert. Solange also nicht ein Wert im dem vom Signal definierten Wertebereich auf dem entsprechenden Port liegt, wird keine weitere Aktion ausgeführt. Das Signal aus der "signal"-Eigenschaft des SignalEvents muss den Stereotypen «EquivalenceClass» oder «DataPoolItem» haben und im Datenpool des Tests definiert sein. Wenn das Signal den Stereotypen «EquivalenceClass» trägt, wird eine Abfrage auf den in den Stereotypen-Attributen definierten Wertebereich generiert. Für Signale mit dem «DataPoolItem» Stereotypen wird der Wertebereich aus den Testfall-Parametern bezogen.

Tabelle 3: Abbildung der Element in Programmcode (Teil 1)

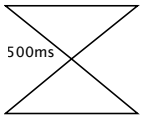
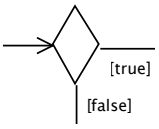
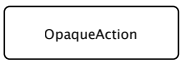
Name des Modell-elements	Modellelement	Abbildung
AcceptEventAction mit TimeEvent		Das AcceptAventAction Element mit einem getriggerten TimeEvent wird als Warte-Funktion realisiert. Der Codegenerator benötigt dazu nur die Information über die Zeitdauer, die er aus dem TimeExpression Element des TimeEvents erhält.
DecisionNode		Das DecisionNode Element verzweigt den Flusslauf der Aktivität. Diese werden in einer If-Else-Struktur abgebildet. Wichtig hierbei ist, dass die Bedingungen der ausgehenden ControlFlow Elementen orthogonal zueinander sind. Sie dürfen keine Überschneidungen in den Bedingungen zueinander habe, sonst erhält man ein unkontrolliertes Verhalten im Testfall. Der Codegenerator macht keine Überprüfung auf so eine Eigenschaft. Die Umsetzung solch einer Überprüfung wäre für weiterführende Studien zu dieser Arbeit interessant. In den Bedingungen der ControlFlow Elemente können Testfall-Parameter für die Steuerung des Testfall-Flusses gut eingesetzt werden.
CallOperationAction		Das Modellelement CallOperationAction wird in einen Funktionsaufruf abgebildet. Die Eigenschaft "operation" des Elements stellt eine Referenz zu einer im Testmodell definierten Operation her. Die Operation im Modell muss ein definiertes Verhalten in der Eigenschaft "method" haben. Der Codegenerator kann OpaqueBehavior für diese Eigenschaft interpretieren. Die "language"-Eigenschaft des OpaqueBehavior Elements muss dafür "Java" und der Programmcode im "body" Java konform sein.
OpaqueAction		Falls die schon aufgezählten Modellelemente nicht ausreichen, um eine Ablaufreihenfolge für die Stimuli im Test zu definieren, können auch die OpaqueAction Elemente herangezogen werden. Sofern die "language"-Eigenschaft des Elementes auf Java gesetzt ist, wird der Programmcode in der "body"-Eigenschaft des Elements vom Codegenerator eins zu eins übernommen.

Tabelle 4: Abbildung der Element in Programmcode (Teil 2)

6. Zusammenfassung

In dieser Arbeit wurde beispielhaft gezeigt, wie man ein Testmodell für das MBT mit SysML aufbauen kann. Mit den Block-Definitions-Diagrammen und den Internen-Block-Diagrammen werden die Schnittstellen der Testkomponenten, die Testdaten und die Funktionen zur Manipulierung der Testkomponenten definiert (4.1.1). In der Definition der Schnittstellen wurde der Aufbau der Testumgebung berücksichtigt. Somit sind auch Designentscheidungen über den Testaufbau in das Testmodell eingeflossen. Die Testumgebung wurde gewissermaßen mit modelliert.

Mithilfe von Aktivitäten als Stimuli-Generator im Testmodell wird ein Generierungsverhalten von Stimuli an das SuT festgelegt. Parameter auf den Aktionen und den Konnektoren, die nach Entscheider-Knoten stehen, steuern die Abfolge der Stimuli und die Daten mit denen stimuliert wird (4.1.2).

Zustandsautomaten beschreiben das Soll-Verhalten des SuT in den einzelnen Zuständen. Sie definieren ein Soll-Verhalten, welches während der Ausführung der Testfälle mitläuft und mit dem Ist-Verhalten verglichen wird (4.1.2).

Weiterhin wurden die Aktivitäten dafür verwendet, die Vorbedingungen für die Testfälle und die Generierungsregeln auf den Testdaten festzulegen (5.3.1).

Über die Informationen der Schnittstellen und den Testdaten werden die Testkonfigurationen für das Mapping der Schnittstellen bezogen. Zustandsdiagramme, welche die Zustände des SuT beschreiben, werden für die Bildung von Testszenarien herangezogen.

Für das Integrieren von test-relevanten Daten in das Testmodell wurden Stereotypen eingeführt. Sie bieten die Möglichkeit über ihre Eigenschaften Werte in das Modell zu integrieren. Somit können alle test-relevanten Daten in das Modell kodiert werden. Außerdem unterstützen sie den Codegenerator die Modellelemente eindeutig zu interpretieren und die test-relevanten Daten direkt aus dem Modell auszulesen. Die Stereotypen sind weitestgehend von der Testnotationssprache unabhängig.

Es wurden Algorithmen diskutiert, die systematisch Informationen aus den Modellen auslesen und sie in einen logischen Zusammenhang bringen, um Testfälle ableiten zu können. Der Algorithmus aus Kapitel 4.2.1 findet den kürzesten Pfad für einen Graphen, der alle Kanten mindestens einmal durchlaufen ist. Ein solcher Pfad kann ein Testfall aus einem Zustandsautomaten sein, der das Zustandsübergangsüberdeckungs-Kriterium erfüllt. Ein weiterer Algorithmus liest systematisch alle Datentypen aus dem Modell, erzeugt Generierungsregeln und übergibt sie einem weiteren Werkzeug, um daraus Testfälle in Form von datengetriebenen Tests zu generieren (4.2). Ein Algorithmus für das Finden und Auslesen von verschiedenen Anwendungsfällen aus einem Aktivitätsdiagramm ist in Kapitel 4.2 erörtert wurden.

In Kapitel 5.2 wurde der Testaufbau für das Fallbeispiel diskutiert und welche Designmöglichkeiten es gibt. In diesem Zusammenhang wurden die Designentscheidungen mit den verwendeten Technologien erklärt und welche Vor- bzw. Nachteile die Technologien und Desi-

gnentscheidungen mit sich bringen. Eine Werkzeugkette für die in dieser Arbeit erarbeiteten Methodologie wurde anhand des Fallbeispiels eingeführt und demonstriert (5.2). Als letztes wurde vorgestellt, wie die Informationen aus dem Modell in Testfälle und anderen Konfigurationsdateien einfließen. Eine M2T-Transformation aus verschiedenen Modellelementen nach Programmcode wurde in 5.3 zusammengestellt.

6.1. Fazit

Die Modellierung eines Testmodells kann als ein zusätzlicher Aufwand wahrgenommen werden, zumal die Schnittstellen und das Verhalten des SuT explizit in dem Modell abgebildet werden müssen. Auch die Umgebungskomponenten sollten in diesem Modell enthalten sein, um die Beziehungen mit anderen Systemen in der Testumgebung klar zu definieren. Sieht man von dem Aufwand der Modellierung dieses Modells aber ab, stehen diverse Nutzeffekte dem gegenüber.

In der Design- und Entwicklungs-Phase eines Systems kann der Testdesigner nebenläufig das Testmodell erstellen. Während der Erstellung des Testmodells lassen sich unter Umständen Lücken in der Spezifikation des SuT finden. Dieser Eindruck wurde auch in dieser Arbeit gewonnen. Das Modellieren der Testfälle warf Fragen zur Spezifikation auf, welche im Nachhinein in der Spezifikation zu neuen oder verfeinerten Anforderungen führte.

Dadurch, dass die Testmodelle auf einem hohen Abstraktionsgrad das Verhalten des SuT beschreiben, kann mit dem Design des Testmodells in einer frühen Phase der Entwicklung begonnen werden. Unabhängig von der Technologie der Testumgebung und von der Implementierung des SuT, wird mit dem Testmodell das Testdesign festgelegt. Die Wahl der Testkriterien für eine bestimmte Aussagekraft der Testfälle ist nach wie vor vom Testdesigner abhängig. Der Testdesigner muss sich aber bei der Modellierung des Testmodells nicht mehr so viele Gedanken über die Test-Technologie machen. Außerdem ist die Qualität der erzeugten Testfälle in der Regel besser als die per Hand geschriebenen Testfälle [30]. Das liegt zum einen Teil an der Fehlerträchtigkeit des Menschen, im Code den Überblick zu verlieren und daher Fehler zu machen, und zum anderen an der gleichbleibenden Qualität des Codes durch die Erstellung eines Codegenerators. So ein Codegenerator kann bei großen Testumgebungen den Vorteil bringen, dass er für verschiedene Simulatoren und Testumgebungen optimierten Programmcode generiert. Der Benutzer muss sich weniger Gedanken über eine Optimierung des auszuführenden Codes auf einem Simulatoren machen.

Für den Beweis der Machbarkeit der MBT Methodologie mit SysML wurde ein zu komplexes Fallbeispiel in dieser Arbeit gewählt. Die funktionalen Tests und die Zusammenhänge der zahlreichen Anforderungen untereinander waren so komplex, dass die Beschreibung des ganzheitlichen Tests zu umfangreich gewesen wäre. Daher wurden nur Teilbereiche und ausgewählte Testfälle in dieser Arbeit diskutiert. Für weitere Machbarkeitsstudien in diesem Themenbereich wird daher empfohlen ein überschaubares Fallbeispiel zu wählen, welches die zu untersuchenden Aspekte beinhaltet aber nicht zu komplex ist. Soweit es sich eignet,

sollte vielleicht ein schon oft verwendetes Fallbeispiel ausgewählt werden, wie das "Two-Tank" System. Dadurch erhält man obendrein eine bessere Vergleichbarkeit mit anderen Arbeiten in diesem Themenbereich, die dasselbe Beispiel verwenden. Der Nachteil, der dabei aber aufkommen kann, ist, dass eine Untersuchung an immer demselben Beispiel nur eine Machbarkeit auf diesem Beispiel zeigt. Für die Erhöhung der Aussagekraft solcher Untersuchungen, wäre eine möglichst breite Palette von Anwendungsbeispielen wünschenswert. Die Intention des Fallbeispiels in dieser Arbeit war die Verwendung eines praxisnahen Beispiels.

Das SuT in dieser Arbeit ist ein Simulationsmodell aus Matlab/Simulink. In Matlab/Simulink werden numerische Simulationen durchgeführt. Die Simulationszeit ist so lang, wie der simulierende Rechner braucht, um die Simulationsschritte auszurechnen. Für sehr langsame SuTs würde das heißen, dass die Simulationszeit viel kürzer als die physikalische Zeit des SuT wäre. Dadurch bekäme man ein Zeitersparnis gegenüber einer real-time Laufzeitumgebung, wie sie in dieser Arbeit verwendet wurde. Zudem würde der simulierende Rechner bei sehr schnellen SuTs sich soviel Zeit nehmen wie er braucht, um die einzelnen Simulationsschritte zu berechnen. Der Simulationsrechner muss nicht notwendigerweise eine sehr schnelle Hardware besitzen, um Simulationen auf schnellen SuTs ausführen zu können. Infolgedessen ist es ratsam bei sehr schnellen SuTs, Simulationen des SuTs in einer Simulationsumgebung wie Matlab/Simulink durchzuführen. Besonders für die ersten Tests zur grundsätzlichen Funktionalität als Probelauf (hier ist Smoke Testing gemeint), eignet sich die Matlab/Simulink Umgebung sehr gut. Man kann sofort nach der Modellierung eine Simulation starten und einen Probelauf durchführen. Wiederum eignet sich Matlab/Simulink nicht zum definieren und treiben von komplexen Testfällen, weil man per Hand jedes einzelne Signal zu einem Zeitpunkt definieren muss. Die Definition von Testfällen auf diese Art und Weise wäre sehr mühselig und aufwenig.

Mit einer real-time Laufzeitumgebung hingegen ist der Vorteil gegeben, dass man XiL-Testmethoden durchführen kann. Man hat die Möglichkeit SuTs in verschiedenen Integrationsstufen mit der gleichen Testumgebung zu testen. Man kann sogar darüber hinaus verschiedene Komponenten in verschiedenen Integrationsstufen miteinander in einer Testumgebung mischen. Bei diesen Test nimmt man aber in Kauf, dass Testdurchführungen in der Zeitdauer sehr lang werden können, wenn das SuT langsam ist, oder dass sehr schnelle und damit kostenintensive Hardware angeschafft werden muss, falls der Test auf dem SuT sehr schnelle Reaktionszeiten fordert.

Bei der Verwendung von Stubs in der Testumgebung ist Vorsicht geboten. Die Stubs haben ihr eigenes Verhalten und laufen unabhängig vom Testcontroller. Aus diesem Grunde können keine Eingabe-Mengen für einen Testfall aus einem Verhaltensmodell erschlossen werden, bei denen einige Eingaben von einem Stub erzeugt werden könnten. Das Stub könnte zu einem ungewollten Verhalten führen. Die Eingänge das SuT, die von einem Stub bedient werden, sollten daher nicht für die Generierung von Eingabe-Mengen für Testfälle herangezogen werden oder man wandelt die Eingaben aus den Testfällen in kontrollierte Funktionsaufrufe auf den Stubs um.

In dem Kapitel 4.2 wurde gezeigt, dass die Verwendung von Algorithmen oftmals Randbedingungen an das Modell fordert. Der Algorithmus (4.2.1) für das Finden von Pfaden, die alle Zustände oder Zustandsübergänge durchlaufen, fordert beispielsweise, dass der Graph stark zusammenhängend ist. Möchte man demnach diesen Algorithmus auf einem Zustandsautomaten anwenden, um einen Testfall zu identifizieren, der alle Zustandsübergänge mindestens einmal ausgelöst hat, muss der Automat so gestaltet sein, dass man von jedem Zustand aus jeden anderen Zustand über eine Eingabe-Menge aktivieren kann. Weiterhin besteht auch die Möglichkeit, falls die Randbedingungen für einen Algorithmus nicht gegeben sind, einen anderen nicht so optimalen Algorithmus zu verwenden, der aber zumindest das Testkriterium erfüllen kann.

Um konkrete Werte oder Programmcode in das Modell zu integrieren, kann man eigene Stereotypen mit Eigenschaften definieren oder die "Opaque"-Elemente in der SysML verwenden. Für die vollständige Abbildung der Tests in den Testmodellen, werden einige testrelevante Informationen für den Test des Fallbeispiels benötigt, die mit den Stereotypen in Abbildung 12 dargestellt sind.

In dieser Arbeit wurde gezeigt, wie man Beziehungen zwischen den Anforderungen und den Testfällen im Testmodell festlegen kann. Die Anforderungen können Beziehungen zu diversen Testfällen haben und die Testfälle abermals mit verschiedenen Anforderungen. In dieser Arbeit wurden nur mentale Anforderungen eingesetzt. Es ist deshalb nicht möglich, Bedingungen für die Verifikation aus den Anforderungen zu generieren. Hatten allerdings alle Testfälle, die mit einer Anforderung in Beziehung stehen, ein positives Testergebnis, so ist diese Anforderung für die Testdokumentation hinreichend gut getestet worden. Durch das Testen können Fehler ausfindig gemacht werden. Damit ist aber nicht die Abwesenheit von Fehlern bewiesen. Es können noch Fehler in den Funktionen des SuT vorhanden sein, die aus verschieden Gründen, die weiter unten angeführt werden, nicht gefunden werden konnten. Im Umkehrschluss heißt das, dass die Testfälle, die negative Ergebnisse hatten, für weitere Untersuchungen herangezogen werden müssen. Solange nicht identifiziert werden kann, wo der Fehler auftritt, gilt, dass eine Anforderung, die in Beziehung zu einem Testfall mit einem negativen Testergebnis steht, nicht erfüllt ist.

Es kann unter Umständen schwierig sein herauszufinden, woher ein auftretender Fehler stammt. Die Ursache von negativen Ergebnissen in den Testfällen kann vielfältige Gründe haben. Die Gründe können folgende sein:

Fehler im SuT Der Grund für das Testen ist das Aufdecken von Fehlern im SuT. Falls ein Fehler entdeckt wird, der seinen Ursprung in einer falsch implementierten Funktion des SuT hat, ist das so gewollt und der Entwickler kann diesen Fehler zur Behebung angehen.

Falsche Abbildung des Verhaltens oder der Schnittstellen des SuT im Testmodell Das modellbasierte Testen kann dem Testdesigner nicht abnehmen, die Testfälle im Modell zu entwickeln. Nach wie vor muss der Test-Designer sich darüber Gedanken

machen, einen sachgemäßen und relevanten Test zu definieren. Er kann beispielsweise ein falsches Verhalten im Testmodell abbilden, welches dann vom Codegenerator in den Test mit übernommen wird. So kann ein Fehler von einem Testfall detektiert werden, der keiner ist. Ereignisse, die nicht im Verhaltensmodell definiert sind, aber beim SuT zu eine korrekten Reaktion führen würden, würden als ein Fehler detektiert werden, der keiner ist. Auch eine falsche Definition der Äquivalenzklassen auf den Wertebereichen der Schnittstellen würde insbesondere bei den Tests mit einer Grenzwertanalyse zu einem negativen Testergebnis führen.

Fehler im Testfall-Generator Der Codegenerator, welcher aus den Testmodell Testfälle generiert, kann Fehler in der Generierung oder Interpretation der Informationen aus dem Testmodell beinhalten. Solch ein Fehler würde sich in den Testfällen fortpflanzen und könnte dazu führen, dass ein Fehler im SuT detektiert wird, der keiner ist. Aber auch umgekehrt können Fehler im SuT von den generierten Testfällen nicht entdeckt werden, weil der Codegenerator die Bedingungen aus dem Testmodell nicht richtig im Testfall umgesetzt hat.

Die Simulationsumgebung für den Test führt zu einen Fehler Selbst die Simulationsumgebung kann dazu beitragen, einen Fehler im SuT zu attestieren, wo keiner ist, oder einen Fehler nicht zu entdecken. Ist die Simulationsschrittweite zum Vergleich der Verhalten sehr groß oder der Simulator nicht schnell genug, kann es vorkommen, dass ein Verhalten des SuT vom Test nicht wahrgenommen werden konnte.

Auch das Nicht-Aufspüren von Fehlern im SuT kann aus diversen Gründen sich ereignen, wie schon in Punkt 3 und 4 angeführt. Zudem können unsinnige Metriken und Testkriterien suggerieren, dass das SuT ausreichend und positiv getestet wurde, aber von der Relevanz nicht viel Aussagekraft haben. Die Aufgabe für die Definition der Metriken und Testkriterien obliegt daher weiterhin dem Testdesigner, weil nur er Entscheidungen über die Relevanz dieser Kriterien treffen und kein Automatismus das übernehmen kann.

Die Verhaltensmodelle im Testmodell müssen nicht das vollständige Verhalten des SuT abbilden. Für den Test reichen die test-relevanten Aspekte im Verhaltensmodell des Testmodells. Es birgt aber wiederum die Gefahr, dass unerwartet ein Verhalten des SuT im Testfall ausgelöst wird, welches nicht im Verhaltensmodell des Testmodells abgebildet ist und dennoch korrekt ist.

Ein großer Gewinn am modellbasierten Testen ist die Wiederverwendbarkeit der Modelle in vielerlei Hinsicht. Bei Änderungen am SuT kann das Testmodell auf diese Änderungen angepasst werden, ohne alle Konfigurationen und Testfälle händisch zu bearbeiten. Durch den hohen Abstraktionsgrad im Testmodell sind die zu ändernden Stellen relativ leicht zu finden. Mit der Änderung der Stellen im Testmodell und einem weiteren Anstoßen des Codegenerators, wären alle Konfigurationen und Testfälle auf die Änderung angepasst. Wenn beispielsweise Anforderungen veraltet sind oder verfeinert werden müssen, können die Anforderungen gezielt im Modell ausfindig gemacht werden und jede Beziehung zwischen der Anforderung und anderen Anforderungen oder Modellelementen verfolgt werden, um die

wichtigen Stellen in den Testfällen und der Testkonfiguration hinsichtlich der Änderung zu identifizieren und ändern zu können.

Bei der Einarbeitung Dritter in die Testfälle, hilft das Testmodell als Kommunikations-Grundlage und Dokumentation. Ferner kann eine Dokumentation über das Testdesign, die Konfiguration und den Testfällen automatisch mit einem Dokumentengenerator erstellt werden.

Die SysML bietet all diese Nutzeffekte für eine Modellierung von Testmodellen, wie sie in dieser Studie diskutiert wurden. Die Konzepte aus der SysML zur Modellierung von Strukturen und Verhalten sind für das MBT gut geeignet. Besonders mit der Einbeziehen der Anforderungen in das Modell bietet die SysML gegenüber vielen anderen Modellierungssprachen, die das nicht haben, einen Vorteil. Durch die Verfolgbarkeit (siehe 4.1.2) im Modell können Zusammenhänge und Rückschlüsse zwischen den Modellelementen und den Testergebnissen gezogen werden.

Die SysML besitzt aber auch viele Modellelemente, welche in der Methodologie dieser Arbeit keine Verwendung finden. Bezüglich der Möglichkeiten des Codegenerators, die Modellelemente und die Beziehungen zueinander zu interpretieren, muss die SysML in den Modellierungsmöglichkeiten der Modellelemente eingeschränkt werden. Es gibt eine starke Abhängigkeit zwischen der Syntax und den vom Codegenerator verwendeten Algorithmen. In dieser Arbeit wurden beispielsweise die Aktivitätsdiagramme stark eingeschränkt. Trifft der Codegenerator auf ein Modellelement, dass er nicht interpretieren kann, so bricht der Generator an dieser Stelle mit einer Fehlermeldung ab. Viele Eigenschaften von Modellelementen werden vom Codegenerator zum Teil auch nicht berücksichtigt. Diese Einschränkung der SysML war nötig, um eine praktikable Methodologie für das MBT in der zur Verfügung stehenden Zeit zu entwickeln.

Eines der größten Schwierigkeiten bei der Modellierung des Testmodell ist der konsistente Aufbau. Bevor nicht alle Signale, Events, Funktionen, Strukturen, Bedingungen, Anforderungen und Eigenschaften auf den einzelnen Modellelementen im Modell definiert sind, können die Verhaltensmodelle nicht entworfen werden, damit aus ihnen Programmcode für Testfälle erzeugt werden kann. Der Codegenerator benötigt diverse Informationen aus den Modellen und bricht bei jedem Fehler oder Fehlen von Informationen ab. Besonders an den Stellen, wo Beziehungen über Namen hergestellt wurden, führen Schreibfehler der Namen zu einer Inkonsistenz und somit zu einem Abbruch des Codegenerators. An den Stellen des Modells, wo Bedingungen oder für den Codegenerator interpretierbarer Programmcode steht, sind solche Inkonsistenzen oft aufgetreten. Durch eine exakte Fehlermeldung durch den Codegenerator oder einem Modell-Checker lässt sich dieses Problem aber in den Griff bekommen. Der Nachteil, dass erst alle vom Codegenerator benötigten Informationen im Modell verhanden sein müssen, bleibt allerdings. Man erhält lediglich einen Hinweis darüber, welche Informationen unter Umständen im Modell fehlen oder wo im Modell beim interpretieren Fehler aufgetreten sind.

6.2. Ausblick

Falls nur MiL-Simulationen im Fokus der Tests steht, könnte eine andere Laufzeitumgebung als in dieser Arbeit verwendet werden. Viele Arbeiten beschäftigen sich mit der Testdurchführung von Matlab/Simulink Modellen in der Simulationsumgebung von Simulink. Die Simulationsumgebung von Simulink oder ähnliche Laufzeitumgebungen bieten an, die Simulationszeit variabel anzupassen. Damit hätte man einige Vorteile gewonnen, wie in vorigem Kapitel aufgeführt.

Bisher wurden die Signale in den Testfällen über die Zeit auf absolute Werte gesetzt. In der Regel hat man bei realen Systemen aber einen Verlauf von Signalwerten. Sprunghafte Signale im analogen Bereich sind eher die Ausnahme. Es wäre zu untersuchen, wie man Signalverläufe in SysML Modelle modellieren und in die Testfälle mit aufnehmen kann. Das Fraunhofer-Institut für experimentelles Software Engineering entwickelt beispielsweise an einem Produkt, dass sich vollständig in Matlab/Simulink integriert und mit dem man MBT betreiben kann. Das Produkt namens SIMOTEST [9] basiert auf dem Standard IEEE 1641, zur Definition verschiedener Signalverläufe. Mit diesem Framework sollen Definitionen von Signalverläufen möglich sein, die man für die Stimulierung des SuT verwenden kann.

Die Anforderungen an das SuT sind in dieser Arbeit in natürlicher Sprache erfasst. Man könnte untersuchen, inwieweit man die funktionalen Anforderungen formalisieren kann. Aus formalisierten Anforderungen wäre es möglich, automatisiert Bedingungen für die Testfälle abzuleiten.

Der Algorithmus aus Kapitel 4.2.1 berücksichtigt für die Berechnung eines kurzen Pfades Kantengewichte. In dieser Arbeit wurde der Algorithmus auf den Zustandsautomaten angewendet und die Kantengewichte wurden immer auf eins gesetzt. Für weitere Studien wäre es interessant zu untersuchen, wie die Kantengewichte verwendet werden können, um die Berechnung von Pfaden insoweit zu beeinflussen, dass bestimmte Pfade in einem Graphen (bspw. Zustandsautomat) bevorzugt verwendet werden und man dadurch als Testdesigner eine Möglichkeit zur Beeinflussung des Algorithmus erhält.

Das reine Black-Box basierte Testvorgehen mit XiL-Methoden ermöglicht die Übertragbarkeit der Testkonfigurationen und Testfälle auf anderen SuT-Implementierungen, unabhängig der Integrationsstufe oder der Version/Variante des SuT. Lediglich die Schnittstellen und das Verhalten sollten mit dem Testmodell übereinstimmen. Ferner wäre aber eine Änderung am SuT relativ einfach in das Testmodell übertragbar, weil das Modell von einer Technologie abstrahiert. Dadurch wären alle aus dem Testmodell generierten Testkonfigurationen und Testfälle gemäß der Änderung konsistent angepasst. Der Aufwand, jeden Testfall oder jede Testkonfiguration anzupassen, wäre relativ gering. Beim Wechsel des SuT auf eine andere Integrationsstufe wäre einzig zu beachten, dass die Technologien für die Schnittstellen der Testumgebung mit dem SuT sich womöglich ändern würden. Die Testkonfiguration und die Testfälle wären an dieser Stelle nicht betroffen. Eine Untersuchung könnte zeigen, inwieweit Probleme auftauchen, wenn die Schnittstellen verändert werden, sowohl technologisch als auch im Testmodell.

Ein SysML Diagrammtyp, welcher in dieser Arbeit weitestgehend vernachlässigt wurde, ist das Zusicherungsdiagramm. Mit diesem Diagrammtyp lassen sich Bedingungen und Zusicherungen in Form von mathematischen Gleichungen zwischen Komponenten oder Schnittstellen modellieren. Eine Untersuchung könnte zeigen, ob und wie man mit den Zusicherungsdiagrammen Soll-Funktionen oder Prüfbedingungen für Testfälle modellieren kann.

Literatur

[1] Alt, O.: Generierung von Systemtestfällen für Car Multimedia Systeme aus domänenspezifischen UML Modellen, pp. 215 – 222. Gesellschaft für Informatik (2006)

[2] Alt, O.: Integration textueller anforderungen und modell-basiertem testen mit sysml. Ph.D. thesis, Fachhochschule Köln (2007)

[3] Apfelbaum, L., Doyle, J.: Model based testing. In: Software Quality Week Conference (1997)

[4] Baker, P., Dai, Z.R., Grabowski, J., Haugen, O., Schieferdecker, I., Williams, C.: Model-Driven Testing. Springer-Verlag Berlin (2007)

[5] Bundesverband Windenergie e.V.: Windenergie (2011). URL http://www.wind-energie.de/

[6] Dr. Horst Kargl: Flexibilität beim modellbasierten testen. Tech. rep., Sparx Systems (2009)

[7] Dr.-Ing. Clemens Gühmann: Modellbildung und testautomatisierung für die hardware-in-the-loop simulation. ACM (2007)

[8] Dr.-Ing. Ingo Stürmer: Maja - matlab simulink and stateflow java adapter (2010). URL http://www.model-engineers.com/de/our-company/research-projects/maja.html

[9] Dr. Robert Eschbach: Modellbasiertes testen von simulink-modellen mit simotest (2010). URL http://www.iese.fraunhofer.de/de/Images/SIMOTEST_dt_2010_05_tcm122-46543.pdf

[10] Fraunhofer-Institut FOKUS: Motion (2011). URL http://www.fokus.fraunhofer.de/de/motion/ueber_motion/arbeitsthemen/modellbasiertes_testen/index.html

[11] Friedenthal, S., et al.: A Practical Guide to SysML. Morgan Kaufmann OMG Press imprint of Elsevier (2009)

[12] Friske, M., Schlingloff, H.: Von use cases zu test cases: Eine systematische vorgehensweise. Tech. rep., Fraunhofer FIRST (2005)

[13] Gaston, C., Seifert, D.: Evaluating coverage based testing (2005). URL http://lfm.iti.uni-karlsruhe.de/download/EvaluatingCoverageBasedTesting2005.pdf

[14] GFSE: Best practices and guidelines for modeling with sysml (2010)

[15] GfSE: Best practices and guidelines for modeling with sysml (2010). URL http://mbse.gfse.de/documents/faq.html

[16] Gnesi, S., Latella, D., Massink, M.: Formal test-case generation for uml statecharts. In: Engineering Complex Computer Systems (2004)

[17] Google Project Hosting: A java api to control and interact with matlab (2010). URL http://code.google.com/p/matlabcontrol/

[18] Götz, H., Nickolaus, M., Rolfner, T., Salomon, K.: IX Studie: Modellbasiertes Testen. Heise Zeitschriften Verlag GmbH und Co KG (2009). URL http://www.heise.de/kiosk/special/ixstudie/09/01/

[19] IBM: What's new in ibm rational rhapsody, version 7.5.2 (2010). URL http://www.ibm.com/developerworks/rational/library/10/whatsnewinrationalrhapsody7-5-2/index.html?ca=drs-

[20] IEEE: Ieee standard glossary of modeling and simulation terminology (1989)

[21] Korff, A.: Modellierung von eingebetteten Systemen mit UML und SysML. Spektrum Akademischer Verlag Heidelberg (2008)

[22] Neto, A., et al.: Characterization of model-based software testing approaches. Tech. rep., SIEMENS Corporate Research (2007). URL www.cos.ufrj.br/uploadfiles/1188491168.pdf

[23] Neto, A.D., et al.: Improving evidence about software technologies - a look at model-based testing. IEEE Xplore (2008)

[24] Offut, J., Li, S., Abdurazik, A., Ammann, P.: Generating test data from state-based specification. In: The Journal of Software Testing, Verification and Reliability (2003)

[25] OMG: Mof model to text transformation language (mofm2t) 1.0 (2008). URL http://www.omg.org/spec/MOFM2T/1.0/

[26] OMG: Omg systems modeling language (sysml) (2010). URL http://www.omg.org/spec/SysML/1.2/

[27] OMG: Webseite der omg (2010). URL http://www.omg.org/

[28] OMG: Uml testing profile (2011). URL http://utp.omg.org/

[29] Pretschner, A.: Zur kosteneffektivität des modellbasierten testens. Tech. rep., ETH Zuerich (2006)

[30] Pretschner, A., Philipps, J.: Methodological issues in model-based testing. In: Model-Based Testing of Reactive Systems, pp. 281–293. Spinger Verlag (2005)

[31] Santos-Neto, P., et al.: Requirements for information systems model-based testing. Tech. rep., Belo Horizonte/MG (2007)

[32] Sarstedt, S.: Semantic foundation and tool support for model-driven development with uml 2 activity diagrams. Ph.D. thesis, Universität Ulm (2006)

[33] Schlager, M.: Hardware-in-the-Loop Simulation. VDM Verlag (2008)

[34] Schlecht, S., Alt, O.: Strategien zur testfallgenerierung aus sysml modellen. Software Engineering 2007- Beiträge zu den Workshops Fachtagung des GI-Fachbereichs Softwaretechnik (2007)

[35] Spillner, A., Linz, T.: Basiswissen Softwaretest. dpunkt.verlag GmbH (2005)

[36] Stachowiak, H.: Allgemeine Modelltheorie. Springer (1974)

[37] Stadtwerke Frankfurt (Oder): Die unsichtbare kraft (2011). URL http://www.energie-wasser-besser-verstehen.de/news/energie.php?idkat=14&kdid=10&layoutid=9

[38] The Math Works: Matlab / simulink (2011). URL http://www.mathworks.de/products/simulink/

[39] Thimbleby, H.: The directed chinese postman problem. Tech. rep., UCLIC (University College London Interaction Centre) (2003)

[40] Utting, M., Legeard, B.: Practical model-based testing. Elsevier (2007)

[41] Weilkiens, T.: Systems Engineering mit SysML/UML. dpunkt.verlag GmbH (2008)

[42] Whitehouse, K.: Calling matlab from java (2010). URL http://www.cs.virginia.edu/~whitehouse/matlab/JavaMatlab.html

[43] Wind River: Wind river compiler (2010). URL http://windriver.com/products/development_suite/wind_river_compiler/

[44] Wind River: Wind river vxworks (2010). URL http://windriver.com/products/vxworks/

A. Anforderungen

Dokumentenname	WKA_Requirements_0.3.xls
Version	0.4
Datum	05.08.2010
Autor	R. Batista und Daniel Lorenz

ID	Kurztext	Anforderungstext
A-CTRL-001	Normaler Betriebsmodus	Der Azimuth-Regler soll einen normalen Betriebsmodus haben.
A-CTRL-002	Manueller Betriebsmodus	Der Azimuth-Regler soll einen manuellen Betriebsmodus haben.
A-CTRL-003	Sicherheitsabschaltungsmodus	Der Azimuth-Reger soll einen Betriebsmodus “Sicherheitsabschaltung” haben.
A-CTRL-004	Fehlermodus	Der Azimuth-Regler soll einen Fehlermodus haben
A-CTRL-005	Eingang “WindrichtungEin”	Der Azimuth-Regler soll einen analogen Signaleingang “WindrichtungEin” haben. Das Signal am Eingang “WindrichtungEin” variiert innerhalb eines Spannungsbereichs 1-2 V, der linear der Position 0-360 Grad entspricht.
A-CTRL-006	Eingang “WindgeschwindigkeitEin”	Der Azimuth-Regler soll einen analogen Signaleingang “WindgeschwindigkeitEin” haben. Das Signal am Eingang “WindgeschwindigkeitEin” variiert innerhalb eines Spannungsbereichs 1-3,5 V, der linear dem Geschwindigkeitsbereich 0-50 m/s entspricht.
A-CTRL-008	Eingang “SicherheitsabschaltungEin”	Der Azimuth-Regler soll einen analogen Signaleingang “SicherheitsabschaltungEin” haben. Das Signal am Eingang “SicherheitsabschaltungEin” liegt zwischen 0..5 V, wobei 1+/-1V als “AN” gewertet werden soll und 4+/-1V als “AUS” gewertet werden soll.
A-CTRL-009	Eingang “ManuellePositionEin”	Der Azimuth-Regler soll einen analogen Signaleingang “ManuellePositionEin” haben. Das Signal am Eingang “ManuellePositionEin” variiert innerhalb eines Spannungsbereichs 1-2V, der linear der Position 0-360 Grad entspricht.
A-CTRL-010	Eingang “AktuellePositionEin”	Der Azimuth-Regler soll einen analogen Signaleingang “AktuellePositionEin” haben. Das Signal am Eingang “AktuellePositionEin” variiert innerhalb eines Spannungsbereichs 1-6V, der linear der Position 0-360 Grad entspricht.
A-CTRL-011	Eingang “ManuellerBetriebEin”	Der Azimuth-Regler soll einen analogen Signaleingang “ManuellerBetriebEin” haben. Das Signal am Eingang “ManuellerBetriebEin” liegt zwischen 1..6 V, wobei 1..2 V als “AN” gewertet werden soll und 4..6V als “AUS” gewertet werden soll.
A-CTRL-012	Ausgang "MotorAnsteuerungAus"	Der Azimuth-Regler soll einen analogen Signalausgang "MotorAnsteuerungAus" haben. Das Signal am Ausgang "MotorAnsteuerungAus" soll zwischen -12 .. 12 V liegen.
A-CTRL-013	Ausgang "BremsenAus"	Der Azimuth-Regler soll einen analogen Signalausgang "BremsenAus" haben. Das Signal am Ausgang "BremsenAus" soll zwischen 0.. 12 V liegen, wobei 0..2 V als "Bremsen anziehen" und Spannungen größer 2 V als "Bremsen lösen" gewertet werden soll.

A-CTRL-014	Wechsel zu Sicherheitsabschaltungsmodus	In jedem Betriebsmodus, außer dem Fehlermodus, soll der Azimuth-Regler in den Betriebsmodus "Sicherheitsabschaltung" wechseln, sobald an der Schnittstelle "SicherheitssystemEin" das Signal "AN" anliegt.
A-CTRL-015	Bremsen anziehen bei Sicherheitsabschaltung	Im Betriebsmodus "Sicherheitsabschaltung" soll der Azimuth-Regler über die Schnittstelle "BremsenAus" das Signal "BremsenAnziehen" senden.
A-CTRL-016	Wechsel von Sicherheitsabschaltung zu normalem Betrieb	Im Betriebsmodus "Sicherheitsabschaltung" soll der Azimuth-Regler in den normalen Betriebsmodus wechseln, sobald an der Schnittstelle "SicherheitsabschaltungEin" das Signal "AUS" anliegt und an der Schnittstelle "ManuellerBetriebEin" kein Signal "AN" anliegt.
A-CTRL-017	Wechsel von Sicherheitsabschaltung und normalem Betrieb zu manuellem Betrieb	Im Betriebsmodus "Sicherheitsabschaltung" und im normalen Betriebsmodus soll der Azimuth-Regler in den manuellen Betriebsmodus wechseln, sobald an der Schnittstelle "SicherheitsabschaltungEin" das Signal AUS anliegt und an der Schnittstelle "ManuellerBetriebEin" das Signal "AN" anliegt.
A-CTRL-018	Drehung der Gondel im manuellem Betrieb	Im manuellen Betriebsmodus soll der Azimuth-Regler die Gondel in die Position drehen, die durch das Signal an der Schnittstelle "ManuellePositionEin" vorgegeben wird, solange diese nicht ausserhalb der Bereichse 0-135 und 225-360 Grad liegt.
A-CTRL-019	Drehung der Gondel im manuellem Betrieb bei ungültiger Zielposition	Im manuellen Betriebsmodus und für den Fall, dass das Signal an der Schnittstelle "ManuellePositionEin" außerhalb der Bereiche 0-135 und 225-360 Grad liegt, soll der Azimuth-Regler die Gondel in die Position drehen, die am nächsten der durch das Signal an der Schnittstelle "ManuellePositionEin" vorgegebenen ist.
A-CTRL-021	Vermeidung von Schwankungen der Soll-Position im manuellem Betrieb	Der Wert vom Signal "ManuellePositionEin" ist nur dann gültig, wenn er mindestens 1 s lang unverändert bleibt.
A-CTRL-022	Drehung der Gondel im normalen Betrieb	Im normalen Betriebsmodus soll der Azimuth-Regler die Gondel in die Position drehen, die durch den 3-Sekunden-laufenden-Durchschnitt des Signals an der Schnittstelle "WindrichtungEin" vorgegeben wird, solange diese nicht ausserhalb der Bereiche 0-135 und 225-360 Grad liegt.
A-CTRL-024	Drehung der Gondel im normalen Betrieb bei ungültiger Zielposition	Im normalen Betriebsmodus und für den Fall, dass das Signal an der Schnittstelle "WindrichtungEin" außerhalb der Bereiche 0-135 und 225-360 Grad liegt, soll der Azimuth-Regler die Gondel in die Position drehen, die am nächsten der durch das Signal an der Schnittstelle "WindrichtungEin" vorgegebenen ist.
A-CTRL-026	Maximale Drehgeschwindigkeit der Gondel	In jedem Betriebsmodus soll der Azimuth-Regler die Drehgeschwindigkeit der Gondel auf 350Grad/5 Minuten begrenzen.

A-CTRL-027	Bremsen lösen vor dem anfahren des Motors	Vor jeder Bewegung der Gondel soll der Azimuth-Regler die Bremsen lösen und darauf warten, dass die Bremsen gelöst sind. Bremsen lösen bedeutet, dass über die Schnittstelle "BremsenAus" das Signal "Bremsen lösen" kontinuierlich geschickt wird. Bremsen gelöst bedeutet, dass über die Schnittstelle „BremsenEin“ das Signal “Bremsen gelöst“ kontinuierlich empfangen wird. "Vor jeder Bewegung" bedeutet: Bremsen sind angezogen und Gondel bewegt sich nicht.
A-CTRL-028	Bremsen ziehen nach jedem Erreichen einer Zielposition	Nach jeder Bewegung der Gondel soll der Azimuth-Regler über die Schnittstelle "BremsenAus" nach 5s nachdem der Motor die Bewegung beendet hat die Bremsen anziehen. Bremsen anziehen bedeutet, dass über die Schnittstelle "BremsenAus" das Signal "Bremsen anziehen" kontinuierlich geschickt wird. "Nach jeder Bewegung" bedeutet: Zielposition erreicht und keine neue Zielposition liegt an.
A-CTRL-029	Bremsen lösen während der Gondelbewegung	Wenn sich die Gondel in Bewegung befindet, soll der Azimuth-Regler die Bremsen lösen. Bremsen lösen bedeutet, dass über die Schnittstelle "BremsenAus" das Signal "Bremsen lösen" kontinuierlich geschickt wird.
A-CTRL-030	Dauer der Signalbearbeitung im manuellen Betrieb	Im manuellen Betriebsmodus soll der Regler die Gondelbewegung spätestens 500ms nach Empfang eines veränderten "ManuellePositionEin" Signals durch das Senden des entsprechenden Signals am Ausgang "MotorAnsteuerungAus" einleiten.
A-CTRL-031	Dauer der Signalbearbeitung im normalen Betrieb	Im normalen Betriebsmodus soll der Regler die Gondelbewegung spätestens 500ms nach Empfang eines veränderten "WindrichtungEin" Signals durch das Senden des entsprechenden Signals am Ausgang "MotorAnsteuerungAus" einleiten.
A-CTRL-032	Pause nach Drehung im normalen Betrieb	Im normalen Betriebsmodus soll der Regler während 5s nach Beendigung der Gondelbewegung keine neue Gondelbewegung initiieren (der 3-Sekunden-laufende-Durchschnitt des "WindrichtungEin" Signals soll weiterhin gebildet werden).
A-CTRL-034	Verhalten, wenn Gondel klemmt	Wenn die Differenz zwischen aktueller und Zielposition sich innerhalb von 3 Sekunden im normalen oder manuellen Betrieb nicht ändert und die Differenz größer als 1 Grad beträgt, soll der Azimuth-Regler in den Fehlermodus wechseln.
A-CTRL-035	Wechsel aus dem Fehlermodus	Im Fehlermodus soll der Regler in keinen anderen Betriebsmodus wechseln. Das WNS kann nur durch ein Reset über die Schnittstelle "Reset" mit dem Signal "Reset AN" aus diesen Modus raus. "Reset AN" bedeutet, dass am Eingang Reset für kurze Zeit 1 V anliegt.
A-CTRL-037	Temperaturbereich	Der Azimuth-Regler soll die spezifizierte Leistung erreichen im einem operativen Temperaturbereich von -30 bis + 40 Grad Celsius.

A-CTRL-038	Kein Motorsteuerungsbefehl, wenn die Bremsen noch angezogen sind	Solange an der Schnittstelle “BremsenEin“ das Signal “BremsenAngezogen“ anliegt, legt der Regler keine Spannung an die Schnittstelle “MotorAnsteuerungAus“ an
A-CTRL-039	Positionssensor spielt Verrückt	Falls an der Schnittstelle “MotorAnsteuerungAus“ keine Spannung anliegt und der Wert an der Schnittstelle “AktuellePositionEin“ sich länger als1 Sekunde ändert, soll der Regler in den Fehlermodus wechseln.
A-CTRL-040	Motor "Aus" bei Sicherheitsabschaltung	Im Betriebsmodus "Sicherheitsabschaltung" soll der Azimuth-Regler über die Schnittstelle "MotorAnsteuerungAus" das Signal "AUS" senden. Das Signal "AUS" ist mit 0 V definiert
A-CTRL-041	Eingang "Reset"	Der Azimuth-Regler soll einen analogen Signaleingang "Reset" haben. Das Signal am Eingang "Reset" liegt innerhalb eines Spannungsbereichs von 0-1 V, wobei 0,5 bis 1 V bedeutet, dass der Regler aus dem Fehlermodus genommen wird.
A-CTRL-042	Eingang "BremsenAngezogenEin"	Der Azimuth-Regler soll einen analogen Signaleingang "BremsenAngezogenEin" haben. Das Signal am Eingang "BremsenAngezogenEin" variiert innerhalb eines Spannungsbereichs 0-1 V. Ein Signal von 0 bis 0.8 V sagt aus, dass die Bremsen angezogen und über 0.8 V das die Bremsen gelöst sind.
A-CTRL-043	Eingang "DrehzahlEin"	Der Azimuth-Regler soll einen analogen Signaleingang "DrehzahlEin" haben, welcher die Drehzahl des Azimutmotors in einem Spannungsbereich von 0 bis 1 V abdeckt. Der Spannungsbereich ist linear auf 0 bis 20 Umderhungen pro Minute abgebildet.
A-CTRL-044	Ausgang "Fehler"	Der Azimuth-Regler soll einen analogen Signalausgang "Fehler" haben. Das Signal am Ausgang "Fehler" soll zwischen 0 und 1 V liegen. Wenn der Regler in den Fehlermodus wechselt soll ein Signal größer 0.5 V angelegt werden
A-CTRL-045	Wechsel zu "WindgeschwindigkeitZuLangsam"	Falls die Windgeschwindikeit unter 4 m/s liegt und der Regler sich weder im Fehlermodus noch "Sicherheitsabschaltungs"-Modus befindet, geht der Regler in den "WindgeschwindigkeitZuLangsam"-Modus.
A-CTRL-046	"WindgeschwindigkeitZuLangsam" Betriebsmodus	Befindet sich der Regler in diesem Modus, sendet der Regler an dem Ausgang "BremsenAus" kontinuierlich das Signal "BremsenAnziehen". Weiterhin wird am Ausgang "MotorAnsteuerungAus" das Signal "Aus" gesendet.
A-CTRL-047	Wechsel von "WindgeschwindigkeitZuLangsam" zu normalen Betrieb	Wenn am Eingang "WindgeschwindigkeitEin" das Signal "Betrieb" und am Eingang "ManuellerBetrieb" das Signal "Aus" anliegt, dann wechselt der Regler in den normalen Betrieb.
A-CTRL-048	Wechsel von "WindgeschwindigkeitZuLangsam" zu manuellem Betrieb	Wenn am Eingang "WindgeschwindigkeitEin" das Signal "Betrieb" und am Eingang "ManuellerBetrieb" das Signal "An" anliegt, dann wechselt der Regler in den manuellen Betrieb.
A-CTRL-049	Wechsel aus dem normalen Betriebsmodus in den Sturmwarnung-Modus	Wenn das System im normalen Betriebsmodus befindet und am Eingang "WindgeschwindigkeitEin" ein Wert für eine Windstärke größer 25m/s anliegt, wechselt das System in den Sturmwarnung-Modus
A-CTRL-050	"Sturmwarnung" Betriebsmodus	Befindet sich der Azimuth-Regler im Sturmwarnung-Betriebsmodus, soll der Azimuth-Regler auf dem kürzesten Wege die Gondel in 90 Grad zur Position drehen, die durch das Signal an der Schnittstelle "WindrichtungEin" vorgegeben wird.

B. Messina-Programmcode für einen Testfall

```
1  import signalBase.AllSignals;
2  import signalBase.params;
3  import java.io.IOException;

5  import com.berner_mattner.messina.executer.vmaccess.ITargetLogger;
6  import com.berner_mattner.messina.vm.testbase.TestCase.
       TestFailedException;



11 public class NormalerBetrieb extends AllSignals {

13   enum States {
14     IDLE, BREMSENLOESEN, BREMSENGELOEST_IDLE, REGELN, FEHLER
15   }
16   private static boolean errorOccured;
17   private static int stateActiveTime;
18   private static double ca = 0.0010;
19   private static int timerStep = 5;
20   private static int stimuliCounter = 0;
21   private static int relativeTimer = 0;
22   private static boolean firstEntry = true;
23   private static int stateHoldTime = 0;

25   States states = States.IDLE;
26   States nextState = states;

28 // Value - Mappings
       ------------------------------------------------------------------------



31 private Double mapping_Drehzahl(String interval){
32   Double tmp = 0.0;
33   if (interval.equals("Null_Drehzahl")) {
34     tmp = 0.0;
35   }   else if(interval.equals("Betrieb_Drehzahl")) {
36     tmp = 0.5;
37   }   return tmp;

39 }

41 private Double mapping_BremsenBefehl(String interval){
42   Double tmp = 0.0;
43   if (interval.equals("Loesen")) {
44     tmp = 7.0;
45   }   else if(interval.equals("Anziehen")) {
46     tmp = 1.0;
47   }   return tmp;

49 }

51 private Double mapping_Fehlerzustand(String interval){
52   Double tmp = 0.0;
53   if (interval.equals("Aus_Fehlerzustand")) {
```

```
54      tmp = 0.0;
55    }   else if(interval.equals("An_Fehlerzustand")) {
56      tmp = 0.5;
57    }   return tmp;

59  }

61  private Double mapping_Windgeschwindigkeit(String interval){
62    Double tmp = 0.0;
63    if (interval.equals("Betrieb_Windgeschwindigkeit")) {
64      tmp = 5.0;
65    }   else if(interval.equals("Zu_Schnell")) {
66      tmp = 25.0;
67    }   else if(interval.equals("Zu_langsam")) {
68      tmp = 0.25;
69    }   return tmp;

71  }

73  private Double mapping_BremsenSensor(String interval){
74    Double tmp = 0.0;
75    if (interval.equals("Angezogen")) {
76      tmp = 0.4;
77    }   else if(interval.equals("Geloest")) {
78      tmp = 0.901;
79    }   return tmp;

81  }

83  private Double mapping_AktuellePosition(String interval){
84    Double tmp = 0.0;
85    if (interval.equals("D_AktuellePosition")) {
86      tmp = 292.501;
87    }   else if(interval.equals("F_AktuellePosition")) {
88      tmp = 22.5;
89    }   else if(interval.equals("B_AktuellePosition")) {
90      tmp = 135.0;
91    }   else if(interval.equals("E_AktuellePosition")) {
92      tmp = 337.5;
93    }   else if(interval.equals("C_AktuellePosition")) {
94      tmp = 225.0;
95    }   else if(interval.equals("A_AktuellePosition")) {
96      tmp = 67.5;
97    }   return tmp;

99  }

101 private Double mapping_Sicherheitsabschaltung(String interval){
102   Double tmp = 0.0;
103   if (interval.equals("An_Sicherheitsabschaltung")) {
104     tmp = 0.5;
105   }   else if(interval.equals("Aus_Sicherheitsabschaltung")) {
106     tmp = 4.0;
107   }   return tmp;

109 }

111 private Double mapping_ManuellePosition(String interval){
```

```
112   Double tmp = 0.0;
113   if (interval.equals("KleinerAbstand_MP")) {
114     tmp = 0.0;
115   }   else if(interval.equals("UnzureichenderAbstand_MP")) {
116     tmp = 0.0;
117   }   else if(interval.equals("GrosserAbstand_MP")) {
118     tmp = 0.0;
119   }   return tmp;

121 }

123 private Double mapping_Motorsteuerungsbefehl(String interval){
124   Double tmp = 0.0;
125   if (interval.equals("Links")) {
126     tmp = -6.0;
127   }   else if(interval.equals("Rechts")) {
128     tmp = 6.001;
129   }   else if(interval.equals("Aus_Motorsteuerungsbefehl")) {
130     tmp = 0.0;
131   }   return tmp;

133 }

135 private Double mapping_ManuellerBetrieb(String interval){
136   Double tmp = 0.0;
137   if (interval.equals("An_ManuellerBetrieb")) {
138     tmp = 1.5;
139   }   else if(interval.equals("Aus_ManuellerBetrieb")) {
140     tmp = 5.0;
141   }   return tmp;

143 }

145 private Double mapping_Windrichtung(String interval){
146   Double tmp = 0.0;
147   if (interval.equals("GrosserAbstand_WR")) {
148     tmp = mapping_AktuellePosition(params.AktuellePosition)+20.0;
149   }   else if(interval.equals("UnzureichenderAbstand_WR")) {
150     tmp = mapping_AktuellePosition(params.AktuellePosition)+0.01;
151   }   else if(interval.equals("KleinerAbstand_WR")) {
152     tmp = mapping_AktuellePosition(params.AktuellePosition)+2.0;
153   }   return tmp;

155 }

157 private Double mapping_Reset(String interval){
158   Double tmp = 0.0;
159   if (interval.equals("Aus_Reset")) {
160     tmp = 0.25;
161   }   else if(interval.equals("An_Reset")) {
162     tmp = 0.75;
163   }   return tmp;

165 }
```

```
170 // Muster
    ------------------------------------------------------------------------

171   private void setValue_IntegratorMockup_IC(double value) throws
          InterruptedException{
172     IntegratorMockup_IC.setValue(value);
173     WinkelsensorResetEin.setValue(1);
174     sleep(10);
175     WinkelsensorResetEin.setValue(0);
176   }

178   private void winkelSonsorSpinnt(){
179     // TODO please implement the Operation
180   }
181   private void gondelKlemmt(){
182     // TODO please implement the Operation
183   }
184   private boolean regeln_DoActivity(double inAktuellePosition, double
          inWindrichtungEin, double inMotorsteuerungAus) {
185     if (inAktuellePosition < inWindrichtungEin && inMotorsteuerungAus
            >= 0.001 && inMotorsteuerungAus <= 12.0){
186       return true;
187     } else if (inAktuellePosition > inWindrichtungEin &&
            inMotorsteuerungAus >= -12 && inMotorsteuerungAus <= -0.001){
188       return true;
189     } else {
190       return false;
191     }
192   }
193   private boolean changeEvent_regeln_abgeschlossen(double
          inAktuellePosition, double inWindrichtungEin) {
194     if ((inWindrichtungEin >= 45 && inWindrichtungEin <= 315 && abs(
            inAktuellePosition - inWindrichtungEin) < 0.1) ||
            inWindrichtungEin >= 0 && inWindrichtungEin <= 44.999 &&
            inAktuellePosition == 45 || inWindrichtungEin >= 315.001 &&
            inWindrichtungEin <= 359.999 && inAktuellePosition == 315) {
195       return true;
196     } else {
197       return false;
198     }

200   }
201   private double abs(double d) {
202     if (d < 0){
203       return d*-1;
204     } else {
205       return d;
206     }
207   }


210   @Override
211   public boolean preCondition() throws IOException,
          TestFailedException,
212       InterruptedException {
```

```
215       setValue_IntegratorMockup_IC(mapping_AktuellePosition(params.
              AktuellePosition));
216       SicherheitsabschaltungEin.setValue(mapping_Sicherheitsabschaltung(
              params.Sicherheitsabschaltung));
217       WindgeschwindigkeitEin.setValue(mapping_Windgeschwindigkeit(params
              .Windgeschwindigkeit));
218       WindrichtungEin.setValue(mapping_Windrichtung(params.
              AktuellePosition));
219       ManuellerBetriebEin.setValue(mapping_ManuellerBetrieb(params.
              ManuellerBetrieb));
220       Reset.setValue(mapping_Reset("An_Reset"));
221       sleep(10);
222       Reset.setValue(mapping_Reset("Aus_Reset"));

224       // register for lokal values
225       MotorsteuerungAus.register();
226       Fehler.register();
227       BremsenAus.register();
228       AktuellePosition.register();
229       WindrichtungEin.register();
230       BremsenEin.register();

232       return super.preCondition();
233     }

235     public int run() throws IOException, TestFailedException,
236         InterruptedException {

238       startTimer(timerStep);
239       return errorOccured?1:0; //0 means PASSED
240   //    WindrichtungEin.setValue(20);
241   //    return 0;

243     }

245     @Override
246     protected boolean timer(int tick, long late) throws IOException,
247         TestFailedException, InterruptedException {

249       double lokalMotorsteuerungsbefehlAus = MotorsteuerungAus.
              getLocalValue();
250       double lokalFehlerAus = Fehler.getLocalValue();
251       double lokalBremsenAnsteuerungAus = BremsenAus.getLocalValue();
252       double lokalAktuellePositionEin = AktuellePosition.getLocalValue()
              ;
253       double lokalWindrichtungEin = WindrichtungEin.getLocalValue();
254       double lokalBremsenAngezogenEin = BremsenEin.getLocalValue();
255       double lokalReset = Reset.getLocalValue();

257       boolean regeln_abgeschlossen = false;
258       /*
259       double BremsenAusLocal = BremsenAus.getLocalValue();
260       double AktuellePositionLocal = AktuellePosition.getLocalValue();
261       double DrehzahlLocal = Drehzahl.getLocalValue();
262       double MotorsteuerungAusLocal = MotorsteuerungAus.getLocalValue();
263       double BremsenEinLocal = BremsenEin.getLocalValue();
264       */
```

```
266     // ---------------- Stimuli generated from the Signal Generator
            Activity Diagram -----------

268     if(stimuliCounter == 0){
269 if(params.Fehlergenerierung == "PositionsSensorFehler"){stimuliCounter
        = 1;}
270 else {stimuliCounter = 3;}
271 }
272 if(stimuliCounter == 1){
273 winkelSonsorSpinnt();
274 stimuliCounter = 2;
275 }
276 if(stimuliCounter == 2){
277 stimuliCounter = -1;
278 }
279 if(stimuliCounter == 3){
280 WindrichtungEin.setValue(mapping_Windrichtung(params.Windrichtung));
281 stimuliCounter = 4;
282 }
283 if(stimuliCounter == 4){
284 if(params.Fehlergenerierung == "MotorVerhaltenFehler"){stimuliCounter
        = 5;}
285 else {stimuliCounter = 8;}
286 }
287 if(stimuliCounter == 5){
288 if((lokalBremsenAngezogenEin >= 0.801 && lokalBremsenAngezogenEin <=
        1)) {
289 stimuliCounter = 6;
290 }
291 }
292 if(stimuliCounter == 6){
293 gondelKlemmt();
294 stimuliCounter = 7;
295 }
296 if(stimuliCounter == 7){
297 stimuliCounter = -1;
298 }
299 if(stimuliCounter == 8){
300 if((lokalBremsenAnsteuerungAus >= 2 && lokalBremsenAnsteuerungAus <=
        12)) {
301 stimuliCounter = 9;
302 }
303 }
304 if(stimuliCounter == 9){
305 if(params.AnzahlRegelungen == "NullRegelungen"){stimuliCounter = 10;}
306 if(params.AnzahlRegelungen == "EineRegelung"){stimuliCounter = 12;}
307 if(params.AnzahlRegelungen == "ZweiRegelungen"){stimuliCounter = 13;}
308 }
309 if(stimuliCounter == 10){
310 WindrichtungEin.setValue(mapping_Windrichtung("
        UnzureichenderAbstand_WR"));
311 stimuliCounter = 11;
312 }
313 if(stimuliCounter == 11){
314 stimuliCounter = -1;
315 }
316 if(stimuliCounter == 12){
317 stimuliCounter = -1;
```

```
318 }
319 if(stimuliCounter == 13){
320 if((lokalBremsenAngezogenEin >= 0.801 && lokalBremsenAngezogenEin <=
        1)) {
321 stimuliCounter = 14;
322 }
323 }
324 if(stimuliCounter == 14){
325 relativeTimer = 0;
326 stimuliCounter = 15;
327 }
328 if(stimuliCounter == 15){
329 if(relativeTimer >= 500) {
330 stimuliCounter = 16;
331 }
332 relativeTimer += timerStep;
333 }
334 if(stimuliCounter == 16){
335 if((lokalMotorsteuerungsbefehlAus >= 0 &&
        lokalMotorsteuerungsbefehlAus <= 0)) {
336 stimuliCounter = 17;
337 }
338 }
339 if(stimuliCounter == 17){
340 WindrichtungEin.setValue((lokalAktuellePositionEin+90)%360);
341 stimuliCounter = 18;
342 }
343 if(stimuliCounter == 18){
344 stimuliCounter = -1;
345 }


348     // ---------------- State specific asserts
            --------------------------


350   states = nextState;
351   switch (states){
352   case IDLE :
353   if(firstEntry){
354     firstEntry = false;
355     stateActiveTime = 0;
356     stateHoldTime = 0;
357   }
358   if(!(lokalMotorsteuerungsbefehlAus == 0 && lokalFehlerAus == 0 &&
        lokalBremsenAnsteuerungAus > 0 && lokalBremsenAnsteuerungAus <
        1.999) && stateActiveTime > 20){
359     errorOccured = true; // hier kann noch eine returnMsg kommen
360     resultString += "Fehler in IDLE";
361   }

363   if (lokalMotorsteuerungsbefehlAus == 0 && lokalFehlerAus == 0 &&
        lokalBremsenAnsteuerungAus > 0 && lokalBremsenAnsteuerungAus <
        1.999) {
364     stateHoldTime += timerStep;
365     if(stateActiveTime > 3000)
366       return false;
367   }
368   if(abs(lokalWindrichtungEin - lokalAktuellePositionEin) > 0.1){
```

```
369       nextState = States.BREMSENLOESEN;
370       firstEntry = true;
371     }if((lokalMotorsteuerungsbefehlAus > -12 &&
            lokalMotorsteuerungsbefehlAus < -0.001 ||
            lokalMotorsteuerungsbefehlAus > 0.001 &&
            lokalMotorsteuerungsbefehlAus < 12)){
372       nextState = States.FEHLER;
373       firstEntry = true;
374     }
375     break;
376     case BREMSENLOESEN :
377     if(firstEntry){
378       firstEntry = false;
379       stateActiveTime = 0;
380       stateHoldTime = 0;
381     }
382     if(!(lokalBremsenAnsteuerungAus > 2 && lokalBremsenAnsteuerungAus <
            12 && lokalFehlerAus == 0 && lokalMotorsteuerungsbefehlAus == 0)
            && stateActiveTime > 20000){
383       errorOccured = true; // hier kann noch eine returnMsg kommen
384       resultString += "Fehler in BREMSENLOESEN";
385     }

387     if (lokalBremsenAnsteuerungAus > 2 && lokalBremsenAnsteuerungAus <
            12 && lokalFehlerAus == 0 && lokalMotorsteuerungsbefehlAus == 0)
            {
388       stateHoldTime += timerStep;
389       if(stateActiveTime > 3000)
390         return false;
391     }
392     if((lokalBremsenAngezogenEin > 0.801 && lokalBremsenAngezogenEin <=
            1)){
393       nextState = States.BREMSENGELOEST_IDLE;
394       firstEntry = true;
395     }if((lokalMotorsteuerungsbefehlAus > -12 &&
            lokalMotorsteuerungsbefehlAus < -0.001 ||
            lokalMotorsteuerungsbefehlAus > 0.001 &&
            lokalMotorsteuerungsbefehlAus < 12)){
396       nextState = States.FEHLER;
397       firstEntry = true;
398     }
399     break;
400     case BREMSENGELOEST_IDLE :
401     if(firstEntry){
402       firstEntry = false;
403       stateActiveTime = 0;
404       stateHoldTime = 0;
405     }
406     if(!(lokalBremsenAnsteuerungAus > 2 && lokalBremsenAnsteuerungAus <
            12 && lokalFehlerAus == 0 && lokalMotorsteuerungsbefehlAus == 0)
            && stateActiveTime > 20){
407       errorOccured = true; // hier kann noch eine returnMsg kommen
408       resultString += "Fehler in BREMSENGELOEST_IDLE";
409     }

411     if (lokalBremsenAnsteuerungAus > 2 && lokalBremsenAnsteuerungAus <
            12 && lokalFehlerAus == 0 && lokalMotorsteuerungsbefehlAus == 0)
            {
```

```
412       stateHoldTime += timerStep;
413       if(stateActiveTime > 3000)
414         return false;
415     }
416     if(abs(lokalWindrichtungEin - lokalAktuellePositionEin) > 0.1){
417       nextState = States.REGELN;
418       firstEntry = true;
419     }if((stateActiveTime == 1000)){
420       nextState = States.IDLE;
421       firstEntry = true;
422     }
423     break;
424     case REGELN :
425     if(firstEntry){
426       firstEntry = false;
427       stateActiveTime = 0;
428       stateHoldTime = 0;
429     }
430     boolean checkBool = regeln_DoActivity(lokalAktuellePositionEin,
          lokalWindrichtungEin, lokalMotorsteuerungsbefehlAus);
431     if(!checkBool && stateActiveTime > 200){
432       errorOccured = true; // hier kann noch eine returnMsg kommen
433       resultString += "Fehler in REGELN";
434     }

436     if (checkBool) {
437       stateHoldTime += timerStep;
438       if(stateActiveTime > 3000)
439         return false;
440       regeln_abgeschlossen = changeEvent_regeln_abgeschlossen(
            lokalAktuellePositionEin, lokalWindrichtungEin);
441     }
442     if(regeln_abgeschlossen){
443       nextState = States.BREMSENGELOEST_IDLE;
444       firstEntry = true;
445     }
446     break;
447     case FEHLER :
448     if(firstEntry){
449       firstEntry = false;
450       stateActiveTime = 0;
451       stateHoldTime = 0;
452     }
453     if(!(lokalFehlerAus > 0.001 && lokalFehlerAus < 1 &&
          lokalBremsenAnsteuerungAus > 0 && lokalBremsenAnsteuerungAus <
          1.999 && lokalMotorsteuerungsbefehlAus == 0) && stateActiveTime >
           10){
454       errorOccured = true; // hier kann noch eine returnMsg kommen
455       resultString += "Fehler in FEHLER";
456     }

458     if (lokalFehlerAus > 0.001 && lokalFehlerAus < 1 &&
          lokalBremsenAnsteuerungAus > 0 && lokalBremsenAnsteuerungAus <
          1.999 && lokalMotorsteuerungsbefehlAus == 0) {
459       stateHoldTime += timerStep;
460       if(stateActiveTime > 3000)
461         return false;
462     }
```

```
463     if((lokalReset > 0.5 && lokalReset < 1)){
464       nextState = States.IDLE;
465       firstEntry = true;
466     }
467     break;

469     default:
470     break;
471   }

473       // ---------------- Global asserts
              ----------------------------------


476       if (errorOccured){
477         resultString += " - Error occured at Tick="+tick;
478         return false;
479       }

481       stateActiveTime += timerStep;

483       return true;
484     }

486     @Override
487     public boolean postCondition() throws IOException,
            TestFailedException,
488           InterruptedException {

490       Reset.setValue(mapping_Reset("An_Reset"));
491       sleep(10);
492       Reset.setValue(mapping_Reset("Aus_Reset"));

494       return super.postCondition();
495     }

497   }
```

Listing 11: Abbildung eines Zustandes in Java

*Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 25. Februar 2011	
Ort, Datum	Unterschrift