HAW HAMBURG

# Bachelorarbeit

Daniel Nisch

## Modellbasierte Testfallgenerierung und Ausführung für das Testen eines Echtzeitsystems auf Konformität

Betreuung durch: Prof. Dr. Bettina Buth / Prof. Dr. Bernd Kahlbrandt
Eingereicht am: 11. Oktober 2019

*Fakultät Technik und Informatik*
*Department Informatik*

*Faculty of Computer Science and Engineering*
*Department Computer Science*

**Daniel Nisch**

**Thema der Arbeit**
Modellbasierte Testfallgenerierung und Ausführung für das Testen eines Echtzeitsystems auf Konformität

**Stichworte**
MBT, Model-Based-Testing, Testautomatisierung, Testfallgenerierung, Statecharts, FSM, Anforderungsbasiertes Testen, Anforderungsmodellierung

**Kurzzusammenfassung**
Das Testen einer Anwendung stellt im Softwareentwicklungsprozess einen hohen Aufwand dar. Ein Ansatz, diesen Aufwand zu reduzieren, kann das Model-Based-Testing bieten. In dieser Arbeit wird ein möglicher Ansatz für dieses Vorgehen am Beispiel der Entwicklung eines Steuerungssystems für eine Sortieranlage beschrieben mit dem Ziel, die Anforderungen an das Gesamtsystem an der Implementation zu testen. Dafür wird im ersten Schritt ein Modell der Anforderungen entwickelt. Anschließend wird ein Algorithmus gewählt, der aus dem Modell Testfälle generiert, und zuletzt eine Anwendung entwickelt, die diese Testfälle automatisiert gegen die Implementation ausführt.

**Title of Thesis**
Model-based test case generation and execution for testing a real-time system for compliance

**Keywords**
MBT, model-based testing, test automation, test case generation, statecharts, FSM, requirements-based testing, requirements modeling

**Abstract**
The testing of an application represents a high effort in the software development process. Model-based testing can offer an approach to reduce this effort. This paper describes a possible approach for this procedure using the example of the development of a control system for a sorting plant with the aim of testing the implementation requirements of the overall system. The first step is to develop a model of the requirements. Then an algorithm is chosen, which generates test cases from the model, and finally an application is developed, which executes these test cases automatically against the implementation.

# Inhaltsverzeichnis

# 1 Einleitung

Software nimmt in unserem Leben und unserer Versorgung eine große Rolle ein. Wenn es in den Anwendungen, die für unsere Unterhaltung zuständig sind, zu einen Fehlerzustand kommt, empfindet das der Nutzer häufig als unzumutbar. Wenn aber sicherheitskritische Anwendungen wie sie in Autos, Flugzeugen, Förderbändern in der Industrieproduktion oder Herzschrittmachern - um ein paar zu nennen - vorkommen einen Fehler zeigen, kann dies zu Sach- und Personenschaden führen. Da Softwarefehler sowohl beim Entwicklungsprozess als auch bei dem restlichen Lebenszyklus immer wieder auftreten können, muss Software umfangreich getestet werden [6]. Testen im Entwicklungsprozess nimmt je nach Branche und Anforderungen unterschiedliche Dimensionen an. Mit einem Anteil von etwa 35% an den Gesamtkosten (diese Angabe schwankt je nach Quelle sehr stark) ist Testen ein großer Kostenpunkt für die Softwareentwicklung, mit steigender Tendenz [9].

Mit steigender Komplexität der Software sind formale Modellierungstechniken schon einige Zeit fester Bestandteil des Softwareengineering [21]. Die Modelle aus der Entwicklung können u.U. direkt zur Testfallentwicklung herangezogen werden, alternativ werden auch spezielle Modelle für das Testen entwickelt. Die Modellierung soll häufig auch ein Ansatzpunkt für automatisierte Generierung und Ausführung von Testfällen sein. Die automatisierte Generierung der Testfälle kann in einigen Fällen die Entwicklungskosten reduzieren ohne die Testabdeckung zu verringern [26]. Die Notwendigkeit dafür zeigt sich im Wachstum des Marktes für Werkzeuge zur Testautomatisierung im Bereich von embedded Systems, der vom Jahr 2003 bis 2004 um 20% gewachsen ist [12].

## 1.1 Problemstellung

Im Softwareentwicklungsprozess müssen einige Entscheidungen, wie die Fragen *mit welchen Tools wird das System modelliert?* und *wie soll die Software getestet werden?* früh getroffen werden. Von diesen Fragestellungen hängt der weitere Entwicklungsprozess maßgeblich ab. Wird die Entscheidung getroffen, die Modelle zur Testautomatisierung zu nutzen, müssen die Modelle anderen Qualitätsanforderungen genügen, als wenn sie nur zum Veranschaulichen und zur Planung genutzt werden. Zusätzlich müssen zur Testautomatisierung passende Werkzeuge gesucht und eingerichtet oder entwickelt werden. Wie diese Schritte zum modellbasierten Testen aussehen können, wird in dieser Arbeit am Beispiel der Entwicklung eines Steuerungssystems für eine Sortieranlage veranschaulicht.

Im Praktikum Embedded System Engineering (ESEP), einer Veranstaltung, die im vierten Semester des Studiengangs Technische Informatik an der HAW Hamburg stattfindet, durchlaufen die Studierenden einen möglichen Ablauf des Softwareentwicklungsprozesses. Sie haben die Aufgabe, die Steuerung einer Sortieranlage nach unterspezifizierten Anforderungen zu planen und zu entwickeln. Um das System gegen die Anforderungen unabhängig von der Implementierung zu testen, muss das System zur Modellierung der Testfälle auf der Schnittstellenebene der Sortieranlage betrachtet werden. Um dies mit modellbasierten Testmethoden zu ermöglichen, muss ein Modell der Anforderungen, abgebildet durch die Elemente der Schnittstelle, entwickelt werden. Für dieses Modell muss ein Algorithmus gefunden werden, der Testfälle aus diesem Modell generiert. Der nächste Schritt ist, eine Testumgebung zu entwickeln, die die Testfälle ausführt und gegen den Zustand der Schnittstelle prüft.

Das Ziel dieser Arbeit ist es, eine Testumgebung zu entwickeln, um die Implementation eines Echtzeitsystems automatisiert auf der Basis von Modellen auf Konformität bezüglich der Anforderung zu testen.

## 1.2 Aufbau der Arbeit

Zu Beginn wird im Grundlagenkapitel die Sortieranlage und eine Simulation dieser Anlage vorgestellt, um eine Grundlage für die Modellierung zu schaffen. Insbesondere wird die Schnittstelle des Systems und der Simulation betrachtet. Sie bietet uns die Möglichkeit, den Systemzustand in Erfahrung zu bringen und das System zu steuern. Anschließend wird der Bereich des Testens, mit dem sich diese Arbeit befasst, erläutert, um einen Kontext zu schaffen. Dieser Teil des Grundlagenkapitels dient auch zur Vorbereitung auf die später vorgestellte Testfallerzeugung und den Testablauf. Am Ende des Grundlagenkapitels befindet sich eine einführende Erklärung zu der verwendeten Modellierungsart 'Statecharts' und der Form, in der sie vorliegt.

Der nächste Schritt ist eine kurze Analyse der Anforderungen, die von dem entwickelten Modell abgedeckt werden sollen. Das nächste Kapitel erklärt das Vorgehen bei der Modellierung der Anforderungen in eine Zustandsmaschine und betrachtet das Ergebnis dieses Arbeitsschrittes. Nachdem erläutert wurde, wie das entwickelte Modell funktioniert und aussieht, werden die Entscheidungen über die Wahl der Kriterien für die Testfälle diskutiert und der gewählte Algorithmus vorgestellt.

Mit dem Wissen um die betrachteten Systeme und um die Testfällen wird die entwickelte Architektur des Systems zur Testausführung vorgestellt. Zum Schluss dieser Arbeit wird das Ergebnis der Entwicklungsarbeit validiert und Probleme, die während der Entwicklung aufgetreten sind, diskutiert.

# 2 Grundlagen

In diesem Kapitel werden die Grundlagen zu dem System, für das das Anwendungsbeispiel umgesetzt wird, sowie die verwendeten Techniken, Werkzeuge und Programme erläutert, die im Rahmen dieser Arbeit verwendet werden. In den Abschnitten 2.1 bis 2.3 wird das eingebettete System, an dessen Beispiel die Umsetzung gezeigt wird, behandelt. Im Abschnitt 2.4 wird ein Einblick in den Bereich Model Based Testing und die Einordnung dieser Arbeit in diesen Bereich gegeben. Die letzten beiden Abschnitte des Grundlagenkapitels behandeln die Auswahl und Funktion des verwendeten Modellierungstools.

## 2.1 Das Festo-Transfersystem

Die Firma FESTO bietet mit dem MPS® Transfersystem ein modulares System für Ausbildungszwecke [13]. Der Zweck dieser Produktreihe ist es, eine Ausbildung mit Komponenten zu ermöglichen, die den in der Industrie eingesetzten entsprechen, und damit die Ausbildungsbedingungen so praxisnah wie möglich zu gestalten. Die von der HAW Hamburg eingesetzen Transfersysteme entsprechen der in Abbildung 1 dargestellten Konfiguration.

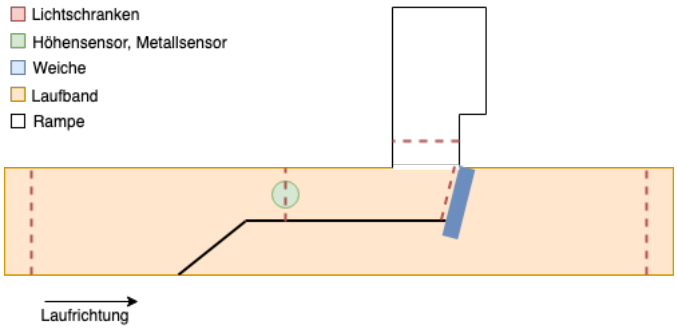


Abbildung 1: Aufbau des genutzten Transfersystems

## 2.2 Die HiL-Simulation des Festo-Transfersystem

Eine HiL-Simulation (Hardware-in-the-Loop Simulation) ist eine virtuelle Echtzeitumgebung, die physische Komponenten wie Sensoren und Aktoren simuliert [24]. HiL-Simulationen werden in vielen Bereichen wie der Automobilindustrie oder der Industrie-

automatisierung für die Entwicklung und den Test von Software eingesetzt. Der Einsatz solcher Simulationen ermöglicht es, während der Entwicklung auf teure Hardware zu verzichten und Testvorgänge zu automatisieren. Ein weiterer Vorteil ist, dass bei Fehlern, die im Realsystem zu Schäden führen, in der Simulation kein physischer Schaden entstehen kann.

Für die von der HAW Hamburg genutzten Transferysteme wurde im Sommer 2018 von Thomas Drauschke im Rahmen seiner Bachelorthesis eine HiL-Simulation entwickelt [11], die in dieser Arbeit verwendet wird. Die Nutzung der Simulation bietet die Möglichkeit, automatisiert Werkstücke hinzuzufügen, zu entfernen und zu manipulieren. Dies bietet gegenüber der Nutzung des physischen Transfersystems den Vorteil, dass die Testfälle automatisiert ausgeführt werden können, ohne dass ein Nutzer die geforderten Aktionen für einen Testfall manuell ausführen muss. Zusätzlich stehen genaue Informationen zu den Werkstücken, wie ihre Position auf dem Laufband, zur Verfügung, Informationen, die das physische System nicht in maschinenlesbarer Form bietet.

Zur Kommunikation bietet die HiL-Simulation zwei Schnittstellen über TCP Sockets an. Über eine Verbindung gegen den Statusport wird periodisch der Zustand des Transfersystems versendet. Über eine Verbindung gegen den Controllerport kann das Transfersystem mit einem festen Befehlssatz gesteuert werden. Die Kommunikation findet über Nachrichten im JSON Format statt.

## 2.3 Die Schnittstelle der Simulation

Der von der Simulation übermittelte Systemzustand enthält den Zustand aller Sensoren (Tabelle 1), aller Aktoren (Tabelle 2), eine Liste aller Werkstücke, die sich auf dem Transfersystem befinden oder heruntergefallen sind (Tabelle 4), sowie zusätzliche Daten, das System oder die Simulation betreffend. Die Daten über die Werkstücke sind nur in der Simulation vorhanden und stehen damit der Implementierung nicht zur Verfügung. Jedes Werkstück hält die Daten über seine Position in den zwei Achsen x, y der Draufsicht, seinen Typ, die Orientierung (z-Achsen Richtung), einer Information, ob das Werkstück vom Transfersystem gefallen ist, und seiner ID.

Angesteuert werden können Aktoren über eine Nachricht mittels TCP. Die Nachricht besteht aus einem Service, der angesprochen wird, und der Payload, der die Aktion festlegt. Beispielsweise kann über den CONTROL_SERVICE und der Aktion BTN_START_PRESS Start gedrückt werden. Über den USER_SERVICE können Werkstücke auf das

Transfersystem gelegt werden, wobei Position, Typ und Orientierung gewählt werden können. Konnte das Werkstück nicht auf das System gelegt werden, wird mit einer Fehlernachricht geantwortet. Wurde das Werkstück hinzugefügt, wird die ID als Antwort zurück geschickt. Mit der ID eines Werkstücks lässt sich dieses in seiner Orientierung verändern oder vom System zu entfernen.

| Daten: Sensoren | | |
|---|---|---|
| Bezeichnung | Beschreibung | Typ |
| HEIGHT | Distanzsensors zur Höhenmessung | Double |
| METAL_SENSOR | Metalsensors | Boolean |
| LED_Q1_STATUS | Status LED Q1 | Boolean |
| LED_Q2_STATUS | Status LED Q2 | Boolean |
| LED_RESET_STATUS | Status LED reset | Boolean |
| LED_START_STATUS | Status LED start | Boolean |
| LB_BEGINNING | Lichtschranke am Anfang des Laufbandes | Boolean |
| LB_END | Lichtschranke am Ende des Laufbandes | Boolean |
| LB_HEIGHT_UNIT | Lichtschranke der Höhenmessung | Boolean |
| LB_RAMP | Lichtschranke der Rampe | Boolean |
| LB_SWITCH | Lichtschranke der Weiche | Boolean |
| LED_GREEN_STATUS | Lichtschranke der Rampe | Boolean |
| LED_RED_STATUS | Lichtschranke der Rampe | Boolean |
| LED_YELLOW_STATUS | Lichtschranke der Rampe | Boolean |

Tabelle 1: Sensoren der Simulation

| Daten: Aktoren | | |
|---|---|---|
| Bezeichnung | Beschreibung | Typ |
| SWITCH_STATUS | Weiche auf/zu | Boolean |
| BTN_ESTOP_STATUS | Notausschalter | Boolean |
| BTN_STOP_STATUS | Stop | Boolean |
| BTN_RESET_STATUS | Zurücksetzten | Boolean |
| BTN_START_STATUS | Start | Boolean |
| ENGINE_LEFT_STATUS | Laufrichtung Links | Boolean |
| ENGINE_RIGHT_STATUS | Laufrichtung Rechts | Boolean |
| ENGINE_SLOW_STATUS | Bandgeschwindigkeit | Boolean |
| ENGINE_STOP_STATUS | Bandstop | Boolean |

Tabelle 2: Aktoren der Simulation

| Daten: Werkstücke | | |
|---|---|---|
| Bezeichnung | Beschreibung | Typ |
| fellAtEnd | Ist das Werkstück auf dem Laufband | Boolean |
| id | Werkstück ID | Integer |
| orientation | Ausrichtung | Boolean |
| x | Position auf der X-Achse | Double |
| y | Position auf der Y-Achse | Double |
| type | Art des Werkstücks | String |

Tabelle 3: Werkstück Daten der Simulation

## 2.4 Model Based Testing (MBT)

Traditionell werden Testfälle vom Tester der Anwendung aus Modellen, Anforderungen, den Erfahrungen des Testers oder dem zu testenden Code manuell abgeleitet. Diese Testfälle werden anschließend manuell vom Tester ausgeführt oder es wird ein Testskript erstellt, um die Ausführung der Testfälle zu automatisieren. Beim Ansatz des modellbasierten Testens wird, wie beim traditionellen Testen, gegen das erwartete Verhalten geprüft. Die dabei verwendeten Modelle müssen in einer Spezifikationssprache modelliert sein, die formal und detailliert genug sein muss, damit ein Algorithmus sinnvolle Testfälle aus diesen Modell ableiten kann [3]. Die generierten Testfälle sollten anschließend automatisiert ausgeführt werden. Es kann also von einem vollständig automatisierten Testprozess gesprochen werden.

Das Vorgehen der automatischen Testableitung und automatischer Ausführung kann für große, komplexe Systeme die Kosten und die Zeit, die für die Entwicklung benötigt wird, stark senken bei gleichzeitiger Erhöhung der Testabdeckung und Qualität der Tests [8]. Änderungen an den verwendeten Modellen erfordern dabei weniger Aufwand, da die Testfälle automatisch aktualisiert werden können. Diesen Vorteilen steht ein großer Aufwand für die Modellierung geeigneter Modelle, die Entwicklung geeigneter Algorithmen zur Testableitung oder das Einbinden von geeigneten Werkzeugen sowie das Entwickeln oder Einbinden einer Umgebung, die für die Testausführung zuständig ist, entgegen.

### 2.4.1 MBT zum Testen gegen die Anforderungen

Anforderungsbasierte Tests werden genutzt, um zu prüfen, ob eine Implementierung mit den Anforderungen übereinstimmt [3]. Das Testende ist erreicht, wenn gezeigt wurde, dass alle Anforderungen von der Implementierung erfüllt sind [21]. Um dies mit dem

MBT Ansatz umzusetzen, kann ein eigenes Modell für die Anforderungen erstellt werden. Das Erstellen eines eigenen Modells für die Anforderungen bedeutet einen Mehraufwand für den Schritt der Modellierung, jedoch kann es zur Validierung der Architektur (Beschrieben in Kapitel 3.3.1) genutzt werden und bietet bei ausreichender Qualität eine saubere Darstellung der Anforderungen, die zur Generierung der Testfälle dient. Ein reines Anforderungsmodell ist implementationsunabhängig, sodass verschiedene Implementationen mit diesem Modell getestet werden können [8]. Bei dem in dieser Arbeit vorgestellten Vorgehen wird ein Modell der Anforderungen ohne Implementationsdetails zur Testfallerzeugung verwendet. Hierbei werden Tests entwickelt, ohne Annahmen über die interne Struktur des SUT (System Under Test) zu treffen. Daher handelt es sich um Blackbox-Testing, auch als funktionale Tests bezeichnet [16].

## 2.5 Harel Statecharts

Zustandsdiagramme werden genutzt, um das Verhalten eines Systems zu beschreiben und zu modellieren. Dabei handelt es sich um eine visuelle Darstellung endlicher Zustandsautomaten (FSM). Statecharts [17, 18], eine Art von Zustandsdiagrammen, wurden 1987 von David Harel entwickelt mit dem Fokus, große und komplexe reaktive Systeme designen zu können. Später führte die Unified Modeling Language (UML) eine Variante auf Basis der Harel Statecharts, bekannt als State Machine Diagram, ein [10].

Der Vorteil von Harel Statecharts ist, dass diese vollständig ausführbar sind und die bis dahin genutzten Zustandsdiagramme um Hierarchien, Nebenläufigkeit und Kommunikation erweitern.

Anforderungen an ein System können als Eingangssignale zu bestimmten Zuständen des Systems mit erwarteten Ausgaben definiert werden. Daher ist die Modellierung der Anforderungen in einer FSM Struktur wie Statecharts naheliegend [8].

## 2.6 SCXML

Zustandsdiagramme sind als grafische Spezifikationssprache definiert. Damit diese verarbeitet werden können, müssen sie in einer Form vorliegen, die maschinenlesbar ist. State Chart XML (SCXML) [28] ist eine vom W3C spezifizierte Zustandsmaschinensprache für allgemeine Zwecke. Sie basiert auf Harel Statecharts und nutzt die ebenfalls vom W3C spezifizierten Sprache CCXML [27], welche sowohl eine Zustandsmaschine als auch eine Ereignishandlungssyntax und Elemente zum Starten von Aufrufen bietet.

| Tag | Beschreibung |
|---|---|
| state | Definiert einen Zustand mit Bezeichnung im Attribut id |
| onentry | Eintrittsaktion, die beim Eintritt in den Zustand, der sein Parent ist, ausgeführt wird. |
| onexit | Austrittsaktionen, die beim Austritt aus den Zustand, der sein Parent ist, ausgeführt wird. |
| initial | Im Attribut target dieses Tags wird der Startzustand festgelegt. Dieses Tag wird sowohl dazu genutzt den Startzustand der Zustandsmaschine als auch den Startzustand in hierarchischen Zuständen festzulegen. |
| transition | Ein Zustandsübergang von dem Zustand, der sein Parent im XML ist, in einen Zustand, dessen id im Attribut target angegeben ist. Es werden Bedingungen, Events und Aktionen unterstützt. |
| final | Definiert einen Endzustand. |
| parallel | Zustände, die als direkten Parent das Tag parallel haben, werden nebenläufig ausgeführt. |
| datamodel | In diesem Tag können Variablen eingeführt werden, die im Modell genutzt werden. |
| assign | Ermöglicht das Setzen von Variablen bei Ausführung einer Transition, beim Eintritt oder beim Austritt in einen Zustand. |

Tabelle 4: Kurzübersicht einiger SCXML Tags

# 3 Modellierung der Anforderungen

Im Abschnitt 3.1 werden die Anforderungen an die Implementierung der Steuerung für das Festo-Transfersystem aufgelistet, auf die getestet werden soll. Abschnitt 3.2 behandelt die Anforderungen, die an das Anforderungsmodell gestellt werden, um Testfälle mit dem in dieser Arbeit vorgestellten Ansatz aus dem Modell generieren zu können. Anschließend wird in 3.3 die Wahl des Modellierungstools, sowie das Vorgehen beim Modellierungsprozess erläutert und zuletzt das aus diesem Prozess entstandene Modell erklärt.

## 3.1 Anforderungen an die Implementation

Die ursprünglichen Anforderungen der Praktikumsaufgabe von ESE an der HAW Hamburg sind auf einen Betrieb mit zwei Festo-Transfersystemen ausgelegt, die über eine serielle Schnittstelle miteinander verbunden sind. Dabei übernehmen die Systeme jeweils eigene Aufgaben bei der Sortierung der Werkstücke. In diesem Beispiel werden nur die Anforderungen an den Betrieb mit einem Festo-Transfersystem betrachtet. Für dieses Beispiel und für die Entwicklung eines Prototyps sind die Anforderungen so zusammengefasst, dass diese in einem Modell dargestellt werden können. Im Regelfall ist dies nicht möglich und es sind mehrere Modelle notwendig, um alle Anforderungen und Details darzustellen [20]. Die Anforderungen, die im folgenden aufgeführt sind, sollen vom Anforderungsmodell abgebildet werden.

1. Ausgangszustand

   Im Ausgangszustand ist das Band angehalten und es leuchtet keine der Signalleuchten. Wird in diesem Zustand die Start-Taste gedrückt, wechselt das Band in den Betriebszustand. Bei Betätigung der Stop-Taste wechselt das System in den Ausgangszustand, wenn kein Fehler vorliegt.

2. Fehlererkennung

   Wird ein Fehler erkannt, leuchtet die rote Signalleuchte. Die beiden anderen Signalleuchten sind ausgeschaltet. Das Laufband wird angehalten.

3. Fehlerbehebung

   Ist es zu einen Fehlerfall gekommen, wird über das Betätigen der Reset-Taste das Band in den Ausgangszustand zurückgesetzt.

4. Betrieb

   Im Betrieb leuchtet die grüne Signalleuchte. Ist kein Werkstück auf dem Laufband, läuft das Band nicht.

   a) Einlegen eines neuen Werkstücks

      Ein Werkstück wird nur verarbeitet, wenn es in der ersten Lichtschranke eingelegt wird.

   b) Unzulässiges Einlegen von Werkstücken

      Ein Werkstück, das zwischen der ersten und zweiten Lichtschranke eingelegt wird, muss als Fehler erkannt werden, wenn das Werkstück in die zweite Lichtschranke eintritt.

      Ein Werkstück, das zwischen der Weiche und der letzten Lichtschranke eingelegt wird, muss beim Eintritt in die letzte Lichtschranke als Fehler erkannt werden.

   c) Unzulässiges Entfernen von Werkstücken

      Ein zulässiges Werkstück, das zwischen der ersten und zweiten Lichtschranke entfernt wird, muss als Fehler erkannt werden, wenn das Werkstück in die zweite Lichtschranke eintreten sollte.

      Ein zulässiges Werkstück, das zwischen der Weiche und der letzten Lichtschranke entfernt wird, muss als Fehler erkannt werden, wenn das Werkstück in die letzte Lichtschranke eintreten sollte.

   d) Höhenmessung

      Während sich ein Werkstück in der Lichtschranke für die Höhenmessung befindet, läuft das Band mit langsamer Geschwindigkeit.

e) Sortierung

Werkstücke mit der Codierung Nr. 1, einem Bohrloch unten und flache Werkstücke werden auf die Rampe aussortiert.

Werkstücke mit der Codierung Nr. 2, einem Bohrloch oben oder einem Metallkern werden bis zum Ende des Laufbandes befördert.

f) Fehlerhafte Werkstücke

Werkstücke, die nicht in 4e behandelt wurden, werden bei der Erkennung in der Höhenmessung als Fehler erkannt.

g) Werkstücke in der letzten Lichtschranke

Ist ein Werkstück in der letzten Lichtschranke, wird das Band angehalten, bis das Werkstück entfernt wurde. Das Werkstück darf dabei nicht vom Band fallen.

h) Gefüllte Rampe

Ist die Rampe voll, wird das Laufband angehalten und die gelbe Signalleuchte angeschaltet. Die anderen Signalleuchten sind ausgeschaltet. Wird die Rampe leer geräumt und die Start-Taste betätigt, wird der Stop des Laufbandes aufgehoben, die gelbe Signalleuchte aus- und die grüne Signalleuchte angeschaltet.

Die Anforderung, dass die Lampe je nach Fehlerzustand in unterschiedlicher Frequenz blinkt, wird so vereinfacht, dass diese nur noch Rot leuchten muss. Der Grund hierfür ist, dass die Übertragung der Daten des Systems nur alle 40ms erfolgt und die Simulation mit einer höheren Geschwindigkeit abläuft, als es im physischen System der Fall ist. Dadurch ist die Messung der Frequenz des Blinkens zu fehleranfällig.

## 3.2 Anforderungen an das Modell

Das Anforderungsmodell wird für das MBT entwickelt. Daher muss das Modell, für das in dieser Arbeit vorgestellte Vorgehen, ein deterministisches Verhalten aufweisen, damit die später generierten Testfälle bei der Ausführung korrekte Ergebnisse liefern. Außerdem muss das Modell in einer formalen Spezifikationssprache modelliert sein, die detailliert genug sein muss, damit ein Algorithmus sinnvolle Testfälle aus diesem Modell ableiten

kann [3]. Dabei muss sichergestellt werden, dass sowohl das Anforderungsmodell als auch das System- und Verhaltensmodell, und damit die Implementierung, dieselben Anforderungen erfüllen.

Das Modell wird erstellt, um spezielle Testziele zu erreichen. Wenn das Modell diese Ziele nicht abbildet, verfehlt es seinen Zweck [20]. Anforderungen oder Teilanforderungen müssen mit allen notwendigen Information abgebildet sein, um die gewünschten Testfälle generieren zu können. Um die Testfallgenerierung mit einem einfachen Algorithmus zu ermöglichen, ohne dass es zu einer Explosion der Anzahl an erzeugten Testfälle kommt, muss das Modell einfach gehalten werden. Als Folge dessen empfiehlt es sich ein komplexes Modell in mehrere einfachere und spezialisierte Modelle aufzuteilen [20]. Für den Algorithmus in Kapitel 4 sollte das Modell keine Verzweigungen besitzen, die dazu führen, dass Bereiche, deren Elemente bereits komplett durch Testfälle abgedeckt sind, Sequenzen enthalten, die weitere relevante Testfälle darstellen.

Das Modell muss die Anforderungen, die von der Testfallgenerierung an das Modell gestellt werden, erfüllen [20]. Für den im Kapitel 4 beschriebenen Algorithmus betrifft dies die Modellierung nebenläufiger Zustände. Die parallelen Zustände werden bei der Testfallgenerierung getrennt betrachtet und sollten daher für einen Bereich oder ein Gruppe von Anforderungen zuständig sein. Diese getrennte Betrachtung führt auch dazu, dass parallele Zustände sich keine Variablen teilen dürfen, deren Wert nicht extern gesetzt wird.

## 3.3 Modellierung

Um Testfälle zu den Anforderungen zu generieren müssen diese in maschinenlesbarer Form vorliegen. Dazu können verschiedene Modellierungsarten herangezogen werden, die eine eindeutige Spezifikation besitzen. Die Unified Modeling Language (UML) [15] bietet einige Spracheinheiten, um Modelle zu verschiedenen Anwendungsgebieten zu modellieren. Für den hier dargestellten Ansatz eignen sich zum Beispiel Statecharts, die in UML als Zustandsdiagramm normiert sind, Aktivitätsdiagramme oder Petrinetze, da diese eine formale Spezifikation besitzen und sich Abläufe durch diese Modelle gut darstellen lassen. Aufgrund einer großen Anzahl an Tools, wie z.B. Modelchecker, die für Statecharts zur Verfügung stehen, wird für die Modellierung in dieser Arbeit Statecharts verwendet.

### 3.3.1 Vorgehen bei der Modellierung

Damit die Konsistenz und Vollständigkeit des Modells gewährleistet werden kann, ist ein methodisches Vorgehen von Vorteil. Einen Ansatz, um während der Modellierung die Qualität bezüglich dieser Kriterien sicherzustellen, bietet Test Driven Modelbased System Engineering (TD-MBSE) [25]. Im Folgenden wird dieses Vorgehensmodell erläutert und anschließend auf die hier vorgestellte Entwicklung des Anforderungsmodells übertragen.

Test Driven Development (TDD) ist in vielen Bereichen der Industrie und Forschung bereits Standard. TDD bedeutet grob umrissen, dass für die Implementierung eines Teilsystems zuerst Tests entwickelt und geschrieben werden und anschließend das Programm implementiert wird. Auf diese Weise wird unter anderem sichergestellt, dass die Implementierung den Anforderungen entspricht und ihre Funktion erfüllt [23].

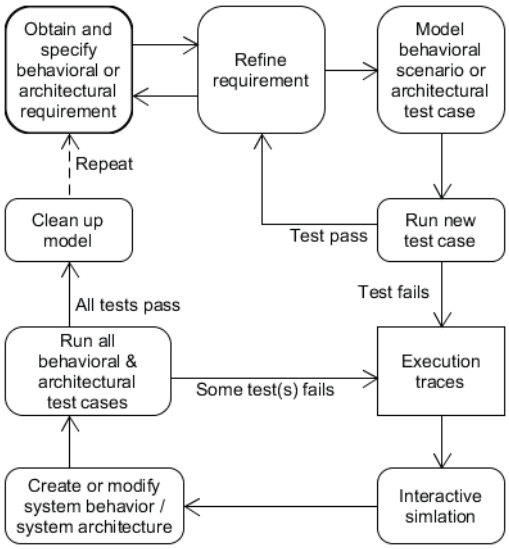


Abbildung 2: TD-MBSE für die Modellierung von Architektur und Verhalten von Alan Munck und Jan Madsen [25]

Beim TD-MBSE, beschrieben von Alan Munck und Jan Madsen, wurde das Vorgehen vom TDD übertragen auf die Modellierung eines Systems. In Abbildung 2 ist das Vorgehensmodell zur Entwicklung des Basismodells zu sehen. Dabei werden im ersten Schritt die Anforderungen herausgearbeitet und daraus zwei Typen von Modellen entwickelt, zum einen das Verhaltensmodell, welches das zeitliche und dynamische Verhalten eines

Systems darstellt, und zum anderen das Systemmodell, dass die Architektur des Systems beschreibt. Für die jeweiligen Anforderungen werden Testfälle erstellt. Ein solcher Testfall kann eine Sequenz von Eingaben und das Prüfen der Ausgaben und Zustandsänderungen des zu testenden Modells sein. Als Werkzeug hierfür eignen sich Modelchecker wie UPPAAL [2] oder YAKINDU [19].

Die Entwicklung hierbei ist ein iterativer Prozess. Die Testfälle werden auf dem Modell ausgeführt und validieren die Korrektheit des Verhaltens- und Systemmodells oder weisen bei Fehlschlag auf Fehler hin. Fehlgeschlagene Testfälle werden analysiert und die Anforderungen oder Modelle angepasst. Dies wird wiederholt, bis alle Tests bestehen. Der Prozess vom Verfeinern der Anforderungen und Erweitern der Modelle wird beliebig oft wiederholt, bis der gewünschte Umfang und Qualität der Modelle erreicht ist.

Für die Entwicklung des Anforderungsmodells wird der beschriebene Prozess des TD-MBSE so abgeändert, dass die Testfälle Abläufe aus dem Anforderungsmodell abbilden. Auf diese Weise werden im TD-MBSE Prozess iterativ sowohl das Verhaltensmodell als auch das Anforderungsmodell gemeinsam entwickelt. Durch dieses Vorgehen wird sichergestellt, das sich das Anforderungsmodell mit dem Verhaltensmodell deckt und das Anforderungsmodell gegen das spätere SUT ausführbar ist. Dies ist für die spätere Testausführung notwendig. Fehler können dadurch bereits vor der Implementierungsphase auffallen und behoben werden. Das Ziel dieses Vorgehens ist es, Modelle von ausreichend hoher Qualität zu schaffen, sodass diese für das MBT verwendet werden können.

### 3.3.2 Events und Systemvariablen

Für das Modell muss festgelegt werden, welche Variablen und Events dem Modell zur Verfügung stehen und wie diese benannt sind. Die Zustände von allen Aktoren und Sensoren des Systems werden der Zustandsmaschine als Variablen mit dem Prefix *sys_* von der Testumgebung, beschrieben in Kapitel 5, zur Verfügung gestellt. Verändert sich der Wert einer Variablen wird ein entsprechendes Event erstellt. Die Events fangen mit dem prefix *ev_* an und hören mit dem aktuellen Zustand auf. Dabei handelt es sich durch die Abstraktion um einen booleschen Wert. Dadurch ergeben sich die möglichen Endungen *_on* für den Wert true und *_off* für den Wert false.

Zusätzlich ist die Nutzung weiterer Variablen im Modell möglich, die bei der Testausführung zur Verfügung gestellt werden. Die Variablen sind *sys_hu_type*, die den Typ des

Werkstücks enthält, das sich in der Lichtschranke der Höhenmessung befindet, *sys_hu_orientation*, die die Orientierung des Werkstücks in der Lichtschranke der Höhenmessung enthält, *sys_switch_type* und *sys_switch_orientation*, die den Typ und die Orientierung des Werkstücks enthalten, das sich in der Lichtschranke der Weiche befindet.

### 3.3.3 Das Anforderungsmodell

Die einzelnen Anforderungen können Abschnitten, Sensoren oder Aktoren des Festo-Transfersystems zugeordnet werden. Daher kann das System in funktionale Abschnitte für die Modelle zerlegt werden. Es hat sich während der Entwicklung des Modells gezeigt, dass fünf Abschnitte für die Funktionalität des Betriebs ausreichend sind (Abb. 3). Abschnitt 1 liegt zwischen der ersten Lichtschranke und der Lichtschranke für die Höhenmessung. Abschnitt 2 behandelt die Höhenmessung. Abschnitt 3 kümmert sich um die Sortierung mittels der Weiche. Abschnitt 4 befasst sich mit der Rampe und der Abschnitt 5 behandelt den Bereich hinter der Weiche bis zur Lichtschranke am Ende des Laufbandes. Funktionen, die das ganze System betreffen, werden außerhalb dieser fünf parallelen Funktionalitäten behandelt.

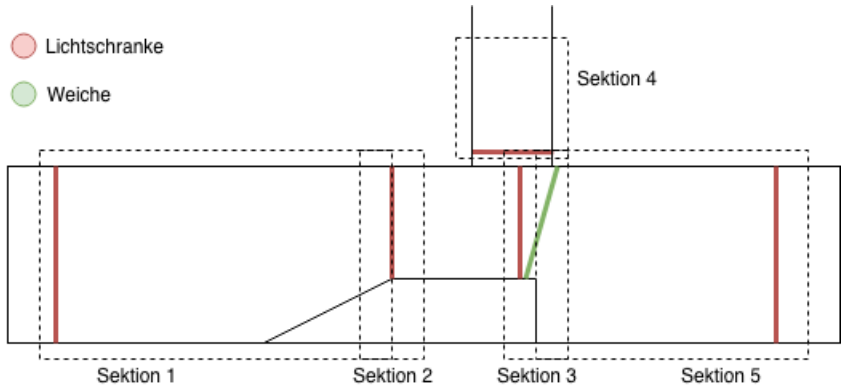


Abbildung 3: Unterteilung des Festo-Transfersystems in funktionale Abschnitte

Die ersten drei Punkte der Anforderungen (1, 2, 3) sind den einzelnen Anforderungen für den Betrieb übergeordnet und bilden damit die oberste Ebene der Hierarchie (Abb. 4). Beginnend beim Startzustand *stopped* führt eine Transition durch Drücken der Start-Taste in den Zustand *wait_for_operating*. In diesem Zustand gibt es eine Transition in den parallelen Zustand *operating*. Eine Transition führt aus diesem Zustand heraus, wenn die Stop-Taste gedrückt wurde, und führt dann in den Zustand *initialise_stop*. Wenn dort alle Anforderungen für den Ausgangszustand *stopped* erfüllt sind, wird die Transition

in diesen Zustand genommen. Damit ist die Anforderung 1 abgedeckt. Die Fehlerfälle aus den Anforderungen (4b, 4c, 4f) sind über Transitionen aus dem Zustand *operating* umgesetzt. Diese führen in den Zustand *wait_for _recognized _error*, in dem gewartet wird, bis die Anforderungen aus 2 erfüllt sind. Von dort aus wird die Fehlerbehandlung (3) über das Drücken der Reset-Taste realisiert.

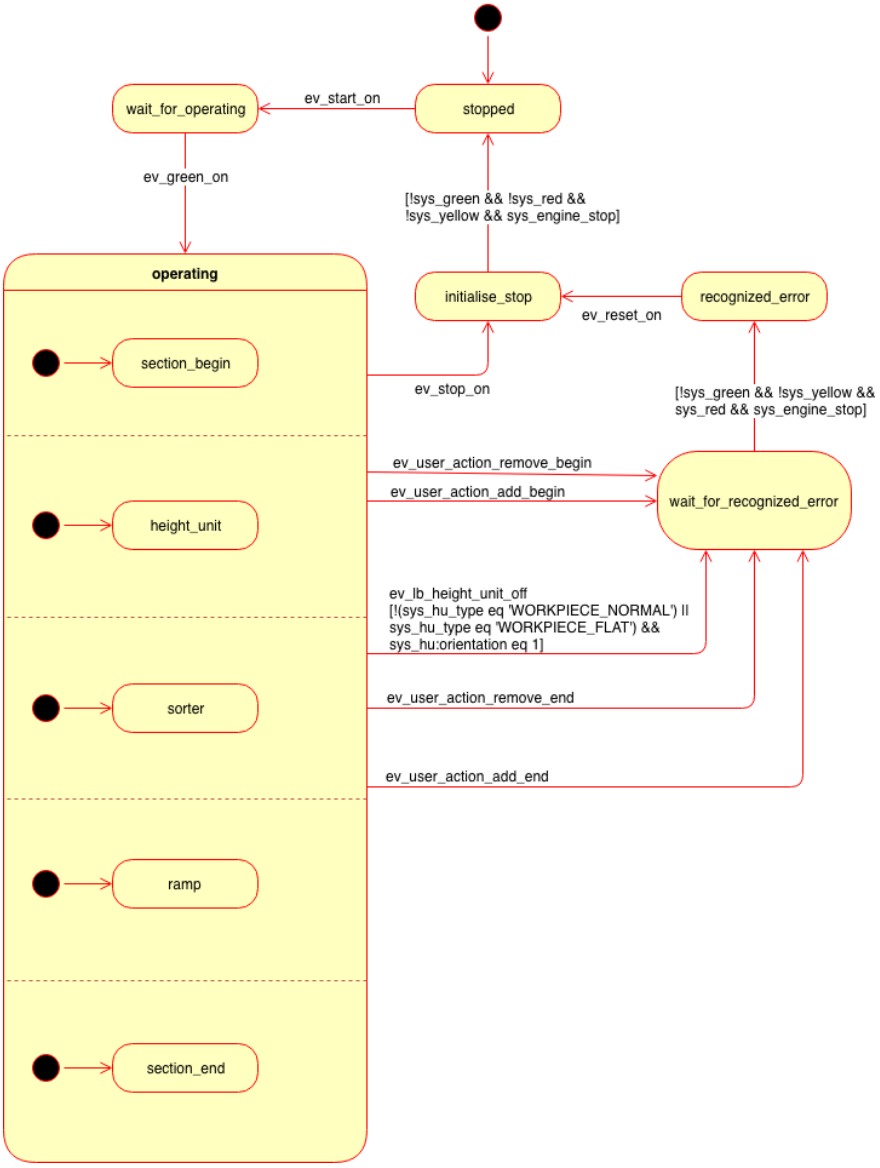


Abbildung 4: Anforderungsmodell der Anforderungen 1, 2 und 3

### section_begin

Der erste parallele Zustand *section_ begin* (Abb. 5) behandelt das Einlegen eines neuen Werkstücks in die Lichtschranke am Anfang des Laufbandes (4a).

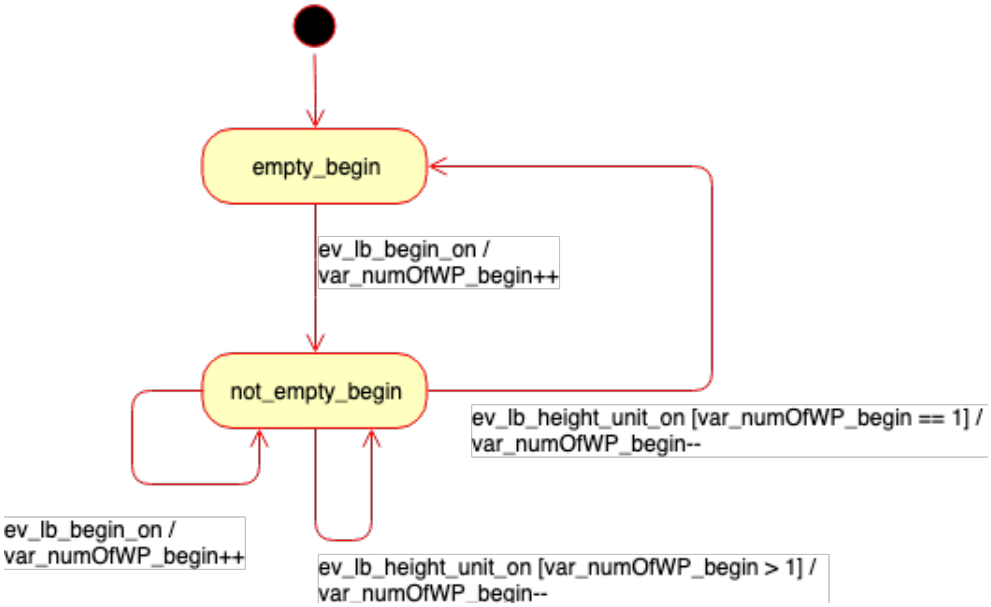


Abbildung 5: Anforderungsmodell des ersten Abschnitts

**height_unit**

Der Zustand *height_unit* behandelt die Anforderung für die Höhenmessung (4d). Ausgehend vom Startzustand *idle_hu* ist das Umschalten auf die langsame Bandgeschwindigkeit realisiert für die Dauer, in der sich ein Werkstück in der Höhenmessung befindet. Zum Ende der Höhenmessung, bevor das Laufband wieder auf die normale Geschwindigkeit zurückgesetzt wird, findet eine Prüfung über ein Guard statt, um sicherzustellen, dass es sich um ein valides Werkstück handelt und die Anforderung 4f nicht übergangen wird.

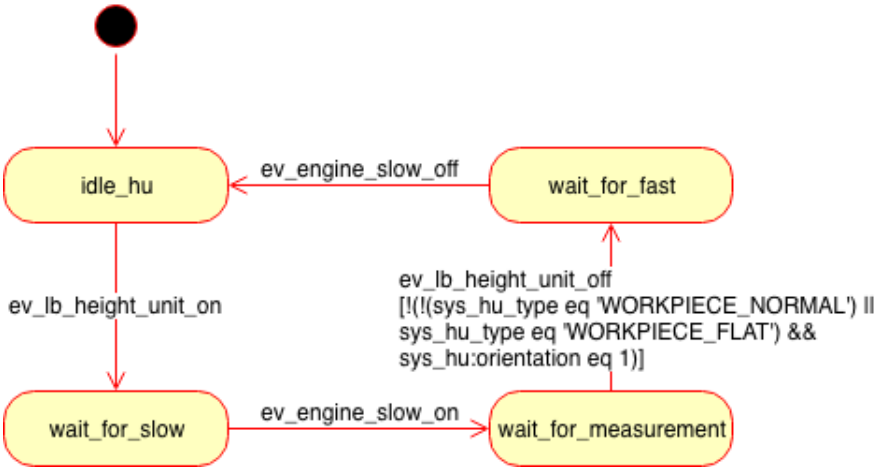


Abbildung 6: Anforderungsmodell des zweiten Abschnitts

**sorter**

In Abbildung 7 ist die Zustandsmaschine visualisiert, die die Anforderung an die Sortierung der Werkstücke (4e) umsetzt. Diese Funktionalität wird an der Lichtschranke der Weiche umgesetzt. Wie in der Höhenmessung kann auch an der Weiche nur ein Werkstück zur Zeit verarbeitet werden. Unterbricht ein Werkstück die Lichtschranke an der Weiche, wird je nach Werkstücktyp und Orientierung entschieden, ob das Werkstück behalten oder aussortiert werden soll. Soll ein Werkstück nach den Anforderungen behalten werden, überführt eine Transition die Zustandsmaschine in den Zustand *keep*. Dort wird solange verblieben, bis die Weiche wieder geschlossen ist und erwartet wird, dass das Werkstück sich anschließend hinter der Weiche auf dem Laufband befindet. Soll das Werkstück aussortiert werden, führt eine Transition in den Zustand *sort_out*. Das Werkstück gilt als aussortiert, wenn es die Lichtschranke der Rampe unterbricht.

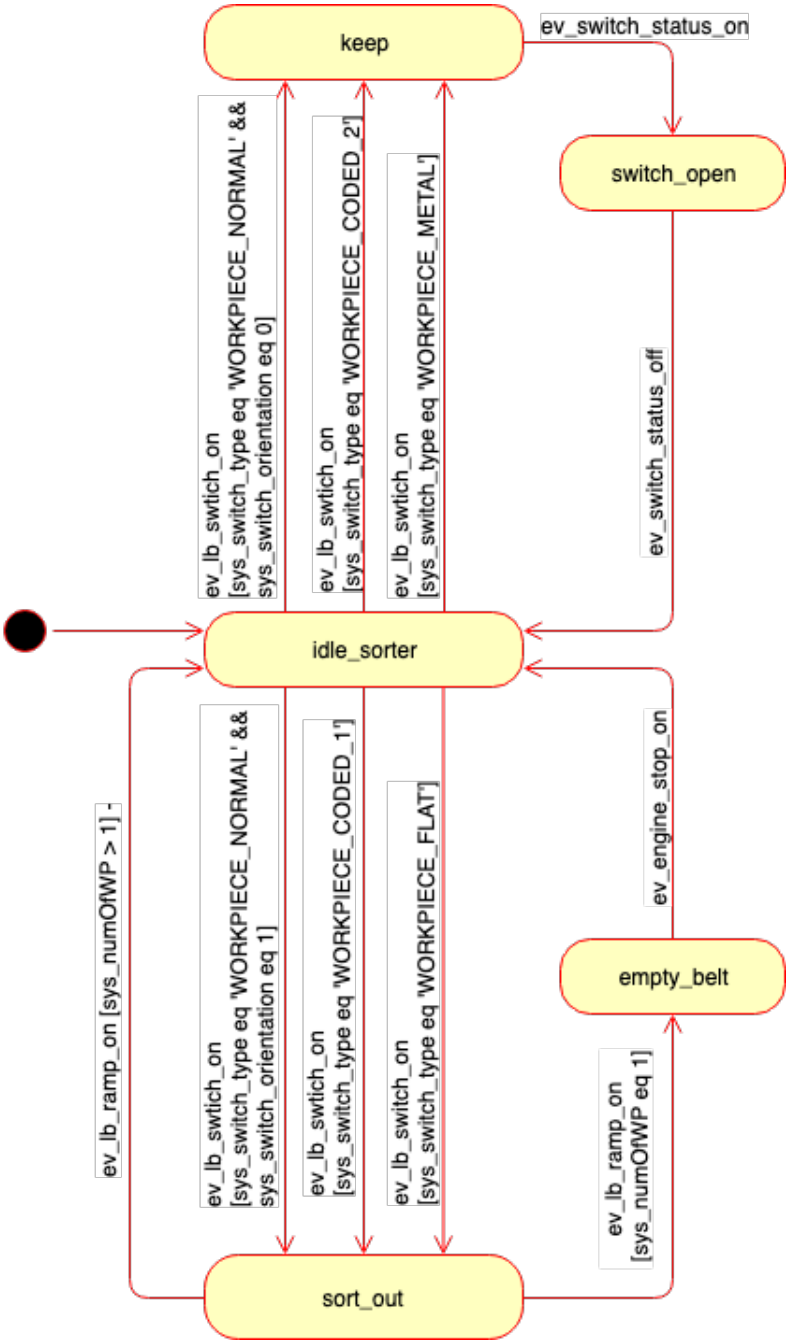


Abbildung 7: Anforderungsmodell des dritten Abschnitts

**ramp**

Der in Abbildung 8 dargestellte funktionale Abschnitt behandelt die Anforderung 4h. Die Rampe gilt als voll, wenn die Lichtschranke der Rampe länger unterbrochen ist als ein Werkstück braucht, um die Rampe so weit runter zu rutschen, dass die Lichtschranke wieder frei ist. Ein Timer wird gestellt sobald die Lichtschranke unterbrochen wird. Läuft er aus, erfolgt ein Übergang in den Zustand *ramp_full*. Von dort gibt es nur noch einen Pfad, der den Ablauf vom Anhalten des Bandes und dem Signalisieren bis zur Wiederinbetriebnahme des Sortiervorgangs modelliert. Ist nach dem Aussortieren kein weiteres Werkstück auf dem Band stoppt es nach der Anforderung 4.

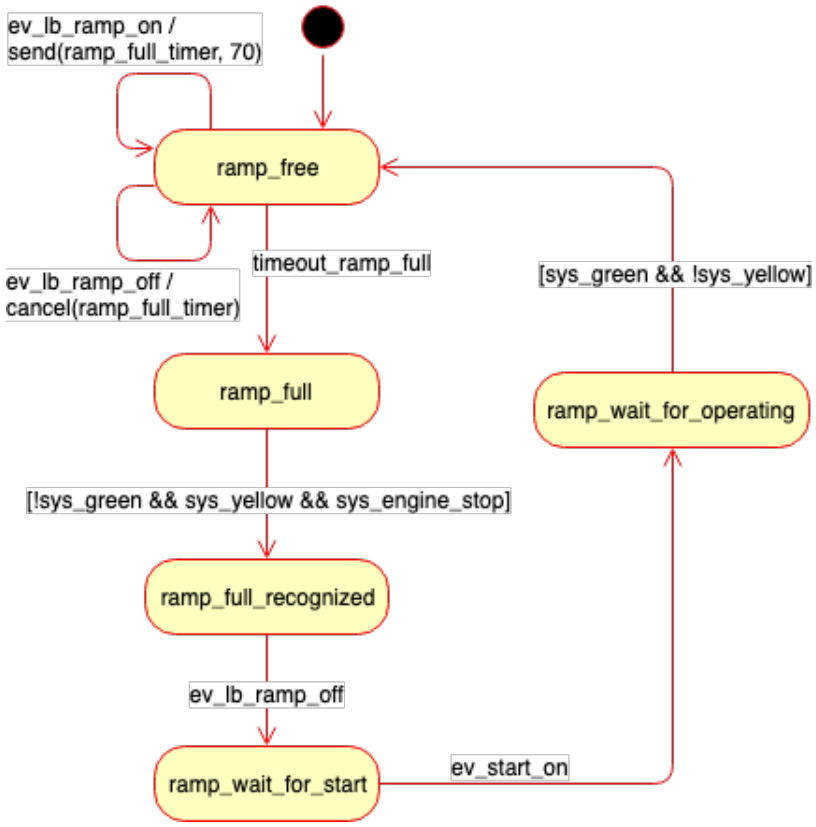


Abbildung 8: Anforderungsmodell des vierten Abschnitts

**section_end**

Im letzten Abschnitt des Systems wird die Anforderung 4g umgesetzt. In diesen Abschnitt (Abb. 9) sind wie im ersten Abschnitt mehrere Werkstücke gleichzeitig erlaubt. Unterbricht ein Werkstück die Lichtschranke am Ende des Laufbandes, wird das Band gestoppt. Wenn das Werkstück entfernt wurde und somit die Lichtschranke wieder frei ist, wird der Bandstop aufgehoben, falls sich noch Werkstücke auf dem Laufband befinden.

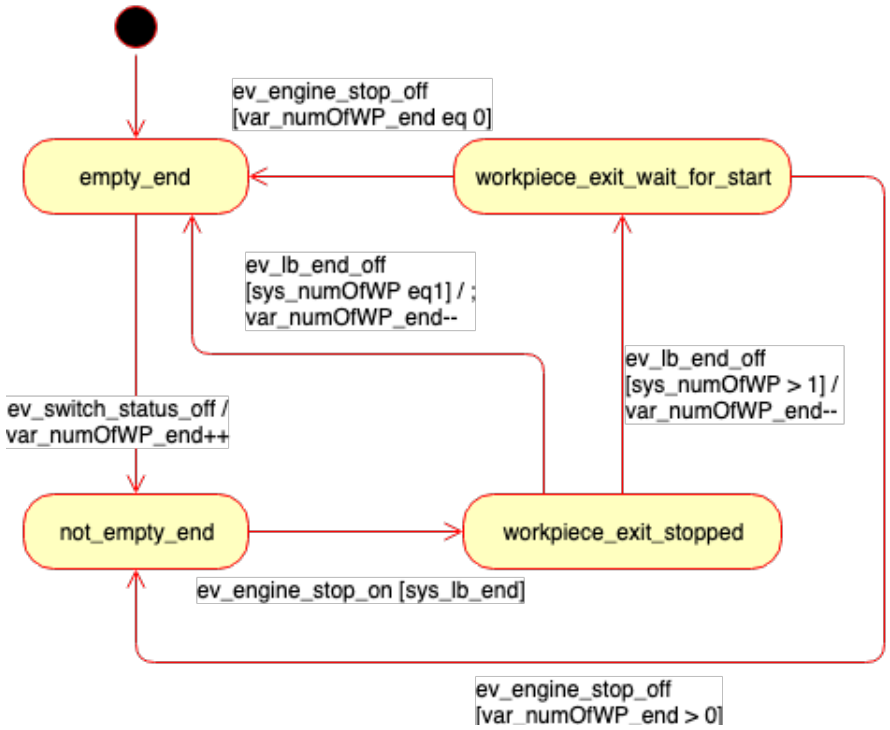


Abbildung 9: Anforderungsmodell des fünften Abschnitts

Das Anforderungsmodell führt keine Aktionen im System aus, sondern modelliert einen Ablauf, der die Anforderungen an das System darstellt.

# 4 Testfallgenerierung

Dieses Kapitel behandelt die Anforderungen an die Testfallgenerierung, den Aufbau der Testfälle, die aus der FSM abgeleitet werden, sowie den Algorithmus, der genutzt wird um diese Testfälle zu generieren.

## 4.1 Anforderungen an die Testfallerzeugung

Beginnend mit dem Startzustand kann das System unterschiedliche Zustände annehmen, abhängig von den Ereignissen, Aktionen oder Veränderung der Umgebungsvariablen. Eine Sequenz verschiedener Zustände und Transitionen, die einen Pfad durch die FSM bilden, stellt einen Testfall dar, der mindestens einer Anforderung oder Teilanforderung zuzuordnen ist. Das Ziel ist es, Testfälle zu erzeugen, die alle Anforderungen beschrieben in 3.1 mit mindestens einem Testfall pro Anforderungen abdecken. Die Qualität der Testfälle, die aus einer FSM abgeleitet werden, hängt stark von der Modellierung der Anforderungen, der Art der Testfallableitung und dem gewählten Überdeckungskriterium ab.

Die Zustandsüberdeckung ist das schwächste Überdeckungskriterium. Dabei wird die Überdeckung anhand der Zustände ermittelt, die durch die Testfälle erreicht werden. Ist jeder Zustand mindestens einmal in einem Testfall vorhanden, ist eine vollständige Zustandsüberdeckung erreicht. Beim Kriterium der Übergangsüberdeckung wird die Überdeckung der Übergänge als Maß verwendet. Sind alle Übergänge von den Testfällen abgedeckt, spricht man von einer vollständigen Übergangsüberdeckung. Eine vollständige Übergangsüberdeckung ist eine echte Obermenge der Zustandsüberdeckung und ist damit ein stärkeres Kriterium. Die Pfadüberdeckung ist das stärkste Überdeckungskriterium und eine echte Obermenge der Übergangsüberdeckung. Eine vollständige Pfadüberdeckung enthält in der Regel wesentlich mehr Pfade, als für eine vollständige Übergangsüberdeckung benötigt wird [21].

Durch manuelles Ableiten von Pfaden aus der FSM und Vergleich dieser Pfade wird erkennbar, welches der eben erläuterten Kriterien sich für die Testfallerzeugung eignet. Da viele der Zustände im Anforderungsmodell mehrere Transitionen besitzen, die jeweils für eine Anforderung relevant sind, und diese auch bei 100% Zustandsüberdeckung nicht abgedeckt werden, ist die Zustandsüberdeckung als Kriterum zu schwach. Das Kriterium der Übergangsüberdeckung deckt bei 100% Überdeckung alle Transitionen ab und damit

auch die Übergänge, die bei der Zustandsüberdeckung übergangen werden. Mit diesem Kriterium wird für jede Anforderung mindestens ein Testfall erzeugt und genügt damit in diesen Beispiel den Anforderungen an die Testfallgenerierung für das in dieser Arbeit vorgestellte Modell. Ist dies nicht der Fall sollte das Modell in mehrere einfachere Modelle aufgeteilt werden, wie es in den Anforderungen an das Modell (3.2) beschrieben ist. Die Pfadüberdeckung bringt im Vergleich zur Übergangsüberdeckung bezüglich der Testfälle viel Redundanz mit sich und bietet damit in diesem Fall wenig Mehrwert.

Die Übergangsüberdeckung ist in diesem Beispiel ausreichend. Die FSM ist so modelliert, dass für jede Anforderung oder Teilanforderung eine Transition oder ein Satz an Transitionen existiert, die diese Anforderung (3.1) modelliert. Dies wird im Kapitel 6 näher erläutert. Für andere Modelle muss dies neu evaluiert werden.

Einige Wächterbedingungen im Modell können erfordern, dass Zyklen durch die FSM genommen werden, um diese Bedingungen zu erfüllen. Für eine vollständige Übergangsüberdeckung kann es daher notwendig sein, dass Zyklen genommen werden, um die vollständige Überdeckung zu erreichen.

Für den Algorithmus müssen Endbedingungen festgelegt werden, damit die erzeugten Testfälle Sinn ergeben. Eine notwendige Endbedingung ist, dass in Zuständen, deren ID mit *wait_for_* oder *initialise_* beginnen oder den Zuständen *sort_out* und *keep* nicht abgebrochen wird, auch wenn jede Transition aus diesem Zustand bereits in einem Pfad vorhanden und damit abgedeckt ist. In diesen Zuständen wird mit einer Transition aus diesem Zustand ein erwartetes Verhalten geprüft. Das Erreichen eines solchen Zustandes bestätigt in dieser Modellierung nicht, dass eine Anforderung erfüllt wurde.

Auch die Länge der Pfade kann als Kriterium für die Qualität der Testfälle herangezogen werden. Angenommen alle Übergänge ließen sich in einem Pfad abhandeln, dann ist wahrscheinlich, dass Fehler maskiert werden, da die Testfallausführung beim ersten Fehler abgebrochen wird. Im Falle eines Fehlers kann nicht sichergestellt werden, dass die Weiterführung des Programms korrekt abläuft. Daher ist es sinnvoll, die Pfade möglichst kurz zu halten. Im Idealfall ist ein Pfad für eine Anforderung oder eine Teilanforderung zuständig. Auf diese Weise ist es leichter, die Erfüllung einzelner Anforderungen zu prüfen und Fehlerursachen zu finden.

Das Ziel ist es Testfälle zu generieren, die hauptsächlich eine Anforderung pro Testfall prüfen und jede Anforderung durch mindestens einen Testfall abgedeckt wird.

## 4.2 Werkzeuge zur Testfallerzeugung

Zur Testfallerzeugung können bestehende Werkzeuge verwendet werden, die aus FSM's Testfälle erzeugen. Dabei gibt es dafür eine Reihe von kommerziellen und Open Source Lösungen. Für die Testfallerzeugung in dieser Arbeit wurden nur Open Source Lösungen betrachtet [14][22][5]. Warum es sich angeboten hat, für diese Arbeit eine eigene Lösung zu erarbeiten, lässt sich unter folgenden drei Punkten zusammenfassen. **1.** Die zur Verfügung stehenden Algorithmen ermöglichen nicht das Erfüllen der in 4.1 genannten Anforderungen an die Testfallerzeugung. Einige Tools bieten zum Zeitpunkt dieser Arbeit nur Algorithmen, die Transitionen zufällig auswählen. Dadurch werden bei jedem Durchlauf unterschiedliche Testfälle erzeugt, wodurch die Reproduzierbarkeit verloren geht. **2.** Tools erfordern eine spezielle DSL (domain-specific language) oder im Code umgesetzte FSM's, wodurch ein Mehraufwand entsteht, da die FSM erst in die entsprechende DSL oder Code umgewandelt werden muss. Bei Änderungen an Modellen müssen diese Änderungen auch im Code vorgenommen werden. **3.** Viele Tools bieten keine Unterstützung von Parallelität, Untermaschinen und auch nicht die Möglichkeit, Variablen zu markieren, die in Bedingungen während der Testfallerzeugung beachtet werden sollen. Dies führt zu einem hohen Aufwand bei der Vorverarbeitung von Modellen, damit diese von den jeweiligen Tools verwendet werden können.

## 4.3 Aufbau der Testfälle

Die generierten Testfälle müssen jeweils einen ausführbaren Pfad bilden. Diese beginnen immer beim Startzustand des Gesamtautomaten und haben eine Transition in den nächsten Zustand, die bei der Ausführung des Testfalls genommen werden soll. Dies wiederholt sich, bis der Zustand erreicht ist, an dem der Test abgebrochen wird. Beim Erreichen dieses Zustandes gilt der Test als erfolgreich. Wird durch Fehlverhalten des SUT ein anderes Event geworfen, dass dazu führt, dass im Anforderungsmodell eine andere Transition genommen wird, als es im Pfad des Testfalls erwartet wird, gilt der Test als fehlgeschlagen. In den Pfaden kann es auch Transitionen geben, die weder ein Event noch einen Guard haben. Diese leeren Transitionen beschreiben Übergänge, die sofort genommen werden z.B. von einen Zustand, der eine Untermaschine enthält, in dessen Startzustand. Diese Pseudo-Transitionen dienen der Übersichtlichkeit und Nachvollziehbarkeit der Pfade. Jeder Testfall erhält durch manuelle Eingabe ein Label, der das Ziel dieses Testfalls beschreibt sowie eine Inputsequenz von Werkstücken.

Notiert werden die Testfälle in folgenden Kapiteln mit einer Auflistung der Zustände und Transitionen in der Sequenz, wie sie bei der Testausführung entstehen sollen.

*{zustand} –{event}[{guard}]–> {zustand} –{event}[{guard}]–> ... {zustand}*

## 4.4 Der Algorithmus zur Testfallgenerierung

Aufgrund den Entscheidungen aus 4.1, dass jeder Testfall einen ausführbaren Pfad in der FSM repräsentiert und dass eine Übergangsüberdeckung von 100% erreicht werden soll, eignet sich die Testfallgenerierung aus einem Übergangsbaum für diese Anforderungen [7].

Der Übergangsbaum besteht aus Knoten, die vier Daten enthalten. Diese sind der Zustand, der von diesem Knoten repräsentiert wird, die Transition, die genommen wurde um diesen Zustand vom vorherigen aus zu erreichen, dem Kontext, der die Variablen der FSM nach Ausführung der Eintrittsaktion und der Aktion der Transition enthält, und einem Flag, das aussagt, ob dieser Knoten bereits vollständig bearbeitet wurde. Im folgenden wird erklärt, wie der rekursive Algorithmus, zur Erzeugung des Übergangsbaumes aussieht; dabei dient Abbildung 10 als Übersicht.

Der Wurzelknoten des Übergangsbaums ist der Startzustand mit leerer Transition und initialisiertem Kontext, auf dem die Eintrittsaktion des Startzustandes ausgeführt wird. Von dort aus wird der Baum im Tiefendurchlauf Ast für Ast aufgebaut.

Angefangen beim Wurzelknoten wird der Zustand, der von dem Knoten gehalten wird, als bearbeitet markiert. Von diesem Zustand aus wird eine Transition genommen. Dabei wird eine Kopie des Kontextes des vorherigen Knotens erstellt, auf dem die Austrittsaktion des vorherigen Zustandes, die Aktion der Transition und die Eintrittsaktion des Zielzustandes ausgeführt wird. Mit diesen Daten, der Transition, dem Zielzustand und dem Kontext, wird ein neuer Knoten angelegt. Ein rekursiver Aufruf bearbeitet den neuen Knoten auf gleiche Weise. Durch den rekursiven Aufruf wird der Knoten weiterbearbeitet, sobald der Ast, der durch den neuen Knoten erstellt wurde, den Abbruchbedingungen entsprechend vollständig aufgebaut ist.

Für die Wahl der Transition wird die Art des Zustandes unterschieden. Enthält der Zustand, der sich in Bearbeitung befindet, eine Untermaschine, wird eine Transition ohne Aktion, Event oder Guard erstellt, die in den Startzustand der Untermaschine führt. Ist der Zustand parallel, wird jeweils eine Transition, wie bei einer Untermaschine für jeden

der Unterzustände erstellt. Handelt es sich um einen normalen Zustand, werden alle davon ausgehende Zustände sowie alle Zustände der Eltern in Betracht gezogen.

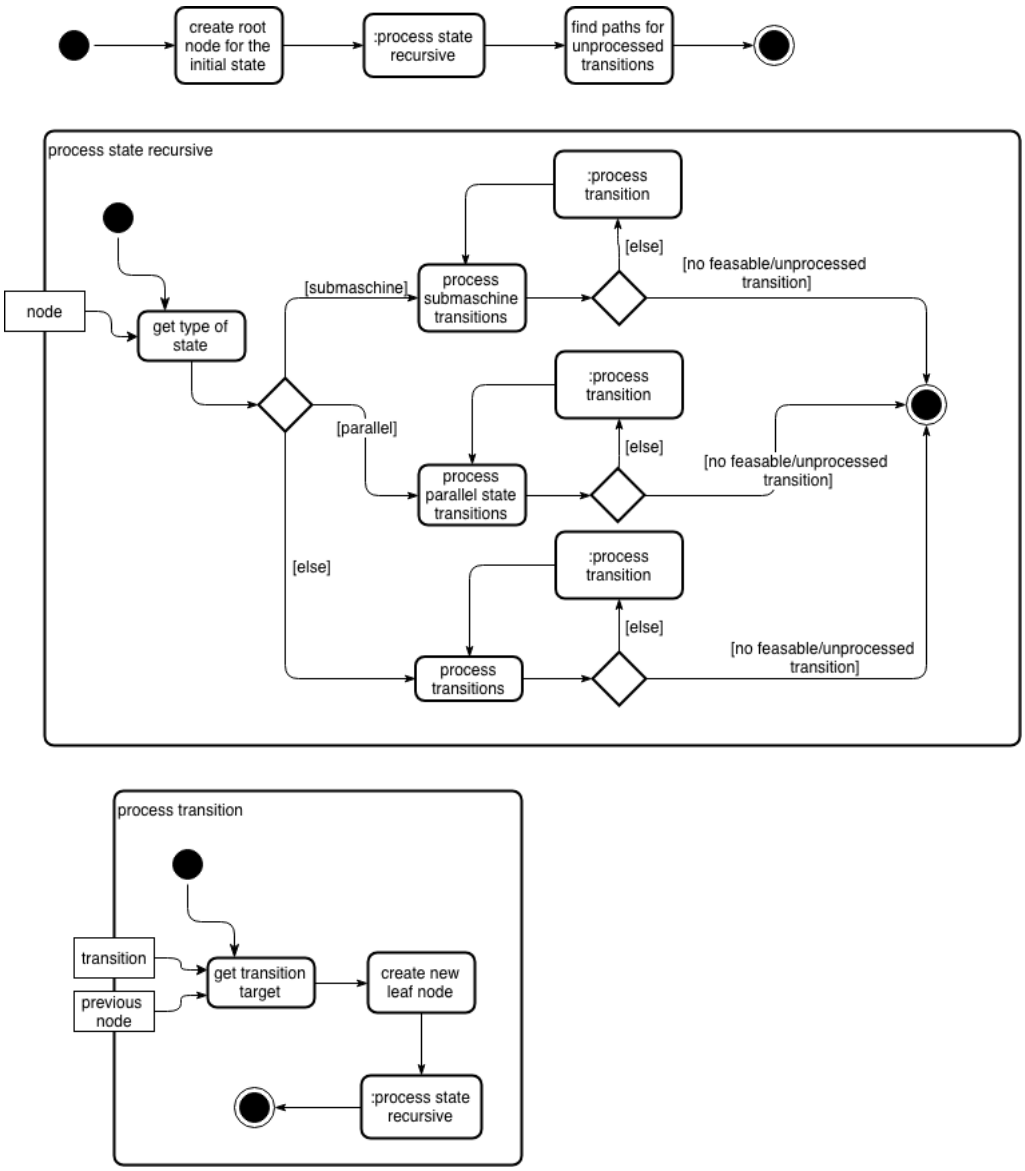


Abbildung 10: Rekursives Aufbauen des Übergangsbaumes

Durch Wächterbedingungen an den Zustandsübergängen ist es nicht immer möglich, alle Zustandsübergänge von einen Knoten aus zu nehmen. Zur Prüfung werden die Systemvariablen (sys_.+), also Variablen, die den Inhalt eines Bits der GPIO-Register des SUT repräsentieren, ignoriert, da diese vom SUT abhängen und kein Rückschluss auf deren Werten während der Testfallgenerierung möglich ist. Zur Prüfung, ob ein Übergang va-

lide ist, werden die Variablen verwendet, die im Datenmodel des SCXML definiert sind und über das *assign* Tag verändert werden können. Bei der Wahl einer Transition wird die Bedingung immer gegen den Kontext des Knotens geprüft.

Abgebrochen wird die Bearbeitung eines Knotens wenn alle Transitionen, die von diesem Zustand aus genommen werden können, abgearbeitet sind. Kann eine Transition aufgrund ihrer Wächterbedingung nicht genommen werden und sind alle anderen Transitionen abgedeckt, wird ebenfalls abgebrochen. Die Ausnahme hierfür bilden die in 4.1 genannten Zustände.

Um die Anforderung möglichst kurzer Pfade zu erfüllen, wird als weiteres Abbruchkriterium das Erreichen eines Zustandes genommen, der bereits genommen wurde. Dieses Abbruchkriterium sorgt dafür, dass für Zustände, aus denen mehrere Transitionen ausgehen, auch dann mehrere Pfade generiert werden, wenn zyklisch zu diesen Zustand zurückgekehrt wird. Dieses Verhalten ist am Beispiel der FSM für die Sortierung (Abb. 7) zu erkennen. Für den Sortiervorgang geht aus dem Zustand *idle_sorter* eine Transition für jeden Werkstücktyp aus. Für jede dieser Transitionen ist ein eigener Testfall gewünscht, da jede Transition eine andere Teilanforderung umsetzt. Die FSM *sorter* führt zyklisch in den Zustand *idle_sorter*. Wenn nun beim erneuten Eintreten in diesen Zustand abgebrochen wird, haben wir mit diesem Pfad einen Testfall für die Sortierung von einen einzelnen Werkstücktyp. Für die anderen Transitionen wird jeweils ein anderer Pfad und damit ein eigener Testfall generiert.

Sind alle Knoten im Baum mit dem eben beschriebenen Algorithmus verarbeitet worden, wird geprüft, ob Transitionen vorhanden sind, die nicht genommen wurden. Gibt es solche Transitionen, wie z.B. Transitionen, die erst nach mehreren zyklischen Durchläufen ihre Bedingungen erfüllen, müssen für diese Transitionen noch Pfade gefunden werden. Um gültige Zustände für die verbliebenen Übergänge zu finden, wird eine maximale Rekursionstiefe definiert, die als Abbruchkriterium gilt, wenn nicht zuvor für jede verbliebene Transition ein Pfad gefunden wurde. Von jedem Blatt des Übergangsbaumes wird der Breite nach jede mögliche Transition genommen und damit der Baum im Breitendurchlauf weiter aufgebaut. Anschließend werden alle Knoten aus dem Baum entfernt, die mit dem Breitendurchlauf hinzugefügt wurden und nicht Teil eines Pfades sind, der eine der verbliebenen Transitionen enthält.

Falls der Aufbau des Übergangsbaumes durch Erreichen der maximalen Rekursionstiefe abbricht, wurde noch nicht für alle Transitionen ein Pfad gefunden. Dieser Fall kann auf einen Fehler im Modell hinweisen.

Jeder Pfad durch den Baum, von der Wurzel bis zu einem Blatt, bildet einen Pfad durch die FSM ab und stellt damit einen Testfall dar.

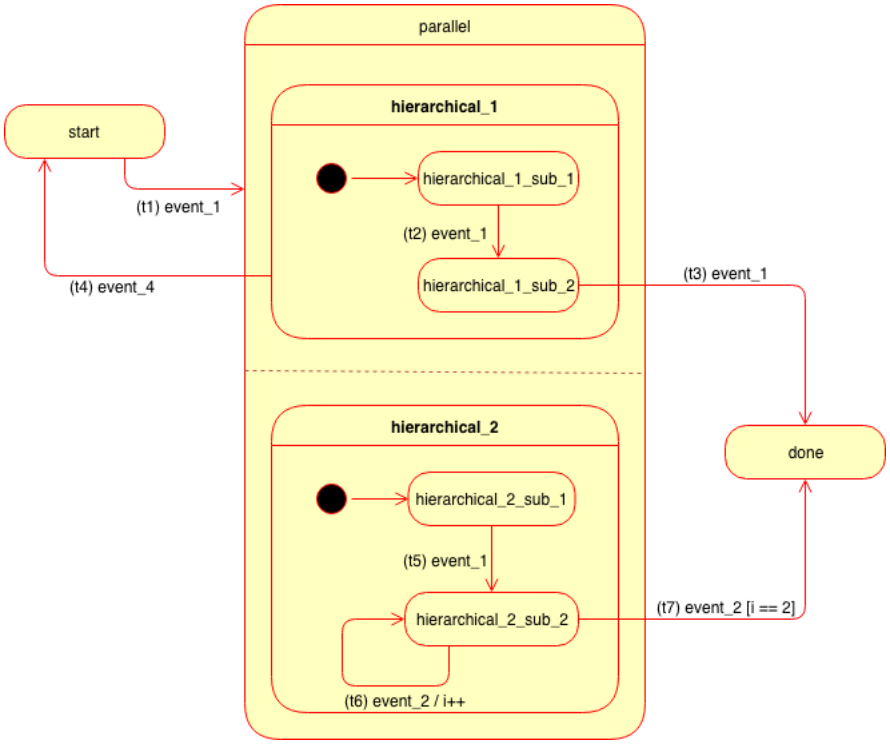


Abbildung 11: Beispiel-FSM

### Beispiel

Wird dieser Algorithmus auf die Beispiel-FSM (Abb. 11) angewandt werden folgende Schritte durchlaufen:

1. Erstellen des Wurzelknoten für den Zustand *start*.
2. Erstellen eines Knotens abgehend von Wurzelknoten für die Transition t1 in den Zustand *parallel*.
3. Erstellen eines Knotens für den ersten nebenläufigen Zustands von *parallel* (*hierarchical_1*).
4. Erstellen eines Knotens für den Startzustand des Unterautomaten des Zustandes *hierarchical_1_sub_1*.
5. Erstellen eines Knotens für die Transition t2 in den Zustand *hierarchical_1_sub_2*.

6. Erstellen eines Knotens für die Transition t3 in den finalen Zustand *done*. Mit dem Erreichen eines finalen Zustandes ist der erste Ast im Tiefendurchlauf vollständig aufgebaut.

7. Der Stack wird abgebaut zur Verarbeitung des in Punkt 5 erstellen Knotens. Von dem Zustand aus gibt es eine weitere Transition t4, die vom Parent *hierarchical_1* ausgeht und genommen werden kann. Für diese Transition wird vom Knoten aus Punkt 5 ein neuer Knoten für den Zustand *start* erstellt.

8. Vom Zustand *start* wurden bereits alle Transitionen verarbeitet. Daher wird der Stack weiter abgebaut zur Verarbeitung des in Punkt 4 erstellten Knotens. Die Transition t3 des Parents wurde von diesen Zustand aus noch nicht genommen. Daher wird ein neuer Knoten für den Zustand *start* erstellt.

9. An diesem Punkt sind alle Transitionen für den ersten nebenläuigen Zustand bearbeitet und der Stack wird bis zur Verarbeitung des Knotens *parallel* (2) abgebaut. Nun wird ein neuer Knoten für den zweiten nebenläufigen Zustand *hierarchical_2* erstellt.

10. Von diesen Punkt aus werden weitere Knoten nach dem bereits beschriebenen Vorgehen erstellt.

11. Ist der Algorithmus bei der Verarbeitung des Zustands *hierarchical_2_sub_2* angekommen, gibt es zwei Transitionen, die von diesem Zustand ausgehen. Die Transition t7 hat eine Wächterbedingung, die besagt, dass die Variable i den Wert zwei haben muss, damit die Transition genommen werden kann. Da die Variable i initial den Wert 0 hat kann diese Transition nicht genommen werden. Daher wird ein neuer Knoten für den Zielzustand *hierarchical_2_sub_2* der Transition t6 erstellt, in der eine Aktion i inkrementiert.

12. In diesen Knoten gibt es nun keine Transition mehr, die noch nicht verarbeitet wurde oder genommen werden kann, da die Variable i den Wert 1 hat und damit die Wächterbedingung der Transition t7 nicht erfüllt. Damit wird der Stack des ersten Teils des Algorithmus, dem rekursiven Aufbauen des Übergangsbaums, vollständig abgebaut, da alle Knoten vollständig bearbeitet wurden.

13. Zum Finden eines Pfades, in der die Transition t7 vorkommt, wird der Baum im Breitendurchlauf erweitert. Dazu wird von den beiden Blättern *start* jeweils ein Knoten für den Zustand *parallel* angelegt und vom Blatt *hierarchical_2_sub_2*

ausgehend ein neuer Knoten für das Ziel der Transition t6, in dem die Aktion i ein weiteres Mal inkrementiert.

14. Da für t7 noch kein Pfad gefunden wurde, werden für die beiden neuen Knoten *parallel* jeweils zwei neue Knoten für die nebenläufigen Zustände angelegt. In dem Pfad vom neuen Knoten *hierarchical_2_sub_2* hat die Variable i nach zwei Durchläufen der Transition t6 den Wert 2. Damit ist die Wächterbedingung von t7 erfüllt und es wird ein Knoten für den Zustand *done* erstellt. An diesem Punkt ist eine vollständige Übergangsüberdeckung erreicht. Alle Knoten, die ab Punkt 13 erstellt wurden und nicht in dem Ast liegen, in dem die Transition t7 genommen wurde, werden entfernt. Damit ist der Baum vollständig aufgebaut und der Algorithmus abgeschlossen.

Der durch den Algorithmus erzeugte Übergangsbaum (Abb. 12) enthält folgende 4 Pfade:

start – event_1 –> parallel —-> hierarchical_1 —-> hierarchical_1_sub_1 – event_1 –> hierarchical_1_sub_2 – event_1 –> done

start – event_1 –> parallel —-> hierarchical_1 —-> hierarchical_1_sub_1 – event_1 –> hierarchical_1_sub_2 – event_4 –> start

start – event_1 –> parallel —-> hierarchical_1 —-> hierarchical_1_sub_1 – event_4 –> start

start – event_1 –> parallel —-> hierarchical_2 —-> hierarchical_2_sub_1 – event_1 – > hierarchical_2 – event_2 –> hierarchical_2_sub_2 – event_2 –> hierarchical_2_sub_2 – event_1 [i == 2] –> done

Mit diesen vier Testfällen ist jede der Transitionen abgedeckt und somit eine vollständige Übergangsüberdeckung erreicht.

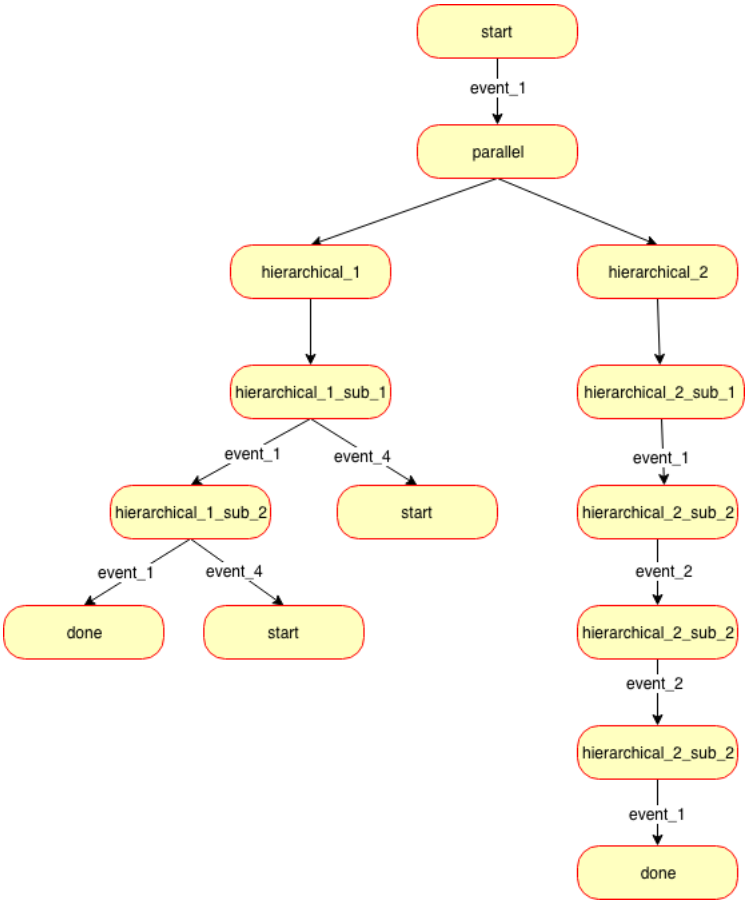


Abbildung 12: Übergangsbaum der Beispiel-FSM. Die Events sind Teils des Knotens unterhalb.

Obwohl in den Testfällen keine Pfade doppelt erstellt werden, kann es Testfälle geben, dasselbe Verhalten darstellen und kaum Mehrwert bietet. Ansätze zum Erkennen und zum Entfernen dieser redundanten Testfälle werden in dieser Arbeit nicht genutzt. Die zur Validierung genutzten Testfälle (Kap. 6) wurden vom Tester um redundante Testfälle reduziert.

## 4.5 Inputsequenzen der Testfälle

Jeder Pfad von der Wurzel bis zu einem Blatt im Übergangsbaum bildet einen Testfall. Um den Testfall ausführbar zu machen, muss der Ablauf der Eingaben am Festo-Transfersystem definiert werden. Das Betätigen eines Schalters wird ausgelöst, wenn im aktuellen Zustand für die nächste Transition ein Schalter-Event (ev_start_on, ev_stop_on, ev_reset_on, ev_estop_on) benötigt wird. Das Hinzufügen oder Manipulieren von Werkstücken muss vom Nutzer nach der Testfallerzeugung manuell definiert werden. Dazu wird ein Zustand ausgewählt, der als Startzustand für den Input von Werkstücken gilt. Wird dieser Zustand erreicht wird das erste definierte Werkstück in die Lichtschranke am Anfang des Bandes gelegt. Die anderen Werkstücke und Aktionen an den Werkstücken sind durch den Abstand zum vorherigen Werkstück bestimmt.

## 4.6 Erweiterung des Anforderungsmodells um Fehlerzustände

Das Anforderungsmodell besitzt keine Fehlerzustände. Jedoch ist die Erweiterung dieses Modells um Fehlerzustände aus verschiedenen Gründen sinnvoll. Wird bei der Testausführung ein Fehlerzustand des Anforderungsmodells erreicht, ist der Testfall fehlgeschlagen und das Erreichen des Fehlerzustandes wird im Testprotokoll mit aufgeführt, was die Fehleranalyse erleichtert. Da ein Echtzeitsystem geprüft wird und daher zeitliche Kriterien bei der Reaktion auf Ereignisse eingehalten werden müssen, ist die Erweiterung um Fehlerzustände notwendig um diese zu prüfen. Dazu werden Zustände, auf die unmittelbar eine Reaktion erwartet wird, um eine Eintrittsaktion erweitert, in der ein Timer gestartet wird. Wenn dieser Zustand verlassen wird, wird der Timer in der Austrittsaktion gestoppt. Wenn der Timer abläuft, weil das System nicht im vorgesehenen Zeitrahmen reagiert, wird ein Event geworfen und eine Transition in den Fehlerzustand genommen. Damit werden die von dem Echtzeitsystem vorausgesetzten Timing-Constraints abgedeckt. Auch ein Verhalten des SUT, das in einem Zustand nicht erwünscht ist wie z.B. das Einschalten einer falschen Signalleuchte, kann mit einer Transition in einen Fehlerzustand geprüft werden. Hierbei wird die Prüfung über Events und Guards realisiert.

# 5 Systemarchitektur

In diesem Kapitel wird die Architektur und Funktionsweise der entwickelten Anwendung beschrieben. Dabei wird erläutert, wie die Komponenten der Anwendung zusammenspielen und wie die Testausführung abläuft.

## 5.1 Kurzübersicht der Anforderungen an die Anwendung

Es soll eine Anwendung entwickelt werden, mit der eine in der HiL-Simulation (2.1) laufende Anwendung durch MBT getestet werden kann. Dazu sind im Folgenden die Anforderungen aufgelistet, die von der hier entwickelten Anwendung erfüllt werden sollen.

Dem Nutzer soll ermöglicht werden, ein Modell im SCXML-Format (2.6) von der Anwendung einlesen zu lassen. Das Modell muss dabei die Anforderungen aus Kapitel 3.2 erfüllen und die in Kapitel 3.3.2 beschriebenen Bezeichnungen für die Events und Guards nutzen.

Die Anwendung muss mit dem in Kapitel 4 beschriebenen Algorithmus Testfälle aus dem SCXML generieren. Diese sollen dem Nutzer visuell dargestellt werden. Dabei soll dieser Testfälle markieren können, die bei der Testausführung nicht mit ausgeführt werden sollen. Zusätzlich muss der Nutzer die Testfälle mit einem Label sowie einer Inputsequenz (4.5) versehen können.

Die Testfälle sollen von der Anwendung gegen das SUT ausgeführt werden können. Dazu muss die Anwendung den Status von der Simulation empfangen sowie die Simulation ansteuern können (2.3). Bei der Testausführung müssen geforderte Aktionen, wie das Betätigen einer Taste, in der Simulation ausgelöst werden. Außerdem muss die Inputsequenz des Testfalls an der Simulation ausgeführt werden. Dabei soll am SCXML geprüft werden, ob das SUT das vom Testfall vorgegebene Verhalten zeigt. Nach der Testausführung muss dem Nutzer das Testprotokoll ausgegeben werden.

## 5.2 Umgebung

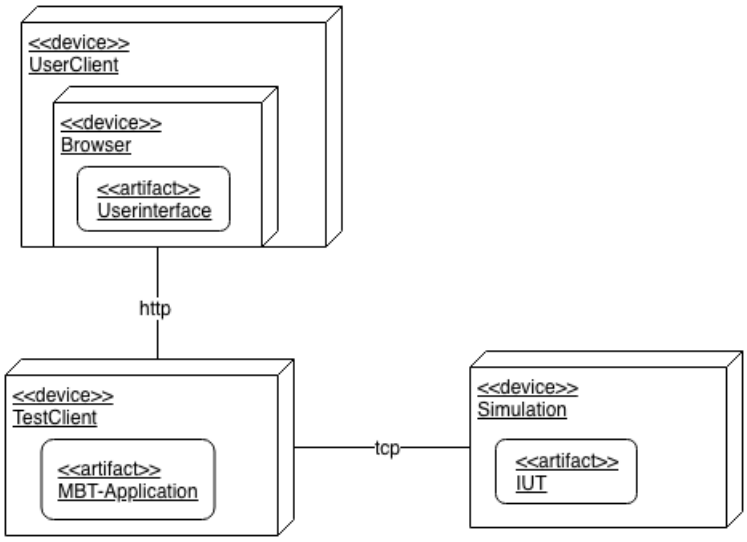


Abbildung 13: Deployment Diagramm des Systems

In Abbildung 13 ist dargestellt, wie das entworfene System in drei Teile unterteilt ist. Die Simulation mit der Implementation under Test (IUT) läuft auf einem Beaglebone Black oder einem Unix-System. Auf dem TestClient läuft die in dieser Arbeit entwickelte Anwendung für das MBT und ist mittels TCP zu der Simulation verbunden. In den folgenden Abschnitten wird diese Anwendung weiter erläutert. Angesteuert wird die Simulation über eine Weboberfläche, die der Nutzer auf seinem Gerät (UserClient) aufrufen kann. Darüber kann der Nutzer eine SCXML-Datei auswählen, aus der die Testfälle generiert werden, die Testfälle verwalten und von der Anwendung ausführen lassen sowie das Testprotokoll einsehen.

## 5.3 Komponenten der Anwendung

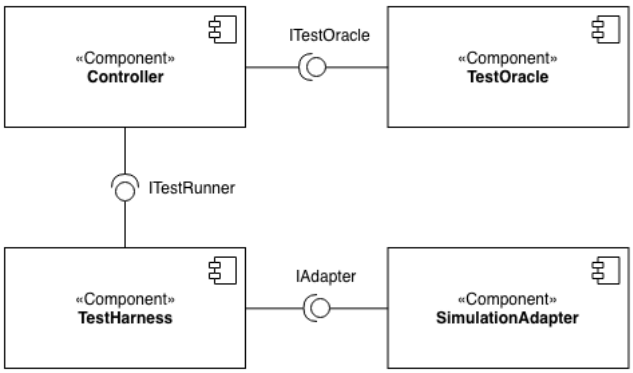


Abbildung 14: Komponenten Diagramm der MBT Anwendung

Abbildung 14 zeigt das System der MBT-Anwendung in vier Kompnenten mit unterschiedlichen Funktionalitäten unterteilt.

### 5.3.1 Controller

Die Komponente Controller bietet zwei Rest-Schnittstellen für die Steuerung mittels HTTP.

```
List<TestCase> getTestCases(@RequestBody Request request)
```

Die Schnittsstelle *getTestCases* lädt die Zustandsmaschine aus einer SCXML Datei. Aus dieser Zustandsmaschine werden anschließend die Testfälle mit der Komponente TestOracle erzeugt und im Response Body zurückgegeben. Der Pfad zur SCXML Datei muss bei der Anfrage im Request Body übergeben werden.

```
TestLog runTestCases(@RequestBody TestCase testCase)
```

Zum Ausführen eines Testfalls dient die Schnittstelle *runTestCases*. Der Testfall wird im Request Body übergeben und an die Komponente *TestHarness* weitergegen. Wurde der Test durchgeführt wird das Testprotokoll zurückgegeben.

### 5.3.2 TestOracle

Die Quelle, die das erwarteten Ergebnisses liefert, gegen das ein SUT geprüft wird wird als test Oracle bezeichnet [1]. Die Komponente *TestOracle* erfüllt unter anderem diese Aufgabe. Sie implementiert das Interface *ITestOracle*, das Methoden zur Testfallerzeugung und Testfallverwaltung anbietet.

```
List<TestCase> createTestCases(SCXML sc, Context context);
```

Die Methode *createTestCases* erstellt die Testfälle aus dem übergebenen SCXML. Zum Prüfen der Bedingungen wird der übergebene Kontext genutzt. Der Kontext entspricht dem *data* Tag aus dem SCXML. Die Testfallerzeugung wie in 4 beschrieben ist hier implementiert.

```
void updateTestCase(TestCase testCase);
```

Die Testfälle werden durch einen Hashwert der Zustände, Transitionen und deren Reihenfolge unterschieden. Die Methode *updateTestCase* aktualisiert die zusätzlichen Informationen Label und Inputsequenz des übergebenen Testfalls. Dabei wird der Testfall festgeschrieben. Wird ein neuer Testfall erzeugt, der den selben Hashwert hat, werden die zusätzlichen Informationen dem neuen Testfall hinzugefügt.

### 5.3.3 SimulationAdapter

Die Komponente *SimulationAdapter* bietet einen Service für die Kommunikation mit der in 2.1 beschriebenen HiL-Simulation.

```
SimulationData receive()
```

Mit der Methode *receive* kann über den Statusport der Zustand der Simulation gelesen werden. Dabei wird blockiert, bis der Status gelesen wurde.

```
void send(ControllerMessage message)
```

Mit der in *send* übergebenen Nachricht kann mit der Simulation interagiert werden.

### 5.3.4 TestHarness

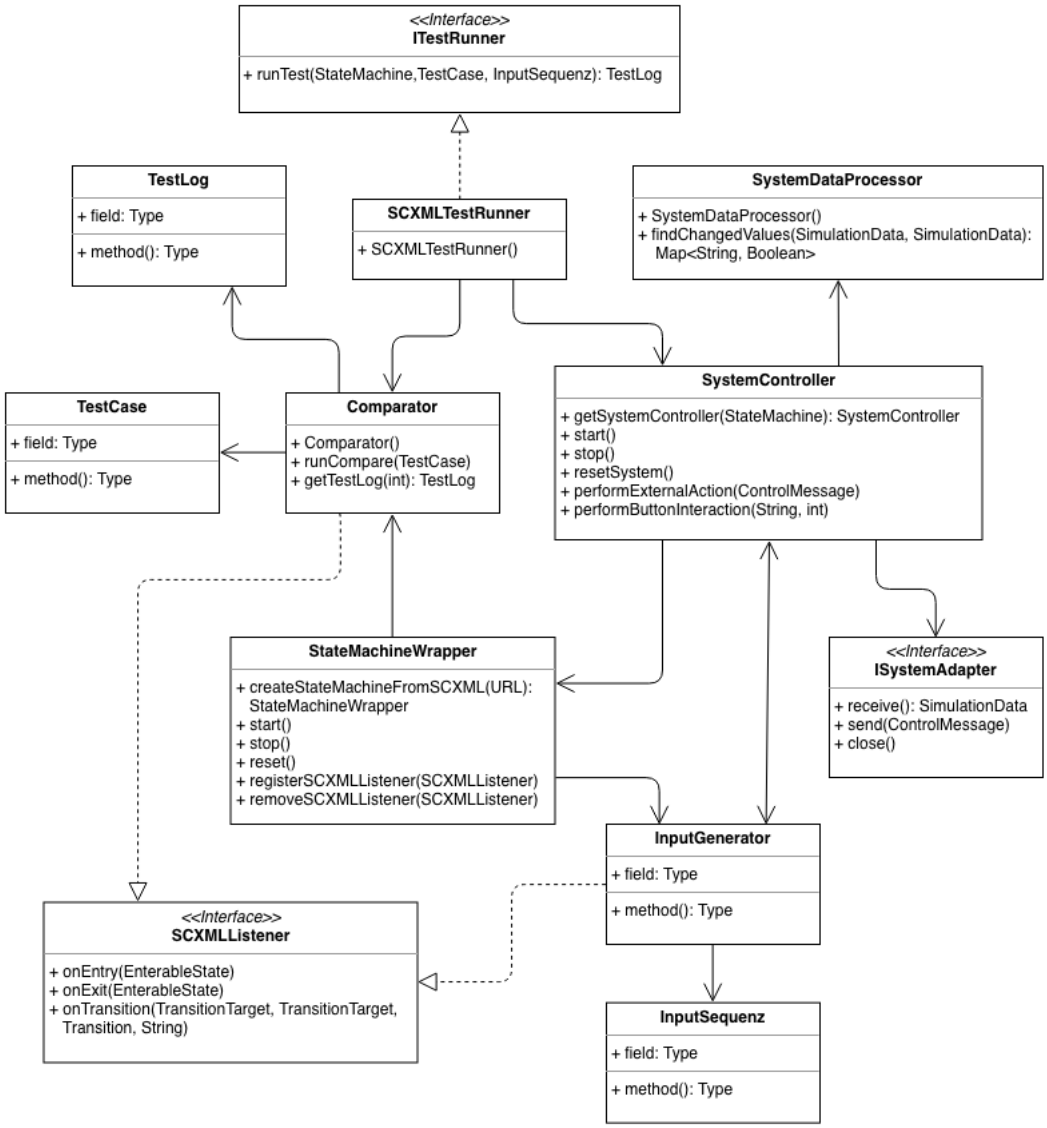


Abbildung 15: Klassendiagram des Test-Harness

Der Test-Harness (Abb. 15) ist für die Testausführung und Auswertung zuständig. Das Interface zur Testausführung (*ITestRunner*) bietet eine Methode an, um einen übergebenen Testfall auszuführen, und gibt das Testprotokoll zurück.

Das Vorgehen bei der Testausführung läuft wie folgt ab:

1. Der *SystemController* baut eine Verbindung zur Simulation auf und setzt diese

in den Ausgangszustand zurück. Anschließend wird die Zustandsmaschine, die im *StateMaschineWrapper* läuft, zurückgesetzt. Im *StateMaschineWrapper* wird die Zustandsmaschine der Anforderungen mit der SCXML Implementation der Apache Foundation [4] ausgeführt. Damit ist der Anfangszustand für die Testausführung hergestellt.

2. Die von der Simulation empfangenen Daten werden im *SystemDataProcessor* aufbereitet, der die zuletzt empfangenen Daten mit den aktuellen vergleicht. Alle Daten, die sich verändert haben, werden zurückgegeben. Der *SystemController* stellt diesen Daten ein *sys_* voran und aktualisiert die Daten im Kontext der FSM. Anschließend werden die Änderungen der Zustandsmaschine wie in 3.3.2 beschrieben als Events mitgeteilt.

3. Der *Comparator* meldet sich als Listener bei der FSM an und prüft bei jeder Zustandsänderung, die ihm von der FSM mitgeteilt wird, ob der neue Zustand und die genommene Transition mit den Vorgaben des Testfalls übereinstimmen. Dabei protokolliert er jeden genommenen Schritt. Wenn der Testfall erfolgreich bis zum Ende ausgeführt wurde, gibt der *Comparator* das Testprotokoll mit dem Flag "erfolgreich" zurück. Im Falle einer Abweichungen vom Testfall wird das Testprotokoll mit dem Flag "nicht erfolgreich" zurückgegeben. In beiden Fällen wird die Testausführung beendet.

4. Der *InputGenerator* meldet sich ebenfalls als Listener bei der FSM an. Wenn eine Transition eine Aktion erfordert, zum Beispiel das Drücken eines Schalters, wird diese Aktion in der Simulation über das Verschicken einer Nachricht, wie in 2.1 beschrieben, ausgelöst. Die Inputsequenz des Testfalls gibt vor, wann Aktionen an Werkstücken vorgenommen werden sollen. Dies übernimmt auch der *InputGenerator*. Dafür wird der Zustand der FSM und die Position von Werkstücken auf dem Laufband verwendet, um die Aktionen zum richtigen Zeitpunkt auszuführen.

# 6 Evaluation

In diesem Kapitel werden die generierten Testfälle betrachtet. Es wird geprüft, in wie weit die Anforderungen von den Testfällen abdeckt werden und ob Fehler im SUT mit den Testfällen und der Testausführung mit der entwickelten Anwendung erkannt werden können.

## 6.1 Die generierten Testfälle

Mit der in 5 beschriebenen Anwendung werden aus dem Modell der Anforderungen die Testfälle erzeugt. Redundante Testfälle werden manuell über das UI aussortiert. Nach der Sortierung bleiben 22 Testfälle, die mit ihren Labeln und den Anforderungen, die sie abdecken, in Tabelle 5 aufgelistet sind.

Mit diesen Testfällen ist jede der in 3.1 aufgelisteten Anforderungen 1 - 4h mit mindestens einem Testfall abgedeckt und damit die Anforderung an die Testabdeckung der Anforderungen aus 4.1 erfüllt. Dabei hat es sich als schwierig herausgestellt, jedem Testfall nur eine Anforderung zuzuweisen. Für die Anforderungen 4a - 4h ist dies möglich, da diese Anforderungen bei der Modellierung getrennt voneinander betrachtet wurden. Die Anforderungen 2 (Fehlerzustand) und 3 (Fehlerbehandlung) folgen auf die Fehlererkennung (4b, 4c, 4f). Daher wird für diese Anforderungen kein eigener Pfad generiert, sondern sie hängen den Pfaden zur Fehlerekennung an.

Zusammenhänge zwischen diesen Anforderungen wurden bei der Modellierung nicht betrachtet und sind daher nicht im Modell der Anforderungen und damit auch nicht durch die Testfälle abgedeckt. Dies betrifft z.B. das Verhalten, dass das Einlegen eines neues Werkstück in die erste Lichtschranke nicht dazu führt, dass das Band gestartet wird, wenn sich ein Werkstück in der letzten Lichtschranke befindet. Dies würde dazu führen, dass das Werkstück, das sich in der letzten Lichtschranke befindet, herunterfällt und damit die Anforderung 4g nicht erfüllt wäre.

Durch die Modellierung mit den zwei Zuständen *empty_ ** und *not_ empty_ ** sind zwei zusätzliche Testfälle (21, 22) entstanden, die nicht explizit in den Anforderungen (3.1) stehen. Diese Testfälle prüfen, ob der Betrieb auch funktioniert, wenn sich zwei Werkstücke auf dem ersten oder letzten Abschnitt des Bandes befinden. Falls diese Testfälle nicht erwünscht sind, können sie über das UI aussortiert werden. In diesen Fall ist es

jedoch besser, das Modell anzupassen, sodass diese beiden Testfälle nicht erzeugt werden und somit die Anforderungen korrekt abbilden.

| Testfälle | | |
|---|---|---|
| Nr. | Label | Anforderungen |
| 1 | Stoppen des Betriebs | 1 |
| 2 | Stoppen des Betriebs während der Höhenmessung | 1 |
| 3 | Starten des Laufbandes, wenn ein Werkstück auf das leere Band eingelegt wurde | 1, 4, 4a |
| 4 | Aufheben des Bandstops nach Unterbrechen der letzten Lichtschranke | 4 |
| 5 | Unzulässiges Hinzufügen eines zweiten Werkstücks auf dem ersten Abschnitt | 2, 3, 4b |
| 6 | Unzulässiges Hinzufügen eines zweiten Werkstücks auf dem letzten Abschnitt | 2, 3, 4b |
| 7 | Unzulässiges Hinzufügen eines Werkstücks auf dem ersten Abschnitt (leer) | 2, 3, 4b |
| 8 | Entfernen eines Werkstücks auf dem ersten Abschnitt | 2, 3, 4c |
| 9 | Entfernen eines Werkstücks auf dem letzten Abschnitt | 2, 3, 4c |
| 10 | Während der Höhenmessung auf langsame Bandgeschwindigkeit stellen | 4d |
| 11 | Sortieren eines Werkstücks mit Bohrung oben | 4e |
| 12 | Sortieren des Werkstücks CODED_2 | 4e |
| 13 | Sortieren des Werkstücks Metall | 4e |
| 14 | Sortieren eines Werkstücks mit Bohrung unten | 4e |
| 15 | Sortieren eines flachen Werkstücks | 4e |
| 16 | Sortieren des Werkstücks CODED_1 | 4e |
| 17 | Erkennen von fehlerhaften Werkstücken | 2, 3, 4f |
| 18 | Anhalten bei einer Unterbrechung der letzten Lichtschranke | 4g |
| 19 | Aufheben des Bandstops nach Unterbrechen der letzten Lichtschranke | 4g |
| 20 | Wiederaufnahme des Betriebs bei einer vollen Rampe | 4h |
| 21 | Verarbeiten von zwei Werkstücken auf dem ersten Abschnitt | - |
| 22 | Verarbeiten von zwei Werkstücken auf dem letzten Abschnitt | - |

Tabelle 5: Testfälle nach der Aussortierung einiger Testfälle, die Anforderungen prüfen, die bereits durch andere Testfälle abgedeckt sind, um die Auswertung in dieser Arbeit übersichtlicher zu gestalten.

## 6.2 Fallstudien

Eine Beispiel-Implementation, die alle hier aufgelisteten Anforderungen erfüllt, wird als IUT (implementation under test) in der HiL-Simulation gestartet und die Testfälle mit der Anwendung dagegen ausgeführt. Alle Testfälle werden dabei erfolgreich ausgeführt. Um eine Aussage über die Qualität der Testfälle treffen zu können wird die IUT jeweils so verändert, dass eine Anforderung nicht erfüllt ist. Die Testfälle werden gegen die veränderte Implementation ausgeführt und geprüft, ob der Fehler erkannt wird und ob die Ursache aus dem Testprotokoll ersichtlich wird. Im Folgenden werden exemplarisch einige dieser Fälle betrachtet.

Anforderung 1

Die IUT wird so verändert, dass beim Betätigen der Stop-Taste nicht reagiert wird. Die Testfälle werden gegen das veränderte IUT ausgeführt. Dabei schlagen die Testfälle 1 und 2 fehl. Da beide Testfälle fehlschlagen, in denen die Stop-Taste betätigt wird, ist das Problem bereits an den Testfällen zu erkennen. Das Testprotokoll zeigt, das vom Zustand *initialise_ stop* durch einen Timeout in den Zustand *error* gewechselt worden ist. Dabei wird im Testprotokoll durch drei Sterne vor dem Zustand, aus dem eine Transition genommen wurde, der Ablauf gekennzeichnet, der sich nicht mit dem Testfall deckt. Also ist nach der Analyse des Testprotokolls anzunehmen, dass das SUT zu spät oder gar nicht auf das Betätigen der Stop-Taste reagiert.

stopped –ev_start_on[]–> wait_for_operating –ev_green_on–> operating —-> section_begin —-> empty_begin –ev_lbBegin_on[]–> not_empty_begin –ev_stop_on–> ***initialise_stop –timeout_initialise_stop–> error

Anforderung 4c

Die IUT wird so verändert, dass das Entfernen eines Werkstücks auf dem letzten Abschnitt nicht erkannt wird. Bei der Ausführung der Testfälle schlägt nur der Testfall 9 (Entfernen eines Werkstücks auf dem letzten Abschnitt) fehl. Bereits an den Testfällen ist in diesem Fall eindeutig zu erkennen, dass das Verschwinden eines Werkstücks auf dem letzten Abschnitt nicht erkannt wird.

# 7 Schluss

In diesem Kapitel gibt es zunächst eine Zusammenfassung der gesamten Arbeit. Anschließend wird im Fazit eine abschließende Bewertung gegeben. Der Abschnitt Ausblick soll zudem weiterführende Fragestellungen aufzeigen, die sich im Zusammenhang mit dieser Arbeit ergeben haben.

## 7.1 Zusammenfassung

In dieser Arbeit wurde ein Einblick in den Bereich des Model Based Testing gegeben. Anhand des Fallbeispiels der Entwicklung eines Steuerungssystems für eine industrielle Sortieranlage, dem Festo-Transfersystem, wurde ein mögliches Vorgehen für das MBT beispielhaft vorgestellt, um das System auf die Erfüllung der Anforderungen (3.1) zu testen.

Dazu wurden im ersten Schritt mögliche Modellierungen betrachtet und anschließend ein Modell der Anforderungen als Statechart umgesetzt. Das Modell muss eine ausreichende Qualität erreichen, um daraus Testfälle generieren zu können (3.2). Um sicherzustellen, dass dieses Modell die benötigte Qualität erreicht wurde ein iteratives Vorgehen genutzt, um das Modell zu entwickeln (3.3.1).

Um aus dem Modell Testfälle ableiten zu können, wurde ein für dieses Beispiel passender Algorithmus gesucht, der anschließend erweitert und angepasst wurde, um die Anforderungen an die erzeugten Testfälle zu erfüllen. Der Algorithmus baut einen Übergangsbaum aus dem Statechart im Breitendurchlauf auf und beachtet dabei vorher definierte Bedingungen, die notwendig sind, damit sinnvolle Testfälle entstehen, die den einzelnen Anforderungen, gegen die geprüft werden soll zuzuordnen sind. Ein Pfad vom Wurzelknoten bis zu einen Blatt stellt dabei einen Testfall dar.

Zuletzt wurde der Prototyp einer Anwendung konzipiert und entwickelt. Diese implementiert den Algorithmus zur Testfallgenerierung und führt die erzeugten Testfälle gegen eine HiL-Simulation des Festo-Transfersystem aus. Die Testergebnisse für Implementationen des zu testenden Systems, die verschiedene Fehler aufweisen wurden analysiert und daraufhin untersucht, ob die Fehler erkannt werden. Damit wurde gezeigt, dass die Anforderung sinnvoll vom Modell abgebildet werden und dass der Algorithmus zur Testfallerzeugung für dieses Beispiel funktioniert.

## 7.2 Fazit

Das Ziel dieser Arbeit lag in der Entwicklung eines Modells, dass die Anforderungen an ein Eingebettetes System modelliert, sodass aus diesem Modell ausführbare Testfälle generiert werden können. Die Modellierung mit dem TD-MBSE Ansatz, beschrieben in Kapitel 3.3.1, hat ein strukturiertes Vorgehen ermöglicht. Dabei wurde das Anforderungsmodell parallel zum Verhaltens- und Systemmodell entwickelt. Das Ausführen von Testfällen, die den Ablauf zwischen den Modellen prüfen, hat sichergestellt, dass beide Modelle ausführbar und korrekt sind. Dieses Vorgehen hatte zum Vorteil, dass das Modell der Anforderungen unabhängig von der MBT-Anwendung entwickelt werden konnte. Damit ist dieses Vorgehen geeignet, während der Planungsphase des Softwareentwicklungsprozesses Modelle zu erstellen, die fürs MBT geeignet sind.

Die Untersuchungen aus Kapitel 6 zeigen, dass mit den generierten Testfällen die Implementation gegen die Anforderungen getestet werden kann. Anforderungen, die nicht oder falsch von der IUT umgesetzt sind, können durch fehlgeschlagenen Testfälle und deren Protokoll identifiziert werden. Damit erfüllen sowohl die Anwendung als auch die Modellierung der Anforderungen in diesem Beispiel ihren Zweck.

Bei der standardmäßigen Simulationsgeschwindigkeit von 1:10 und der Rate von 40ms, in der die Simulation den Systemzustand übermittelt, laufen einige Testfälle nicht stabil. Durch falsches Timing und der Auswertung von mehreren relevanten Änderungen in einer falschen Reihenfolge, die in den Zeitraum der 40ms fallen, schlagen einige Testfälle fehl, obwohl diese korrekt ablaufen. Eine mögliche Lösung für dieses Problem wird in Kapitel 7.3 erläutert.

Die Tools, die zur Testfallgenerierung zur Verfügung stehen, sind Aufgrund der Komplexität und Vielseitigkeit, die von den unterschiedlichen Modellierungsarten ermöglicht werden, entweder sehr spezialisiert oder behandeln nur grundlegende Funktionen. Dementsprechend ist die in dieser Arbeit entwickelte Anwendung ebenfalls auf den hier vorgestellten Anwendungsfall spezialisiert und lässt sich nicht ohne erheblichen Aufwand auf andere Anwendungsfälle übertragen. Dies führt dazu, dass MBT einen hohen initialen Entwicklungsaufwand erfordert. Daher ist dieses Vorgehen nur für große Projekte oder langwierige Entwicklungsprozesse geeignet, in denen sich der Aufwand für das Entwickeln und Einrichten eines solchen Systems relativiert.

## 7.3 Ausblick

Die Anwendung als erster Prototyp mit dem hier vorgestellten Modell funktioniert bereits. Da die Anwendung für diese Modellierung entwickelt wurde, kann untersucht werden, in wie weit sie mit anderen Statechart-Modellen umgehen kann und damit Allgemeingültigkeit besitzt. Der Algorithmus kann angepasst werden, sodass parallele Zustände nicht getrennt betrachtet werden, sondern auch Testfälle generiert werden, die Beziehungen zwischen parallelen Zuständen ermöglichen. Auch die Unterstützung anderer Modellierungsarten wie Aktivitätsdiagramme kann in weiteren Arbeiten entwickelt werden.

Des weiteren kann die Stabilität der Testausführung erhöht werden. Ein Ansatzpunkt dazu ist das Umstellen der Simulation, sodass nicht periodisch der gesamte Systemzustand übermittelt wird sondern nur eine Änderung, sobald sie eintritt. Dies würde es ermöglichen, die Frequenz des Blinkens einer Signalleuchte zu messen. Außerdem gehen Änderungen, die in der 40ms Periode mehrmals überschrieben werden nicht verloren und würden damit die Stabilität erhöhen.

Im Hinblick auf die Laboraufgabe im Rahmen der Veranstalltung ESE an der Hochschule kann das Testen der Implementation mit Modellen als Service angeboten werden. Momentan ist die Anwendung auf einen Nutzer und eine Simulation beschränkt. Über das UI kann dem Nutzer ermöglicht werden ein Modell und eine Implementation zu übermitteln, für die eine eigene Simulation gestartet wird. Mit dieser Erweiterung können mehrere Teams parallel unterschiedliche Implementationen mit eigenen Modellen testen, ohne sich um das Setup kümmern zu müssen.

# Literatur


[1] : *Test oracle.* – URL http://istqb-glossary-explanations.blogspot.com/2016/07/test-oracle.html. – Zugriffsdatum: 2019-08-23

[2] : *UPPAAL.* – URL http://www.uppaal.org. – Zugriffsdatum: 2019-08-11

[3] ISO/IEC/IEEE International Standard - Software and systems engineering – Software testing –Part 1:Concepts and definitions. In: *ISO/IEC/IEEE 29119-1:2013(E)* (2013), Sep., S. 1–64

[4] Apache: *Commons SCXML.* 2015. – URL https://commons.apache.org/proper/commons-scxml/. – Zugriffsdatum: 2019-08-15

[5] Artho, Biere A.: *Modbat.* – URL https://people.kth.se/~artho/modbat/. – Zugriffsdatum: 2019-07-28

[6] Butler, R. W. ; Finelli, G. B.: The infeasibility of quantifying the reliability of life-critical real-time software. In: *IEEE Transactions on Software Engineering* 19 (1993), Jan, Nr. 1, S. 3–12. – ISSN 0098-5589

[7] Chen, Y. ; Wang, A. ; Wang, J. ; Liu, L. ; Song, Y. ; Ha, Q.: Automatic Test Transition Paths Generation Approach from EFSM Using State Tree. In: *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, July 2018, S. 87–93

[8] Clarke, Jim B.: Automated test generation from a behavioral model, 1998

[9] Cliff, S.: *Application testing costs set to rise to 40% of IT budget.* – URL https://www.computerweekly.com/news/4500253336/Application-testing-costs-set-to-rise-to-40-of-IT-budget. – Zugriffsdatum: 2019-08-11

[10] Douglass, Bruce P.: UML statecharts. In: *Embedded systems programming* 12 (1999), Nr. 1, S. 22–42

[11] Drauschke, Thomas: *Hardware in the Loop Simulationssoftware eines Festo-Transfersystems*, HAW Hamburg, Bachelor's Thesis

[12] Fast: Study of worldwide trends and RD programmes in embedded systems in siew of maximising the impact of a technology platform in the area. (2014), Nov.

[13] FESTO: *MPS® Transfersystem*. – URL https://www.festo-didactic.com/de-de/lernsysteme/mechatronische-systeme-mps/mps-transfersystem/?fbid=ZGUuZGUuNTQ0LjEzLjIwLjExMjM. – Zugriffsdatum: 2019-08-11

[14] GRAPHWALKER.ORG: *GraphWalker*. – URL http://graphwalker.github.io. – Zugriffsdatum: 2019-07-28

[15] GROUP, Object M.: *Unified Modeling Language*. – URL https://www.omg.org/spec/UML/. – Zugriffsdatum: 2019-08-09

[16] HAMBLING, Brian ; MORGAN, Peter ; SAMAROO, Angelina ; THOMPSON, Geoff ; WILLIAMS, Peter: *Software Testing: An ISTQB-BCS Certified Tester Foundation guide*. BCS, The Chartered Institute for IT, 2019. – ISBN 1780174926

[17] HAREL, David: Statecharts: A visual formalism for complex systems. In: *Science of computer programming* 8 (1987), Nr. 3, S. 231–274

[18] HAREL, David ; POLITI, Michal: *Modeling Reactive Systems With Statecharts : The Statemate Approach*. McGraw-Hill, 1998. – ISBN 0070262055

[19] ITEMIS: *YAKINDU STATECHART TOOLS*. – URL https://www.itemis.com/en/yakindu/state-machine/. – Zugriffsdatum: 2019-08-11

[20] KRAMER, Anne ; LEGEARD, Bruno: *Model-Based Testing Essentials - Guide to the ISTQB Certified Model-Based Tester: Foundation Level*. Wiley, 2016. – ISBN 1119130018

[21] LIGGESMEYER, Peter: *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. Spektrum Akademischer Verlag, 2002. – ISBN 3827411181

[22] MARKU: *Model-Based Testing with JUnit*. – URL https://sourceforge.net/p/modeljunit/wiki/Home/. – Zugriffsdatum: 2019-07-28

[23] MARTIN, Robert C.: *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002. – ISBN 0135974445

[24] MATHWORKS: *HIL-Simulation (Hardware-in-the-Loop)*. – URL https://de.mathworks.com/discovery/hardware-in-the-loop-hil.html. – Zugriffsdatum: 2019-08-11

[25] Munck, A. ; Madsen, J.: Test-driven modeling of embedded systems. In: *2015 Nordic Circuits and Systems Conference (NORCAS): NORCHIP International Symposium on System-on-Chip (SoC)*, Oct 2015, S. 1–4

[26] Pretschner, A. ; Prenninger, W. ; Wagner, S. ; Kühnel, C. ; Baumgartner, M. ; Sostawa, B. ; Zölch, R. ; Stauner, T.: One Evaluation of Model-based Testing and Its Automation. In: *Proceedings of the 27th International Conference on Software Engineering*. New York, NY, USA : ACM, 2005 (ICSE '05), S. 392–401. – URL http://doi.acm.org/10.1145/1062455.1062529. – ISBN 1-58113-963-2

[27] W3C: *Voice Browser Call Control: CCXML Version 1.0*. 2011. – URL https://www.w3.org/TR/ccxml/. – Zugriffsdatum: 2019-08-15

[28] W3C: *State Chart XML (SCXML): State Machine Notation for Control Abstraction*. 2015. – URL https://www.w3.org/TR/scxml/. – Zugriffsdatum: 2019-08-11

# Abkürzungsverzeichnis

**HiL** Hardware-in-the-Loop

**MBT** Model Based Testing

**SUT** System under Test

**IUT** Implementation under Test

**FSM** Finite State Machine

**UML** Unified Modeling Language

**SCXML** State Chart XML

**TD-MBSE** Test Driven Modelbased System Engineering

**TDD** Test Driven Development

## Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „– bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] – ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen."

*Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI*

## Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: _____

Vorname: _____

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

**Modellbasierte Testfallgenerierung und Ausführung für das Testen eines Echtzeitsystems auf Konformität**

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

| _____ | _____ | _____ |
|---|---|---|
| Ort | Datum | Unterschrift im Original |