UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

s-lab
Software Quality Lab

Dissertation

Automatische Testfallgenerierung aus kontrolliert natürlichsprachlichen Anforderungsspezifikationen für reaktive Echtzeitsysteme

Schriftliche Arbeit
zur Erlangung des Grades
„Doktor der Naturwissenschaften“

vorgelegt von

Matthias Schnelte

Paderborn, März, 2011

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation / Problemstellung

Der Einsatz von Software durchdringt immer mehr unseren Alltag. Manchmal ist ihr Einsatz offensichtlich, z. B. bei der Arbeit am PC. Weit häufiger aber offenbart sich Software erst auf den zweiten Blick. Dies gilt insbesondere für den Bereich der eingebetteten Systeme, in dem die Software Teil eines komplexeren technischen Systems ist. Beispiele hierfür sind Alltagsgegenstände wie Wasch- und Spülmaschinen, Fernsehgeräte und Kraftfahrzeuge.

Dass in diesen Systemen Software zum Einsatz kommt, bleibt dem Benutzer in den meisten Fällen verborgen – zumindest solange sie fehlerfrei funktioniert. Der rasante Anstieg von Einsatz von Software führt jedoch zwangsläufig zu einem höheren Potential für Softwarefehler. Als Beispiel sei hier die Automobilindustrie genannt, auf der der Fokus dieser Arbeit liegt.

Im Laufe der letzten Jahre ist die Anzahl der elektronischen Steuergeräte im PKW kontinuierlich gewachsen. So sind in aktuellen PKW der Oberklasse bis zu 80 Steuergeräte verbaut [BBK98]. Ein Blick in die Rückruf-Datenbank des ADAC [ADAC09] zeigt, dass teure und mit Imageverlusten behaftete Rückrufaktionen nicht selten aufgrund von Softwarefehlern durchgeführt werden müssen.

Um solche Fehler zu verhindern und die Qualität von Software zu gewährleisten, gibt es verschiedene Methoden, wie Codeinspektion, statische Analyse durch Werkzeuge (z.B. Modellprüfung) und *Testen*.

Beim *funktionalen Testen* wird das zu testende System ausgeführt und stichprobenartig mit ausgewählten Eingaben stimuliert. Ziel des Testens ist es, zu überprüfen, ob das System sich so verhält, wie es laut vorher definierter *Anforderungen* gewünscht ist, daher wird funktionales Testen häufig auch als *anforderungsbasiertes* Testen bezeichnet.

Die Anforderungen sind in der industriellen Praxis typischerweise *informell* in *textueller Form* beschrieben. Beispiele für Anforderungen aus der Praxis sind in den Abbildungen 1 und 2 zu sehen, die aus Anforderungsdokumenten aus der Automobilindustrie stammen. In diesen Anforderungen ist unter anderem zu sehen, dass hierbei *Zeitbedingungen* eine wichtige Rolle spielen.

In der Automobilindustrie ist das Testen die verbreitetste Methode der Qualitätssicherung. Die Kosten für das Entwickeln und Durchführen von Tests machen hierbei einen Großteil der gesamten Entwicklungskosten aus. Um dieses Kosten zu senken, sind Verfahren zur Automatisierung der Testdurchführung und Testentwicklung von großem Interesse.

Während im Zusammenhang mit der Testdurchführung und Auswertung bereits ein sehr hoher Automatisierungsgrad erreicht wurde, indem zum Beispiel Methoden wie Hardware-In-The-Loop-Simulation eingesetzt werden, erfolgt die Spezifikation und Implementierung der Tests noch weitgehend manuell, basierend auf den Anforderungsdokumenten der Elektroniksysteme. In dieser Arbeit soll untersucht werden, wie dieser Prozess verbessert

werden kann.

Scheinwerferreinigung

Funktion:

Bei eingeschaltetem Standlicht wird beim Betätigen des Tasters Scheinwerferreinigung (SWR) die Pumpe der Scheinwerferreinigungsanlage für eine Zeit von 800ms angesteuert. Die Dauer der Pumpenansteuerung ist über EOL applizierbar zu gestalten.

Die Ansteuerung der Pumpe ist nur nach der Beendigung eines Einschaltzyklus möglich (Funktion nicht retriggerbar), und falls der Flüssigkeitsbehälter der Scheinwerferreinigungsanlage nicht leer ist.

Abbildung 1: Anforderung für eine Scheinwerferreinigung

Aktivierung des Automatikbetriebs bei Tag-Nacht-Wechsel mit LS/RLS

Anf-123 Wenn der Lichtsensor einen Wechsel von Tag zu Nacht meldet, wird der Automatikbetrieb aktiviert.

Vorbedingungen: - Normalbetrieb aktiv.
- Tag

Trigger: Wechsel von Tag nach Nacht.

Suchbeleuchtung bei Entriegelung

Anf-456 Das Entriegeln des Fahrzeugs bei geschlossenen Türen bewirkt das Einschalten der Innenleuchten.

Vorbedingungen: - Türen vorn und hinten geschlossen.
- Automatikbetrieb aktiv.
- Türen vorn und hinten geschlossen.

Trigger: Entriegelung per Funk

Abbildung 2: Anforderungen für eine Lichtsteuerung

1.2 Zielsetzung

Eine wichtige Rahmenbedingung für die zu entwickelnde Lösung ist, dass sie sich möglichst nahtlos in bereits bestehende Arbeitsprozesse einfügt. Ein Teil dieser bestehenden Arbeitsprozesse ist es, dass Tests auf Grundlage von Anforderungsdokumenten entwickelt werden, die typischerweise in informeller Textform vorliegen, wie bereits dargelegt worden ist.

Ein in der aktuellen wissenschaftlichen Diskussion sehr oft beschriebener Ansatz zur systematischen Testentwicklung ist der Ansatz des *modellbasierten Testens,* der in Abbildung 3 vereinfacht dargestellt ist. Hierbei wird ausgehend von den informellen Anforderungen ein formales Modell des zu testenden Systems manuell erstellt. Dieses formale Modell ist ein abstraktes Modell der Anforderungen an das System und beschreibt somit das ge-

wünschte Verhalten. Anschließend werden durch einen Testfallgenerator automatisch Testfälle aus diesem Modell hergeleitet.

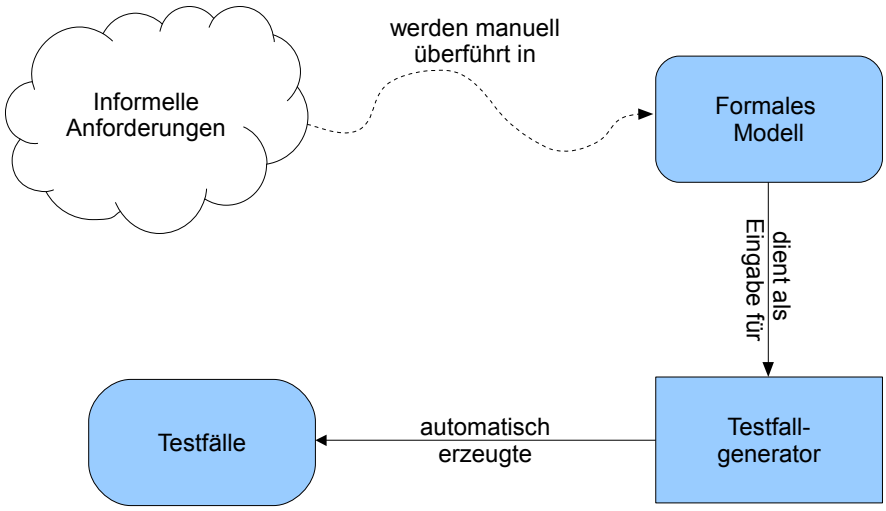


Abbildung 3: Ablauf des modellbasierten Testens

Die Notationen des formalen Modells sind jedoch weit entfernt von den ursprünglichen, in natürlicher Sprache verfassten Anforderungen, da diese Modelle in formalen Sprachen wie z. B. der Sprache Z oder durch graphische Notationen wie Timed Automata erfasst werden. Der Nachteil von solch einer formalen Repräsentation ist, dass hierfür erst die Notation der formalen Beschreibungssprache erlernt werden muss und die formale Notation sich sehr von den ursprünglich textuell formulierten Anforderungen entfernt. Es ist somit gerade für Personen, die keine Kenntnisse von formalen Methoden haben, schwierig nachzuvollziehen, ob die formale Repräsentation noch den originalen Anforderungen entspricht.

Typischerweise sind gerade bei der Erfassung von Anforderungen viele Fachexperten aus verschiedenen Bereichen beteiligt. Diese Experten haben häufig unterschiedliche Ausbildungen genossen und in nur wenigen Fällen Kenntnisse von formalen Methoden, wie sie in der Informatik entwickelt worden sind. Die Zielsetzung dieser Arbeit ist es daher, die Lücke zwischen den informellen, natürlichsprachlich verfassten Anforderungen und dem formalen Modell, das zur Testfallgenerierung dient, zu verkleinern, wobei der Fokus hierbei insbesondere auf Anforderungsdokumenten aus der Automobilindustrie liegt.

1.3 Lösungsansatz

Um die Lücke zwischen der informellen Beschreibung und dem formalen Modell zu verkleinern, wird in dieser Arbeit eine *kontrollierte natürliche Sprache* entwickelt, um die Anforderungen formal zu erfassen.

Eine kontrollierte natürliche Sprache ist eine eingeschränkte Version einer natürlichen Sprache, die semantisch eindeutig ist; jedoch gleichzeitig natürlich sprachlich erscheint. Da diese Sprache an eine natürliche Sprache angelehnt ist, ist sie ein geeignetes Mittel, um Informationen zwischen den verschiedenen beteiligten Fachexperten austauschen zu können. Die Sprache ist aufgrund von vorgenommen Einschränkungen jedoch formal genug, um ein-

deutig auf ein formales Modell abgebildet zu werden. Aufgrund der letztgenannten Eigenschaft kann solch eine Sprache als Basis zur Testfallgenerierung dienen.

Die in kontrollierter Sprache verfassten Anforderungen können also automatisch in ein eindeutiges formales Modell der Anforderungen, welches als Basis für die Testfallgenerierung dient, übersetzt werden. Die Fragestellung, die sich dabei ergibt, ist, welche Arten von Testfällen aus dem Modell hergeleitet werden sollen. Da die Menge der möglichen Testfälle unendlich groß ist, besteht eine Herausforderung bei der automatischen Testfallgenerierung darin, *Testauswahlkriterien* zu bestimmen, die aus der Menge der möglichen Testfälle, eine Teilmenge an sinnvollen Testfällen auszuwählen.

Bei dieser Fragestellung orientiert sich die in dieser Arbeit entwickelte Lösung an der Art, wie heutzutage in der Praxis Testfälle aus Anforderungen abgeleitet werden. So können für jede erfasste Anforderung sowohl *Positiv*- als auch *Negativtests* generiert werden.

Bei einem Positivtest wird das zu testende System in einen Zustand gebracht, der laut Anforderungen dazu geeignet ist, die zu testende Funktion auszuführen. Danach wird das Ereignis ausgelöst, das laut den Anforderungen die zu testende Funktion zur Ausführung bringt. Im letzten Schritt wird schließlich die beobachtete Reaktion des Systems mit der laut Anforderungen zu erwartenden Reaktion verglichen.

Negativtests unterscheiden sich von Positivtests dadurch, dass das System bewusst in einen Zustand gebracht wird, der die in den Anforderungen beschriebenen Vorbedingungen für eine Funktion *nicht* erfüllt. Danach wird überprüft, ob das auslösende Ereignis die erwartete Reaktion nicht hervorruft.

Beiden Testarten ist gemein, dass der Testfall das System von einem definierten Anfangszustand in einen bestimmten Zustand bringen muss. Die Information, auf welche Arten ein gewünschter Systemzustand eingestellt werden kann, ist in den Anforderungen beschrieben. So wird zum Beispiel in der Anforderung *Anf-123* aus Abbildung 2 erläutert, wie der Automatikbetrieb zu aktivieren ist, was wiederum Teil der Vorbedingungen von Anforderung *Anf-456* ist. Ein Testentwickler kann also aus den Anforderungen ableiten, auf welchem Wege er die Vorbedingungen für eine Anforderung herstellen kann. Eine automatische Testfallgenerierung muss ebenfalls in der Lage sein, diese Zusammenhänge aus den Anforderungen herzuleiten.

Das modellbasierte Testen leistet diese Aufgabe, indem der Suchraum, den das formale Modell beschreibt, nach Pfaden durchsucht wird, die den gewünschten Zustand einstellen. Hierfür müssen zwei Fragestellungen beantwortet werden:

1. Welche formale Repräsentation wird gewählt, um die Anforderungen abzubilden?
2. Wie kann die Suche effizient gestaltet werden?

Für beide Fragestellungen wird in dieser Arbeit auf Lösungen aus dem Bereich der *automatischen Handlungsplanung* – eine Problemstellung der Informatik aus dem Gebiet der künstlichen Intelligenz – zurückgegriffen.

Der Vorteil der Verwendung von Lösungen aus diesem Gebiet ist es, dass es für beide Fragestellungen Lösungen anbietet: Es existieren Formalisierungen, die ausdrucksstark genug sind, um die Anforderungen darzustellen. Insbesondere sind diese Formalisierungen in der Lage die *Echtzeiteigenschaften* der Anforderungen darzustellen. Darüber hinaus bietet

dieses Gebiet Lösungen, die in der Lage sind auch große Suchräume, wie sie durch Anforderungsdokumente aus der Praxis beschrieben werden, zu durchsuchen.

1.4 Beitrag

Diese Arbeit leistet in zweierlei Hinsicht einen Beitrag zum Stand der Forschung im Bereich der Sicherstellung von Softwarequalität. Der erste Beitrag ist in der Methode zu sehen, in der die Spezifikation des Verhaltens des zu testenden Systems erfasst wird. Durch die Verwendung einer kontrollierten natürlichen Sprache passt sich die Methode in bestehende Arbeitsprozesse ein, in der Anforderungen überwiegend in Form von natürlicher Sprache formuliert werden.

Der zweite Beitrag ist in dem Verfahren der Testfallgenerierung zu sehen. Diese Arbeit verwendet hierfür automatische Handlungsplanung. Diese Methode ist bisher nicht für Testfallgenerierung im Kontext von Echtzeitsystemen zum Einsatz gekommen. Die Tragfähigkeit des vorgestellten Ansatzes wird durch Evaluierung mit Hilfe von industriellen Fallstudien gezeigt.

1.5 Aufbau der Arbeit

Zu Beginn der Arbeit werden in Kapitel 2 die Grundlagen für das Verständnis der Arbeit gelegt. Hierfür werden reaktive Echtzeitsysteme, modellbasiertes Testen, automatische Handlungsplanung, Verarbeitung von natürlicher Sprache sowie die Grundlagen für kontrollierte natürliche Sprachen eingeführt.

Im darauf folgenden Kapitel wird auf den Stand der Forschung, der für diese Arbeit relevant ist, eingegangen. Das Kapitel befasst sich mit bestehenden kontrollierten natürlichen Sprachen und gibt schließlich einen Überblick über modellbasiertes Testen von Echtzeitsystemen.

Der Hauptteil der Arbeit findet sich in Kapitel 4, in dem das entwickelte Lösungskonzept vorgestellt wird. Das Kapitel führt zunächst ein Beispiel für ein Anforderungsdokument aus der Praxis ein, welches als durchgehendes Anwendungsbeispiel im gesamten Kapitel genutzt wird. Anschließend wird die entwickelte kontrollierte natürliche Sprache vorgestellt, wobei die einzelnen Elemente mit Beispielen aus dem Anwendungsbeispiel erläutert werden. Anschließend wird darauf eingegangen, wie aus dem formalen Modell Testfälle automatisch erzeugt werden.

Im Kapitel 5 wird das vorgestellte Lösungskonzept anhand mehrerer Fallstudien evaluiert und es wird gezeigt, dass das entwickelte Konzept für Anforderungen aus der industriellen Praxis tragfähig ist. Die Arbeit schließt mit einer Zusammenfassung und Ausblick.

2 Grundlagen

2.1 Reaktive Echtzeitsysteme

Der Begriff *reaktives System* ist durch Harel und Pnueli 1985 geprägt worden [HP85]. Ein reaktives System wird dadurch charakterisiert, dass es in *ständiger* Interaktion zu seiner Umgebung steht: es reagiert auf Eingaben aus der Umgebung, verarbeitet diese und liefert eine Ausgabe an die Umgebung zurück.

Als Gegenstück zu reaktiven Systemen werden in [HP85] *transformale* Systeme genannt. Diese Systeme verarbeiten einmalig eine Menge von Eingaben, indem sie diese in eine Menge von Ausgaben transformieren. Der Aspekt der ständigen Interaktion mit der Umgebung spielt dabei keine Rolle. Als Beispiel für transformale Systeme können Compiler genannt werden.

Reaktive Systeme sind demnach eng in ihre Umgebung eingebettet und müssen auf Eingaben aus der Umgebung reagieren. Häufig ist es wichtig, dass diese Reaktion innerhalb gewisser Zeitvorgaben geschieht, da zum Beispiel Sensordaten nur für einen gewissen Zeitraum gültig sind, weil sich der Zustand der Umgebung mit der Zeit ändert.

Ein System, welches innerhalb eines gewissen Zeitraums auf externe Ereignisse reagieren und Ergebnisse liefern muss, da diese Ergebnisse sonst wertlos sind, wird als *Echtzeitsystem* bezeichnet. In einem Echtzeitsystem hängt die Korrektheit eines Ergebnisses also nicht nur von dem logischen Ergebnis einer Berechnung ab, sondern auch von dem Zeitpunkt, zu dem dieses Ergebnis geliefert wird.

2.2 Testen

Testen ist Teil der Qualitätssicherung im Softwareentwicklungsprozess und dient dazu, das Vertrauen in die Qualität der Software zu erhöhen. Hierzu wird das Testobjekt durch stichprobenhaftes Ausführen überprüft. Während der Ausführung wird das Testobjekt mit Eingaben stimuliert und es wird überprüft, ob die Ausgaben des Testobjekts den erwarteten Ausgaben entsprechen. Solch eine Sequenz von Eingaben und den erwarteten Ausgaben wird als *Testfall* bezeichnet und eine Menge von Testfällen wird als *Testsuite* bezeichnet.

Um Testfälle zu erstellen, gibt es verschiedene Verfahren, die sich in zwei Kategorien einteilen lassen: *Blackbox*- und *Whitebox*-Verfahren. Bei Whitebox-Verfahren werden Testfälle aus Kenntnissen über die innere Struktur der Software und Implementierungsdetails hergeleitet. Bei Blackbox-Verfahren werden Testfälle aus den Anforderungsdokumenten, die die SOLL-Funktionalität des Testobjektes beschreiben, hergeleitet. Da bei Blackbox-Testen die Funktionalität des Testobjektes im Vordergrund steht, werden diese Verfahren häufig als *funktionale* oder *anforderungsbasierte* Testverfahren bezeichnet [SL03]. Die Menge aller Testfälle, die durchgeführt werden sollen, werden in einer *Testspezifikation* beschrieben.

In einigen Fällen können die Anforderungsdokumente *nichtdeterministisches* Verhalten beschreiben. Hierbei müssen zwei verschiedene Arten von nichtdeterministischen Anforderungen unterschieden werden. Die erste Art sind Anforderungen, bei den auf eine Eingabe verschiedene Ausgaben erfolgen können. Diese Art von Nichtdeterminismus kann zum Beispiel durch eine Anforderung der Form „*wenn Eingabe a empfangen wird, wird Ausgabe b oder c ausgegeben*“ erzeugt werden. Falls eine Spezifikation keinen Nichtdeterminismus dieser Art enthält, so spricht man davon, dass die Ausgaben *isoliert* (engl.: *isolated*) sind.

Die zweite Art von Nichtdeterminismus kann in Anforderungen an Echtzeitsystemen entstehen, indem Anforderungen der Form „*wenn Eingabe a empfangen wird, wird Ausgabe b nach nicht mehr als 10 Zeiteinheiten ausgegeben*“ verwendet werden. Ausgaben deren Zeitverhalten eindeutig sind (z. B. „Ausgabe b nach exakt 5 Zeiteinheiten“) werden als *dringend* (engl.: *urgent*) bezeichnet [KT04].

Die Konsequenz des Nichtdeterminismus ist es, dass die Testfälle nicht mehr als Sequenz von Testschritten dargestellt werden können, sondern dass ein Testfall durch einen Baum, der alle möglichen Abläufe enthält, dargestellt werden muss. Zur Ausführungszeit des Testfalles werden je nach Ausgabe- und Zeitverhalten des Testobjektes die passenden Pfade des Baumes ausgewählt. Solche Testfälle werden als *adaptive Testfälle* bezeichnet.

2.3 Modellbasiertes Testen

Ein systematisches Verfahren, welches die Testspezifikation aus einem Modell des Testobjektes herleitet, ist das *modellbasierte Testen*, das jetzt vorgestellt wird. Die Idee des modellbasierten Testens ist es, ein Modell des Testobjekts zur automatischen Generierung von Testfällen zu verwenden. Anstatt Hunderte von Testfällen per Hand zu schreiben, wird ein abstraktes Modell des Testobjekts entworfen, aus dem dann ein modellbasiertes Testwerkzeug eine Menge von Testfällen automatisch generiert.

Der Prozess des modellbasierten Testens wird in [UL06] in die folgenden fünf Schritte unterteilt (s. Abbildung 4):

1. Erstellen eines Modells des Testobjekts und/oder seiner Umgebung.
2. Generierung von abstrakten Testfällen aus dem Modell.
3. Konkretisierung der abstrakten Testfälle, um sicherzustellen, dass diese ausführbar sind.
4. Ausführen der Testfälle und jeden Testdurchlauf mit einem Urteil versehen, ob der Test bestanden worden ist oder nicht.
5. Analyse der Testresultate.

In Schritt 1 des Prozesses erstellt der Testdesigner aus den informellen Anforderungen ein Modell des Testobjekts. Für die Modellierung solch eines Modells gibt es verschiedene Möglichkeiten, wie zum Beispiel UML [UML], Timed Automata [BY03] oder Pre/Post basierte Notationen wie B [Schneide02] oder Z [Potter91].

In Schritt 2 wird aus dem Modell eine Menge von Testfällen mit Hilfe von Werkzeugen automatisch generiert. Die Frage, die sich dabei stellt, ist, nach welchen Kriterien diese

Menge von Testfällen generiert wird. Um diese Menge näher zu definieren, werden *Testauswahlkriterien* verwendet. Auf diese wird in Abschnitt 2.3.1 näher eingegangen.

Schritt 3 wird benötigt, um die abstrakten Testfälle durch ein Testwerkzeug ausführen zu können. Hierfür kommt ein Testskriptgenerator zum Einsatz, der die abstrakten Testfälle in eine Eingabe umwandelt, die vom Testwerkzeug verarbeitet werden kann. Weiterhin wird ein Testadapter erstellt, welcher jede abstrakte Operation, die in den Testfällen verwendet wird, auf eine oder mehrere konkrete Operationen des Testobjekts umsetzt. Dieser Adapter wird manuell erstellt.

In Schritt 4 werden die Tests ausgeführt und in Schritt 5 werden die Testresultate durch den Testdesigner analysiert.

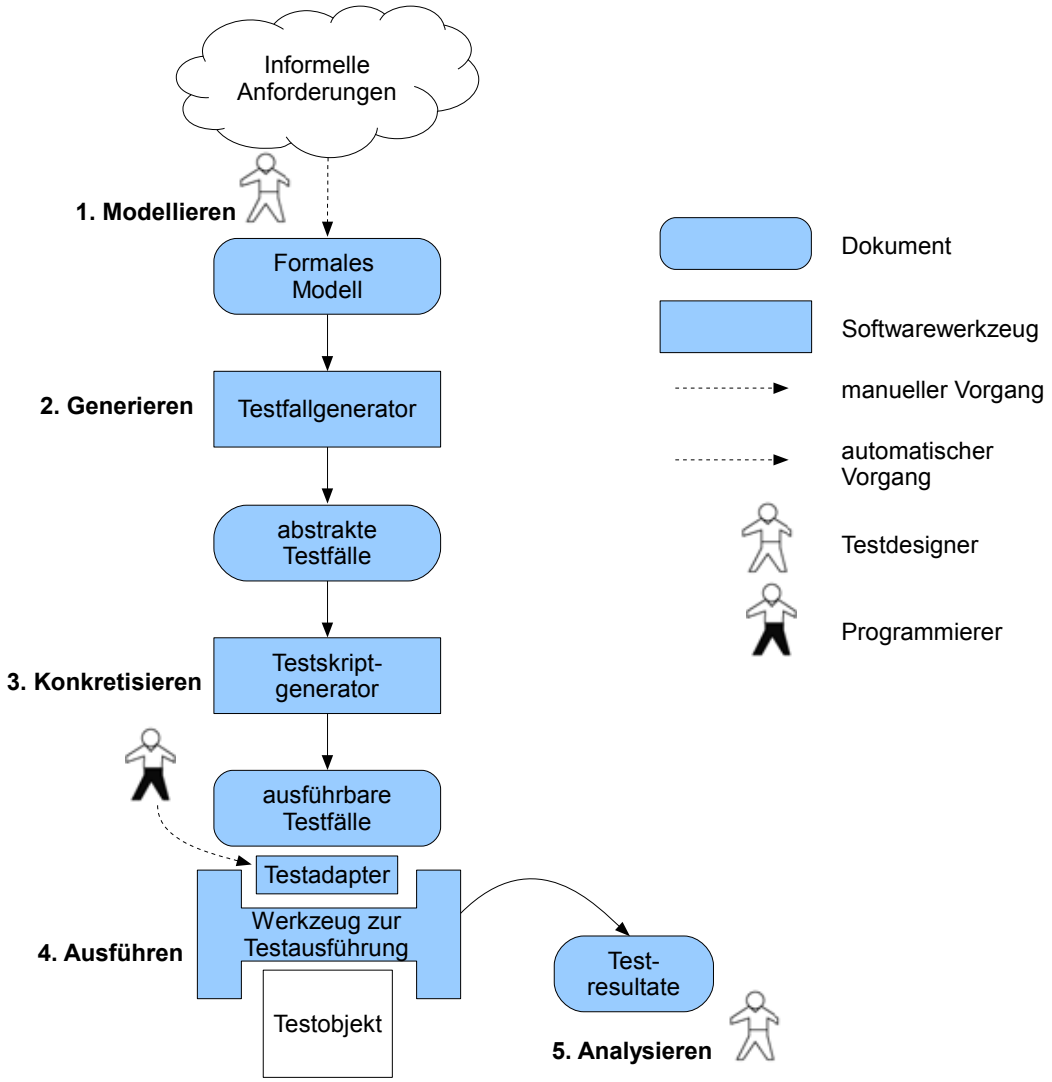


Abbildung 4: Prozess des modellbasierten Testens

2.3.1 Testauswahlkriterien

Um die Menge von Testfällen, die bei der Testfallgenerierung erzeugt wird, sinnvoll einzuschränken, werden sogenannte *Testauswahlkriterien* verwendet. Es gibt verschiedene Arten von Kriterien, die teilweise an die Art der Modellierung des Modells gebunden sind. In [UL06] werden diese in sechs verschiedene Gruppen aufgeteilt:

- Bei der **strukturellen Modellüberdeckung** werden die Testfälle derart gestaltet, dass die Menge der generierten Testfälle eine möglichst hohe Überdeckung gewisser Strukturen des Modells erreicht.
- Die **Datenüberdeckung** misst, in welchem Maße die möglichen Eingabewerte einer Operation oder einer Transition des Modells überdeckt werden. Wird Datenüberdeckung als Testauswahlkriterium verwendet, so gilt, dass die Menge der Testfälle eine möglichst hohe Überdeckung erreichen soll.
- Um eine Menge von Testfällen zu generieren, die in der Lage sind, bestimmte Arten von Fehlern zu finden, werden **Fehlermodelle** verwendet.
- **Anforderungsbasierte Kriterien** werden verwendet, um eine Menge von Testfällen zu generieren, die sicherstellt, dass alle informellen Anforderungen getestet werden. Hierzu ist es nötig, dass zwischen den Anforderungen und dem Modell Beziehungen hergestellt werden.
- In einigen Fällen ist es wünschenswert, dass die generierten Testfälle einen bestimmten Testfall enthalten. In solchen Fällen können **explizite Testfallspezifikationen** vom Testdesigner in einer formalen Notation angegeben werden.
- Zufallsbasierte Testfallgenerierung ist eine einfache Möglichkeit eine Menge von Testfällen zu generieren, die viele Bereiche der Funktionalität des Testobjektes überprüft.

Auf einige der genannten Kriterien wird nun näher eingegangen, da sie im weiteren Verlauf dieser Arbeit Verwendung finden.

2.3.1.1 Strukturelle Modellüberdeckung

Zur strukturellen Modellüberdeckung zählen unter anderen kontrollflussorientierte Kriterien, die ihren Ursprung in der Codeüberdeckung haben. Bei dieser Art der Überdeckungskriterien wird analysiert, inwieweit Anweisungen, Entscheidungen oder Pfade überdeckt werden.

Ähnliche Konzepte lassen sich bei allen Modellierungsarten anwenden, in denen Bedingungen Teil der Modellierung sind. Dies ist zum Beispiel bei den Pre/Postconditions von UML/OCL oder bei B der Fall. Im Folgenden wird nun ein Überblick über die am häufigsten verwendeten kontrollflussorientierten strukturellen Überdeckungskriterien gegeben.

Hierzu müssen zuerst einige Begriffe eingeführt werden. Hier werden nicht die Begriffe aus der Originaldefinition des MC/DC Kriteriums [CM94] verwendet, sondern die Begriffe aus [AOH03]. Der Grund hierfür ist, dass die Originalbegriffe mit anderen Begriffen kollidieren, die im weiteren Verlauf dieser Arbeit verwendet werden. Weiterhin beseitigen die Definitionen aus [AOH03] einige Unklarheiten der Originaldefinition. Die häufiger anzu-

treffenden Begrifflichkeiten aus der Originaldefinition werden zum besseren Verständnis in Klammern aufgeführt.

Ein *Prädikat (Entscheidung)* ist ein Ausdruck, der zu *wahr* oder *falsch* ausgewertet werden kann. Ein Prädikat besteht aus einer oder mehreren *Klauseln (Bedingungen)*, die durch bool'sche Operatoren, wie $\wedge$, $\vee$ oder $\neg$ verknüpft werden. Zum Beispiel enthält das Prädikat $(a > b) \wedge C$ zwei Klauseln: den relationalen Ausdruck $a > b$ und die boolesche Variable C.

Basierend auf diesen Begriffen lassen sich nun verschiedene Überdeckungsmaße definieren:

- **Klauselüberdeckung (Bedingungsüberdeckung)**: Eine Menge von Testfällen erreicht eine vollständige Klauselüberdeckung, genau dann, wenn jede Klausel im Modell mindestens einmal während eines Testdurchlaufes zu wahr und einmal zu falsch ausgewertet wird.
- **Prädikatüberdeckung (Entscheidungsüberdeckung)**: Eine Menge von Testfällen erreicht eine vollständige Prädikatüberdeckung, wenn jedes Prädikat im Modell während eines Testdurchlaufes mindestens einmal zu wahr und einmal zu falsch ausgewertet wird.
- **Modifizierte Bedingungs-/Entscheidungsüberdeckung (MC/DC)**: Hierbei wird verlangt, dass jede Klausel k eines Prädikats p einmal zu wahr und einmal zu falsch ausgewertet wird. Darüber hinaus muss sichergestellt werden, dass die Klausel k direkt mit dem Ergebnis des Prädikats p korreliert. Dies ist dann der Fall, wenn ein Wechsel des Wertes von k, während alle anderen Klauseln aus p ihren Wert behalten, zu einem Wechsel des Wertes von p führt.
  Diese Definition von MC/DC ist in [AOH03] als *Restricted Active Clause Coverage* aufgeführt.

Andere strukturelle Überdeckungskriterien sind für die Modellierung mit Hilfe von Transitionssystemen entwickelt worden. Ein Beispiel hierfür ist die Kantenüberdeckung.

2.3.1.2 Fehlermodelle

Das Ziel des Testens ist es, Fehler aufzudecken. Um bestimmte Arten von Fehlern zu finden, können *Fehlermodelle* eingesetzt werden. Diese Modelle beschreiben eine bestimmte Menge von Fehlern, von denen man typischerweise annimmt, dass diese Arten von Fehlern häufig vorkommen.

Eine Möglichkeit solche Fehler zu modellieren ist es, mit *Mutationen* zu arbeiten. Die Idee von Mutationen ist es, den Programmcode des Testobjektes an einigen Stellen gezielt zu verändern. Wie diese Veränderungen aussehen, wird durch sogenannte *Mutationsoperatoren* beschrieben, wobei ein Mutationsoperator im Idealfall einen typischen Fehler, der durch Programmierer gemacht wird, darstellt. Ein Beispiel für einen Mutationsoperator ist es alle Vergleichsoperatoren $\leq$ durch $\geq$ zu ersetzen.

Durch Mutationsoperatoren können demnach viele fehlerhafte Versionen des Originalprogrammcodes, sogenannte *Mutanten*, erzeugt werden. Mit Hilfe dieser Mutanten kann die Qualität einer Testsuite gemessen werden, indem die Testsuite auf einer Menge von Mutanten ausgeführt wird und anschließend überprüft wird, in wie vielen Fällen die Testfälle einen

durch den Mutationsoperator injizierten Fehler aufdeckt. Wird solch ein Fehler aufgedeckt, so spricht man davon, dass der Mutant *getötet* (engl.: *killed*) wurde.

Das gerade beschriebene Vorgehen dient dazu, für eine bereits erzeugte Testsuite die Qualität zu messen. Es liegt nahe, dieses Vorgehen ebenfalls als Testauswahlkriterium zu verwenden, also zum Zeitpunkt der Testfallerzeugung solche Testfälle zu erzeugen, die in der Lage sind, einen Mutanten zu töten. Im modellbasierten Testen wird hierzu ein Mutationsoperator auf das Modell angewendet und anschließend wird ein Testfall gesucht, der in der Lage ist zwischen dem Originalmodell und dem Mutanten zu unterscheiden.

2.4 Automatische Handlungsplanung

In dieser Arbeit wird automatische Handlungsplanung zur Testfallgenerierung eingesetzt. Automatische Handlungsplanung ist ein Zweig der künstlichen Intelligenz, der sich mit dem Problem beschäftigt aus einer gegebenen Menge von Aktionen diejenigen in der richtigen Reihenfolge auszuführen, sodass ein vorgegebenes Ziel erreicht wird.

Typischerweise besteht das Problem aus drei Eingaben: einer Beschreibung des Anfangszustands, einem Zielzustand und einer Menge von möglichen Aktionen mit Vorbedingungen und Effekten. Die Aufgabe von Planung ist es, eine gültige Sequenz von Aktionen zu finden, die den Anfangszustand in den Zielzustand überführen. Hierbei wird eine Sequenz als gültig bezeichnet, wenn jede Aktion nur dann ausgeführt wird, wenn ihre Vorbedingungen zum Zeitpunkt der Ausführung erfüllt sind.

Ein anschauliches Beispiel ist in Abbildung 5 gegeben. Diese Abbildung zeigt eine Welt mit einem Roboter, verschiedenen Laderampen und Containern, wobei die linke Abbildung einen Zustand beschreibt, in dem sich der Roboter an der Laderampe 1 befindet, Container 1 auf Laderampe 1 und Container 2 auf Laderampe 3.

Der Roboter in diesem Beispiel ist fähig verschiedene Aktionen durchzuführen. So ist der Roboter in der Lage Container eigenständig auf seine Ladefläche zu heben, wobei dies nur geschehen kann, falls sich der Roboter an der Laderampe befindet, auf der sich auch der Container befindet. Weiterhin kann er sich von einer Laderampe zu einer anderen Laderampe bewegen und dort seine Ladung abladen. Hierbei gilt jedoch die Einschränkung, dass sich auf jeder Rampe maximal ein Container befinden darf.

Ein mögliches Planungsproblem ist nun dadurch gegeben, dass die Welt von einem Zustand wie in der linken Abbildung beschrieben in einen Zustand überführt wird, in dem der Container 1 sich auf Laderampe 3 befindet, während die Position der anderen Objekte irrelevant ist (siehe rechte Seite der Abbildung).

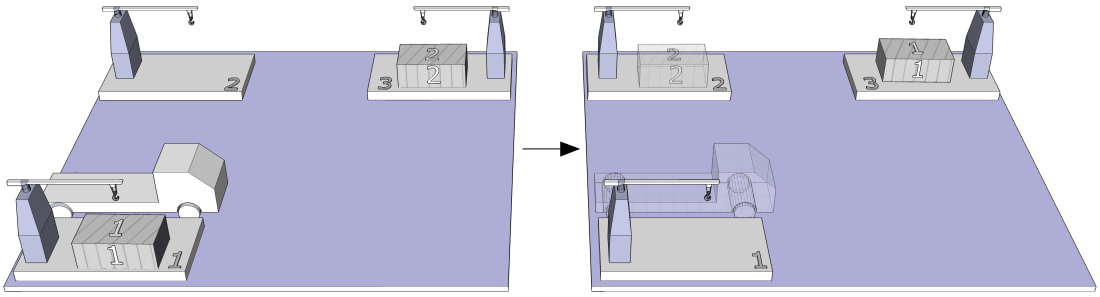


Abbildung 5: Beispiel für ein Planungsproblem

Eine mögliche Lösung für dieses Problem ist die folgende Sequenz von Aktionen:

1. bewege Roboter zu Rampe 3,
2. lade Container 2 auf Roboter,
3. bewege Roboter zu Rampe 2,
4. entlade Container 2 auf Rampe 2,
5. bewege Roboter zu Rampe 1,
6. lade Container 1 auf Roboter,
7. bewege Roboter zu Rampe 3
8. entlade Container 1 auf Rampe 3

Um nun solche Planungsprobleme computergestützt zu lösen, muss das Problem in einem ersten Schritt formalisiert werden, sodass es von Computern verarbeitet werden kann. Nachdem das Problem in einer computerlesbaren Form beschrieben worden ist, muss in einem nächsten Schritt eine Sequenz gefunden werden, die ausgehend vom Anfangszustand den Zielzustand erreicht. Beide Problematiken werden auf den nachfolgenden Seiten diskutiert.

2.4.1 Repräsentationen von Planungsproblemen

Im Folgenden werden zwei der gebräuchlichsten Repräsentationen für Planungsprobleme beschrieben, mit deren Hilfe Zustände sowie Aktionen beschrieben werden können [GNT04].

2.4.1.1 Klassische Repräsentation

Am weitesten verbreitet ist die Repräsentation von Planungsproblemen mit Hilfe einer Notation, die sich an der Prädikatenlogik orientiert. Hierbei wird ein Zustand durch eine Menge von einzelnen logischen Aussagen beschrieben, die zu *wahr* oder *falsch* ausgewertet werden können. Aktionen werden durch Planungsoperatoren beschrieben, die den Wahrheitswert dieser logischen Aussagen ändern.

Eines der ersten Planungssysteme, das STRIPS System [FN71], verwendete diese Art der Notation und aufbauend auf dieser Arbeit wurden weitere Beschreibungssprache entwickelt. Als wichtigster Vertreter ist hier die Planning Domain Description Language (PDDL) zu nennen [FL03, EH04, GL05, IPC08a]. Diese Beschreibungssprache wird bei den International Planning Competitions verwendet, ein von der AAAI ausgerichteter Wettbewerb bei den die neusten Entwicklungen im Bereich der KI-Planung vorgestellt und miteinander verglichen werden.

Zur Veranschaulichung der Beschreibung eines Zustandes in dieser Repräsentation wird nun als Beispiel die Beschreibung der Situation der linken Seite aus Abbildung 5 aufgeführt. Um die dargestellte Situation beschreiben zu können, werden die Prädikate `at(?obj, ?loc)` und `clear(?obj)`, sowie die Konstanten `Container1`, `Container2`, `Rampe1`, `Rampe2`, `Rampe3` und `Roboter` verwendet.

Hierbei wird mit der Notation `?v` eine Variable `v` bezeichnet, der eine beliebige Konstante zugewiesen werden kann. Das Prädikat `at(?obj, ?loc)` drückt aus, dass sich das

Objekt `obj` am Ort `loc` befindet, während das Prädikat `clear(?obj)` ausdrückt, dass sich kein anderes Objekt auf `obj` befindet.

Mit Hilfe dieser Prädikate und Konstanten kann die Situation nun folgendermaßen beschrieben werden: `at(Container1, Rampe1)`, `at(Container2, Rampe3)`, `at(Roboter, Rampe1)`, `clear(Rampe2)` und `clear(Roboter)`. Bei dieser Zustandsbeschreibung wird die Annahme einer *geschlossenen Welt* hinzugezogen. Diese Annahme besagt, dass alle Aussagen, die nicht explizit in einer Zustandsbeschreibung aufgeführt sind, zu *falsch* ausgewertet werden.

2.4.1.2 Aktionen

Aktionen werden durch ein Tripple $a = (name(a), precond(a), effects(a))$ beschrieben, wobei *name(a)* ein eindeutiger Bezeichner für die Aktion *a* ist. Die Bedingungen für das Ausführen der Aktion werden in *precond(a)* beschrieben, während *effects(a)* beschreibt, wie sich das Ausführen der Aktion auf den aktuellen Zustand auswirkt. Die Bedingungen und Effekte werden hierbei durch Prädikatenlogische Formeln dargestellt.

Um Aktionen möglichst kompakt beschreiben zu können, sind diese typischerweise parametrisierbar. Ein Beispiel hierfür ist die Aktion `bewege_Roboter`:

```
bewege_Roboter(?von, ?nach)
      precond: at(Roboter, ?von)
      effects: at(Roboter, ?nach) ∧ ¬at(Roboter, ?von)
```

Solch eine parametrisierbare Aktion kann in mehrere sogenannte *geerdete* Aktionen umgewandelt werden, indem alle möglichen Kombinationen von Konstanten für die Parameter der Aktion gebildet werden.

Für das Beispiel aus Abbildung 5 können nun unter anderem die folgenden geerdeten Aktionen bestimmt werden:

```
bewege_Roboter_von_Rampe1_nach_Rampe2
      precond: at(Roboter, Rampe1)
      effects: at(Roboter, Rampe2) ∧ ¬at(Roboter, Rampe1)

lade_Container1_an_Rampe1_auf_Roboter
      precond: at(Roboter, Rampe1) ∧ at(Container1, Rampe1)
               ∧ clear(Roboter)
      effects: ¬clear(?Roboter) ∧ at(Container1, Roboter)
               ∧ ¬at(Container1, Rampe1)

entlade_Container1_an_Rampe1_von_Roboter
      precond: at(Roboter, Rampe1) ∧ at(Container1, Rampe1)
               ∧ clear(Rampe1)
      effects: clear(Roboter) ∧ at(Container1, Rampe1)
               ∧ ¬at(Container1, Roboter) ∧ ¬clear(Rampe1)
```

Die hier dargestellten Bedingungen und Effekte der Aktionen sind einfache UND-Verknüpfungen von Literalen. Die klassische Repräsentation ist jedoch nicht auf solch einfache Formeln beschränkt, sondern kann durch die aus der Prädikatenlogik bekannten Konzepte erweitert werden, wie z.B. Quantoren und Disjunktionen [Schoening00].

Diese erweiterten Darstellungen erhöhen jedoch nicht die Ausdrucksmächtigkeit der Repräsentation, da sie auf (evtl. exponentiell viele) geerdete Aktionen abgebildet werden können. Sie erleichtern jedoch das Modellieren von komplexeren Welten [GNT04].

2.4.1.3 Repräsentation mit Zustandsvariablen

Eine Alternative zu der klassischen Repräsentation ist die Repräsentation von Planungsproblemen mit Hilfe von Zustandsvariablen. Obwohl beide Repräsentationen die gleiche Ausdrucksmächtigkeit besitzen, kann die Repräsentation mit Hilfe von Zustandsvariablen von Vorteil bei der Beschreibung bestimmter Eigenschaften sein.

Ein Beispiel hierfür ist die Beschreibung der Position eines Objektes. In der klassischen Repräsentation ist das in dem aufgeführten Beispiel mit Hilfe des Prädikats `at(?obj, ?loc)` realisiert. Diese Repräsentation ist jedoch nicht in der Lage die Tatsache zu erfassen, dass sich ein Objekt immer an einer Position befinden muss, sich jedoch niemals an zwei unterschiedlichen Positionen gleichzeitig befinden kann. Um dies dennoch sicherzustellen, müssen die Aktionen dementsprechend modelliert werden. So stellt zum Beispiel die Aktion `bewege_Roboter` sicher, dass das Prädikat `at(Roboter, ?von)` nach der Aktion nicht mehr gilt und somit die Position des Roboters eindeutig bestimmt ist.

Wird hingegen für die Beschreibung der Position des Roboters eine Zustandsvariable $\texttt{position}_{\texttt{roboter}}$ mit dem Wertebereich (`Rampe1`, `Rampe2`, `Rampe3`) verwendet, so lässt sich die Beschreibung der Aktion vereinfachen.

Die Situation aus Abbildung 5 kann unter Benutzung von Zustandsvariablen folgendermaßen beschrieben werden: $\texttt{position}_{\texttt{Roboter}}$=`Rampe1`, $\texttt{position}_{\texttt{Container1}}$=`Rampe1`, $\texttt{position}_{\texttt{Container2}}$=`Rampe3`, $\texttt{clear}_{\texttt{Roboter}}$=`true`, $\texttt{clear}_{\texttt{Rampe1}}$=`false`, $\texttt{clear}_{\texttt{Rampe2}}$=`true`, und $\texttt{clear}_{\texttt{Rampe3}}$=`false`, wobei die Zustandsvariablen $\texttt{position}_{\texttt{x}}$ den Wertebereich (`Rampe1`, `Rampe2`) und die Zustandsvariablen $\texttt{clear}_{\texttt{x}}$ den Wertebereich (`true`, `false`) besitzen.

Die geerdeten Aktionen aus dem vorherigen Abschnitt lassen sich demnach folgendermaßen beschreiben:

`bewege_Roboter_von_Rampe1_nach_Rampe2`

precond: $\texttt{position}_{\texttt{Roboter}}$=`Rampe1`

effects: $\texttt{position}_{\texttt{Roboter}}$=`Rampe2`

`lade_Container1_an_Rampe1_auf_Roboter`

precond: $\texttt{position}_{\texttt{Roboter}}$=`Rampe1` $\wedge$ $\texttt{position}_{\texttt{Container1}}$=`Rampe1`
$\wedge$ $\texttt{clear}_{\texttt{Roboter}}$=`true`

effects: $\texttt{position}_{\texttt{Container1}}$=`Rampe1` $\wedge$ $\texttt{clear}_{\texttt{Roboter}}$=`false`

```
entlade_Container1_an_Rampe1_von_Roboter
      precond: position_Roboter=Rampe1 ∧ position_Container1=Roboter
               ∧ clear_Rampe1=true
      effects: position_Container1=Roboter ∧ clear_Roboter=true
```

2.4.1.4 Vergleich der Repräsentationen

Wie bereits im vorherigen Abschnitt erwähnt, besitzen beide Repräsentationen die gleiche Ausdrucksmächtigkeit. Beide Repräsentationen können ineinander überführt werden, wobei bei geerdeten Aktionen die dafür benötigte Größe der Beschreibung maximal linear ansteigt [GNT04]. Somit ist es letztlich eine Frage der zu modellierenden Domäne und der zur Verfügung stehenden Werkzeuge, für welche Repräsentation man sich entscheidet.

2.4.2 Bedingte Effekte

Aktionen werden häufig um bedingte Effekte erweitert. Da in dieser Arbeit bedingte Effekte von tragender Bedeutung sind, werden sie an dieser Stelle eingeführt. Die Grundidee von bedingten Effekten ist es Aktionen abzubilden, deren Effekte sich in Teilen nach der Situation richten, in der die Aktion zur Ausführung kommt.

Zur Notation eines bedingten Effektes wird die Form: `when (Condition) → (Effect)` verwendet. Die folgende Aktion enthält solch einen bedingten Effekt.

```
action_name
      precond: cond_1
      effects: effects_1
               ∧ when (cond_2)
                  → (effects_2)
```

Diese Aktion hat die Bedingungen $cond_1$ und die Effekte $effects_1$. Zusätzlich enthalten die Effekte der Aktion noch einen bedingten Effekt mit der Bedingung $cond_2$ und dem Effekt $effects_2$. Die Aktion kann wie gehabt nur dann ausgeführt werden, wenn die Bedingung $cond_1$ zum Zeitpunkt der gewünschten Ausführung gilt. Das Ausführen der Aktion stellt die Effekte, die in $effects_1$ notiert sind, sicher.

Gilt nun während der Ausführung der Aktion zusätzlich zur Bedingung $cond_1$ auch die Bedingung $cond_2$ des bedingten Effekts, so werden auch die Effekte $effects_2$ hergestellt.

Ein einfaches Beispiel für bedingte Effekte kann gegeben werden, indem das Beispiel des Laderoboters um einen Anhänger `trailer` erweitert wird, der an den Roboter angehängt werden kann. Die Tatsache, dass der Anhänger mit dem Roboter verbunden ist, wird hierbei durch das Prädikat `connected` und die Position des Anhängers wird durch das Prädikat `at` beschrieben.

Mit Hilfe von bedingten Effekten, kann die Aktion `bewege_Roboter` nun so notiert werden, dass ein Standortwechsel des Roboters auch einen Standortwechsel des Anhängers bewirkt, falls der Anhänger mit dem Roboter verbunden ist.

```
bewege_Roboter(?von, ?nach)
      precond: at(Roboter, ?von)
      effects: at(Roboter, ?nach) ∧ ¬at(Roboter, ?von)
               ∧ when(connected) → at(trailer, ?nach)
```

2.4.3 Repräsentation von Zeit

In eingebetteten Systemen spielt Zeit häufig eine wichtige Rolle bei der Beschreibung der Funktionalität des Systemes. Um dieser Tatsache gerecht zu werden, widmet sich dieser Abschnitt der Repräsentation von Zeit in Planungsproblemen.

Aufgrund der Popularität von PDDL wird zuerst auf die Repräsentation von Zeit in PDDL eingegangen, die jedoch im Vergleich zu anderen Ansätzen starke Einschränkungen bei der Modellierung von Zeitverhalten aufweist. Über die eingeschränkte Modellierung von Zeit in PDDL hinaus gibt es weitere Ansätze zur Modellierung von Zeit in Planungsproblemen. In [GNT04] – ein Lehrbuch zur automatischen Handlungsplanung – widmet sich ein Kapitel der Planung unter Einbeziehung von Zeit. Hierbei werden die existierenden Ansätze in zwei unterschiedliche Gruppen eingeteilt: Planung mit *Temporally qualified expressions* und Planung mit *Chronicles*. Diese Arbeit folgt dieser Einteilung und Begriffswelt und stellt beide Ansätze vor.

2.4.3.1 Zeit in PDDL

In PDDL 2.1 [FL03] sind sogenannte *durative actions*, also Aktionen mit Zeitdauer, eingeführt worden, um zeitbehaftete Planungsprobleme modellieren zu können. Hierbei wird einer Aktion eine Dauer zugewiesen, wobei diese Dauer entweder fest vorgegeben sein kann oder aber durch eine Funktion mit mehreren Variablen bestimmt wird.

Weiterhin muss für jede Vorbedingung angegeben werden, ob diese zu Beginn, zum Ende oder über die gesamte Dauer der Aktion gelten muss. Für Effekte gilt Ähnliches jedoch kann hier lediglich der Beginn und das Ende der Aktion als möglicher Zeitpunkt für das Eintreten eines Effektes angegeben werden. Die folgende Aktion ist ein Beispiel für die Modellierung von zeitbehafteten Aktionen in PDDL.

Die Aktion modelliert das Erhitzen von Wasser in einem Behältnis `?p` auf eine Temperatur von 100 Grad. Die Dauer der Aktion hängt von der Temperatur von `?p` zum Startzeitpunkt der Aktion und der „Erhitzungsrate“ `heat-rate` ab, wobei `?p` eine Variable ist und `heat-rate` eine Konstante. Bedingungen für die Ausführung der Aktion sind, dass das Gefäß zum Beginn und während der Aktion gefüllt ist (`(full ?p)`) und sich auf dem der Heizquelle befindet (`(onHeatSource ?p)`). Weiterhin wird verlangt, dass die Heizquelle während der gesamten Aktion eingeschaltet ist (`(heating ?p)`). Als Effekte schaltet die Aktion zu Begin der Aktion die Heizquelle ein und zum Ende der Aktion aus. Weiterhin ist zum Ende der Aktion die Temperatur von ?p auf 100 gesetzt.

```
(:durative-action heat-water
 :parameters (?p - pan)
 :duration (= ?duration (/ (- 100 (temperature ?p)) (heat-rate)))
 :condition (and (at start (full ?p))
                 (at start (onHeatSource ?p))
                 (over all (full ?p))
                 (over all (onHeatSource ?p))
                 (over all (heating ?p)))
 :effect (and  (at start (heating ?p))
                 (at end (not (heating ?p)))
                 (at end (assign (temperature ?p) 100)))
)
```

Die Tatsache, dass es in PDDL nicht möglich ist, andere Zeitpunkte als die Endpunkte des Intervalls der Aktionsdauer zu referenzieren stellt eine Schwäche von PDDL bei der Modellierung von Zeitverhalten dar. Um diese Einschränkung zu umgehen, ist es nötig zusätzlichen Aufwand zu betreiben, in dem z.B. eine Aktion in mehrere Einzelaktionen unterteilt wird. Diese Art der Modellierung ist einerseits unübersichtlich und führt darüber hinaus zu zusätzlichen Suchaufwand während der Lösung des Planungsproblems [Smith03].

Der Begriffswelt von [GNT04] folgend werden nun zwei Repräsentationen vorgestellt, die diese Einschränkungen nicht beinhalten.

2.4.3.2 Temporally qualified expressions

In der klassischen Repräsentation werden die Vorbedingungen von Aktionen so definiert, dass in einem Zustand gewisse Aussagen wahr oder falsch sein müssen. Ähnliches gilt für die Effekte einer Aktion. In diesem Abschnitt wird die Repräsentation dahingehend verändert, dass die Aussagen nicht mehr in einem Zustand, sondern über zeitliche Intervalle gelten müssen. Um dies zu erreichen, werden mit Zeit angereicherte Ausdrücke, sogenannte *temporally qualified expressions,* verwendet, die nun definiert werden.

Definition 2.1. *Eine* temporally qualified expression (TQE) *ist ein Ausdruck der Form* $P(\zeta_1,\ldots,\zeta_k)@[t_s,t_e)$*, wobei P ein k-stelliges Prädikat ist,* $\zeta_1,\ldots,\zeta_k$ *Konstanten oder Variablen sind und* t_s *und* t_e *temporale Variablen sind, für die* $t_s<t_e$ *gilt. Eine TQE* $P(\zeta_1,\ldots,\zeta_k)@[t_s,t_e)$ *sagt aus, dass das Prädikat* $P(\zeta_1,\ldots,\zeta_k)$ *für alle Zeitpunkte t aus* $t_s\leq t<t_e$ *gilt.*

Mit Hilfe von TQEs kann die klassische Repräsentation sehr geradlinig um ausdrucksstarkes Zeitverhalten erweitert werden, indem in den Vorbedingungen und Effekten die Prädikate durch TQEs ersetzt werden.

Als Beispiel wird die `bewege_Roboter` Aktion so erweitert, dass der Roboter die Rampe `?von` zum Zeitpunkt t_s verlässt, danach befindet er sich auf der Straße und erreicht schließlich nach 6 Zeiteinheiten sein Ziel, die Rampe `?nach`. Die Bedingung für das Aus-

führen der Aktion ist, dass sich der Roboter zum Zeitpunkt t_s an Rampe `?von` befindet und zwar seit mindestens 2 Zeiteinheiten.

Nachfolgend ist die Aktion mittels TQEs notiert und in Abbildung 6 grafisch dargestellt. In der grafischen Darstellung ist `Roboter` durch `r` abgekürzt. Weiterhin sind temporale Variablen, die nicht durch Randbedingungen eingeschränkt sind ohne vertikalen Strich gekennzeichnet (t_3).

```
bewege_Roboter(?von, ?nach)
      precond:     at(Roboter, ?von)@[t1, ts)
      effects:     at(Roboter, straße)@[ts, t2)
                   ∧ at(Roboter, ?nach)@[t2, t3)
      constraints: t1 ≤ ts − 2 ∧ t2 = ts + 6
```

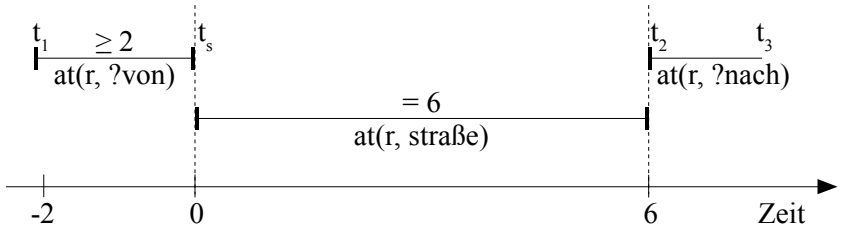


Abbildung 6: Grafische Darstellung der Aktion `bewege_Roboter`

Vergleicht man die zeitbehaftete Aktion mit der zeitlosen Variante aus Abschnitt „Klassische Repräsentation“, so fällt auf, dass die Aktion nur positive Effekte enthält und nichts darüber aussagt, welche Aussagen nicht mehr gelten. Für die `bewege_Roboter` Aktion bedeutet dies, dass die Aktion nicht beschreibt, dass sich der Roboter ab Zeitpunkt t_s nicht länger an Position `?von` befindet. Dieses Problem wird durch die Verwendung von Domänen-Axiomen angegangen.

Definition 2.2. *Ein* Domänen-Axiom *ist ein bedingter Ausdruck der Form*

$$\rho = \mathrm{cond}(\rho) \rightarrow \mathrm{disj}(\rho),$$

wobei $\mathrm{cond}(\rho)$ *eine Menge von TQEs ist und* $\mathrm{disj}(\rho)$ *eine Disjunktion von Randbedingungen an (temporale) Variablen.*

In dem angeführten Beispiel kann ein Domänen-Axiom benutzt werden, um auszudrücken, dass sich der Roboter zu keiner Zeit an unterschiedlichen Positionen gleichzeitig befinden kann:

$$\{\mathrm{at}(\mathrm{Roboter}, ?\mathrm{p})@[t_s, t_e), \mathrm{at}(\mathrm{Roboter}, ?\mathrm{p}')@[t'_s, t'_e)\} \rightarrow (?\mathrm{p} = ?\mathrm{p}') \vee (t_e \leq t'_s) \vee (t'_e \leq t_s)$$

In Worten ausgedrückt legt dieses Axiom fest, dass es für den Fall, dass es zwei TQEs gibt, die beide eine Aussage über die Position des Roboters machen, entweder die Position gleich sein muss, oder aber die Zeitintervalle dürfen sich nicht überschneiden.

2.4.3.3 Chronicles

Die Beschreibung von Zeitverhalten mit Hilfe von TQEs stelle eine Erweiterung der klassischen Repräsentation dar. In diesem Abschnitt soll nun eine Erweiterung der Zustandsvariablen Repräsentation um Zeitbeschreibungen vorgestellt werden.

Die Verwendung von Chronicles zur Beschreibung unterscheidet sich in zwei Punkten von der Verwendung von TQEs: Erstens werden Zustandsvariablen verwendet, was z.B. die Verwendung des gerade vorgestellten Domänen-Axioms überflüssig macht, da Zustandsvariablen bereits ausdrücken, dass eine Variable zu jeder Zeit nur einen Wert annehmen kann; zweitens wird durch die Verwendung von Chronicles die Unterscheidung zwischen Effekten und Bedingungen aufgelöst. Anstelle der Unterscheidung zwischen Bedingungen und Effekte tritt die Unterscheidung zwischen *fortdauernden Bedingungen* und *Ereignissen*.

Definition 2.3. *Eine* fortdauernde Bedingung *an die Zustandsvariable x wird notiert als* $x@[t_1, t_2):v$. *Dies spezifiziert, dass der Wert von x im Intervall* $[t_1, t_2)$ *den Wert v besitzt. Es gilt somit*

$$x@[t_1, t_2):v \equiv \forall\, t(t_1 \le t < t_2) \wedge x(t) = v.$$

Definition 2.4. *Ein* Ereignis *wird notiert als* $x@t:(v_1, v_2)$ *und beschreibt den sofortigen Wechsel des Wertes von x von* v_1 *zu* v_2 *zum Zeitpunkt t, wobei* $v_1 \neq v_2$. *Diese Aussage kann mit Hilfe der vorherigen Definition beschrieben werden:*

$$x@t:(v_1, v_2) \equiv \exists t_1, t_2 (t_1 < t < t_2) \wedge x@[t_1, t):v_1 \wedge x@[t, t_2):v_2 \wedge v_1 \neq v_2$$

Für beide Definitionen gilt, dass t, t_1 und t_2 temporale Variablen und v, v_1 und v_2 Konstanten oder Variablen aus dem Wertebereich von x sind.

Um nun die `bewege_Roboter` Aktion mit Hilfe von Chronicles zu beschreiben, lohnt sich ein Blick auf Abbildung 6. Dabei wird deutlich, dass es 2 Ereignisse gibt, zu denen sich die Position des Roboters ändert: zum Zeitpunkt t_s von `?von` zu `straße` und zum Zeitpunkt t_2 von `straße` zu `?nach`. Außerdem zeigt die Abbildung drei fortdauernde Bedingungen: im Intervall $[t_1, t_s)$ muss sich der Roboter an Position `?von`, im Intervall $[t_s, t_2)$ an Position `straße` und schließlich im Intervall $[t_2, t_3)$ an Position `?nach` befinden.

Folgt man diesen Beobachtungen, lässt sich die Aktion folgendermaßen mit Hilfe von Chronicles beschreiben:

bewege_Roboter(?von, ?nach)

chronicles: $\text{position}_{\text{Roboter}}@[t_1, t_s):$?von,
$\text{position}_{\text{Roboter}}@t_s:$(?von,straße),
$\text{position}_{\text{Roboter}}@[t_s, t_s):$straße,
$\text{position}_{\text{Roboter}}@t_2:$(straße, ?nach),
$\text{position}_{\text{Roboter}}@[t_2, t_3):$?nach,

constraints: $t_1 \leq t_s - 2 \wedge t_2 = t_s + 6$

2.4.3.4 Vergleich der Zeitdarstellungen

Ähnlich wie bei dem Vergleich der zeitlosen Varianten hängt die Entscheidung, welche Repräsentation zu bevorzugen ist, in erster Linie von der zu modellierende Domäne ab. Im Konzeptkapitel wird die Frage, welche Repräsentation für diese Arbeit zu verwenden ist, diskutiert werden.

2.4.4 Planungsalgorithmen

Ausgehend von der formalen Beschreibung des Planungsproblems ist der nächste Schritt die Suche nach einem Lösungsplan, also einer gültigen Sequenz von Aktionen, die den Startzustand des Problems in den gewünschten Zielzustand überführt.

Zur Lösung dieser Aufgabe gibt es verschiedene Möglichkeiten, die sich in drei Gruppen unterteilen lassen: die Suche im Zustandsraum, die Suche im Planraum und Planung als Erfüllbarkeitsproblem.

2.4.4.1 Suche im Zustandsraum

Eine Möglichkeit solch einen Lösungsplan zu finden, ist es den *Zustandsraum*, der durch das Planungsproblem beschrieben wird, zu durchsuchen. Hierzu muss man sich zu Begin klarmachen, dass ein Planungsproblem ein Transitionssystem beschreibt, in dem die Knoten einem Zustand der Welt entsprechen und jede Transition einer Aktion entspricht. Dies wird durch das folgende Beispiel verdeutlicht. Das dargestellte Beispiel ist eine vereinfachte Version des bisher verwendeten Beispiels mit nur einem Container und zwei Laderampen. Abbildung 7 zeigt das Transitionssystem für dieses einfache Beispiel.

Um nun einen Lösungsplan zu finden, kann der Zustandsraum des Transitionssystems vorwärts durchsucht werden, indem ausgehend vom Startzustand der Zustandsraum generiert wird, bis ein Zustand gefunden wird, der das gewünschte Planungsziel erfüllt. Die zweite Möglichkeit ist es eine Rückwärtssuche durchzuführen, das heißt ausgehend vom Ziel, einen Pfad zum Initialzustand zu finden.
Das Problem bei dieser Herangehensweise ist es, dass der Zustandsraum exponentiell in der Anzahl der geerdeten Aktionen wächst. Somit ist ein einfaches Durchsuchen des Zustandsraumes bei größeren Planungsproblemen nicht realistisch. Um diesem Problem Herr zu werden, werden *heuristische Suchen* – oft auch *gerichtete* oder *informierte* Suchen genannt – verwendet. Hierauf wird in Kapitel 2.5 näher eingegangen.

In den letzten Jahren sind einige sehr erfolgreiche Heuristiken für die geführte Suche entwickelt worden, die dazu geführt haben, dass Planer mit heuristischer Suche im Zu-

standsraum in den letzten Jahren die International Planning Competitions dominiert haben [IPC08b, IPC06, IPC04, IPC02, IPC00].

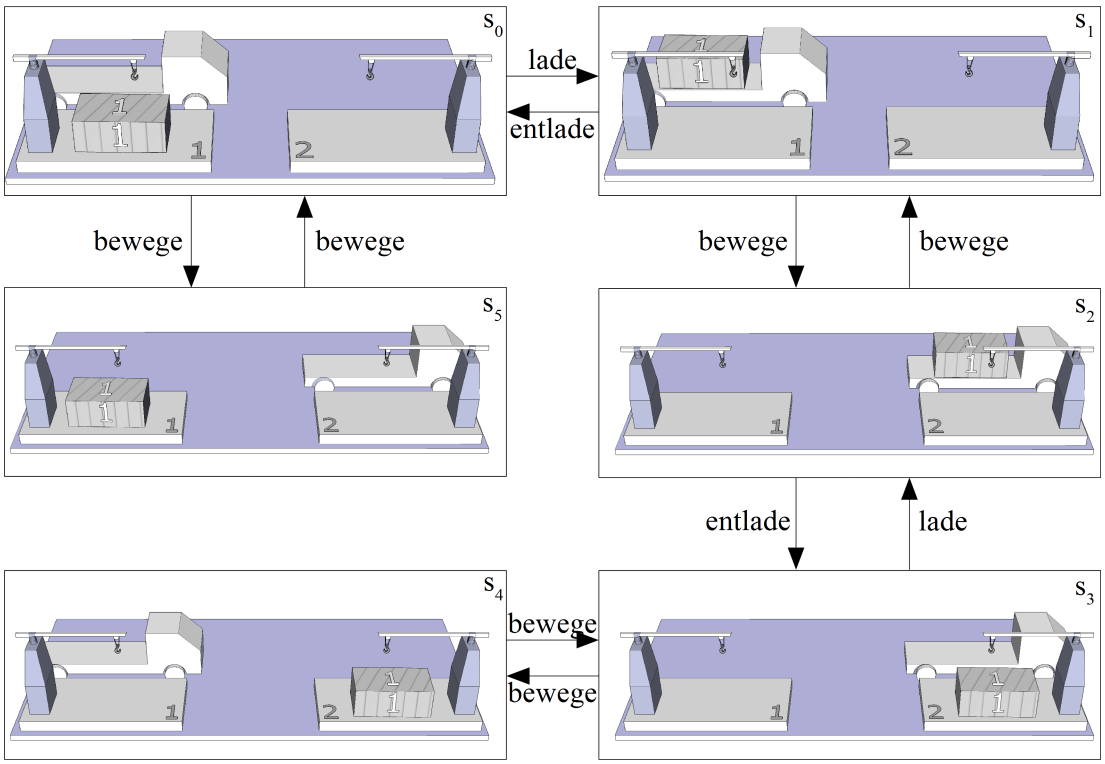


Abbildung 7: Transitionssystem für ein Planungsproblem

Einbeziehung von Zeit

Der Erfolg der Planer mit heuristischer Suche im Zustandsraum hat dazu geführt, dass dieser Ansatz auch auf zeitbehaftete Planungsprobleme angewendet worden ist. Die Erweiterung um Zeit hat zur Folge, dass das Transitionssystem nicht mehr nur Transitionen enthält, die einer Aktion entsprechen, sondern, dass jeder Zustand auch durch das Verstreichen von Zeit in einen neuen Zustand überführt werden kann. Dies wiederum führt zu der Problematik, dass hierdurch unendlich viele Zustände entstehen, da sich ein Zustand durch das Verstreichen von Zeit kontinuierlich verändert.

Um dieses Problem zu umgehen, schränken die Planer die möglichen Zeitpunkte, die während einer Suche berücksichtigt werden, auf sogenannte Entscheidungsepochen (engl. decision epochs) ein [CKM+07]. Dieses Vorgehen ist stark an das eingeschränkte Zeitmodell in PDDL gekoppelt, da die Entscheidungsepochen durch die Start- und Endpunkte der Aktionen bestimmt sind, mit der Begründung, dass nur zu diesen Zeitpunkten Änderungen an der Welt geschehen und Entscheidungen getroffen werden müssen.

2.4.4.2 Suche im Planraum

In einer weiteren Planungsmethode wird ein Suchraum durchsucht, in dem die Knoten des Suchraumes durch partielle Pläne und die Kanten durch Verfeinerungsoperationen für diese partiellen Pläne definiert sind. Ein Vorteil dieses Verfahrens ist es, das es sehr geradlinig er-

weitert werden kann, um Zeitbedingungen zu unterstützen. Dies gilt sowohl für Repräsentationen mit Hilfe von TQEs als auch für Repräsentationen mit Hilfe von Chronicles [GNT04].

Da dieses Verfahren im Lösungskonzept zum Einsatz kommt, soll es an dieser Stelle nicht näher erläutert werden, da es im Kapitel 4 detailliert beschrieben wird.

2.4.4.3 Planung als Erfüllbarkeitsproblem

Eine weitere Möglichkeit der Lösung eines Planungsproblems ist es, das Problem auf ein anderes Problem abzubilden, für das bereits effektive Algorithmen zur Lösung existieren. Für Planungsprobleme kann dies geschehen, indem das Planungsproblem als aussagenlogisches Erfüllbarkeitsproblem (kurz *SAT*, englisch für *satisfiability*) dargestellt wird [KS92].

Das Planungsproblem wird bei diesem Vorgehen zunächst in eine aussagenlogische Formel übersetzt und für diese Formel wird anschließend nach einer erfüllenden Belegung der Variablen der Formel gesucht. Diese Lösung der Formel kann als Lösung des Planungsproblems verwendet werden.

Da das SAT-Problem ein bekanntes und häufig untersuchtes Gebiet der Informatik darstellt, existieren effiziente Algorithmen und Programme (sogenannte *SAT-Solver*), die in der Lage sind eine solche Lösung zu finden.

Aussagenlogische Formeln sind allerdings nicht geeignet, um Zeitbedingungen darzustellen, daher kann dieser Ansatz nicht auf Planung mit Zeitbedingungen angewandt werden. Um auch Zeitbedingungen zu unterstützen, ist es jedoch möglich das Problem auf eine Formel, die neben einfachen Aussagen auch lineare Ungleichungen enthält, abzubilden. Auch für solche Formeln existieren effiziente Verfahren (sogenannte *SMT-Solver*) [NOT06, BMS07, MB08], die eine erfüllende Belegung der Formel finden können. In Abschnitt 4.10 „Alternatives Lösungskonzept“ wird näher auf diese Möglichkeit eingegangen.

2.5 Suchalgorithmen

In diesem Abschnitt werden verschiedene Methoden vorgestellt, um einen *Suchbaum* zu durchsuchen. Ein Suchbaum entsteht, indem ausgehend vom Startzustand mehrere Aktionen ausgeführt werden können. Das Ausführen solch einer Aktion erzeugt einen neuen Zustand, in dem wiederum verschiedene Aktionen ausgeführt werden können und so weiter. Die Anzahl der Aktion, die in jedem Knoten des Baumes ausgeführt werden können, werden als Verzweigungsfaktor b bezeichnet. Abbildung 8 zeigt eine abstrakte Darstellung für einen Suchbaum.

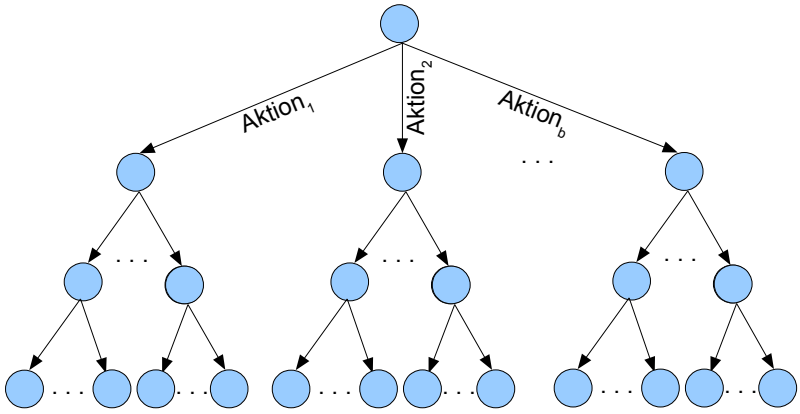


Abbildung 8: Suchbaum

Solch ein Suchbaum kann auf verschiedene Arten durchlaufen werden. Hierbei wird zwischen *uninformierter* und *heuristischer* Suche unterschieden [Ertel09].

2.5.1 Uninformierte Suchen

Zu den uninformierten Suchen zählen die *Breiten-* und die *Tiefensuche*. Bei der Breitensuche wird der Suchbaum von oben nach unten, von links nach rechts durchsucht. Abbildung 9a zeigt, in welcher Reihenfolge die Knoten des Suchbaumes erzeugt werden. Die Zeitkomplexität beträgt hierbei $O(b^d)$ und der Speicherbedarf liegt ebenfalls bei $O(b^d)$.

Bei der Tiefensuche wird immer zuerst der erste Nachfolger eines Knotens erzeugt und überprüft. Erfüllt dieser Knoten nicht das gesuchte Ziel, wird wiederum dessen erster Nachfolgeknoten erzeugt und überprüft. Auf diese Weise wird die Suche in die Tiefe des Baumes getrieben.

Erst wenn ein Knoten keine Nachfolger mehr hat, wird mittels *Backtracking* rückwärts bei der letzen Verzweigung der nächste noch nicht untersuchte Knoten erzeugt, und so weiter. Abbildung 9b zeigt, in welcher Reihenfolge die Knoten des Suchbaumes bei der Tiefensuche erzeugt und überprüft werden.

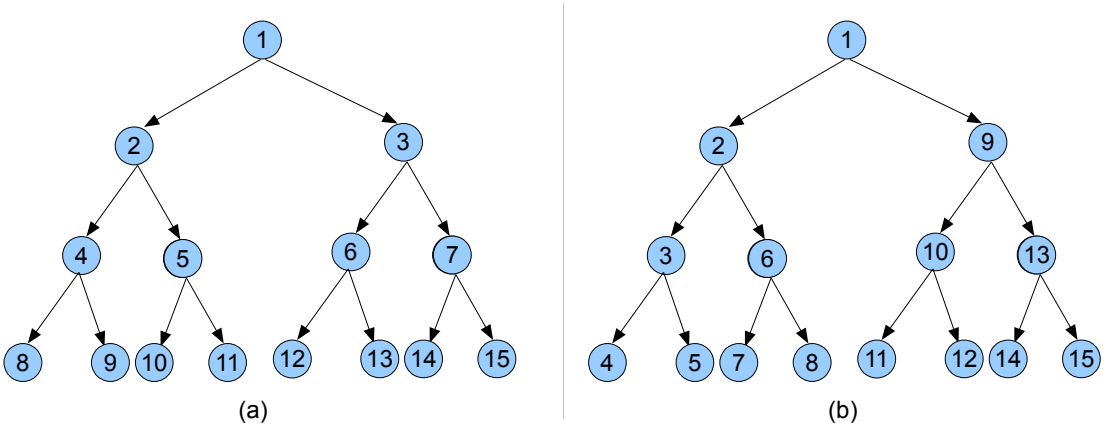


Abbildung 9: Beispiel für Breitensuche (a) und Tiefensuche (b)

Zu beachten ist, dass die Tiefensuche bei unendlich tiefen Bäumen nicht vollständig ist. Falls auf dem linken Ast keine Lösung liegt, wird dieser Ast immer weiter in die Tiefe verfolgt. Um diese Endlosschleife zu verhindern, kann eine *Tiefenschranke* verwendet werden, das heißt, der Baum wird nur bis zu einer definierten Tiefe durchsucht. Selbstverständlich werden dadurch nur Lösungen gefunden, die innerhalb des in der Tiefe beschränkten Suchbaumes liegen. Um zu garantieren, dass eine Tiefensuche eine vorhandene Lösung findet, kann die Tiefenschranke solange erhöht werden, bis die Suche erfolgreich ist (*iterative deepening*).

Tiefensuche hat im Gegensatz zur Breitensuche einen sehr moderaten Speicherbedarf. Wenn m die maximale Tiefe der Suche bezeichnet, beträgt der Speicherbedarf der Suche $O(bm)$. Die Zeitkomplexität liegt ebenso wie die Breitensuche bei $O(b^m)$.

Beide Verfahren sind mit dem Problem behaftet, dass ein vollständiges Durchsuchen des Suchbaumes bei realistischen Problemen häufig nicht möglich ist, da die Größe des Suchbaumes exponentiell mit der Tiefe wächst. Daher sind Suchverfahren von Interesse, die es ermöglichen in großen Suchbäumen eine Lösung zu finden. Solche Verfahren werden nun vorgestellt.

2.5.2 Heuristische Suche

Bei einer heuristischen Suche – oft auch informierte oder gerichtete Suche genannt – werden zuerst die Teile des Suchbaumes durchsucht, die bezüglich einer *heuristischen Bewertungsfunktion* die größte Wahrscheinlichkeit haben, dass sie den gesuchten Zielzustand enthalten. Häufig kann auf diese Art und Weise der Suchvorgang erheblich beschleunigt werden. Dies gilt jedoch nur, wenn das gesuchte Ziel auch tatsächlich erreichbar ist.

Best-first Suchen

Abbildung 10 zeigt einen Algorithmus für die heuristische Suche. Dieser Algorithmus startet, indem zuerst die Knotenliste mit dem Startknoten des Suchbaumes initialisiert wird. Dann wird in der Schleife der erste Knoten aus der Liste entfernt und geprüft, ob dieser Knoten dem gesuchten Ziel entspricht. Wenn nicht, werden durch die Funktion „Nachfolger“ alle Nachfolger des Knotens erzeugt und anschließend mit Hilfe der Funktion „Einsortieren“ in die Knotenliste eingefügt. Die Funktion „Einsortieren(X,Y)“ fügt die Elemente aus X aufsteigend sortiert in die Liste Y ein. Als Sortierschlüssel wird hierbei die heuristische Bewertungsfunktion $f(s)$ verwendet, die die Kosten zum Ziel für den Knoten s berechnet. Als Resultat dieser Einfügeoperation stehen in der Liste Y immer die Elemente mit der niedrigsten heuristischen Bewertung am Anfang der Liste.

```
1.   HeuristischeSuche(Start, Ziel)
2.    knotenliste=[Start]
3.    while True
4.     If knotenliste = ∅ Return "keine Lösung"
5.     knoten = Erster(knotenliste)
6.     knotenliste = Rest(knotenliste)
7.     If Ziel_erreicht(knoten, ziel) Return Knoten
8.     knotenliste=Einsortieren(Nachfolger(knoten,knotenliste)
```

Abbildung 10: Algorithmus für heuristische Suche [Ertel09]

Da die Berechnung der tatsächlichen Kosten für einen Knoten *s* das Lösen des Suchproblems erfordert, werden nicht die tatsächlichen Kosten als Bewertungsfunktion verwendet, sondern es wird eine *Schätzfunktion h(s)* verwendet.

Für ein Planungsproblem mit einem Zielzustand der Form $g=g_1 \wedge \ldots \wedge g_n$ lässt sich z. B. die folgende einfache Schätzfunktion angeben:

$$h(s)=\text{Anzahl der } g_i, \text{ die in } s \text{ nicht erfüllt sind.} \qquad (2.1)$$

Das heißt, ein Knoten, der viele Teilziele von *g* erfüllt, hat niedrigere geschätzte Kosten, als ein Knoten in dem wenige Teilziele erfüllt sind.

Bei der *gierigen Suche* wird die Bewertungsfunktion gleich der Schätzfunktion gesetzt, das heißt $f(s)=h(s)$. Diese Art der Suche ist jedoch nicht vollständig, da ähnlich wie bei der Tiefensuche ein Ast unendlich tief verfolgt werden kann.

Ein heuristisches Suchverfahren, das Vollständigkeit garantiert, ist die A*-Suche. Bei diesem Verfahren werden die Kosten, die bereits auf dem Weg bis zum aktuellen Knoten *s* angefallen sind, mit berücksichtigt. Die bisher angefallenen Kosten werden durch die Kostenfunktion *g(s)* bezeichnet und die Bewertungsfunktion wird auf $f(s)=g(s)+h(s)$ gesetzt.

Der Schwachpunkt der bisher vorgestellten Verfahren ist der hohe Speicherbedarf, da die Anzahl der Knoten in der Knotenliste exponentiell mit der Länge der Lösung wächst [RN95].

Hill-Climbing

Um das Problem der schnell wachsenden Knotenliste zu umgehen, kann *Hill-Climbing* eingesetzt werden. Der Name des Algorithmus kommt daher, dass das Vorgehen mit einem Bergsteiger, der im dichten Nebel auf der Suche nach dem Gipfel ist, vergleichbar ist. Aufgrund des Nebels hat der Bergsteiger nur eine begrenzte Sicht und wählt daher in jedem Schritt immer die Richtung aus, die ihn zu dem höchsten Punkt führt, den er momentan sieht. Die begrenzte Sicht drückt sich im Algorithmus dadurch aus, dass keine Knotenliste mitgeführt wird.

Ein Problem des Hill-Climbing Algorithmus liegt darin, dass ein Pfad eingeschlagen werden kann, der ein lokales Maximum enthält. Im Bergsteiger-Beispiel ist dies mit einer Situation zu vergleichen, in der sich der Bergsteiger an einem Punkt befindet, von dem aus es in jede Richtung bergab geht, es sich bei diesem Punkt jedoch nicht um den Gipfel handelt.

```
1.   HillClimbing(start, ziel)
2.    best=f(start)
3.    knoten=start
4.    while True
5.      next=nachfolger n mit kleinstem f(n)
6.      if Ziel_Erreicht(knoten, ziel) return knoten
7.      if f(next)<f(knoten) return "lokales Maximum"
8.      knoten=next
```

Eine naheliegende Lösung ist es, in solchen Fällen Backtracking einzusetzen [GRS03]. Ein weiteres Verfahren diesem Problem zu begegnen ist das sogenannte *Enforced Hill-Climbing* [HN01], das im Bereich der Handlungsplanung erfolgreich eingesetzt worden ist. Hierbei wird von dem Knoten, der ein lokales Maximum darstellt, solange eine Breitensuche ausgeführt, bis ein Knoten mit niedrigeren Kosten gefunden wird. Von diesem Knoten wird anschließend mittels Hill-Climbing weiter gesucht.

2.6 Verarbeitung von natürlicher Sprache

Da sich diese Arbeit mit der Testfallgenerierung auf Grundlage von in natürlicher Sprache verfassten Anforderungsdokumenten befasst, wird ein kurzer Abriss über die elektronische Verarbeitung von natürlicher Sprache gegeben.

Bei der elektronischen Verarbeitung von natürlicher Sprache wird der Eingabetext in einem Schritt einer Syntaxanalyse unterzogen. Dieser Vorgang erfolgt in zwei Phasen [JM00]: In der ersten Phase wird jedem Wort der Eingabe eine *Wortklasse* (engl.: *part of speech*) (z.B. Verb, Nomen, Artikel) zugewiesen. Im nachfolgenden Schritt wird aus der so vorbereiteten Eingabe ein Syntaxbaum erstellt.

Das Problem bei der Verarbeitung von natürlicher Sprache liegt in der Mehrdeutigkeit von natürlicher Sprache, die in beiden Verarbeitungsschritten auftreten kann. So können Worte verschiedenen Wortklassen zugeordnet werden. Als Beispiel sei hier das englische Wort „book“ genannt, dass sowohl Verb als auch Nomen sein kann. Diese Mehrdeutigkeiten können mit den vorhandenen Methoden recht zuverlässig aufgelöst werden.

Schwieriger wird diese Auflösung bei mehrdeutigen Syntaxbäumen. So kann der Satz „We saw the Eiffel tower flying to Paris.“ zu zwei verschiedenen Syntaxbäumen führen, die beide grammatikalisch korrekt sind. Ein weiteres Beispiel für Mehrdeutigkeiten sind *Koordinations Mehrdeutigkeiten*, wie zum Beispiel „old men and women“. In diesem Beispiel ist die Klammerung mehrdeutig. Der Satz kann als „old [men and women]“ oder als „[old men] and women“ interpretiert werden.

Die momentan weit verbreitetste Methode, um mit Mehrdeutigkeiten umzugehen, ist es, mit statistischen Methoden zu arbeiten, indem zum Beispiel probabilistische kontextfreie Grammatiken verwendet werden. Probabilistische Grammatiken weisen jeder Produktionsregel die Wahrscheinlichkeit ihres Auftretens zu. Die Wahrscheinlichkeiten werden mit Hilfe *annotierter Textcorpora* gewonnen. Dies ist eine Menge von Texten, in der jeder Satz mit einem Syntaxbaum annotiert ist, wobei jeder Syntaxbaum durch Menschen manuell auf seine Korrektheit überprüft worden ist. Ein Beispiel hierfür ist die Penn-Treebank [MMS93].

Ein Parser, der mit solch einer Methode arbeitet, liefert als Ergebnis einen Wald von Syntaxbäumen zurück, wobei den einzelnen Bäumen Wahrscheinlichkeiten zugeordnet sind [Collins03].

Die einzelnen Ergebnisbäume entsprechen somit nur mit einer gewissen Wahrscheinlichkeit der gewollten syntaktischen Bedeutung. Daher ist bereits bei der Syntaxanalyse mit einigen Ungenauigkeiten zu rechnen. Als Konsequenz wird in dieser Arbeit auf Freitext verzichtet und stattdessen zur natürlich-sprachlichen Formulierung von Anforderungen eine *kontrollierte natürliche Sprache* eingesetzt. Auf diese Sprachen wird im folgenden Abschnitt eingegangen.

2.7 Kontrollierte natürliche Sprache

Natürliche Sprache ist sicherlich die am weitesten verbreite Methode, um Wissen auszutauschen. Die Vorteile hierfür liegen auf der Hand: Natürliche Sprache ist für Menschen einfach zu benutzen und zu verstehen und darüber hinaus ist natürliche Sprache ausdrucksstark genug, um auch sehr komplexe Probleme zu beschreiben.

Gerade diese Komplexität führt jedoch zu Problemen, wenn natürliche Sprache durch Computer verarbeitet werden soll, wie im vorhergehenden Abschnitt dargelegt worden ist. Allerdings stellt natürliche Sprache nicht nur Computer vor Probleme. Auch bei der Kommunikation von Mensch zu Mensch kann natürliche Sprache zu Verständnisproblemen führen. Komplexe, verschachtelte Sätze, die häufige Verwendung von Nebensätzen, die den Lesefluss stören, wobei sie – ohne ihre originäre Semantik zu verlieren – viel besser in einzelne, kurze Sätze zerlegt hätten werden können sowie die Verwendung von Fremdwörtern und überflüssige Ausschmückungen führen häufig zu Missverständnissen. Insbesondere bei technischen Dokumenten, die eventuell sicherheitsrelevante Dinge beschreiben, sind solche Missverständnisse möglichst zu vermeiden.

Weiterhin führt die zunehmende Globalisierung dazu, dass immer mehr Dokumente in Englisch verfasst werden, jedoch häufig von Nicht-Muttersprachlern gelesen werden. Speziell für diese Gruppe ist eine klare und einfach strukturierte Sprache sehr hilfreich.

Um diese Probleme anzugehen, können kontrollierte natürliche Sprachen (KNS) eingesetzt werden. Diese Sprache sind eingeschränkte Formen einer natürlichen Sprache: Alle Sätze in der kontrollierten Sprache sind gültige Sätze der natürlichen Sprache, aber nicht alle Sätze der natürlichen Sprache sind korrekte Sätze im Sinne der kontrollierten Sprache [Hebling02].

Wie die Einschränkungen einer KNS aussehen, hängt stark vom Einsatzgebiet der KNS ab. Traditionell werden hierbei zwei Gruppen unterschieden: am Menschen orientierte KNS und computer-orientierte KNS [Schwitter10].

Die Zielrichtung der ersten Gruppe ist es, die Lesbarkeit von natürlicher Sprache zu verbessern, um Probleme bei der Mensch-zu-Mensch Kommunikation zu vermeiden. Solche Sprachen werden häufig auch *vereinfachte* (engl: *simplified*) oder *technische* (engl: *technical*) Sprachen genannt. Die Einschränkungen der Sprache bestehen aus einem *Wörterbuch*, welches das Vokabular vorgibt, das verwendet werden darf und einer Menge von Schreibregeln. Ein Beispiel solch einer Sprache wird nun mit dem *ASD Simplified Technical English* [ASD10], einem Standard der *AeroSpace and Defence Industries Association of Europe,* gegeben.

Um einen Eindruck dieser Sprache zu gewinnen, sind nachfolgend Auszüge aus den Schreibregeln der Sprache aufgeführt.

- Choose the words from:
  - Approved words in the Dictionary
  - Words that qualify as Technical Names
  - Words that qualify as Technical Verbs
- Use approved words from the Dictionary only as the part of speech given.
- Do not use slang words.
- Do not use different Technical Names for the same thing.
- If you have a choice, use the simplest and shortest name.

Wie die ersten beiden Regeln nahelegen, gibt es ein Wörterbuch, dass eine Menge von Wörtern enthält, die in der Sprache verwendet werden dürfen. Das folgende Beispiel zeigt einen Auszug aus dem Wörterbuch des *Simplified Technical English*.

Keyword (part of speech)	**Approved Meaning/ ALTERNATIVES**	**APPROVED EXAMPLE**	**Not Approved**
absolutely (adv)	FULLY	MAKE SURE THAT THE LATCH IS FULLY ENGAGED.	Ensure the latch is absolutly engaged.

Wie an dem angeführten Beispiel zu sehen, sind die Schreibregeln, die die KNS einschränken eher informell gehalten. Dies ist bei der zweiten Gruppe, der computer-orientierten KNS, nicht der Fall.

Die Zielrichtung der zweiten Gruppe ist es, die KNS durch Computer weiterzuverarbeiten. Um dies zu ermöglichen, haben diese Sprachen eine formale Syntax und syntaktisch korrekte KNS Texte können in eine formale Sprache (wie z. B. Prädikatenlogik) abgebildet werden. Ähnlich wie die erste Gruppe, bestehen die KNS der zweiten Gruppe aus den Komponenten:

1. Wörterbuch, das das zu verwendende Vokabular vorgibt.
2. Einer formal definierten Grammatik, die eine eingeschränkte Grammatik einer natürlichen Sprache darstellt.

Zusätzlich zu diesen beiden Komponenten, besitzen KNS der zweiten Gruppe eine weitere Komponente:

3. Eine Menge von Übersetzungsregeln, die die KNS eindeutig auf eine formale Sprache abbilden.

In Kapitel 3 werden Vertreter dieser Gruppe ausführlich dargestellt, sodass an dieser Stelle nicht weiter darauf eingegangen wird.

3 Stand der Forschung

In diesem Kapitel wird ein Überblick über den Stand der Forschung gegeben, der für diese Arbeit relevant ist.

Das Kapitel gliedert sich in zwei Teile, die sich mit den zwei Kernproblemen dieser Arbeit befassen: die Formalisierung von natürlich-sprachlichen Anforderungen und die automatische Testfallgenerierung für Echtzeitsysteme aus formalen Modellen.

Wie bereits angesprochen, wird in dieser Arbeit eine kontrollierte natürliche Sprache zur Erfassung der Anforderungen verwendet. Daher wird zu Beginn auf bestehende kontrollierte natürliche Sprachen zu Anforderungserfassung eingegangen.

3.1 Kontrollierte natürliche Sprachen

3.1.1 Attempto Controlled English

Die Spezifikationssprache *Attempto Controlled English* (ACE) [Schwitter98, FSS99] ist eine kontrollierte Sprache, die ursprünglich zur Beschreibung von Anforderungen entwickelt worden ist. Ein Blick auf die Publikationsseite des Internetauftritts des Attempto Projektes [Attempto] offenbart allerdings einen Wechsel des Schwerpunktes zur Nutzung von Attempto im Rahmen des Semantic Webs.

Wie bereits erwähnt, bestehen kontrollierte Sprachen aus den beiden Komponenten Wörterbuch und einer Menge von Schreibregeln. Dies ist auch bei ACE der Fall. Allerdings werden Schreibregeln im Attempto-Kontext als Konstruktionsregeln bezeichnet und zusätzlich zu diesen beiden Komponenten gibt es in ACE noch *Interpretationsregeln*, die die Semantik von korrekten ACE Sätzen definieren. Diese drei Komponenten werden an dieser Stelle vorgestellt.

3.1.1.1 Wörterbuch

Das Vokabular von ACE besteht aus

- vordefinierten Funktionsworten (z.B. Artikel, Konjunktionen, Präpositionen) und
- benutzerdefinierte, domänenspezifische Worte (Nomen, Verben, Adjektive, Adverbien).

Das Attempto System liefert ein Wortlexikon mit, indem einige Basisworte definiert sind. Darüber hinaus kann der Benutzer eigene Lexika definieren. Hierfür muss der Benutzer jedoch ein gewisses Maß an Grammatikkenntnissen mitbringen. So müssen die zu definierenden Worte unter anderem erst einmal in eine der folgenden Wortklassen eingeteilt werden: Adverbien, intransitive Adjektive, transitive Adjektive, abzählbare Nomen, Stoffnamen, Eigennamen, intransitive Verben, transitive Verben und ditransitive Verben.

3.1.1.2 Konstruktionsregeln

Die Form von ACE Sätzen wird durch Konstruktionsregeln festgelegt. Die Konstruktionsregeln sind so ausgelegt, dass typische Quellen von Ungenauigkeiten in natürlicher Sprache vermieden werden. Eine ACE-Spezifikation ist eine Sequenz von Sätzen. Es gibt

- Einfache Sätze,
- zusammengesetzte Sätze und
- Fragesätze.

Einfache Sätze haben die Form

```
subject + verb + complement + adjunct.
```

Ein Beispiel für solch einen Satz ist

> The driver stops the train at the station.

Zusammengesetzte Sätze werden aus einfachen Sätzen gebildet, wobei die folgenden Konstrukte zum Einsatz kommen:

- Koordination (`and`, `or`)
- Subordination durch Konditionalsätze (`if ... then ...`)
- Subordination durch Relativsätze (`who`, `which`, `that`)
- Negation
- Quantifizierung (`a`, `there is a`, `every`, `for every`)

Um Zusammenhänge zwischen einzelnen Sätzen herzustellen, können ACE Sätze mittels Anaphern[1] verknüpft werden, das heißt, ein Satz kann sich auf vorhergehende Nominalphrasen beziehen. Als Anaphern können Personalpronomen und bestimmte Nominalphrasen verwendet werden. In dem Beispiel

> A passenger of a train alerts a driver. He stops the train.

bezieht sich das Personalpronomen `he` auf die Nominalphrase `a driver` und die bestimmte Nominalphrase `the train` bezieht sich auf die unbestimmte Nominalphrase `a train`.

Des weiteren können in ACE *Fragesätze* gebildet werden, die die folgende Form haben:

- `yes/no` Fragen
- `wh-` Fragen (`who`, `where`...).

1 Der Verweis eines Satzteiles auf einen anderen, vor ihm stehenden Satzteil wird als *Anaphorik* bezeichnet; man sagt auch, der Satzteil baue eine anaphorische Verbindung zu einem anderen Satzteil auf. Der vordere Satzteil wird *Antezedens* genannt, der hintere Satzteil *Anapher* (Quelle: Wikipedia.de)

Lexikalische Einschränkungen

ACE Sätze unterliegen einigen lexikalischen Einschränkungen und typografischen Konventionen, wie zum Beispiel:

- Verben werden nur im *simple present*, *aktiv*, *indikativ* und der dritten Person verwendet (`presses`)
- Keine Modalverben (`may`, `can`, `must`, etc.) oder Verben wie `hope`, `know`, `believe`, etc.
- Keine modalen Adverbien (`possibly`, `probably`, etc.)
- ACE bietet dem Benutzer die Möglichkeit, Synonyme zu definieren
- Wörter können entweder einfach sein (`train`) oder zusammengesetzt (`alarm signal button, alarm-signal-button`)

3.1.1.3 Einschränken von Mehrdeutigkeiten

Ein großes Problem bei der elektronischen Verarbeitung von natürlicher Sprache ist die Mehrdeutigkeit selbiger. Als Beispiel dafür nennen Fuchs et. al. [FSS99] die folgenden drei Sätze:

- `The driver stops the train with the smashed head-light.`
- `The driver stops the train with great effort.`
- `The driver stops the train with the defect brake.`

Man kann feststellen, dass alle drei Sätze die gleiche grammatikalische Struktur aufweisen und dass sie mehrdeutig sind. Der Grund für Letzteres ist die Präpositionalphrase beginnend mit `with`. Diese kann sowohl dem Verb `stops`, als auch dem Nomen `train` zugeordnet werden.

Die ersten beiden Sätze sind für einen menschlichen Leser eindeutig, da der gesunde Menschenverstand nur eine Interpretation zulässt. Der dritte Satz hingegen liefert zwei mögliche, plausible Interpretationen, da sich `with the defect brake` sowohl auf `train`, als auch auf `stops` beziehen kann. In diesem Fall ist Kontextwissen erforderlich, um eine der beiden Interpretationen auszuwählen.

Um die Notwendigkeit von Kontextwissen zu umgehen, wird bei ACE ein Ansatz verwendet, der Mehrdeutigkeiten mittels Syntax auflöst. Hierfür werden drei einfache Mechanismen angewandt:

- einige mehrdeutige Konstrukte sind nicht Teil der Sprache; für diese Konstrukte stellt die Sprache eindeutige Alternativen bereit,
- die verbleibenden mehrdeutigen Konstrukte werden deterministisch mit Hilfe von *Interpretationsregeln* aufgelöst, die auf syntaktischen Informationen beruhen; das Ergebnis dieser Interpretation wird durch Umschreibungen angezeigt,
- der Benutzer kann die ausgewählte Interpretation entweder akzeptieren, oder die Eingabe umformulieren, um eine andere Interpretation zu erlangen.

Einige Beispiele für Interpretationsregeln sind:

- Relativsätze beziehen sich immer auf das Nomen und nicht auf das Verb.
- Präpositionalphrasen in der freien Ergänzung beziehen sich immer auf das Verb und nicht auf evtl. vorhandene Nominalphrasen.

Für den dritten Beispielsatz würde das ATTEMPTO-System die Umschreibung

```
The driver {stops the train with the defect brake.}
```

generieren. Das heißt, die Präpositionalphrase wird dem Verb zugeordnet. Der Grund hierfür liegt in der oben aufgeführten zweiten Interpretationsregel, die besagt, dass Präpositionalphrasen immer dem Verb zugeordnet werden.

Da diese Interpretation wahrscheinlich nicht die vom Benutzter gewünschte ist, muss dieser die Eingabe neu formulieren. Hierzu kann die erste Interpretationsregel verwendet werden, die besagt, dass Relativsätze immer dem Nomen zugeordnet werden. Der Satz

```
The driver stops the train that has a defect brake.
```

resultiert in der Umschreibung

```
The driver stops {the train that has a defect brake.}
```

3.1.1.4 Logische Repräsentation

Attempto kann automatisch in eine formale Sprache übersetzt werden, sodass die Semantik eindeutig definiert ist. Die logische Repräsentation von ACE Sätzen erfolgt durch *Discourse Representation Structures* (DRS) [BB06]. Da weder ATTEMPTO noch DRS im Lösungskonzept dieser Arbeit verwendet werden, wird an dieser Stelle nicht weiter darauf eingegangen. Der interessierte Leser wird auf [FKK07] verwiesen.

3.1.1.5 Bewertung

ATTEMPTO Controlled English ist eine kontrollierte Sprache, die u.a. zur Beschreibung von Anforderungen entwickelt worden ist. Wie jede kontrollierte Sprache muss die Benutzung der Sprache vom Anwender erlernt werden. ACE erhebt den Anspruch, dass damit Anforderungen erfasst werden können, die nicht auf ein bestimmtes Anwendungsgebiet eingeschränkt sind [Schwitter98]. Als Resultat dieses Anspruches ist ACE eine sehr komplexe und damit - aus Sicht des Autors dieser Arbeit - schwer zu verwendende Sprache. Hinzu kommt, dass ACE für Echtzeitsysteme nicht geeignet ist, da die Sprache keine Möglichkeiten vorsieht, Echtzeitbedingungen zu spezifizieren.

Da der Anwendungsbereich dieser Arbeit auf die Spezifikation von funktionalen Anforderungen beschränkt ist, und darüber hinaus ein Fokus auf Anforderungsspezifikationen aus dem Bereich der Steuergeräte für Automobile besteht, wird im Rahmen dieser Arbeit eine KNS entwickelt, die diesen Tatsachen Rechnung trägt. Das Konzept der Interpretationsregeln wird für diese Sprache von ACE übernommen.

3.1.2 Processable English

Wie im vorhergehenden Abschnitt erwähnt, muss jede kontrollierte Sprache erlernt werden, bevor sie benutzt werden kann. Mit steigender Komplexität der Sprache steigt dieser Aufwand ebenso wie die Wahrscheinlichkeit, dass der Benutzer Fehler bei der Benutzung der Sprache macht und inkorrekte Sätze bildet. Die Akzeptanz der Benutzung von kontrollierter Sprache hängt daher unter anderen davon ab, ob dem Verfasser Unterstützung durch Software angeboten werden kann, die die Eingabe auf Richtigkeit prüft und Korrekturvorschläge anbietet [WH97].

Ein Beispiel für solch eine Softwareunterstützung ist in [SLH03] zu finden. In dieser Veröffentlichung ist ein Editor für die kontrollierte Sprache *Processable English* (PENG) beschrieben, der dem Benutzer Hilfestellung bei der Eingabe von PENG Sätzen bietet. Auf die Einzelheiten der KNS selber wird an dieser Stelle nicht eingegangen, da PENG mit ACE vergleichbar ist.

Der Editor führt den Benutzer bei der Eingabe, in dem er laufend die Eingabe analysiert und dem Benutzer Hinweise gibt, welche Eingabemöglichkeiten als Nächstes zur Verfügung stehen. Die Autoren erläutern dies anhand des Beispielsatzes `A person lives in Dreadsbury Mansion`. Während der Eingabe liefert der Editor die folgenden Hinweise:

```
A [ adjective | common noun ]
A person [ verb | negation | relative sentence ]
A person lives [ '.' | prepositional phrase | adverb ]
```

Die Vorschläge sind als Hyperlink implementiert, sodass ein Mausklick auf diese den Benutzer einen Hilfetext anbietet. Gibt der Benutzer Worte ein, die bisher noch nicht im Lexikon vorhanden sind, so bietet das System dem Benutzer einen Dialog an, in dem das Lexikon mit diesem Begriff vervollständigt werden kann.

3.1.2.1 Bewertung

Für die PENG gelten die gleichen Vor- und Nachteile wie für ACE. Interessant ist jedoch die verwendete Werkzeugunterstützung, die dem Benutzer Hilfestellung bei der Eingabe anbietet.

3.1.3 Kontrollierte Sprachen für Logiken

In diesem Abschnitt werden verschiedene kontrollierte Sprachen vorgestellt, die eine natürlich-sprachliche Darstellung einer formalen Sprache ermöglichen. Allen vorgestellten Ansätzen ist hierbei gemein, dass das Einsatzgebiet dieser Logiken nicht der Anforderungsbeschreibung der Funktionalität der Software dient. Stattdessen dienen die Logiken zur Beschreibung von Anforderungen an Entwicklungsartefakte, die während des Softwareentwicklungsprozesses entstehen. Insbesondere dienen die Logiken der Verwendung im Rahmen der automatischen Modellprüfung und der Beschreibung von OCL Bedingungen in UML-Modellen.

KNS zur Spezifikation von Eigenschaften für formale Methoden

Formale Methoden, wie automatische Modellprüfung (engl.: *model checking*) [CGP99], oder automatische Theorembeweiser dienen dazu, für ein formales Modell nachzuweisen, dass dieses gewissen Eigenschaften (nicht) genügt.

Um dies zu bewerkstelligen, müssen diese Eigenschaften formal notiert werden. Dies kann z.B. durch temporale Logiken wie CTL oder LTL [CGP99] geschehen. Eine Eigenschaft mittels einer dieser Logiken zu beschreiben erfordert jedoch eine gewisse Erfahrung und stellt eine Hürde für den Einsatz dieser Technik dar.

Um die Überwindung dieser Hürde zu erleichtern, wird in [DAC99] eine Menge von Mustern zur Beschreibung von häufig vorkommenden Eigenschaften beschrieben. Jedes Muster wird durch einen Freitext beschrieben, in welchen Situationen dieses Muster anwendbar ist und es werden Beispiele für die Anwendung bereitgestellt.

Auf Basis dieser Muster ist in [SAC03] eine kontrolliert-natürliche Sprache beschrieben, wobei in der Veröffentlichung der Begriff *Disciplined Natural Language* (DNL) verwendet wird. Die Veröffentlichung beschreibt ein Werkzeug Namens PROPEL, das den Nutzer anhand eines Fragekatalogs zu einem geeigneten Muster führt. Ein Beispiel für eine Frage aus diesem Katalog ist das Folgende:

> How many events are in your behavior?
> One Event:
> 1. The event must happen.
> 2. The event must NOT happen.
> Two Events:
> 3. The first event causes the second event to happen.
> 4.The first event enables the second event to happen.

Nachdem der Fragekatalog durch den Benutzer abgearbeitet worden ist, wird dem Benutzer ein passendes Muster präsentiert. Jedes Muster wird mit Hilfe der DNL dargestellt, wobei die DNL selber aus Lückentexten besteht, die der Benutzer an den entsprechenden Stellen ausfüllen kann.

Parallel zu der DNL-Darstellung bietet das Werkzeug eine formale Darstellung der Eigenschaft in Form eines endlichen Automaten an. Der Benutzer kann gewisse Merkmale des Automaten ändern, die sich in Änderungen an der DNL-Darstellung niederschlagen, genauso wie sich Änderungen in der DNL-Darstellung in der Automaten-Darstellung niederschlagen. Ein Beispiel für eine ausgefüllte DNL-Darstellung und der zugehörigen Automaten-Darstellung ist in Abbildung 11 zu sehen.

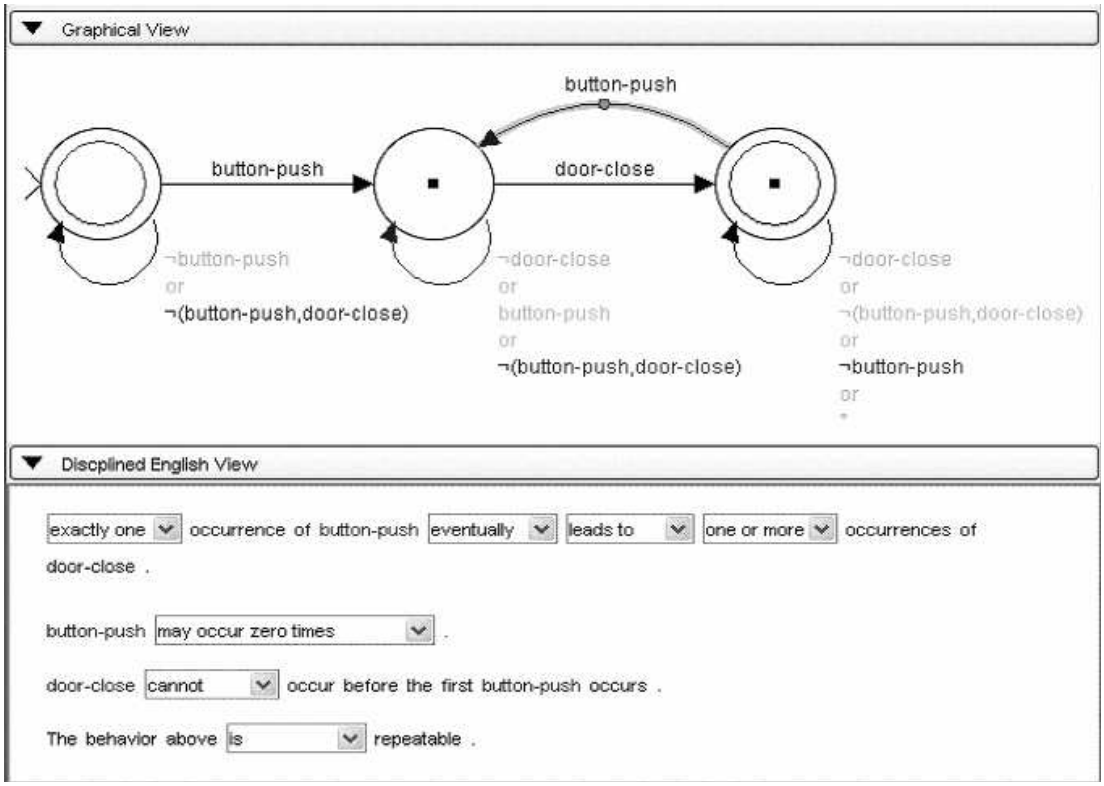


Abbildung 11: Bildschirmfoto des PROPEL Werkzeugs

Die Muster aus [DAC99] bieten keine Möglichkeit, Echtzeiteigenschaften zu spezifizieren. Somit gilt diese Einschränkung auch für die gerade vorgestellte kontrollierte Sprache.

Um diese Einschränkung aufzuheben, wird in [KC05, KC05a] eine Arbeit präsentiert, die die in [DAC99] vorgestellten Muster um Echtzeiteigenschaften erweitert und zur natürlich-sprachlichen Beschreibung der Eigenschaften eine kontrollierte Sprache entwickelt.

Die Syntax dieser Sprache ist durch eine kontextfreie Grammatik definiert, die in Abbildung 12 abgebildet ist. In dieser Abbildung sind Nicht-Terminale kursiv dargestellt und `P,Q,R,S,T,Z` und `c` sind Platzhalter.

[KC05] beschreibt ein Werkzeug, welches den Benutzer durch das Erstellen einer Echtzeiteigenschaft führt. Ausgehend vom Nicht-Terminal `property` führt dieses Werkzeug den Benutzer Schritt für Schritt durch die Grammatik, indem dem Benutzer für jedes Nicht-Terminal alle Möglichkeiten angeboten werden und der Benutzer jeweils eine dieser Möglichkeiten auswählt. Abbildung 13 zeigt ein Bildschirmfoto von diesem Vorgang.

Die so entstandenen kontrolliert-natürlich sprachlichen Eigenschaften können anschließend automatisch auf eine Echtzeitlogik abgebildet werden [KC05a]. Als mögliche Ziellogiken stehen hierbei MTL [Koymans90], TCTL [Alur91] und RTGIL [MRK+97] zur Verfügung.

In [FM00] ist ein ähnlicher Ansatz beschrieben, der ebenfalls eine KNS, deren Syntax durch eine kontextfreie Grammatik spezifiziert ist, auf eine Echtzeitlogik abbildet.

Start	1: property	::=	*scope* “, ” *specification* “.”
Scope	2: scope	::=	“Globally” | “Before ” R | “After ” Q | “Between ” Q “ and ” R | “After ” Q “ until ” R
General	3: specification	::=	*qualitativeType* | *realtimeType*
Qualitative	4: qualitativeType	::=	*occurrenceCategory* | *orderCategory*
	5: occurrenceCategory	::=	*absencePattern* | *universalityPattern* | *existencePattern* | *boundedExistencePattern*
	6: absencePattern	::=	“it is never the case that ” P “ holds”
	7: universalityPattern	::=	“it is always the case that ” P “ holds”
	8: existencePattern	::=	P “ eventually holds”
	9: boundedExistencePattern	::=	“transitions to states in which ” P “ holds occur at most twice”
	10: orderCategory	::=	“it is always the case that if ” P “ holds” (*precedencePattern* | *precedenceChainPattern1-2* | *precedenceChainPattern2-1* | *responsePattern* | *responseChainPattern1-2* | *responseChainPattern2-1* | *constrainedChainPattern1-2*)
	11: precedencePattern	::=	“, then ” S “ previously held”
	12: precedenceChainPattern1-2	::=	“ and is succeeded by ” S “, then ” T “ previously held”
	13: precedenceChainPattern2-1	::=	“, then ” S “ previously held and was preceded by ” T
	14: responsePattern	::=	“, then ” S “ eventually holds”
	15: responseChainPattern1-2	::=	“, then ” S “ eventually holds and is succeeded by ” T
	16: responseChainPattern2-1	::=	“ and is succeeded by ” S “, then ” T “ eventually holds after ” S
	17: constrainedChainPattern1-2	::=	“, then ” S “ eventually holds and is succeeded by ” T “, where ” Z “ does not hold between ” S “ and ” T
Real-time	18: realtimeType	::=	“it is always the case that ” (*durationCategory* | *periodicCategory* | *realtimeOrderCategory*)
	19: durationCategory	::=	“once ” P “ becomes satisfied, it holds for ” (*minDurationPattern* | *maxDurationPattern*)
	20: minDurationPattern	::=	“at least ” c “ time unit(s)”
	21: maxDurationPattern	::=	“less than ” c “ time unit(s)”
	22: periodicCategory	::=	P “ holds ” *boundedRecurrencePattern*
	23: boundedRecurrencePattern	::=	“at least every ” c “ time unit(s)”
	24: realtimeOrderCategory	::=	“if ” P “ holds, then ” S “ holds ” (*boundedResponsePattern* | *boundedInvariancePattern*)
	25: boundedResponsePattern	::=	“after at most ” c “ time unit(s)”
	26: boundedInvariancePattern	::=	“for at least ” c “ time unit(s)”

Abbildung 12: Grammatik zur Beschreibung von Eigenschaften mit Echtzeit [KC05]

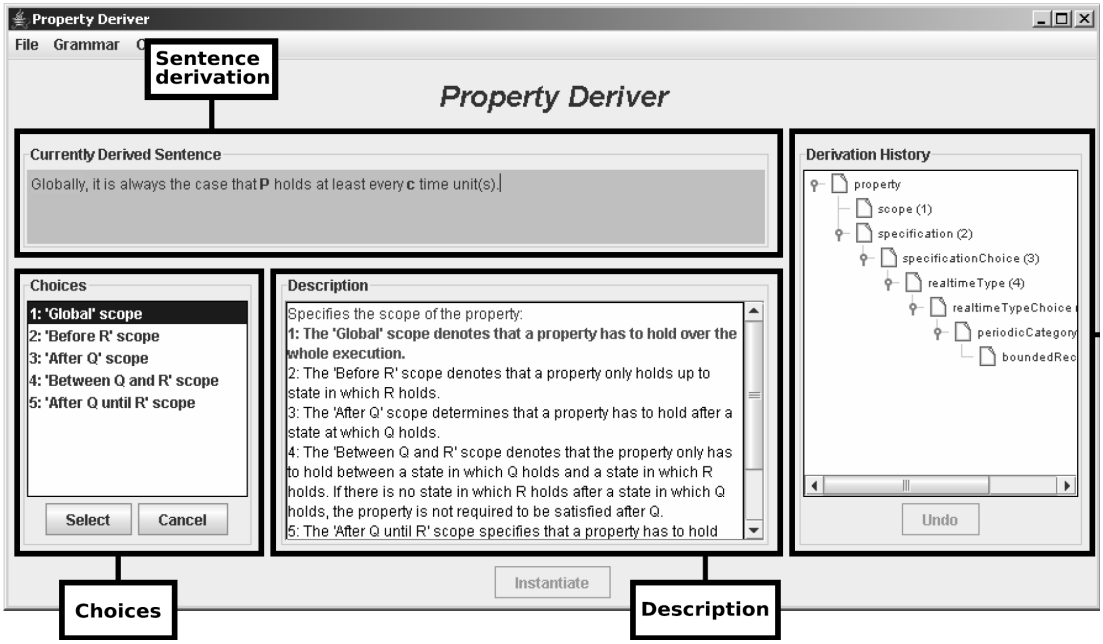


Abbildung 13: Bildschirmfoto des Werkzeuges aus [KC05]

KNS zur Beschreibung von OCL Bedingungen

In [Johannis07] wird ein Ansatz vorgestellt, der es ermöglicht, Bedingungen in der *Object Constraint Language* (OCL) durch eine kontrolliert-natürliche Sprache zu spezifizieren. Die OCL ist Bestandteil der *Unified Modeling Language* (UML) und kann benutzt werden, um innerhalb von UML-Modellen Bedingungen zu formulieren. So können zum Beispiel Invarianten in Klassendiagrammen definiert werden, oder Methodenaufrufe können mit Vor- und Nachbedingungen versehen werden.

3.1.3.1 Bewertung

Die beschriebenen Sprachen haben einen klar umrissenen Einsatzbereich: die Beschreibung von zu prüfenden Eigenschaften im Rahmen des Einsatzes von formalen Methoden oder der Beschreibung von OCL Bedingungen innerhalb von UML-Modellen. Dementsprechend orientieren sich die Sprachen stark an den formalen Methoden für die die Sprachen eine natürlich-sprachliche Schnittstelle bilden. Dies steht im Widerspruch zur Zielsetzung dieser Arbeit.

3.1.4 Satzmuster für Echtzeitsysteme

Das Ziel der Arbeit [Denger02, DBK03] von Denger et. al. ist es, dem Ersteller von Anforderungen von eingebetteten Systemen eine Menge von Muster zur Verfügung zu stellen, die den Autor durch den Prozess der Anforderungserstellung leiten, sodass die resultierenden Anforderungen vollständiger, präziser und weniger mehrdeutig sind. Die beschriebenen Muster sind nicht formalisiert und erlauben darüber hinaus uneingeschränkten Freitext. Die Arbeit ist somit in die Gruppe der KNS mit Fokus auf Mensch-zu-Mensch Kommunikation einzuordnen.

Zusätzlich zu den Mustern werden Regeln eingeführt, die zusammen mit den Mustern genutzt werden sollen. Diese Regeln beschreiben, in welchen Situationen welche Muster eingesetzt werden sollen, sodass die Qualität der Anforderungen erhöht wird.

Im Folgenden wird nun eine Auswahl der Muster vorgestellt. Die vollständigen Muster sowie Regeln können in [Denger02] nachgelesen werden.

3.1.4.1 Satz Muster (*Sentence pattern*)

Mit dem Muster `SP2` wird ein diskretes Verhalten beschrieben, das heißt, eine Systemreaktion wird durch ein Ereignis ausgelöst.

> **SP2:***Phrase that contains an event* (EVENT) **then** *Phrase that contains a reaction* (REACTION)
>
> **Beispiel**: „If the system receives ‘abc’ (EVENT), then the system activates the motor (REACTION)”

Mit dem Muster `SP3` wird ein kontinuierliches Verhalten beschrieben, das, nachdem es durch ein Ereignis gestartet worden ist, solange ausgeführt wird bis entweder ein anderes Ereignis eintrifft oder eine Bedingung wahr wird.

SP3: *Phrase that contains a start event* (EVENT)
Phrase that contains a continues behaviour (COMPUTATION) **for as long as** *Phrase that contains a condition* (CONDITION) | **until** *Phrase that contains a stop event* (EVENT)

Beispiel: "The system displays the oil pressure on the LCD (COMPUTATION) for as long as the system is active (CONDITION)"

3.1.4.2 Ereignis Muster

Um Ereignisse beschreiben zu können, werden *Event Pattern* benutzt. Einige dieser Muster sind hier aufgeführt. So beschreibt das folgende Muster `EP1` ein Ereignis, das durch einen Wechsel des Wertes einer Variablen ausgelöst wird.

EP1: <**When | If**> (conjunction) *noun phrase* (VARIABLE) *verb* (VALUE CHANGE) *numeral adjective* (VARIABLE VALUE) | *noun phrase* (VARIABLE)

Beispiel: „If (conjunction) the water pressure (VARIABLE) rises above (VALUE CHANGE) the maximum value (VARIABLE),..."

In eingebetteten Systemen spielt das Zeitverhalten des Systems eine wichtige Rolle. So kann ein Ereignis zum Beispiel durch das Verstreichen von Zeit eintreten. Denger trägt dem in seiner Arbeit Rechnung, in dem er Muster einführt, die Zeitverhalten beschreiben können. So beschreibt das Muster `EP4` ein Ereignis, das eine bestimmte Zeit nach einem anderem Ereignis/einer Reaktion ausgelöst wird.

EP4:<**at the earliest after | at the latest after |at the earliest after** *time*, **but at the latest after**> *time* (variable of type Time) **after** (conjunction) < *reference to an event | reference to a reaction* >

Beispiel: "At the latest after (conjunction) 5 sec (variable of type Time) after the system received the signal 'stop' (reference to an event),..."

Wie bereits erwähnt, hat Denger zusätzlich zu den Mustern Regeln entwickelt, wie die Muster zu verwenden sind. An dieser Stelle soll dies beispielhaft durch die folgende Regel verdeutlicht werden:

[Rule23]: Specify an event reference by repeating the referenced event in past tense. Exception:The event needs not to be repeated if and only if the referenced event is stated in its own sentence that contains only the referenced event and the reference to the event is needed in the sentence that directly follows the event.

Im Falle der Anforderung `The system receives the signal stop. After 5 sec...` muss laut der obigen Regel das Ereignis, auf das sich die Zeitangabe bezieht nicht wieder-

holt werden. In der Anforderung `The system receives the signal stop, then the system activates the engine. After 5 sec after the system received the signal stop,...` muss das Ereignis hingegen wiederholt werden, um klar zu stellen, auf welches Ereignis sich die Zeitangabe `after 5 sec` bezieht.

3.1.4.3 Muster für Bedingungen

Neben den Mustern für Ereignisse hat Denger auch Muster zum Beschreiben von Bedingungen entworfen. Bevor diese beschrieben werden, muss erst einmal der Unterschied zwischen Bedingungen und Ereignissen klargestellt werden: Ein Ereignis ist eine Änderung eines Wertes einer Variablen oder des Systemzustandes, während eine Bedingung ein Test einer Wertbelegung einer Variable, eines Signals oder des Systemzustandes ist.

Als Beispiel für ein Muster für Bedingungen sei das `BCP` Muster für einfache boolesche Bedingungen genannt.

BCP: **If** (conjunction) *noun phrase* (VARIABLE)
is | is not(CURRENT VALUE)
<*adjective phrase* (comparison statement, e.g. greater than, less than, etc.)>
comparison complement (VARIABLE VALUE) | (VARIABLE)
<**TCP1 - TCP6**> (DURATION)
then (conjunction)<**ERP1**>

Beispiel: "If (conjunction) the battery value (VARIABLE) is (CURRENT VALUE) less than (comparison statement) 9 V (VARIABLE VALUE) for at least 4 sec (DURATION), then..."

Die (optionale) Dauer für die die Bedingung gelten muss, wird durch die Muster `TCP1` – `TCP6` dargestellt. Im obigen Beispiel kommt das Muster `TCP2` zum Einsatz (`for at least 4 sec`).

TCP1: **for** *time* (variable of the type Time)
TCP2: **for at least** *time* (variable of the type Time)
TCP3: **for not more than** | **for at most** *time* (variable of the type Time)
TCP4: **TCP2 (but|and) TCP3**
TCP5: **for as long as** | **while ABCP** | **BCP** | **SCP**
TCP6: **until EP1** | **EP2** | **EP3**

Die Bedingung kann mit dem optionalen `ERP1` Muster ergänzt werden, mit dem die Systemreaktion beschrieben werden kann, falls die erste Bedingung nicht erfüllt worden ist.

ERP1: **else**
RPx | **CCP** (exceptional REACTION)

3.1.4.4 Muster für Systemreaktionen

Denger unterscheidet bei den Systemreaktionen zwischen Systemreaktionen mit Bedingungen und Reaktionen ohne Bedingungen. Systemreaktionen ohne Bedingungen werden zum Beispiel durch das folgende Muster realisiert:

RP1: <**EP1** | **EP2** | **EP3** | **EP4**> (BEGIN)
noun phrase (ACTOR)
verb (ACTION)
noun phrase (ACTUATOR)
<*Phrase that contains a timed condition*> (DURATION) |
<*Phrase expressing a completion time*> (COMPLETION)
<*Phrase expressing how the reaction is realized*> (REALIZATION)

Beispiel: “The system receives the signal ‘xyz’ (EVENT). After that, when the battery power exceeds 9V (BEGIN), the system activates the engines (REACTION).”

Sollen die Systemreaktion nur in bestimmten Systemzuständen ausgeführt werden, so kann das *conditioned reaction pattern* angewendet werden.

CRP: **BCP** | **SCP** | **TCPx** (CONDITION)
RP1 | **RP2** (REACTION)

Beispiel:“Event. If the current speed is greater than 60 km/h and if the distance to a car in front of the vehicle is less than 30 meters, then the system initializes the automatic braking”

3.1.4.5 Bewertung

Die Arbeit von Denger hat zum Ziel die Genauigkeit von natürlichsprachlichen Anforderungen aus der Domäne der eingebetteten Systeme zu verbessern, indem dem Autor von Anforderung Muster für die Formulierung an die Hand gegeben werden. Diese Menge von Satzmustern bildet eine sehr gute Grundlage für die Entwicklung einer kontrollierten Sprache zur Formulierung von Anforderungen aus dem Bereich der reaktiven Echtzeitsysteme.

Einige der verwendeten Konzepte finden sich daher auch in der KNS wieder, die im Lösungskonzept beschrieben wird. Insbesondere die Beschreibung von Zeitinformationen sowie die Unterteilung in Bedingungen, Ereignisse und Systemreaktionen.

3.2 Modellbasiertes Testen von Echtzeitsystemen

Die Idee des modellbasierten Testens ist auch im Bereich der Echtzeitsysteme in verschiedenen Veröffentlichungen diskutiert worden. In diesem Abschnitt wird ein Überblick über den Stand der Forschung für dieses Gebiet gegeben. Dazu wird zuerst auf Verfahren zur Modellierung von Echtzeitverhalten eingegangen, anschließend werden Verfahren zur Testfallgenerierung vorgestellt und schließlich wird die Behandlung von Nichtdeterminismus diskutiert.

3.2.1 Modellierung des Echtzeitverhaltens

Viele Arbeiten beschreiben Testfallgenerierung auf Grundlage von formalen Methoden wie Timed Automata [HNT+99,CO00, SVD01, Khoumsi02, HLN+03, NS03, HP04, KJM04, KT04a, KT04, LMN04, LMN+05, Hessel07], echtzeitfähigen temporalen Logiken [MMM95, PMM00, HJL03] oder zeitbehafteten Petri-Netzen [BFM97]. Da dies der Zielsetzung dieser Arbeit widerspricht, soll hierauf nicht weiter eingegangen werden.

Eine Arbeit, die Echtzeitverhalten mit Hilfe einer kontrollierten natürlichen Sprache beschreibt, ist die Arbeit von Esser und Struss [ES07]. Da die Arbeit in Problemstellung und Zielsetzung der vorliegenden Arbeit entspricht, wird auf diese Arbeit näher eingegangen.

Esser und Struss schlagen zur Erfassung von Anforderungen eine kontrollierte natürliche Sprache vor, die aus einer Menge von Lückentexten besteht, mit deren Hilfe typische Anforderungen an Steuergeräte aus der Automobilindustrie erfasst werden können. Die Lücken in den Lückentexten können dabei durch Zustandsbeschreibungen und Zeitinformationen vervollständigt werden.

Jede erfasste Anforderung wird anschließend in eine *Formal Requirement Language* (FRL) genannte formale Sprache übersetzt. In dieser Sprache werden Anforderungen als Tupel (*start-condition*, *consequence*, *end-condition*) beschrieben, wobei die Bedingungen und Konsequenzen durch Ausdrücke spezifiziert werden, die in der Lage sind Systemzustände, Zeitintervalle und Zeitbedingungen ausdrücken.

Abbildung 14 zeigt für drei Beispielanforderungen deren Erfassung durch Lückentexte, die Übersetzung in die FRL sowie eine grafische Darstellung der jeweiligen FRL.

	Prosa Requirement	Requirement Sentence Template	In FRL $R = A \overset{\text{then}}{\rightarrow} B \overset{\text{until}}{\underset{\text{first}}{\rightarrow}} C$	Graphical
R_1	If button B4 has not been down during the last 5 seconds, then lamp L3 must be lit immediately after button B4 changed to down.	If Button B_4 is not down P_1 held the last 5 seconds T_1 and now Button B_4 is down P_2 occurs, then must Lamp L_3 is lit P_3 follow immediately.	$A = [P_1]_{t1}^{t2} \wedge [[P_2]_{t3}^{t4} \wedge (t_2 \mid t_3) \wedge (t_1 = t_2 - T_1)$ $B = [P_3]_{t5}^{t6} \wedge (t_3 \mid t_5)$	T_1, P_1, $\neg P_2$, P_2, if, then, P_3
R_2	Immediately after button B4 is released while the system is in mode m_1 and lamp L3 is not lit, the lamp L3 must be lit until button B4 is down again or the system leaves m_1.	If Button B_4 is down P_1 occurs, during System is in mode m_1 AND Lamp L_3 is not lit P_2 holds, then Lamp L_3 is lit P_3 must hold immediately, until Button B_4 is up OR System is not in mode m_1 P_4 hold.	$A = [[P_1]_{t_1}^{t_2} \wedge [P_2]_{t_3}^{t_4} \wedge (t_3 < t_1 < t_4)$ $B = [P_3]_{t_5}^{t_6} \wedge (t_1 \mid t_5)$ $C = [P_4]_{t_7}^{t_8} \wedge (t_6 \mid t_7)$	until, $\neg P_4$, P_4, P_2, if, $\neg P_1$, P_1, then, P_3
R_3	Immediately after button B4 is pressed, Lamp L3 must be red for 10 seconds.	If Button B_4 is down P_1 occurs, then Lamp L_3 is red P_2 hold immediately, until 10 seconds T_1 elapsed.	$A = [[P_1]_{t_1}^{t_2}$ $B = [P_2]_{t_3}^{t_4} \wedge (t_2 \mid t_3)$ $C = (t_4 = t_3 + 10s)$	T_1, until, true, if, $\neg P_1$, P_1, then, P_2

Abbildung 14: Beispielanforderungen und Übersetzungen aus [ES07]

Die Menge der durch FRL ausgedrückten Anforderungen bildet das sogenannte *OK-Modell*. Dieses Modell stellt das erwartete Verhalten des Testobjekts dar.

Die Arbeit beschreibt weiterhin eine Menge von Mutationsoperatoren, die nach Ansicht der Autoren typische Fehler bei der Entwicklung von Software für Steuergeräte widerspiegelt. Diese Mutationsoperatoren werden benutzt, um mutierte Modelle zu erzeugen.

Nachdem mutierte Modelle erzeugt worden sind, wird für jedes mutierte Modell eine Sequenz von Eingaben gesucht, sodass sich die Ausgabe des mutierten Modells von der Ausgabe des OK-Modells unterscheidet. Solch eine Sequenz dient schließlich als Testfall. Auf den Aspekt, wie solch eine Sequenz gefunden werden soll, wird in der genannten Veröffentlichung nicht näher eingegangen. Es existieren keine neueren Veröffentlichungen, die dieses Problem klären.

3.2.2 Testfallgenerierung

Die Idee des modellbasierten Testens ist in den Grundlagen beschrieben worden. Dort ist dargelegt worden, dass aus einem formalen Modell eine Menge von Testfällen generiert wird, die einem definierten Auswahlkriterium genügt. Somit beschreibt der Vorgang der Testfallgenerierung ein Suchproblem*:* Das Modell beschreibt einen Suchraum und die Auswahlkriterien beschreiben das Ziel der Suche.

Ein großes Problem hierbei ist, dass der Zustandsraum, den die Modelle beschreiben exponentiell in der Anzahl der verwendeten Variablen wächst. Dieses Problem ist als *Zustandsraumexplosion* bekannt. Verfahren, die den gesamten Zustandsraum im Speicher halten müssen, sind somit nur für Modelle mit geringer Größe geeignet.

Die Arbeit [HLN+03] verwendet den Model-Checker Uppaal [BDL04, Upp10] um eine Testsuite zu erzeugen, die Knoten- oder Kantenüberdeckung auf einem Netzwerk von Timed Automata garantiert. Um dem Problem der Zustandsraumexplosion zu begegnen, wird eine A*-Suche verwendet. Die Schätzfunktion zum Abschätzen der verbleibenden Kosten muss hierbei jedoch spezifisch für das verwendete Modell, dessen Zustandsraum durchsucht wird, angegeben werden [BFH+01]. Das heißt, für jedes Modell muss eine eigene Schätzfunktion vom Benutzer zur Verfügung gestellt werden. In [KHD+06, KWN+08] werden heuristisch geführte Suchen für Timed Automata vorgestellt, wobei hier Heuristiken aus dem Bereich der automatischen Handlungsplanung verwendet werden. Diese Arbeiten sind jedoch nicht im Zusammenhang mit Testfallgenerierung zum Einsatz gekommen.

Die Suche nach einer Sequenz in einem Netzwerk von Automaten, die alle Knoten oder Kanten der einzelnen Automaten des Netzwerkes überdeckt, verschärft das Problem der Zustandsraumexplosion, da zusätzlich zu allen Zuständen S des Netzwerkes alle möglichen Überdeckungsmengen C gespeichert werden müssen. Dies kann man sich klar machen, indem man sich überlegt, dass für jeden Knoten/jede Kante eine zusätzliche Variable eingeführt werden muss, die speichert, ob der Knoten/die Kante überdeckt ist. Die Größe des Zustandsraumes entspricht der Größe des kartesischen Produktes von S und C [HP07]. Die Arbeiten [HP04] und [HP07] befassen sich mit diesem Problem und stellen Lösungen vor, die die Größe des zu durchsuchenden Raumes auf die Größe von S beschränken und dennoch eine Kanten-/Knotenüberdeckung erzielen.

Ein anderes Verfahren, um mit dem Problem der Zustandsraumexplosion bei der Testfallgenerierung zu umgehen, ist *Online-Testen* [KT04a, LMN04, LMN+05, Tron10]. Hierbei wird der Testfall nicht bereits vor der Testdurchführung generiert, sondern die Testerzeugung erfolgt parallel zur Testdurchführung. Hierfür wird randomisiert eine Eingabe aus dem Modell erzeugt und diese Eingabe wird an das Testobjekt weitergereicht. Anschließend wird

das Ausgabe- und Zeitverhalten des Testobjektes beobachtet und überprüft, ob es konform zu dem Verhalten des Modells ist. Ist dies der Fall, wird erneut eine Eingabe aus dem Modell ausgewählt. Dieser Prozess wird solange wiederholt, bis entschieden wird den Test abzubrechen oder ein Fehler entdeckt wird. Entscheidungskriterien für einen Abbruch des Tests sind z. B. die Dauer des Testes oder die Anzahl der Testschritte. Online-Testen hat den Vorteil, dass es dem Problem der Zustandsraumexplosion entgeht, da immer nur der aktuelle Zustand des Modells im Speicher gehalten werden muss. Der Nachteil des Online-Testens ist jedoch, dass kein Überdeckungskriterium garantiert werden kann.

In [HRV+03] wird eine Fallstudie vorgestellt, die untersucht, inwieweit der Ansatz von Model-Checking für die Testfallgenerierung aus Modellen mit Größen, wie sie in der industriellen Praxis vorkommen, geeignet ist. Die Studie befasst sich nicht mit Echtzeitsystemen, die Ergebnisse lassen sich aber auf Echtzeitsysteme übertragen.

Die Studie kommt zu dem Schluss, dass weder explizites noch symbolisches Model-Checking für die Testfallgenerierung geeignet ist, da diese Ansätze zu schlecht in der Größe der Modelle skalieren. Die Methode des *Bounded-Model-Checking* mit Hilfe von SAT-Solver [BCC+99] hat sich jedoch als vielversprechend herausgestellt. Das Verfahren ist ähnlich dem Verfahren „Planung als Erfüllbarkeitsproblem“ was im Grundlagenkapitel vorgestellt worden ist. Es existieren Verfahren, die das Prinzip des Bounded-Model-Checking durch SAT-Solver auf Echtzeitsyteme übertragen haben [ACK+02, Sorea02], somit steht diese Methode als Testfallgenerierungsverfahren für Echtzeitsysteme zur Verfügung.

Andere Arbeiten verwenden Verfahren aus der automatischen Handlungsplanung zur Testfallgenerierung (z. B. [MSD+99, MPS99, FL00, AZS+02, SG10]). Diese Arbeiten behandeln jedoch keine Echtzeitsysteme.

Pretschner vergleicht in [Pretschn01, Pretschn03] verschiedene Suchstrategien für die Testfallgenerierung und kommt zu dem Schluss, dass heuristische Suchen für den Fall, dass eine gute Heuristik zur Verfügung steht, die zu bevorzugende Lösung darstellt. Für den Fall, dass keine Heuristik zur Verfügung steht, schlägt er eine randomisierte Tiefensuche vor. Eine andere Arbeit, die ebenfalls informierte Suchen zur Testfallgenerierung verwendet, ist [CLP04]. In [CLP04] wird die Leistung von Vorwärts- und Rückwärtssuchen verglichen, wobei in beiden Fällen eine best-first Suche zum Einsatz kommt. Für die untersuchten Fallstudien wird festgestellt, dass Rückwärtssuche der Vorwärtssuche überlegen ist. Weder Pretschner noch [CLP04] befassen sich mit Echtzeitsystemen.

3.2.3 Behandlung von Nichtdeterminismus

Im Grundlagenkapitel ist dargelegt worden, dass Anforderungen, die Nichtdeterminismus beschreiben, spezielle Ansprüche an die Testfälle stellen: Testfälle, die nichtdeterministisches Verhalten testen, müssen adaptive Testfälle sein.

Einige Arbeiten (z. B. [HLN+03, HP04, HP07]) umgehen das Problem des Nichtdeterminismus, indem sie Spezifikation von nichtdeterministischen Verhalten verbieten. Das heißt, es wird verlangt, dass alle Ausgaben *isoliert* und *dringend* sind. Andere erzeugen adaptive Testfälle. Das trifft zum Beispiel für die Ansätze des Online-Testens zu.

[HNT+99] betrachtet Testfallgenerierung für Timed Automata mit nichtdeterministischem Zeitverhalten und definiert *must*- und *may-traceabilty* für die Testsequenzen. Musttraceabilty bezeichnet Sequenzen, bei denen das Zeitverhalten der Eingabeaktion so gewählt werden kann, dass die Sequenzen immer ausgeführt werden können, egal, zu welchen Zeit-

punkten die Ausgaben des Testobjektes auftreten. Testfälle, die das Kriterium der must-traceability erfüllen, sind demnach in der Lage, nichtdeterministisches Zeitverhalten zu testen, ohne das es sich um adaptive Testfälle handelt. May-tracabilty hingegen bezeichnet Sequenzen, bei denen dies nicht möglich ist.

4 Lösungskonzept

In diesem Kapitel wird das entwickelte Lösungskonzept vorgestellt. Abbildung 15 zeigt einen Überblick über das entwickelte Vorgehen. Der Prozess entspricht weitestgehend dem Prozess des modellbasierten Testens, mit dem Unterschied im ersten Schritt, der die informellen Anforderungen durch eine kontrollierte natürliche Sprache erfasst und somit den Schritt von den informellen Anforderungen zu einer formalen Darstellung vereinfacht.

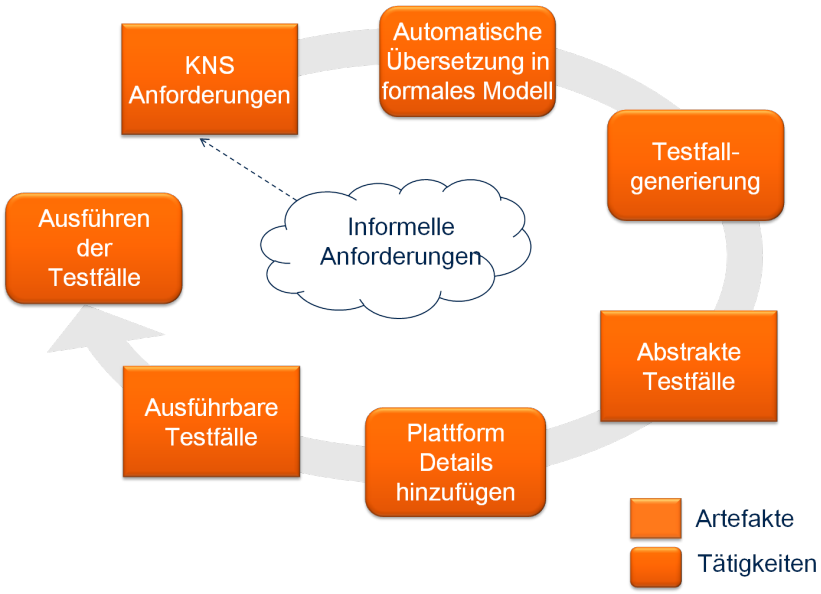


Abbildung 15: Überblick über das Lösungskonzept

Im Folgenden wird das Lösungskonzept vorgestellt, wobei die Struktur des Kapitels der Struktur des Prozesses folgt. Zunächst wird ein Beispiel für informell formulierte Anforderungen aus der Automobilindustrie eingeführt. Anschließend wird die kontrollierte natürliche Sprache eingeführt und dargelegt, wie diese eindeutig auf ein formales Modell abgebildet werden kann.

Anschließend wird die Testfallgenerierung beschrieben und erläutert, wie aus den dabei entstandenen Testfällen ausführbare Testfälle erzeugt werden können. In Teilen ist dieses Lösungskonzept in [Schnelte09] veröffentlicht worden sowie in der Europäischen Union und den USA zum Patent angemeldet worden [LTS10, STL10].

4.1 Anwendungsbeispiel

An dieser Stelle wird ein Anwendungsbeispiel eingeführt. Soweit möglich werden die Konzepte anhand dieses Beispiels näher erläutert. Bei diesem Beispiel handelt es sich um ein Anforderungsdokument für ein Türsteuergerät eines PKWs, das in [HP02] veröffentlicht worden ist. Das Türsteuergerät umfasst die folgenden Funktionen: Einstellungen der Sitze, Einstellungen der Außenspiegel, Ansteuerung der Türschlösser sowie der Innenraumbeleuchtung.

Das Beispiel basiert auf realen Anforderungsdokumenten, wie sie bei DaimlerChrysler verwendet werden, und entspricht in Komplexität von Form realen Anforderungsdokumenten aus der Automobilindustrie. Daher eignet sich dieses Dokument gut als praxisnahes Anwendungsbeispiel.

4.1.1 Aufbau des Dokumentes

Das vorliegende Anforderungsdokument gliedert sich in acht Abschnitte. In Abschnitt 1 „Allgemeines“ werden allgemeine Angaben zum Absatzmarkt, Entwicklungsablauf, Anforderungen an die Dokumentation und Ähnlichem gemacht. Abschnitt 2 gibt eine Übersicht über die einzelnen Funktionen des Türsteuergerätes. Abschnitt 3 „Komponentenbeschreibung“ erläutert nähere technische Anforderungen an die Hardwarekomponente, wie z.B. Gehäuseabmessungen. Abschnitt 4 beschreibt die Kommunikation des Türsteuergeräts über den CAN-Bus und Abschnitt 5 beschreibt Einstellmöglichkeiten des Türsteuergeräts. Abschnitt 6 enthält detaillierte Beschreibungen der einzelnen Funktionen des Türsteuergerätes, die in Abschnitt 2 nur umrissen worden sind. Abschnitt 7 beinhaltet Anforderungen an die Komponente aus physikalischer Sicht, wie elektromagnetische Verträglichkeit und Umwelteinflüsse. Das Dokument schließt schließlich in Abschnitt 8 mit einem Glossar.

Für diese Arbeit interessant sind in erster Linie die Abschnitte, die die Funktionalität des Steuergeräts beschreiben. In Teilen sind jedoch auch die Beschreibungen der Schnittstellen und die CAN-Kommunikation von Interesse. Diese Abschnitte werden nun erläutert.

4.1.2 Schnittstellenbeschreibung

In dieser Arbeit liegt der Fokus auf Funktionstests von reaktiven Systemen. In reaktiven Systemen ist die Funktionalität von Systemen darüber definiert, wie das System bei Änderungen an den Eingabesignalen zu reagieren hat; wobei die Reaktionen durch Änderungen an den Ausgabesignalen beschrieben werden.

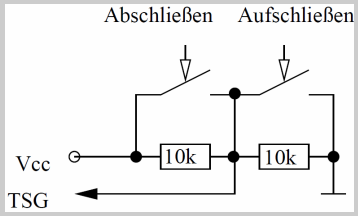
Demnach ist die Erläuterung der Schnittstelle mit der Beschreibung der Ein- und Ausgabesignale und den dazugehörigen Wertebereichen ein notwendiger Teil zum Verständnis der Funktionalität des Systems. Daher wird hier kurz auf die Schnittstellenbeschreibung im Beispieldokument eingegangen. Die Beschreibung liegt in tabellarischer Form vor, der nachfolgend für jedes Signal eine Beschreibung der Kodierung folgt. Nachfolgend die Tabelle in Auszügen:

Anschluß	Bezeichnung	In/Out	Beschreibung
1	MASSE	-	Signalmasse
2	FHB_VL	In	Fensterheber-Taster vorne links (Fahrertür, nicht belegt bei Einbau in Beifahrertür)
6	F_BEWEG	In	Scheibenbewegung Fahrer bzw. Beifahrerfenster
7	MGMT_1	In	Benutzermanagement-Taster Einstellung 1 (nicht belegt in der Beifahrertür)
12	T_OFFEN	In	Fahrer- bzw. Beifahrertür offen
14	T_LICHT	Out	Ausstiegsleuchte in Fahrer- bzw. Beifahrertür
19	KEY_STATE	In	Status Türschloß

Tabelle 1: Tabellarische Schnittstellenbeschreibung

Für die einzelnen Anschlüsse werden anschließend deren Charakteristiken beschrieben, wie im folgenden Beispiel zu sehen ist.

Schloßnußschalter
Anschlüsse: 19 (KEY_STATE)
Charakteristik: Über ein Widerstandsnetzwerk werden die verschiedenen Tasterstellungen codiert:



4.1.3 Funktionalität

Die Funktionen des Türsteuergeräts sind in Abschnitt 2 folgendermaßen zusammengefasst:

- *Sitzeinstellung*
  Verstellen des Winkels der Lehnen der Sitze, der horizontalen Sitzposition, der Höhe des vorderen und hinteren Sitzbereichs und der Schalung des Sitzes.
- *Benutzermanagement*
  Die Einstellung der Sitze können für bis zu vier verschiedene Benutzer gespeichert und abgerufen werden.
- *Türschloss*
  Auf- und Zuschließen des Fahrzeuges über Schlüssel, Funkfernbedienung oder CAN.

- *Fensterheber*
  Heben und Senken der Fensterscheiben des Fahrzeugs.
- *Innenraumbeleuchtung*
  Beleuchtung des Fahrzeuginnen als Hilfe beim Ein- und Aussteigen.
- *Außenspiegeleinstellung*
  Verstellen der Außenspiegel.

Für jede Funktion folgt danach eine informelle Beschreibung der Funktionen aus Benutzersicht. Beispielhaft sei hier die Innenraumbeleuchtung aufgeführt:

> Um dem Benutzer bei Dunkelheit das Ein– und Aussteigen aus dem Fahrzeug zu erleichtern, leuchtet bei geöffneter Tür die Innenraumbeleuchtung an der Fahrzeuginnendecke. Diese Beleuchtung geht aus, wenn die Zündung aktiviert wird oder aber 10 sec. nach dem Schließen aller Türen. Die eigentliche Ansteuerung der Innenraumbeleuchtungslampe im Fahrzeughimmel wird vom Himmel–Steuergerät übernommen. Dieses berücksichtigt auch weitergehende Benutzerwünsche (z.B. Leselampe für Beifahrer).
>
> In den geöffneten Türen leuchtet für die Dauer des Ein– oder Aussteigens, d.h. solange diese Tür geöffnet ist, zudem eine rote Ausstiegsleuchte, um bei Dunkelheit vorbeifahrenden Fahrzeugen zu signalisieren, daß das Fahrzeug über die gewöhnlichen Abmessungen hinausragt.
> Bei eingeschalteter Beleuchtung (Standlicht, Abblendlicht oder Fernlicht) werden die Bedienelemente, die direkt zum TSG gehören, beleuchtet. In Abschnitt 6.7 wird die Innenraumbeleuchtung näher erläutert.

In Abschnitt 6 wird für jede Funktion eine detailliertere Beschreibung aufgeführt. Hierbei werden jeweils die beteiligten Ein- und Ausgänge und eine genaue Verhaltensbeschreibung aufgeführt. Als Beispiel wird wiederum auf die Innenraumbeleuchtung eingegangen:

> **Eingänge**
>
> - Konfiguration des TSG
> CAN.LL, CAN.RL, CONFIG.TSG LEFT
> - Fahrzeugtyp
> CAN.FCODE_T0, CAN.FCODE_T1
> - Zustand Zündung CAN.KL_15
> - Batteriespannung CAN.BATT
> - Zustand der Fahrzeugtüren
> S1.T_OFFEN, S2.FT_OFFEN, CAN.FT_OFFEN, falls Beifahrer–TSG
>
> **Ausgänge**
>
> - Zustand Innenlicht
> CAN.I_LICHT

- Zustand der Fahrzeugtüren
  CAN.DOOR_STATE, falls Beifahrer–TSG
- Zustand der dem TSG zugeordneten Fahrzeugtüren
  CAN.FT_OFFEN, falls Fahrer–TSG

Verhalten

Fahrer–TSG: Das Fahrer–TSG übermittelt laufend den Status der Fahrertür und der Fondtür auf der Fahrerseite über die CAN–Botschaft FT_OFFEN (siehe auch Abschnitt 6.2).

Beifahrer–TSG: Das Beifahrer–TSG empfängt laufend den Türstatus vom Fahrer–TSG, ergänzt diesen um den Status Türen auf der Beifahrerseite und sendet den kombinierten Türstatus über die CAN–Botschaft DOOR_STATE (siehe auch Abschnitt 6.2).

Liegt ein Timeout der Botschaft FT_OFFEN vor, so nimmt das Beifahrer–TSG an, daß die Türen auf der Fahrerseite geschlossen sind.

Innenraumbeleuchtung: Das Beifahrer–TSG ermittelt, ob — gemäß dem Zustand der Fahrzeugtüren — die Innenraumbeleuchtung brennen sollte und übermittelt diesen Zustand laufend über die CAN–Botschaft I_LICHT. Dabei sind nachfolgende Regeln zu beachten.

- Die Innenraumbeleuchtung leuchtet solange eine der Fahrzeugtüren offen ist, aber nur maximal 10min. nachdem eine Tür geöffnet wurde (das Schließen und Öffnen der Tür setzt dieses Zeitintervall zurück)
- Die Innenraumbeleuchtung leuchtet für 30 sec. nachdem alle Türen geschlossen wurden.
- Die Innenraumbeleuchtung endet, sobald die Zündung KL 15 aktiviert wird.

Hinweis: Die eigentliche Ansteuerung der Innenbeleuchtung, die Überwachung der Batteriespannung sowie die Dimmung wird vom Deckensteuergerät übernommen.

Ansteuerung der Ausstiegsleuchten: Für jede Fahrzeugtür überwacht das jeweils zuständige TSG, ob die Tür offen ist und steuert gemäß folgender Regel die jeweilige Ausstiegsleuchte an:
- Die Ausstiegsleuchte einer Tür leuchtet, solange die Tür offen ist.
- Die Ansteuerung der Ausstiegsleuchten endet sobald die Batteriespannung BATT unter den Grenzwert von 10V fällt. Sobald die Batteriespannung wieder über 10.5V ist, wird die Ansteuerung der Ausstiegsleuchten wieder aufgenommen.

4.1.4 Alarmsystem

Nicht alle Konzepte lassen sich mit Hilfe der Funktionen des Anforderungsdokumentes des Türsteuergeräts erläutern. Daher wird das Beispiel hier um einige Funktionen eines Alarm-

systems eines PKWs erweitert. Die Funktionen des Alarmsystems passen sich gut in das Beispiel des Türsteuergerätes ein, da die Schnittmenge der verwendeten Eingabesignale sehr hoch ist.

Die nun folgenden Funktionsbeschreibungen sind sinngemäß aus einem Anforderungsdokument eines Automobilhersteller übernommen. Auf eine wortgenaue Übernahme muss aus rechtlichen Gründen verzichtet werden.

Alarmfunktionen

Der Alarm kann mit Hilfe der Funkfernbedienung gesetzt werden, indem der Knopf der Funkfernbedienung für 2 Sekunden gedrückt wird. Auf Setzen des Alarms folgt eine „pre-arm“ Phase von 6 Sekunden. Während dieser Zeit können alle Türen geöffnet werden, was zu einem Zurücksetzen des Alarms führt, um zu verhindern, dass geöffnete Türen scharf-geschaltet sind.

Nach Ablauf der 6 Sekunden sind alle geschlossenen Türen scharf-geschaltet, sodass ein Öffnen einer scharf-geschalteten Tür zum Auslösen eines akustischen Alarms führt. Für alle Türen, die nach Ablauf der Pre-Arming Phase geöffnet sind, gilt, dass sie scharf-geschaltet werden sobald sie geschlossen werden.

4.2 Erfassen der Anforderungen

Im Grundlagenkapitel ist dargelegt worden, dass uneingeschränkte natürliche Sprache nicht zur eindeutigen Abbildung auf ein formales Modell geeignet ist. Aus diesem Grund werden in dieser Arbeit die Anforderungen mit Hilfe einer kontrollierten natürlichen Sprache erfasst.

Im Rahmen dieser Arbeit ist eine kontrolliert natürliche Sprache (KNS) entstanden, die es ermöglicht funktionale Anforderungen aus dem Bereich der Steuergeräteentwicklung für die Automobilindustrie zu erfassen und eindeutig auf ein formales Modell abzubilden. Die KNS ist das Resultat einer Analyse von verschiedenen Anforderungsdokumenten, die von Zulieferern und Automobilherstellern zur Verfügung gestellt worden sind.

Diese Sprache ist eine eingeschränkte Version der englischen Sprache und besteht aus den Komponenten Wörterbuch, Grammatik inklusive einer Anforderungsschablone sowie einigen Interpretationsregeln. Diese Komponenten werden nun anhand von Beispielen eingeführt, um dem Leser einen Überblick über die KNS zu verschaffen, bevor im Kapitel 4.3 eine formale Definition der Syntax erfolgt.

4.2.1 Wörterbuch

Als Resultat der Analyse der Anforderungsdokumente besteht das Wörterbuch aus Signalen und ihren möglichen Zuständen. Wie auch im Anwendungsbeispiel zu sehen, findet sich solch eine Auflistung aller Signale mit ihren möglichen Zuständen häufig in Anforderungsdokumenten wieder. Dort dient es der Beschreibung der Schnittstelle des zu entwickelnden Systems. Typischerweise enthält diese Auflistung weiterhin die Information, ob es sich einem Signal um ein Ein- oder Ausgabesignal handelt. Auch das Wörterbuch beinhaltet diese Information.

Das Wörterbuch enthält darüber hinaus die Information, aus welchen Zuständen das Signal in einem anderen Zustand wechseln darf. Ein Beispiel hierfür ist das Signal *Klemme 15*, welches den Zustand des Zündschlosses codiert. Die möglichen Zustände des Signals sind: *Schlüssel nicht gesteckt, Schlüssel steckt, Radio Ein, Zündung Ein, Motor starten*. Die einzelnen Zustände können durch Drehen des Schlüssels eingestellt werden, was zur Folge hat, dass bestimmte Werte nur aus bestimmten vorher geltenden Zuständen erreicht werden können. So kann der Zustand *Radio Ein* aus dem Zustand *Schlüssel steckt* (Drehen im Uhrzeigersinn) oder aus dem Zustand *Zündung Ein* (Drehen gegen den Uhrzeigersinn) erreicht werden. Die Auflistung der möglichen Zustände, aus denen in einen gegebenen Zustand gewechselt werden darf, wird im Folgenden als *Ausgangszustände* bezeichnet.

Tabelle 2 zeigt ein mögliches Wörterbuch für das fortlaufende Beispiel. Zu erkennen ist, dass die Bezeichnungen für die Signale eher der natürlich-sprachlichen Beschreibung aus dem Anforderungsdokument entsprechen als der ursprünglichen „technischen“ Bezeichnung. Weiterhin ist zu sehen, dass Ausgangszustände nur für Eingabesignale definiert sind. Dies rührt aus der Tatsache, dass die Information, wann ein Ausgabesignal in einen anderen Wert wechselt, in der Funktionsbeschreibung der Anforderungen niedergeschrieben ist.

In einigen Fällen ist es zur Beschreibung der Anforderungen notwendig, Variablen zu verwenden, die nicht einem Ein- oder Ausgangssignal entsprechen, sondern die einen internen Systemzustand beschreiben. Solche Variablen vom Typ `internal` können ebenfalls im Wörterbuch definiert sein.

Signal	Zustände	Ausgangszustände	Initialzustand	Typ
terminal 15	key out	key inserted	key out	Input
	key inserted	key out, radio on		
	radio on	key inserted, ignition on		
	ignition on	radio on, start		
	start	ignition on		
key state	lock	unlock	unlock	Input
	unlock	lock		
driver door	open	closed	close	Input
	closed	open		
alarm	set		unset	Output
	unset			

Tabelle 2: Wörterbuch

4.2.2 Grammatik

Die Grammatik legt fest, wie mit Hilfe der Einträge aus dem Wörterbuch einfache englische Sätze gebildet werden können, die Systemzustände beschreiben. Da für die Domäne der eingebetteten Systeme Zeitbedingungen eine wichtige Rolle spielen, sind in der Grammatik weiterhin Möglichkeiten vorgesehen, um solche Zeitbedingungen zu formulieren.

Während der Analyse der Anforderungsdokumente haben sich verschiedene typische Satzkonstrukte herauskristallisiert, die benötigt werden, um Systemzustände im Bereich der

eingebetteten Systeme zu beschreiben. Die verwendete Grammatik ist in der Lage diese typischen Konstrukte abzubilden. Eine weitere Basis für den Entwurf der Grammatik sind die in [Denger02, DBK03] vorgestellten Sprachmuster für Anforderungen aus dem Bereich der eingebetteten Systeme.

Beispiele für mögliche Sätze, die in der KNS formuliert werden können sind `key state is locked for at least 2 sec and driver door is open` und `alarm is set for at least 2 sec, but less than 4 sec`. Eine formale Definition der Grammatik erfolgt im Kapitel 4.3.

4.2.2.1 Anforderungsschablone

Mit Hilfe des Wörterbuches und der Grammatik ist der Benutzer in der Lage Systemzustände über gewisse Zeitintervalle zu beschreiben. Um nun auch Zustandsübergänge zwischen diesen Zuständen beschreiben zu können, wird dem Benutzer eine tabellarische Anforderungsschablone zur Verfügung gestellt, die dies leistet. Die Schablone ist aus der Analyse der bestehenden Anforderungsdokumente entstanden, da in einigen dieser Dokumente ähnliche Schablonen verwendet worden sind. Die Schablone ist in der folgenden Tabelle abgebildet:

Bezeichnung der Anforderung	
Condition before the trigger	Systemzustand, der vor dem Auslöser (Trigger) gelten muss.
Trigger	Löst die entsprechende Reaktion aus.
Condition after the trigger	Systemzustand, der nach dem Auslöser gelten muss.
Reaction	Systemzustand, der von der Reaktion hergestellt wird.

Tabelle 3: Schablone für die Anforderungserfassung

Jede Anforderung hat eine Bezeichnung, die vom Benutzer beliebig gewählt werden kann, jedoch eindeutig sein muss. Das heißt, keine zwei Anforderungen dürfen den gleichen Bezeichner haben. Im Laufe der Arbeit wird nur dann eine Bezeichnung für eine Anforderung angegeben, wenn dies für das Verständnis notwendig ist.

Die Semantik der weiteren Einträge der Schablone wird durch die Grafik in Abbildung 16 erläutert. Hier ist zu sehen, dass der Trigger als der Zeitpunkt definiert ist, zu dem ein Zustandswechsel erfolgt; und zwar ein Wechsel von einem Systemzustand, in dem der im Feld `Trigger` beschriebene Zustand nicht gilt, zu einem Systemzustand, in dem der Trigger gilt.

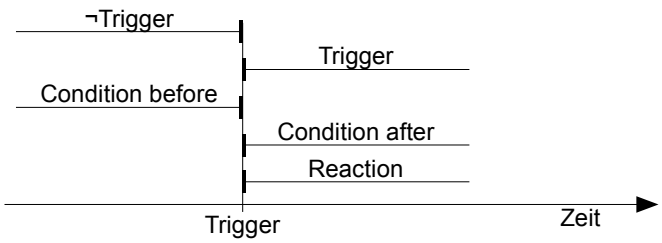


Abbildung 16: Grafische Darstellung der Anforderungsschablone

Zu beachten ist, dass der Trigger dem zufolge ein Ereignis darstellt, obwohl der Eintrag im Feld Trigger einen Systemzustand beschreibt. In manchen Fällen kann diese Darstellung auf dem ersten Blick etwas verwirrend sein; zum Beispiel der Satz `driver door is open` stellt im Englischen kein Ereignis, sondern einen Zustand dar. In den meisten Fällen kann der Systemzustand aber als ein Ereignis im Passiv gelesen werden (z.B. `driver door is closed`).

Die Felder `Condition before the trigger` und `Condition after the trigger` beschreiben die Bedingungen, die vor und nach dem Auslöser gelten müssen, damit das auslösende Ereignis zu der beschriebenen Reaktion führt.

Schließlich wird im Feld `Reaction` der Systemzustand beschrieben, der nach dem Auslöser im System herrschen muss. An dieser Stelle muss hervorgehoben werden, dass nicht verlangt wird, dass die Reaktion vorher nicht galt. Eine alternative Bezeichnung für dieses Feld wäre demnach `Postcondition`. Die Bezeichnung `Reaction` ist ein Zugeständnis an die Begriffswelt, die in den untersuchten Anforderungsdokumenten vorgefunden worden ist.

Als Beispiel für die Benutzung der Anforderungsschablone folgt nun ein Beispiel, dass die Aktivierung des Alarms aus dem Anwendungsbeispiel umsetzt. Zur Erinnerung zuerst der Originaltext und anschließend die daraus resultierende Anforderungsschablone.

> Der Alarm kann mit Hilfe der Funkfernbedienung gesetzt werden, indem der Knopf der Funkfernbedienung für 2 Sekunden gedrückt wird.

Set Alarm	
Condition before the trigger	
Trigger	remote lock button is pressed.
Condition after the trigger	remote lock button is pressed for at least 2 sec.
Reaction	2 sec after the trigger the alarm is set.

Tabelle 4: Anforderungsschablone für das Setzen des Alarms

Diese Anforderungsschablone kann grafisch, wie in Abbildung 17 zu sehen, dargestellt werden.

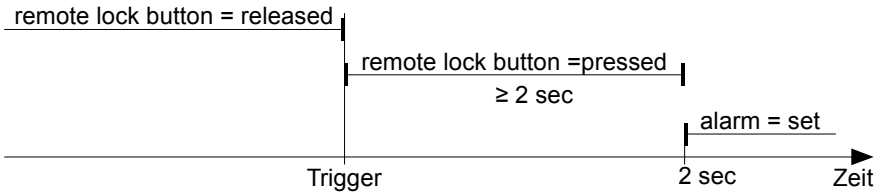


Abbildung 17: Grafische Darstellung des Beispiels aus Tabelle 4

Die bisher an Beispielen eingeführte KNS soll in den nun folgenden Seiten formal definiert werden.

4.3 Syntax der KNS

In diesem Abschnitt wird die Syntax der Anforderungen und somit der kontrollierten natürlichen Sprache beschrieben. Ein Spezifikationsdokument besteht immer aus einem Wörterbuch, in dem die Signale definiert sind und einem Teil, der die Anforderungen beschreibt.

Die Grammatik wird mit Hilfe einer erweiterten Backus-Naur-Form beschrieben, wie sie im Parsergenerator Eli [ELI] verwendet wird. Hierbei drückt ein Schrägstrich '/' Alternativen aus, ein Stern '*' wird verwendet, um auszudrücken, dass ein Ausdruck beliebig häufig (inklusive null Mal) vorkommen darf, ein Plus '+' wird verwendet, um auszudrücken, dass ein Ausdruck beliebig häufig aber mindestens ein Mal vorkommen muss und ein Ausdruck, der in eckige Klammern '[...]' gesetzt ist, ist optional.

4.3.1 Grundsymbole

Die Grundsymbole der kontrolliert-natürlichen Sprache bestehen aus den Einträgen aus dem Wörterbuch und reservierten Wörtern.

Signalnamen und -werte sind Bezeichner. Sie können aus einem oder mehren Teilwörtern bestehen. Ein Wort wird durch den folgenden regulären Ausdruck beschrieben:

```
Word: [a-zA-Z][a-zA-Z0-9_]*
```

Somit beginnt ein Wort stets mit einem Buchstaben, gefolgt von beliebig vielen Buchstaben, Zahlen oder Unterstrichen. Der Unterstrich ist in natürlicher Sprache nicht Teil von Wörtern, wird jedoch gelegentlich in technischen Dokumenten bei der Bezeichnung von Signalnamen verwendet.

Das Literal `Number` erlaubt es, ganzahlige Werte anzugeben:

```
Number: [0-9]+
```

4.3.2 Wörterbuch

Das Wörterbuch beschreibt Signale, wie bereits im vorherigen Abschnitt in tabellarischer Form dargestellt worden ist. Ein Eintrag der Tabelle hat die folgende Form:

Signalname	Zustände	Ausgangszustände	Initialzustand	Typ
`Word+ .`	`Word+ .`	`Word+ (',' Word+)* .`	`Word+ .`	`'Input'/ 'Output'/ 'Internal' .`
	.	.		
	.	.		
	.	.		
	`Word+ .`	`Word+ (',' Word+)* .`		

Das heißt, ein Signal hat einen Namen, einen Wertebereich sowie einen Typ. Der Wertebereich besteht aus einer Aufzählung von textuellen Signalwerten, wobei für jeden dieser Werte eine Aufzählung von Ausgangszuständen definiert ist. Der Typ eines Signals kann aus der Menge {`Input`, `Output`, `Internal`} gewählt werden.

4.3.3 Anforderungen

Die tabellarische Schablone, mit deren Hilfe Anforderungen verfasst werden, ist bereits vorgestellt worden. Nun soll erläutert werden, wie die Sätze gebildet werden können, die in die einzelnen Felder der Schablone eingetragen werden können.

Hierfür werden erst einmal Satzteile beschrieben, die in allen Einträgen verwendet werden können. Allen Sätzen der Sprache ist gemein, dass sie in der Lage sind, den Zustand eines Signals zu beschreiben. Die Grundlage hierfür wird durch eine Produktion geschaffen, die es ermöglicht einem Signal einen Zustand zuzuweisen.

```
SignalState:
      ['the'] SignalName 'is' SignalValue .
```

Da sich diese Arbeit mit Echtzeitsystemen auseinandersetzt, bietet die KNS weiterhin die Möglichkeit diese Zustandsbeschreibung mit zeitlichen Ausdrücken anzureichern. Hierfür enthält die Sprache Konstrukte, die es erlauben Zeitdauer und Zeitpunkte zu beschreiben.

```
TimeExpr: Number TimeUnit
TimeUnit: 'ms' / 'sec' / 'min' .
UpperBound:
  'for less than' TimeExpr /
  'for at most' TimeExpr .
LowerBound:
  'for at least' TimeExpr /
  'for more than' TimeExpr .
Duration:
  'for exactly' TimeExpr /
  UpperBound /
  Lowerbound [',' 'but' UpperBound] .
```

Die obigen Regeln erlauben es Zeitdauern zu beschreiben, die exakt einer Zeitspanne entsprechen (`for exactly`), bzw. Zeitdauern die echt kleiner (`for less than`), kleiner gleich (`for at most`), größer gleich (`for at least`) oder echt größer (`for more than`) als eine angegebene Zeitspanne sind.

Weiterhin ist es möglich, Zeitdauern zu beschreiben, die innerhalb eines angegebenen Intervalls liegen. Hierzu kann eine untere Schranke mit einer obereren Schranke kombiniert werden, wobei die beiden Ausdrücke mittels (`, but`) zusammengehalten werden.

Beispiel: for at least 2 sec, but for at most 4 sec.

Standardmäßig beziehen sich alle Zeitangaben immer relativ auf den Auslöser der beschriebenen Funktion. Um diesen relativen Zeitpunkt variieren zu können, enthält die Sprache Konstrukte zum Beschreiben von Zeitpunkten.

```
UpperTimePoint: 'at the latest after' TimeExpr .
LowerTimePoint: 'at the earliest after' TimeExpr .
SimpleTimePoint: TimeExpr 'after the trigger' .
```

Mithilfe dieser Regeln kann ein Zeitpunkt angegeben werden, der zeitlich hinter dem Auslöser liegt.

4.3.3.1 Bedingung vor dem Auslöser

Das Feld `Condition before the trigger` in der Anforderungsschablone ermöglicht es, Bedingungen zu formulieren, die zeitlich vor dem Auslöser gelten müssen. Diese Bedingungen bestehen aus einzelnen Bedingungen an Signalzustände, die optional mit Zeitdauern versehen werden können. Die einzelnen Bedingungen können mit `and` zu einer komplexeren Bedingung verbunden werden.

```
ConditionBeforeSentence: ComplexConditionBefore '.' .
ComplexConditionBefore:
      SimpleConditionBefore ('and' SimpleConditionBefore)* .
SimpleConditionBefore:
      SignalState [Duration] .
```

Beispiel: engine is running and driver door is open for at least 2 sec.

Das obige Beispiel zeigt, dass diese Verknüpfung einzelner Bedingungen mittels `and` zu mehrdeutigen Aussagen führt, da nicht klar ist, auf welche Signalzustände sich die Zeitbedingung bezieht. Für das obige Beispiel könnten zwei mögliche Interpretationen angegeben werden:

1. [engine is running and driver door is open] for at least 2 sec.
2. engine is running and [driver door is open for at least 2 sec].

Diese Mehrdeutigkeiten werden mit einer Interpretationsregel aufgelöst, die besagt, dass sich eine Zeitbedingung immer nur auf den letzten Signalzustand bezieht. Für das Beispiel bedeutet dies, dass die zweite Interpretation gewählt wird. Um die erste Interpretation zu erlangen, muss der Satz folgendermaßen formuliert werden:

engine is running for at least 2 sec and driver door is open for at least 2 sec.

4.3.3.2 Auslöser

Ein Auslöser stellt eine einfache Bedingung an den Wert eines Signales.

```
TriggerSentence: SignalState '.' .
```

Beispiel: driver door is open.

4.3.3.3 Bedingung nach dem Auslöser

Da die Reaktionen einer zu beschreibenden Funktion nicht zwingend unmittelbar nach dem Auslöser erfolgen müssen, sondern unter Umständen erst nach einer angegebenen Verzögerung erfolgen, kann es notwendig sein, Bedingungen an das Verhalten während dieser Verzögerungszeit zu stellen. Dies kann mithilfe des Feldes `Condition after the trigger` realisiert werden.

Die Grammatik für die Bedingung nach dem Auslöser entspricht weitestgehend der Bedingung vor dem Auslöser. Zusätzlich zu einer Dauer kann solch einer Bedingung ein Zeitpunkt (`SimpleTimePoint`) hinzugefügt werden, auf den sich die Bedingung bezieht. Weiterhin ist es möglich eine Bedingung zu formulieren, die aussagt, dass ein angegebenes Signal im angegebenen Zeitraum nicht in Reaktionen verwendet werden darf.

```
ConditionAfterSentence: ComplexConditionAfter '.' .
ComplexConditionAfter:
      SimpleConditionAfter ('and' SimpleConditionAfter)* .
SimpleConditionAfter:
      [SimpleTimePoint]
      ((SignalState [Duration])
      /(SignalName 'is' 'not used in other reactions' LowerBound)).
```

Ein Beispiel für die Verwendung der Bedingung nach dem Auslöser ist bereits in Tabelle 4 (Seite 55) vorgestellt worden. Ein weiteres Beispiel, das den Nutzen der Angabe eines Zeitpunktes verdeutlicht, ist aus der Spezifikation des KFZ-Alarmsystems übernommen. In diesem Beispiel wird die `Condition after the trigger` in Verbindung mit einem Zeitpunkt benutzt, um eine Bedingung an einen Signalzustand zu stellen, der zum Zeitpunkt der Reaktion gelten muss:

Condition before the trigger	
Trigger	`alarm is set.`
Condition after the trigger	`alarm is set for at least 6 sec and 6 sec after the trigger the driver door is closed.`
Reaction	`6 sec after the trigger the driver door sensor is armed.`

Bedingungen, die die Nutzung eines Signals innerhalb einer anderen Reaktion ausschließen, können verwendet werden, um Prioritäten zwischen einzelnen Anforderungen zu formulieren: Wird die Nutzung eines Signals in einer „Bedingung nach dem Trigger“ einer Anforderung *A* ausgeschlossen, so führt das Auslösen einer anderen Anforderung *B*, die dieses Signal in ihrer Reaktion verwendet, dazu, dass die „Bedingung nach dem Trigger“ von *A* verletzt wird und somit die Reaktion von Anforderung *A* nicht ausgelöst wird. Die Anforderung *B* hat demnach eine höhere Priorität als Anforderung *A*.

Ein Beispiel soll dies verdeutlichen. Ausgangspunkt hierfür ist die folgende Anforderung für die Ansteuerung für das Deckenlicht:

> Die Innenraumbeleuchtung leuchtet solange eine der Fahrzeugtüren offen ist, aber nur maximal 10min. nachdem eine Tür geöffnet wurde (das Schließen und Öffnen der Tür setzt dieses Zeitintervall zurück)

Diese Anforderung beinhaltet zwei Ereignisse: das Einschalten der Beleuchtung beim Öffnen einer Tür und das Ausschalten der Beleuchtung nach 10 Minuten. Die Beleuchtung wird jedoch nur dann ausgeschaltet, falls innerhalb dieser 10 Minuten keine Tür geöffnet worden ist, da diese Aktion den Zeitzähler zurücksetzt. Dieses Verhalten kann für die Fahrertür folgendermaßen notiert werden (Die Anforderungen für die anderen Türen sind analog hierzu aufgebaut):

Condition before the trigger	
Trigger	`driver door is open.`
Condition after the trigger	
Reaction	`interior light is on.`

Condition before the trigger	
Trigger	`driver door is open.`
Condition after the trigger	`interior light is not used in other reactions for at least 10 min.`
Reaction	`10 min after the trigger the interior light is off.`

Durch die „Bedingung nach dem Trigger“, die verlangt, dass keine andere Reaktion das Innenlicht ansteuert, wird sichergestellt, dass das Innenlicht nur dann nach 10 Minuten erlischt, falls keine andere Tür während dieser Zeit geöffnet worden ist, da ein Öffnen einer Tür mit der Ansteuerung des Innenlichts einhergeht, was die „Bedingung nach dem Auslöser“ verletzt.

4.3.3.4 Reaktion

Mit Hilfe des Feldes `Reaction` können die Effekte, der durch die Anforderung beschriebenen Funktion, notiert werden. Wie bei den Bedingungen werden die Reaktionen durch Signalzustände beschrieben. Diese Zustände können um eine Dauer und einen Zeitpunkt ergänzt werden. Die Grammatik der möglichen Sätze zur Beschreibung von Reaktionen wird nun aufgeführt.

```
ReactionSentence: SimpleReaction ('and' SimpleReaction)* '.' .
ReactionTimePoint:
```

```
     SimpleTimePoint /
     LowerTimePoint ',' 'but' UpperTimePoint /
     UpperTimePoint .
SimpleReaction: [ReactionTimePoint] SignalState [ReactionDuration].
ReactionDuration:
    (UpperBound / 'for exactly') 'and afterwards' SignalValueUse /
    LowerBound /
    LowerBound ',' 'but' UpperBound 'and afterwards' SignalValueUse.
```

Aus der Grammatik wird deutlich, dass bei der Angabe eines Zeitpunktes immer eine maximale Verzögerung für das Auftreten einer Reaktion angegeben werden muss. So muss die z. B. die Angabe einer minimalen Verzögerung immer um eine maximale Verzögerungszeit ergänzt werden.

Beispiel: at the earliest after 10 sec, but at the latest after 15 sec the light is on.

Die Notwendigkeit der Angabe einer maximalen Verzögerungszeit rührt aus der Tatsache, dass ein Fehlen dieser Angabe ein nicht testbares Verhalten beschreibt. Hierzu betrachte man die folgende Beschreibung einer Reaktion: `at the earliest after 10 sec the light is on`. Für dieses Verhalten kann kein Test entwickelt werden, da niemals entschieden werden kann, ob lange genug auf das Auftreten der Reaktion gewartet worden ist.

Weiterhin legt die Grammatik fest, dass falls der Signalzustand mit einer Dauer versehen ist und diese Dauer nach oben begrenzt ist, die Information hinzugefügt werden muss, in welchen Zustand das Signal wechselt, nachdem diese Dauer verstrichen ist. Dies wird über das Konstrukt `'and afterwards' SignalValueUse` realisiert.

Beispiel: the light is on for exactly 10 sec and afterwards off.

Diese Angabe wird verlangt, da ohne diese Angabe das Verhalten nicht vollständig spezifiziert ist. Ohne die Angabe fehlt die Information in welchen Zustand das Signal nach Ablauf der Zeitschranke wechseln muss.

4.3.4 Statische Semantik

Die statische Semantik der KNS lässt sich folgendermaßen angeben:

- In Anforderungen verwendete Signalnamen müssen im Wörterbuch definiert sein. Gleiches gilt für die Verwendung von Signalzuständen: Jeder verwendete Signalzustand muss für das Signal im Wörterbuch definiert sein.
- Wird bei der `ReactionDuration` eine `LowerBound` zusammen mit einer `UpperBound` angegeben, so muss die `UpperBound` größer die `LowerBound` sein. Gleiches gilt für `Duration`.
- Der früheste Zeitpunkt, zu dem eine „Bedingung nach dem Trigger“ wahr werden kann, darf nicht später sein, als der früheste Zeitpunkt einer Reaktion. Der Hintergrund hierfür liegt darin begründet, dass solch ein Verhalten nicht implementierbar ist, da zu dem

Zeitpunkt zu dem entschieden werden muss, ob eine Reaktion eintreten soll, nicht sichergestellt werden kann, ob die Bedingung in der Zukunft wahr wird.

4.4 Spracherweiterungen

Um das Formulieren von Anforderungen in der KNS zu vereinfachen und kompaktere Spezifikationen zu ermöglichen, werden im Folgenden Erweiterungen der KNS vorgestellt. Diese Erweiterungen sind so gestaltet, dass sie auf eine oder mehrere Konstrukte aus der bisher vorgestellten KNS (im folgenden Basis-KNS genannt) reduziert werden können. In diesem Abschnitt werden die Erweiterungen motiviert und vorgestellt. Des weiteren wird für jede Erweiterung vorgestellt, wie die Abbildung auf die Basis-KNS vorgenommen wird.

Da im weiteren Verlauf der Arbeit alle weiteren Konzepte anhand der Basis-KNS erläutert werden, können die, zum Teil sehr detaillierten Ausführungen der Abbildungen auf die Basis-KNS vom Leser bei Bedarf übersprungen werden, ohne das das Verständnis der Arbeit im weiteren Verlauf gefährdet ist.

4.4.1 Numerische Signalwerte

Die bisher vorgestellte KNS verwendet nur Signale mit textuell definierten Signalzuständen, wie z.B. `open` oder `closed` für das Signal `driver door`. Dies reicht jedoch nicht immer aus, um Anforderungen zu beschreiben. So enthält zum Beispiel das Anwendungsbeispiel das Signal `battery voltage`, dessen Wertebereich numerisch ist. Verwendet wird dieses Signal dort unter anderem folgendermaßen:

> Die Ansteuerung der Ausstiegsleuchten endet sobald die Batteriespannung BATT unter den Grenzwert von 10V fällt.

Um solche Bedingungen an Signalwerte abbilden zu können, wird die Basis-KNS um Signale mit numerischen Werten erweitert, sodass für das obige Beispiel folgende Spezifikation möglich ist:

Condition before the trigger	
Trigger	`battery voltage < 10000.`
Condition after the trigger	
Reaction	`exist light is off.`

Um dies zu ermöglichen, können im Wörterbuch zusätzlich numerische Signalwerte folgendermaßen tabellarisch definiert werden:

Signalname	Wertebereich	Initialwert	Typ
`Word+ .`	`'[' Number '..' Number ']'.`	`Number`	`'Input'/'Output' /'Internal' .`

Ein numerisches Signal besteht demnach aus einem Signalnamen und einem Wertebereich, der durch die Angabe eines Intervalls beschrieben ist. Als Initialwert kann ein numerischer Wert aus dem Wertebereich gewählt werden und der Typ kann aus der Menge (Input, Output, Internal) gewählt werden. Das Signal `battery voltage` ist zum Beispiel folgendermaßen definiert:

Signalname	Wertebereich	Initialwert	Typ
`battery voltage`	`[0..13000]`	`12500`	`Input`

Um nun einen Zustand eines numerischen Signals beschreiben zu können, wird die Produktion `SignalState` erweitert:

```
SignalState:
      ['the'] SignalName 'is' ( SignalValue)/
      SignalName ('<'|'<='|'='|'>='|'>') MathExpr .
```

Somit ist es möglich, einen Signalzustand an ein numerisches Signal über einen mathematischen Ausdruck zu definieren. Innerhalb des mathematischen Ausdrucks sind die vier Grundrechenarten zugelassen. Als Variablen innerhalb des Ausdrucks können andere Signale mit numerischen Wertebereich verwendet werden.

4.4.1.1 Abbildung auf die Basis-KNS

Um nun diese Erweiterung auf die Basis-KNS abzubilden, wird folgendes Vorgehen gewählt: Für jedes Signal muss vom Testdesigner eine Menge W von Werten angegeben werden, die vom Designer als sinnvoll für die Testerzeugung angesehen werden. Für jeden mathematischen Ausdruck in den Anforderungen werden anschließend die Variablen durch einen konkreten Wert $w \in W$ ersetzt. Falls der Ausdruck mehrere numerische Signale enthält, geschieht dies für alle möglichen Kombinationen der Werte dieser Signale. Anschließend wird der Ausdruck für die gewählte Variablenbelegung zu seinem Ergebnis e ausgewertet. Falls es sich um eine Bedingung handelt, wird ausgewertet, ob die gewählte Variablenbelegung die Bedingung erfüllt. Falls dies nicht der Fall ist, wird für diese Belegung keine Anforderung in der Basis-KNS erstellt. Falls es sich um eine Reaktion handelt, wird der mathematische Ausdruck durch `signal is v_e` ersetzt.

Um die gewählte Variablenbelegung, die zu dem Ergebnis e geführt hat, festzuhalten, wird für jedes numerische Signal s, welches in dem mathematischen Ausdruck benutzt worden ist, die Bedingung `s is v_w` hinzugefügt. Hierbei drückt w den für s gewählten Wert aus.

Ein abstraktes Beispiel erläutert dieses Vorgehen. In diesem Beispiel existieren zwei numerische Signale num_a und num_b. Als mögliche Werte für num_a werden 1 und 3 ausgewählt. Für num_b wird 2 als möglicher Wert festgelegt.

Condition before the trigger	$num_a > num_b$ `for at least 2 sec.`
Trigger	trigger
Condition after the trigger	condafter
Reaction	$num_a = num_b$

Das oben erläuterte Verfahren belegt demnach die Variablen num_a und num_b mit den Werten (num_a=1,num_b=2) und (num_a=3, num_b=2). Für die erste Belegung wird die Bedingung zu falsch ausgewertet. Demnach wird nur für die zweite Belegung eine Anforderung erzeugt. Diese hat die folgende Form:

Condition before the trigger	num_a `is v_3 for at least 2 sec and` num_b `is v_2 for at least 2 sec.`
Trigger	trigger
Condition after the trigger	condafter
Reaction	num_a `is v_2.`

4.4.2 Disjunktionen

Die Basis-KNS erlaubt keine Disjunktionen in den Bedingungen oder im Trigger. Dies hat zur Folge, dass Anforderungen, in denen ein Trigger bei unterschiedlichen Bedingungen die gleiche Reaktion auslöst, in mehrere Anforderungen aufgeteilt werden muss. Gleiches gilt für Anforderungen in denen unterschiedliche Trigger die gleiche Reaktion auslösen können.

Um diese Aufteilung zu verhindern und kompaktere Spezifikationen zu ermöglichen, werden die Bedingungen, wie auch der Trigger um Konstrukte zur Bildung von Disjunktionen erweitert. Zur Erläuterung dient nun eine abgewandelte Anforderung aus dem Anwendungsbeispiel.

> Diese [Innenraum]Beleuchtung geht aus, wenn die Zündung aktiviert wird oder aber […] nach dem Schließen der Fahrertür.

Condition before the trigger	

Trigger	`driver door is closed`
	`or`
	`ignition is on.`

Condition after the trigger	
Reaction	`the interior light is off.`

Um solche Ausdrücke zu ermöglichen, wird die Grammatik folgendermaßen erweitert:

```
ConditionBeforeSentence: DisjunctionConditionBefore '.' .
DisjunctionConditionBefore:
```

```
      ComplexConditionBefore ('or' ComplexConditionBefore)* .
ComplexConditionBefore:
      SimpleConditionBefore ('and' SimpleConditionBefore)* .
SimpleConditionBefore: SignalState [Duration] .
```

Die Bedingung nach dem Auslöser wird analog dazu behandelt. Der Trigger wird folgendermaßen erweitert:

```
TriggerSentence: SignalState ('or' SignalState)* '.' .
```

4.4.2.1 Abbildung auf die Basis-KNS

Diese Erweiterung kann folgendermaßen auf die Basis-KNS abgebildet werden: Da die erweiterte KNS keine Klammerung enthält und das `AND` stärker als das `OR` bindet, liegen die Bedingungen in disjunktiver Normalform (DNF) vor. Diese Disjunktionen werden in die Basis-KNS übersetzt, in dem für jede Klausel aus der DNF eine Anforderung erzeugt wird.

Für das oben aufgeführte Beispiel bedeutet diese, dass die Disjunktionen im Trigger aufgelöst werden, indem für beide Klauseln der ODER-Verknüpfung eine Anforderung erstellt wird.

Condition before the trigger	
Trigger	`driver door is closed.`
Condition after the trigger	
Reaction	`the interior light is off.`

Condition before the trigger	
Trigger	`ignition is on.`
Condition after the trigger	
Reaction	`the interior light is off.`

4.4.3 Konjunktionen im Trigger

In der Basis-KNS besteht ein Trigger immer aus einem einzigen Signalzustand. Konjunktionen im Auslöser sind nicht gestattet. Diese Einschränkung soll an dieser Stelle aufgehoben werden. Konjunktion werden durch die folgende Erweiterung der Syntax für Sätze im Feld Trigger erlaubt:

```
TriggerSentence:
      Signalstate ('and' SignalState)* .
```

Als Beispiel hierfür sei eine Anforderung aus dem Anwendungsbeispiel genannt:

> Diese [Innenraum]Beleuchtung geht aus […] nach dem Schließen aller Türen.

Unter der Annahme, dass in dem verwendeten Beispiel nur zwei Türen – Fahrer- und Beifahrertür – vorhanden sind, lässt sich diese Anforderungen mit Hilfe von Konjunktionen im Trigger folgendermaßen formulieren:

Condition before the trigger	
Trigger	`driver door is closed and passenger door is closed.`
Condition after the trigger	
Reaction	`interior light is off.`

4.4.3.1 Abbildung auf die Basis-KNS

Für die Übersetzung in die Basis-KNS ist die Überlegung hilfreich, dass es zwei mögliche einzelne Auslöser für diese Anforderung gibt: Wenn die Fahrertür bereits geschlossen ist, löst ein Schließen der Beifahrertür die Anforderung aus und bei bereits geschlossener Beifahrertür löst das Schließen der Fahrertür die Anforderung aus.

Eine weitere Möglichkeit wäre, dass die Türen zum *exakt* gleichen Zeitpunkt geschlossen werden. Ein Testfall, der verlangt, dass zwei Aktionen zum exakt gleichen Zeitpunkt ausgeführt werden ist jedoch zum einen nicht sehr realitätsnah und zum anderen setzt er voraus, dass die Testumgebung es ermöglicht, zwei Aktionen zur exakt gleichen Zeit am Testobjekt durchzuführen. Diese Annahme kann jedoch nur in wenigen Fällen gemacht werden.

Des Weiteren ist zu beachten, dass zwei Aktionen, die hintereinander ausgeführt werden durchaus im Testobjekt als gleichzeitig ausgeführte Aktionen interpretiert werden können. Dies ist dann der Fall, falls der zeitliche Abstand zwischen den beiden Aktionen kleiner ist, als die kleinste mögliche Zeiteinheit, die das Testobjekt messen kann. Solche eine kleinste mögliche Zeiteinheit existiert immer, da es sich bei dem Testobjekt um Software handelt, die auf einem diskreten System ausgeführt wird. Daher wird diese dritte Möglichkeit nicht betrachtet und die Originalanforderung wird zu den folgenden beiden Anforderungen übersetzt:

Condition before the trigger	`passenger door is closed.`
Trigger	`driver door is closed.`
Condition after the trigger	
Reaction	`interior light is off.`

Condition before the trigger	`driver door is closed.`
Trigger	`passenger door is closed.`
Condition after the trigger	
Reaction	`the interior light is off.`

Aus diesem Beispiel lässt sich nun das allgemeine Übersetzungsmuster ableiten. Die folgende Anforderung enthält im Trigger eine Konkatenation von *n* Signalzuständen $sigState_1$, …, $sigState_n$.

Condition before the trigger	*condBefore*
Trigger	$sigState_1$ and $sigState_2$ … and $sigState_n$
Condition after the trigger	*condAfter*
Reaction	*reaction*

Diese Anforderung wird in *n* Anforderungen übersetzt, wobei die *m*-te Anforderung des Resultats folgende Form hat:

Condition before the trigger	$sigState_1$ … and $sigState_{m-1}$ and $sigState_{m+1}$ … and $sigState_n$ and *condBefore*
Trigger	$sigState_m$
Condition after the trigger	*condAfter*
Reaction	*reaction*

Das heißt, in der Bedingung vor dem Auslöser finden sich nun alle Teilbedingungen der Originalauslösebedingung bis auf $sigState_m$ wieder. Diese befindet sich jedoch im Trigger, sodass nach dem Trigger die gesamte Originalauslösebedingung gilt.

4.4.4 Reihung von Triggern

Eine weitere sinnvolle Erweiterung der Basis-KNS besteht in der Möglichkeit Anforderungen zu notieren, welche Reaktionen beschreiben, die erst nach einer gewissen Abfolge von Ereignissen eintreten. Ein Beispiel hierfür ist das sogenannte *Double-* oder auch *SafetyLock*[1] Merkmal, das häufig in modernen Fahrzeugen zu finden ist. Um dieses Merkmal zu aktivieren, muss der Verschließen-Knopf der Funkfernbedienung zweimal hintereinander innerhalb einer vorgegebenen Zeitspanne gedrückt werden.

1 *Double-* oder *SafetyLock* ist die Möglichkeit ein Fahrzeug so zu verriegeln, dass das Fahrzeug nicht von innen durch Ziehen an den Türöffnern geöffnet werden kann. Dieses Merkmal dient der Diebstahlsicherung.

Condition before the trigger	
Trigger	`remote lock button is pressed.`
Trigger	`at the latest after 2 sec the remote lock button is pressed.`
Reaction	`double lock is active.`

Für diese Erweiterung muss sowohl die Anforderungsschablone als auch die Grammatik für den Auslöser erweitert werden. Die Anforderungsschablone wird dahingehend erweitert, dass nun mehrere Trigger zusammen mit „Bedingungen vor dem Auslöser“ hintereinander in einer Anforderung notiert werden können. Nach dem letzten Auslöser muss eine Reaktion notiert werden und optional kann dort eine „Bedingung nach dem Auslöser“ angegeben werden. Die Anforderungsschablone hat also die folgende Form:

Condition before the trigger	
Trigger	`SimpleTrigger`
Condition before the trigger	
Trigger	`TimedTrigger`
. . .	
Condition before the trigger	
Trigger	`TimedTrigger`
Reaction	

Der erste Trigger in der Reihung von Trigger wird in der Grammatik durch die Produktion `SimpleTrigger` bestimmt. Hier können keine Zeitbedingungen angegeben werden.

```
SimpleTrigger:
      SignalState '.' .
```

Für die nachfolgenden Trigger, kann ein Zeitpunkt angegeben werden, der sich immer auf den vorhergehenden Trigger bezieht. Die Angabe eines Zeitpunktes ist optional. Falls ein Zeitpunkt angegeben wird, so kann dieser die folgenden Formen haben:

- `at the latest after` *time*
- `at the earliest after` *time*
- `at the earliest after` $time_1$`, but at the latest after` $time_2$
- `at exactly after` *time*

Condition before the trigger	
Trigger	`remote lock button is pressed.`
Trigger	`at the latest after 2 sec the remote lock button is pressed.`
Reaction	`double lock is active.`

```
TriggerTimePoint:
      LowerTimePoint [',' 'but' UpperTimePoint] /
      UpperTimePoint .
TimedTrigger:
      [TriggerTimePoint] SignalState '.' .
```

4.4.4.1 Abbildung auf Basis KNS

Auch in diesem Fall wird die Erweiterung auf die Basis-KNS abgebildet. Hierfür wird für jede Anforderung, die eine Reihung von Trigger enthält, eine neues internes Signal *part_reqName* eingeführt, wobei *reqName* dem Namen der Anforderung entspricht.

Die Funktion des internen Signals besteht darin, sicherzustellen, dass das Auslösen eines Triggers in der Reihung nur dann zu einer Reaktion führt, wenn alle anderen Trigger vorher bereits ausgelöst worden sind. Dies wird dadurch realisiert, indem das Signal festhält, welcher der verschiedenen Trigger bereits ausgelöst worden ist. Ist noch keiner der aufgelisteten Trigger ausgelöst worden, so ist der Wert des Signals *p0*. Nach dem Auslösen des ersten Triggers wird der Wert des Signals auf *p1* gesetzt, etc. Nach dem Auslösen des letzten Triggers, wird das Signal wieder auf *p0* zurückgesetzt, sodass die Reihung erneut durchlaufen werden kann.

Jede Anforderung, die eine Reihung von *n* Triggern enthält, wird in *n* verschiedene Anforderungen in der Basis-KNS übersetzt. Hierbei wird die Übersetzung dahingehend ausgeführt, dass die *n*-te Anforderung in der Reaktion sicherstellt, dass das Signal *part_reqName* auf den Wert p_n gesetzt wird. In der Bedingung vor dem Auslöser der *n*-ten Anforderung wird außerdem verlangt, dass dieses Signal den Wert p_{n-1} besitzt. Auf diese Weise kann die *n*-te Anforderung erst nach dem Ausführen der (*n*-1)-ten Anforderung ausgeführt werden.

Je nachdem welcher Zeitpunkt bei einem Trigger innerhalb einer Reihung angegeben ist, wird ein anderes Übersetzungsmuster gewählt, um diese Zeitbedingung zu kodieren. Für den Zeitpunkt „at the latest after“ wird nun das Übersetzungsmuster aufgeführt. Die anderen Zeitpunkte folgen einem ähnlichen Übersetzungsmuster, werden an dieser Stelle jedoch nicht aufgeführt. Der interessierte Leser findet sie im Anhang dieser Arbeit.

At the latest after

Typischerweise wird bei der Reihung von Triggern eine obere Zeitschranke angegeben. Diese Zeitschranke drückt aus, dass der zweite Auslöser innerhalb einer gewissen Zeit erfolgen muss, um die gewünschte Reaktion auszulösen. Das Double-Lock Beispiel mit Angabe einer oberen Zeitschranke kann folgendermaßen notiert werden:

Req1	
Condition before the trigger	
Trigger	`remote lock button is pressed.`
Trigger	`at the latest after 2 sec the remote lock button is pressed.`
Reaction	`double lock is active.`

Der vorgestellten Idee folgend, wird die Anforderungen in die folgenden zwei Anforderungen unterteilt.

Condition before the trigger	`part_Req1 is p0.`
Trigger	`remote lock button is pressed.`
Condition after the trigger	
Reaction	`part_Req1 is p1 for 2 sec and afterwards p0.`

Condition before the trigger	`part_Req1 is p1.`
Trigger	`remote lock button is pressed.`
Condition after the trigger	
Reaction	`double lock is active and partReq1 is p0.`

Diese Übersetzung führt dazu, dass das erstmalige Drücken des Knopfes der Funkfernbedienung, die erste Anforderung auslöst, da im Initialzustand der Wert des internen Signals part_Req1 auf p0 gesetzt ist. Als Reaktion setzt diese Anforderung den Wert auf p1, sodass ein weiteres Drücken des Knopfes die zweite Anforderung auslöst. Um jedoch auszudrücken, dass die zweite Anforderung nur dann ausgelöst wird, wenn der zweite Trigger innerhalb von 2 sec auftritt, wird das Signal *part_Req1* nur für zwei Sekunden auf den Wert *p1* gesetzt. Nach Ablauf dieser zwei Sekunden wird das Signal wieder auf *p0* zurückgesetzt.

Dieses Vorgehen soll nun allgemein als Übersetzungsmuster angegeben werden. Zu sehen ist in der folgenden Anforderungsschablone ein Ausschnitt einer Reihung von n Triggern, bei der Auslöser $trigger_m$ dem Auslöser $trigger_{m-1}$ folgt, wobei der zweite Trigger die Zeitbedingung `at the lastest after` *time* besitzt.

Condition before the trigger	$condBefore_{m-1}$
Trigger	$trigger_{m-1}$
Condition before the trigger	$condBefore_m$
Trigger	`at the latest after` *time* $trigger_m$

Solch eine Reihung von Auslösern wird übersetzt zu:

Condition before the trigger	$condBefore_{m-1}$ `and` *part_reqName* `is` p_{m-1}.
Trigger	$trigger_{m-1}$
Reaction	*part_reqName* `is` p_m `for` *time* `and afterwards p0`.

Condition before the trigger	$condBefore_m$ `and` part_reqName `is` p_m.
Trigger	$trigger_m$
Reaction	*nextPart*.

Die Reaktion der zweiten Anforderung enthält den Ausdruck *nextPart*. Dieser Ausdruck wird zu unterschiedlichen Konstrukten übersetzt, wobei hier zwei Fälle unterschieden werden:

1. Für den Fall, dass $trigger_m$ der letzte Trigger in der Reihung ist, wird *nextPart* zu $reaction_m$ `and part_reqName is p0` übersetzt. Das heißt, die Reaktion wird ausgelöst und *part_ReqName* auf `p0` gesetzt, sodass die Reihung erneut durchlaufen werden kann.
2. Falls ein weiterer Trigger in der Reihung folgt, werden die Übersetzungsregeln für den Zeitpunkt angewandt, den der nachfolgende Trigger $trigger_{m+1}$ besitzt.

4.4.5 Beschreibung gleichartiger Anforderungen

In den untersuchten Anforderungsdokumenten werden häufig gleichartige Funktionen beschrieben, die sich nur in den verwendeten Signalen unterscheiden. Im Falle des Anwendungsbeispiels ist dies unter anderen bei der Beschreibung der Lichtansteuerung in den Türen des Fahrzeuges zu finden:

> In den geöffneten Türen leuchtet für die Dauer des Ein– oder Aussteigens, d.h. solange diese Tür geöffnet ist, zudem eine rote Ausstiegsleuchte

Um dieses Verhalten zu beschreiben, muss es bei Verwendung der bisher definierten KNS für jede Tür einzeln beschrieben werden. Dies ist mühsam und erhöht die Wahrscheinlichkeiten für Inkonsistenten, da mehrere Kopien der gleichen Funktion manuell auf demselben Stand gehalten werden müssen.

Eine Möglichkeit für eine kompaktere Beschreibung ist durch die Verwendung von Quantoren gegeben, wie sie in häufig in formalen Sprachen zu finden sind. So lässt sich das oben beschriebene Beispiel in Prädikatenlogik mit Hilfe des Allquantors folgendermaßen darstellen:

$$\forall\, x\big(\big(istTür(x) \wedge offen(x)\big) \rightarrow AusstiegsLeuchteAn(x)\big).$$

Eine Erweiterung der KNS durch Konzepte, die denen von Quantoren ähneln, wird in dieser Arbeit nicht verfolgt. Der Grund hierfür ist darin zu sehen, dass solch eine Erweiterung zu einer komplexeren und damit schwerer zu beherrschenden Sprache führt.

Stattdessen wird in dieser Arbeit ein Ansatz verfolgt, der einer Idee folgt, die aus der Analyse von bestehenden Anforderungsdokumenten erwachsen ist. In Anforderungsdokumenten werden gleichartige Funktionen häufig einmal exemplarisch beschrieben und anschließen wird ergänzt, dass die Funktionalität mehrmals im System vorhanden ist. Für das obige Beispiel bedeutet dies, dass das Verhalten einmal für die Fahrertür beschrieben wird; es aber analog für die Beifahrertür gilt.

Diese Idee wird über Werkzeugunterstützung umgesetzt, indem sogenannte v*erlinkte Anforderungen* eingeführt werden. Eine verlinkte Anforderung besteht aus einem Verweis auf eine normale Anforderung *V* zusammen mit einer Menge von *Ersetzungsregeln*. Eine Ersetzungsregel ist ein Paar ($o \rightarrow r$), wobei sowohl *o* als auch *r* Zeichenketten darstellen. Die verlinkte Anforderung entspricht der Anforderung *V*, mit dem Unterschied, dass alle Vorkommen von *o* in den Sätzen der Anforderung durch *r* ersetzt werden.

Für das angeführte Beispiel der Ausstiegsbeleuchtung wird demnach das Verhalten einmal exemplarisch für die Fahrertür beschrieben

Trigger	`driver door is open.`
Reaction	`driver door exit light is on.`

Trigger	`driver door is closed.`
Reaction	`driver door exit light is off.`

Das Verhalten für die Beifahrertür wird nun beschrieben, indem für beide Anforderungen eine verlinkte Anforderung erstellt wird, die die Ersetzungsregel (*driver door* → *passenger door*) enthält. Das Ergebnis sind die folgenden beiden Anforderungen, die jedoch keine Kopie darstellen; werden die Anforderungen für die Fahrertür verändert, so ändern sich automatisch die verlinkten Anforderungen.

Trigger	`passenger door is open.`
Reaction	`passenger door exit light is on.`

Trigger	`passenger door is closed.`
Reaction	`passenger door exit light is off.`

Dieses Konzept hat den Vorteil, dass gleichartiges Verhalten nur einmal zentral an einer Stelle beschrieben wird, was Inkonsistenten vorbeugt. Darüber hinaus hat dieses Vorgehen auch Vorteile bei der Testfallgenerierung, was im weiteren Verlauf der Arbeit zur Sprache kommen wird.

4.5 Formales Modell

Die in einer kontrollierten natürlichen Sprache verfassten Anforderungen müssen in einem weiteren Schritt auf ein formales Modell abgebildet werden, das zur automatischen Testfallgenerierung genutzt werden kann. In dieser Arbeit wird hierfür ein Aktionenmodell aus der Planung verwendet.

Diese Entscheidung beruht auf folgender Überlegung: Ein Tester kann verschiedene *Aktionen* am Testobjekt ausführen, um das Verhalten zu überprüfen. Im fortlaufenden Beispiel sind diese Aktionen zum Beispiel das Öffnen und Schließen von Türen oder das Betätigen von Knöpfen im Cockpit. Das Ausführen einer solchen Aktion führt zu unterschiedlichen Reaktionen des Testobjekts, wobei die Reaktionen davon abhängen, in welchem Systemzustand sich das Testobjekt zum Zeitpunkt der Ausführung der Aktion befindet.

Welche Reaktionen unter welchen Bedingungen ausgelöst werden, ist im Anforderungsdokument beschrieben. Als Formalisierung für diese bedingten Reaktionen bieten sich daher bedingte Effekte, wie sie aus der automatischen Handlungsplanung bekannt sind, an.

Diese Idee soll nun anhand eines Beispieles verdeutlicht werden. Hierbei wird noch eine informelle Notation für die Aktionen verwendet, die dann im weiteren Verlauf weiter formalisiert wird. Als Beispiel dient hier die Alarmfunktion des Anwendungsbeispiels, deren Beschreibung noch einmal wiederholt wird.

> Auf Setzen des Alarms folgt eine „pre-arm“ Phase von 6 Sekunden. Während dieser Zeit können alle Türen geöffnet werden, was zu einem Zurücksetzen des Alarms führt, um zu verhindern, dass geöffnete Türen scharf-geschaltet sind. Nach Ablauf der 6 Sekunden sind alle geschlossenen Türen scharf-geschaltet, sodass ein Öffnen einer scharf-geschalteten Tür zum Auslösen eines akustischen Alarms führt.

Diese Anforderungen lassen sich mit Hilfe der Anforderungsschablone folgendermaßen ausdrücken:

Condition before the trigger	`alarm is set for less than 6 sec.`
Trigger	`driver door is open.`
Condition after the trigger	
Reaction	`alarm is unset.`

Tabelle 5: Zurücksetzen des Alarms durch Öffnen der Fahrertür (Anf_1)

Condition before the trigger	`alarm is set for at least 6 sec.`
Trigger	`driver door is open`
Condition after the trigger	
Reaction	`alarm sounder is active.`

Tabelle 6: Auslösen des akustischen Alarms durch Öffnen der Fahrertür (Anf_2)

Diese beiden Anforderungen beschreiben, welche Reaktionen erwartet werden, wenn die Fahrertür geöffnet wird. Welche Reaktionen dies sind, hängt davon ab, in welchem Zustand sich das System befindet, wenn die Aktion „öffne Fahrertür“ ausgeführt wird: Wird die Aktion zu einem Zeitpunkt ausgeführt, zu dem der Alarm für weniger als 6 Sekunden gesetzt ist, so wird der Alarm zurückgesetzt. Wird die Aktion hingegen zu einem Zeitpunkt ausgeführt, zu dem der Alarm für mindestens 6 Sekunden gesetzt ist, so wird der Alarm nicht zurückgesetzt, sondern ein akustisches Signal ertönt. Die Anforderungen beschreiben demnach bedingte Effekte, der Aktion „öffne Fahrertür“. Informell kann diese Aktion folgendermaßen beschrieben werden:

```
open driver door
      precond: driver door=closed
      effect:  driver door=open
               ∧ when (alarm=set for less than 6 sec)
                  → alarm=unset
               ∧ when (alarm=set for at least 6 sec)
                  → sounder=active
```

Die Frage, die an dieser Stelle im Raum steht, ist nun, welche Repräsentation gewählt werden soll, um die Aktionen und insbesondere die komplexen Zeitbedingungen darstellen zu können. Im Grundlagenkapitel sind verschiedene Repräsentationen für Planungsprobleme vorgestellt worden. Diese werden nun auf ihre Eignung für diese Arbeit untersucht.

4.5.1 Repräsentation durch PDDL

Die Repräsentation durch PDDL ist eine sehr wünschenswerte Wahl, da PDDL einen Quasi-Standard im Bereich der KI-Planung darstellt. Eine Darstellung durch PDDL hätte demnach den Vorteil, dass auf viele bereits existierende Planer zurückgegriffen werden kann, die PDDL verarbeiten. Weiterhin wäre es damit leicht möglich von Fortschritten im Bereich der

Planungsalgorithmen zu profitieren, da zukünftige Planer mit hoher Wahrscheinlichkeit ebenfalls PDDL unterstützen werden.

Leider ist die Darstellung von Zeit in PDDL sehr eingeschränkt: Aktionen haben eine festgelegte Zeitdauer und für jede Vorbedingung muss angegeben werden, ob diese zu Beginn, zum Ende oder über die gesamte Dauer der Aktion gelten muss. Für Effekte gilt Ähnliches jedoch kann hier lediglich der Beginn und das Ende der Aktion als möglicher Zeitpunkt für das Eintreten eines Effektes angegeben werden kann.

Für die Darstellung der hier benötigten Zeitbedingungen ist diese eingeschränkte Repräsentation nicht geeignet. Hierzu betrachte man die Bedingung vor dem Auslöser aus dem vorausgegangenen Beispiel: „driver door is open for at least 6 sec“. Um diese Aussage zu formalisieren, sind Beschreibungsmöglichkeiten nötig, die Aussagen über den Zustand eines Signals/einer Variable machen können, die für ein gewisses Zeitintervall gelten, das für dieses Beispiel vor dem Zeitpunkt der Ausführung der Aktion liegt. Dies ist in PDDL nicht ohne Weiteres möglich.

Im Grundlagenkapitel sind jedoch zwei weitere Repräsentationsformen vorgestellt worden, die dies leisten: TQEs und Chronicles.

4.5.2 Repräsentation durch TQEs und Chronicles

Der Repräsentation durch TQEs als auch durch Chronicles ist gemeinsam, dass Aussagen über Wahrheitswerte über ein Zeitintervall gemacht werden, was im Rahmen dieser Arbeit eine wünschenswerte Eigenschaft ist. Bei TQEs werden die Aussagen mit Hilfe von Prädikaten ausgedrückt, während bei Chronicles Zustandsvariablen benutzt werden.

Im Hinblick auf diese Arbeit ist die Verwendung von Zustandsvariablen sinnvoll, da hier der Zustand von Signalen beschrieben werden soll, was dadurch geschehen kann, dass jedes Signal durch eine Zustandsvariable repräsentiert wird. Aufgrund dieser Tatsache bietet sich die Verwendung von Chronicles als Repräsentation an.

Chronicles, wie im Grundlagenkapitel vorgestellt, verwenden jedoch keine Aktionen mit Vorbedingungen und Effekten, sondern unterscheiden zwischen fortdauernden Bedingungen und Ereignissen. Schaut man sich die Anforderungsschablone der KNS an, so liegt diese mit den Bedingungen und Reaktionen jedoch näher an der klassischen Repräsentation mit Vorbedingungen und Effekten.

Aufgrund dieser Situation werden in dieser Arbeit TQEs verwendet, die jedoch anstelle von Prädikaten Zustandsvariablen verwenden. Somit kann die Repräsentation mit Vorbedingungen und Effekten beibehalten werden und der Zustand eines Signals kann durch eine Variable beschrieben werden. Die Tatsache, dass die Bedingung „driver door is open for at least 6 sec“ zu einem Zeitpunkt t_{start} gilt, kann nun folgendermaßen beschrieben werden:

$$(driver\ door = open)@[t_1, t_{start}] \wedge t_{start} - t_1 \geq 6 \tag{4.1}$$

wobei hier der Zustand der Fahrertür durch die Variable *driver door* mit dem Wertebereich {*open*, *closed*} abgebildet wird. Zu beachten ist, dass das Intervall, welches den Zeitraum beschreibt, ein geschlossenes Intervall ist, während die TQEs aus dem Grundlagenkapitel halb-offene Intervalle verwenden. Hierauf wird im weiteren Verlauf der Arbeit noch eingegangen.

Die zweite Bedingung mit Zeitangaben aus dem vorausgegangenen Beispiel lautet „driver door is open for less than 6 sec“. Auch diese Aussage kann mit Hilfe von TQEs beschrieben werden, wobei hierfür zwei TQEs benötigt werden:

$$(driver\,door = open)@[t_{1,}\, t_{start}] \wedge (driver\,door = closed)@[t_{2,} t_3] \wedge t_{start} - t_3 < 6 \wedge t_3 < t_1 \quad (4.2)$$

Hier beschreibt die erste TQE, dass die Tür zum Zeitpunkt t_{start} geöffnet ist, während die zweite sicherstellt, dass die Tür spätestens 6 Sekunden vor t_{start} noch geschlossen war. Zusammen genommen stellen die TQEs sicher, dass die Tür zum Zeitpunkt t_{start} für weniger als 6 Sekunden geöffnet ist.

Die Arbeit [GNT04] aus der die Darstellung durch TQEs übernommen worden ist, macht keine Aussagen über die Negation von TQEs. Dass die Negation jedoch eine Herausforderung bei der Formalisierung darstellt, lässt sich an den gerade angeführten Beispielen erläutern.

4.5.2.1 Negation

Für die Negation von Aussagen über Zeitintervalle gibt es zwei verschiedene Interpretationen [AF94]. In der *schwachen Interpretation* gilt die Negation $\neg(x@[t_s, t_e])$ genau dann wenn $\exists t : (t_s \le t \le t_e) \wedge \neg x(t)$, das heißt die Negation gilt, wenn irgendwo im Intervall $[t_s, t_e]$ $\neg x$ gilt.

Eine weitere Interpretation der Negation ist die *starke Interpretation*, für die $\neg(x@[t_s, t_e])$ genau dann gilt, wenn $\forall\, t : (t_s \le t \le t_e) \wedge \neg x(t)$, das heißt die Negation gilt, wenn über das gesamte Intervall $[t_s, t_e]$ $\neg x$ gilt.

Für diese Arbeit sind beide Interpretationen nicht geeignet, was am Beispiel der Aussage „driver door is open for at least 6 sec“ dargelegt werden kann, die positiv wie in Formel 4.1 zu sehen ausgedrückt wird. Die erwartete Negation dieser Aussage lässt sich natürlichsprachlich ausdrücken durch „driver door is open for less than 6 sec or driver door is closed“. Diese Bedeutung wird weder von der schwachen noch von der starken Negation erfasst.

Die schwache Interpretation der Negation führt für dieses Beispiel zu

$$\exists t : (t_1 \le t \le t_{start}) \wedge \neg(driver\,door = open). \quad (4.3)$$

Aufgrund der Tatsache, dass in der positiven Darstellung $t_{start} - t_1 \ge 6$ definiert worden ist, kann folgende Belegung für t_1, t und t_{start} gewählt werden, für die Aussage 4.3 wahr wird: $t_1 = t_{start} - 7, t = t_1$. Diese Belegung entspricht der grafischen Darstellung in Abbildung 18 . Diese Abbildung macht deutlich, dass diese Interpretation nicht sicherstellt, dass die Originalaussage negiert wird, da für das Intervall $[t_{1,} t_{start}]$ keine Aussage über den Zustand des Signals `driver door` gemacht wird.

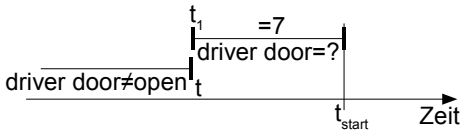


Abbildung 18: Schwache Interpretation der Negation

Gleiches gilt für die starke Negation. Die Bedeutung der starken Negation lässt sich für dieses Beispiel mit „driver door is closed for at least 6 sec“ natürlich-sprachlich darstellen, was nicht der erwartenden Negation entspricht.

Ein weiteres Problem wird bei der Aussage „driver door is closed for less than 6 sec“ deutlich. Wie in der Formalisierung 4.2 zu sehen, wird diese Aussage durch eine Verknüpfung von zwei TQEs ausgedrückt. Eine Negation dieser Aussage muss sich demnach auch auf diese Verknüpfung beziehen. Eine Negation von jeweils nur einer der beiden TQEs würde der erwarteten Negation nicht gerecht werden.

Um diese Problematik anzugehen, wird für die Negation ein Ansatz gewählt, bei dem die möglichen Negationen einer Aussage in der Formalisierung explizit aufgezählt werden müssen. Hierfür wird nun der Begriff *Complex temporally qualified expression* eingeführt:

Definition 4.1. *Eine* Complex temporally qualified expression (CTQE) *hat die Form*

$$(\text{pos}(\text{tqe}_1, \ldots, \text{tqe}_{np};\text{Constraints}), \text{neg}_1(\text{tqe}_1, \ldots, \text{tqe}_{n1};\text{Constraints}),..., \text{neg}_n((\text{tqe}_1, \ldots, \text{tqe}_{nn};\text{Constraints}))$$

wobei pos *eine Menge von TQEs zusammen mit Nebenbedingungen an die in den TQEs verwendeten temporalen Variablen beinhaltet.Die Elemente* $\text{neg}_1, \ldots, \text{neg}_n$ *haben die gleiche Form wie* pos*, wobei jedes Element eine Möglichkeit der Negation von* pos *darstellt.*

Beispiel. Der Ausdruck „driver door is open for at least 6 sec“ kann durch eine CTQE folgendermaßen dargestellt werden:

$$(\text{pos}((driver\ door = open)@[t_1, t_{start}]; t_{start} - t_1 \geq 6), \text{neg}_1((driver\ door \neq open)@[t_1, t_{start}]),$$
$$\text{neg}_2((driver\ door = open)@[t_1, t_{start}] \wedge (driver\ door = closed)@[t_2, t_3]; t_{start} - t_3 \leq 6 \wedge t_3 < t_1).$$

In diesem Beispiel drückt die erste Negation neg_1 aus, dass die Fahrertür nicht geöffnet ist, während die zweite Negation neg_2 ausdrückt, dass die Fahrertür für weniger als 6 Sekunden geöffnet ist.

4.5.2.2 Ausschluss von Effekten

Zur Modellierung von Prioritäten zwischen einzelnen Aktionen ist es sinnvoll, in der Lage zu sein, das Auftreten von Effekten für ein gegebenes Zeitintervall auszuschließen. In der KNS ist dies über Bedingungen der Form `signal is not used in other reactions` möglich.

In der Formalisierung wird die Tatsache, dass ein Effekt, der ein Signal *s* auf einen beliebigen Wert setzt, für ein Zeitintervall $[t_1, t_2]$ ausgeschlossen wird, durch

$s = not\ used\ \text{in}\ other\ reactions@[t_1, t_2]$ ausgedrückt. Die Negation hiervor wird als $s = used\ \text{in}\ other\ reactions@[t_1, t_2]$ formalisiert. Diese Formalisierung drückt aus, dass im Intervall $[t_1, t_2]$ mindestens ein Effekt auftreten muss, der das Signal *s* auf einen beliebigen Wert setzt.

4.5.2.3 Aktionen

Als formales Modell für die in der KNS verfassten Anforderungen werden Aktionen verwendet, die durch Tupel der Form $\langle aktion-name, bedingungen, effekte, bedingte-effekt, t_{start}\rangle$ dargestellt werden, wobei die Bedingungen und Effekte Auflistungen von CTQEs sind und mit t_{start} der Startzeitpunkt der Aktion bezeichnet wird. Die bedingten Effekte haben die Form when(Bedingungen) → (Effekte), wobei Bedingungen auch hier Auflistungen von CTQEs sind, während die Effekte eine Auflistung von CTQEs und bedingten Effekten beinhalten. Auf diese Weise sind geschachtelte bedingte Effekte möglich.

Nachdem das formale Modell eingeführt worden ist, kann nun das in den Tabellen 5 und 6 dargestellte Beispiel (Seite 74), mit Hilfe dieses Modells formuliert werden. Der erste bedingte Effekt formuliert hierbei die Anforderung aus Tabelle 5 (Anforderung 1), während der zweite bedingte Effekte die Anforderung aus Tabelle 6 abbildet (Anforderung 2). Um eine bessere Lesbarkeit zu gewährleisten, sind die Negationen hier nicht aufgeführt. Abbildung 19 zeigt eine grafische Darstellung der Aktion.

```
open driver door
 condition:((driver door=closed)@[t1,tstart])
 effect: ((driver door=open)@[tstart,t2])
 conditional effects:
  when((alarm≠set)@[tc1,tc2]∧(alarm=set)@[tc3,tstart];tc2≤tc3,tstart-tc2<6)
       →((alarm=unset)@[tstart, tc5])
  when((alarm=set)@[tc6, tstart]; tstart-tc6≥6)
       →((sounder=active@[tstart,tc8]:active)
```

Bei näherer Betrachtung der Definition der Aktion `open driver door` fällt auf, dass sich die Bedingung $(\text{driver door=closed})@[t_1, t_{start}]$ und der Effekt $(\text{driver door=open})@[t_{start}, t_2]$ widersprechen, da sich ihre Intervalle bei widersprüchlicher Aussage im Punkt t_{start} überlappen. Somit ist eine legale Ausführung der Aktion eigentlich nicht möglich, da die Bedingung immer durch den Effekt verletzt wird.

In [GNT04] wird dieses Problem umgangen, indem die Intervalle der TQEs als rechtsoffene Intervalle definiert sind. Diese Definition löst zwar das Problem, dass sich Bedingungen und Effekte im Zeitpunkt t_{start} widersprechen; bei genauerer Betrachtung führt dies jedoch zu einem weiteren Problem, was anhand eines Beispieles dargelegt wird.

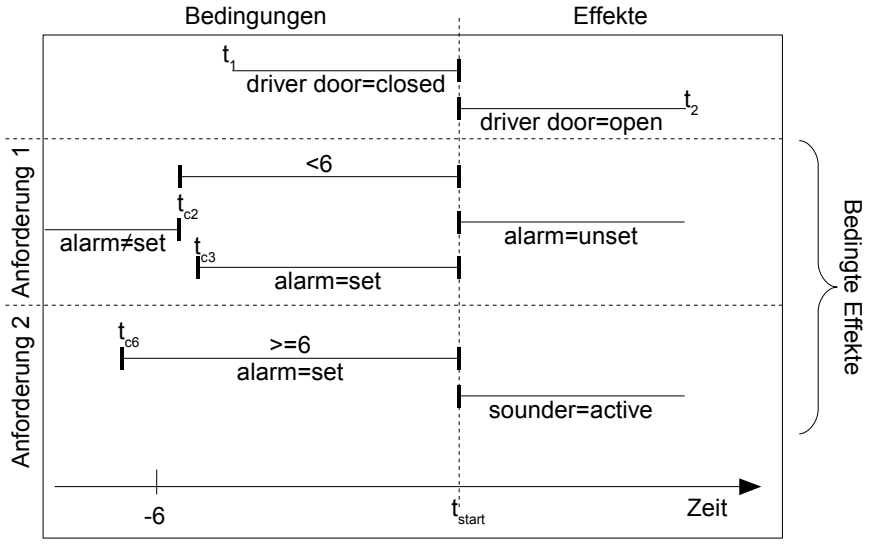


Abbildung 19: Grafische Darstellung der Aktion "open driver door"

Das Beispiel benutzt zwei Aktionen, die die Position des Zündschlüssels eines Autos ändern. Beide Aktionen haben als Vorbedingung, dass sich der Schlüssel in Position 1 befindet. Die Aktion `turn left` dreht den Schlüssel links herum in Position 0, während die Aktion `turn right` den Schlüssel rechts herum in Position 2 bringt.

Werden rechts-offene Intervalle in den TQEs verwendet, so kann sich ein gültiger Plan ergeben, der zur Folge hat, dass sich der Schlüssel in zwei Positionen gleichzeitig befindet. Abbildung 20 zeigt dieses Problem.

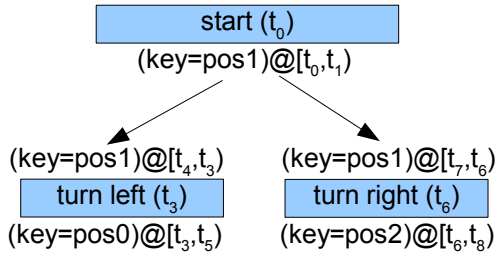


Abbildung 20: Aktionen mit rechts-offenen Intervallen

Die Abbildung zeigt einen Plan mit drei Aktionen *start*, *turn left* und *turn right*, mit den Startzeitpunkten t_0, t_3 und t_6. Die Aktionen sind jeweils durch ausgefüllte Rechtecke dargestellt, wobei die TQEs über den Rechtecken Bedingungen und die TQEs unter den Rechtecken Effekte darstellen.

Setzt man nun die Startzeitpunkte der beiden Aktionen, die den Schlüssel drehen, gleich (t_3=t_6), so erhält man einen gültigen Plan, da die Bedingungen beider Aktionen erfüllt sind und sie somit ausgeführt werden dürfen. Als Ergebnis dieser Ausführung befindet sich der Schlüssel jedoch gleichzeitig in zwei unterschiedlichen Positionen.

Aufgrund dieser Problematik werden TQEs daher (wie in der Aktion *open driver door* zu sehen) durch geschlossene Intervalle definiert, da damit *atomare Zustandsübergänge* aus-

gedrückt werden können. In dem Beispiel der Aktion `open driver door` wechselt der Zustand der Tür zum Zeitpunkt t_{start} von *geschlossen* zu *offen*.

Um das bei dieser Definition bestehende Problem der sich widersprechenden Bedingungen und Effekte zu umgehen, wird festgelegt, dass sich Bedingungen und Effekte, die der *gleichen Aktion* angehören, in ihren *Endpunkten* widersprechen dürfen.

4.6 Übersetzungsvorgang

Auf den vorherigen Seiten ist das formale Modell und anhand von Beispielen der Zusammenhang zu den Anforderungen in der KNS beschrieben worden. Auf den folgenden Seiten soll nun das Schema beschrieben werden, nach dem die Anforderungen automatisch in die Aktionenrepräsentation übersetzt werden. Die Übersetzung ist mit Hilfe des Werkzeugs Eli [WKS07, ELI] im Rahmen einer Bachelorarbeit implementiert worden [Muehe09]. Der Übersetzungsvorgang lässt sich in drei Schritte unterteilen:

1. Erzeugung einer Aktion für jeden Signalwert von jedem Eingabesignal aus dem Wörterbuch, die das Signal auf den Signalwert setzt.
2. Erzeugung von bedingten Effekten aus den Anforderungen.
3. Zuordnung der bedingten Effekte zu den Aktionen.

4.6.1 Erzeugung der Aktionen aus den Eingabesignalen

Für jeden Zustand *v*, den ein Eingabesignal *s* annehmen kann, wird mindestens eine Aktion erzeugt. Diese Aktion hat den Effekt, dass das Signal *s* auf den Zustand *v* gesetzt wird, unter der Bedingung, dass das Signal vor Ausführung der Aktion einen Zustand *a* aus der Menge der Ausgangszustände hatte. Solch eine Aktion wird für jeden Zustand aus der Menge der Ausgangszustände erzeugt. Der Name der Aktion setzt sich aus dem Namen des Signals *s* gefolgt vom Zustand *v* zusammen. Mit Hilfe von TQEs läßt sich solch eine Aktion folgendermaßen darstellen

```
Aktion s v
       condition: (s=a)@[t1,tstart]
       effect: (s=v)@[tstart,t1]
```

Diese Aktion setzt also den Zustand des Signals *s* auf *v*. Da die Bedingung verlangt, dass das Signal vorher den Zustand *a* hat, stellt diese Aktion einen unmittelbaren Zustandswechsel des Signals von *a* zu *v* zum Zeitpunkt t_{start} dar, wobei *a* laut Wörterbuch ein Ausgangszustand ist, aus dem das Signal zu *v* wechseln kann.

Mithilfe dieser Grundaktionen können alle Signale des Systems aus allen möglichen Ausgangszuständen auf alle ihre möglichen Zustände gesetzt werden. Abbildung 21 zeigt den Zusammenhang zwischen einem Eintrag im Wörterbuch und den daraus erzeugten Aktionen.

Signalname	Zustände	Ausgangs-zustände	Initialzustand	Typ
driver door	open	closed	open	Input
	closed	open		

```
driver door open
  condition: (driver door=closed)@[t1,tstart]
  effect: (driver door=open)@[tstart, t2]

driver door closed
  condition: (driver door=open)@[t1,tstart]
  effect: (driver door=closed)@[tstart, t2]
```

Abbildung 21: Aktionen erzeugt aus Signal im Wörterbuch

4.6.2 Bedingte Effekte aus Anforderungen

Im nächsten Schritt wird aus jeder Anforderung ein bedingter Effekt erzeugt. Hierfür werden die Bedingungen vor dem Auslöser und die Bedingungen nach dem Auslöser den Bedingungen des bedingten Effektes zugeordnet, während die Effekte aus den Reaktionen der Anforderung gebildet werden. Abbildung 22 zeigt, wie eine Anforderung mit dem Namen *R* auf einen bedingten Effekt abgebildet wird.

R	
Condition before the trigger	condBefore
Trigger	trigger
Condition after the trigger	condAfter
Reaction	reaction

```
when (condBefore ∧ condAfter) → (reaction),name=R
```

Abbildung 22: Beziehung zwischen Anforderung und bedingtem Effekt

Um die Zuordnung zwischen bedingten Effekten und den Anforderungen zu erhalten, erhält jeder bedingte Effekt ein Attribut *name*, welches auf den Namen der Anforderung gesetzt wird, für die der Effekt erzeugt worden ist.

Beim Betrachten dieser Abbildung stellt man fest, dass der Trigger nicht an der Bildung des bedingten Effektes beteiligt ist. Es stellt sich also die Frage, wie der Trigger der Anforderung in das formale Modell abgebildet wird. Diese Frage wird nun erörtert.

4.6.3 Zuordnung zu Aktionen

Die erzeugten bedingten Effekte müssen im nächsten Schritt Aktionen zugeordnet werden. Hierbei wird jeder bedingte Effekt den Aktionen zugeordnet, die den Zustand herstellem, der im Feld Trigger notiert ist. Wenn zum Beispiel im Trigger der Ausdruck `driver door is open` notiert ist, so wird der bedingte Effekt, der aus dieser Anforderung entsteht, der Aktion

`driver door open`, die im Schritt eins aus dem Wörterbuch erzeugt worden sind, zugeordnet.

Solch eine Zuordnung kann jedoch nur dann vorgenommen werden, wenn im Trigger ein Eingabesignal verwendet wird. Die Zuordnung, die vorgenommen wird, falls das im Trigger verwendete Signal ein Ausgabesignal ist, ist folgendermaßen gestaltet: Nehmen wir an, dass im Trigger der Anforderung *A* der Ausdruck „*s* is *v*“ notiert ist, wobei das Signal *s* ein Ausgabesignal ist. In diesem Fall kann das Signal nur auf den Wert *v* gesetzt werden, indem eine Anforderung *B* zur Ausführung gebracht wird, deren Reaktion beinhaltet, dass das Signal *s* auf den Wert *v* gesetzt wird. Das Ausführen dieser Anforderung *B* führt dann dazu, dass das Ereignis „*s* is *v*“ eintritt, was dem Trigger der Anforderung *A* entspricht. Das Ausführen der Anforderung *B* löst also unter Umständen das Ausführen der Anforderung *A* aus.

Um dieses Verhalten auszudrücken, werden geschachtelte bedingte Effekte verwendet. Der bedingte Effekt der Anforderung *A* entspricht, wird in den bedingten Effekt der Anforderung *B* geschachtelt. Abbildung 23 zeigt dieses Vorgehen.

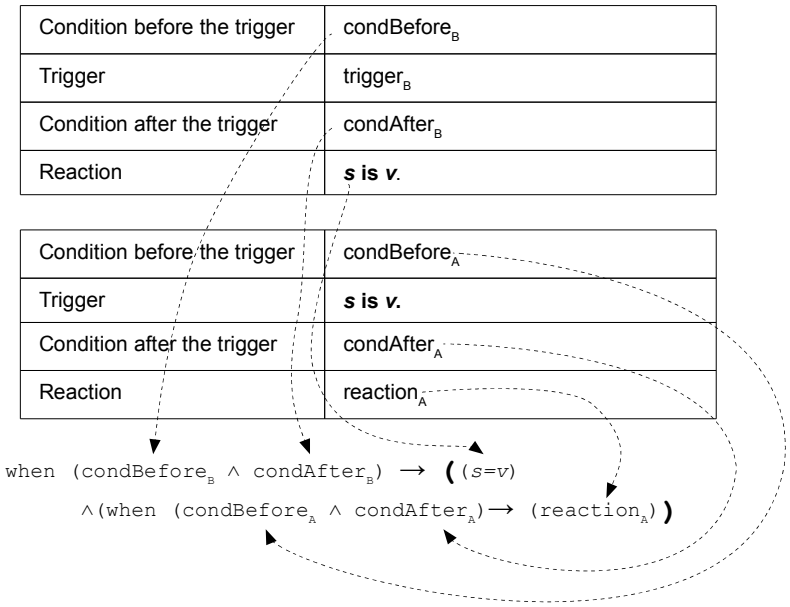


Abbildung 23: Geschachtelter bedingter Effekt

Auf den nächsten Seiten wird die Frage geklärt, wie die einzelnen Satzteile, die in den Feldern der Anforderungsschablone notiert sind, in die Notation des formalen Modells übersetzt werden.

4.6.4 Übersetzungsmuster

Für jedes Element aus der Grammatik sind Regeln definiert, wie dieses Element in die formale Notation zu übersetzen ist. Im Folgenden werden die Übersetzungsregeln vorgestellt, die für jede Produktion der Grammatik definiert sind. Hierbei werden die Produktionen aus der Grammatik für jeden Eintrag aus der Anforderungsschablone wiederholt und anschlie-

ßend wird dargestellt, wie die jeweilige Produktion in CTQEs übersetzt wird. Zum besseren Verständnis werden die CTQEs durch Grafiken verdeutlicht.

4.6.4.1 Bedingungen vor dem Auslöser

Die Bedingungen vor dem Auslöser werden durch Sätze der Form `signal is value [duration]` gebildet, die mittels `and` verknüpft werden können. Die Angabe von `Duration` ist hierbei optional und kann von der Form

- `for` (`at least` / `more than`) *time*
- `for` (`less than` / `at most`) *time*
- `for` (`at least` / `more than`) *time* , `but for` (`at most` / `less than`) *time*
- `for exactly` *time*

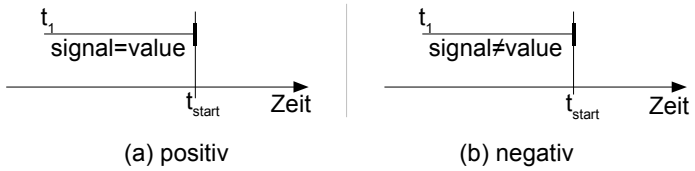
sein, wobei *time* eine Zeitdauer angibt (also z.B. „6 sec“.). Für jede dieser Möglichkeiten wird nun die Übersetzung erläutert. Aufgrund der Tatsache, dass die Bedingung vor dem Auslöser liegen muss, werden alle Intervalle, welche in der Formalisierung benutzt werden, vor dem Auslösezeitpunkt t_{start} liegen.

Ohne Zeitdauer

Die Zeitbedingung `signal is value` ohne Zeitdauer wird in folgende CTQE übersetzt:

$$\text{pos}((signal = value)@[t_1, t_{start}]),$$
$$\text{neg}_1((signal \neq value)@[t_1, t_{start}])^*.$$

Die positive Aussage sagt aus, dass das Signal `signal` irgendwann vor dem Auslösezeitpunkt auf den Wert `value` gesetzt worden ist und diesen Wert bis zum Auslösezeitpunkt behält. Die Zeitachse in Abbildung 24a stellt dies grafisch dar. Die Tatsache, dass t_1, also der Beginn des Intervalls, dort nicht durch einen vertikalen Strich gekennzeichnet ist, drückt aus, dass die Variable nicht eingeschränkt ist. Es muss nur gelten, dass $t_1 < t_{start}$. Dies bedeutet, dass der Beginn und somit die Länge des Intervalls unwichtig sind. Für die Negation wird verlangt, dass der Wert im gleichen Intervall nicht gilt.



(a) positiv (b) negativ

Abbildung 24: Bedingung vor dem Auslöser ohne Zeitdauer

Von unten beschränkte Zeitdauer

Die Bedingung `signal is value for at least` *time* wird folgendermaßen in eine CTQE übersetzt:

$$\text{pos}((signal = value)@[t_1, t_{start}]; t_{start} - t_1 \geq time),$$
$$\text{neg}_1((signal \neq value)@[t_1, t_{start}]), \text{neg}_2(...)$$

Diese Formalisierung verlangt, dass das Signal zum Auslösezeitpunkt seit mindestens *time* Zeiteinheiten auf den Wert *value* gesetzt ist. Die Negation neg_1 drückt aus, dass das Signal zum Auslösezeitpunkt nicht den gewünschten Wert hat. Diese Negation ist allen Bedingungen gemein und wird daher in den nachfolgenden Übersetzungsmustern nur noch als $neg_1(...)$ aufgeführt. Die zweite Negation neg_2 entspricht der Aussage `signal is value for less than` *time*. Da sich die Negation durch einen positiven Satz in der KNS ausdrücken lässt, wird im weiteren nicht näher auf die Negationen eingegangen, sondern es wird auf die Erklärung der positiven Aussagen verwiesen. Die Bedingung `signal is value for more than` *time* wird analog hierzu übersetzt.

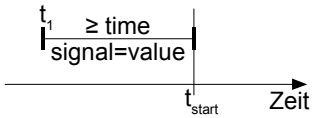


Abbildung 25: signal is value for at least time

Nach oben beschränkte Zeitdauer

Die Bedingung `signal is value for less than` *time* wird folgendermaßen in eine CTQE übersetzt:

$$\text{pos}((signal \neq value)@[t_2, t_3] \wedge (signal = value)@[t_1, t_{start}]; t_{start} - t_3 \leq time \wedge t_3 < t_1),$$
$$\text{neg}_1(...), \text{neg}_2(...)$$

In der positiven Aussage beschreibt die zweite TQE, dass das Signal zum Zeitpunkt t_{start} den Wert *value* besitzt, während die erste sicherstellt, dass das Signal spätestens *time* Zeiteinheiten vor t_{start} noch einen Wert ungleich *value* besaß. Zu beachten ist hier, dass aus $t_{start} - t_3 \leq time \wedge t_3 < t_1$ folgt, dass $t_{start} - t_1 < time$. Zusammen genommen stellen die TQEs sicher, dass das Signal zum Zeitpunkt t_{start} für weniger als *time* Zeiteinheiten den Wert *value* besitzt. Abbildung 26 zeigt eine grafische Darstellung des positiven Anteils der CTQE.

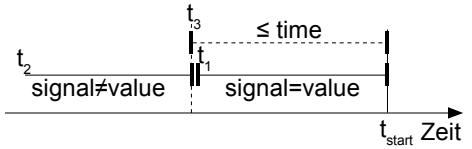


Abbildung 26: signal is value for less than time

Neben der Standardnegation neg_1 kann die Aussage durch `signal is value for at least` *time* negiert werden. Diese Aussage wird in neg_2 formalisiert. Die zweite Möglichkeit eine Bedingung mit nach oben beschränkter Zeitdauer auszudrücken ist `signal is value for less than` *time*. Das Übersetzungsmuster hierfür wird analog zu dem gerade vorgestellten gebildet.

Von unten und nach oben beschränkte Zeitdauer

Die Bedingung `signal is value for at least` *time₁*`, but for less than` *time₂* wird folgendermaßen übersetzt:

$$\begin{aligned}&\mathrm{pos}\big((signal \neq value)@[t_2, t_3] \wedge (signal = value)@[t_1, t_{start}];\\ &\quad t_{start} - t_3 \leq time_2 \wedge t_3 < t_1 \wedge \boldsymbol{t_{start} - t_1 \geq time_1}\big),\\ &\mathrm{neg}_1(...), \mathrm{neg}_2(...), \mathrm{neg}_3(...)\end{aligned}$$

Diese Übersetzung entspricht der Übersetzung von `for less than` *time₂*, die hier durch die fett gedruckte Bedingung aus `for at least` *time₁* ergänzt worden ist. Die Negation *neg₂* entspricht `signal is value for more than` *time₂* und die Negation *neg₃* entspricht `signal is value for less than` *time₁*. Abbildung 27 zeigt eine grafische Darstellung der positiven Aussage.

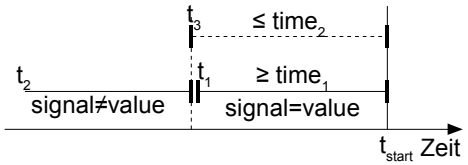


Abbildung 27: signal is value for at least time₁, but for less than time₂

Exakte Zeitdauer

Eine korrekte Formalisierung der Bedingung `signal is value for exactly` *time* sähe folgendermaßen aus:

$$\begin{aligned}&\mathrm{pos}((signal \neq value)@[t_2, t_1) \wedge (signal = value)@[t_1, t_{start}]; t_{start} - t_1 = time),\\ &\mathrm{neg}_1(...), \mathrm{neg}_2(...), \mathrm{neg}_3(...)\end{aligned}.$$

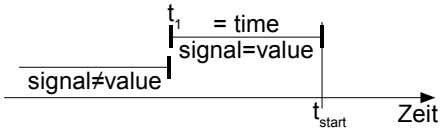


Abbildung 28: Mathematisch korrekte Formalisierung von `signal is value for exactly` time

Ähnlich wie bei den nach oben beschränkten Zeitdauern wird die Bedingung mit Hilfe von zwei TQEs kodiert. Hier stellt die zweite TQE sicher, dass das Signal für exakt *time* Zeiteinheiten den gewünschten Wert *value* besitzt, während die erste TQE benutzt wird, um sicherzustellen, dass das Signal in einem früheren, direkt angrenzenden Intervall nicht auf den Wert gesetzt ist.

Diese Kodierung setzt allerdings voraus, dass der Zeitpunkt t_1, zudem der Signalwechsel vorgenommen wird, exakt bestimmt werden kann. In einer realistischen Umgebung ist dies jedoch nicht möglich, da eingebettete Systeme häufig mit einer gewissen Verzögerung reagieren. Um dies zu berücksichtigen, wird typischerweise bei der Entwicklung eingebetteter Systeme eine maximale Verzögerungszeit angegeben, innerhalb der das System reagieren

muss. Dies hat zur Folge, dass für das Eintreten kein exakter Zeitpunkt, sondern nur ein Intervall angegeben werden kann.

Um dies zu berücksichtigen, wird bei der Formalisierung der exakten Zeitdauer eine gewisse Toleranz *d* angegeben, deren genauer Wert vom Benutzer gesetzt werden kann. Dies führt zu der folgenden Formalisierung:

$$\text{pos}((signal \neq value)@[t_2, t_3] \wedge (signal = value)@[t_1, t_{start}]; t_{start} - t_1 = time \wedge t_3 - t_1 \leq d \wedge t_3 < t_1),$$
$$\text{neg}_1(...), \text{neg}_2(...), \text{neg}_3(...)$$

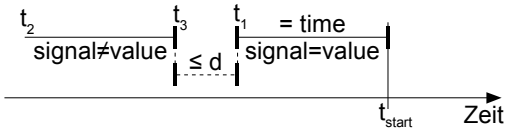


Abbildung 29: `signal is value for exactly` `time` *mit Toleranz*

Das heißt die Formalisierung verlangt, dass das Signal zum Auslösezeitpunkt für eine Dauer zwischen *time* und *time*+*d* Zeiteinheiten auf den Wert *value* gesetzt ist. Die Negationen dieser Aussage sind die Standardnegation, `signal is value for more than` *time+d* sowie `value is signal for less than` *time*.

4.6.4.2 Bedingungen nach dem Auslöser

Die Bedingungen nach dem Auslöser können auf ähnlich Weise wie die Bedingungen vor dem Auslöser gebildet werden. Sie können durch Ausdrücke der Form `[SimpleTimePoint] signal is value [Duration]` gebildet werden, wobei `Duration` den bereits beschriebenen Ausdrücken zur Beschreibung von Zeitdauern entspricht. Darüber hinaus sind Ausdrücke der Form `[SimpleTimePoint] signal is not used in other reactions LowerBound` erlaubt. Auf diese Form der Bedingung wird am Ende dieses Abschnittes eingegangen.

Die optionale Angabe eines Zeitpunktes auf die sich die Bedingung bezieht kann mittels eines `SimpleTimePoint` der Form *time* `after the trigger` (z.B. „`6 sec after the trigger`“) geschehen. Wird kein Zeitpunkt angegeben, bezieht sich die Bedingung auf den Auslösezeitpunkt t_{start}. Mittels `and` können die Ausdrücke verknüpft werden.

Die Übersetzungsmuster ähneln den Bedingungen vor dem Auslöser. Aufgrund der Tatsache, dass sich die Bedingungen auf die Zeit nach dem Auslösezeitpunkt beziehen, werden in den Formalisierungen Intervalle spezifiziert, die hinter dem Auslösezeitpunkt t_{start} liegen. Wird kein Zeitpunkt angegeben, entsprechen die Übersetzungsmuster denen der Bedingungen vor dem Auslöser, wenn sie am Zeitpunkt t_{start} gespiegelt werden. Dies lässt sich am Ausdruck `signal is value for at least` *time* verdeutlichen, der folgendermaßen übersetzt wird:

$$\text{pos}((signal = value)@[t_{start}, t_1]; t_1 - t_{start} \geq time),$$
$$\text{neg}_1((signal \neq value)@[t_{start}, t_1]), \text{neg}_2(...)$$

Die grafische Darstellung macht deutlich, dass diese Formalisierung der am Zeitpunkt t_{start} gespiegelten Formalisierung für die Bedingung vor dem Auslöser entspricht.

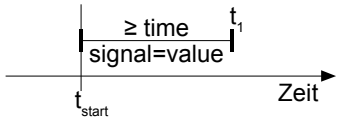


Abbildung 30: `signal is value for at least` `time` *als Bedingung nach dem Auslöser*

Da dieses Vorgehen für alle anderen Ausdrücke gilt, wird an dieser Stelle nicht näher auf die weiteren Übersetzungsmuster eingegangen. Es muss jedoch noch erläutert werden, wie die Angabe eines Zeitpunktes behandelt wird.

Wird eine Bedingung um einen Zeitpunkt $time_n$ `after the trigger` ergänzt, so werden die Übersetzungsmuster dahingehend verändert, dass ein neuer Zeitpunkt $t'_{start}=t_{start}+time_n$ definiert wird. In den Übersetzungsmustern werden anschließend alle t_{start} durch t'_{start} ersetzt. Dies bewirkt, dass die Bedingungen um $time_n$ Zeiteinheiten auf der Zeitachse nach rechts verschoben werden. Dies soll beispielhaft am Ausdruck $time_n$ `after the trigger the signal is value for less than` *time* dargestellt werden. Dieser Ausdruck wird folgendermaßen übersetzt:

$$\begin{aligned}&\mathrm{pos}\big((signal\neq value)@[t_3,t_2]\wedge(signal=value)@[\boldsymbol{t'_{start}},t_1];\\&\quad \boldsymbol{t'_{start}=t_{start}+time_n}\wedge t_3-\boldsymbol{t'_{start}}\leq time\wedge t_1<t_3\big),\\&\mathrm{neg}_1(\ldots),\mathrm{neg}_2(\ldots),\mathrm{neg}_3(\ldots)\end{aligned}$$

Diese Formalisierung ist die gespiegelte Version der Formalisierung aus den Bedingungen vor dem Auslöser, die um $time_n$ Zeiteinheiten nach rechts verschoben ist, wie in der nächsten Abbildung deutlich wird.

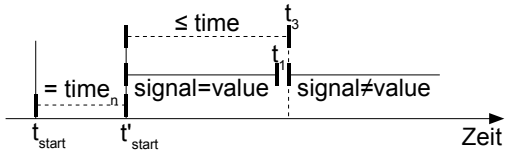


Abbildung 31: `time`$_n$ *`after the trigger the signal is value for less than`* `time` *als Bedingung nach dem Trigger*

Ausdrücke der Form `SimpleTimePoint] signal is not used in other reactions LowerBound` werden ähnlich wie die bereits vorgestellten Ausdrücke übersetzt. Im Positivfall werden diese Ausdrücke so übersetzt, als ob es sich bei `not used in other reactions` um einen Signalwert handelt, sodass die bereits vorgestellten Übersetzungsregeln zum Tragen kommen. Die Negation wird analog zum Positivfall übersetzt, mit der Ausnahme, dass das Signal für den beschriebenen Zeitraum den Wert `used in other reactions` annimmt.

Demzufolge wird der Ausdruck `signal is not used in other reactions for at least` *time* folgendermaßen übersetzt:

$$\begin{aligned}&\mathrm{pos}((signal=not\,used\ \mathrm{in}\ other\ recations)@[t_{start},t_1];t_1-t_{start}\geq time),\\&\mathrm{neg\_1}((signal=used\ \mathrm{in}\ other\ recations)@[t_{start},t_1];t_1-t_{start}\geq time)\end{aligned}$$

Abbildung 32 stellt dies grafisch dar.

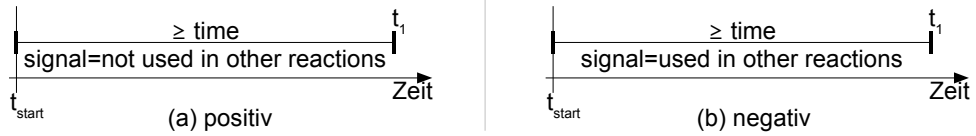


Abbildung 32: `signal is not used in other reactions` als Bedingung nach dem Trigger

4.6.4.3 Reaktionen

Über das Feld `Reaction` wird festgelegt, welche Signale zu welchen Zeitpunkten ihren Wert ändern. Wie bereits erwähnt ist es für eingebettete Systeme üblich, dass Reaktionen nicht punktgenau erfolgen, sondern innerhalb einer gewissen Verzögerung. Hier wird dieser Tatsache Rechnung getragen, indem der Zeitpunkt einer Reaktion innerhalb eines bestimmten Intervalls erfolgen kann. Hierfür wird die bereits bei den Bedingungen erwähnte Konstante *d* benutzt.

Reaktionen können, ähnlich wie die Bedingungen nach dem Auslöser, mit einem Zeitpunkt und einer Zeitdauer versehen werden. Die Möglichkeiten für die Angabe eines Zeitpunktes sind jedoch vielfältiger. Reaktionen haben die Form `[ReactionTimePoint] signal is value [ReactionDuration]`. Die Dauer einer Reaktion kann, ähnlich wie die bisher kennengelernten Dauern, angegeben werden. Falls jedoch eine nach oben beschränkte, oder exakte Dauer angegeben wird, so muss der Ausdruck immer durch Angabe eines Wertes ergänzt werden, auf den das Signal nach Ablauf der Dauer gesetzt wird.

Für die verschiedenen Möglichkeiten eine Reaktion zu spezifizieren werden nun die Übersetzungsmuster vorgestellt.

Ohne Zeitpunkt und Dauer

Ausdrücke der Form `signal is value` spezifizieren, dass das Signal frühestens zum Auslösezeitpunkt mit einer maximalen Verzögerungszeit *d* auf den Wert *value* gesetzt wird. Diese Ausdrücke werden folgendermaßen in eine CTQE übersetzt:

$$\begin{array}{l}\mathrm{pos}((signal = value)@[t_R, t_2]; t_R \geq t_{start} \wedge t_R \leq t_{start} + d), \\ \mathrm{neg}_1((signal \neq value)@[t_1, t_2]; t_1 \leq t_{start} \wedge t_2 \geq t_{start} + d)\end{array}.$$

Im positiven Teil der CTQE wird mit t_R der Zeitpunkt beschrieben, zu dem die Reaktion eintritt. Die Negation in der Übersetzung sagt aus, dass die erwartete Reaktion innerhalb der maximalen Verzögerungszeit nicht eingetreten ist. Diese Negation ist allen folgenden Übersetzungen gemein und wird daher im Folgenden durch $neg_1(...)$ abgekürzt. Abbildung 33a zeigt den positiven Fall und Abbildung 33b den Negativen.

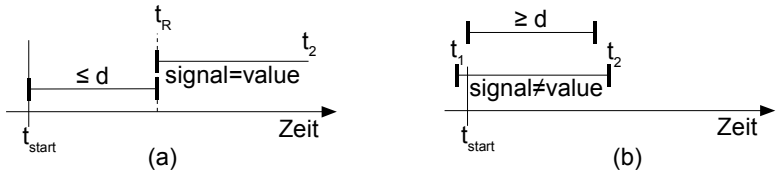


Abbildung 33: `signal is value` als Reaktion. Positiv (a) und Negativ (b)

Reaktion mit Zeitpunkt

Wird bei den Reaktionen ein Zeitpunkt angegeben, so überschreibt dieser die Standardverzögerungszeit *d*. Somit wirkt sich die Angabe eines Zeitpunktes auf die Nebenbedingungen aus, die an den Zeitpunkt der Reaktion t_R gestellt werden.

Für die Angabe eines Zeitpunktes gibt es bei den Reaktionen drei verschiedene Möglichkeiten. Diese werden hier nun zusammen mit den Vorschriften zur Angabe von t_R aufgeführt:

- `at the latest after` *time*$_n$: $t_R \geq t_{start} \wedge t_R \leq t_{start} + time_n$
- `at the earliest after` *time*$_1$, `but at the latest after` *time*$_2$: $t_R \leq t_{start} + time_2 \wedge t_R \geq t_{start} + time_1$

Falls eine exakte Verzögerung angegeben wird, so wird bei der Übersetzung die Standardverzögerungszeit *d* hinzugefügt:

- *time*$_n$ `after the trigger`: $t_R \geq t_{start} + time_n \wedge t_R \leq t_{start} + time_n + d$

Beispielhaft ist in Abbildung 34 die grafische Darstellung für die Aussage `at the latest after time the signal is value` dargestellt.

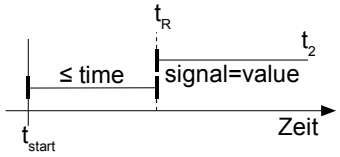


Abbildung 34: `at the latest after time` als Zeitpunkt einer Reaktion

Reaktionen mit von unten beschränkter Dauer

Reaktionen mit von unten beschränkter Dauer haben die Form `signal is value for (at least /more than)` *time* und werden folgendermaßen in eine CTQE übersetzt:

$$\begin{array}{l} \mathrm{pos}((signal = value)@[t_R, t_1]; t_R \geq t_{start} \wedge t_R \leq t_{start} + d \wedge t_1 \geq t_R + time), \\ \mathrm{neg}_1(...) \end{array}.$$

Die positive Formulierung sagt aus, dass das Signal innerhalb der maximalen Verzögerungszeit *d* den Wert *value* annehmen muss und das das Signal diesen Wert mindestens *time* Zeiteinheiten behält.

Reaktionen mit nach oben beschränkter Dauer

Ist die Dauer nach oben beschränkt, folgt aus der Reaktion, dass das in der Reaktion angegebene Signal nur für eine gewisse Dauer den angegebenen Wert besitzt und danach einen anderen Wert annimmt. Die KNS verlangt, dass bei einer nach oben beschränkten Dauer dieser zweite Wert spezifiziert wird, um eine eindeutige Anforderungsspezifikation zu gewährleisten.

Aussagen, die nach oben beschränkte Zeitdauern verwenden, haben demnach immer die folgende Form: `signal is value`$_1$ `(for at most/for less than/for exactly)` *`time`* `and afterwards value`$_2$.

Eine Aussage dieser Art kann als Beschreibung zweier Reaktionen aufgefasst werden: in der ersten Reaktion wird das Signal auf den Wert $value_1$ gesetzt und in der zweiten Reaktion auf den Wert $value_2$. Folgt man dieser Idee, so liegt es nahe, diese Aussage in zwei CTQES zu übersetzen. Beispielhaft soll hier die Übersetzung für `signal is value`$_1$ `for exactly` *`time`* `and afterwards value`$_2$ gezeigt werden. Diese Aussage wird in die folgenden zwei CTQEs übersetzt:

$$\begin{aligned}&\big(\mathrm{pos}((signal = value)@[t_R, T_e];\ t_R \geq t_{start} \wedge t_R \leq t_{start} + d \wedge \boldsymbol{T_e} = t_R + time),\\ &\ \mathrm{neg}_1(...),\end{aligned}$$

$$\begin{aligned}&\big(\mathrm{pos}((signal = value)@[t_{R2}, t_1];\ t_{R2} \geq \boldsymbol{T_e} \wedge t_{R2} \leq \boldsymbol{T_e} + d),\\ &\ \mathrm{neg}_1(...)\big)\end{aligned}$$

Es wird deutlich, dass der Zeitpunkt für die zweite Reaktion t_{R2} über den Endpunkt der ersten Reaktion bestimmt wird. Die Information, wann die erste Reaktion endet, wird über die globale temporale Variable T_e von der ersten CTQE in die zweite übertragen.

Die zwei verbliebenen Übersetzungsmuster für Angaben von Zeitdauern mittels `for at most` oder `for less than` unterscheiden sich von der gerade angeführten Übersetzung in der Nebenbedingung, die in der ersten CTQE an T_e gestellt wird. Für `for at most` *`time`* gilt $T_e \leq t_R + time$ und für `for less than` *`time`* gilt $T_e < t_R + time$.

Wird zusätzlich zu der nach oben beschränkten Dauer eine Beschränkung von unten hinzugefügt, so wird eine weitere Nebenbedingung an den Endpunkt T_e der Reaktion gestellt, wie im vorherigen Abschnitt gezeigt worden ist.

4.7 Testauswahlkriterien

Auf den vorher gehenden Seiten ist ein eindeutiges formales Modell für die Anforderungen präsentiert worden. Im Folgenden soll sich nun der Frage gewidmet werden, wie dieses Modell zur Testfallgenerierung verwendet werden kann.

Der erste Schritt, der hierfür nötig ist, ist geeignete Testauswahlkriterien zu bestimmen, die auf das definierte formale Model angewendet werden können. Hierfür werden zunächst zwei verschiedene Arten von zu erzeugenden Testfällen beschrieben. Die beiden Testarten sind aus Diskussionen mit erfahrenen Testern im Bereich des Steuergerätetests, sowie durch die Analyse von Testfallspezifikationsdokumenten eines Industriepartners entstanden.

Die beiden Testarten sollen nun vorgestellt werden und anschließend wird diskutiert, in welchem Zusammenhang diese zu den klassischen Überdeckungskriterien stehen.

Auf den folgenden Seiten wird das Beispiel des KFZ Alarmsystems zur näheren Erläuterung herangezogen. Die in der KNS formulierten Anforderungen werden daher hier noch einmal wiederholt.

Anforderung 1	
Condition before the trigger	`alarm is set for less than 6 sec.`
Trigger	`driver door is open.`
Condition after the trigger	
Reaction	`alarm is unset.`

Tabelle 7: Zurücksetzen des Alarms durch Öffnen der Fahrertür

Anforderung 2	
Condition before the trigger	`alarm is set for at least 6 sec.`
Trigger	`driver door is open`
Condition after the trigger	
Reaction	`alarm sounder is active.`

Tabelle 8: Auslösen des akustischen Alarms durch Öffnen der Fahrertür

Anforderung 3	
Condition before the trigger	
Trigger	`remote lock button is pressed.`
Condition after the trigger	`remote lock button is pressed for at least 2 sec.`
Reaction	`2 sec after the trigger the alarm is set.`

Tabelle 9: Setzen des Alarms

4.7.1 Positivtests

Die offensichtlichste Testart sind Positivtests, bei der eine Anforderung zur Ausführung gebracht wird und das erwartete Ergebnis überprüft wird. Als Beispiel hierfür dient die Anforderung 2 aus Tabelle 8: Ein Positivtest stellt einen Zustand ein, in dem der Alarm für mindestens 6 Sekunden gesetzt ist und der Signalgeber für den Alarm ausgeschaltet ist. Als Nächstes wird die Fahrertür geöffnet und überprüft, ob der Alarmton nun an ist.

Definition 4.2 (Positivtest). *Für eine Anforderung A mit der Bedingung B, dem Trigger T und der Reaktion R wird ein Positivtest erzeugt, indem das System in einen Zustand gebracht wird, in welchem die Bedingung B gilt, während die Reaktion R nicht gilt. Anschließend wird der Trigger T ausgeführt. Der Test gilt als bestanden, wenn nach dem Ausführen des Triggers die Reaktion R beobachtet werden kann.*

Formalisiert wird dieses Testziel durch Angabe eines Planungsproblems in Form eines *partiellen Plans* inklusive einer Menge von offenen Bedingungen. Dieser partielle Plan definiert zwei Aktionen, die eine Testsequenz enthalten muss, um einen Positivtest zu gewährleisten: eine Aktion *start*, deren Effekte den Initialzustand aller Signale herstellen, so wie sie im Wörterbuch definiert sind und die Aktion *Act* aus dem formalen Modell, die den bedingten Effekt der Anforderung *A* enthält. Die Menge der offenen Bedingungen enthält Bedingungen, für die der Testfallgenerierungsalgorithmus sicherstellen muss, dass diese erfüllt sind. Dies geschieht, indem der partielle Plan weiter verfeinert wird, indem zum Beispiel weitere Aktionen hinzugefügt werden. Dies wird im Abschnitt 4.8 näher erläutert.

In der Definition des Positivtests wird verlangt, dass der Trigger *T* ausgeführt wird. Da die Aktion *Act* den bedingten Effekt der Anforderung *A* enthält, entspricht das Ausführen dieser Aktion dem Auslösen des Triggers der Anforderung. Weiterhin wird verlangt, dass zur Ausführung des Triggers die Bedingungen der Anforderung gelten. Um dies zu gewährleisten, werden die Bedingungen des bedingten Effektes zu der Menge der offenen Bedingungen hinzugefügt.

Um die Reaktion überprüfen zu können, muss sichergestellt sein, dass die Reaktion *R* nicht bereits vor dem zu überprüfenden Zeitpunkt gilt. Um dies zu gewährleisten, wird die Bedingung des bedingten Effektes erweitert.

Sei $R = \wedge_{i=1}^{k} r_i$, wobei r_i jeweils eine CTQE darstellt. Sei $\text{pos}(r_i) \equiv (s_{ri} = v_{ri})@[ts_{ri}, te_{ri}]$ der positive Teil der CTQE und t_{start} der Startzeitpunkt der Aktion *Act*. Sei weiterhin tf_{ri} der früheste Zeitpunkt von ts_{ri} relativ zu t_{start}, also der früheste Zeitpunkt für das Auftreten der Teilreaktion r_i. Die Bedingung des bedingten Effektes wird nun für jede Teilreaktion r_i um die TQE $(s_{ri} \neq v_{ri})@[t_{ri}, tf_{ri}], t_{ri} < tf_{ri}$ erweitert.

Abbildung 35 zeigt den original bedingten Effekt und Abbildung 36 den bedingten Effekt, nach dem er durch die oben beschriebene Prozedur verändert worden ist. Die TQEs über dem ausgefüllten Kasten entsprechen den Bedingungen des bedingten Effektes, während die TQEs unter dem Kasten den Effekten entsprechen.

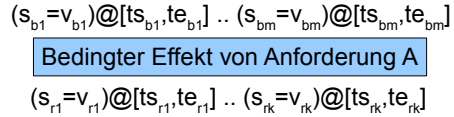


Abbildung 35: Original bedingter Effekt

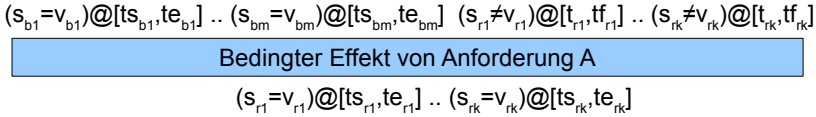


Abbildung 36: Erweiterte Bedingungen des bedingten Effektes

Mit Hilfe dieser Erweiterungen der Bedingungen wird sichergestellt, dass die zu überprüfende Reaktion nicht bereits vor dem Prüfungszeitpunkt hergestellt worden ist. Zu guter Letzt werden die Erweiterungen zu den offenen Bedingungen hinzugefügt. Für das obige Beispiel der Anforderung aus dem Alarmsystem sieht der partielle Plan demnach wie in Abbildung 37 zu sehen aus.

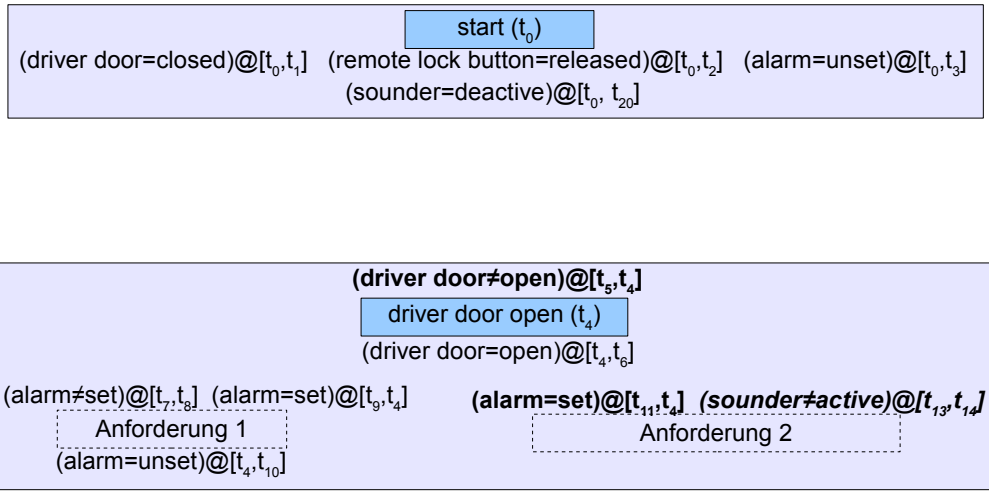


Abbildung 37: Partieller Plan für Positivtest von Anforderung 2

In dieser Abbildung ist die Aktion *start* abgebildet, deren Effekte den Initialzustand des Testobjekts herstellen. In diesem Beispiel ist der Alarm nicht gesetzt, der Knopf der Funkfernbedienung nicht gedrückt und die Fahrertür ist geschlossen. Die Aktion startet zum Zeitpunkt t_0.

Weiterhin ist die Aktion *driver door open* abgebildet, da das Öffnen der Fahrertür der Trigger der zu testenden Anforderung 2 ist. Die Aktion beinhaltet die bedingten Effekte, die den Anforderungen 1 und 2 entsprechen. Die offenen Bedingungen im partiellen Plan sind fett gedruckt dargestellt. Die Bedingungen, die aufgrund der oben besprochen Erweiterungen hinzugefügt worden sind, sind kursiv dargestellt.

4.7.2 Negativtests

Zusätzlich zu Positivtests können Negativtests automatisiert erzeugt werden. Hierbei wird das zu testende System in einen Zustand gebracht, der die Bedingungen einer Anforderung nicht erfüllt. Ein Auslösen des Triggers darf in diesem Fall *nicht* zu der beschriebenen Reaktion führen.

Allerdings ist zu beachten, dass eine Anforderung nur eine „*wenn dann*“ Beziehung und keine „*genau dann wenn*“ Beziehung beschreibt. Das heißt, wenn die Bedingungen einer Anforderung erfüllt sind, wird die beschriebene Reaktion erwartet. Für den Fall, dass die Bedingungen einer Anforderung nicht erfüllt sind, macht eine einzelne Anforderung keine Aussage über die zu erwartende Reaktion.

Um dennoch eine Aussage über die erwartete Reaktion bei nicht erfüllten Bedingungen machen zu können, wird die folgende Annahme getroffen: Jede zu beobachtende Reaktion auf ein Ereignis, die nicht in den Anforderungen niedergeschrieben worden ist, ist ein Feh-

ler. Unter dieser Annahme können nun Negativtests erzeugt werden, indem nicht mehr nur eine einzelne Anforderung betrachtet wird, sondern alle Anforderungen in ihrer Summe.

Definition 4.3 (Negativtest). *Für jede Anforderung A mit der Bedingung B, dem Trigger T und der Reaktion R wird ein Negativtest erzeugt, indem das System in einen Zustand S gebracht wird, für den gilt:*

1. *Die Bedingung B, sowie die Reaktion R gelten nicht.*
2. *Der Testfall löst keine Anforderung aus, welche die Reaktion R enthält.*

Nachdem sich das System im Zustand S befindet, wird der Trigger T ausgelöst. Der Test ist bestanden, wenn die Reaktion R nicht ausgelöst wurde.

Nach dieser Definition von Negativtests muss demnach ein Zustand eingestellt werden, der die Bedingung der Anforderung nicht erfüllt. Dieser Zustand wird jedoch nicht beliebig hergestellt, sondern nach einem Schema, das möglichst sinnvolle Negativtests generiert. Hierfür wird die folgende Idee verfolgt: Eine Bedingung besteht aus UND-Verknüpfungen verschiedener Signalzustände (z.B. Fahrertür ist geschlossen und Klemme 15 ist aus und Innenlicht ist aus.) Um nun Negativtestfälle zu erzeugen, wird jedes Element dieser Verknüpfung einmal negiert und hierfür ein Testfall erstellt.

Die so erzeugten Testfälle werden deshalb als sinnvoll erachtet, da so typische Fehler, die während der Entwicklung der Software des Steuergerätes passieren, entdeckt werden können:

- Der Entwickler vergisst, einen Teil der Vorbedingung abzuprüfen.
- Der Entwickler überprüft die falschen Bedingungen (zum Beispiel `Klemme 15=an` anstelle von `aus`)

Als Beispiel für Negativtests sei die folgende Anforderung genannt:

Condition before the trigger	`automatic mode is active and light-sensor is night.`
Trigger	`driver door is open.`
Condition after the trigger	-
Reaction	`interior light is on.`

Für diese Anforderung können zwei Negativtests generiert werden:

1	2
set automatic mode = **inactive** set lightsensor = night set driver door = open check if interior light is off.	set automatic mode = active set lightsensor = **day** set driver door = open check if interior light is off.

Existiert jedoch eine weitere Anforderung, die einem dieser Negativtests widerspricht, dann muss laut Punkt 2 der Definition des Negativtests entweder sichergestellt werden, dass diese Anforderung durch den Testfall nicht ausgelöst wird oder dieser Testfall darf nicht generiert werden. Dies wäre zum Beispiel gegeben, wenn zusätzlich die folgende Anforderung existiert, welche dem Negativtestfall 2 widerspricht:

Condition before the trigger	`automatic mode is active.`
Trigger	`driver door is open.`
Condition after the trigger	-
Reaction	`interior light is on.`

Wie bereits bei den Positivtests gesehen, wird auch dieses Testziel durch einen partiellen Plan formalisiert: Sei $B = \wedge_{i=1}^{k} b_i$ die Bedingung des bedingten Effektes der Anforderung A. Dann wird für jede Teilbedingung b_i ein partieller Plan erzeugt. Dieser Plan enthält zwei Aktionen: eine Aktion, deren Effekte den Initialzustand aller Signale herstellen, so wie sie im Wörterbuch definiert sind und die Aktion *Act* aus dem formalen Modell, die den bedingten Effekt der Anforderung A enthält.

Die Bedingungen B werden nun wie bereits skizziert mutiert, indem b_i negiert wird, während alle anderen Teilbedingungen unverändert bleiben. Weiterhin müssen die Bedingungen des bedingten Effektes wie bei den Positivtests erweitert werden, um das fehlende Auftreten der Reaktion beobachtbar zu machen.

Darüber hinaus muss sichergestellt sein, dass keine andere Anforderung die Reaktion herstellt, also keine Anforderung existiert, die dem Negativtest widerspricht. Dies geschieht, indem die Bedingungen abermals erweitert werden: Für jede Teilreaktion r_i wird die Negation von r_i zu den Bedingungen hinzugefügt.

Da die Reaktion aufgrund der negierten Bedingung nicht mehr auftreten darf, müssten die Effekte des bedingten Effektes eigentlich gelöscht werden. Da bei der späteren Testausführung jedoch die Information benötigt wird, welche Reaktion nicht eintreten darf, werden die Effekte als inaktiv markiert, anstatt sie zu löschen.

4.7.3 Zusammenhang zu anderen Testauswahlkriterien

An dieser Stelle soll geklärt werden, wie die vorgestellte Generierung von Positiv- und Negativtests zu den aus der Literatur bekannten Testauswahlkriterien steht, wie sie in Kapitel 2.3.1 vorgestellt worden sind.

Bezüglich des Kriteriums der Anforderungsüberdeckung kann festgehalten werden, dass für jede Anforderung, die in der KNS formuliert worden ist, mindestens ein Positivtest generiert wird. Somit wird jede Anforderung durch einen Testfall abgedeckt.

In der Beschreibung der Negativtests ist bereits deutlich geworden, dass diese Tests in der Lage sein sollen, bestimmte Arten von Fehlern zu finden. Dies wird erreicht, indem die Formalisierung des Originalmodells durch Mutationen verändert wird. Dieses Vorgehen entspricht der Idee von Fehlermodellen als Testauswahlkriterium.

Für die strukturellen Testauswahlkriterien lässt sich Folgendes festhalten.

Theorem 4.1. *Die Menge von Positiv- und Negativtests erfüllt die MC/DC Variante* Restricted Active Clause Coverage *in Bezug auf die in den Anforderungen beschriebenen Prädikate und Klauseln.*

Beweis: Restricted Clause Coverage verlangt, dass jede Klausel k eines Prädikats p einmal zu wahr und einmal zu falsch ausgewertet wird. Darüber hinaus muss sichergestellt werden, dass die Klausel k direkt mit dem Ergebnis des Prädikats p korreliert. Dies ist dann der Fall, wenn ein Wechsel des Wertes von k, während alle anderen Klauseln aus p ihren Wert behalten, zu einem Wechsel des Wertes von p führt.

Die Menge von Positiv- und Negativtests erfüllt dieses Kriterium, da

1. in der Basis KNS die Bedingungen (Prädikate) als UND Verknüpfung vorliegen. Daher stellt ein Positivtest sicher, dass jede Teilbedingung (Klausel) einmal zu wahr ausgewertet wird, da während des Positivtests die gesamte Bedingung erfüllt sein muss.

2. Durch die Negativtests wird darüber hinaus sichergestellt, dass jede Klausel einmal zu falsch auswertet wird während alle anderen Klauseln unverändert zu wahr ausgewertet werden. Aufgrund der Tatsache, dass die Bedingungen als UND Verknüpfung vorliegen führt dies dazu, dass das Prädikat p seinen Wert wechselt.

□

4.8 Testfallgenerierung

Die Testziele, welche im vorherigen Abschnitt definiert worden sind, liegen als partielle Pläne mit einer Menge von offenen Bedingungen vor. Um aus diesen nun ausführbare Testfälle zu generieren, müssen diese so lange verfeinert werden, bis alle offenen Bedingungen durch andere Effekte im Plan erfüllt sind. Dies kann zum Beispiel dadurch geschehen, indem dem Plan eine Aktion hinzugefügt wird, deren Effekt eine offene Bedingung erfüllt.

Im Folgenden wird ein Algorithmus vorgestellt, der dieses Planungsproblem löst. Der verwendete Planungsalgorithmus basiert auf der Idee der *Partial Order Causal Link* (POCL) Planung [Weld94]. Der Umgang mit Zeitbedingungen basiert zu großen Teilen auf den Ausführungen in [GNT04] bezüglich Planung mit Zeitbedingungen.

4.8.1 POCL Planungsalgorithmus mit Zeitbedingungen

Der zur Testfallgenerierung verwendete Planungsalgorithmus wird nun erläutert. Hierzu werden zuerst einige Begriffe definiert. Anschließend wird der Algorithmus als Pseudocode dargestellt und erläutert. Schließlich wird anhand des Beispiels des Positivtests für Anforderung 2, die Funktionsweise noch einmal am Beispiel veranschaulicht.

Ein partieller Plan $\pi = \langle A, C, L, O \rangle$ besteht aus einer Menge von Aktionen A, einer Menge von Constraints C an die temporalen Variablen der TQEs, einer Menge von *kausalen Links* sowie einer Menge von offenen Bedingungen O. Ein kausaler Link $<u \rightarrow b>$ wird vom Planungsalgorithmus eingefügt, wenn der Effekt u die Bedingung b unterstützt. Hierbei wird u als *Unterstützer* bezeichnet und b als *Benutzer*. Ein Effekt u unterstützt eine Bedingung b, wenn der Effekt eine Variable auf einen Wert setzt, wie die Bedingung es verlangt

$$var(u) = var(b) \wedge val(u) = val(b) \tag{4.4}$$

und der Effekt im gesamten Intervall der Bedingung gilt:

$$start(u) \leq start(b) \wedge end(u) \geq end(b)\ . \tag{4.5}$$

Ein kausaler Link $<u \rightarrow b>$ wird *gefährdet* genannt, wenn eine andere Aktion im Plan existiert, die einen Effekt hat, der u widerspricht. Abbildung 38 zeigt einen gefährdeten Link. Der Effekt von Aktion T gefährdet den Link zwischen dem Effekt von Aktion S und der Bedingung C, da dieser dem Effekt des Unterstützers widerspricht.

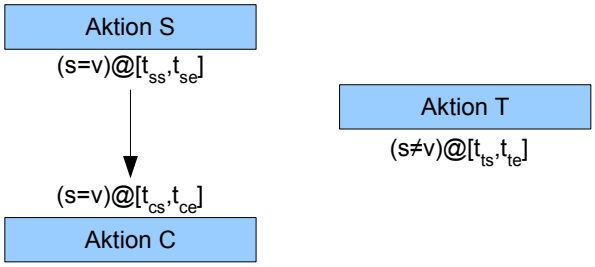


Abbildung 38: Gefährdeter Link

Die Menge von *Flaws* entspricht der Vereinigung von offenen Bedingungen und gefährdeten Links. Die Menge aller Bedingungen, die erfüllt sein müssen, damit ein Effekt e zur Ausführung kommen kann, wird mit *allcond*(e) bezeichnet. Diese Menge enthält die Bedingungen der Aktion, der e zugeordnet ist und, falls e einem bedingten Effekt zugeordnet ist, die Bedingungen des bedingten Effektes.

Mit Hilfe dieser Definitionen kann nun ein POCL Planer beschrieben werden, der die Menge aller partiellen Pläne durchsucht, bis ein Plan mit konsistenten Constraints und ohne Flaws gefunden ist. Abbildung 39 zeigt den Pseudocode für diesen Algorithmus.

Zur Testfallgenerierung wird dem Algorithmus ein partieller Plan übergeben, so wie er anhand der Testauswahlkriterien entstanden ist. Anschließend wird eine beliebige offene Bedingung ausgewählt und es wird versucht einen Effekt u zu finden, der die offene Bedingung unterstützt. Dies kann entweder ein Effekt einer Aktion sein, die bereits im Plan enthalten ist, oder eine neue Aktion wird dem Plan hinzugefügt. Wird dem Plan eine neue Aktion hin-

zugefügt, so werden auch immer die Constraints, die zu der Aktion gehören zu C hinzugefügt und die Menge *allcond(u)* wird zu den offenen Bedingungen hinzugefügt.

Algorithm 1 POCL procedure

POCL(π)
1: if $O = \emptyset$ then return π
2: select any open condition $o \in O$
3: $relevant \leftarrow$Supporters(o, π)
4: if $relevant = \emptyset$ then return(failure)
5: do
6: nondeterministically choose an effect $e \in relevant$
7: $relevant \leftarrow relevant \setminus e$
8: if e belongs to a new action a_{new} then do:
9: update $\mathcal{A}$ with a_{new}
10: update O with $allcond(e)$
11: $\mathcal{L} \leftarrow \mathcal{L} \cup \{\langle e \longrightarrow o \rangle\}$
12: if $\mathcal{C}$ is not consistent then do:
13: $\mathcal{L} \leftarrow \mathcal{L} \setminus \{\langle e \longrightarrow o \rangle\}$
14: if a_{new} added then remove a_{new}
15: goto 6
16: until $\mathcal{C}$ is consistent or $relevant = \emptyset$
17: if $relevant = \emptyset$ return(failure)
18: for each unsafe link $l \in UL(\pi)$ do:
19: $resolvers \leftarrow$Resolvers(l, π)
20: if $resolvers = \emptyset$ then return(failure)
21: nondeterministically choose a resolver in $resolvers$
22: add this resolver to π
23: return POCL(π)
end

Abbildung 39: POCL Algorithmus

Sobald ein unterstützender Effekt ausgewählt worden ist, wird ein kausaler Link in den Plan eingefügt. Dies bedeutet, dass die Constraints wie in Formel (4.5) in C eingefügt werden. Wenn dies zu inkonsistenten Constraints führt, wird ein anderer Unterstützer ausgewählt. Wenn kein unterstützender Effekt gefunden wird, der zu konsistenten Constraints führt, wird für diesen Rekursionsschritt ein Fehler zurückgeliefert. Dieser Vorgang ist in den Zeilen 5 – 17 des Pseudocodes aufgeführt.

Das Hinzufügen einer Aktion oder eines kausalen Links kann zu gefährdeten Links führen (siehe Abbildung 38). Hier wird der Link zwischen Aktion S und C durch den Effekt von

Aktion *T* gefährdet. Um sicherzustellen, dass der Effekt von *T* den Link nicht mehr gefährdet, gibt es drei Möglichkeiten:

1. Bei der *Promotion* wird sichergestellt, dass der gefährdende Effekt erst dann beginnt, nachdem die Bedingung den Link nicht mehr benötigt. Hierfür wird die Nebenbedingung $t_{ce} < t_{ts}$ eingefügt.
   Falls der gefährdende Effekt der gleichen Aktion wie die Bedingung zugeordnet ist, wird anstelle der oben genannten Nebenbedingung die Bedingung $t_{ce} \leq t_{ts}$ eingefügt. Dies ist damit begründet, dass im Abschnitt 4.5.2.3 definiert worden ist, dass sich Effekte und Bedingungen in ihren Endpunkten widersprechen dürfen, falls diese der gleichen Aktion zugeordnet sind. Dies ist nötig, um atomare Zustandswechsel zu ermöglichen.

2. Bei der *Demotion* wird sichergestellt, dass der gefährdende Effekt vor dem unterstützenden Effekt ausgelöst wird. Dies wird durch die Bedingung $t_{ts} < t_{ss}$ sichergestellt.

3. Wenn es sich bei dem gefährdenden Effekt um einen bedingten Effekt handelt, kann verhindert werden, dass dieser eintritt, indem sichergestellt wird, dass die Bedingung des bedingten Effekts nicht wahr werden kann. Hierzu wird ein Element der Bedingung negiert und zu den offenen Bedingungen hinzugefügt. Dieses Vorgehen wird als *Konfrontation* bezeichnet [Weld94].

Bisher nicht erwähnt worden ist, wie der Algorithmus mit TQEs der Form $(signal = not\ used\ \text{in}\ other\ reactions)@[t_1, t_2]$ umgeht, was nun nachgeholt wird: Für offene Bedingungen dieser Form wird ein Link eingefügt, in dem die offene Bedingung sowohl Unterstützer als auch Benutzer ist. Abbildung 40 zeigt dieses Vorgehen.

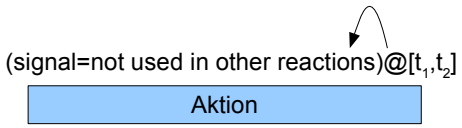


Abbildung 40: Einfügen eines Links zum Schützen eines Intervalls

Das beschriebene Vorgehen führt dazu, dass jeder Effekt, der das Signal *signal* benutzt, den Link gefährdet und somit kein solcher Effekt in das Intervall $[t_1, t_2]$ fallen kann. Eine Ausnahme bilden hierbei Effekte, die dem gleichen bedingten Effekt wie die gefährdete Bedingung angehören. Diese Effekte sind Teil der Anforderung, zu der auch die Bedingung gehört und daher wird definiert, dass sie diese Bedingung nicht gefährden, da die Bedingung verlangt, dass das Signal nicht in *anderen* Reaktionen verwendet werden darf.

Eine offene Bedingung der Form $(signal = used\ \text{in}\ other\ reactions)@[t_1, t_2]$ hingegen wird aufgelöst, indem ein unterstützender Effekt für diese Bedingung dem Plan hinzugefügt wird. Ein Effekt *u* unterstützt die genannte Bedingung, wenn der Effekt die Variable *signal* auf einen Wert beliebigen Wert setzt

$$var(u) = signal \tag{4.6}$$

und der Effekt während des Intervalls $[t_1, t_2]$ auftritt

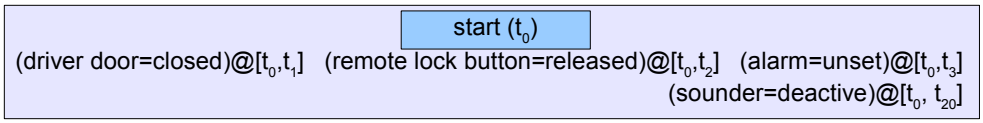
$$start(u) \geq t_1 \wedge start(u) \leq t_2 \,. \tag{4.7}$$

Im Gegensatz zu den bisher vorgestellten Teilen des Algorithmus wird für diesen unterstützenden Effekt kein kausaler Link eingefügt, da die offene Bedingung keine zeitliche Ausdehnung beschreibt, sie somit nicht durch andere Effekte gefährdet werden kann und daher auch nicht durch einen Link geschützt werden muss.

4.8.2 Beispiel

Als Beispiel für die Erreichbarkeitsanalyse wird ein Positivtest für Anforderung 2 aufgeführt. Der partielle Plan für dieses Testziel ist bereits in Abschnitt 4.7.1 hergeleitet worden. Dieser Plan dient als initialer Plan, von dem aus die Suche startet.

Der initiale Plan ist in Abbildung 41 dargestellt. Die offenen Bedingungen sind fett gedruckt. Das Ziel der Suche ist es nun alle offenen Bedingungen durch Effekte anderer Aktionen zu unterstützen, sodass sichergestellt ist, dass die Bedingungen zum Zeitpunkt der Ausführung der zugehörigen Aktion gelten.



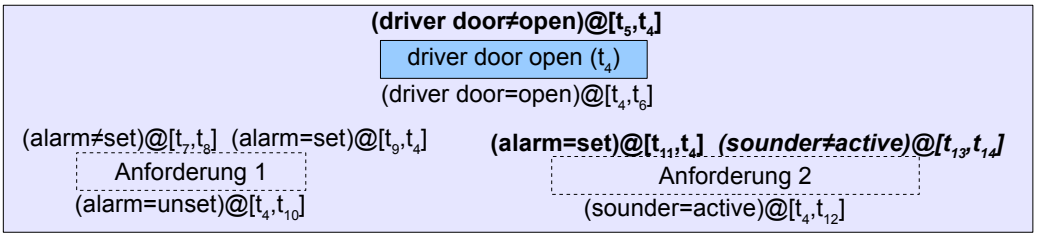


Abbildung 41: Partieller Plan für Positivtest von Anforderung 2

Sei $(alarm=set)@[t_{11}, t_4]$ die offene Bedingung, die als Erstes unterstützt werden soll. Da im bisherigen Plan kein Effekt vorhanden ist, der `alarm` auf `set` setzt, muss eine neue Aktion hinzugefügt werden, die einen Effekt beinhaltet, der dies leistet. Dies trifft auf den bedingten Effekt zu, der Anforderung 3 zugeordnet ist (s. Tabelle 9, Seite 91). Daher wird die Aktion, die diesen bedingten Effekt enthält, hinzugefügt. Weiterhin wird der Plan um einen kausalen Link erweitert, wie in Abbildung 42 zu sehen ist.

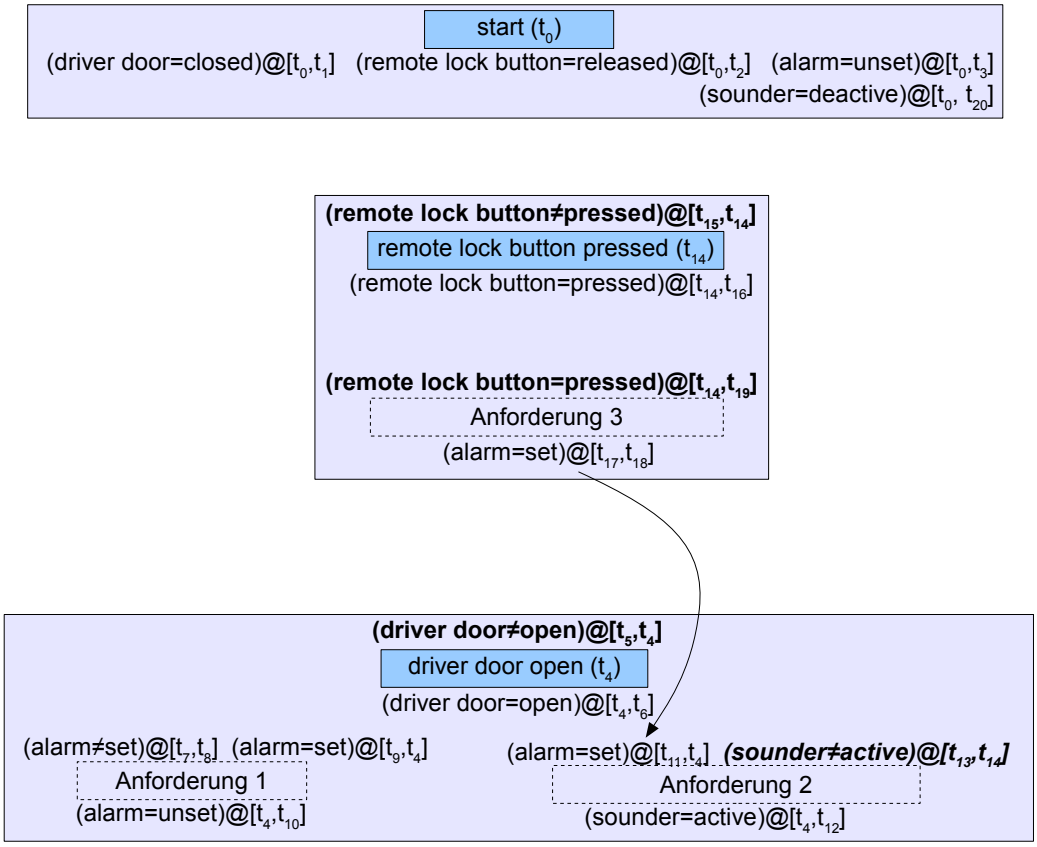


Abbildung 42: Verfeinerter Plan

Der neu hinzugefügte Link ist gefährdet, da die Startaktion den Effekt $(alarm=unset)@[t_0, t_3]$ besitzt und dieser gefährdet den Link. Dieser Effekt ist kein bedingter Effekt und daher kann nur Promotion oder Demotion eingesetzt werden, um die Gefährdung aufzulösen. Promotion würde das Einfügen des Constraints $t_4 < t_0$ verlangen. Dies widerspricht der Annahme, dass alle Zeitpunkte größer als t_0 sind, da t_0 dem frühestmöglichen Zeitpunkt entspricht. Somit ist Demotion die einzig verbliebende Möglichkeit die Gefährdung aufzulösen. Dies wird durch das Einfügen von $t_0 < t_{17}$ erreicht.

Ein weiterer Effekt, der den Link gefährdet, ist der Effekt $(alarm=unset)@[t_4, t_{10}]$ des bedingten Effektes *Anforderung 1*. In diesem Fall ist die Promotion erfolgreich. Hierbei ist zu beachten, dass der gefährdende Effekt zu der gleichen Aktion gehört, wie der benutzende Effekt. Somit dürfen sich die Bedingung, die der Link unterstützt und der gefährdende Effekt an ihren Endpunkten widersprechen. Demnach wird die Promotion durch das Einfügen von $t_4 \geq t_4$ erreicht.

Anstelle der Promotion kann hier auch Konfrontation angewandt werden, da es sich um einen bedingten Effekt handelt. Konfrontation wird erreicht, in dem die Bedingung des bedingten Effektes negiert wird. Die Bedingung von Anforderung 1 lautet `alarm is set for less than 6 sec`. Laut dem vorgestellten Übersetzungsmustern ist eine Negation hiervon `alarm is set for at least 6 sec`. Dies entspricht der Bedingung von Anforderung 2. Daher kann diese Negation durch den gleichen Effekt unterstützt werden, wie die Bedingung

von Anforderung 2. Aufgrund der Konfrontation kann der gefährdende Effekt nun nicht mehr zur Ausführung kommen. Abbildung 43 zeigt diese Konstellation.

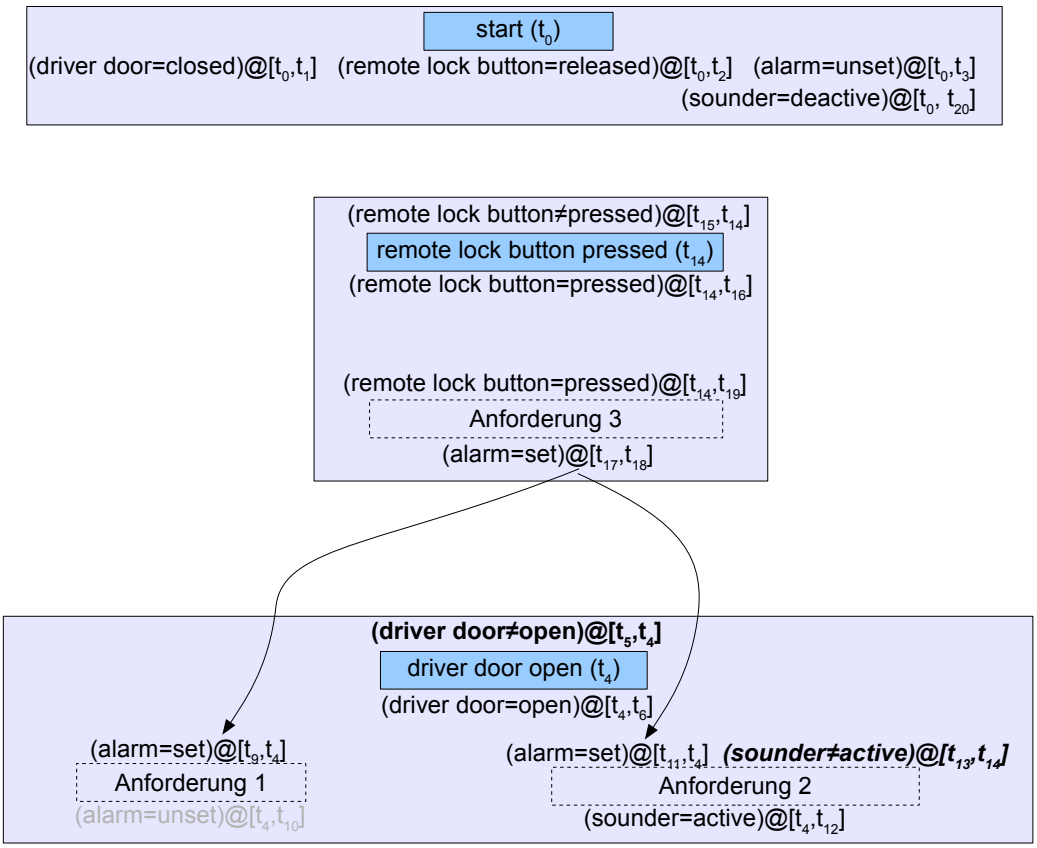


Abbildung 43: Auflösen einer Gefährdung durch Konfrontation

Ausgehend von diesem partiellen Plan, werden im weiteren Verlauf die verbliebenen offenen Bedingungen durch Links unterstützt. Das Ergebnis ist in Abbildung 44 zu sehen.

Bei jedem Einfügen von Aktion sind die temporalen Variablen mitsamt den dazugehörigen Constraints zu der Menge von Constraints des Plans hinzugefügt worden. Diese Menge ist weiterhin bei jedem Hinzufügen eines Links verfeinert worden.

Mit Hilfe dieser Menge von Constraints an die temporalen Variablen im Plan können die möglichen Startzeitpunkte der Aktionen bestimmt werden. In dem hier angeführten Beispiel muss die Aktion `remote lock button pressed` vor der Aktion `driver door open` ausgeführt werden, wobei die Aktion `driver door open` frühestens 8 Sekunden nach der Aktion `remote lock button pressed` ausgeführt werden darf.

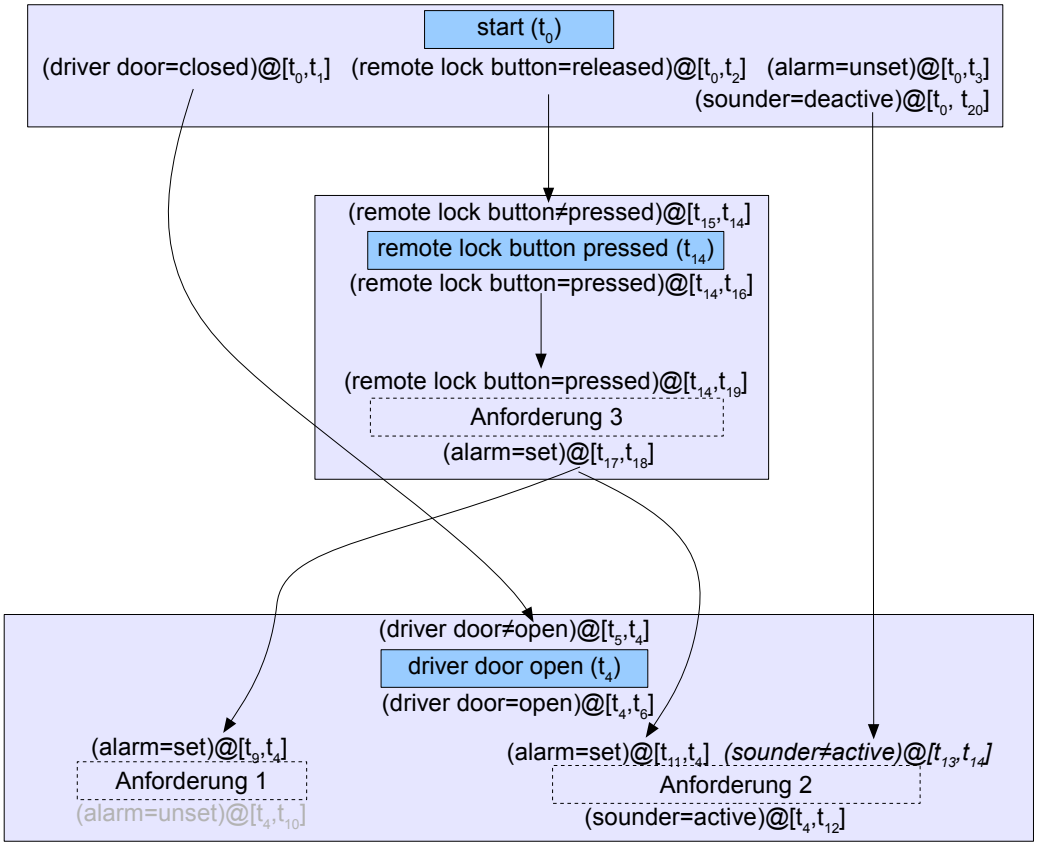


Abbildung 44: Lösungsplan

4.8.3 Erklärung der Testfälle

Die automatisch erzeugten Testfälle sollten für den Entwickler möglichst gut nachvollziehbar sein. Dies hilft einerseits bei der Validierung der Testfälle, indem diese bereits vor der Ausführung vom Entwickler analysiert werden können. Andererseits sind nachvollziehbare Testfälle wichtig bei der Analyse der Testfälle, deren Testergebnis negativ wahr.

Um die Testfälle dem Benutzer zu erklären, können die kausalen Links des Plans zu Hilfe genommen werden. Der Planungsalgorithmus fügt eine Aktion immer nur dann ein, wenn diese benötigt wird, um eine offene Bedingung zu unterstützen. Welche Bedingung durch diese Aktion unterstützt wird, wird durch mindestens einen kausalen Link angezeigt. Diese Links können daher benutzt werden, um zu erklären, warum eine Aktion eingefügt worden ist. In Abbildung 45 ist der Testfall inklusive Erklärungen für den Lösungsplan aus dem vorangegangenen Beispiel zu sehen.

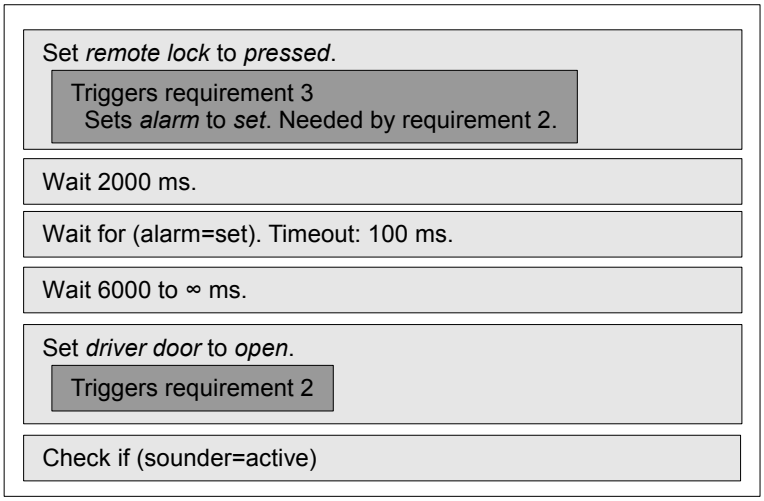


Abbildung 45: Beispiel für die Testfalldarstellung

4.8.4 Heuristiken für die Suche

Der gerade dargelegte Algorithmus enthält zwei Punkte, die sehr wichtig für die Effizienz der Suche sind. Dies ist zum einen die Reihenfolge, in der die offenen Bedingungen ausgewählt werden. Diese Auswahl passiert in Zeile 2 des Algorithmus. Die Auswahl der Reihenfolge stellt keinen Backtrackingpunkt dar. Das heißt, es ist nicht nötig, alle möglichen Reihenfolgen für die Suche zu berücksichtigen. Dennoch ist es sehr wichtig eine gute Reihenfolge zu finden, sodass möglichst wenig Pläne durchsucht werden müssen, wie gleich anhand eines Beispieles erläutert wird.

Der Schritt in Zeile 6 hingegen stellt einen Backtrackingpunkt dar: Alle möglichen Effekte, die die offenen Bedingungen unterstützen können, müssen für die Suche berücksichtigt werden. Auch hier ist die Auswahl des erfolgversprechendsten Effekts wichtig für die Leistung. Für beide Punkte werden in [YS03] Heuristiken vorgestellt, die die Suche an diesen Stellen lenken. Auf diese wird nun näher eingegangen.

4.8.4.1 Heuristik für die Unterstützerauswahl

Für die Auswahl des Effektes, der die gewählte offene Bedingung unterstützen soll, stellt [YS03] eine *additive Heuristik* vor. Um die Heuristik zu beschreiben, starten wir mit der Beschreibung einer Heuristik, welche die Kosten abschätzt, die nötig sind, um eine TQE zu erfüllen. Um diese zu definieren, müssen zunächst einmal einige Definitionen eingeführt werden.

Sei q eine TQE. Sei $U(q)$ die Menge aller Effekte, die q unterstützen, wobei die Zeitbedingungen nicht beachtet werden. Das heißt, es gilt $\forall\, u \in U(q) : var(u) = var(q) \wedge val(u) = val(q)$.

Die Kosten um q zu erfüllen, sind durch die additive Heuristik h_{add} folgendermaßen rekursiv definiert:

$$h_{add}(q)=\begin{cases} 0 & \text{, falls } q \text{ durch eine bereits im Plan vorhandene Aktion unterstützt wird} \\ \min\left(h_{add}(e), \forall\ e\in U(q)\right) & \text{, falls } U(q)\neq\emptyset \\ \infty & \text{, sonst} \end{cases}$$

Die Kosten $h_{add}(e)$ eines Effektes *e* sind definiert als

$$h_{add}(e)=1+h_{add}(allcond(e)). \tag{4.8}$$

Die gezeigte Heuristik schätzt demnach die Anzahl der Aktionen ab, die für das Erfüllen einer offenen Bedingung noch in einen bestehenden Plan eingefügt werden müssen. Zu beachten ist, dass die Kosten im Allgemeinen überschätzt werden, da die Heuristik davon ausgeht, dass einzelne Ziele unabhängig voneinander sind. Das heißt, es wird nicht in Betracht gezogen, dass das Einfügen einer Aktion eventuell mehrere Bedingungen unterstützen kann. Weiterhin werden jegliche Zeitbedingungen ignoriert.

Die Gesamtkosten eines Plans π mit den offenen Bedingungen *O* entsprechen der Summe der Kosten aller offenen Bedingungen:

$$costs(\pi)=\sum_{q\in O} h_{add}(q). \tag{4.9}$$

Für jeden potentiellen Unterstützer *u*, der der Aktion *action(u)* zugeordnet ist, werden die Kosten des partiellen Plans π' berechnet, wobei π' sich aus dem aktuellen partiellen Plan $\pi=\langle A, C, L, O\rangle$ berechnet: $\pi'=\langle A\cup action(u), C, L, O\rangle$. Das heißt, es werden die Kosten für den Plan berechnet, der entsteht falls, die Aktion *action(u)* dem aktuellem Plan hinzugefügt wird.

4.8.4.2 Heuristik für offene Bedingungen

Ein weiterer Punkt, bei der Heuristiken zum Einsatz kommen, ist die Frage, in welcher Reihenfolge offene Bedingungen aufgelöst werden sollen. Dass diese Frage wichtig ist, kann man sich am folgenden Beispiel klar machen.

In dem Beispiel ist ein Plan mit den offenen Bedingungen o_1, o_2 und o_3 gegeben, wobei es für jede offene o_i Bedingung jeweils drei potentielle Unterstützer $s_1(o_i)$,$s_2(o_i)$, $s_3(o_i)$ gibt. Es gilt jedoch die Einschränkung, dass die einzige Möglichkeit, o_1 und o_3 in einem Plan zu unterstützen, durch die Unterstützer $s_3(o_1)$ und $s_3(o_3)$ gegeben ist. Alle anderen Kombinationen der Unterstützer für o_1 und o_2 widersprechen sich.

Abbildung 46 zeigt die Suchabläufe für zwei unterschiedliche Reihenfolgen bei der Abarbeitung der offenen Bedingungen, wobei die Notation $o_i \leftarrow s_j$ bedeutet, dass Bedingung o_i durch Unterstützer $s_j(o_i)$ unterstützt wird. Abbildung 46a zeigt die Reihenfolge o_1, o_2, o_3 während Abbildung 46b die Abarbeitung der Reihenfolge o_2, o_1, o_3 darstellt. Es wird deutlich, dass sich der Aufwand der Suche bei unterschiedlicher Reihenfolge unterscheidet.

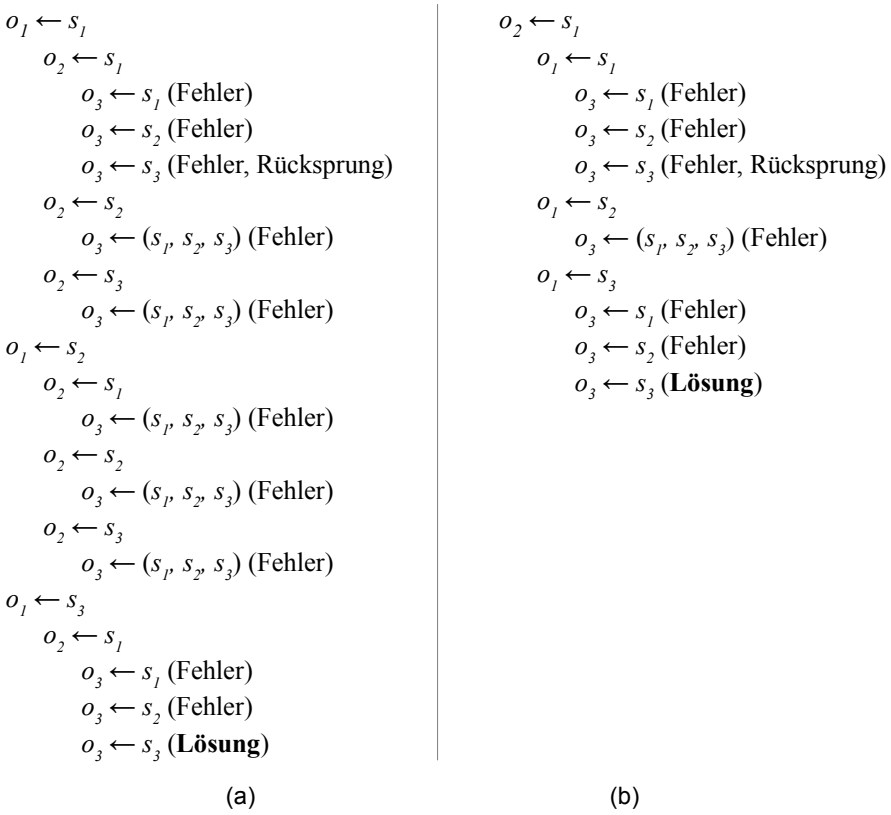


Abbildung 46: Unterschiedliche Reihenfolge der offenen Bedingungen

Es ist somit von Interesse, die offenen Bedingungen in einer Reihenfolge abzuarbeiten, sodass möglichst wenig Pläne durchsucht werden müssen. Um solch eine Reihenfolge zu finden, kommen wiederum Heuristiken zum Einsatz, die aus [YS03] übernommen worden sind.

Eine Möglichkeit verschiedene Reihenfolgen herzustellen, ist es dem FIFO- (First In-First Out) oder dem LIFO- (Last In-First Out) Prinzip zu folgen. In [SG95] wird argumentiert, dass das LIFO-Prinzip vorteilhaft ist, da es dazu führt, dass während des Planungsvorganges ein Ziel verfolgt wird, anstatt mehrere Ziele gleichzeitig zu verfolgen. In [YS03] wird dieses Argument übernommen und experimentell gezeigt, dass das LIFO-Prinzip die bessere Leistung liefert. Die Frage, wie offene Bedingungen sortiert werden sollen, ist damit jedoch noch nicht vollständig beantwortet, wie folgende Überlegung zeigt.

Sei c_{last}, die Menge der Bedingungen, die im letzen Schritt zu der Menge der offenen Bedingungen hinzugefügt worden sind. Dann sagt das LIFO-Prinzip nur aus, dass die Menge c_{last} als erstes abgearbeitet werden soll. Es ist jedoch unklar, in welcher Reihenfolge die Elemente aus c_{last} abgearbeitet werden sollen.

Kostenbasierte Sortierung

Die gerade aufgeworfene Problemstellung wird in [YS03] durch den Einsatz der oben beschriebenen additiven Heuristik h_{add} beantwortet. Hierbei werden die Elemente der Menge c_{last} nach ihren Kosten gemäß h_{add} sortiert.

MC-Loc (Most Cost) bezeichnet eine Sortierung, die die offenen Bedingungen nach dem LIFO-Prinzip sortiert, wobei die Menge c_{last} absteigend nach ihren Kosten sortiert ist, das heißt, die offenen Bedingungen mit den höchsten Kosten werden zuerst bearbeitet.

LC-Loc (Least Cost) entspricht MC-Loc, mit dem Unterschied, dass c_{last} aufsteigend nach den Kosten sortiert ist, also offene Bedingungen mit den niedrigsten Kosten zuerst bearbeitet werden.

Konflikt-orientierte Sortierung

Die Idee der konflikt-orientierten Sortierung [YS03] ist es, die offenen Bedingungen zuerst zu erfüllen, die zu Konflikten mit bereits getroffenen Entscheidungen führen können. Auf diese Weise können große Teile des Suchraumes ausgeschlossen werden. Diese Idee wird verfolgt, indem *unsichere* offene Bedingungen zuerst abgearbeitet werden. Eine offene Bedingung wird als unsicher bezeichnet, wenn jeder Link, der zur Unterstützung dieser Bedingung eingefügt würde, gefährdet wäre.

Auch dieser Ansatz wird in dieser Arbeit verwendet und zwar derart, dass zuerst alle unsicheren Bedingungen und anschließend alle Übrigen offenen Bedingungen abgearbeitet werden.

4.8.4.3 Heuristiken für Konfrontation

Die bisher vorgestellten Heuristiken basieren auf der Arbeit von Younes und Simmons [YS03], die auf früheren Arbeiten [BLG97, BG01, NK01] zur heuristischen Suche aufbauen. Keine der genannten Arbeiten geht auf bedingte Effekte bei der Suche im Planraum ein. Der Grund hierfür ist unter anderem darin zu sehen, dass in den Planungsproblemen, die als Benchmark verwendet werden, bedingte Effekte häufig keine Rolle spielen.

Bedingte Effekte spielen in der vorliegenden Arbeit jedoch eine große Rolle, sodass es von Wert ist, einen Blick auf die Frage zu werfen, inwieweit Heuristiken bei der Behandlung bedingter Effekte sinnvoll eingesetzt werden können.

Der Einsatz von Heuristiken im Zusammenhang mit bedingten Effekten kommt zum Tragen, wenn *Konfrontation* eingesetzt wird, um die Gefährdung eines Links aufzulösen. Beim Einsatz von Konfrontation wird die Gefährdung eines Links durch einen bedingten Effekt dadurch aufgehoben, dass die Bedingung des gefährdenden bedingten Effektes nicht wahr werden kann und somit der Effekt niemals zur Ausführung kommen wird. Dies geschieht, indem die Bedingung des bedingten Effektes negiert wird. Hierfür werden alle möglichen Negationen ausprobiert und zwar solange, bis entweder ein Lösungsplan gefunden wurde, oder alle möglichen Negationen erfolglos waren.

Auch hierbei ist es wichtig eine gute Reihenfolge zu bestimmen, um möglichst wenig Pläne durchsuchen zu müssen. Hierfür kommt die additive Heuristik zum Zuge, indem die Negationen aufsteigend nach ihren Kosten sortiert werden. Dies hat zur Folge, dass zuerst die Negationen ausprobiert werden, für die die niedrigsten Kosten erwartet werden.

4.8.5 Temporales Netzwerk

Der vorgestellte Suchalgorithmus benötigt Verfahren, die es erlauben, eine Menge von Bedingungen an temporale Variablen zu verwalten und die Konsistenz dieser Bedingungen zu überprüfen. In [GNT04] werden hierfür *einfache temporale Netzwerke* (*Simple Temporal*

Network (STN)) [DMP91] vorgeschlagen. Diese Arbeit folgt diesem Vorschlag und daher werden STNs nun vorgestellt.

Werden beliebige Gleichungen als Constraints zugelassen, so ist das Problem, eine Lösung für ein Netzwerk von Constraints zu finden, NP-vollständig [Dechter03]. Da der vorgestellte POCL-Algorithmus jedoch nur Constraints der Form $t_i - t_j \leq a_{ij}, a_{ij} \in \mathbb{R}$ verwendet, kann das Problem in polynomieller Zeit gelöst werden.

Dies geschieht, indem das Netzwerk durch einen gerichteten Graphen mit gewichteten Kanten dargestellt wird. In diesem – *Distanzgraphen* genannten – Graphen repräsentiert jeder Knoten eine temporale Variable. Für jede Bedingung $t_i - t_j \leq a_{ij}$ wird eine Kante $t_i \rightarrow t_j$ eingefügt und diese Kante wird mit $-a_{ij}$ gewichtet.

Die Frage, ob eine gegebene Menge von Bedingungen an temporale Variablen konsistent ist, lässt sich mit Hilfe des Distanzgraphen effizient beantworten.

Theorem 4.2. [DMP91] *Eine gegebene Menge von Bedingungen der Form $t_i - t_j \leq a_{ij}$ ist genau dann konsistent, wenn der zugehörige Distanzgraph keine negativen Kreise enthält.*

Um zu bestimmen, ob der Graph negative Kreise enthält, können Algorithmen zur Bestimmung von kürzesten Wegen in Graphen zum Einsatz kommen. Im speziellen kann entweder der ALL-PAIRS-SHORTEST-PATH Algorithmus von Floyd und Warshall oder aber der SINGLE-SOURCE-SHORTEST-PATH Algorithmus von Bellmann und Ford zur Anwendung kommen. Beide Algorithmen lösen das Problem in polynomieller Zeit (Flyod-Warshall $O(n^3)$, Bellmann-Ford $O(n^2 m)$, wobei *n* die Anzahl der Knoten und *m* die Anzahl der Kanten im Distanzgraph darstellt) [CLR90].

Zusätzlich zu der Frage, ob ein gegebenes temporales Netzwerk konsistent ist, ist die Frage interessant, wie zwei temporale Variablen zeitlich zueinander in Bezug stehen. Um diese Frage zu beantworten, können ebenfalls die kürzesten Wege im Distanzgraphen zur Hilfe genommen werden.

Theorem 4.3. [DMP91] *Bei einem gegebenen konsistenten STN, entspricht die frühestmögliche Zeit für den Zeitpunkt t_i relativ zum Zeitpunkt t_j dem Wert $-d_{ij}$, während die spätestmögliche Zeit dem Wert d_{ji} entspricht, wobei d_{ij} dem kürzesten Weg zwischen dem Knoten t_i und t_j im Distanzgraphen entspricht.*

Die beiden vorgestellten Theoreme zeigen, dass einfache temporale Netzwerke eine effiziente Möglichkeit darstellen, um die im POCL-Algorithmus verwendeten zeitlichen Bedingungen zu verwalten.

Im POCL-Algorithmus wird das STN in jedem Rekursionsschritt um eine oder mehrere Bedingungen erweitert und auf Konsistenz geprüft. Eine Verwendung einer der beiden Algorithmen zur Berechnung der kürzesten Pfade bedeutet, dass in jedem Schritt der gesamte Graph nach negativen Kreisen durchsucht wird, obwohl die Änderungen den Distanzgraphen eventuell nur minimal verändert haben.

Um den Aufwand der Konsistenzprüfung zu reduzieren, wird in dieser Arbeit eine modifizierte Version des Bellmann-Ford Algorithmus eingesetzt, die in [CO96] vorgestellt wor-

den ist. Diese Variante greift bei jeder Veränderung des Distanzgraphen auf die Lösung des vorherigen Schrittes zurück, sodass in den meisten Fällen nur ein Teil der kürzesten Wege neu berechnet werden muss. Auch wenn die Worst-Case Laufzeit weiterhin $O(n^2 m)$ beträgt, so trägt dieses Vorgehen im praktischen Einsatz zu erheblichen Geschwindigkeitsgewinnen bei. Dies ist in [CO96] durch empirische Untersuchungen gezeigt worden.

4.8.6 Nicht deterministisches Zeitverhalten

Die KNS erlaubt es, dass der Benutzer nichtdeterministisches Zeitverhalten spezifiziert. Das hat zur Folge, dass die Zeitpunkte zu denen eine Reaktion eintritt, nicht eindeutig bestimmt sind, sondern die Reaktionen können innerhalb eines gewissen Intervalls auftreten. Somit kann nicht garantiert werden, dass die gefundene Sequenz für jedes Zeitverhalten des Testobjektes ausführbar ist.

Der POCL Algorithmus kann jedoch so verändert werden, dass die erzeugten Pfade must-traceability garantieren. Dies stellt sicher, dass sie für jedes Zeitverhalten des Testobjektes ausführbar sind. Die notwendigen Veränderungen werden nun beschrieben.

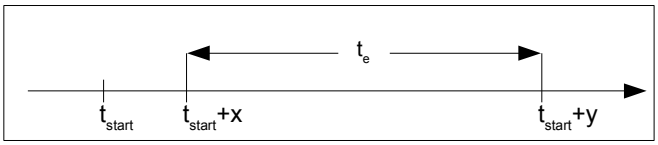


Abbildung 47: Mögliche Zeitpunkte für das Auftreten eines Effekts

In Abbildung 47 ist ein Effekt zu sehen, der zum Zeitpunkt t_e eintritt. Dieser Zeitpunkt kann irgendwo innerhalb des Intervalls $[t_{start}+x, t_{start}+y]$ auftreten, wobei t_{start} den Zeitpunkt des Triggers beschreibt. Dies ist eine typische Situation, wie sie in Anforderungen beschrieben ist. Ein Testfall, welcher must-traceability garantiert, muss derart gestaltet sein, dass er für alle möglichen Zeitpunkte von t_e gültig ist.

Um dies zu erreichen, ist der POCL Algorithmus so verändert worden, dass immer dann wenn ein Constraint eingefügt wird, welches eine temporale Variable t_k relativ zu t_e einschränkt, dieses Constraint durch ein neues Constraint ersetzt wird, so dass t_e immer noch alle Werte aus dem Intervall $[t_{start}+x, t_{start}+y]$ annehmen kann.

Hierfür müssen drei verschiedene Arten von Constraints berücksichtigt werden, wobei die Werte von *x* und *y* durch Anfragen an das temporale Netzwerk bestimmt werden können.

1. Alle Constraints der Form $t_k + a \le t_e$ bzw $t_k + a < t_e, a \in \mathbb{N}$ werden durch $t_k + a \le t_{start} + x$ bzw $t_k + a < t_{start} + x$ ersetzt.
2. Alle Constraints der Form $t_k + a \ge t_e$ bzw. $t_k + a > t_e$ werden durch $t_k + a \ge t_{start} + y$ bzw. $t_k + a > t_{start} + y$ ersetzt.
3. Constraints der Form $t_k + a = t_e$ sind nicht zulässig.

Da diese Ersetzungen stärker sind als die ursprünglichen Bedingungen sind diese Ersetzungen zulässig, in dem Sinne, dass alle erzeugten Pläne gültige Lösungen sind.

Es muss allerdings festgehalten werden, dass nicht für jeden Testfall ein Plan erzeugt werden kann, der must-traceability erfüllt. Als Beispiel seien die folgenden vier Anforderungen *A0 – A3* genannt.

In diesem Beispiel stellen die Signale *a*, *b* und *c* Ausgabesignale dar. Die Signale *a* und *b* werden von der Anforderung *A0* in den Zustand *on* versetzt, wobei *a* sofort in diesen Zustand wechselt, während *b* den Zustand spätestens mit einer Verzögerung von 4 Sekunden einnimmt.

A0	
Trigger	`trigger_0`
Reaction	`a is on` **`and`** `at the latest after 4 sec b is on.`

A1	
Condition before the trigger	`a is on for at most 2 sec` **`and`** `b is on.`
Trigger	`trigger_1`
Reaction	`c is on.`

A2	
Condition before the trigger	`a is on for more than 2 sec` **`and`** `b is on.`
Trigger	`trigger_2`
Reaction	`c is on.`

A3	
Condition before the trigger	`c is on.`
Trigger	`trigger_3`
Reaction	`some reaction.`

Schaut man sich diese Anforderungen genauer an, so fällt auf, dass ein Positivtestfall für die Anforderung *A3* auf zwei verschiedene Arten durchgeführt werden kann, wobei die Entscheidung welche der beiden Varianten gewählt wird, vom konkreten Zeitverhalten der Anforderung *A0* abhängt. Abbildung 48 verdeutlicht diesen Zusammenhang.

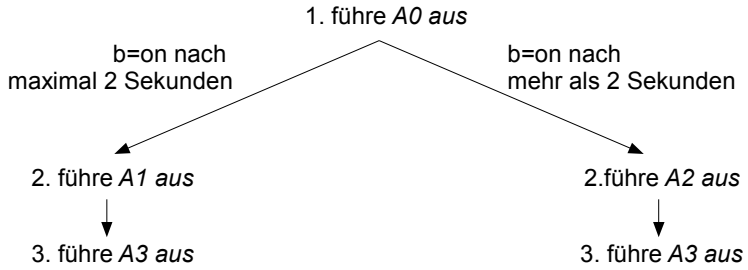


Abbildung 48: Verschiedene Pfade für Positivtest von Anforderung A3

Solche adaptiven Testfälle, die nicht nur aus einer festgelegten Abfolge von Testschritten bestehen, sondern die je nach Zeitverhalten einen anderen Pfad einschlagen müssen, können durch den vorgestellten Algorithmus nicht erzeugt werden. In den untersuchten Fallstudien sind solche Fälle jedoch nicht vorgekommen, sodass die Einschätzung naheliegt, dass diese Einschränkung in der Praxis nicht relevant ist.

4.8.7 Nicht erreichbare Suchziele

Bei der Generierung von Positivtests kann davon ausgegangen werden, dass jede gestartete Suche nach einem erreichbaren Ziel sucht. Dies liegt darin begründet, dass bei Positivtests ein Pfad gesucht wird, der die zu testende Anforderung zur Ausführung bringt, was bei vollständigen und fehlerfreien Anforderungen für jede Anforderung möglich sein muss. Dementsprechend sind die offenen Bedingungen, die dem Suchziel entsprechen, erfüllbar.

Im Falle der Negativtests ist dies nicht der Fall, da hier Teile der Bedingung der zu testenden Anforderung negiert werden und darüber hinaus die offenen Bedingungen des Suchzieles um weitere Bedingungen ergänzt werden (siehe Abschnitt 4.7.2 Negativtests). Somit sind die offenen Bedingungen nicht immer erfüllbar. In einigen Fällen ist der POCL-Algorithmus in der Lage, solche nicht erfüllbaren Bedingungen zu erkennen und die Suche abzubrechen. In diesen Fällen kann festgestellt werden, dass das gesuchte Ziel nicht erreichbar ist.

Ein Beispiel dafür, wie nicht erreichbare Suchziele bei der Suche nach Negativtests entstehen können und welche Faktoren für die Tatsache, ob die Nichterreichbarkeit erkannt wird, wird nun gegeben. Hierzu betrachte man die folgende Anforderung:

Condition before the trigger	c_1 and … and c_n
Trigger	driver door is closed.
Condition after the trigger	driver door is closed for at least 30 sec.
Reaction	...

Bei der Erzeugung von Negativtests wird jede Teilbedingung einmal negiert. Somit wird auch ein Negativtest erzeugt, bei dem die Bedingung nach dem Auslöser negiert wird. Für diesen Negativtest wird der folgende partielle Plan erzeugt.

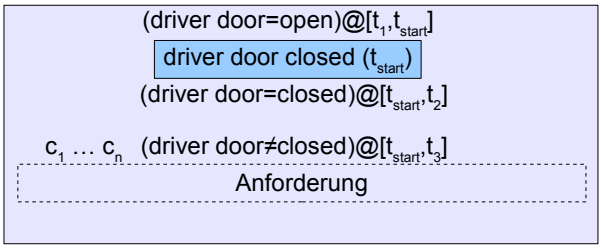


Abbildung 49: Plan mit nicht erfüllbarer Bedingung

Der Plan in Abbildung 49 zeigt, dass die Bedingung nicht erfüllbar ist, da sich der Effekt $(\text{driver door=closed})@[t_{start}, t_2]$ und die Bedingung $(\text{driver door} \neq \text{closed})@[t_{start}, t_3]$ widersprechen. Der POCL-Algorithmus ist in der Lage diesen Widerspruch aufzudecken, da jeder Link, der die Bedingung unterstützt, durch den Effekt gefährdet ist und diese Gefährdung weder durch Promotion, Demotion noch Konfrontation aufgelöst werden kann.

Entscheidend ist an dieser Stelle jedoch, in welcher Reihenfolge die offenen Bedingungen abgearbeitet werden. Wird zuerst versucht die Bedingung $(\text{driver door} \neq \text{closed})@[t_{start}, t_3]$ zu unterstützen, kann die Suche direkt abgebrochen werden, da dieser Versuch fehlschlägt. Werden hingegen zuerst die Bedingungen c_1 bis c_n unterstützt, so kann die Suche eventuell nie abgebrochen werden.

4.8.8 Testfälle für verlinkte Anforderungen

Für die Beschreibung von gleichartigen Funktionen, das heißt Funktionen, die sich nur in den verwendeten Signalen unterscheiden, ist das Konzept der verlinkten Anforderungen beschrieben worden. Dieses Konzept ermöglicht nicht nur ein komfortables Beschreiben von gleichartigen Funktionen, sondern kann auch bei der Testfallgenerierung vorteilhaft ausgenutzt werden, was anhand eines Beispiels dargelegt wird.

Das Beispiel ist ein Auszug aus den Anforderungen an das Türsteuergerät und beschreibt das Verhalten der Ausstiegsbeleuchtung, die in jeder der Türen verbaut ist.

> In den geöffneten Türen leuchtet für die Dauer des Ein– oder Aussteigens, d.h. solange diese Tür geöffnet ist, zudem eine rote Ausstiegsleuchte

Dieses Verhalten wird zunächst einmal exemplarisch für die Fahrertür beschrieben.

driver door exit light on	
Trigger	`driver door is open.`
Reaction	`driver door exit light is on.`

driver door exit light off	
Trigger	`driver door is closed.`
Reaction	`driver door exit light is off.`

Das Verhalten für die Beifahrertür wird nun beschrieben, indem für beide Anforderungen eine verlinkte Anforderung erstellt wird, die die Ersetzungsregel (*driver door → passenger door*) enthält. Das Ergebnis sind die folgenden beiden Anforderungen.

passenger door exit light on	
Trigger	`passenger door is open.`
Reaction	`passenger door exit light is on.`

passenger door exit light off	
Trigger	`passenger door is closed.`
Reaction	`passenger door exit light is off.`

Aufgrund der Tatsache, dass verlinkte Anforderungen sich von den Originalanforderungen nur darin unterscheiden, dass sie über unterschiedliche Signalnamen sprechen, liegt die Erkenntnis nahe, dass auch die Testfälle von verlinkten Anforderungen denen der Originalanforderungen ähneln. So hat zum Beispiel der Plan für den Positivtest für Anforderung *driver door exit light off* die Form wie in Abbildung 50 zu sehen.

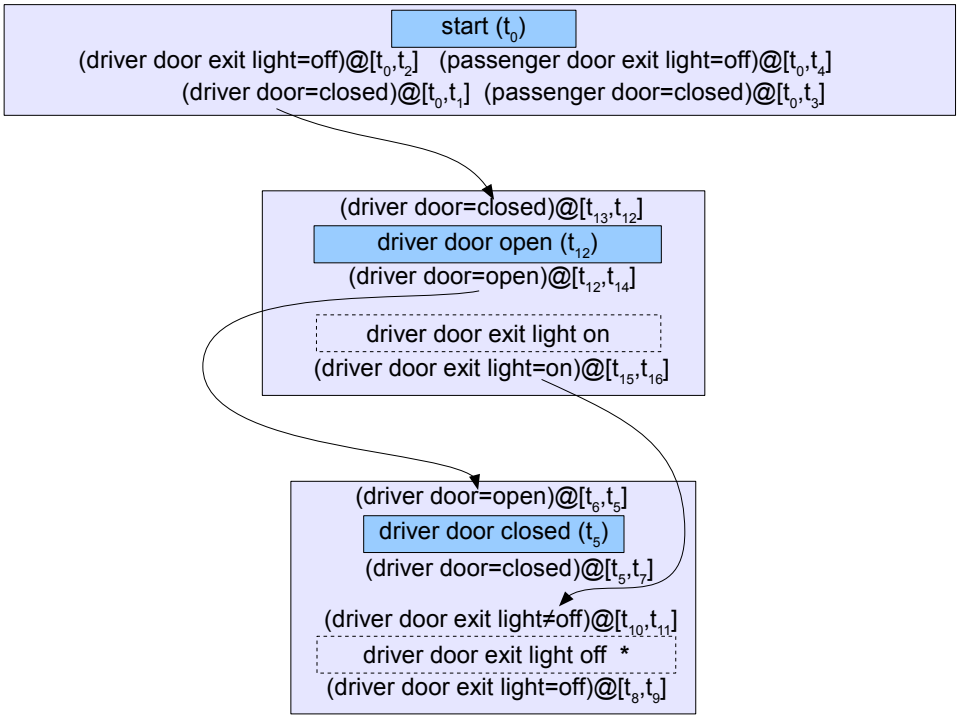


Abbildung 50: Plan für Positivtest der Anforderung driver door exit light off

Der Plan für einen Positivtest der verlinkten Anforderung *passenger door exit light off* hat erwartungsgemäß eine ähnliche Form (Abbildung 51)

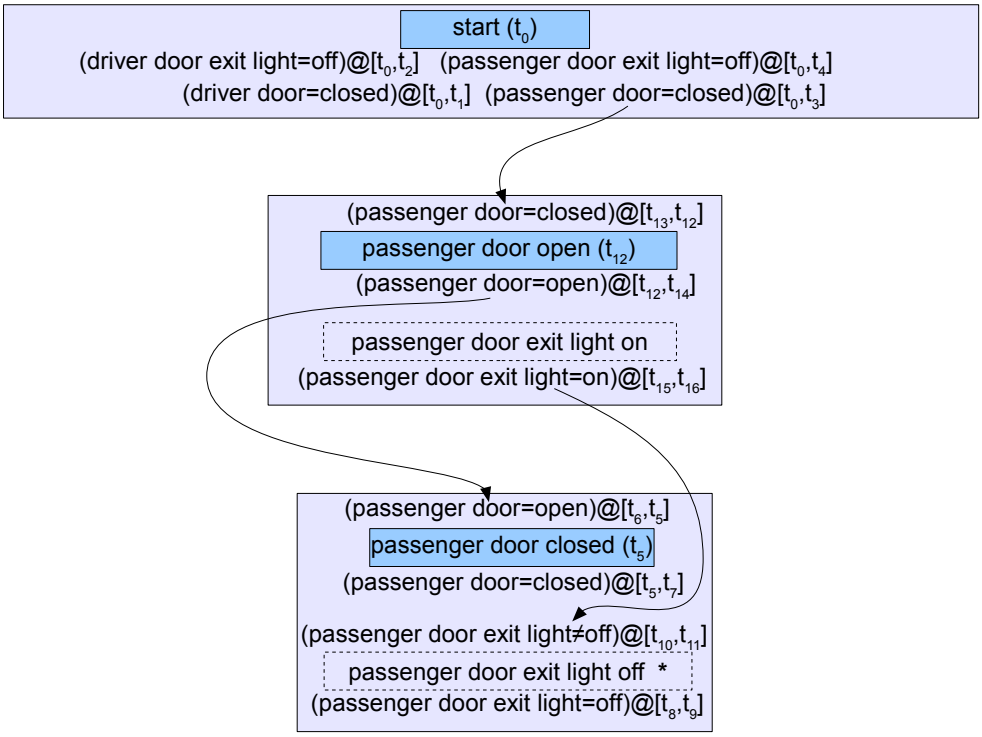


Abbildung 51: Plan für einen Positivtest der Anforderung passenger door exit light off

Vergleicht man beide Pläne, so fällt auf, dass der Plan der verlinkten Anforderung dem Plan der Originalanforderungen entspricht, mit der Ausnahme, dass die verwendeten Aktionen über die Fahrertür anstatt über die Beifahrertür sprechen. Dieser Unterschied entspricht den verwendeten Ersetzungsregeln der verlinkten Anforderung, die jedes Vorkommen von d*river door* durch *passenger door* ersetzten.

Um diese Tatsache für die Testfallgenerierung auszunutzen, werden zunächst die Testfälle für nicht verlinkte Anforderungen erzeugt und anschließend wird versucht, aus den entstandenen Testfällen gültige Testfälle für die entsprechenden verlinkten Anforderungen herzuleiten. Hierfür wird folgendermaßen vorgegangen:

Sei V eine verlinkte Anforderung mit der Originalanforderung O und den Ersetzungsregeln $R=(o_1 \rightarrow r_1, \ldots, o_n \rightarrow r_n)$. Sei P_O ein Plan, der einem Testfall für die Anforderung O entspricht. Um nun einen entsprechenden Testfall für V zu erstellen, wird zunächst wie gehabt ein partieller Plan P_V erzeugt, der dem gewünschten Testziel für V entspricht. Anstatt nun diesen Plan mittels POCL Suche zu verfeinern, wird versucht, die für die Verfeinerung notwendigen Aktionen aus dem bereits existierenden Plan P_O abzuleiten.

Hierzu wird für jede Aktion (außer der Start- und Endaktion) aus P_O die Menge der Ersetzungsregeln auf den Namen der Aktion angewendet und anschließend eine Aktion mit dem dadurch entstandenen Namen in P_V eingefügt. Gibt es im formalen Modell mehrere Aktionen mit diesem Namen, werden alle diese Aktionen durchprobiert.

Nach dem dies für alle Aktionen durchgeführt worden ist, wird überprüft, ob der partielle Plan mit Hilfe dieser Aktionen zu einem Lösungsplan verfeinert werden kann. Ist dies der Fall, so ist ein Testfall für V gefunden worden. Für den Fall, dass durch dieses Vorgehen kein Lösungsplan erzeugt werden kann, wird eine POCL-Suche durchgeführt.

Das oben beschriebene Verfahren gilt nur für den Fall, dass für die Originalanforderung O ein Testfall gefunden worden ist. Falls kein Testfall für O gefunden worden ist, so kann daraus *nicht* geschlossen werden, dass auch kein Testfall für V existiert und es müsste eigentlich für die verlinkte Anforderung erneut eine POCL-Suche durchgeführt werden.

In der Praxis ist es jedoch ratsam diese Entscheidung dem Benutzer zu überlassen, da in den meisten Fällen verlinkte Anforderungen so strukturiert sind, dass in den Fällen in denen für die Originalanforderung kein Testfall gefunden worden ist, auch kein Testfall für die verlinkte Anforderung existiert.

4.9 Ausführen und Bewerten der Testfälle

Das Resultat der Testfallgenerierung ist eine abstrakte Sequenz von Testschritten, die vor der Ausführung noch weiter konkretisiert werden muss. Die Konkretisierung muss hierbei an zwei Stellen erfolgen: Festlegen der Startzeitpunkte der einzelnen Schritte und Implementierung eines Testadapters für den konkreten Zugriff auf das Testobjekt.

Der aus der Suche resultierende Plan enthält Testschritte, deren Startzeitpunkte durch die Bedingungen an die temporalen Variablen im Plan definiert sind. Dies hat zur Folge, dass die Startzeitpunkte nicht durch konkrete Werte festgelegt sind.

Um die nur partiell geordneten Aktionen im Plan zu einer total geordneten Sequenz umzuwandeln, werden die temporalen Variablen, die den Startzeitpunkten entsprechen, sortiert. Dies geschieht durch einen beliebigen Sortieralgorithmus. Der Vergleich $t_i < t_j$ zwischen zwei Startzeitpunkten wird hierbei durch eine Anfrage an das temporale Netzwerk bearbeitet. Ist das Ergebnis des Vergleichs positiv, wird $t_i < t_j$ zu den Bedingungen des Netzwerks hinzugefügt. Bei negativem Ergebnis wird $t_i > t_j$ hinzugefügt. Zum Ende der Sortierung sind schließlich alle Startzeitpunkte zueinander geordnet.

Trotz totaler Ordnung der Aktionen sind die Zeitpunkte, zu denen eine Aktion relativ zu ihrer Vorgängeraktion starten kann, nicht fest, sondern durch ein Intervall möglicher Werte angegeben. Das Intervall möglicher Lösung zwischen zwei Startzeitpunkten kann durch eine Anfrage an das temporale Netzwerk beantwortet werden. Aus diesem Intervall können nun konkrete Werte für einen oder mehrere Testfälle ausgewählt werden. In der prototypischen Implementierung werden hierfür die Grenzwerte des Intervalls gewählt.

Die einzelnen Schritte des Testfalls bestehen in dem Setzen und Überprüfen von Signalen, die im Wörterbuch definiert sind. Jede Aktion des Plans entspricht dem Setzen eines Signales. Jeder Effekt, der verlinkt ist und demnach für die Ausführung des Testfalles benötigt wird, entspricht dem Prüfen eines Signalzustandes am Testobjekt. Neben den verlinkten Effekten müssen weiterhin alle Effekte des bedingten Effektes, der zu der Anforderung gehört, für die der Testfall erzeugt worden ist, überprüft werden.

Die erwarteten Signalzustände lassen sich aus den TQEs des Planes ablesen und die konkreten Zeiten, zu denen die erwarteten Signalzustände herrschen müssen, lassen sich aus den temporalen Variablen, welche die Start- und Endzeitpunkte des Effektes beschreiben, herleiten. Stimmen die Reaktion des Testobjektes mit den erwarteten Werten überein, wird der Test als erfolgreich bewertet, ansonsten gilt der Test als durchgefallen.

Um die Sequenzen ausführen zu können, müssen die abstrakten Schritte des Setzens und Überprüfens von Signalen aus dem Wörterbuch durch konkrete Schritte ersetzt werden.

In diesen konkreten Schritten müssen die Signale tatsächlich im Testobjekt gesetzt/überprüft werden, indem auf die Hardware zugegriffen wird.

In dieser Arbeit wird hierfür ein Testadapter verwendet, der manuell implementiert werden muss. Der Testadapter muss für jeden Wert eines Signals eine Routine zur Verfügung stellen, die diesen Wert am Testobjekt setzt bzw. überprüft.

4.10 Alternatives Lösungskonzept

In diesem Abschnitt wird kurz auf eine Technologie zur Testfallgenerierung eingegangen, die in einer Frühphase dieser Arbeit verfolgt worden ist. Da die Resultate jedoch nicht vielversprechend waren, ist diese Richtung nicht weiterverfolgt worden. Der Vollständigkeit halber werden der Ansatz sowie die Resultate von durchgeführten Experimenten nun aufgeführt.

In [HRV+03] ist untersucht worden, inwieweit sich Modellprüfer zur Testfallgenerierung aus formalen Modellen eignen. Dazu sind verschiedene Model-Checking Verfahren untersucht worden. Hierbei hat sich bei den durchgeführten Experimenten gezeigt, das Bounded-Model-Checking (BMC) [BCC+03] mit Hilfe von SAT-Solvern die vielversprechendste Lösung darstellt. Aufgrund dieser Tatsache ist im Rahmen dieser Arbeit die Eignung von BMC für die Testfallgenerierung für Echtzeitsysteme untersucht worden.

BMC verfolgt ein ähnliches Konzept, wie „Planung als Erfüllbarkeitsproblem“ [BCC+03]. In beiden Fällen ist ein Suchproblem zu lösen und in beiden Fällen wird dieses Problem auf das Lösen des Erfüllbarkeitsproblems zurückgeführt. Hierzu werden das Modell und die nachzuprüfende Eigenschaft durch eine aussagenlogische Formel ausgedrückt. Diese Formel ist genau dann erfüllbar, falls ein Gegenbeispiel für die Eigenschaft existiert.

BMC, das auf reinen SAT-Solvern beruht, kann nicht für zeitbehaftete Systeme angewendet werden. In verschiedenen Veröffentlichungen ist jedoch gezeigt worden, wie das Prinzip des BMC auf zeitbehaftete Systeme übertragen werden kann [ACK+02, Sorea02, DS04]. Hierfür werden anstelle von SAT-Solvern SMT-Solver verwendet, da diese nicht nur aussagenlogische Formeln verarbeiten können, sondern Formeln, die neben einfachen booleschen Ausdrücken auch lineare Ungleichungen enthalten können. Mit Hilfe dieser linearen Ungleichungen können Zeitbedingungen modelliert werden. Diese Möglichkeit wird in [DS04] aufgegriffen und dort wird aufgezeigt, wie zeitbehaftete Systeme für den Model-Checker SAL [BGL+00] modelliert werden können. SAL enthält einen Bounded-Model-Checker, der auf den SMT-Solver YICES [Yices10] zurückgreift.

In einer frühen Phase dieser Arbeit ist die Eignung von BMC zur Testfallgenerierung anhand des Model-Checkers SAL untersucht worden. Hierzu sind die KNS-Anforderungen folgendermaßen in ein Zustandstransitionssystem (ZTS) übersetzt worden:

Zunächst wird für jedes Eingabesignal ein ZTS generiert, welches die möglichen Zustände des Eingabesignals repräsentiert. Beispielhaft ist ein solches ZTS für das Signal `driver door` in Abbildung 52 abgebildet. Die in der Abbildung zu sehende Variable `c_driver_door` stellt eine Uhr dar, die jedes Mal wenn sich der Zustand des Signal ändert, auf Null zurückgesetzt wird. Somit repräsentiert der Wert dieser Uhr die Dauer, für die der aktuelle Wert des Signals gilt.

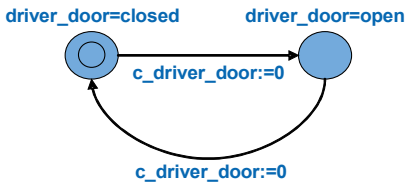


Abbildung 52: Automat für Eingabesignal `driver door`

Aus je einer Anforderung in der Basis-KNS wird dann ein ZTS erzeugt, wie in Abbildung 53 zu sehen. Die Bedingungen vor dem Trigger sind in der Abbildung mit *preCondition* abgekürzt und die Bedingungen nach dem Trigger sind in dieser Abbildung der Einfachheit halber nicht aufgeführt.

Abbildung 53: ZTS für eine Anforderung

Dass das abgebildete ZTS die Semantik der Anforderungsschablone widerspiegelt, kann man sich folgendermaßen klar machen: Durch die Tatsache, dass im zweiten Zustand die Bedingung vor dem Trigger gelten muss, der Trigger jedoch nicht gelten darf, während im darauf folgenden Zustand der Trigger zusammen mit der Bedingung gelten muss, wird sichergestellt, dass während des Zustandsübergangs vom zweitem zum dritten Zustand der zu der Anforderung gehörige Trigger ausgelöst wurde.

Das Gesamtmodell des durch die Anforderungen beschriebenen Systems wird schließlich durch Parallelkomposition aller Anforderungs- und Eingabeautomaten gewonnen. Das Vorgehen soll nun anhand eines Beispiels verdeutlicht werden.

4.10.1 Beispiel

Als Beispiel dienen die folgenden beiden einfachen Anforderungen.

Anforderung 1	
Condition before the trigger	`terminal 15 is off.`
Trigger	`driver door is open.`
Condition after the trigger	
Reaction	`light is on.`

Anforderung 2	
Condition before the trigger	`light is on and driver door is open for at least 5 sec.`
Trigger	`remote unlock is pressed.`
Condition after the trigger	
Reaction	`light is off.`

Die Anforderungen enthalten drei Eingabesignale (*terminal 15*, *driver door*, *remote unlock*), für die jeweils ein Automat erzeugt wird. Weiterhin wird für jede Anforderung ein Anforderungsautomat erzeugt. Diese Automaten sind in Abbildung 54 zu sehen.

Bei diesem Automaten wird davon ausgegangen, dass sich das System zu Beginn in einem Zustand befindet, indem die Tür geschlossen, die Funkfernbedienung nicht gedrückt und Klemme 15 aus ist.

Um nun einen Positivtest für Anforderung 2 zu generieren, ist es nötig, in dem Netzwerk aller Automaten einen Pfad zum Reaktionszustand des Anforderungsautomaten `Anf 2` zu finden, da dieser einem Pfad entspricht, in dem die Anforderung 2 zur Ausführung gekommen ist. Für diese Pfadsuche kann der SAL Model-Checker verwendet werden.

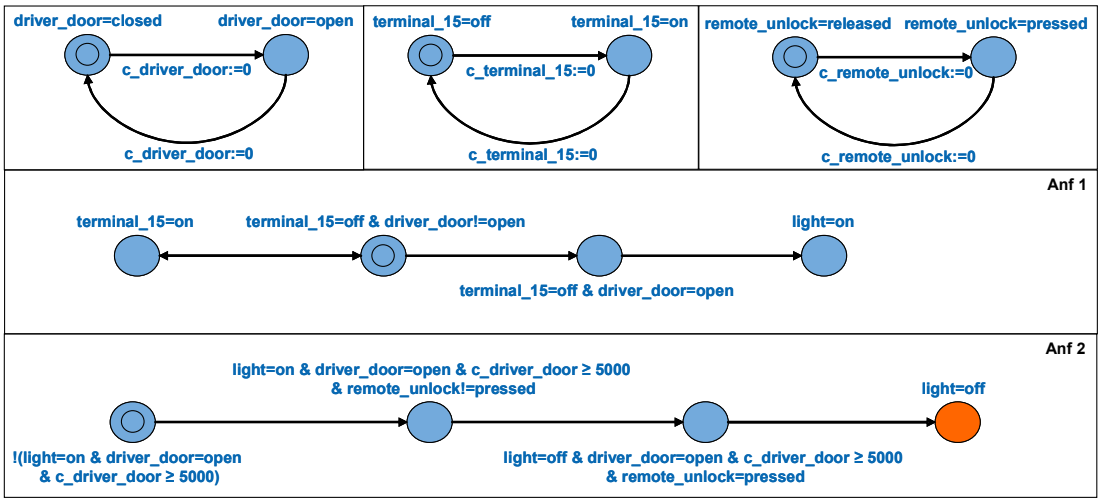


Abbildung 54: Automatennetzwerk

4.10.2 Evaluierung

Tabelle 10 zeigt das Zeitverhalten des Bounded-Model-Checkers SAL bei der Testfallgenerierung im Vergleich zum Planungsalgorithmus. Basis für die Generierung ist eine Anforderungsspezifikation einer Innenlichtsteuerung. Für die Auswertung, die in der Tabelle zu sehen ist, ist die Spezifikation abgeändert worden, sodass sie nur aus 13 Anforderungen und 24 Signalen bestand. Im ersten Durchlauf wurde ein korrekter Testfall gefunden, der aus nur 3 Schritten bestand. Die Spezifikation ist dann schrittweise so verändert worden, dass sich die Länge des Testfalls um jeweils einen Schritt vergrößerte. Es ist deutlich zu sehen, dass die Generierungszeit ab einer Tiefe von 11 sehr schlecht skaliert. Wird für das gleiche Pro-

blem der Planungsalgorithmus verwendet, wie er im Lösungskonzept vorgestellt wurde, so beträgt die Laufzeit für die Berechnung des Pfades Bruchteile einer Sekunde für die Pfadtiefen 3 – 15. Dies zeigt deutlich die Überlegenheit des Planungsalgorithmus für diese Probleminstanz.

#Schritte	3	4	5	6	7	8	9	10	11	15
BMC	2,0s	2,0s	3,2s	3,6s	4,7s	6,5s	7,3s	18s	15Min	22Min
Planung	~50ms	~50ms	~50ms	~50ms	~50ms	~50ms	~50ms	~50ms	~50ms	~50ms

Tabelle 10: Laufzeiten Bounded Modelchecker und Planungsalgorithmus

Zur weiteren Evaluierung wurde die Spezifikation sukzessive um weitere Anforderung ergänzt, wobei die Länge des Testfalls fix blieb. Die hinzugefügten Anforderungen sind so gewählt worden, dass sie nicht mit der Anforderung, für die der Testfall gesucht wurde, kollidieren. Das heißt, die neu hinzugefügten Anforderungen ändern keines der Signale, die für den gesuchten Testfall benötigt wurden. Ein wünschenswertes Ergebnis für diesen Fall wäre ein nahezu konstantes Zeitverhalten, da die hinzugefügten Anforderungen für den Testfall irrelevant sind.

In Tabelle 11 ist das Ergebnis der Auswertung zu sehen. Es ist zu sehen, dass die Laufzeit starke Schwankungen aufweist. Dies ist durch die Arbeitsweise des SMT-Solvers zu erklären, der dem Modelchecker zugrunde liegt. Dieser benutzt zur Lösung des Problems heuristische Suchverfahren, die von der Struktur der Eingabe geleitet werden. Durch Hinzufügen bzw. Entfernen einer Anforderung ändert sich diese Struktur, was aufgrund der Heuristik zu unterschiedlichen Suchbäumen und somit zu unterschiedlichen Laufzeiten führt.

Während beim Bounded Modelchecker die Laufzeit durch das Hinzufügen von irrelevanten Anforderungen beeinflusst wird, bleibt die Laufzeit beim Planungsalgorithmus konstant, was durch die Rückwärtssuche zu erklären ist.

#Anforderungen	13	14	15	16	17	18	19	20	21
BMC	18s	27s	16s	21s	56s	39s	17s	46s	36s
Planning	~50ms	~50ms	~50ms	~50ms	~50ms	~50ms	~50ms	~50ms	~50ms

Tabelle 11: Zeitverhalten relativ zur Anzahl der Anforderungen

5 Realisierung und Evaluierung

In diesem Abschnitt werden Ergebnisse einer Evaluierung des vorgestellten Konzeptes erörtert. Um eine Evaluation des Ansatzes zu ermöglichen, ist das vorgestellte Verfahren prototypisch implementiert worden. Die Implementierung umfasst einen Editor, der eine komfortable Eingabe von Spezifikationen in der KNS ermöglicht, sowie einen Testfallgenerator, der aus solch einer Spezifikation Positiv- und Negativtests erzeugt.

Zur Evaluierung wurden drei Fallstudien durchgeführt. Als Grundlage für die Fallstudien dienten verschiedene Anforderungsdokumente. Die erste Fallstudie entspricht dem Anwendungsbeispiel aus Kapitel 4, welches ein Türsteuergerät modelliert. Als Grundlage für die zweite Fallstudie diente die Spezifikation eines KFZ-Alarmsystems, welche von einem Automobilhersteller zur Verfügung gestellt worden ist. Beide Spezifikationen enthalten Anforderungen im praxisnahen Umfang und sind daher geeignet, die Anwendbarkeit des Lösungskonzeptes in der Praxis zu zeigen.

Die dritte Fallstudie basiert auf einem Modell eines Hausalarmsystems. Dieses Modell ist ein Beispielmodell für das Modellierungswerkzeug MATLAB/SIMULINK. In dieser Fallstudie wurde die gesamte Prozesskette bis zum Ausführen und Bewerten der Testfälle durchlaufen, in dem die erzeugten Testfälle auf dem SIMULINK Modell des Testobjekts ausgeführt worden sind.

5.1 Prototypische Implementierung

Die prototypische Umsetzung der beschriebenen Verfahren soll an dieser Stelle kurz beschrieben werden. Zum einen wird ein Werkzeug vorgestellt, welches zur Eingabe der Spezifikation in der KNS dient und zum anderen werden einige Implementierungsdetails vorgestellt.

5.1.1 Spezifikationseditor

Um den Benutzer den Einstieg in die Verwendung einer KNS so leicht wie möglich zu machen, ist es sinnvoll Werkzeugunterstützung hierfür anzubieten. So sind unter anderen für die kontrollierten Sprachen ATTEMPTO [FSS99] und PENG [SCHWITTER02] Werkzeuge vorhanden, die dem Benutzer Hilfestellungen beim Verfassen von syntaktisch korrekten Sätzen in der KNS bieten [SLH03].

Das durch den Prototyp vorgestellte Konzept folgt der Idee eines voraus-schauenden Editors, wie in [SLH03, KS08] vorgestellt. Hierbei bietet der Editor dem Benutzer Vorschläge an, wie er einen angefangenen Satz vollenden kann, sodass dieser syntaktisch korrekt ist. In [Kuhn08] ist in einer Benutzerstudie gezeigt worden, dass diese Art der Hilfestellung es unerfahrenen Benutzer ermöglicht, komplexe Sätze syntaktisch und semantisch korrekt in einer ihnen vorher unbekannten KNS zu formulieren.

Solche Hilfestellungen sind auch unter den Begriffen *code completion*, oder *IntelliSense* aus Entwicklungsumgebungen für Programmiersprachen bekannt. So bietet u.a. die Entwicklungsumgebung/Rich Client Platform NETBEANS [Boeck06] diese Art der Hilfestellung für verschiedene Programmiersprachen. Um diese Hilfestellung auch für andere Sprachen zur Verfügung zu stellen, bietet NETBEANS eine Schnittstelle, über die weitere Sprachen eingebunden werden können [NBCC10]. In dieser Arbeit wird daher die Rich Client Platform NETBEANS als Basis des Spezifikationseditors verwendet. Über die vorhandene Schnittstelle zur Sprachunterstützung wird die Benutzerunterstützung für die KNS implementiert.

Zur Generierung der Vorschläge für den Benutzer kommt der Parsergenerator JavaCC [Kodagana04] zum Einsatz. Hierfür wird die KNS in einer JavaCC Syntax spezifiziert und schließlich ein Parser durch den Parsergenerator erzeugt. Wird dem erzeugten Parser ein unvollständiger Satz in der KNS übergeben, so gibt der Parser eine Fehlermeldung zurück. Diese Fehlermeldung enthält die Information, welche Token der Parser als Nächstes erwartet hätte. Diese Informationen werden ausgewertet und dem Benutzer als Vorschlag zum Fortführen des Satzes präsentiert.

Der erzeugte Parser wird darüber hinaus verwendet, um nicht definierte Signalnamen und/oder Signalwerte in den Spezifikationen aufzufinden und dem Benutzer durch Hinweise darauf aufmerksam zu machen. Abbildung 55 zeigt ein Bildschirmfoto des Editors.

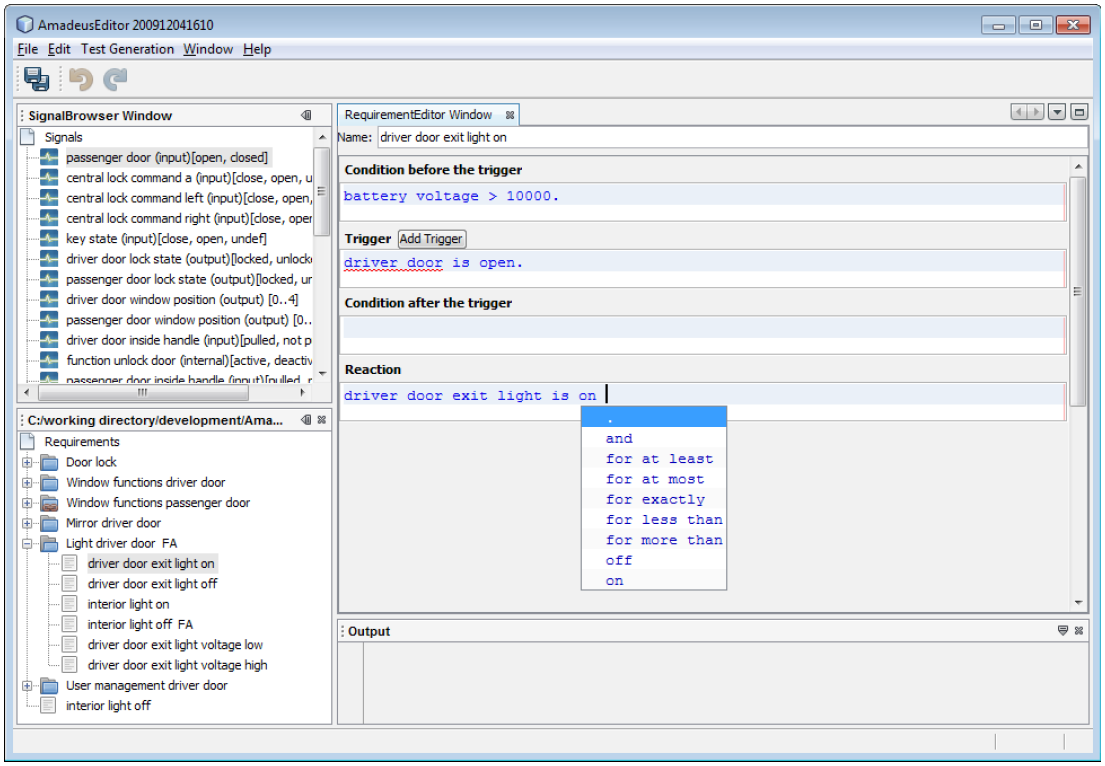


Abbildung 55: Bildschirmfoto des Anforderungseditors mit Benutzerunterstützung

5.1.2 Implementierungsdetails

Die Übersetzer, die die Spracherweiterungen der KNS auf die Basis KNS abbilden, sind mit Hilfe des Compilerbauwerkzeuges ELI realisiert worden. Gleiches gilt für den Übersetzer von der Basis KNS zum Aktionenmodell.

Der implementierte Testfallgenerator liest die textuelle Repräsentation des Aktionenmodells, wie es durch den Eli-Übersetzer generiert wird und erzeugt Positiv- und Negativtests mittels POCL Planung. Die erzeugten Testsequenzen werden in eine XML-Datei ausgegeben, die mittels XSL Skripts für verschiedene Testwerkzeuge aufbereitet werden kann.

Der Algorithmus zur Pfadsuche ist als eigenständiges Werkzeug in Java umgesetzt worden. Der verwendete Algorithmus entspricht Hill-Climbing mit Backtracking, wobei die zu erzeugenden Nachfolger bezüglich der additiven Heuristik sortiert werden. Weiterhin ist die Tiefe der Suche durch einen Parameter *MAX* beschränkt, der durch den Testdesigner vorgegeben werden muss. Im Falle des Erreichens der Tiefenschranke wird ebenfalls Backtracking angewendet.

Ein interessantes Implementierungsdetail des Suchalgorithmus betrifft das temporale Netzwerk. Hierzu muss man sich noch einmal in Erinnerung rufen, welche Arten von Constraints während des POCL-Algorithmus eingefügt werden müssen, wenn ein Effekt eine Bedingung unterstützt. Abbildung 56 zeigt eine Situation mit den Aktionen *S* und *C*, die zu den Zeitpunkten t_s und t_c starten.

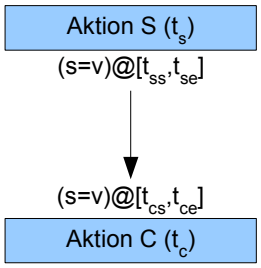


Abbildung 56: Effekt unterstützt eine Bedingung

Der Effekt der Aktion S soll die Bedingung der Aktion *C* unterstützen. Hierfür werden die Constraints $t_{ss} \leq t_{cs}$ (Ungleichung *a*) und $t_{se} \geq t_{ce}$ (Ungleichung *b*) eingefügt. Hierbei drückt Ungleichung *a* aus, dass der Effekt beginnen muss, bevor die Bedingung benötigt wird und Ungleichung *b* sagt aus, dass der Effekt nicht vor der Bedingung beendet sein darf.

Auf Ungleichung *b* kann jedoch verzichtet werden, was durch den folgenden Umstand begründet ist: Die KNS definiert, dass für den Fall, dass die Dauer einer Reaktion nach oben beschränkt ist, immer ein nachfolgender Signalzustand definiert sein muss. Das heißt, dass für den Fall, dass ein Effekt eine nach oben beschränkte Dauer hat, immer ein zweiter Effekt existiert. Falls nun die Dauer des ersten Effektes kürzer ist, als es die zu unterstützende Bedingung verlangt, so gefährdet der zweite Effekt immer den eingefügten Link und diese Gefährdung kann nicht durch Promotion oder Demotion aufgelöst werden. Somit ist Ungleichung *b* nicht notwendig, da entweder der Effekt lange genug andauert, oder aber der Link durch einen zweiten Effekt gefährdet ist.

Für Ungleichung *a* kann festgehalten werden, dass t_{ss} ein Startzeitpunkt eines Effektes ist und daher die Ersetzungsregeln, welche must-traceability garantieren, angewendet wer-

den müssen. Das Resultat der Anwendung dieser Ersetzungsregeln lautet $t_s + y \leq t_{ce}$, wobei y der späteste Zeitpunkt für t_{ss}, relativ zum Startzeitpunkt t_s der Aktion ist.

Für t_{ce} kann festgehalten werden, dass diese Variable nur auf zwei mögliche Weisen eingeschränkt sein kann:

- $t_{ce} \leq t_c + x$, falls es sich bei der zu unterstützenden Bedingung um eine Bedingung vor dem Trigger handelt, oder
- $t_{ce} = t_c + x$, falls es sich bei der zu unterstützenden Bedingung um eine Bedingung nach dem Trigger handelt.

In beiden Fällen ist x durch die in der KNS-Anforderung angegebenen Zeitbedingungen definiert. Daraus folgt, dass Ungleichung a durch $t_s + y \leq t_c + x$ ersetzt werden kann.

Aus diesem Grund werden vor der Suche für alle temporalen Variablen jeder Aktion die möglichen Werte relativ zum Startzeitpunkt ihrer Aktion berechnet und gespeichert. Aus diesen Werten können x und y abgeleitet werden und es werden während des Suchvorgangs nur noch Constraints der Form $t_s + y \leq t_c + x$ in das temporale Netzwerk eingefügt. Dies hat zur Folge, dass das temporale Netzwerk pro Aktion nur eine temporale Variable enthalten muss, nämlich die Variable, die den Startzeitpunkt der Aktion speichert.

Durch die Vorberechnung der temporalen Netzwerke der Aktionen kann demnach das temporale Netzwerk, welches während der Suche benötigt wird, erheblich verkleinert werden.

5.2 Fallstudie Türsteuergerät

In diesem Abschnitt wird beschrieben, wie das entwickelte Lösungskonzept auf das Anwendungsbeispiel des Türsteuergerätes angewandt worden ist. Es werden zunächst die dabei entstanden KNS-Anforderungen beschrieben, um dem Leser einen Eindruck davon zu vermitteln, wie mit der KNS gearbeitet werden kann.

Anschließend wird die Leistung des Testfallgenerierungsalgorithmus bei der Generierung von Positiv- und Negativtests für die erarbeitete KNS-Spezifikation diskutiert. Hierbei werden die Leistungen der verschiedenen Heuristiken gegenübergestellt.

5.2.1 KNS Spezifikation

Im Folgenden wird aufgeführt, wie die Anforderungen des Türsteuergeräts mit Hilfe der KNS beschrieben werden können. Hierbei wird immer zuerst der Originaltext aus dem Anforderungsdokument aufgeführt und anschließend wird erläutert, wie dieser in der KNS umgesetzt worden ist.

5.2.1.1 Türschloss

Generell: Die Funktion soll dem Prinzip folgen, daß entweder alle Fahrzeugtüren verriegelt oder alle Fahrzeugtüren entriegelt sind. Wird eine einzelne Tür ver- bzw. entriegelt, so müssen alle anderen Türen diesem Vorbild folgen. Im Zweifelsfalle hat Entriegeln Vorrang vor Verriegeln (z.B. wenn ein Fahrgast die Tür von innen öffnet

[=entriegeln], während der Fahrer die Türen mittels Funksender verriegeln möchte). Solange wenigstens eine Fahrzeugtür offen ist, kann das Fahrzeug nicht verriegelt werden.

Aufschließen des Fahrzeugs:

Die Fahrzeugtüren werden entriegelt, wenn die Fahrzeugtüren verriegelt sind und entweder:

- mindestens eine Tür (mechanisch) geöffnet wird (d.h. F_OFFEN = 1)
- oder das Signal zum Öffnen der Türen erkannt wird (CAN.ZV_SCHL_A = 01, CAN.ZV_SCHL_L = 01 oder CAN.ZV_SCHL_R = 01)
- oder der Schlüssel zum Aufschließen der Tür verwendet wird (KEY STATE = Aufschließen)

Um die Fahrzeugtüren entriegeln zu können, muß die Batteriespannung BATT mindestens 9V betragen. Unterhalb von 9V arbeiten die Motoren der Zentralverriegelung nicht mehr zuverlässig. Ein selbständiges Nach-Entriegeln der Türen zu einem späteren Zeitpunkt erfolgt i.d.R. nicht.

Ausnahme: Wird zu dem Zeitpunkt, an dem auf Entriegeln erkannt wird, die Entriegelung nicht durchgeführt, weil die Batteriespannung unter 9V liegt und wird gleichzeitig ein Anlaßversuch unternommen (KL 15START= 1), so wird nach einerWartezeit von 1 sec. nachdem KL_15START = 0 gilt nochmals selbständig geprüft, ob die Batteriespannung nun ausreichend ist. Falls ja, wird die Tür nach-entriegelt. [...]

Beim Entriegeln wird [...] an die Ausgänge ZENTR_MOT1 und ZENTR_MOT2 sowie FZENTR_MOT1 und FZENTR_MOT2 [...] jeweils die Spannung -12V für einen Zeitraum von mindestens 2 sec. angelegt. Wird innerhalb von 3 sec. nicht das Entrieglen der Tür erkannt, so wird der Entriegelungsvorgang abgebrochen […]

Um die beschriebene Funktionalität in der KNS umsetzen zu können, müssen zunächst die beteiligten Signale im Wörterbuch definiert werden. Die Signale sind in der nachfolgenden Tabelle aufgeführt, wobei hier nur die Definitionen der Signale für eine Tür – der Fahrertür – zu sehen sind.

Signalname	Zustände	Ausgangszustände	Initialzustand	Typ
battery voltage	[0..13000]		12500	Input
central lock command a	open	close, undef	undef	Input
	close	open, undef		
	undef	open, close		
klemme_15	key out	key inserted	key out	Input
	key inserted	key out, radio on		

	radio on	key inserted, ignition on		
	ignition on	radio on, start		
	start	ignition on		
function unlock door flag	set		unset	Internal
	unset			
key state	idle	open, close	idle	Input
	open	idle		
	close	idle		
driver door lock state	locked		unlocked	Output
	unlocked			
driver door	open	closed	closed	Input
	closed	open		

Mit Hilfe dieser Signale kann nun die Funktion des Entriegelns der Fahrzeugtür durch zwei Anforderungen beschrieben werden. In beiden Anforderungen wird das interne Signal *flag function unlock door* verwendet. Dies dient zur Beschreibung der Prioritäten zwischen Ver- und Entriegelung des Fahrzeuges. Wie genau dies geschieht, wird bei der Betrachtung der KNS-Anforderung zur Beschreibung der Verriegelung klar werden.

Condition before the trigger	
Trigger	driver door is open **or** central lock command a is open **or** central lock command left is open **or** central lock command right is open **or** key state is open.
Condition after the trigger	battery voltage >= 9000 for at least 3 sec.
Reaction	at the latest after 3 sec the driver door lock state is unlocked **and** the flag function unlock door is set.

Die beschriebene Funktion ist eine Abstraktion des Originaltextes in dem Sinne, dass die Ansteuerung der Motoren zum Verschließen der Türen nicht erwähnt wird. Es wird lediglich beschrieben, dass die Türen nach spätestens 3 Sekunden verriegelt sein müssen.

Im Originaltext wird darüber hinaus noch erwähnt, dass unter gewissen Umständen eine Nach-Entriegelung der Türen erfolgt. Diese Funktionalität wird in einer weiteren KNS-Anforderung beschrieben, wie in der nachfolgenden Tabelle zu sehen.

Condition before the trigger	`klemme_15 is start` **`and`** `battery voltage < 9000.`
Trigger	`driver door is open` **`or`** `central lock command a is open` **`or`** `central lock command left is open` **`or`** `central lock command right is open` **`or`** `key state is open.`
Trigger	`klemme_15 is ignition on`
Condition after the trigger	`1 sec after the trigger the battery voltage >= 9000 for at least 3 sec.`
Reaction	`at the latest after 3 sec the driver door lock state is unlocked` **`and`** `the flag function unlock door is set.`

Die oben angeführten zwei Anforderungen beschreiben das Verhalten der Fahrertür. Da das Verhalten der Beifahrertür dem Verhalten der Fahrertür entspricht, kann das Verhalten für die Beifahrertür durch eine verlinkte Anforderung mit den Ersetzungsregeln (*driver door → passenger door*) beschrieben werden.

Als Nächstes wird die Beschreibung der Verriegelung der Türen beschrieben. Dazu wird zunächst der Originaltext zur Beschreibung der Verriegelungsfunktion aufgeführt.

> *Abschließen des Fahrzeugs:* Die Fahrzeugtüren werden verriegelt, wenn die Türen des Fahrzeugs entriegelt sind, alle Fahrzeugtüren geschlossen sind und entweder:
> - das Signal zum Abschließen der Türen erkannt wird
>   (ZV SCHL A= 10, ZV SCHL L = 10 oder ZV SCHL R = 10)
> - oder der Schlüssel zum Abschließen der Tür verwendet wird
>   (KEY STATE = Abschließen)
>
> Um die Fahrzeugtüren verriegeln zu können, muß die Batteriespannung BATT mindestens 9V betragen. Unterhalb von 9V arbeiten die Motoren der Zentralverriegelung nicht mehr zuverlässig. Ein selbständiges Nach-Verriegeln der Türen zu einem späteren Zeitpunkt erfolgt nicht.

Diese Funktion wird folgendermaßen in der KNS umgesetzt:

Condition before the trigger	`driver door is closed and passenger door is closed.`
Trigger	`central lock command a is close` **`or`** `central lock command left is close` **`or`** `central lock command right is close` **`or`** `key state is close.`
Condition after the trigger	`the flag function unlock door is not used in other reactions for at least 3 sec` **`and`** `the battery voltage > 9000 for at least 3 sec.`
Reaction	`at the latest after 3 sec the driver door lock state is locked` **`and`** `at the latest after 3 sec the passenger door lock state is locked.`

Die Bedingung the `flat function unlock door is not used in other reactions for at least 3 sec` in der Bedingung nach dem Auslöser modelliert die Tatsache, dass die Funktion des Entriegelns eine höhere Priorität besitzt als die Funktion des Verriegelns: Ein Auslösen des Entriegelns innerhalb von 3 Sekunden nach dem Auslösen des Verriegelns verletzt die Bedingung des Verriegelns, und somit wird die Verriegelung nicht durchgeführt.

5.2.1.2 Sitzeinstellungen

Im Anforderungsdokument ist beschrieben, wie der Benutzer seine Sitzposition gemäß seinen Wünschen einstellen kann. Der Benutzer kann

- den Winkel der Lehne,
- die Entfernung des Sitzes vom Lenkrad,
- die Höhe des hinteren Sitzbereichs,
- die Höhe des vorderen Sitzbereichs und
- die Schalung des Sitzes

verstellen. Diese Möglichkeit bietet sich sowohl dem Fahrer als auch dem Beifahrer. Um zu vermeiden, dass bei jedem Fahrerwechsel jeder Fahrer seine Einstellungen erneut vornehmen muss, können bis zu vier Sitzeinstellungen des Fahrersitzes gespeichert werden. Hierfür steht ein Benutzermanagement zur Verfügung, über das Einstellungen des Sitzes gespeichert und abgerufen werden können. Die Möglichkeit des Benutzermanagements steht nur für den Fahrersitz zur Verfügung.

Zunächst wird erläutert, wie der Sitz manuell in seiner Position verändert werden kann. Hierfür wird zuerst der relevante Teil aus den Originalanforderungen zitiert.

> Ein Verstellen der Sitzposition über die Sitztaster ist nur möglich, wenn die dem TSG zugeordnete Vordertür geöffnet ist (F_OFFEN = 1). Das Verstellen der Sitzposition über das Benutzermanagement (betrifft nur Fahrerseite) ist auch bei geschlossener

Tür möglich.
Die Sitzposition wird entweder entsprechend der vom Benutzermanagement gesendeten Positionsangaben oder den Sitztasten eingestellt. Dabei gilt das Prinzip, daß immer die zuletzt benutzte Taste (Benutzermanagement oder Sitztaste) die Bewegung des Sitzes bestimmt.

Bewegung des Sitzes: Zur Bewegung des Sitzes werden die in Tabelle [...] angegebenen Spannungen auf die Sitzmotoren gelegt. Die Bewegung wird solange durchgeführt wie

- Ist–Wert und Soll–Wert nicht übereinstimmen (bei Anfahren einer Sitzposition über das Benutzermanagement) bzw. die Sitztasten gedrückt werden (bei Verstellen der Sitzposition über die Sitztasten)
- und der Wert der Sitzposition keine Unterbrechung erkennt.

Bewegung über Sitztasten: Bei der Verwendung der Sitztasten können maximal zwei Bewegungsrichtungen gleichzeitig verwendet werden. Wird während der Sitzverstellung über die Sitztasten eine Taste des Benutzermanagements gedrückt, so wird die Sitzverstellung über Tasten abgebrochen und die Sitzverstellung über das Benutzermanagement begonnen.

Ist die Batteriespannung BATT während der Sitzverstellung kleiner als 10V, so werden die Sitze nicht bewegt bzw. wird die Sitzbewegung abgebrochen. Statt dessen wird die Meldung B LOW SITZ = 1 generiert.

Zur Bewegung der Sitze sind die in der folgenden Tabelle aufgeführten Spannungen an die jeweiligen Motoren anzulegen:

Sitzeigenschaft	***Kürzel***	***Bewegung bei +12V***	***Bewegung bei -12V***	***Geschw.***
Entfernung Sitz – Lenkrad	HOR	nach vorne	nach hinten	0.9 cm/sec.
Sitzfläche vorne	V	heben	senken	0.7 cm/sec.
Sitzfläche hinten	H	heben	senken	0.7 cm/sec.
Schalung	S	heben	senken	0.5 cm/sec.
Lehnenwinkel	W	steiler	senken	3.3°/sec.

Um die aufgeführte Funktionalität mit Hilfe der KNS zu beschreiben, müssen zunächst die benötigten Signale im Wörterbuch definiert werden. Dies geschieht wie folgt, wobei Signale, die bereits im vorherigen Abschnitt definiert worden sind, nicht erneut aufgeführt werden. Weiterhin beschränkt sich die Auflistung auf die Angabe der Signale für den Winkel der Lehne, die anderen Sitzeigenschaften sind analog hierzu definiert.

Signalname	Zustände	Ausgangszustände	Initialzustand	Typ
driver door seat angle position	[-2..2]		0	Output
driver door seat button counter	[0..5]		0	Internal
driver door seat angle button	idle	up, down	idle	Input
	up	idle		
	down	idle		

Tabelle 12: Wörterbuch für die Funktion „Verstellen des Lehnenwinkels“

Das Signal *driver door seat angle* hält den eingestellten Winkel der Sitzlehne fest. Hier ist anzumerken, dass dies aufgrund der Tatsache, dass numerische Signale lediglich ganzzahlige Werte annehmen können, nur eine Abstraktion des tatsächlichen Verhaltens zulässt, da der Winkel der Sitzlehne durch das Steuergerät kontinuierlich verändert wird, was durch den ganzzahligen Wertebereich nicht erfasst werden kann.

Das interne Signal *driver door seat button counter* dient dazu, die Anzahl der gleichzeitig betätigten Einstellungstasten festzuhalten.

Diese Signale werden nun verwendet, um das Verhalten in der KNS zu beschreiben. Hierzu werden zunächst zwei Anforderungen beschrieben, die dazu dienen den *seat button counter* zu erhöhen, falls der Knopf für die Lehnwinkel-Einstellung gedrückt worden ist, bzw. ihn wieder zu vermindern, falls der Knopf losgelassen worden ist.

Trigger	driver door seat angle button is up **or** driver door seat angle button is down.
Reaction	driver door seat button counter=driver door seat button counter + 1.

Trigger	driver door seat angle button is idle.
Reaction	driver door seat button counter=driver door seat button counter - 1.

Anschließend ist es möglich, die Bewegung der Sitzlehne zu modellieren. Auch für diese Funktion werden zwei Anforderungen verwendet. Die Erste beschreibt das Vergrößern des Winkels der Lehne, während die Zweite das Verringern beschreibt.

In beiden Fällen wird vom tatsächlichen Verhalten dahingehend abstrahiert, dass kein kontinuierliches Verändern des Winkels der Lehne beschrieben wird. Anstelle des kontinuierlichen Verhaltens wird die Veränderung beschrieben, nachdem der entsprechende Knopf für genau eine Sekunde gedrückt worden ist.

Hierbei werden die Positionen abstrakt als Positionen -2 bis +2 modelliert. Das Mapping von diesen abstrakten Positionen auf die tatsächlichen Positionen muss manuell vom Testingenieur im Testadapter kodiert werden.

Condition before the trigger	driver door seat button counter<2 **and** driver door is open.
Trigger	driver door seat angle button is up.
Condition before the trigger	driver door seat angle button is up for at least 1 sec **and** battery voltage > 10000 for at least 1 sec **and** driver door is open for at least 1 sec **and** driver door seat angle position is not used in other reactions for at least 1 sec.
Trigger	at exactly after 1 sec the driver seat angle button is idle.
Reaction	driver door seat angle position=driver door seat angle position + 1.

Tabelle 13: Erhöhen des Winkels der Lehne

Condition before the trigger	driver door seat button counter<2 **and** driver door is open.
Trigger	driver door seat angle button is down.
Condition before the trigger	driver door seat angle button is down for at least 1 sec **and** battery voltage > 10000 for at least 1 sec **and** driver door is open for at least 1 sec **and** driver door seat angle position is not used in other reactions for at least 1 sec.
Trigger	at exactly after 1 sec the driver seat angle button is idle.
Reaction	driver door seat angle position=driver door seat angle position - 1.

Tabelle 14: Verringern des Winkels der Lehne

Über die Bedingung `driver door seat button counter < 2`, die vor dem Auslöser gelten muss, wird sichergestellt, dass nicht bereits mehr als ein Knopf zur Sitzverstellung gedrückt worden ist. Dies entspricht der Anforderung, dass maximal zwei Motoren gleichzeitig laufen dürfen.

Die Anforderungen modellieren darüber hinaus die Tatsache, dass die zuletzt angeforderte Funktion (manuelles Verstellen, oder Verstellen mittels Benutzermanagement) eine höhere Priorität genießt. Dies wird über die Bedingung nach dem Auslöser `driver door seat angle position is not used in other reactions for at least 1 sec` erreicht.

Die beschriebenen Anforderungen gelten ebenso für die anderen Einstellmöglichkeiten. Somit können die Verstellmöglichkeiten der verbliebenen vier Sitzeigenschaften durch verlinkte Anforderungen beschrieben werden. So kann zum Beispiel die Einstellmöglichkeit der Höhe der Sitzfläche vorne durch eine Verlinkung der oben aufgeführten vier Anforderungen und der Ersetzungsregel (*seat angle → seat height front*) erreicht werden. Hierfür müssen im Wörterbuch die Signale *driver door seat height front button*, *driver door seat height front position* analog zu den Signalen für den Lehnenwinkel aus Tabelle 12 definiert werden.

Da die beschriebene Funktionalität der Fahrertür ebenso für die Beifahrerseite gilt, können die Funktionen für die Beifahrerseite durch verlinkte Anforderungen mit der Ersetzungsregel (*driver door → passenger door*) beschrieben werden.

5.2.1.3 Fensteröffner

Das Türsteuergerät übernimmt auch die Ansteuerung der Fensterheber. Hierzu stehen dem Benutzer Knöpfe zum Öffnen und Schließen zur Verfügung. Die Funktion des Fensterhebers ist im Anforderungsdokument folgendermaßen beschrieben.

> *Öffnen einer dem TSG zugeordneten Scheibe:* Falls ein Signal anliegt, das das Öffnen einer Scheibe zur Folge hat (siehe Tabelle 8), wird auf dem entsprechenden Motor die Spannung 12V angelegt.
> Liegt die Batteriespannung zum Beginn der Scheibenbewegung unterhalb von 10V, so wird die Scheibenbewegung nicht durchgeführt. Statt dessen wird die CAN–Botschaft B_LOW_WIN = 1 gesendet.
>
> Für die Bewegung sind folgende Fälle zu beachten:
>
> Ist die relevante Schalterstellung gleich *Fenster runter man.* oder ist die relevante CAN–Botschaft WIN_*x*_OP = 01, so wird die Scheibe nach unten bewegt. Die Bewegung endet, wenn das entsprechende Signal nicht mehr anliegt (bzw. nicht mehr gesendet wird), oder sich die Scheibe in der unteren Position befindet (d.h. F UNTEN bzw. FF UNTEN), oder ein anderer Befehl zum Bewegen dieser Scheibe zu einem späteren Zeitpunkt eingeht; in diesem Fall wird der neue Bewegungsbefehl bearbeitet […]
>
> Wurde die Schalterstellung *Fenster runter auto.* oder die CAN–Botschaft WIN_*x*_OP = 10 erkannt, so wird die Scheibe solange nach unten bewegt, bis sich die Scheibe in der unteren Position befindet (d.h. F UNTEN bzw. FF UNTEN), oder ein anderer Befehl zum Bewegen dieser Scheibe zu einem späteren Zeitpunkt eingeht; in diesem Fall wird der neue Bewegungsbefehl bearbeitet

Die Anforderungen für das Öffnen des Fensters sind analog hierzu aufgebaut und daher an dieser Stelle nicht aufgeführt. Gleiches gilt für die Beschreibung des Fensterhebers für die Beifahrertür.

Um diese Anforderungen in der KNS zu beschreiben, müssen zunächst die notwendigen Einträge in das Wörterbuch getätigt werden. Diese haben die folgende Form:

Signalname	Zustände	Ausgangszustände	Initial-zustand	Typ
driver door window position	[0..4]		0	Output
driver door window switch	open manually	idle	idle	Input
	open auto	idle		
	idle	open manually, open auto, close manually, close auto		
	close auto	idle		
	close manually	idle		

In den Anforderungen wird das Signal *driver door window position* verwendet, um die aktuelle Position der Fensterscheibe abzubilden. Hierbei entspricht der Wert 0 einem geschlossenen und der Wert 4 einem geöffneten Fenster. Mit Hilfe dieser Signale können nun die Anforderungen in der KNS beschrieben werden. Hierbei werden nur die Anforderungen für das Schließen des Fensters aufgeführt. Das Öffnen des Fensters kann analog hierzu beschrieben werden.

Condition before the trigger	battery voltage >= 10000.
Trigger	driver door window switch is close auto.
Condition after the trigger	driver door window position is not used in other reactions for at least 4 sec.
Reaction	at the latest after 4 sec driver door window position=0.

Tabelle 15: Automatisches Schließen der Fensterscheibe

Diese Anforderung beschreibt das automatische vollständige Schließen des Fensters, nachdem der Knopf des Fensterhebers in die Position *auto close* gebracht worden ist. Mittels der aufgeführten Bedingung nach dem Auslöser wird modelliert, dass eine später ausgelöste Funktion, die ebenfalls die Fensterscheibe bewegt, eine höhere Priorität besitzt. Das manuelle Schließen der Fensterscheibe wird durch die nachfolgende Anforderung beschrieben, wobei diese Anforderung ein Szenario beschreibt, in dem der Knopf zum Schließen des Fensters für 1 Sekunde gedrückt wird.

Condition before the trigger	`battery voltage >=10000.`
Trigger	`driver door window switch is close manually.`
Condition before the trigger	`driver door window switch is close manually for at least 1 sec` **`and`** `driver door window position is not used in other reactions for at least 1 sec.`
Trigger	`driver door window switch is idle.`
Reaction	`driver door window position=driver door window position - 1.`

Tabelle 16: Manuelles Schließen der Fensterscheibe

Die Position der Fensterscheibe wird in diesen Anforderungen abstrakt durch die Werte 0 bis 4 dargestellt. Die Abbildung auf die tatsächlichen Positionen der Fensterscheibe des Testobjekts muss manuell vom Testingineur im Testadapter kodiert werden.

Weiterhin geht die Abstraktion davon aus, dass die Scheibe innerhalb einer Sekunde um mindestens ¼ des gesamten möglichen Weges bewegt werden kann. Diese Annahme muss selbstverständlich dem tatsächlichen Verhalten des Testobjekts entsprechen oder die in den KNS-Anforderungen verwendeten Zeiten müssen dem Verhalten des Testobjekts angepasst werden.

5.2.1.4 Benutzermanagement

Wie bereits im vorherigen Abschnitt erwähnt, können die Sitzeigenschaften für den Fahrersitz mittels einem Benutzermanagement abgespeichert und wiederhergestellt werden. Diese Funktion ist im Anforderungsdokument folgendermaßen beschrieben:

> *Bewegung über Benutzermanagement:* Die Sitzverstellung über das Benutzermanagement ist nur möglich, solange die Fahrzeuggeschwindigkeit (FZG_V) kleiner als 5 km/h ist. Überschreitet die Fahrzeuggeschwindigkeit 5 km/h, so wird die Sitzbewegung sofort abgebrochen.
>
> Es sind zwei Fälle zu unterscheiden:
>
> Fall 1: (Auswahl einer Einstellung über Benutzermanagementtaste)
> In diesem Fall ist anzunehmen, daß der Fahrer bereits auf dem Fahrersitz sitzt. Um die Bewegung so angenehm wie möglich zu gestalten, sind folgende Regeln bei der Ansteuerung der Sitzposition zu beachten: Zuerst werden die Bewegungen durchgeführt, die eine Entspannung der Sitzposition zur Folge haben, d.h. das Vergrößern der Entfernung Sitz–Lenkrad, das Flacher–stellen des Lehnenwinkels, das Absenken der Sitzfläche (vorne und hinten) sowie das Öffnen der Schalung. Anschließend werden die entgegengesetzten Bewegungen durchgeführt. Es dürfen zu einer Zeit maximal zwei Richtungen gleichzeitig bewegt werden. Dabei gilt die Reihenfolge Entfernung

Sitz–Lenkrad, Lehnenwinkel, Schalung, Sitzfläche vorne, Sitzfläche hinten.

Ist die Batteriespannung BATT zu Beginn der Sitzverstellung kleiner als 10V, so werden die Sitze nicht bewegt. Statt dessen wird die Meldung B LOW SITZ = 1 generiert.

Verwendung der Tasten des Benutzermanagements (nur Fahrer–TSG): Betätigt der Fahrer eine der Tasten MGMT 1, MGMT 2, MGMT 3 oder MGMT 4, so wird die Einstellung, die unter dieser Speicherposition abgelegt ist, abgerufen. Dazu wird […] die Sitzeinstellung angestoßen, wobei als Soll–Werte die Werte genommen werden, die unter der Speicherposition abgelegt sind, die der gedrückten Taste entsprechen. […]

Abspeicherung mittels der Tasten des Benutzermanagements (nur Fahrer–TSG): Betätigt der Fahrer die Taste MGMT SET und gleichzeitig eine der Tasten MGMT 1, MGMT 2, MGMT 3 oder MGMT 4, so werden die aktuellen Einstellungen unter der zugehörigen Speicherposition abgelegt.

Um die beschriebene Funktionalität mit Hilfe der KNS zu beschreiben, müssen zunächst einmal die folgenden Einträge im Wörterbuch vorhanden sein. Abgebildet sind an dieser Stelle nur die Signale für eine Speicherposition sowie die Signale, die den Speicherplatz für zwei Sitzeigenschaften (Lehnenwinkel und Höhe der Sitzfläche vorne) modellieren. Die verbliebenen drei Speicherpositionen sowie die anderen Sitzeigenschaften werden analog zu den beschriebenen modelliert.

Signalname	Zustände	Ausgangszustände	Initialzustand	Typ
speed	[0..200]		0	Input
mgmt_1	pressed	released	released	Input
	released	pressed		
mgmt_1 driver door seat angle position	[-2..2]		0	Output
mgmt_1 driver door seat height front position	[-2..2]		0	Output
mgmt_set	pressed	released	released	Input
	released	pressed		
driver door	open	closed	closed	Input
	closed	open		

Die Funktionen des Benutzermanagements können durch die folgenden zwei KNS-ANforderungen beschrieben werden. Die Erste beschreibt das Abspeichern einer Position und die Zweite das Abrufen.

Trigger	mgmt_set is pressed **and** mgmt_1 is pressed.
Reaction	mgmt_1 driver door seat height front position=driver door seat height front position **and** mgmt_1 driver door seat angle position=driver door seat angle position.

Tabelle 17: Abspeichern der Sitzeigenschaften an Speicherplatz 1

Condition before the trigger	battery voltage >= 10000
Trigger	mgmt_1 is pressed.
Condition after the trigger	speed < 5 for at least 5 sec **and** driver door seat height front position is not used in other reactions for at least 5 sec **and** driver door seat angle position is not used in other reactions for at least 5 sec.
Reaction	at the latest after 5 sec driver door seat height front position=mgmt_1 driver door seat height front position **and** at the latest after 5 sec driver door seat angle position=mgmt_1 driver door seat angle position

Tabelle 18: Abrufen einer gespeicherten Sitzposition

Bei der Beschreibung des Einstellens einer abgespeicherten Sitzeinstellung wird das Zeitverhalten abstrahiert indem angenommen wird, dass spätestens nach 5 sec die einzustellende Sitzposition erreicht ist. Die Bedingungen an die Geschwindigkeit und die Modellierung der Prioritäten zwischen manuellem Einstellen und dem Benutzermanagement werden für die gesamten 5 Sekunden definiert. In der Realität kann dieser Zeitraum variieren, je nachdem wie weit sich Soll- und Ist-Position unterscheiden.

5.2.1.5 Weitere Funktionen

Das Anforderungsdokument beschreibt noch weitere Funktionen. Dazu gehört die Möglichkeit, die Positionen der Außenspiegel zu verändern. Die Funktion entspricht weitestgehend der Funktion der Sitzeinstellungen, sodass die Umsetzung in der KNS hier nicht erneut aufgeführt wird.

Weiterhin wird noch die Ansteuerung des Innenlichtes sowie der Ausstiegsleuchten der Türen durch das Türsteuergerät übernommen. Diese Funktionen dienten bereits als erklärende Beispiele im Konzeptkapitel und werden daher hier nicht wiederholt.

5.2.2 Evaluation der Testfallgenerierung

Um die Leistung des Testfallgenerierungsalgorithmus zu messen, sind aus den erfassten Anforderungen Positiv und Negativtests generiert worden. Bei der Suche sind die beschriebenen Heuristiken zum Einsatz gekommen und deren Leistung verglichen worden.

Die Tests sind aus zwei Versionen der KNS-Anforderungen erzeugt worden, die sich in der Anzahl der Türen unterscheiden. Ein Hinzufügen einer neuen Tür ist durch Einfügen von verlinkten Anforderungen geschehen. Die nachfolgende Tabelle zeigt, welche Komplexität die beiden Versionen der Anforderungen des Türsteuergerätes aufweisen.

# Türen	*# Signale*	*Größe des Zustandsraumes*	*# Aktionen*	*# bedingte Effekte*
1	57	2^{86}	55	163
2	92	2^{141}	81	253

Tabelle 19: Komplexität der Anforderungen des Türsteuergerätes bei unterschiedlicher Anzahl von Türen

Die Tabelle zeigt, wie viele Signale in den Anforderungen definiert sind, die Größe des Zustandsraumes (erreichbare und nicht erreichbare Zustände zusammen) sowie die Anzahl der Aktionen und bedingten Effekte, die aus den Anforderungen generiert worden sind.

5.2.2.1 Positivtests

Für beide Versionen der Anforderungen sind Positivtests generiert worden, wobei verschiedene Heuristiken während der Suche zum Einsatz gekommen sind. Tabelle 20 zeigt die Anzahl der erfolgreichen Suchen für verschiedene Heuristiken. Die Spalte „MC-Loc“ bezeichnet eine Suche, bei der die offenen Bedingungen gemäß der Heuristik „MC-Loc“ abgearbeitet worden sind. Die Spalte „LC-Loc“ bezeichnet dementsprechend eine Suche bei der hierfür die Heuristik „LC-Loc“ zum Einsatz gekommen ist. In beiden Fällen sind die Unterstützer gemäß der Heuristik h_{add} ausgewählt worden. Um die heuristisch geführte Suche mit einer uninformierten Suche zu vergleichen, ist weiterhin die Spalte „keine Heuristik“ aufgeführt.

Die Anzahl der gestarteten Suchen bei der Suche nach Positivtests lag bei 68. Da die Anforderungen der zweiten Tür als verlinkte Anforderungen eingefügt worden sind, wurden hierfür keine zusätzlichen Suchen benötigt, da die Testfälle für die Beifahrertür aus den Testfällen für die Fahrertür hergeleitet werden können.

Die maximale Suchtiefe ist auf 10 Aktionen beschränkt gewesen und die maximale Zeit pro Suche betrug 10 Minuten. Diese Werte gelten, soweit nicht anders angegeben, für alle nun folgenden Ergebnisse von Suchdurchläufen.

Die Tabelle macht verschiedene Aspekte deutlich. Erstens ist festzuhalten, dass die heuristisch geführten Suchen der uninformierten Suche überlegen sind. Darüber hinaus ist zu sehen, dass die Heuristik MC-Loc die besten Ergebnisse liefert, da die Suche in nur 2 Fällen fehlschlägt, während LC-Loc und „keine Heuristik“ in 8 bzw. 22 Fällen zu keinem Ergebnis führen.

	Anzahl erfolgreiche Suchen / benötigte Gesamtzeit inkl. Suchen mit Zeitüberschreitungen		
# Türen	MC-Loc	LC-Loc	keine Heuristik
1	66 / 22 Min	60 / 1 Std. 23Min.	46 / 3 Std. 40 Min.
2	66 / 22 Min.	60 / 1 Std. 23 Min.	46 / 3 Std. 41 Min.

Tabelle 20: Ergebnisse der Generierung der Positivtests des Türsteuergerätes

Ein weiterer interessanter Aspekt ist, dass eine Vielzahl der Suchen auch bei der nicht geführten Tiefensuche erfolgreich waren. Dies ist dadurch zu erklären, dass der POCL-Algorithmus eine Rückwärtssuche durchführt, der Suchbaum der Rückwärtssuche in vielen Fällen einen geringen Verzweigungsfaktor aufweist und auf vielen Pfaden im Suchbaum eine Lösung liegt. Der geringe Verzweigungsfaktor hängt damit zusammen, dass die Anforderungen häufig nicht mit vielen anderen Anforderungen und/oder Signalen interagieren. Ein Beispiel dafür sind die Anforderungen für die Beschreibung der Funktionalität des Türschlosses.

Das gleiche Argument kann als Erklärung für die Tatsache gegeben werden, dass ein Einfügen einer weiteren Tür keinen großen Einfluss auf die Leistung des Suchvorganges hat: Wenn ein Testfall für das Verschließen des Fahrerfensters gesucht wird, tragen Anforderungen, die die Beifahrertür betreffen, nicht zur Größe des Verzweigungsfaktors des Suchbaumes dieser Suche bei.

Tabelle 21 zeigt die Ergebnisse der Suchen nach Positivtests bei steigender Suchtiefe. Als Heuristik ist die MC-Loc Heuristik gewählt worden. Es ist zu sehen, dass die Suchzeit ansteigt und schließlich bei Suchtiefe 15 zwei weitere Suchen fehlschlagen. Das Fehlschlagen der Suchen ist jedoch auch dem verwendeten Suchverfahren (Hill-Climbing) geschuldet. Eine manuelle Untersuchung des Suchverlaufes bei den fehlgeschlagenen Suchen lässt den Schluss zu, dass eine A*-Suche in diesen Fällen erfolgreich wäre. In der prototypischen Implementierung ist jedoch nur Hill-Climbing als Suchverfahren umgesetzt.

Suchtiefe	***# erfolgreiche Suchen***	***Gesamtzeit***
10	66	22 Min.
11	66	24 Min.
12	66	27 Min.
13	66	31 Min.
14	66	38 Min.
15	64	40 Min.

Tabelle 21: Ergebnisse der Suchen bei steigender Suchtiefe (MC-Loc Heuristik)

5.2.2.2 Negativtests

Tabelle 22 zeigt die Ergebnisse der Suche nach Negativtests, wobei hierbei die Version der Anforderungen mit nur einer Tür zum Einsatz gekommen ist und die maximale Suchzeit auf

5 Minuten festgelegt wurde. Insgesamt sind 228 Suchen gestartet worden. Die Tabelle zeigt, wie viele der Suchen erfolgreich waren, wie viele der Suchen erfolgreich nicht erreichbare Ziele erkannt haben und wie viele Suchen aufgrund von Zeitüberschreitungen abgebrochen worden sind. Die Tabelle zeigt, dass auch im Falle der Suche nach Negativtests die heuristische Suche der uninformierten Suche überlegen ist.

Heuristik	***# erfolgreiche Suchen***	***# erkannte nicht erreichbare Ziele***	***# Zeitüberschreitungen***	***Gesamtzeit***
MC-Loc	192	6	30	2 Std 43 Min
keine	92	3	133	11 Std 14 Min

Tabelle 22: Ergebnisse der Generierung der Negativtests des Türsteuergerätes

5.3 Fallstudie KFZ-Alarmsystem

Eine weitere Fallstudie beruht auf einem Anforderungsdokument aus der Automobilindustrie, welches ein KFZ-Alarmsystem beschreibt. Da das Anforderungsdokument vertraulich ist, können an dieser Stelle keine Details der Anforderungen vorgestellt werden.

Auch für diese Fallstudie sind Positiv- und Negativtests erzeugt worden. Die Komplexität der zugrunde liegenden Anforderungen ist in Tabelle 23 aufgeführt.

# Signale	***Größe des Zustandsraumes***	***# Aktionen***	***# bedingte Effekte***
40	2^{47}	42	115

Tabelle 23: Komplexität der Anforderungen des KFZ-Alarmsystems

Die Tabellen 24 zeigt die Ergebnisse der Suchläufe für die Positivtests. Die Anzahl der gestarteten Suchen im Falle der Positivtests beträgt 86 Suchen.

Heuristik	***# erfolgreiche Suchen***	***# Zeitüberschreitungen***	***Gesamtzeit***
MC-Loc	86	0	0,2 Sek.
LC-Loc	86	0	0,2 Sek.
keine	86	0	0,2 Sek.

Tabelle 24: Ergebnisse der Generierung der Positivtests des KFZ-Alarmsystems

Die Tabelle zeigt, dass alle Verfahren die gesuchten Ziele innerhalb von Bruchteilen einer Sekunde auffinden konnten. Dies liegt darin begründet, dass auf allen verfolgten Pfaden eine Lösung lag: Keine Suche hat die gesetzte Tiefenschranke erreicht und musste auf Backtracking zurückgreifen.

Heuristik	*# erfolgreiche Suchen*	*# erkannte nicht erreichbare Ziele*	*# Zeitüber-schreitungen*	*Gesamtzeit*
MC-Loc	174	84	29	5 Std 11 Min
keine	141	25	121	20 Std 10 Min

Tabelle 25: Ergebnisse der Generierung der Negativtests des KFZ-Alarmsystems

Die Ergebnisse der Generierung von Negativtests sind in Tabelle 25 zu sehen. Hierbei wurden 287 Suchen gestartet. Hier ist klar zu sehen, dass die heuristische Suche wesentlich mehr Fälle von nicht erreichbaren Zielen erkannt hat. Dies schlägt sich auch in der Anzahl der Suchen, die nach einer Zeitüberschreitung abgebrochen worden sind, nieder. Die Gesamtzeit der Suchen nach nicht erreichbaren Zielen beträgt bei der uninformierten Suche ca. 20 Stunden, während die heuristische Suche knapp unter 5 Stunden damit verbracht hat.

5.4 Fallstudie Hausalarmsystem

Für beide bisher vorgestellte Fallstudien gilt, dass die Anforderungsdokumente zur Verfügung standen, aber kein Testobjekt. Somit konnte der gesamte Prozessm wie er zu Beginn des Kapitels 4 vorgestellt worden ist, nicht bis zum letzten Schritt durchgeführt werden. Diese Lücke wird durch die jetzt vorgestellte Fallstudie geschlossen.

Die Fallstudie befasst sich mit einem Alarmsystem für ein Wohnhaus. Die Studie basiert auf einem Simulink Modell, welches vom Hersteller The Mathworks als Demonstration für Simulink zur Verfügung gestellt wird. Die Demonstration besteht aus einer informellen Beschreibung des Alarmsystems sowie einem Simulink Modell, welches die beschriebene Funktionalität umsetzt.

Zum Zwecke der Evaluierung ist die informell beschriebene Funktionalität mit Hilfe der KNS beschrieben worden. Anschließend sind Positiv- und Negativtests generiert worden, die mit Hilfe des Testautomatisierungswerkzeug AutomationDesk [Lamberg07, AUD10] der Firma dSPACE ausgeführt worden sind.

Hierzu war es in einem ersten Schritt nötig, manuell den benötigten Testadapter zu erstellten, der die in den Testfällen beschriebenen abstrakten Testschritte durch AutomationDesk Befehle umsetzt. Anschließend wurden die durch den Testfallgenerator erzeugten XML Dateien automatisch mit Hilfe eines XSL Skriptes in eine von AutomationDesk lesbare Datei umgewandelt.

In weiteren Schritten sind die so entstandenen Testfälle ausgeführt worden und während der Testausführung ist die Modellüberdeckung durch das Werkzeug Simulink protokoliert worden. Als Überdeckungsmaß ist hierbei MC/DC gewählt worden.

Der gesamte Vorgang ist mehrmals durchgeführt worden, da in den ersten Durchläufen nicht alle der Tests ohne Fehler durchgelaufen sind. Dies ist darauf zurückzuführen, dass die informelle Beschreibung der Funktionalität nicht eindeutig ist und die erstellten KNS-Anforderungen aufgrund der Mehrdeutigkeiten nicht das Verhalten des Modells beschrieben haben. Nachdem diese Fehler in der Verhaltensbeschreibung entfernt worden sind, sind alle Positiv- und Negativtests erfolgreich durchgeführt worden.

Tabelle 26 zeigt Daten von zwei dieser Durchläufe. Hervorzuheben ist, dass der fehlerfreie Durchlauf eine vollständige MC/DC Überdeckung des Testobjektes erreicht hat.

# Anforderungen	*# Positivtests (# Fehler)*	*# Negativtests (# Fehler)*	*MC/DC Überdeckung des Testobjektes*
8	8 (0)	13 (3)	67 %
13	19 (0)	50 (0)	100 %

Tabelle 26: Ergebnisse verschiedener Testläufe

6 Zusammenfassung und Ausblick

In dieser Arbeit ist eine Methode vorgestellt worden, die es ermöglicht, aus Anforderungen, die in einer kontrollierten natürlichen Sprache erstellt worden sind, automatisch Testfälle abzuleiten. Dieses Vorgehen verringert eine Lücke im Bereich des modellbasierten Testens, da es ein Verfahren bereitstellt, welches den Prozess von den informellen in natürlicher Sprache verfassten Anforderungen, zu einem formalen Testmodell vereinfacht.

Hierfür ist eine kontrollierte natürliche Sprache entworfen worden, die es erlaubt Anforderungen an reaktive Echtzeitsysteme zu formulieren. Die Ausdrücke, die sich mit der KNS formulieren lassen, sind Ausdrücke der englischen Sprache und darüber hinaus orientiert sich die Sprache an bestehenden Anforderungsdokumenten aus der Praxis. Somit ist diese Sprache einem Großteil der an der Anforderungserfassung beteiligten Fachexperten leicht zugänglich.

Des weiteren ist eine Werkzeugunterstützung für die Verwendung der kontrollierten natürlichen Sprache entwickelt worden, die das Benutzen der Sprache für ungelernte Benutzer vereinfacht.

Die Auswahl der Testfälle orientiert sich ebenso wie die kontrollierte natürliche Sprache an der bisherigen industriellen Praxis. Aufgrund von bestehenden Testfallspezifikationen sind zwei verschiedene Testarten beschrieben worden, die automatisch aus den erfassten Anforderungen erzeugt werden können. Für die so erzeugten Testfälle ist gezeigt worden, dass diese eine MC/DC Überdeckung der in den Anforderungen niedergeschriebenen Entscheidungen und Bedingungen erreichen.

Weiterhin ist eine Methode zur Testfallgenerierung vorgestellt worden, die bisher im Bereich der Testfallgenerierung für Echtzeitsysteme noch nicht zum Einsatz gekommen ist. Die Verwendung von Formalismen aus dem Bereich der automatischen Handlungsplanung ermöglicht eine intuitive Übersetzung der Anforderungen in ein formales Modell mit Echtzeitbedingungen. Die Verwendung dieser bestehenden Formalismen hat die Möglichkeit eröffnet, auf bereits entwickelte Verfahren zur heuristischen Suche zurückzugreifen. Anhand von durchgeführten Fallstudien ist gezeigt worden, dass die heuristischen Verfahren in der Lage sind, die Testfallgenerierung für Modelle mit großen Suchräumen effizient zu gestalten. Darüber hinaus sind die erzeugten Testfälle in der Lage, Testobjekte mit nichtdeterministischem Zeitverhalten zu testen.

6.1 Ausblick

Ein Ziel dieser Arbeit war es, den entwickelten Prozess möglichst nah in bestehende industrielle Prozesse einzubetten, was mittels der kontrollierten natürlichen Sprache ermöglicht werden soll. Es sind Argumente genannt worden, die die Annahme untermauern, dass die Verwendung einer kontrollierten natürlichen Sprache dieses Ziel erreicht.

Abschließend kann diese Frage durch reine theoretische Betrachtungen jedoch nicht geklärt werden. Eine interessante Erweiterung dieser Arbeit stellt daher eine Benutzerstudie dar, die der Frage der Akzeptanz der KNS im industriellen Umfeld nachgeht.

Eine weitere interessante Frage, die in dieser Arbeit nicht behandelt worden ist, ist die Erzeugung von effizienten Testsuiten. Die Art und Weise, wie die Testfälle erzeugt werden, kann zu einer Menge von redundanten Testfällen führen. So enthält z. B. ein Testfall, der eine Anforderung A ausführt, um eine bestimmte Bedingung herzustellen, unter Umständen bereits einen Positivtest für Anforderung A. Solche Redundanzen in der Testsuite sollten erkannt und entfernt werden.

Literaturverzeichnis

[ADAC09] Webseite ADAC Pannenstatistik, http://www1.adac.de/Auto_Motorrad/pannenstatistik_maengelforum/Rueckrufe/rueckruf_popup.asp, Zuletzt gesichtet Oktober 2010.

[AF94] Allen, J. F. & Ferguson, G. (**1994**). *Actions and Events in Interval Temporal Logic*, Technischer Bericht, Nr. 521, The University of Rochester.

[Alur91] Alur, R. (**1991**). *Techniques for Automatic Verification of Real-Time Systems*, Doktorarbeit, Department of Computer Science, Stanford University.

[AOH03] Ammann, P.; Offutt, A. J. & Huang, H. (**2003**). Coverage Criteria for Logical Expressions. In: *Proceedings of the 14th International Symposium on Software Reliability Engineering* : 99-107, IEEE Computer Society.

[AZS+02] Andrews, A. A.; Zhu, C.; Scheetz, M.; Dahlman, E. & Howe, A. E. (**2002**). *AI Planner Assisted Test Generation*, Software Quality Journal 10 : 225-259.

[ASD10] ASD (**2010**). *ASD Simplified Technical English - ASD Specification STE100*, Technischer Bericht, ASD Simplified English Maintenance Group, Aerospace and Defence Industries Association of Europe.

[Attempto] Webseite Attempto Publikationen, http://attempto.ifi.unizh.ch/site/pubs/index.html, Zuletzt gesichtet September 2010.

[AUD10] AutomationDesk Webseite, http://dspace.de/de/gmb/home/products/sw/expsoft/automdesk.cfm, Zuletzt gesichtet Oktober 2010.

[ACK+02] Audemard, G.; Cimatti, A.; Kornilowicz, A. & Sebastiani, R. (**2002**). Bounded Model Checking for Timed Systems. In: *Formal Techniques for Networked and Distributed Sytems — FORTE 2002* : 243-259, Lecture Notes in Computer Science, Band 2529, Springer Berlin / Heidelberg.

[BMS07] Barrett, C.; de Moura, L. & Stump, A. (**2007**). *Design and results of the 2nd annual satisfiability modulo theories competition (SMT-COMP 2006)*, Formal Methods in System Design 31 : 221-239.

[BDL04] Behrmann, G.; David, A. & Larsen, K. G. (**2004**). A Tutorial on Uppaal. In: *Formal Methods for the Design of Real-Time Systems* : 33-35, Lecture Notes in Computer Science, Band 3185, Springer Berlin / Heidelberg.

[BFH+01] Behrmann, G.; Fehnker, A.; Hune, T.; Larsen, K.; Pettersson, P. & Romijn, J. (**2001**). Efficient Guiding Towards Cost-Optimality in UPPAAL. In: *Tools and Algorithms for the Construction and Analysis of Systems*, Springer Berlin / Heidelberg.

[BY03] Bengtsson, J. & Yi, W. (**2003**). Timed Automata: Semantics, Algorithms and Tools. In: *Lectures on Concurrency and Petri Nets* : 87-124, Lecture Notes in Computer Science, Band 3098, Springer.

[BGL+00] Bensalem, S.; Ganesh, V.; Lakhnech, Y.; Muñoz, C.; Owre, S.; Rueß, H.; Rushby, J.; Rusu, V.; Saidi, H.; Shankar, N.; Singerman, E. & Tiwari, A. (**2000**). An Overview of SAL. In: *LFM 2000: Fifth NASA Langley Formal Methods Workshop* : 187-196.

[BCC+03] Biere, A.; Cimatti, A.; Clarke, E.; Strichman, O. & Zhu, Y. (**2003**). Bounded Model Checking. In: *Advances in Computers*, Academic press.

[BCC+99] Biere, A.; Cimatti, A.; Clarke, E. M. & Zhu, Y. (**1999**). Symbolic Model Checking without BDDs. In: *Tools and Algorithms for the Construction and Analysis of Systems. Part of European Conferences on Theory and Practice of Software* : 193-207, Lecture Notes in Computer Science, Band 1579, Springer.

[BB06] Blackburn, P. & Bos, J. (**2006**). Working with Discourse Representation Theory. In: *The 18th European Summer School in Logic, Language and Information*.

[BG01] Bonet, B. & Geffner, H. (**2001**). *Planning as Heuristic Search*, Artificial Intelligence 129 : 5-33.

[BLG97] Bonet, B.; Loerincs, G. & Geffner, H. (**1997**). A Robust and Fast Action Selection Mechanism for Planning. In: *Proceedings of the 14th National Conference on Artificial Intelligence and 9th Innovative Applications of Artificial Intelligence Conference (AAAI-97/IAAI-97* : 714-719.

[BFM97] Braberman, V.; Felder, M. & Marré, M. (**1997**). Testing Timing Behavior of Real-Time Software. In: *International Software Quality Week*.

[BBK98] Broy, M.; von der Beeck, M. & Krüger, I. (**1998**). *SOFTBED: Problemanalyse für ein Großverbundprojekt "Systemtechnik Automobil-Software für eingebettete Systeme"*, Technischer Bericht, Technische Universität München, Institut für Informatik.

[Boeck06] Böck, H., (**2006**). *NetBeans Platform 6, Rich-Client-Entwicklung mit Java*. Galileo Press.

[CO00] Cardell-Oliver, R. (**2000**). *Conformance Tests for Real-Time Systems with Timed Automata Specifications*, Formal Asp. Comput 12 : 350-371.

[CO96] Cesta, A. & Oddi, A. (**1996**). Gaining Efficiency and Flexibility in the Simple Temporal Problem. In: *Proceedings of TIME-96*.

[CM94] Chilenski, J. J. & Miller, S. P. (**1994**). *Applicability of modified condition/decision coverage to software testing*, Software Engineering Journal 9 : 193-200.

[CGP99] Clarke, E. M.; Grumberg, O. & Peled, D. A., (**1999**). *Model Checking*. The MIT Press, Cambridge, Massachusetts.

[CLP04] Colin, S.; Legeard, B. & Peureux, F. (**2004**). *Preamble computation in automated test case generation using constraint logic programming*, The Journal of Software. Testing, Verification and Reliability 14 : 213-235.

[Collins03] Collins, M. (**2003**). *Head-Driven Statistical Models for Natural Language Parsing*, Computational Linguistics 29 : 589-637.

[CLR90] Cormen, T.; Leiserson, C. & Rivest, R., (**1990**). *Introduction to Algorithms*. MIT Press.

[CKM+07] Cushing, W.; Kambhampati, S.; Mausam & Weld, D. S. (**2007**). When is Temporal Planning Really Temporal?. In: *Proceedings of the 20th International Joint. Conference on Artificial Intelligence* : 1852-1859.

[Dechter03] Dechter, R., (**2003**). *Constraint Processing*. Morgan Kaufmann.

[DMP91] Dechter, R.; Meiri, I. & Pearl, J. (**1991**). *Temporal constraint networks*, Artificial Intelligence 49 : 61-95.

[Denger02] Denger, C. (**2002**). *High Quality Requirements Specifications through Natural Language Patterns*, Masterarbeit, Frauenhofer Insitute for Experimental Software Engineering, Kaiserslautern.

[DBK03] Denger, C.; Berry, D. M. & Kamsties, E. (**2003**). Higher Quality Requirements Specifications through Natural Language Patterns. In: *SWSTE '03: Proceedings of the IEEE International Conference on Software-Science, Technology & Engineering* : 80, IEEE Computer Society.

[DS04] Dutertre, B. & Sorea, M. (**2004**). *Timed Systems in SAL*, Technischer Bericht, SRI International.

[DAC99] Dwyer, M. B.; Avrunin, G. S. & Corbett, J. C. (**1999**). Patterns in Property Specifications for Finite-State Verification. In: *Proc. 21st International Conference on Software Engineering* : 411-420, IEEE Computer Society Press, ACM Press.

[EH04] Edelkamp, S. & Hoffmann, J. (**2004**). *PDDL2.2: The Language for the Classical Part of the 4th International Planning Competition*, Technischer Bericht, Nr. 195, Albert-Ludwigs-Universität Freiburg, Institut für Informatik.

[ELI] Webseite des Übersetzerwerkzeuges Eli, eli-project.sourceforge.net/, Zuletzt gesichtet April 2010.

[Ertel09] Ertel, W., (**2009**). *Grundkurs Künstliche Intelligenz*. Vieweg und Teubner Verlag / GWV Fachverlage GmbH, Wiesbaden.

[ES07] Esser, M. W. & Struss, P. (**2007**). Obtaining Models for Test Generation from Natural-language-like Functional Specifications. In: *18th International Workshop on Principles of Diagnosis* : 75-82.

[FN71] Fikes, R. & Nilsson, N. J. (**1971**). *STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving*, Artificial Intelligence-4, 1971 2 : 189-208.

[FM00] Flake, S. & Mueller, W. (**2000**). Structured English for Model Checking Specification. In: *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*.

[FL03] Fox, M. & Long, D. (**2003**). *PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains*, Journal of Artificial Intelligence Research 20 : 61-124.

[FL00] Fröhlich, P. & Link, J. (**2000**). Automated Test Case Generation from Dynamic Models. In: *ECOOP 2000 -- Object-Oriented Programming 14th European Conference* : 472-491, Lecture Notes in Computer Science, Band 1850.

[FKK07] Fuchs, N. E.; Kaljur, K. & Kuhn, T. (**2007**). *Discourse Representation Structures for ACE 5.5*, Technischer Bericht, Nr. ifi-2007.05, Department of Informatics, University of Zurich.

[FSS99] Fuchs, N. E.; Schwertel, U. & Schwitter, R. (**1999**). Attempto Controlled English - Not Just Another Logic Specification Language. In: *Logic-Based Program Synthesis and Transformation*, Lecture Notes in Computer Science, Springer.

[GL05] Gerevini, A. & Long, D. (**2005**). *Plan Constraints and Preferences in PDDL3*, Technischer Bericht, Nr. 2005-08-47, Dipartimento di Elettronica per l'-Automazione, Università degli Studi di Brescia.

[GNT04] Ghallab, M.; Nau, D. & Traverso, P., (**2004**). *Automated Planning: Theory and Practice*. Morgan Kaufmann.

[GRS03] Görz, G.; Rollinger, C. & Schneeberger, J., (**2003**). *Handbuch der künstlichen Intelligenz*. Oldenbourg Wissenschaftsverlag GmbH.

[HP85] Harel, D. & Pnueli, A. (**1985**). On the development of reactive systems. In: *Logics and Models of Concurrent Systems*, Springer-Verlag.

[HRV+03] Heimdahl, M. P. E.; Rayadurgam, S.; Visser, W.; Devaraj, G. & Gao, J. (**2003**). Auto-generating Test Sequences Using Model Checkers: A Case Study. In: *Proceedings of the Workshop on Formal Approaches to Testing of Software, FATES '03* : 42-59, Lecture Notes in Computer Science, Band 2931, Springer.

[Hessel07] Hessel, A. (**2007**). *Model-Based Test Case Generation for Real-Time Systems*, Doktorarbeit, Department of Information Technology, Uppsala University.

[HLN+03] Hessel, A.; Larsen, K. G.; Nielsen, B.; Pettersson, P. & Skou, A. (**2003**). Time-Optimal Real-Time Test Case Generation Using Uppaal. In: *Proceedings of the Workshop on Formal Approaches to Testing of Software, FATES '03* : 114-130, Lecture Notes in Computer Science, Band 2931, Springer.

[HP04] Hessel, A. & Pettersson, P. (**2004**). A test case generation algorithm for real-time systems. In: *Proceedings of the Fourth International Conference on Quality Software* : 268 - 273.

[HP07] Hessel, A. & Pettersson, P. (**2007**). *A Global Algorithm for Model-Based Test Suite Generation*, Electronic Notes in Theoretical Computer Science 190 : 47 - 59.

[HNT+99] Higashino, T.; Nakata, A.; Taniguchi, K. & Cavalli, A. R. (**1999**). Generating Test Cases for a Timed I/O Automaton Model. In: *IFIP TC6 12th International Workshop on Testing Communicating Systems* : 197-214, IFIP Conference Proceedings, Band 147, Kluwer.

[HN01] Hoffmann, J. & Nebel, B. (**2001**). *The FF Planning System: Fast Plan Generation Through Heuristic Search*, Journal of Artificial Intelligence Research 14 : 253-302.

[HP02] Houdek, F. & Paech, B. (**2002**). *Das Türsteuergerät – eine Beispielspezifikation*, Technischer Bericht, Nr. 002.02/D, Fraunhofer IESE.

[HJL03] Håkansson, J.; Jonsson, B. & Lundqvist, O. (**2003**). *Generating online test oracles from temporal logic specifications*, International Journal on Software Tools for Technology Transfer (STTT) 4 : 456-471.

[IPC08a] IPC-2008 PDDL Resources, http://ipc.informatik.uni-freiburg.de/PddlResources, Zuletzt gesichtet März 2010.

[IPC08b] Webseite der sechten International Planning Competition, http://ipc.informatik.uni-freiburg.de/HomePage, Zuletzt gesichtet März 2010.

[IPC06] Webseite der fünften International Planning Competition, http://www.tzi.de/~edelkamp/ipc-4/, Zuletzt gesichtet März 2010.

[IPC04] Webseite der vierten International Planning Competition, http://www.tzi.de/~edelkamp/ipc-4/, Zuletzt gesichtet März 2010.

[IPC02] Webseite der dritten International Planning Competition, http://planning.cis.strath.ac.uk/competition/, Zuletzt gesichtet März 2010.

[IPC00] Webseite der zweiten International Planning Competition, http://www.cs.toronto.edu/aips2000/, Zuletzt gesichtet März 2010.

[Johannis07] Johannisson, K. (**2007**). *Natural Language Specifications*. In: Bernhard Beckert, R. H. & Schmitt, P. H. (Ed.), *Verification of Object-Oriented Software. The KeY Approach*, Springer Verlag.

[JM00] Jurafsky, D. & Martin, J. H., (**2000**). *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice Hall, Englewood Cliffs, New Jersey.

[KS92] Kautz, H. A. & Selman, B. (**1992**). Planning as Satisfiability. In: *Proccedings of the 10th European Conference on Artificial Intelligenc* : 359-363.

[Khoumsi02] Khoumsi, A. (**2002**). A Method for Testing the Conformance of Real Time Systems. In: *Formal Techniques in Real-Time and Fault-Tolerant Systems*, Springer Berlin / Heidelberg.

[KJM04] Khoumsi, A.; Jéron, T. & Marchand, H. (**2004**). Test Cases Generation for Nondeterministic Real-Time Systems. In: *Formal Approaches to Software Testing*, Springer Berlin / Heidelberg.

[Kodagana04] Kodaganallur, V. (**2004**). *Incorporating Language Processing into Java Applications: A JavaCC Tutorial*, IEEE Software 21 : 70-77.

[KC05a] Konrad, S. & Cheng, B. H. C. (**2005**). Realtime Specification Patterns. In: *Proceedings of the 27th International Conferencec on Software Engineering (ICSE05)*.

[KC05] Konrad, S. & Cheng, B. H. C. (**2005**). Facilitating the Construction of Specification Pattern-based Properties. In: *Proceedings of the 13th IEEE International Conference on Requirements Engineering* : 329-338, IEEE Computer Society.

[Koymans90] Koymans, R. (**1990**). *Specifying real-time properties with metric temporal logic*, Real-Time Systems 2 : 255-299.

[KT04a] Krichen & Tripakis (**2004**). Real-Time Testing with Timed Automata Testers and Coverage Criteria. In: *FTRTFTS: Formal Techniques in Real-Time and Fault-Tolerant Systems: International Symposium Organized Jointly with the Working Group Provably Correct Systems -- ProCoS*, LNCS, Springer-Verlag.

[KT04] Krichen, M. & Tripakis, S. (**2004**). Black-Box Conformance Testing for Real-Time Systems. In: *Proceedings of the 11th International SPIN Workshop, Barcelona, Spain* : 109-126, Lecture Notes in Computer Science, Band 2989, Springer.

[Kuhn08] Kuhn, T. (**2008**). AceWiki: A Natural and Expressive Semantic Wiki. In: *Proceedings of Semantic Web User Interaction at CHI 2008: Exploring HCI Challenges*, CEUR Workshop Proceedings.

[KS08] Kuhn, T. & Schwitter, R. (**2008**). Writing Support for Controlled Natural Languages. In: *Proceedings of the Australasian Language Technology Association Workshop 2008* : 46-54.

[KHD+06] Kupferschmid, S.; Hoffmann, J.; Dierks, H. & Behrmann, G. (**2006**). Adapting an AI Planning Heuristic for Directed Model Checking. In: *Proceedings of the 13th International SPIN Workshop (SPIN 2006)*.

[KWN+08] Kupferschmid, S.; Wehrle, M.; Nebel, B. & Podelski, A. (**2008**). Faster Than Uppaal?. In: *Proceedings of the 20th International on Computer Aided Verification* : 552-555, Lecture Notes in Computer Science, Band 5123, Springer.

[Lamberg07] Lamberg, K. (**2007**). *Trends und Perspektiven beim automatisierten Steuergerätetest*, Automobil Elektronik 6.

[LTS10] Lamberg, K.; Thiessen, C. & Schnelte, M. (**2010**). *Method for testing a control apparatus and test device*, United States Patent Application Publication, US20100218046A1.

[LMN04] Larsen, K.; Mikucionis, M. & Nielsen, B. (**2004**). Online Testing of Real-time Systems Using sc Uppaal. In: *Formal Approaches to Testing of Software*, Lecture Notes in Computer Science.

[LMN+05] Larsen, K. G.; Mikucionis, M.; Nielsen, B. & Skou, A. (**2005**). Testing real-time embedded software using UPPAAL-TRON: an industrial case study. In: *the 5th ACM international conference on Embedded software* : 299 - 306, ACM Press New York, NY, USA.

[MMM95] Mandrioli, D.; Morasca, S. & Morzenti, A. (**1995**). *Generating Test Cases for Real-Time Systems from Logic Specifications*, ACM Transactions on Computer Systems 13 : 365-398.

[MMS93] Marcus, M. P.; Marcinkiewicz, M. A. & Santorini, B. (**1993**). *Building a large annotated corpus of English: the penn treebank*, Computational Linguistics 19 : 313 - 330.

[MSD+99] von Mayrhauser, A.; Scheetz, M.; Dahlman, E. & Howe, A. E. (**1999**). Trade-offs in planner representation for automated software testing. In: *IEEE Aerospace Conference*.

[MPS99] Memon, A. M.; Pollack, M. E. & Soffa, M. L. (**1999**). Using a Goal-driven Approach to Generate Test Cases for GUIs. In: *Proceedings of the 21s International Conference on Software Engineering* : 257-266, ACM Press.

[MRK+97] Moser, L. E.; Ramakrishna, Y. S.; Kutty, G.; Melliar-Smith, P. M. & Dillon, L. K. (**1997**). *A Graphical Environment for the Design of Concurrent Real-Time Systems*, ACM Transactions on Software Engineering and Methodology 6 : 31-79.

[MB08] de Moura, L. & Bjørner, N. (**2008**). Z3: An Efficient SMT Solver. In: *Tools and Algorithms for the Construction and Analysis (TACAS)* : 337-340, Lecture Notes in Computer Science, Band 4963, Springer-Verlag.

[Muehe09] Mühe, M. (**2009**). *Übersetzung von kontrolliert-natürlichsprachlichen Anforderungsspezifikationen in formale Modelle*, Bachelorarbeit, Universität Paderborn.

[NBCC10] Netbeans Code Completion Tutorial, http://platform.netbeans.org/tutorials/nbm-code-completion.html, Zuletzt gesichtet August 2010.

[NK01] Nguyen, X. & Kambhampati, S. (**2001**). Reviving Partial Order Planning. In: *Proceedings of the seventeenth International Conference on Artificial Intelligence (IJCAI-01)* : 459-466, Morgan Kaufmann Publishers, Inc..

[NS03] Nielsen, B. & Skou, A. (**2003**). *Automated test generation from timed automata*, STTT 5 : 59-77.

[NOT06] Nieuwenhuis, R.; Oliveras, A. & Tinelli, C. (**2006**). *Solving SAT and SAT Modulo Theories: From an abstract Davis--Putnam--Logemann--Loveland procedure to DPLL(T)*, Journal of the ACM 53 : 937-977.

[PMM00] Pietro, P. S.; Morzenti, A. & Morasca, S. (**2000**). *Generation of Execution Sequences for Modular Time Critical Systems*, IEEE Transactions on Software Engineering 26 : 128-149.

[Potter91] Potter, B., (**1991**). *An introduction to formal specification and Z*. Prentice Hall.

[Pretschn01] Pretschner, A. (**2001**). Classical Search Strategies for Test Case Generation with Constraint Logic Programming. In: *In Proc. Formal Approaches to Testing of Software* : 47-60, BRICS.

[Pretschn03] Pretschner, A. (**2003**). *Zum modellbasierten funktionalen Test reaktiver Systeme*, Doktorarbeit, Technische Universität München.

[RN95] Russell, S. & Norvig, P., (**1995**). *Artificial Intelligence: A Modern Approach*. Prentice Hall.

[Schneide02] Schneider, S., (**2002**). *The B-method. An introduction*. Palgrave.

[Schnelte09] Schnelte, M. (**2009**). Generating Test Cases for Timed Systems from Controlled Natural Language Specification. In: *Proceedings 3nd IEEE International Conference on Secure System Integration and Reliability Improvement (SSIRI'09)* : 348-353, IEEE Computer Society.

[SG10] Schnelte, M. & Güldali, B. (**2010**). Test Case Generation for Visual Contracts Using AI Planning. In: *INFORMATIK 2010, Beiträge der 40. Jahrestagung der Gesellschaft für Informatik e.V. (GI)* : 369 - 374, GI.

[STL10] Schnelte, M.; Thiessen, C. & Lamberg, K. (**2010**). *Verfahren zum Test eines Steuergerätes*, Europäische Patentanmeldung, EP 2221697A1.

[SG95] Schubert, L. K. & Gerevini, A. (**1995**). Accelerating partial order planners by improving plan and goal choices. In: *Proceedings of the 7th IEEE International Conference on Tools with Artificial Intelligence* : 442-450, IEEE Computer Society Press.

[Schwitter98] Schwitter, R. (**1998**). *Kontrolliertes Englisch für Anforderungsspezifikationen*, Doktorarbeit, Universität Zürich.

[Schwitter02] Schwitter, R. (**2002**). English as a formal specification language. In: *Proceedings of the 13th international Workshop on Database and Expert Systems Applications* : 228 - 232.

[Schwitter10] Schwitter, R. (**2010**). Controlled Natural Languages for Knowledge Representation. In: *Proceedings of COLING 2010*.

[SLH03] Schwitter, R.; Ljungberg, A. & Hood, D. (**2003**). ECOLE: A Look-ahead Editor for a Controlled Language. In: *Proceedings of EAMT-CLAW03, Joint Conference combining the 8th International Workshop of the European Association for Machine Translation and the 4th Controlled Language Application Workshop* : 15-17.

[Schoening00] Schöning, U., (**2000**). *Logik für Informatiker.* Spektrum Akademischer Verlag Heidelberg, Berlin.

[Smith03] Smith, D. E. (**2003**). *The Case for Durative Actions: A Commentary on PDDL2.1*, Journal of Artificial Intelligence Research 20 : 149-154.

[SAC03] Smith, R.; Avrunin, G. & Clarke, L. (**2003**). From Natural Language Requirements to Rigorous Property Specifications. In: *Workshop on Software Engineering for Embedded Systems (SEES 2003): From Requirements to Implementation* : 40-46.

[Sorea02] Sorea, M. (**2002**). *Bounded Model Checking for Timed Automata*, Electronic Notes in Theoretical Computer Science 68.

[SL03] Spillner, A. & Linz, T., (**2003**). *Basiswissen Softwaretests*. Dpunkt Verlag.

[SVD01] Springintveld; Vaandrager & D'Argenio (**2001**). *Testing Timed Automata*, TCS: Theoretical Computer Science 254.

[Tron10] Uppaal-TRON Webseite, http://www.cs.aau.dk/~marius/tron/index.html, Zuletzt gesichtet Oktober 2010.

[UML] UML Resource Page, http://www.uml.org/, Zuletzt gesichtet Juli 2010.

[Upp10] Webseite des Model-Checkers Uppaal, www.uppaal.com, Zuletzt gesichtet Oktober 2010.

[UL06] Utting, M. & Legeard, B., (**2006**). *Practical Model-Based Testing: A Tools Approach*. Morgan-Kaufmann.

[WKS07] Waite, W.; Kastens, U. & Sloane, A. M., (**2007**). *Generating Software from Specifications*. Jones and Bartlett Publishers, Inc., USA.

[Weld94] Weld, D. S. (**1994**). *An Introduction to Least Commitment Planning*, AI Magazine 15 : 27-61.

[WH97] Wojcik, R. H. & Hoard, J. E. (**1997**). *Controlled languages in industry*, Survey of the state of the art in human language technology : 238-239.

[Yices10] Webseite des SMT Solvers Yices, http://yices.csl.sri.com/, Zuletzt gesichtet Oktober 2010.

[YS03] Younes, H. L. S. & Simmons, R. G. (**2003**). *VHPOP: Versatile Heuristic Partial Order Planner*, J. Artif. Intell. Res. (JAIR) 20 : 405-430.

Anhang A

Reihung von Triggern. Abbildung auf die Basis-KNS

Auf den folgenden Seiten werden die Übersetzungsmuster von Reihungen von Triggern auf die Basis-KNS beschrieben, die in der Arbeit nicht beschrieben worden sind.

Kein Zeitpunkt

Der einfachste Fall ist die Angabe eines Triggers, der keine Zeitbedingung enthält. Ein Beispiel hierfür ist die folgende Anforderung.

Req1	
Condition before the trigger	
Trigger	`remote lock button is pressed.`
Trigger	`remote lock button is pressed.`
Condition after the Trigger	
Reaction	`double lock is active.`

Diese Anforderung drückt aus, dass wenn innerhalb eines belieben Zeitraumes der Knopf der Funkfernbedienung zweimal gedrückt worden ist, das Double-Lock Merkmal aktiviert wird. Der vorgestellten Idee folgend, wird diese Anforderung in zwei einzelne Anforderungen übersetzt:

Condition before the trigger	`part_Req1 is p0.`
Trigger	`remote lock button is pressed.`
Condition after the trigger	
Reaction	`part_Req1 is p1.`

Condition before the trigger	`part_Req1 is p1.`
Trigger	`remote lock button is pressed.`
Condition after the trigger	
Reaction	`double lock is active and part_Req1 is p0.`

Diese Übersetzung führt dazu, dass ein einmaliges Drücken des Knopfes der Funkfernbedienung die erste Anforderung auslöst, da im Initialzustand der Wert der Variable *part_Req1* auf *p0* gesetzt ist. Als Reaktion setzt diese Anforderung den Wert auf *p1*, sodass ein weiteres Drücken des Knopfes zur Ausführung der zweiten Anforderung führt, da nun deren Vorbedingung erfüllt ist.

Die Reaktion der zweiten Anforderung aktiviert das Double-Lock Merkmal und setzt darüber hinaus den Wert der Variable *part_Req1* zurück auf *p0*. Dies bewirkt, dass die Reihung der Trigger erneut durchlaufen werden kann.

Dieses Vorgehen soll nun allgemein als Übersetzungsmuster angegeben werden. Zu sehen ist in der folgenden Anforderungsschablone ein Ausschnitt einer Reihung von *n* Trigger, bei der Auslöser $trigger_m$ dem Auslöser $trigger_{m-1}$ folgt.

Condition before the trigger	$condBefore_{m-1}$
Trigger	$trigger_{m-1}$
Condition before the trigger	$condBefore_m$
Trigger	$trigger_m$

Wenn zwei Trigger aufeinanderfolgen und der zweite Trigger nicht mit einem Zeitpunkt versehen ist, so wird die Reaktion des zweiten Triggers ausgeführt, sobald der zweite Trigger das erste Mal mit erfüllten Bedingungen ausgeführt wird. In die Basis KNS kann dies folgendermaßen übersetzt werden.

Condition before the trigger	$condBefore_{m-1}$ `and` *part_reqName* `is` p_{m-1}.
Trigger	$trigger_{m-1}$
Reaction	*part_reqName* `is` p_m.

Condition before the trigger	$condBefore_m$ `and` *part_reqName* `is` p_m.
Trigger	$trigger_m$
Reaction	*nextPart*

Die Reaktion der zweiten Anforderung enthält den Ausdruck *nextPart*. Dieser Ausdruck wird zu unterschiedlichen Konstrukten übersetzt, wobei hier zwei Fälle unterschieden werden:

1. Für den Fall, dass $trigger_m$ der letzte Trigger in der Reihung ist, wird *nextPart* zu $reaction_m$ `and part_reqName is p0` übersetzt. Das heißt, die Reaktion wird ausgelöst und *part_ReqName* auf `p0` gesetzt, sodass die Reihung erneut durchlaufen werden kann.
2. Falls ein weiterer Trigger in der Reihung folgt, werden die Übersetzungsregeln für den Zeitpunkt angewandt, den der nachfolgende Trigger $trigger_{m+1}$ besitzt.

Der Ausdruck *nextPart* wird auch in den folgenden Übersetzungsmustern genauso verwendet.

At the earliest after

Eine weitere Möglichkeit bei der Reihung von Auslösern ist es eine Zeitdauer anzugeben, die mindestens zwischen zwei aufeinanderfolgenden Auslösern verstreichen muss. Dies kann mittels `at the earliest after` *time* notiert werden.

Als Beispiel hierfür wird die folgende Anforderung betrachtet.

Trigger	`remote lock button is pressed.`
Trigger	`at the earliest after 2 sec the remote lock button is pressed.`
Reaction	`double lock is active.`

Eine Frage, die sich bei näherer Betrachtung dieser Anforderung stellt, ist, welche Reaktion zu erwarten ist, falls der Knopf dreimal gedrückt wird, wobei das zweite Drücken vor dem Ablauf der zwei Sekunden erfolgt. In Abbildung 57 ist solch ein Szenario dargestellt: Zum Zeitpunkt t_1 wird der Knopf das erste Mal gedrückt. Noch vor dem Verstreichen von zwei Sekunden wird der Knopf erneut gedrückt (Zeitpunkt t_2) und schließlich erneut zu einem Zeitpunkt t_3, der weniger als zwei Sekunden hinter t_2, aber mehr als zwei Sekunden hinter t_1, liegt.

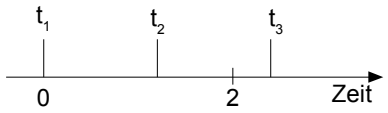


Abbildung 57: Mehrmaliges Drücken des Knopfes

Für solch eine Situation sind verschiedene Interpretationen der Bedeutung der Anforderung denkbar:

1. Das erneute Drücken des Knopfes t_2 wird ignoriert, da noch keine zwei Sekunden verstrichen sind. Zum Zeitpunkt t_3 wird das Merkmal Double Lock aktiviert.
2. Das erneute Drücken zum Zeitpunkt t_2 führt zu einem Rücksetzen der Zeitzählung, sodass zum Zeitpunkt t_3 noch keine zwei Sekunden verstrichen sind und somit das Double-Lock Merkmal nicht aktiviert wird.

In dieser Arbeit wird die erste Interpretation gewählt und dies wird folgendermaßen durch eine Abbildung auf die Basis-KNS dargestellt:

Condition before the trigger	`part_Req1 is p0.`
Trigger	`remote lock button is pressed.`
Reaction	`part_Req1 is pIgnore for 2 sec and afterwards p1.`

Condition before the trigger	`part_Req1 is p1.`
Trigger	`remote lock button is pressed.`
Reaction	`double lock is active and part_Req1 is p0.`

Nach dem ersten Drücken des Knopfes, wird *part_Req1* auf *pIgnore* gesetzt und erst nachdem zwei Sekunden verstrichen sind auf *p1*. Ein erneutes Drücken noch vor dem Verstreichen der zwei Sekunden löst daher weder die erste noch die zweite Anforderung aus, da für beide Anforderungen die Bedingung vor dem Auslöser nicht erfüllt ist, da *part_Req1* auf *pIgnore* gesetzt ist. Wird der Knopf jedoch zwei Sekunden nach dem ersten Drücken erneut

gedrückt, so ist *part_Req1* auf *p1* gesetzt, sodass die zweite Anforderung zur Ausführung gebracht wird und das Double-Lock Merkmal aktiviert wird.

Dieses Vorgehen soll nun allgemein als Übersetzungsmuster angegeben werden. Zu sehen ist in der folgenden Anforderungsschablone ein Ausschnitt einer Reihung von *n* Triggern, bei der Auslöser $trigger_m$ dem Auslöser $trigger_{m-1}$ folgt, wobei der zweite Trigger die Zeitbedingung `at the earliest after` *time* besitzt.

Condition before the trigger	$condBefore_{m-1}$
Trigger	$trigger_{m-1}$
Condition before the trigger	$condBefore_m$
Trigger	`at the earliest after` *time* $trigger_m$

Solch eine Reihung von Auslösern wird übersetzt zu:

Condition before the trigger	$condBefore_{m-1}$ `and` *part_reqName* `is` p_{m-1}.
Trigger	$trigger_{m-1}$
Reaction	*part_reqName* `is pIgnore for` *time* `and afterwards` p_m.

Condition before the trigger	$condBefore_m$ `and` *part_reqName* `is` p_m.
Trigger	$trigger_m$
Reaction	*nextPart*.

At the earliest after, but at the latest after

Wird in einer Reihung von Auslösern sowohl eine unterer als auch eine obere Zeitschranke angegeben, so ist das Übersetzungsmuster eine Kombination der Übersetzungsmuster für die obere und untere Zeitschranke. Dementsprechend wird die Anforderung

Trigger	`remote lock button is pressed.`
Trigger	`at the earliest after 2 sec, but at the latest after 5 sec the remote lock button is pressed.`
Reaction	`double lock is active.`

zu den folgenden zwei Anforderungen in der Basis-KNS übersetzt:

Condition before the trigger	`part_Req1 is p0.`
Trigger	`remote lock button is pressed.`
Reaction	`part_Req1 is pIgnore and 2 sec after the trigger part_Req1 is p1 for 3 sec and afterwards p0.`

Condition before the trigger	`part_Req1 is p1.`
Trigger	`remote lock button is pressed.`
Reaction	`double lock is active and part_Req1 is p0.`

Dieses Vorgehen soll nun allgemein als Übersetzungsmuster angegeben werden. Zu sehen ist in der folgenden Anforderungsschablone ein Ausschnitt einer Reihung von *n* Triggern, bei der Auslöser $trigger_m$ dem Auslöser $trigger_{m-1}$ folgt, wobei der zweite Trigger die Zeitbedingung `at the earliest after` $time_1$`, but at the latest after` $time_2$ besitzt.

Condition before the trigger	$condBefore_{m-1}$
Trigger	$trigger_{m-1}$
Condition before the trigger	$condBefore_m$
Trigger	`at the earliest after` $time_1$`, but at the latest after` $time_2$ $trigger_m$

Solch eine Reihung von Auslösern wird übersetzt zu:

Condition before the trigger	$condBefore_{m-1}$ `and` *part_reqName* `is` p_{m-1}.
Trigger	$trigger_{m-1}$
Reaction	*part_reqName* `is pIgnore and` $time_1$ `after the trigger part_ReqName is` p_m `for` $time_2$`-`$time_1$ `and afterwards p0.`

Condition before the trigger	$condBefore_m$ `and` *part_reqName* `is` p_n.
Trigger	$trigger_n$
Reaction	*nextPart*.

At exactly after

Ein exakter Zeitpunkt der Form `at exactly after` *time* wird genauso in die Basis-KNS übersetzt, wie ein Ausdruck `at the earliest after` *time*`, but at the latest after` *time*+ε, wobei ε so gewählt wird, dass es kleiner ist, als die kleinste messbare Zeiteinheit auf dem Testobjekt.