Audi
Dissertationsreihe

Sebastian Thiel

Petri-Netz basierte Verifikation von funktionalen Testfällen

Cuvillier Verlag Göttingen

Audi-Dissertationsreihe, Band 57

Petri-Netz basierte Verifikation von funktionalen Testfällen

Vom Promotionsauschuss des Fachbereichs 4: Informatik der Universität Koblenz-Landau zur Verleihung des akademischen Grades

Doktor der Naturwissenschaften (Dr. rer. nat.)

genehmigte Dissertation

von
Herrn Diplom Informatiker Sebastian Thiel
geboren am 14.06.1982 in Koblenz

Datum der wissenschaftlichen Aussprache: 25. Januar 2012

Vorsitzender des Promotionsausschusses:

Prof. Dr. Harald F.O. von Kortzfleisch

Berichterstatter:

Prof. Dr. Kurt Lautenbach (Universität Koblenz-Landau)

Prof. Dr. Reinhard German (Friedrich-Alexander-Universität Erlangen-Nürnberg)

Dr. Frank Derichsweiler (AUDI AG)

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliographie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.
1. Aufl. - Göttingen: Cuvillier, 2012
Zugl.: Koblenz-Landau, Univ., Diss., 2012

978-3-95404-056-8

Nonnenstieg 8, 37075 Göttingen
Telefon: 0551-54724-0
Telefax: 0551-54724-21
www.cuvillier.de

1. Auflage, 2012
Gedruckt auf säurefreiem Papier

978-3-95404-056-8

Kurzfassung

Moderne Fahrzeuge enthalten heute eine Vielzahl hochgradig vernetzter Fahrzeugfunktionen. Neben der Entwicklung ist die zunehmende Anzahl und Komplexität der Funktionen eine große Herausforderung für die Qualitätsabsicherung. Immer kürzere Entwicklungszyklen, ein hoher Kostendruck, eine zunehmende Parallelisierung der Fahrzeugprojekte und die Reduzierung von Prototypen erfordern einen effizienten Testprozess und den Einsatz von Simulationstechniken.

Für die Funktionsabsicherung auf Komponenten- und Systemebene werden im Volkswagen-Konzern Hardware-in-the-Loop-Prüfstände eingesetzt. Die Steuerung der Prüfstände erfolgt mit Hilfe des Testautomatisierungssystems EXAM. Die Testfälle werden dabei basierend auf einem UML-Dialekt grafisch modelliert und im gesamten Volkswagen-Konzern verwendet.

Die hohen Effizienzvorgaben bei der Funktionsabsicherung erfordern den parallelen Einsatz der Hardware-in-the-Loop-Prüfstände für mehrere Fahrzeugprojekte und eine größtmögliche Auslastung der zur Verfügung stehenden Prüfstandszeiten. Jede Störung bei der Testausführung, wie z.B. ein irrtümlicher Abbruch des Testfalls oder eine unnötige Wiederholung, führt zu einer nutzlosen Verwendung der ohnehin sehr limitierten und sehr teuren Prüfstände und hat eine direkte Auswirkung auf den gesamten Funktionsabsicherungsprozess.

Analog zur Komplexität der Fahrzeugfunktionen steigt die Komplexität der in EXAM modellierten Testfälle. Dies hat zur Folge, dass die Anzahl der syntaktischen und semantischen Fehler pro Testfall deutlich zugenommen hat. Bei der Ausführung fehlerhafter Testfälle sind die Testergebnisse jedoch unbrauchbar und die Testfälle müssen wiederholt werden. Aufgrund der Größe der Testfälle (mehrere zehntausend Testschritte) können die Fehler nicht mehr ausschließlich durch manuelle Reviews der Testfälle gefunden werden. Eine automatisierte Überprüfung der Testfälle auf syntaktische Fehler ist bereits heute in EXAM integriert. Semantische Fehler in den Testfällen können heute noch nicht automatisiert erkannt werden.

In dieser Arbeit entwickeln wir ein automatisiertes Verfahren zur Überprüfung der Testfälle auf semantische Fehler, die zu einem Problem oder einem Fehler bei der Testausführung führen können. Das Verfahren basiert auf der Transformation der Testfälle in Formeln einer Aktionslogik, dargestellt mit Hilfe (standardisierter) Petri-Netze. Wir klassifizieren die möglichen Fehler und definieren die einzuhaltenden Richtlinien mit Hilfe von Petri-Netzen. Die Richtlinien können im praktischen Einsatz der Methode von den Anwendern selbst definiert werden. Das von uns entwickelte Verifikationsverfahren, basierend auf Invariantenanalysen, ermöglicht die Überprüfung dieser Richtlinien. Mit Hilfe einer prototypischen Implementierung der vorgestellten Methode wird die praktische Anwendbarkeit und deren Auswirkungen im Testprozess bei der AUDI AG validiert. Der Ansatz liefert einen wichtigen Beitrag zur Kosteneinsparung und zur weiteren Effizienzsteigerung im Absicherungsprozess bei der AUDI AG.

Abstract

Nowadays modern vehicles contain a huge amount of high-grade networked vehicle functions. The quality assurance of the vehicle functions is one of the biggest challenges in the whole development process. Shortened development cycles, a growing cost pressure and a reduction of prototype vehicles place high demands on the safeguarding process. An efficient test process and simulation methods are required for handling these challenges.

The Volkswagen group uses Hardware-in-the-loop-simulation in the development and test process. The test-automation-system EXAM enables an efficient usage of the test stands. EXAM offers an graphical modeling language (based on an UML-dialect) to formally describe the test cases.

The high time- and cost pressure in the development process require an efficient usage ratio of the test stands. Therefore, the Hardware-in-the-loop-simulations are shared across multiple vehicle projects. Each failure in the test execution e.g. an incorrect termination of the test case or an unnecessary test case repetition leads to additional workload to the limited and expensive test facilities and has a direct negative effect to the whole test process (time and costs).

The rising complexity of the vehicle functions requires also more complex test cases in EXAM. We are confronted with a rising probability of syntactic or semantic errors in the EXAM test cases. The execution of syntactic or semantic erroneous test cases leads to unusable test results and requires a repeated test case execution. The test cases contain in average more than ten thousand test steps. This prohibits a manual review process to detect all errors in the test cases. Today an approach for the automatic identification of syntactic errors is integrated in EXAM. A similar approach for the verification of semantic errors is currently missing. In this thesis we present a new approach for the verification of semantic errors in the EXAM test cases and show the practical applicability.

We have developed an automatic verification method to find semantic errors in the EXAM test cases. The method is based on the transformation of the test cases into formula of a logic of actions. For the verification process we represent the formulas with the aid of Petri nets. The Petri nets allows to specify the causal dependencies between the operation of the EXAM function library. We classify the dependencies in nine categories and show for each an appropriate verification algorithm. Furthermore, we use invariant analysis methods in order to verify the defined specifications. We prove the usage of the method with a prototypical realisation and validate the method in the test process of the AUDI AG. With the aid of the method, we make an important contribution to the time and cost reduction in the test process.

Danksagung

Die vorliegende Arbeit ist während meiner Tätigkeit in der Abteilung *Hardware-in-the-Loop-Funktionserprobung* bei der AUDI AG in Ingolstadt in Verbindung mit der Forschungsgruppe *Informationssysteme, Datenbanken und Netztheorie* von Herrn Prof. Dr. Lautenbach an der Universität Koblenz-Landau entstanden.

Prof. Dr. Kurt Lautenbach gilt mein Dank für die hervorragende Betreuung und die sehr interessanten und sehr hilfreichen Diskussion in Koblenz. Mit vielen guten und anregenden Ideen hat er entscheidend zum Gelingen dieser Arbeit beigetragen.

Bei Herrn Prof. Dr. Reinhard German möchte ich mich sowohl für die Möglichkeit meine Arbeit in seiner Arbeitsgruppe vorzustellen als auch für das von ihm erstellte Gutachten bedanken.

Den beiden Projektleitern von EXAM, Gerhard Kiffe und Dr. Frank Derichsweiler möchte ich für die Betreuung meiner Arbeit, die vielen hilfreichen Ratschläge und für die vielen spannenden Diskussionen und konstruktiven Kritiken danken.

Ein besonderer Dank gilt Dirk Zitterell für die vielen hilfreichen Diskussionen und die freundschaftliche Unterstützung bei allen meinen Fragen und Problemen.

Ganz herzlich bedanken möchte ich mich bei Stefanie Wunderle und meiner Familie, die mir stets Mut zugesprochen und mich in meiner Arbeit unterstützt haben.

Ingolstadt, den 01.03.2012
Sebastian Thiel

Inhaltsverzeichnis

1 Einleitung

Die Entwicklung neuer Fahrzeugfunktionen in der Automobilindustrie hat sich in den letzten Jahren deutlich gewandelt. Durch die gestiegenen Anforderungen an die Fahrzeugfunktionen, getrieben durch wachsende Kundenansprüche, gesetzliche Vorgaben (z. B. Reduktion der CO_2-Emissionen oder Normen, wie die DIN EN 61508 [ISO09] und die entstehende ISO/DIS 26262 *Road vehicles - Functional safety* [LPP10] für sicherheitskritische Systeme) sowie Innovationen in der Unterhaltungselektronik ist die Anzahl der Fahrzeugfunktionen in den letzten Jahren zunehmend angestiegen. Die Herausforderungen bei der Elektrifizierung der Fahrzeuge wird diesen Trend in den nächsten Jahren weiter beschleunigen. Dabei sind schon heute die Funktionen eines Fahrzeugs neben dem Design und den klassischen Fahrzeugeigenschaften (wie z.B. dem Kraftstoffverbrauch, der Motorleistung, dem Kofferraumvolumen, der Beinfreiheit etc.) das wichtigste Merkmal zur Differenzierung gegenüber dem Wettbewerb. Hierbei ist die Realisierung der Funktionen durch Software zu dem Innovationstreiber in der Automobilindustrie geworden. Aufgrund dieser besonderen Bedeutung spielt die Qualität und die Zuverlässigkeit der Funktionen für die Automobilhersteller eine sehr wichtige Rolle.

Durch die steigende Anzahl der Fahrzeugfunktionen und die zunehmende Vernetzung der Funktionen werden neben der Entwicklung der Funktionen auch neue größere Herausforderung an die Absicherung der Funktionen gestellt. Wie auch in der Softwaretechnik ist die Absicherung der Funktionen in der Automobilindustrie ein integraler Bestandteil des Entwicklungsprozesses ([Sax08], [SZ10]). Die Funktionsabsicherung ist dabei durch einen starken Kostendruck, eine Verkürzung des Entwicklungsprozesses und die starke Zunahme von Derivaten beeinflusst.

Im Volkswagen-Konzern hat sich für die Funktionsabsicherung der Steuergerätefunktionen die Hardware-in-the-Loop-Simulation in Verbindung mit dem Testautomatisierungssystem EXAM[1] etabliert. EXAM erlaubt, basierend auf einem UML-Dialekt, den Testablauf zu beschreiben, die Testfälle im Anschluss automatisiert am Prüfstand auszuführen und die Ergebnisse zu bewerten.

Die Zunahme und die wachsende Komplexität der Fahrzeugfunktionen haben einen direkten Einfluss auf die Funktionsabsicherung. Bei der Funktionsabsicherung mit Hilfe von EXAM wird dies besonders deutlich an der Anzahl und der gestiegenen Komplexität der Testabläufe in EXAM. Mit einem Umfang von mehreren zehntausend einzelnen Testschritten pro Testfall und einer Vielzahl an Verzweigungen, Schleifen und parallelen Abläufen ist die Komplexität der Testfälle mit Softwareprogrammen vergleichbar. Für die Beherrschung der entstandenen Komplexität und für einen effizienten Testprozess spielt die Qualität der in EXAM modellierten Testfälle eine entscheidende Rolle. Aus diesem Grund sind zum einen die Einhaltung definierter Modellierungsrichtlinien bei

[1] EXAM steht für EXtended Automation Method, www.exam-ta.de, letzter Zugriff 14.02.2011

der Erstellung der Testfälle Voraussetzung für die gemeinschaftliche, markenübergreifende Entwicklung der Testfälle im Volkswagen-Konzern. Zum anderen kann es durch fehlerhafte Testfälle (semantische Modellierungsfehler) zu Ausführungsfehlern an den HIL-Prüfständen kommen, die zu einer Verzögerung des Testprozesses führen. Dies hat zur Folge, dass der gesamte Testablauf erneut ausgeführt werden muss, was bei einer Testausführungszeit von mehreren Stunden die ohnehin sehr limitierten und sehr teuren Prüfstandsressourcen und somit indirekt den gesamten Testprozess erheblich belastet. Aufgrund des hohen Zeit- und Kostendrucks ist es entscheidend, dass die Hardware-in-the-Loop-Prüfstände ständig ausgelastet sind und unnötige Testausführungen vermieden werden.

Die Hauptursache für die Ausführungsfehler am Prüfstand ist dabei eine Missachtung der kausalen Abhängigkeiten (Reihenfolge in der die Operationen während des Testablaufs ausgeführt werden müssen) zwischen den Operationen der Funktionsbibliothek in EXAM. Die kausalen Abhängigkeiten ergeben sich durch die Logik der Fahrzeugfunktionen oder der verwendeten Prüfstandshardware. Als Beispiel kann die ESP®-Funktion (Elektronisches Stabilitätsprogramm) nur überprüft werden, wenn das virtuelle Fahrzeug zuvor ein entsprechendes Fahrmanöver ausführt und somit das ESP-Steuergerät auf die Situation reagieren kann. Ebenso kann die Auswertung von Bus-Botschaften nur erfolgen, wenn diese während des Testablaufs auch aufgezeichnet wurden. Bisher können die kausalen Abhängigkeiten der Operationen in EXAM und die Modellierungsrichtlinien nicht formal spezifiziert werden und nicht vor der Ausführung der Testabläufe am Prüfstand überprüft werden. Fehler werden somit erst bei der Ausführung der Testabläufe am Prüfstand erkannt und führen daher häufig zu erheblichen Verzögerungen im Testprozess und somit indirekt zu höheren Entwicklungskosten.

Wir stellen in dieser Arbeit, basierend auf der von [Fid93], [Sim00] und [LS00] eingeführten Aktionslogik und Petri-Netzen [Pet62], eine Methode zur Überprüfung der Ausführungsreihenfolge (kausale Abhängigkeiten) der EXAM-Operationen in den Testabläufen und der Einhaltung der Modellierungsrichtlinien vor. Der Ansatz erlaubt, die kausalen Abhängigkeiten zwischen den Operationen in EXAM formal zu definieren und die Verletzungen der kausalen Abhängigkeiten ohne den Einsatz der HIL-Prüfstände zu erkennen. Dieser Ansatz erlaubt die Fehler in den Testabläufen vor der Ausführung der Testfälle am Prüfstand zu korrigieren. Ziel ist es, auf diese Art die Qualität der Testabläufe zu erhöhen, damit nicht notwendige Wiederholungen der Testabläufe an den Prüfständen reduziert werden können. Dies führt zu einer Entlastung der sehr teuren Prüfstände und somit auch indirekt zu einer Steigerung der Effizienz des gesamten Testprozesses. Durch die dadurch gewonnene Prüfstandszeit kann die Testtiefe bei der Funktionsabsicherung weiter gesteigert werden, da insgesamt mehr Testzeit zur Verfügung steht. Mit Hilfe einer prototypischen Implementierung der beschriebenen Methode demonstrieren wir die praktische Anwendbarkeit der Methode im Testprozess bei der AUDI AG.

Im Einzelnen gliedert sich die Arbeit wie folgt:

In **Kapitel 2** beschreiben wir die Herausforderungen bei der Entwicklung von Fahrzeugfunktionen in der Automobilindustrie. Neben der Betrachtung des Entwicklungsprozesses gehen wir dabei im Schwerpunkt auf die Absicherung der Funktionen mit Hilfe von Hardware-in-the-Loop-Prüfständen ein. Für den effizienten Betrieb der Prüfstände ist eine Automatisierung des Prüfablaufs notwendig. Dazu hat sich im Volkswagen-Konzern das Testautomatisierungssystem EXtented Automation Method (EXAM) etabliert, welches wir im Detail vorstellen.

Aufgrund der steigenden Anzahl an Fahrzeugfunktionen und einer verkürzten Entwicklungszeit ergeben sich neue Herausforderungen für die Funktionsabsicherung. Die Qualität der für die Überprüfung der Fahrzeugfunktionen verwendeten Testfälle in EXAM hat einen großen Einfluss auf den Absicherungsprozess. In **Kapitel 3** definieren wir, wie die Qualität der EXAM-Testfälle beschrieben werden kann. Im Fokus steht dabei die Betrachtung der Auswirkungen fehlerhafte Testfälle auf die Testausführung und den gesamten Entwicklungsprozess sowie die Auswirkungen auf die gemeinschaftliche Entwicklung der Testfälle im Team. Als Grundlage für die weiteren Betrachtungen in der Arbeit skizzieren wir am Ende des Kapitels eine Lösungsidee zur Überprüfung und Steigerung der Testfallqualität und konkretisieren die Zielsetzung der Arbeit.

Die für das Verständnis der weiteren Arbeit benötigten mathematischen Grundlagen haben wir in **Kapitel 4** zusammengefasst. Neben einer kurzen Definition der Aussagenlogik werden Petri-Netze und Lösungsverfahren für lineare Gleichungssysteme betrachtet. Eine Diskussion über die benötigten Lösungszeiten der Gleichungssysteme schließen das Kapitel ab.

Die in [Fid93], [Sim00] und [LS00] beschriebene Aktionslogik bildet die Grundlage für die Überprüfung der in EXAM modellierten Testabläufe. In **Kapitel 5** beschreiben wir neben der Syntax und Semantik der Aktionslogik die Darstellung der Module (Formeln der Aktionslogik) mit Hilfe von Petri-Netzen. Auf Basis der Petri-Netz-Darstellung stellen wir ein Beweisverfahren vor, welches es ermöglicht zu entscheiden, ob die mit einem Modul spezifizierten Bedingungen von einem weiteren Modul erfüllt werden. In diesem Zusammenhang führen wir die Begriffe *erfüllbar* und *vollständig* ein.

Ausgehend von der skizzierten Problemstellung in Kapitel 3 stellen wir in **Kapitel 6** das entwickelte Verfahren zur Verifikation der kausalen Abhängigkeiten innerhalb der Testabläufe vor. In diesem Kapitel diskutieren wir alle notwendigen Schritte für die Verifikation im Detail. Der Fokus liegt dabei auf den notwendigen Erweiterungen der verwendeten Aktionslogik, der Transformation der EXAM-Elemente in eine korrespondierende Petri-Netz-Darstellung und der Beschreibung von drei *Häufigkeitsattribute*n und drei *Voraussetzungsattributen* für die Einteilung der Spezifikationen in neun Kategorien.

In **Kapitel 7** demonstrieren wir die Anwendbarkeit der vorgestellten Methode anhand eines Anwendungsbeispiels. Dazu betrachten wir die Überprüfung eines vereinfachten EXAM-Testfalls auf die Einhaltung verschiedener zuvor definierter Spezifikationen.

In **Kapitel 8** zeigen wir, wie die Anwendbarkeit der Methode in der Praxis mit Hilfe eines Prototypen als Erweiterung des EXAM-Modellers validiert wurde. Aufgrund der Anzahl der Testschritte pro Testfall ist für den praktischen Einsatz der Methode eine Optimierung notwendig. Die Optimierung und deren Auswirkungen diskutieren wir anhand verschiedener Beispiele. Ebenso demonstriert das Kapitel, wie der Anwender in der Praxis durch die Erweiterung des EXAM-Modellers bei der Definition der Spezifikation, der Durchführung der Verifikation und bei der Fehlerbehebung unterstützt wird.

In **Kapitel 9** fassen wir die Ergebnisse der Arbeit zusammen und geben einen Ausblick auf mögliche Erweiterungen und weitere Anwendungsbereiche des Ansatzes.

2 Funktionsentwicklung in der Automobilindustrie

Der Schwerpunkt in der Entwicklung von Fahrzeugfunktionen hat sich in den letzten Jahren deutlich gewandelt. Heute stellen nicht mehr mechanische sondern elektrische/elektronische Komponenten den Schwerpunkt der Entwicklung dar. Hierbei hat sich die Realisierung der Funktionen mit Hilfe von Software als der Innovationstreiber in der Automobilindustrie entwickelt.

In diesem Kapitel stellen wir zuerst im Allgemeinen die Entwicklung der Fahrzeugfunktionen und das zugrunde liegende Vorgehensmodell dar. Im zweiten Teil des Kapitels wird die Absicherung der Funktionen beschrieben. Im dritten Teil des Kapitels ist die Integration und Absicherung von Steuergerätefunktionen mit Hilfe der Hardware-in-the-Loop-Simulation detaillierter beschrieben. Im letzten Abschnitt stellen wir die im Volkswagen-Konzern eingesetzte Testautomatisierung EXAM für die automatisierte Testausführung und Ansteuerung der HIL-Prüfstände im Detail vor.

2.1 Entwicklung von Fahrzeugfunktionen

Die Entwicklung der Fahrzeugfunktionen ist in den letzten Jahren geprägt durch den Einsatz elektrischer/elektronischer Systeme und der steigenden Anzahl der durch Software realisierten (Teil-)Funktionen. Durch die gestiegenen Anforderungen an die Fahrzeugfunktionen, getrieben durch wachsende Kundenansprüche, gesetzliche Vorgaben (z. B. Reduktion der CO_2-Emissionen oder Normen, wie die DIN EN 61508 [ISO09] und die entstehende ISO/DIS 26262 *Road vehicles - Functional safety* [LPP10] für sicherheitskritische Systeme) und Innovationen in der Unterhaltungselektronik, ist die Anzahl der Fahrzeugfunktionen in den letzten Jahren überproportional angestiegen.

Bevor wir den Entwicklungsprozess in der Automobilindustrie skizzieren, wollen wir mit Hilfe der Abbildung 2.1 (nach [SZ10]) die Entwicklung der Steuergerätefunktionen von 1970 bis zum Jahr 2010 betrachten.

Die Anzahl der Funktionen pro Fahrzeug ist in dem dargestellten Zeitraum deutlich angestiegen. Bis etwa zum Jahr 2005 hat die Anzahl der Steuergeräte pro Fahrzeug und die Anzahl der Funktionen pro Steuergerät in etwa im gleichen Verhältnis zugenommen. Ab dem Jahr 2005 hat sich der Anstieg der Anzahl der Steuergeräte pro Fahrzeug abgeflacht. Im Gegensatz dazu steigt die Anzahl der Funktionen pro Steuergerät stärker an. Diese Entwicklung ist unter anderem mit der

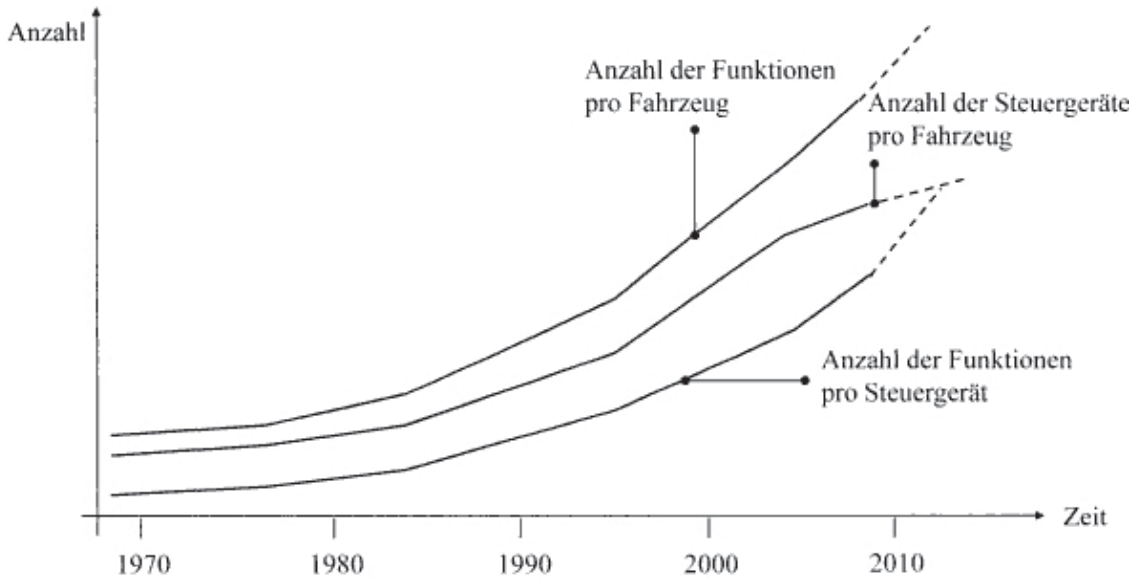

Abbildung 2.1: Entwicklung der Steuergerätefunktionen nach [SZ10]

Zunahme vernetzter Fahrzeugfunktionen zu begründen. Hierbei verteilt sich die eigentliche Funktion auf mehrere Steuergeräte, die über die Bussysteme im Fahrzeug vernetzt sind. Im Folgenden betrachten wir die vernetzten Fahrzeugfunktionen im Detail und geben ein Beispiel.

Erst durch den zunehmenden Einsatz von Bussystemen in den Fahrzeugen und der Vernetzung der Steuergeräte wurde die Realisierung von steuergeräteübergreifender Funktionen ermöglicht. In der Automobilindustrie werden für die Vernetzung der Steuergeräte CAN-, LIN-, MOST- und FlexRay-Bussysteme eingesetzt. Eine detaillierte Beschreibung der Bussysteme im Fahrzeug geben z.B. [Rei08] und [ZS08].

Als Beispiel für eine stark vernetzte Fahrzeugfunktion betrachten wir im Folgenden das Adaptive-Cruise-Control-System (ACC). Das ACC ist eine Weiterentwicklung der klassischen Geschwindigkeitsregelanlage (GRA). Zusätzlich zur Geschwindigkeitsregelung des Fahrzeug wird über einen Radarsensor der Abstand, die Relativgeschwindigkeit und die relative Position der vorausfahrenden Fahrzeuge ermittelt. Mit diesen Daten stellt das System durch einen aktiven Bremseingriff oder der Beschleunigung des Fahrzeugs einen konstanten Abstand zu den vorausfahrenden Fahrzeugen bis zum Erreichen der eingestellten Zielgeschwindigkeit sicher bzw. hält die maximal mögliche Geschwindigkeit unter Berücksichtigung des vom Fahrer definierten Sicherheitsabstandes.

In der Abbildung 2.2 sind schematisch die wichtigsten an der ACC-Funktion beteiligten Steuergeräte dargestellt. Durch die Daten des Radarsensors kann das ACC-Steuergerät die Entfernung und die relative Position der vorausfahrenden Fahrzeuge bestimmen. Auf Basis dieser Daten beeinflusst die Funktion über das Motorsteuergerät das Motormoment und über das Getriebesteuergerät das Getriebe (Übersetzungsverhältnis und Kraft-/Momentenverteilung). Über das ESP-Steuergerät wird bei Bedarf durch einen aktiven Bremseingriff das Fahrzeug verzögert. Diese Steuergeräte sind alle in der Abbildung 2.2 mit Hilfe des *Bus1* verbunden und können somit direkt Botschaften und Signale austauschen. Sind die an der Funktion beteiligten Steuergeräte auf

unterschiedliche Busse verteilt, werden die einzelnen Bussysteme über ein Gateway-Steuergerät verbunden, welches als Übersetzer verschiedener Bustechnologien und als Router für die Verteilung der Botschaften dient. So kann bei der ACC-Funktion mit Hilfe eines Displays in der Instrumententafel (Anzeigedisplay) dem Fahrer eine Rückmeldung über den aktuellen Zustand der ACC-Funktion gegeben werden, obwohl sich das zugehörige Steuergerät an einem anderen Bus (in der Abbildung 2.2 als *Bus2* bezeichnet) befindet.

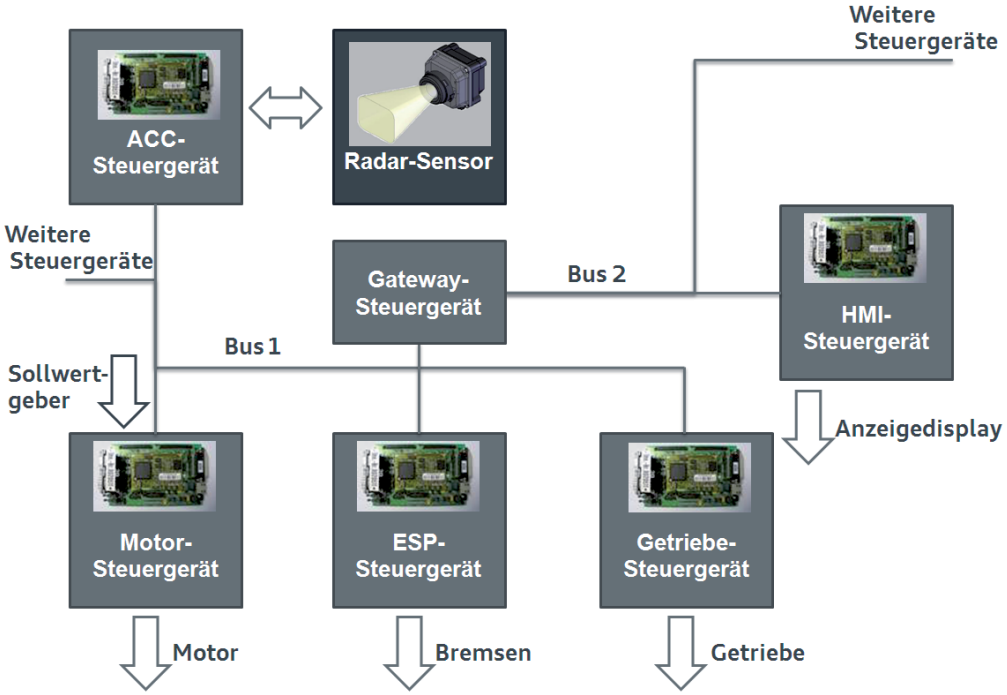

Abbildung 2.2: Schematische Darstellung der an der ACC-Funktion beteiligten Steuergeräte

Nach der exemplarischen Betrachtung einer stark vernetzten Fahrzeugfunktionen stellen wir im Folgenden die Entwicklung der vernetzten Funktionen in den letzten Jahren vor. Die Abbildung 2.3 nach [Sch08] stellt die Entwicklung der vernetzten Fahrzeugfunktionen am Beispiel ausgewählter Fahrzeugmodelle der AUDI AG von 1997 bis zum Jahr 2010 dar.

Die Funktionen sind dabei den Bereichen

- Licht,
- Fahrerassistenz,
- Infotainment,
- Fahrwerk,
- Komfort,
- Antrieb und
- Kombi

zugeordnet.

Der Anteil der vernetzten Fahrzeugfunktionen ist in allen Bereichen in den letzten Jahren massiv angestiegen. Ein deutlicher Anstieg ist in den Bereichen *Fahrerassistenz*, *Fahrwerk* und *Infotainment* zu beobachten. Mit der Entwicklung der Hybrid- und Elektrofahrzeugen ist in Zukunft mit einer noch stärkeren Zunahme der vernetzten Fahrzeugfunktionen zu rechnen.

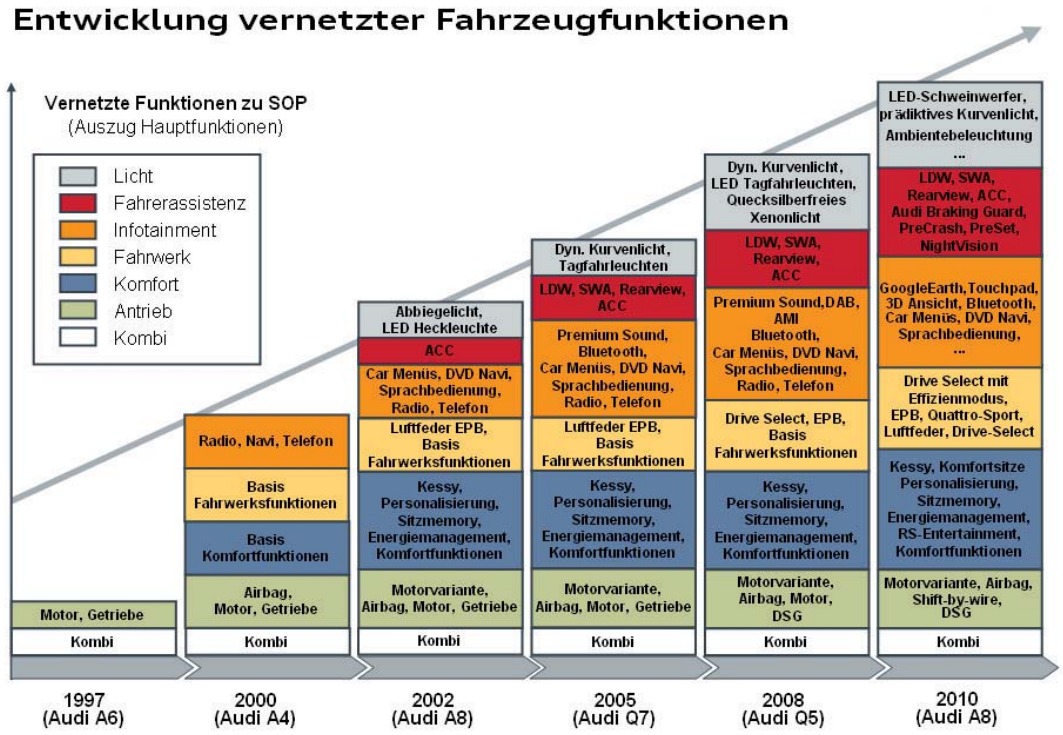

Abbildung 2.3: Entwicklung der vernetzten Fahrzeugfunktionen am Beispiel von ausgewählten Fahrzeugmodellen der Marke Audi nach [Sch08]

Im Gegensatz zu elektronischen Komponenten in der Unterhaltungselektronik sind die Steuergeräte im Fahrzeug extremen Bedingungen ausgesetzt. Die Verträglichkeit bzgl. wechselnder Temperaturbedingungen, Feuchtigkeiten und Erschütterungen sind offensichtliche Anforderungen. Ebenso ergeben sich spezielle Anforderungen an die elektromagnetische Verträglichkeit (EMV), wie zum Beispiel die Störfestigkeit gegen elektromagnetische Felder oder die Aussendung eigener Störsignale und elektromagnetischer Felder. Aufgrund der Lebensdauer der Fahrzeuge werden auch besondere Anforderungen an die Lebensdauer der Komponenten gestellt. Diese Anforderungen müssen sowohl in der Entwicklung berücksichtigt werden als auch in der Absicherung der Komponenten überprüft werden.

Als Vorgehensmodell wird in vielen Bereichen in der Automobilindustrie für die Entwicklung und Absicherung von Fahrzeugfunktionen das V-Modell ([vmo05], [Rei08]) eingesetzt. Wir konzentrieren uns im Folgenden auf die Entwicklung von Funktionen, die mit Hilfe von Software realisiert werden. Die wichtigsten Prozessschritte des V-Modells für die Entwicklung der Softwarefunktionen sind in der Abbildung 2.4 dargestellt und werden im weiteren Verlauf kurz beschrieben.

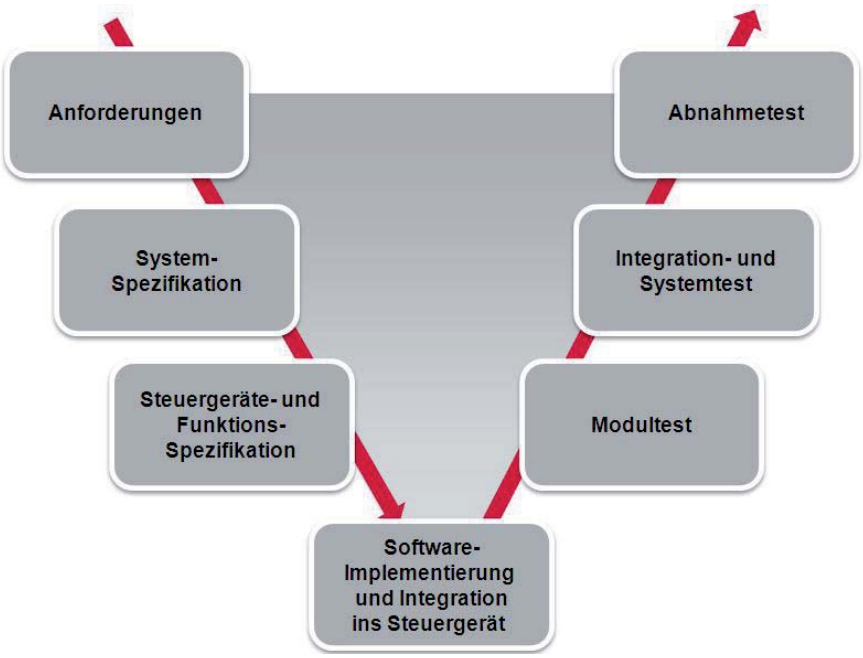

Abbildung 2.4: Entwicklung von Fahrzeugfunktionen nach dem V-Modell

Im ersten Prozessschritt werden die Anforderungen bzw. Eigenschaften (z. B. Art des Fahrzeugs, Kraftstoffverbrauch, Art des Motors etc.) an ein neues Fahrzeugprojekt definiert. Im darauf folgenden Schritt (Systemspezifikation) wird basierend auf den definierten Anforderungen das Fahrzeug entworfen (Design des Fahrzeugs, CAD-Modelle, Steuergerätearchitektur, Vernetzungspläne etc.) und aus den Eigenschaften die Fahrzeugfunktionen abgeleitet. Basierend auf diesen Ergebnissen werden die Funktionen und Steuergeräte im nächsten Prozessschritt (Steuergeräte- und Funktionsspezifikation) im Detail spezifiziert. Hierbei wird die Auslegung der Steuergerätehardware als auch die Verteilung der Softwarekomponenten auf die verschiedenen Steuergeräte festgelegt. Parallel zur Implementierung der Software erfolgt die Fertigung der Steuergerätehardware im folgenden Prozessschritt. Nach der Fertigstellung der Hard- und der Software werden die Softwarekomponenten in die Steuergeräte integriert.

Alle Prozessschritte auf der rechten Seite des V-Modells dienen der Qualitätssicherung. Bei dem Modultest werden die einzelnen Steuergeräte isoliert getestet. Im folgenden Prozessschritt (Integration- und Systemtest) werden die Steuergeräte und die Softwarefunktionen im Verbund abgesichert. Dazu werden die Steuergeräte wie im realen Fahrzeug mit Hilfe der Bussysteme vernetzt. Nach der Integration der Steuergeräte ins Fahrzeug kann im letzten Prozessschritt validiert werden, ob die spezifizierten Eigenschaften des Fahrzeugs erfüllt werden.

In der Regel wird für jedes Steuergerät, basierend auf der Steuergeräte- und Funktionsspezifikation, die Entwicklung der Steuergerätehardware, die Implementierung der Software, die Integration der Software ins Steuergerät und die Absicherung der Steuergerätesoftware auf Komponentenebene von einem einzigen Zulieferer übernommen. Durch die steigenden Bedeutung der Softwarefunktionen zur Differenzierung gegenüber dem Wettbewerb wird immer häufiger die Implementierung der Softwaremodulle und die Integration der Software ins Steuergerät direkt vom Automobilhersteller ausgeführt.

Aufgrund des Anstiegs der Komplexität der Funktionen hat sich der in der Softwaretechnik bereits weit verbreitete Ansatz der modellgetriebenen Softwareentwicklung (engl.: Model-Driven Software Development (MDSD) siehe z. B. [SVEH07]) und der Ansatz des modellbasierten Testen (MBT) ([RBGW10], [BK08b]) auch bei der Entwicklung von Steuergerätesoftware etabliert.

Basierend auf der System- und Komponentenspezifikation werden, z. B. mit Hilfe von der Software MATLAB®, Simulink® und Stateflow®[1] die Struktur und das Verhalten der Steuergerätesoftware modelliert. In frühen Phasen des Entwicklungsprozesses wird die Entwicklung der Funktionen und der Modelle durch Rapid-Prototyping unterstützt. Nach der Erstellung der Funktionsmodelle werden aus diesen automatisiert, z. B. mit Hilfe der Software TargetLink®[2], die Softwarekomponenten generiert. Diese werden im nächsten Schritt mit anderen Komponenten nach Vorgaben der Komponentenspezifikation ins Steuergerät integriert. Zur Verifikation (Back-To-Back-Test) der Codegenerierung werden diese Modelle anschließend in der Testphase als Referenz verwendet.

Bei der Entwicklung sicherheitskritischer Systeme (z. B. ESP®) müssen vorgegebene definierte Richtlinien im Entwicklungs- und Freigabeprozess in Abhängigkeit von der Kritikalität der Funktion berücksichtigt werden. In der ISO-Norm 26262 *Road vehicles - Functional safety* [LPP10] sind für die Einstufung der Kritikalität einer Funktion vier Kategorien unter der Bezeichnung Automotive Safety Integrity Level (ASIL) definiert. In Abhängigkeit des ASIL-Level sind in dieser Norm eine Vielzahl von Maßnahmen zur Sicherstellung der Qualität der Funktion gefordert. Eine wichtige Forderung ist z. B. eine detailliert dokumentierter Entwicklungsprozess zur Sicherstellung der Qualität des Entwicklungsprozesses.

In der Softwareentwicklung wird zur Bewertung von Entwicklungsprozessen für die Softwareerstellung der Standard SPICE (Software Process Improvement and Capability Determination) [aut10b] bzw. die ISO/IEC 15504 [ISO06] verwendet. Diese Standards orientieren sich dabei an der ISO-Norm ISO/IEC 12207 - *Prozesse im Software-Lebenszyklus* [ISO08b].

Für die Bewertung des Entwicklungsprozesses im Automobilbereich gibt es eine domänspezifische Variante des Standards unter dem Namen Automotive SPICE [aut10b]. Dieser Standard erlaubt es, durch Prüfungen (Assesments) den Entwicklungsprozess der Steuergerätesoftware zu bewerten. Dabei wird der Entwicklungsprozess in eine der fünf SPICE-Level eingestuft. Eine detaillierte Beschreibung und viele Beispiele für die praktische Anwendung gibt unter anderem [MHDZ07].

Neben einem einheitlichen Entwicklungsstandard spielt auch die Architektur der Softwaresysteme und die Wiederverwendung von bestehenden Komponenten eine wichtige Rolle bei der Entwicklung der Steuergerätesoftware. Für die Entwicklung standardisierter und wiederverwendbarer Steuergerätesoftware ist unter einer Initiative der deutschen Automobilhersteller in den letzten Jahren eine einheitliche Architektur unter dem Namen AUTOSAR (AUTomotive Open System ARchitecture, [aut10a]) entstanden. Hierbei wird eine logische Aufteilung der Steuergeräte-

[1] MathWorks® MATLAB®, Simulink® und Stateflow® sind Simulations- und Modellierungswerkzeuge der Firma MathWorks®, http://www.mathworks.de/

[2] Werkzeug für die Generierung von Steuergerätecode basierend auf Modellen in Simulink® und Stateflow® der Firma dSPACE®, http://www.dspace.de/de/gmb/home/products/sw/pcgs/targetli.cfm

software in steuergerätespezifische Basis-Software und steuergeräteunabhängige Anwendungs-Software vorgeschlagen. Die einzelnen Komponenten werden dabei über eine Middleware, der AUTOSAR-Laufzeitumgebung, verbunden. Einen detaillierten Einblick in das Thema gibt z. B. [KF09].

Für eine umfassendere Einführung in die Entwicklung der Fahrzeugfunktionen und die zugehörigen Richtlinien und Normen verweisen wir an dieser Stelle auf die entsprechende Fachliteratur zu diesem Thema. Eine Übersicht über die aktuellen Fahrzeugfunktionen und eine detaillierte Beschreibung der Entwicklung und Absicherung von Fahrzeugfunktionen geben z. B. [Rei08], [Sax08], [Bor10], [Bos07], [WHW09] oder [SZ04].

2.2 Funktionsabsicherung

Die Funktionsabsicherung ist ein integraler Bestandteil des Entwicklungsprozesses. Aus Sicht der Hersteller und der Zulieferer soll die Absicherung der Fahrzeugfunktionen möglichst kostengünstig, schnell, sicher (keine Gefahr für den Tester oder das zu testende Objekt), einfach und zuverlässig sein. Diesen Forderungen stehen die gestiegene Anzahl und die wachsende Komplexität der Funktionen sowie eine kürzere Entwicklungszeit und eine zunehmende Derivatisierung gegenüber.

Die Absicherung von elektrischen und mechanischen Eigenschaften wie Temperaturfestigkeit, mechanische Belastbarkeit, elektromagnetische Verträglichkeit (EMV) etc. erfolgt in Dauerlaufprüfständen, Klimakammern, EMV-Prüfständen und in Fahrzeugerprobungen. Für die Absicherung der Funktionen wird unter anderem neben den Fahrzeugerprobungen als Testumgebung die *Model-in-the-Loop-*, *Software-in-the-Loop-*, *Processor-in-the-Loop-* und *Hardware-in-the-Loop-Simulationen* im Volkswagen-Konzern verwendet.

Bei der Erprobung der Funktionen mit Hilfe einer *X-in-the-Loop-Simulation* ergeben sich einige Herausforderungen. Aufgrund der Verteilung der Funktionen auf verschiedene Steuergeräte ist ein isolierter Test eines einzigen Steuergerätes in den meisten Fällen nicht möglich. Für den Test der Funktionen müssen die Steuergeräte wie im realen Fahrzeug vernetzt sein. Ebenso spielt für den Test der Funktion der aktuelle Zustand des Fahrzeugs und der Fahrzeugumgebung eine entscheidende Rolle, da die Funktionen immer situativ reagieren.

Im realen Fahrzeug sind die Steuergeräte bzw. die Funktionen in einem Regelkreis integriert. Der Regler (Steuergerät) erfasst den aktuellen Zustand der Regelstrecke und den aktuellen Sollwert mit Hilfe verschiedener Sensoren. Weicht der aktuelle Zustand der Regelstrecke von den Sollwerten ab, berechnet das Steuergerät die entsprechenden Ausgangsgrößen und reguliert über verschiedene Aktoren die Regelstrecke. Der neue Zustand der Regelstrecke wird wieder über die Sensoren erfasst und mit dem aktuellen Sollwert verglichen.

Zur Darstellung eines Regelkreises werden in der Regelungstechnik Blockschaltbilder verwendet. In Abbildung 2.5 (Abbildung nach [SZ10]) ist das Regelungsmodell von Fahrzeugfunktionen mit Hilfe eines Blockschaltbildes dargestellt. Die Blöcke symbolisieren alle an der Regelung beteiligten Komponenten. Die Pfeile zwischen den Blöcken geben die Signalflüsse zwischen den Komponenten an.

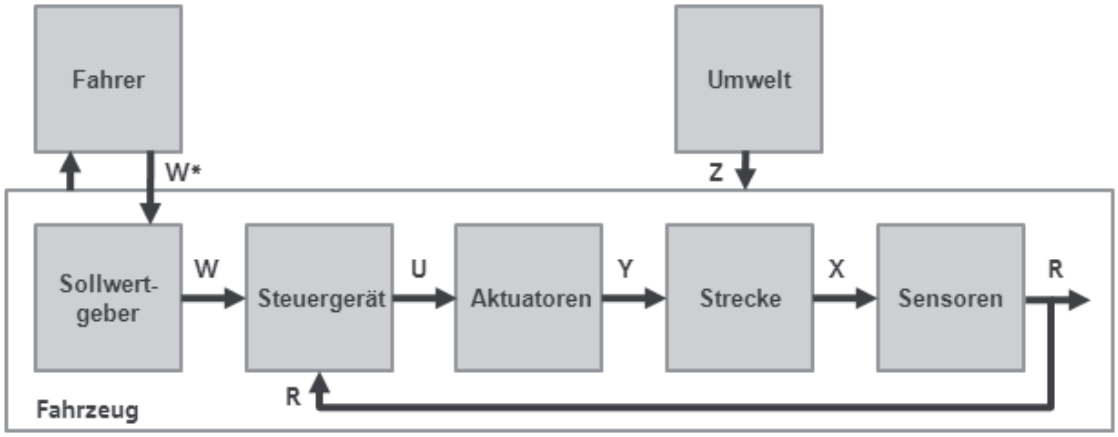

Abbildung 2.5: Reglungsmodell von Fahrzeugfunktionen als Blockschaltbild nach [SZ10]

Der Fahrer nimmt mit Hilfe von Sollwertgebern (Lenkrad, Pedale, Schalter etc.) Einfluss auf das Fahrzeug, indem er Sollwerte vorgibt (**W***). Diese Sollwerte werden an die beteiligten Steuergeräte weitergegeben (**W**) und mit den aktuellen Ist-Werten der Regelstrecke (**R**) verglichen. Die Erfassung der Ist-Werte ist über Sensoren realisiert. Basierend auf dem Vergleich der Ist- und Soll-Werte berechnet das Steuergerät die entsprechenden Ausgangsgrößen (**U**) zur Ansteuerung der Aktuatoren. Die Aktuatoren regulieren entsprechend der Vorgaben der Steuergeräte die Regelstrecke (**Y**). Zusätzlich zu dem Einfluss des Fahrers hat auch die Umwelt einen Einfluss auf das System (**Z**), welcher bei der Regelung berücksichtigt werden muss. Der neue Zustand der Regelstrecke wird über die Sensoren wieder erfasst und im Steuergerät mit den vorgegebenen Sollwerten verglichen. Die dadurch entstandene Rückkopplung ermöglicht erst die Regulierung der Regelstrecke.

Für die Absicherung der Fahrzeugfunktion ist entscheidend, dass die Funktion sich wie im realen Fahrzeug verhält. Daher ist es notwendig, die Umgebung der Funktionen entsprechend nachzubilden, damit der Regelkreis geschlossen werden kann. Für diesen Zweck werden die vier verschiedenen X-in-the-Loop-Simulationen eingesetzt, die wir im Folgenden beschreiben. Eine detaillierte Beschreibung der verschiedenen Methoden geben z. B. [SZ04] oder [Sax08].

1. **Model-in-the-Loop-Simulation (MIL-Simulation)**
   Die System- und Funktionsmodelle, die während des *Rapid-Prototyping* bzw. in der Phase der System- bzw. Komponentenspezifikation entstehen, können mit Hilfe der *Model-in-the-Loop-Simulation* überprüft werden. Die Modelle werden dazu in eine Simulationsumgebung auf einem PC integriert. Zusätzliche Modelle simulieren die Umgebung der Funktion, so dass alle benötigten Ein- und Ausgabesignale zur Verfügung stehen. In der Regel werden für die Modelle des Testrahmens existierende Modelle aus Vorgängerprojekten, einfache Simulationsmodelle oder auch konstante Testvektoren zur reinen Stimulation der Modelle eingesetzt.

2. **Software-in-the-Loop-Simulation (SIL-Simulation)**
   Bei der *Software-in-the-Loop-Simulation* wird der generierte Zielcode überprüft. Der Funktionscode wird jedoch dabei noch auf einem Entwicklungsrechner ausgeführt und nicht

auf dem eigentlichen Zielsystem (Steuergerät). Wie bei der *Model-in-the-Loop-Simulation* werden alle benötigten Eingangs- und Ausgangssignale durch Simulationsmodelle bereitgestellt. Die Reaktion des Steuergeräts wird durch die Simulationsmodelle verarbeitet und daraus werden entsprechend neue Eingangssignale für die Softwarekomponente erzeugt. Dadurch wird mit Hilfe der Simulationsmodelle der Regelkreis geschlossen. Hierbei sprechen wir dann von einer *Closed-Loop-Simulation*. Ebenso ist es möglich eine sogenannte *Open-Loop-Simulation* durchzuführen. Hierbei werden vorgegebene Testvektoren als Eingangssignale der Softwarekomponenten vorgegeben. Die Ausgangssignale der Komponenten werden dann zum einen mit den Ergebnissen der *Model-in-the-Loop-Simulation* und zum anderen mit der Komponenten- bzw. Funktionsspezifikation verglichen. Dieses Vorgehen ist vergleichbar mit z. B. JUnit-Tests von Java-Programmen.

3. **Processor-in-the-Loop-Simulation (PIL-Simulation)**
   Der nächste Schritt nach der *Software-in-the-Loop-Simulation* ist die *Processor-in-the-Loop-Simulation*. Hierbei werden die gleichen Tests wie bei der SIL-Simulation durchgeführt, jedoch wird hierbei die Software schon auf dem Zielprozessor zur Ausführung gebracht. Der Prozessor ist dabei in der Regel auf einem speziellen Experimentierboard eingebettet. Dadurch ergeben sich mehr Möglichkeiten zur Überwachung und Diagnose der Softwarefunktion als bei der späteren Integration ins eigentliche Steuergerät.

4. **Hardware-in-the-Loop-Simulation (HIL-Simulation)**
   Nach der Integration der Funktion ins Steuergerät erfolgt die Überprüfung mit Hilfe von der *Hardware-in-the-Loop-Simulation* (HIL-Simulation), die wir aufgrund ihrer Bedeutung für das Verständnis der Arbeit separat im nächsten Abschnitt ausführlich beschreiben.

2.3 Hardware-in-the-Loop-Simulation

Die Hardware-in-the-Loop-Simulation wird in der Automobilindustrie für die Integration, Inbetriebnahme und Absicherung von Funktionen auf Komponenten- und Systemebene eingesetzt. Nach der Beschreibung der Funktionsweise der Methode und der Darstellung der einzelnen Komponenten der Prüfstände gehen wir auf die Vor- und Nachteile der Methode ein. Für eine umfassendere Einführung verweisen wir auf [Sax08] oder [SZ10].

2.3.1 Funktionsweise der HIL-Simulation

Wie in den vorherigen Abschnitten beschrieben, sind die Funktionen im realen Fahrzeug in einen Regelkreis eingebunden. Zusätzlich können sich die Funktionen auf verschiedene Steuergeräte verteilen. Im Gegensatz zu der MIL-, SIL- und PIL-Simulation, muss nach der Integration der Software ins Steuergerät für die Kommunikation zwischen den einzelnen Softwarekomponenten (Steuergeräte) der reale Fahrzeugbus eingesetzt werden. Daher werden für die Inbetriebnahme der Fahrzeugfunktion oder für den Funktionstest mit der HIL-Simulation die Steuergeräte mit dem

jeweiligen Bussystemen verbunden und alle benötigten Sensoren und Aktoren zur Verfügung gestellt. Die Sensoren können dabei real am Prüfstand verbaut sein oder durch System- oder Verhaltensmodelle simuliert werden. Alle weiteren Komponenten, die in dem Regelkreis beteiligt sind, werden entweder durch echtzeitfähige Modelle simuliert oder als reale Systeme (Bauteile) zur Verfügung gestellt. An dieser Stelle ist zu erwähnen, dass für die HIL-Simulation alle eingesetzten Simulationsmodelle echtzeitfähig sein müssen, da sonst die Kommunikation mit den realen Bauteilen nicht funktioniert. Die Verbindung und die Synchronisation der einzelnen Bauteile und Modelle wird dabei vom HIL-Prüfstand übernommen.

Die Methode erlaubt, schon während der Entwicklung die Funktionen in einer virtuellen Fahrzeugumgebung in Betrieb zu nehmen und erste Tests durchzuführen. Durch die Möglichkeit, reale Steuergeräte und Funktionsmodelle zu koppeln, kann eine Funktion bereits untersucht werden, bevor alle benötigten Komponenten der Funktion als Echtteil zur Verfügung stehen. Als Beispiel könnten bei der in Abbildung 2.2 auf Seite 7 beschriebenen ACC-Funktion das Motorsteuergerät und das Getriebesteuergerät als Echtteil am Prüfstand verbaut werden und alle weiteren Steuergeräte und Komponenten (Motor, Fahrwerk, ACC-Sensor etc.) durch (echtzeitfähige) Funktionsmodelle ersetzt werden.

Die Funktionsweise eines HIL-Prüfstands ist in der Abbildung 2.6 schematisch dargestellt.

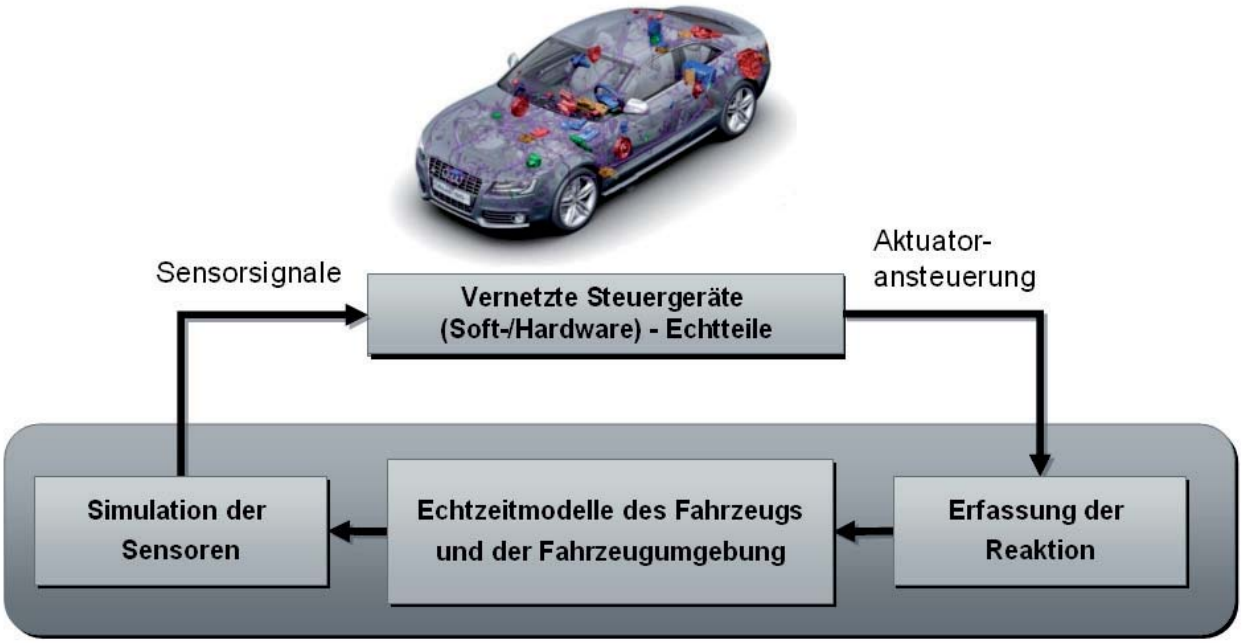

Abbildung 2.6: Prinzip der HIL-Prüfstände

Die für die Realisierung der zu überprüfenden Funktion benötigten Steuergeräte werden als Echtteile an dem Prüfstand verbaut und wie im realen Fahrzeug mit Hilfe der entsprechende Bussysteme vernetzt. Alle weiteren Komponenten des Fahrzeugs, welche nicht als Echtteile am Prüfstand verbaut sind, werden durch echtzeitfähige System- bzw. Verhaltensmodelle simuliert. In der Regel werden alle mechanischen Komponenten des Fahrzeugs (Motor, Getriebe, Lenkung etc.), die Sensoren und Aktoren, die Fahrzeugdynamik, die Fahrzeugumgebung und fehlende Steuergeräte mit Hilfe von echtzeitfähigen Simulationsmodellen simuliert. Die Ansteuerung der Aktuatoren

wird von der Echtzeitsimulation erfasst und verarbeitet. Die sich dadurch ergebenen Veränderungen des Fahrzeugzustands werden von den Sensormodellen erfasst und die entsprechenden Botschaften werden über den realen Bus wieder an die Steuergeräte gesendet. Diese reagieren wiederum auf die neuen Sensordaten und wirken erneut regulierend auf die Aktuatoren ein. Somit kann der reale Regelkreis wie im Fahrzeug in Echtzeit ausgeführt werden. Die Einhaltung einer harten Echtzeit ist hier erforderlich, da sonst aufgrund der Eigendiagnosse der Steuergeräte eine Überprüfung der Funktionen nicht möglich ist.

Zur Absicherung von Fahrzeugfunktionen, deren Sensordaten auf Beobachtung des Fahrzeugumfelds bzw. auf Navigationsdaten basieren, ist zusätzlich eine realitätsnahe Simulation der Fahrzeugumgebung mit anderen Verkehrsteilnehmern und die Integration realistischer Navigationsdaten notwendig. Bei der AUDI AG wird dazu das System *Virtual-Test-Drive* (VTD) [MSK+09] eingesetzt. Dieses System erlaubt eine realistische Simulation der Fahrzeugumgebung und erlaubt die Interaktion mit anderen simulierten Fahrzeugen. Mit dieser Erweiterung können auch Funktionen wie z. B. das prädiktive Kurvenlicht, Adaptive Cruise Control (ACC), PreCrash etc. mit den HIL-Prüfständen abgesichert werden.

Neben dem entwicklungsbegleitenden Testen ist die HIL-Simulation im Volkswagen-Konzern integraler Bestandteil der Funktionsfreigabe. Wie mit realen Versuchsfahrzeugen werden mit den Prüfständen Abnahmetests durchgeführt und entsprechende Freigabeempfehlungen für die getesteten Funktionen ausgesprochen.

Durch die steigende Derivatisierung der Fahrzeugprojekte und der Verkürzung der Entwicklungszeit werden immer mehr Fahrzeuge quasi-parallel entwickelt. Aufgrund des sich dadurch ergebenen Zeit- und Kostendrucks kann die Inbetriebnahme und Funktionsabsicherung nicht für alle Fahrzeugprojekte mit Prototypen oder Versuchsfahrzeugen durchgeführt werden. Bereits heute werden einige Derivate schon fast ausschließlich ohne Prototypen (prototypenfreie Entwicklung) entwickelt. Dadurch ist in den letzten Jahren die Bedeutung der HIL-Simulation als virtueller Prototyp stark gestiegen.

Neben der Reduktion von Prototypen ist die Erprobung einer Vielzahl von Funktionen heute mit realen Fahrzeugen sehr schwierig oder mit einem hohen Gefahrenpotential für das Fahrzeug, den Fahrer oder die Umgebung verbunden. Beispiele für solche Funktionen sind

- die automatische Notbremsung,
- PreCrash,
- ACC oder der
- Spurwechselassistent.

Damit zum Beispiel die angesprochene Funktion PreCrash, die kurz vor einem unausweichlichen Unfall den Gurtstraffer aktiviert, alle Fenster und das Schiebedach automatisch schließt, das Auslösen der Airbags vorbereitet und eine automatische Notbremsung einleitet, überhaupt getestet werden kann, muss die entsprechende Situation (ein unausweichlicher Unfall) dargestellt werden. Diese Situation führt jedoch bei jeder Erprobung mit einem realen Fahrzeug zu einer Zerstörung des sehr teuren Prototypen und zu einer Gefährdung des Testfahrers. Die HIL-Simulation ermöglicht es, die Funktionen ohne Gefahr für alle Beteiligten im Labor mit einer realistischen Aussage zu erproben.

2.3.2 Komponenten eines HIL-Prüfstands

Nachdem wir das Prinzip der HIL-Simulation betrachtet haben, wollen wir in den folgenden Abschnitten auf die einzelnen Komponenten der Prüfstände detaillierter eingehen. Ein Hardware-in-the-Loop-Prüfstand besteht aus vier Hauptkomponenten, die in der Abbildung 2.7 zusammengefasst dargestellt sind. Die vier Komponenten sind

- die zu testenden Steuergeräte inklusive aller Verkabelungen und eingesetzten Bussysteme,
- die Echtzeithardware für die Berechnung der Echtzeitmodelle,
- die Echtzeitmodelle und
- ein Testautomatisierungssystem.

Die zu *testenden Steuergeräte* bzw. die zu testende Funktionen werden in diesem Zusammenhang oft auch als *System Under Test* (SUT) bezeichnet. Neben den realen Steuergeräten sind für die Simulation der restlichen Fahrzeugkomponenten, der Fahrdynamik und der Fahrzeugumgebung *echtzeitfähige System- und Verhaltensmodelle* notwendig. Die Modelle sind in der Regel mit Hilfe von MATLAB® - Simulink® modelliert. Aus diesen Modellen wird echtzeitfähiger Programmcode generiert, der auf der *Echtzeithardware* zur Ausführung gebracht wird. An die Modelle und die Echtzeithardware bestehen hohe Anforderungen bezüglich dem Zeitverhalten des Simulationssystems, da für die Simulation das harte Echtzeitkriterium eingehalten werden muss. In der Regel wird mit einer Simulationsschrittweite von einer Millisekunde gerechnet. Bei besonders sicherheitskritischen Systemen, wie z. B. dem ESP® sind auch kürzere Simulationsschrittweiten notwendig. Die vierte Komponente der Prüfstände ist die *Testautomatisierung*, die eine automatisierte Testdurchführung ermöglicht. Der Betrieb der HIL-Prüfstände ist zwar auch ohne ein Testautomatisierungssystem möglich, jedoch können erst durch den Einsatz einer Testautomatisierung die Prüfstände effizient betrieben werden. Aus diesem Grund betrachten wir in dieser Arbeit auch die Testautomatisierung als eine Komponente der Prüfstände. Im Volkswagen-Konzern wird als Testautomatisierung die *EXtended Automation Method* (EXAM) eingesetzt, die wir im Kapitel 2.4 im Detail beschreiben.

Neben den vier Hauptkomponenten sind eine Vielzahl weiterer Komponenten an dem Prüfstand verbaut. Für die Darstellung von Fehlerszenarien, wie z. B. ein Kurzschluss, Leitungsunterbrechungen, falsche oder fehlende Lasten (Wiederstände), enthalten die Prüfstände eine Fehleraufschaltungseinheit (engl.: Fault Injection Unit, FIU). Für die Schnittstelle zwischen der realen Hardware und dem Simulationskern ist eine Signalkonditionierung für die Umwandlung zwischen den elektrischen Pegeln des Testobjekts und der Simulation erforderlich. Hierzu sind eine Reihe von A/D-Wandlern, digitalen Signalgeneratoren und entsprechende Schnittstellen zu den Bussystemen notwendig. Ebenso sind eine Reihe von externen Geräten z. B. für die Aufzeichnung der Buskommunikation (Datenlogger), für den Zugriff auf die Steuergerätediagnose oder Kameras für die Überwachung der Bedien- und Anzeigeinstrumente notwendig.

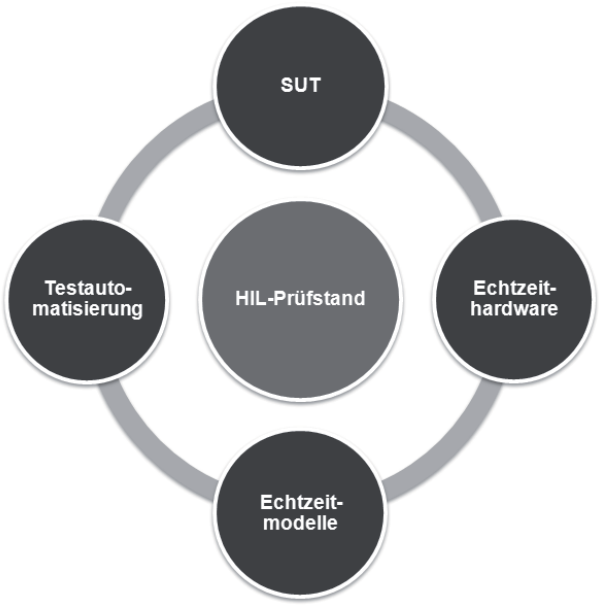

Abbildung 2.7: Bestandteile eines HIL-Prüfstands

2.3.3 Einordnung in den Entwicklungsprozess

Im Allgemeinen können die HIL-Prüfstände in Komponenten-HILs bzw. Einzel-HILs und vernetzte HILs bzw. System-HILs eingeteilt werden. Der Fokus bei Komponenten-HILs liegt auf dem Test eines einzelnen Steuergeräts. Die für den Betrieb des Steuergeräts erforderlichen Busse, Signale und Versorgungsspannungen werden vom HIL-Prüfstand bereitgestellt. Der HIL muss hierbei die Umgebung des Geräts nachbilden bzw. simulieren, damit das Steuergerät im plausiblen Zustand (keine Fehlerspeichereinträge im Steuergerät) getestet werden kann.

Aufgrund der verteilten Funktionsarchitektur, ist der Test eines Einzelsteuergeräts entsprechenden Einschränkungen unterworfen. Für die Inbetriebnahme und Absicherung vernetzter Funktion an einem Komponenten-HIL müssen an der Schnittstelle (z. B. CAN-Bus) entsprechende Simulationsmodelle betrieben werden, die das Verhalten der nicht vorhandenen Komponenten (weitere Steuergeräte, Sensoren, Aktoren, Fahrzeugumgebung etc.) nachbilden.

Der Fokus bei den System-HILs liegt zum einen auf der Integration der Steuergeräte zu einem Steuergeräteverbund und zum anderen auf der Absicherung von verteilten Steuergerätefunktionen. Dabei werden alle Steuergeräte bzw. Funktionen einer der vier Hauptdomänen

- Antrieb,
- Komfort,
- Infotainment und
- Fahrwerk

zugeordnet und entsprechend an dem Prüfstand verbaut und wie im realen Fahrzeug mit Hilfe der Bussysteme vernetzt. Vom Prüfstand werden dazu die erforderlichen Bussysteme und Versorgungsspannungen den Steuergeräten zur Verfügung gestellt. Alle weiteren an der Funktion

beteiligten Komponenten, wie Sensoren, Aktoren, die Fahrzeugumgebung und weitere Fahrzeugkomponenten werden vom Prüfstand mit Hilfe echtzeitfähiger Funktionsmodelle simuliert oder als reales Bauteil integriert.

Abbildung 2.8 stellt die Einordnung der Prüfstände in den Integrations- und Absicherungsprozess bei der AUDI AG dar. Wie bereits beschrieben, werden für die Absicherung eines einzelnen Steuergerätes Einzel-HIls beim Lieferanten oder in der jeweiligen Fachabteilung bei der AUDI AG eingesetzt. Der Schwerpunkt liegt hier auf der Testtiefe, also eine detaillierte Betrachtung der einzelnen Zustände des Steuergeräts. Bei zunehmender Integration der Steuergeräte im Steuergeräteverbund werden vernetzte HILs eingesetzt. Diese werden zum einen für die Integration der Steuergeräte und zum anderen für den Funktionstest eingesetzt. Hierbei liegt der Fokus auf der Testbreite, also eine möglichst umfassende Absicherung der Funktionen ohne detailliert auf die internen Zustände der Steuergeräte einzugehen. Nach der Funktionsabsicherung mit Hilfe der Hardware-in-the-Loop-Simulation erfolgt die Absicherung und der Intensivtest im Fahrzeug.

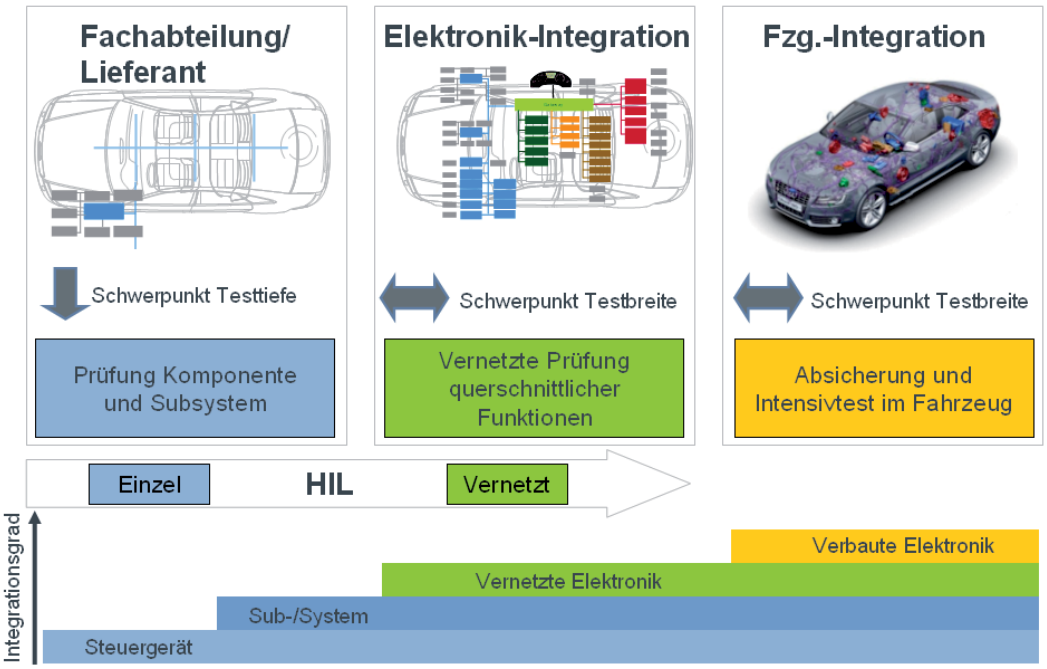

Abbildung 2.8: Einordnung der Hil-Prüfstände in den Testprozess

2.3.4 Vor- und Nachteile der HIL-Simulation

Im Folgenden fassen wir die Vor- und Nachteile der HIL-Simulation kurz zusammen.

Die wichtigsten **Vorteile** der HIL-Simulation sind

- die Inbetriebnahme und Integration der Steuergeräte im Labor ohne Prototypen-Fahrzeuge,
- die entwicklungsbegleitende Absicherung der Fahrzeugfunktionen,
- die Durchführung von Regressionstests unter gleichen (simulierten) Umwelteinflüssen,
- die gefahrlose Erprobung der Funktionen,

- die Kombination von realer Hardware und simulierten Komponenten,
- die automatisierte Testausführung und Auswertung durch Testautomatisierungssysteme und
- der Dauerbetrieb der Prüfstände 24 Stunden am Tag, 7 Tage die Woche und 365 Tage im Jahr.

Wie in den vorherigen Abschnitten beschrieben, ermöglicht die HIL-Simulation die vorzeitige Inbetriebnahme elektronischer Komponenten und die entwicklungsbegleitende Absicherung der Funktionen, ohne dass ein Prototypen-Fahrzeug existieren muss. Aufgrund der Durchführung der Testfälle im Labor, wird das wiederholte Testen aufeinander folgender Entwicklungsversionen unter gleichen Kriterien und Bedingungen (Regressionstests) unter gleichen (simulierten) Umwelteinflüssen möglich. Auf diese Weise ergeben sich reproduzierbare Ergebnisse, die bei einem Fehlverhalten der getesteten Funktion helfen, den Fehler schneller zu finden. Der größte Vorteil der Prüfstände ist die Möglichkeit den Testablauf und die Testauswertung mit einem Testautomatisierungssystem zu steuern. Damit wird der Betrieb der Prüfstände 24 Stunden am Tag, 7 Tage die Woche und 365 Tage im Jahr ermöglicht.

Als **Nachteile** der HIL-Simulation sind

- die vereinfachte Darstellung der Realität durch Simulationsmodelle,
- die Diskrepanz zwischen realem Komponenten- oder Fahrversuch und der Simulation,
- die Abhängigkeit der Testergebnisse von der Güte der Simulationsmodelle,
- die Kosten für die Anschaffung und den Betrieb der HIL-Prüfstände und
- den Aufwand zur Sicherstellung einer fehlerfreien Prüfumgebung

zu betrachten.

Wie auch bei den Vorteilen der HIL-Simulation erläutern wir im Folgenden die aufgezählten Nachteile. Die HIL-Simulation stellt durch den Einsatz der Simulationsmodelle nur eine vereinfachte Realität dar. Dabei wird es immer eine Diskrepanz zwischen realen Versuchen und der Simulation geben. Die Ergebnisse sind daher abhängig von der Güte der Simulationsmodelle. Erst durch qualitativ hochwertige und realitätsnahe Simulationsmodelle können verlässige Testergebnisse mit Hilfe der Prüfstände erreicht werden. Fehler oder eine nicht ausreichende Genauigkeit in den Simulationsmodellen führt dazu, dass die Steuergeräte und die Funktionen sich anders verhalten als im realen Fahrzeug und somit bei der Erprobung falsche Rückschlüsse auf die Qualität der Fahrzeugfunktionen möglich sind.

Neben der Güte der Simulationsmodelle sind die Kosten der Prüfstände als Nachteil der HIL-Simulation anzugeben. Trotz des möglichen Dauerbetriebs ist der HIL-Prüfstand eine sehr teure Ressource. Zum einen sind die Anschaffungskosten der Prüfstandshardware und zum anderen die notwendigen Maßnahmen bis zur Testbereitschaft sehr teuer. Bis zur Testbereitschaft müssen alle vier Komponenten des Prüfstands (SUT, Echtzeithardware, Simulationsmodelle und Testautomatisierung bzw. Testabläufe, vergleiche Abbildung 2.7 auf Seite 17) entsprechend aufgebaut beziehungsweise erstellt werden. Die Steuergeräte und die weiteren realen Komponenten sind wie im realen Fahrzeug zu vernetzen. Die Modelle für die fehlenden Komponenten müssen entwickelt, validiert und integriert werden. Ebenso müssen die Echtzeithardware und alle weiteren

Geräte (Datenlogger, Diagnosetester, Kamera etc.) entsprechend aufgebaut und konfiguriert werden. Das Testautomatisierungssystem muss an den Prüfstand angebunden werden und die Testabläufe für die zu überprüfenden Anforderungen müssen erstellt werden.

In Summe wird für den initialen Aufbau eines System-Prüfstands mit allen Komponenten typischerweise ein Betrag von über einer Million Euro benötigt. Neben den initialen Anschaffungskosten muss der laufende Betrieb des Prüfstands sichergestellt werden, d.h. während der Entwicklung sind verschiedene Versionen und Kombinationen der Steuergerätesoftware an einem Prüfstand notwendig. Die Test- und Funktionsmodelle sind zu adaptieren und müssen entsprechend neu validiert werden. Weiterhin erfolgt auch parallel zum Betrieb der Prüfstände die Entwicklung der Testfälle. Mit der Betreuung eines System-Prüfstands, für die Erstellung aller benötigten Modelle und Testspezifikationen sowie für die Testausführung und Ergebnisauswertung sind pro Domäne in der Regel in Summe mehr als zehn Personen in Vollzeit beschäftigt.

2.4 Testautomatisierungssystem EXAM

Für den funktionalen Steuergerätetest an HIL-Prüfständen wird im Volkswagen-Konzern als Testautomatisierung die *EXtended Automation Method* (EXAM) eingesetzt. EXAM basiert auf einem modellgetriebenen Testentwicklungsansatz (engl.: Model Driven Test Development) [Pre03] bzw. einem modellbasierten Testansatz (MBT) [RBGW10] und beschreibt die Modellierung, die Ausführung und die Auswertung der Testabläufe. Die Modellierung der Testabläufe erfolgt dabei grafisch auf Basis eines UML-Dialekts.

Im folgenden Abschnitt beschreiben wir neben dem Einsatz der Methode, den EXAM-Prozess, die Modellierung der Testabläufe und die Ansteuerung der Prüfstände mit EXAM. Einen detaillierten Überblick über EXAM geben [Thi09], [SD04], [TZ08] und [ZT10].

2.4.1 Testmethode und Testprozess

EXAM beschreibt nicht nur die Modellierung der Testinhalte, sondern definiert auch den zugehörigen Testprozess. Der Prozess umfasst alle Prozessschritte beginnend bei der textuellen Beschreibung der Testabläufe im Anforderungsmanagement, der grafischen prüfstandsunabhängigen Modellierung der Testabläufe in EXAM, der Generierung ausführbarer Testskripte und deren Ausführung sowie die Bewertung der Testergebnisse.

In Anlehnung an etablierte Testprozesse in der Softwareentwicklung wie z. B. das ISTQB (International Software Testing Qualifications Board) [SL10] oder dem TMap (Test management Approach) [KABV07] gliedert sich der von EXAM definierte Testprozess in die Bereiche *Testspezifizierung*, *Testvorbereitung*, *Testausführung* und *Testergebnisreview*. Übergreifend zu diesen Prozessschritten wird der gesamte Testprozess von dem *Testausführungsmanagement* gesteuert und überwacht.

Abbildung 2.9 stellt den Zusammenhang zwischen den einzelnen Phasen und den benötigten Eingabe- und Ausgabeartefakten dar.

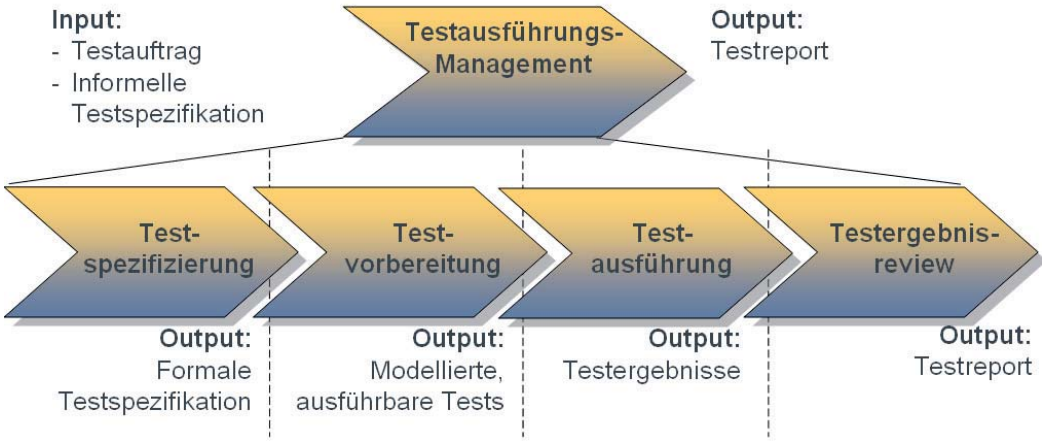

Abbildung 2.9: Prozessschritte von EXAM

Neben der Einteilung des Testprozesses in die einzelnen Phasen wird eine klare Rollentrennung zur Durchführung der verschiedenen Aufgaben beschrieben. Die klare Rollentrennung in EXAM erlaubt den Anwendern sich auf ihre spezifischen Aufgaben zu konzentrieren. In EXAM werden die vier Rollen

- Testspezifikateur,
- Testdesigner,
- Testanwender und
- Testreviewer

definiert. Die Zuordnung zu den Prozessschritten ist in Abbildung 2.10 in Form eines kombinierten Prozess- und Rollenmodells dargestellt.

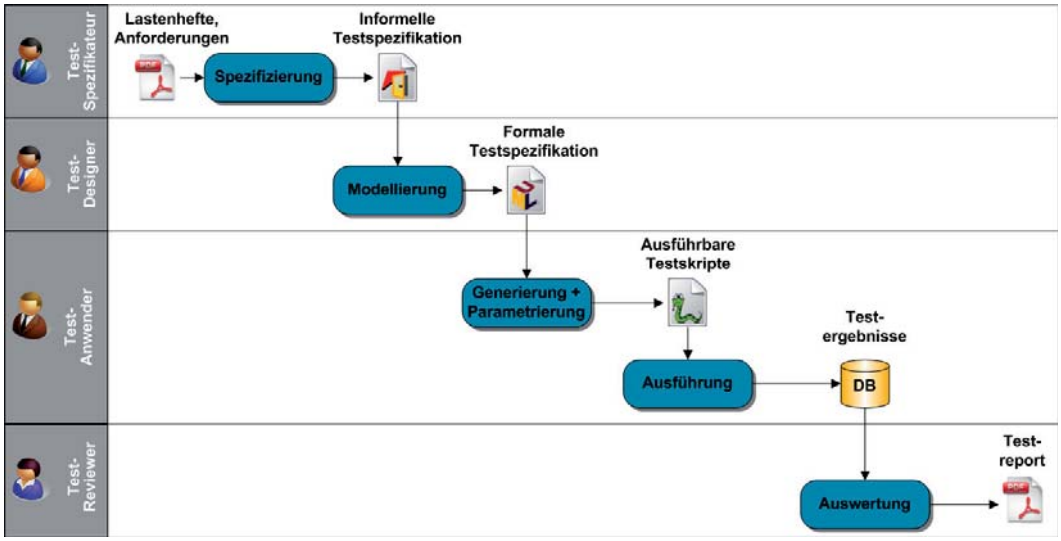

Abbildung 2.10: Kombiniertes Prozess- und Rollenmodell

Der *Testspezifikateur* spezifiziert auf Basis von Funktionslastenheften und Funktionsanforderungen gemeinsam mit dem Funktions- bzw. Steuergeräteverantwortlichen die textuelle Testspezifikation in einem Anforderungsmanagementsystem (z. B. IBM® Rational DOORS®[3]).

Ausgehend von den Spezifikationsdokumenten erstellt der *Testdesigner* mit Hilfe des domänenspezifischen EXAM-UML-Profils ([ZT10]) die formale Testspezifikation in Form logischer Testabläufe unabhängig von einem konkreten Prüfstand in EXAM. Der Modellierer kann dabei auf eine große Anzahl an bestehenden Testfragmenten und auf eine Funktionsbibliothek zugreifen. Die Funktionsbibliothek, welche verschiedene Operationen zur Ansteuerung der Prüfstände und zur Auswertung der Testergebnisse bereitstellt, wird von einer weiteren Person in der Rolle des *Funktionsbibliotheksentwicklers* erstellt und gewartet.

Für die Ausführung der Testabläufe am Prüfstand müssen die logischen Testabläufe parametriert werden und mit Hilfe eines Codegenerators in ausführbare Testskripte automatisiert überführt werden. Die Mehrzahl aller HIL-Hersteller bieten eine Automatisierungsschnittstelle in der Programmiersprache Python ([Haj08]) für ihre Prüfstände an. Historisch gewachsen setzt daher EXAM als Zielsprache für die Testskripte auch die Programmiersprache Python ein. Die Ausführung der Testfälle am Prüfstand wird vom *Testanwender* unterstützt durch die EXAM-Tools durchgeführt.

Nach der Ausführung der Testfälle am Prüfstand werden die Testergebnisse in einer Ergebnisdatenbank gespeichert und von dem *Testreviewer* gemeinsam mit dem zuständigen Steuergeräteentwickler bewertet und für die Erstellung des Testreports aufbereitet.

Eine detaillierte Beschreibung des EXAM-Prozesses ist in [TZ08] dargestellt.

2.4.2 Elemente von EXAM

Im Folgenden beschreiben wir die wichtigsten Elemente von EXAM, die für die Erstellung der Testabläufe benötigt werden.

Zur Verdeutlichung der Testfallerstellung und der Testausführung sind in Abbildung 2.11 die verschiedenen Schichten zur Testausführung in EXAM dargestellt. Die EXAM-Anwender modellieren mit Hilfe von Sequenz- bzw. Aktivitätsdiagrammen den logischen Testablauf unter Verwendung der EXAM-Elemente *TestCase*, *TestSequence* und *TestActivity*. Die für die Parametrierung der Testfälle benötigten Werte sind in konkreten Instanzen von Parameterklassen gespeichert. Der lesende und schreibende Zugriff auf Prüfstandswerte ist über die entsprechenden Instanzen einer *MappingClass* realisiert.

Die Funktionsbibliothek stellt alle benötigten Operationen z. B. zur Stimulation des Prüfstands, des SUT und für die Auswertung der Messdaten zur Verfügung. Alle Aktionen zur Steuerung des virtuellen Fahrzeugs und zur Konfiguration der Fahrzeugumgebung sind in der Funktionsbibliothek *stdAbstractCar* zusammengefasst.

[3] DOORS ist ein Anforderungsmanagementsystem der Firma IBM. Produktseite für weitere Informationen http://www-01.ibm.com/software/awdtools/doors/

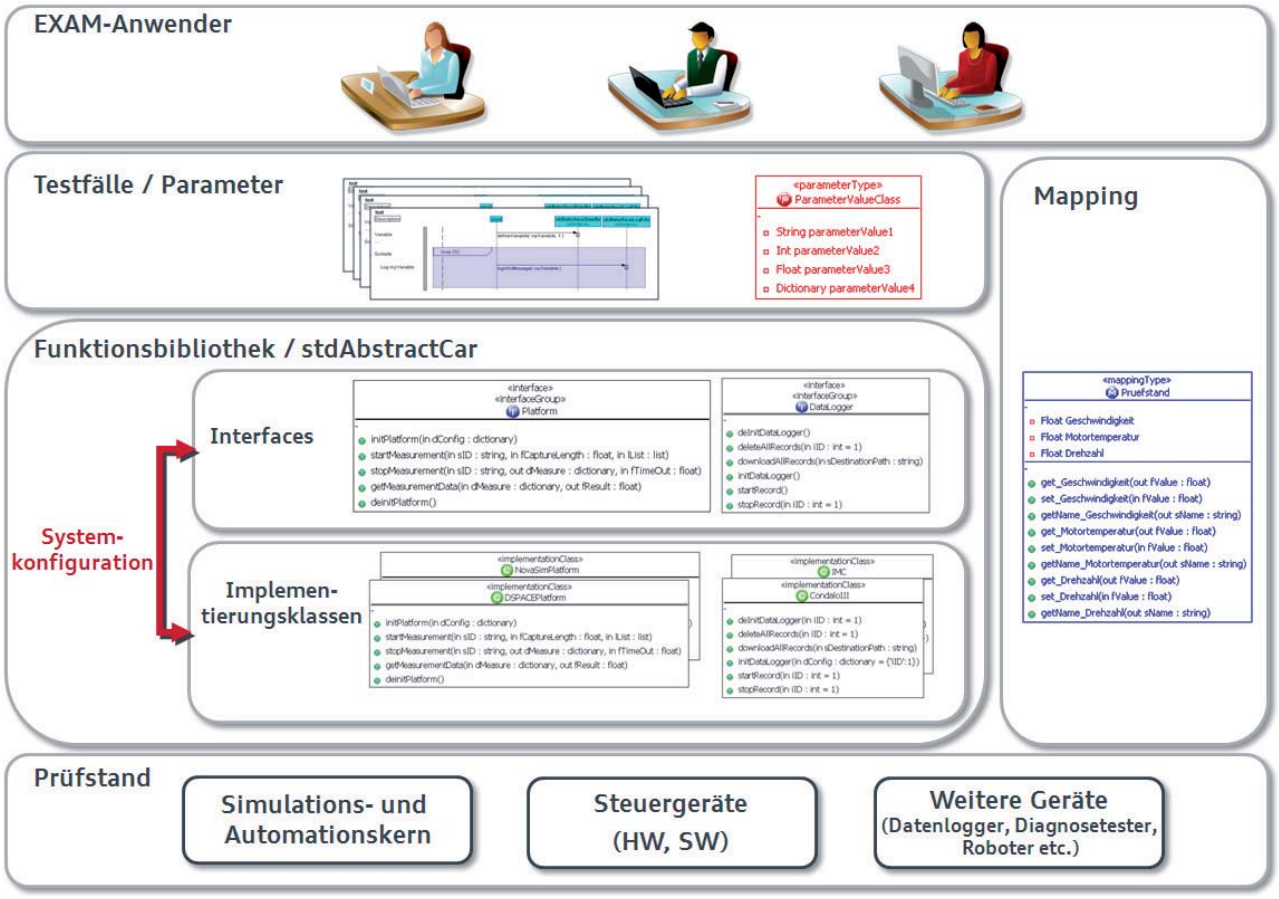

Abbildung 2.11: Schichten von EXAM

Die Modellierung der Testfälle erfolgt dabei plattformunabhängig (unabhängig von einer konkreten Instanz eines Prüfstandes) durch den Aufruf von Interfaceoperationen. Für jedes Interface können beliebig viele Implementierungsklassen existieren, die plattformspezifisch die Ansteuerung der entsprechenden Soft- oder Hardwarekomponenten übernehmen.

Damit in einem Testfall z. B. die Datenaufzeichnung der Prüfstandshardware verwendet werden kann, wird die Interfaceoperation *startRecord()* aufgerufen. Abhängig auf welcher Prüfstandshardware der Testfall ausgeführt wird, unterscheidet sich die Ansteuerung der Hardware. Der Programmcode für die konkrete Ansteuerung über die entsprechende API der Prüfstandshardware ist in einer Implementierungsklasse enthalten. Ein weiteres Beispiel ist das Einschalten der Zündung des Fahrzeugs. Abhängig von dem eingesetzten Prüfstand ist diese Funktion entweder über ein Relais, über die Bedienung des Fahrzeugschlüssels mit Hilfe eines Roboters oder durch das Setzen einer Variablen im Echtzeitmodell realisiert. Aus EXAM-Anwender-Sicht wird in jedem Testfall nur die Interfaceoperation *ZündungEinschalten()* aufgerufen.

In einer Systemkonfiguration wird eine konkrete Zuordnung zwischen Implementierungsklassen und Interfaces definiert. Hierbei wird jedem Interface genau eine Implementierungsklasse zugeordnet. In einem EXAM-Modell existieren in der Regel für verschiedene Prüfstände und Fahrzeugprojekte mehrere Systemkonfigurationen. Vor der Ausführung des Testfalls am Prüfstand

kann der Anwender eine Systemkonfiguration auswählen und somit festlegen, welche Implementierungsklasse jeweils zur Laufzeit verwendet werden soll.

Nach dem allgemeinen Überblick beschreiben wir im Folgenden die wichtigsten Elemente zur Modellierung der Testabläufe in EXAM auf Basis des EXAM-Domänenmodells ([ZT10]). Ein Auszug des Modells mit den wichtigsten Elementen ist in der Abbildung 2.12 dargestellt.

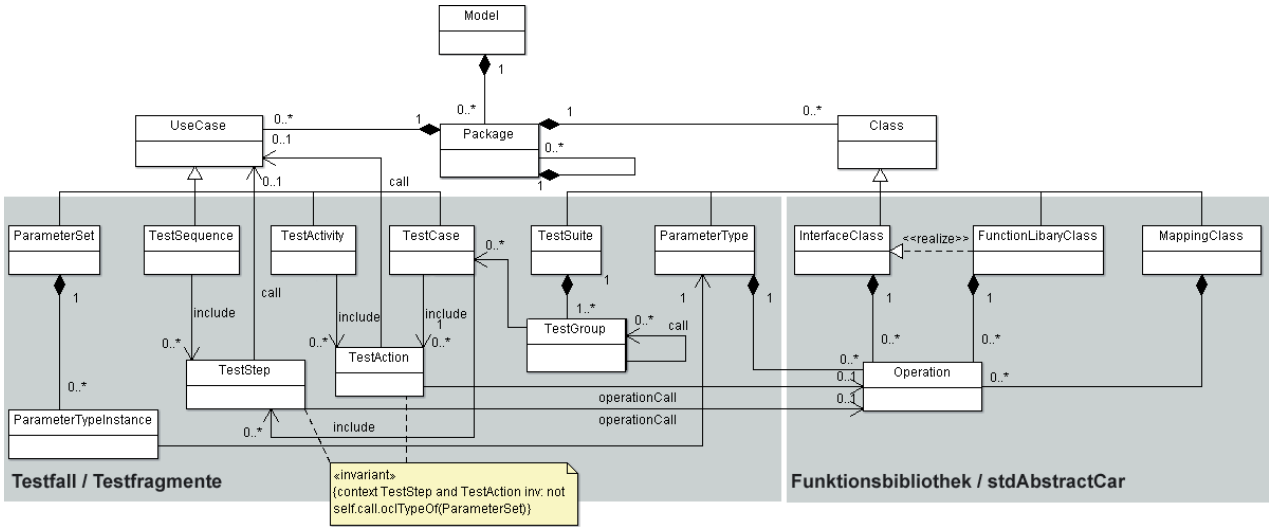

Abbildung 2.12: Auszug aus dem EXAM-Domänenmodell

Zur besseren Orientierung ist das Modell in die Bereiche *Testfall/Testfragmente* und *Funktionsbibliothek/stdAbstractCar* unterteilt. Alle Elemente, die in einem Testablauf enthalten bzw. für dessen Modellierung eingesetzt werden, sind dem Bereich *Testfall/Testfragmente* zugeordnet. Alle Elemente, die für die Modellierung der Prüfstandsanbindung und für die Abstraktion der Prüfstandshardware benötigt werden, sind dem Bereich *Funktionsbibliothek/stdAbstractCar* zugeordnet.

Bevor wir im Folgenden die einzelnen Elemente des Domänenmodells und deren Beziehungen im Detail beschreiben, geben wir einen kurzen Überblick der wichtigsten Elemente.

In EXAM werden für die Modellierung des Testablaufs die Elemente

- *TestSequence*,
- *TestActivity*,
- *TestCase* und
- *TestSuite*

verwendet.

Für die Modellierung wiederverwendbarer Fragmente eines Testablaufs werden die Elemente *TestSequence* und *TestActivity* verwendet. Die *TestSequence* erlaubt einen sequentiellen Ablauf mit Hilfe von Sequenzdiagrammen zu modellieren. Die *TestActivity* erlaubt sowohl einen sequentiellen als auch einen parallelen Ablauf mit Hilfe von Aktivitätsdiagrammen zu modellieren. Für

die Modellierung eines vollständigen Testablaufs wird in EXAM das Element *TestCase* verwendet. Innerhalb eines *TestCases* können beliebig viele Fragmente (*TestSequence* und *TestActivity*) referenziert werden. Ein *ParameterSet* erlaubt die Parametrierung der Testfälle. Die *TestSuite* dient der Gruppierung verschiedener *TestCases* zu einem Testthema.

Innerhalb einer *TestSequence*, einer *TestActivity* oder eines *TestCase* werden Operationen aus der EXAM-Funktionsbibliothek aufgerufen. Jede Operation stellt eine definierte Funktionalität, wie z. B. die Ansteuerung der Prüfstandshardware, Funktionen zur Signalaufzeichnung, Funktionen zur Messdatenauswertung, Funktionen für das Reporting der Testabläufe zur Verfügung. Mit Hilfe der Operationen einer *MappingClass* (Zuordnung zwischen einer Variablen in EXAM und einer Variablen der Simulationsmodelle) kann innerhalb eines Testablaufs lesend und schreibend auf die Variablen der Simulationsmodelle, welche auf der Echtzeithardware der HIL-Prüfstände ausgeführt werden, zugegriffen werden.

Für die Funktionsbibliothek und für das Mapping zwischen Variablen in EXAM und Variablen der Simulationsmodelle werden in EXAM die folgenden Elemente verwendet:

- *InterfaceClass*,
- *FunctionLibaryClass*,
- *Operation* und
- *MappingClass*

Betrachten wir nun die Elemente des EXAM-Domänenmodells (siehe Abbildung 2.12 auf Seite 24) und deren Beziehungen im Detail.

Gemäß der Definition im EXAM-Konzeptpapier [Kif09] sind alle Elemente, die für die Modellierung der Testabläufe benötigt werden, in einem gemeinsamen Modell enthalten. An dieser Stelle ist der Begriff Modell als strukturierte Ablage von Testfällen, Testfallfragmenten, Parametern und Funktionsbibliotheken zur Ansteuerung der Prüfstände zu verstehen und nicht als System- oder Verhaltensmodell.

Aufgrund der Größe der EXAM-Modelle und der Verteilung der Testinhalte auf die verschiedenen Fahrzeugdomänen (Antrieb, Komfort, Fahrwerk und Infotainment) existieren pro Fahrzeugdomäne bzw. pro HIL-Prüfstand eigene Modelle. Die Modelle werden dabei zentral in einer Datenbank abgelegt und ermöglichen somit die gemeinsame Entwicklung von Testfällen und die Wiederverwendung von Modellkomponenten für alle Mitarbeiter, welche am Testprozess der Domäne beteiligt sind.

Wir beschreiben nun die zentralen Elemente der EXAM-Modelle. Zur Strukturierung haben wir die Elemente im Folgenden in die Bereiche *Testablauf*, *Funktionsbibliothek*, *Parameter* und *Mapping* aufgeteilt. Die Beschreibung orientiert sich dabei an den Elementen des EXAM-Domänenmodells (siehe Abbildung 2.12 auf Seite 24).

Elemente der EXAM-Modelle

Für die Strukturierung der Elemente innerhalb eines EXAM-Modells werden Ordner verwendet, wobei jeder Ordner selbst wieder beliebig viele Ordner enthalten kann. Jeder Ordner kann beliebig viele Elemente vom Typ *UseCase* oder *Class* enthalten. Ein *UseCase* kann dabei entweder ein *TestCase*, eine *TestActivity*, eine *TestSequence* oder ein *ParameterSet* sein. Eine Instanz des Elements *Class* ist entweder eine *TestSuite*, ein *ParameterType*, eine *InterfaceClass* oder eine *FunctionLibaryClass*.

Testablauf
Ein *TestCase* entspricht genau einem Testfall der informellen Testspezifikation. Die Modellierung des Testablaufs erfolgt dabei mit Hilfe eines dem *TestCase* zugeordneten Sequenz- oder Aktivitätsdiagramms. Die einzelnen Schritte in einem Sequenzdiagramm werden in EXAM als *TestSteps* und die Aktionen des Aktivitätsdiagramms als *TestActions* bezeichnet. Diese beiden Elemente können entweder ein weiteres *UseCase*-Elemente (*UseCaseCall*) oder eine Operation der EXAM-Funktionsbibliothek (*OperationCall*) aufrufen. Bei einem *UseCaseCall* wird dabei entweder eine *TestSequence* oder eine *TestActivity* aufgerufen (referenziert).

Eine *TestSequence* bzw. eine *TestActivity* erlaubt, wiederverwendbare Testablauffragmente mit Hilfe von Sequenz- respektive Aktivitätsdiagrammen zu modellieren, die wiederum eine beliebige Anzahl an *TestSteps* bzw. *TestActions* enthalten. In einem *TestCase* werden diese Elemente entsprechend zu einem vollständigen Testablauf zusammengefügt.

Eine *TestSuite* ermöglicht es, Testfälle zu einem bestimmten Themengebiet z. B. für Nacht- oder Wochenendläufe zu gruppieren. Die *TestSuite* ist von *Class* abgeleitet und kann daher Operationen besitzen, die in EXAM als *TestGroups* bezeichnet werden. Innerhalb einer *TestGroup* können weitere *TestGroups* und eine beliebige Anzahl von Testfällen aufgerufen werden. Für die Strukturierung der einzelnen Aufrufe werden Sequenzdiagramme verwendet, die der jeweiligen *TestGroup* zugeordnet sind.

Funktionsbibliothek
Die bisherigen Elemente erlauben es, die einzelnen durchzuführenden Aktionen des Testablaufs kausal (in einer definierten Reihenfolge) zu ordnen. Die für den Testablauf notwendigen Operationen (Ansteuerung der Prüfstandshardware, Signalaufzeichnung, Messdatenauswertung, Reporting etc.) sind in EXAM in den Operationen der Funktionsbibliothek enthalten und können von einem *TestStep* oder einer *TestAction* via *OperationCall* aufgerufen werden.

Für eine plattformunabhängige Modellierung des Testablaufs werden innerhalb der Testfälle nur Operationen eines Interface aufgerufen. Eine *InterfaceClass* dient dazu, Operationen logisch zu gruppieren. Für jedes Interface existieren eine oder mehrere Implementierungsklassen (*FunctionLibClass*), welche die entsprechenden prüfstandspezifischen Implementierungen der Operationen enthalten. Die Implementierung kann dabei in Form von Pythoncode oder mit Hilfe eines Sequenzdiagramms oder eines Aktivitätsdiagramms erfolgen.

Die Verwendung eines Sequenz- oder Aktivitätsdiagramms für die Modellierung der Operation ermöglicht jedem EXAM-Anwender, auch ohne Programmiererfahrungen eigene Operationen

zu erstellen. Innerhalb der Sequenzdiagramme können dabei jedoch nur bestehende Operationen der Funktionsbibliothek verwendet werden. Beispiele für die Erstellung einer Funktionsbibliothek mit Hilfe eines Sequenzdiagramms findet sich unter anderem in dem *stdAbstractCar*. Die Bibliothek fasst wiederverwendbare Aktionen für die Interaktion mit dem Fahrzeug und der Fahrzeugumgebung zusammen und wird von den Anwender auf Basis bestehender Bibliotheksfunktionen eigenständig erstellt.

Parameter
Für die Parametrierung der Testfälle werden *ParameterSets* verwendet. Ein *ParameterSet* kann beliebig viele Instanzen eines *ParameterTypes* enthalten. Die Parameter sind dabei als Attribute der *ParameterType*-Klasse modelliert. Der Zugriff auf die Parameter erfolgt durch einen *OperationCall* der entsprechenden Getter-Methode der Klasse.

Mapping
Damit während des Testablauf die aktuellen Werte der Simulation verändert bzw. gelesen werden können, wird in EXAM über eine Instanz einer *MappingClass* die Möglichkeit des lesenden und schreibenden Zugriffs auf die Prüfstandsvariablen (Variablen in den Simulationsmodellen) ermöglicht. Für jede Prüfstandsvariable wird ein entsprechendes Attribut (mit Datentyp) in einer *MappingClass* angelegt. Der lesende und schreibende Zugriff auf die Attribute der Klassen erfolgt über automatisch generierte Getter- und Setter-Operationen.

2.4.3 Modellierung des Testablaufs

Die Beschreibung des Testablaufs in EXAM erfolgt mit Hilfe der in der UML2 (vergleiche z. B. [RHQ+05]) beschriebenen Sequenz- bzw. Aktivitätsdiagramme. Aufgrund der Anforderungen von EXAM an die Modellierung des Testablaufs sind die beiden Diagramme entsprechend um spezifische Kontrollstrukturen erweitert worden. Im Folgenden beschreiben wir die beiden Diagrammtypen und die entsprechenden Konstrukte zur Steuerung des Kontrollflusses für die Modellierung des Testablaufs in EXAM.

Sequenzdiagramm
In EXAM werden die Sequenzdiagramme für die Modellierung eines linearen Testablaufs eingesetzt. Die Diagramme sind dabei den EXAM-Elementen *TestCase* oder *TestSequence* zugeordnet. Für die Gruppierung der Testfälle innerhalb einer *TestSuite* werden in einer *TestGroup* Sequenzdiagramme zur Definition der Ausführungsreihenfolge der *TestCases* verwendet. Eine Zeile in dem Sequenzdiagramm wird in EXAM als *TestStep* bezeichnet.

Nach der Definition in UML2 besteht ein Sequenzdiagramm aus einer Menge von geordneten Operationsaufrufen, Klassen, Lebenslinien und Interaktionsfragmenten. Eine detaillierte Beschreibung gibt z. B. [RHQ+05].

Für den Einsatz in EXAM werden die in der UML2 definierten Sequenzdiagramme um weitere Interaktionsfragmente zur Steuerung des Testablaufs erweitert. In einen Sequenzdiagramm in EXAM können folgende Interaktionsfragmente eingesetzt werden:

- **ref** Verweis (*UseCaseCall*) auf ein weiteres EXAM-Element
- **alt** Alternative Ausführung von Operationsaufrufen

- **for** Wiederholung einer definierten Anzahl an Operationsaufrufen mit Schleifenzähler
- **while** Wiederholung einer definierten Anzahl an Operationsaufrufen
- **break** Vorzeitiger Abbruch eines Interaktionsfragments
- **return** Vorzeitiger Abbruch der Ausführung des Sequenzdiagramms

In Abbildung 2.13 ist ein Beispiel für ein Sequenzdiagramm mit allen in EXAM definierten Interaktionsfragmenten dargestellt.

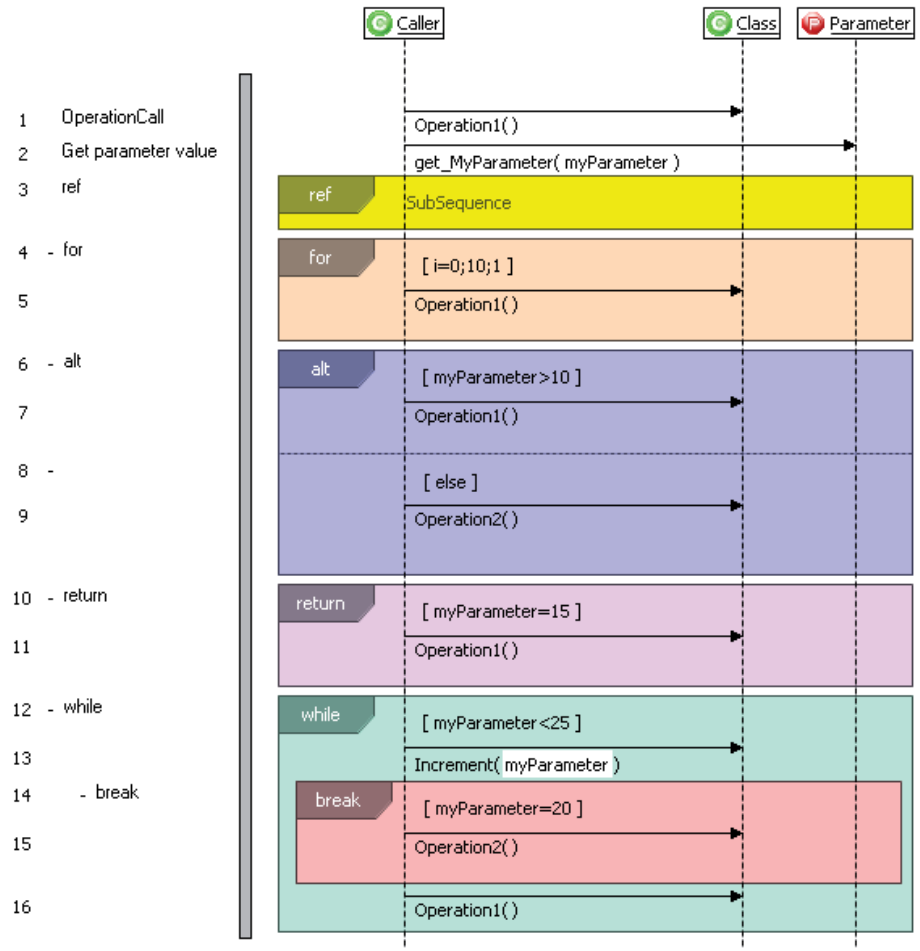

Abbildung 2.13: Kontrollflusselemente von Sequenzdiagrammen

In der ersten Zeile des Sequenzdiagramms in Abbildung 2.13 ist der Aufruf einer Bibliotheksoperation mit dem Namen *Operation1()* dargestellt (*OperationCall*). Die zweite Zeile enthält den Aufruf einer Getter-Methode einer Parameterklasse. Der in dem Attribut *MyParameter* gespeicherte Wert wird durch diesen Aufruf in die lokale Variable *myParameter* gespeichert. Diese Variable kann im weiteren Verlauf des Sequenzdiagramms verwendet werden.

Das Interaktionsfragment *ref* wird für einen *UseCaseCall* benötigt und ermöglicht in einem *TestStep* ein EXAM-Element von Typ *TestCase*, *TestSequence* oder *TestActivity* zu referenzieren. Zur

Laufzeit werden anstelle des Interaktionsfragments die Operationsaufrufe des referenzierten Elementes ausgeführt. In diesem Beispiel wird in der dritten Zeile die *TestSequence* mit dem Namen *SubSequence* aufgerufen.

Für den wiederholten Aufruf von Operationen wird das Fragment *for* (Zeile 4-5) oder *while* (Zeile 12-16) eingesetzt. Das Fragment *for* entspricht einer For-Schleife und erlaubt die Anzahl der Wiederholungen vor der Ausführung der Schleife zu definieren. Die Syntax der Bedingung entspricht dabei der einer For-Schleife in Python. In dem dargestellten Beispiel wird die *Operation1()* in Zeile 5 zehn Mal aufgerufen, wobei die Variable *i* in jedem Schleifendurchlauf um eins erhöht wird.

Für die Modellierung einer Alternative im Testablauf wird das Interaktionsfragment *alt* verwendet (Zeile 6-9). Wird die Bedingung des Fragments als wahr ausgewertet, werden die Operationsaufrufe innerhalb des Fragments ausgeführt. Wenn die Bedingung als falsch ausgewertet wird, werden die Operationen innerhalb des Fragments übersprungen. In dem dargestellten Sequenzdiagramm wird die *Operation1()* in Zeile 7 aufgerufen, wenn der Wert der Variablen *myParameter* größer als zehn ist. Im Anschluss wird die Ausführung in der Zeile 10 fortgesetzt. Ist die Bedingung nicht erfüllt, wird die Bedingung des kombinierten *alt*-Fragment in Zeile 8 überprüft. Das kombinierte *alt*-Fragment ist vergleichbar mit einer If-Else-Anweisung in einer Programmiersprache.

Das *while*-Fragment entspricht semantisch der While-Schleife in Python. Die Operationen innerhalb der Schleife werden solange wiederholt ausgeführt bis die Bedingung falsch ist. Im dargestellten Beispiel werden die Operationen in den Zeilen 13 bis 16 solange wiederholt, wie der Wert der Variablen *myParameter* kleiner als 25 ist. Der Wert der Schleifenvariablen wird bei jedem Schleifendurchlauf in der Zeile 13 durch die Operation *Increment(myParameter)* jeweils um eins erhöht.

Das Interaktionsfragment *break* wird dazu verwendet ein Interaktionsfragment vorzeitig zu verlassen. Ist die Bedingung des *break*-Fragments erfüllt, werden alle Operationsaufrufe innerhalb des Fragments aufgerufen und im Anschluss das direkt umgebende Interaktionsfragment beendet. In dem angegebenen Beispiel wird das *while*-Fragment bereits bei dem Wert 20 der Variablen *myParameter* nach dem Aufruf der *Operation2()* in Zeile 15 verlassen.

Das Interaktionsfragment *return* erlaubt die Ausführung des Sequenzdiagramms vorzeitig zu unterbrechen. Ist die Bedingung des Fragments erfüllt, werden alle enthaltende Operationsaufrufe ausgeführt, bevor die Ausführung des gesamten Sequenzdiagramms beendet wird.

Aktivitätsdiagramm

Die Aktivitätsdiagramme werden in EXAM für die Modellierung eines linearen und/oder parallelen Testablaufs eingesetzt. Die Diagramme sind in EXAM einer *TestActivity* oder einem *TestCase* zugeordnet. Für eine allgemeine Beschreibung der Aktivitätsdiagramme verweisen wir auf die entsprechende Literatur (z. B. [RHQ+05]).

Bei den in der UML2 beschriebenen Aktivitätsdiagrammen sind neben den Notationen der Datenflussdiagramme und Nassi-Shneidermann-Diagramme einige wesentliche Elemente von Petri-Netzen aufgegriffen worden (z. B. das Token-Konzept von Petri Netzen (vgl. [RHQ+05], [Val03])).

Die Aktivitätsdiagramme enthalten eine Menge von Aktionen, Kontrollknoten und gerichtete Verbindungen zwischen den Knoten. Die Aktionen der Diagramme werden in EXAM als *TestAction* bezeichnet. Eine *TestAction* ruft genau eine Operation der EXAM-Funktionsbibliothek oder eine *TestSequence* bzw. eine *TestActivity* auf.

Jedes Aktivitätsdiagramm in EXAM enthält genau einen *Startknoten* (*InitialNode*) und beliebig viele *Zielknoten* (*FinalNode*). Bei dem Erreichen eines *Zielknotens* wird die Ausführung des Aktivitätsdiagramms beendet (vgl. *return*-Fragment bei Sequenzdiagrammen). Zur Modellierung einer parallelen Ausführung von Aktionen werden *Parallelisierungsknoten* (*ForkNode*) und *Synchronisationsknoten* (*JoinNode*) verwendet. Verzweigungen des Kontrollflusses werden mit *Verzweigungsknoten* (*DecisionNode*) und *Verbindungsknoten* (*MergeNode*) modelliert. Welcher Weg nach einem Kontrollknoten ausgeführt wird kann über die den Kanten zugeordneten Bedingungen definiert werden.

In Abbildung 2.14 ist exemplarisch ein Aktivitätsdiagramm in EXAM mit den entsprechenden Kontrollknoten und Aktionen dargestellt. Für eine bessere Übersicht bei der Modellierung können mehrere *TestActions* eines Aktivitätsdiagramms zu Gruppen (*TestAction*) zusammengefasst werden. Die Aktion *SubFlow(result)* ist ein Beispiel für eine solche Aktion.

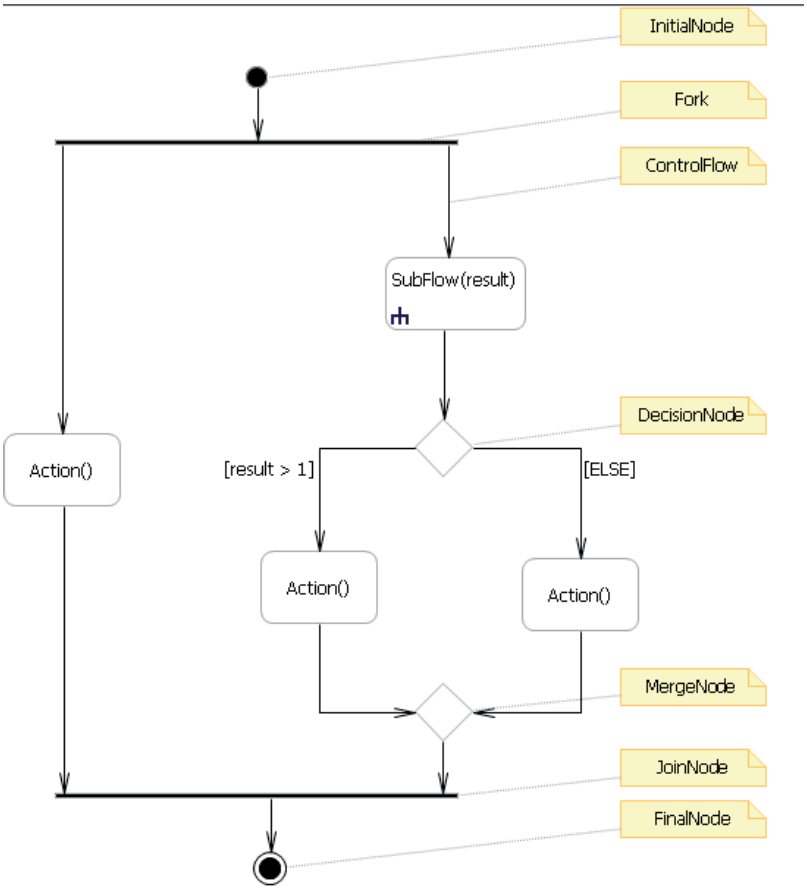

Abbildung 2.14: Kontrollflusselemente von Aktivitätsdiagrammen

2.4.4 Beispiel eines EXAM-Testfalls

Nach der Beschreibung der einzelnen EXAM-Elemente stellen wir in diesem Unterkapitel ein vollständiges Beispiel eines Testablaufs in EXAM inklusive der Parametrierung und dem Zugriff auf den Prüfstand mit Hilfe einer Mappingklasse dar.

Ziel des Testfalls ist die Überprüfung der Blinkfrequenz bei den Funktionen *Autobahnblinken-Links* und *Autobahnblinken-Rechts*. Dazu wird in dem Testfall das Fahrzeug auf eine definierte Geschwindigkeit beschleunigt. Nach dem Erreichen der Zielgeschwindigkeit wird jeweils einmal die Funktion *Autobahnblinken-Links* und *Autobahnblinken-Rechts* ausgeführt. Bei der Ausführung werden die relevanten Bussignale aufgezeichnet. In der Auswertung des Testfalls wird überprüft, ob die aufgezeichnete Blinkfrequenz mit einer vorgegebenen Frequenz (Wert im Parametersatz) übereinstimmt.

Der Testablauf ist dabei, wie von den EXAM-Modellierungsrichtlinien empfohlen, in die drei Sequenzen

- *PreSequence*,
- *ActionSequence* und
- *PostSequence*

unterteilt.

Vor der eigentlichen Stimulation der zu testenden Funktion muss der Prüfstand und das virtuelle Fahrzeug in einen definierten Zustand gebracht werden. Alle dafür notwendigen Aktionen sind in der *PreSequence* enthalten. Die eigentliche Stimulation des zu testenden Objekts erfolgt in der *ActionSequence*. Die Auswertung der aufgezeichneten Messdaten, die Erstellung des Testreports und alle notwendigen Aktionen zur kontrollierten Deinitialisierung des Prüfstands sind in der *PostSequence* enthalten. Dabei werden die drei Elemente in der Regel jeweils als *TestSequence* modelliert und per *UseCaseCall* von dem Testfall referenziert und entsprechend mit Hilfe von Parametern parametriert.

Das in Abbildung 2.15 dargestellte *UseCaseDiagram* zeigt die Aufteilung des Testfalls *EXAM-Testfall* in die drei Sequenzen. Zusätzlich ist die Parametrierung der beiden *TestSequences PreSequence* und *PostSequence* mit dem *ParamterSet Geschwindigkeit* und *BlinkFrequenz* dargestellt. Die Darstellung gibt dabei nur an, welche Elemente von dem Testfall aufgerufen werden bzw. welche Parameter verwendet werden sollen. Das Diagramm gibt keine Auskunft über die Reihenfolge der einzelnen *TestSequences* zueinander.

Die in den *ParameterSets* gespeicherten Werte sind in Abbildung 2.16 dargestellt. Für den Testablauf ist somit die Wunschgeschwindigkeit mit 100 km/h angegeben und die minimale und maximale Blinkfrequenz mit 1 bzw. 3 Hertz.

Wir betrachten nun den Testablauf im Detail und gehen dabei auf die Verwendung der Parameter und den Zugriff auf Prüfstandsgrößen mit Hilfe einer Mappingklasse ein.

In dem Sequenzdiagramm des *TestCases EXAMTestfall* (siehe Abbildung 2.17) sind drei Test-Steps enthalten, die jeweils durch einen *UseCaseCall* die *PreSequence*, *ActionSequence* und *Post-Sequence* in der angegebenen Reihenfolge aufrufen.

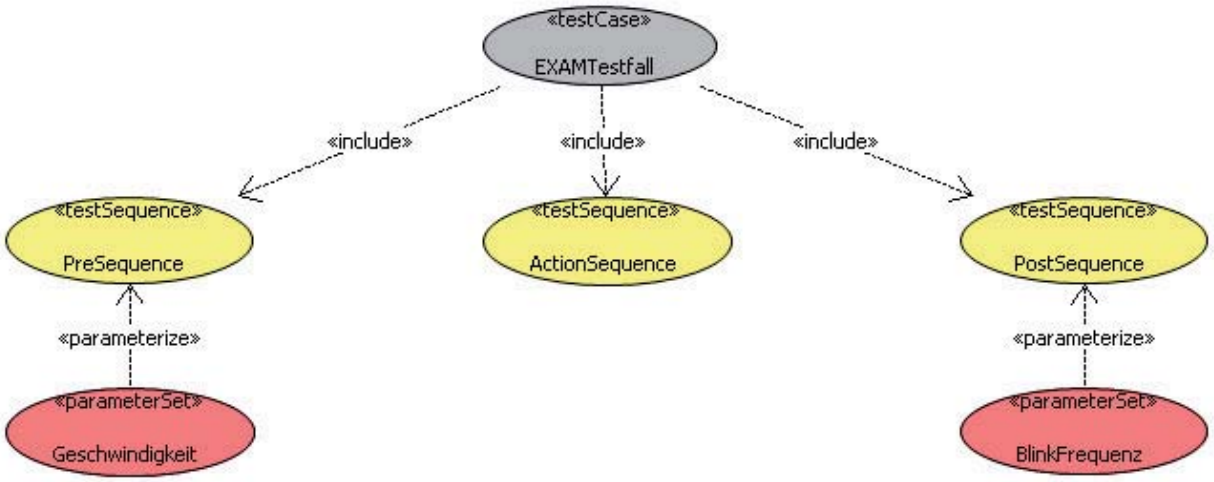

Abbildung 2.15: Darstellung des *TestCases TestCaseExample* mit Parametrierung

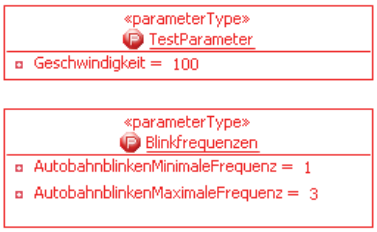

Abbildung 2.16: Parameterwerte für den Testablauf

Damit die Überprüfung der Frequenz der Funktion *Autobahnblinken* durchgeführt werden kann, muss der Prüfstand initialisiert, die Signalaufzeichnung und das Fahrzeug konfiguriert werden. Die dazu notwendigen Operationsaufrufe sind in der in Abbildung 2.18 dargestellten *TestSequence PreSequence* enthalten.

Nach der Initialisierung des Prüfstands im ersten *TestStep* erfolgt die Konfiguration und der Start der Signalaufzeichnung. Dazu wird im zweiten *TestStep* die Operation startMeasurement([„BlinkerSignalLinks“, “BlinkerSignalRechts“]) aufgerufen. Die Parameter der Operation geben an, dass die Signale für die Ansteuerung der Blinker auf der linken und der rechten Fahrzeugseite aufgezeichnet werden sollen. Nach der Aufzeichnung der Signale wird zusätzlich ein Datenlogger für die Aufzeichnung der verwendeten Bussysteme im dritten und vierten Schritt initialisiert und gestartet. In den Schritten fünf bis sieben wird das Fahrzeug gestartet. Im Anschluss wird aus dem zugehörigen Parametersatz *Geschwindigkeit* der Wert des Parameter Geschwindigkeit abgefragt und in eine lokale Variable gespeichert. In dem folgenden Interaktionsfragment von Typ *alt* wird überprüft, ob der Wert der Variablen größer als 25 km/h ist. Ist der Wert kleiner als 25 km/h wird im Schritt 10 die Variable *Geschwindigkeit* auf den Wert 25 gesetzt. In Schritt 11 erfolgt die (virtuelle) Beschleunigung des Fahrzeugs auf den Wert der Variablen *Geschwindigkeit*. Zur Kon-

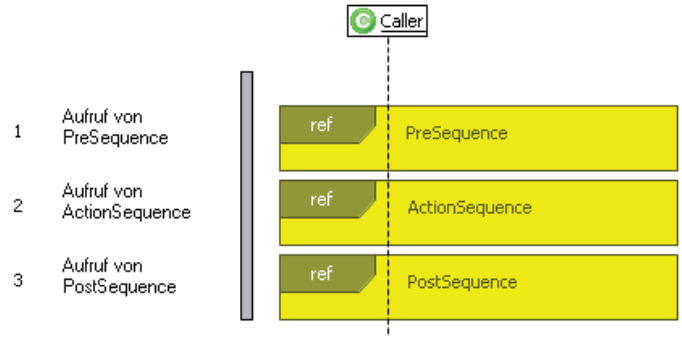

Abbildung 2.17: Testablauf des *TestCases TestCaseExample*

trolle wird der aktuelle Geschwindigkeitswert aus dem Simulationsmodell im nächsten Schritt mit Hilfe der Mappingklasse abgefragt. Im letzten Schritt der *TestSequence* wird dieser Wert in ein Logfile geschrieben.

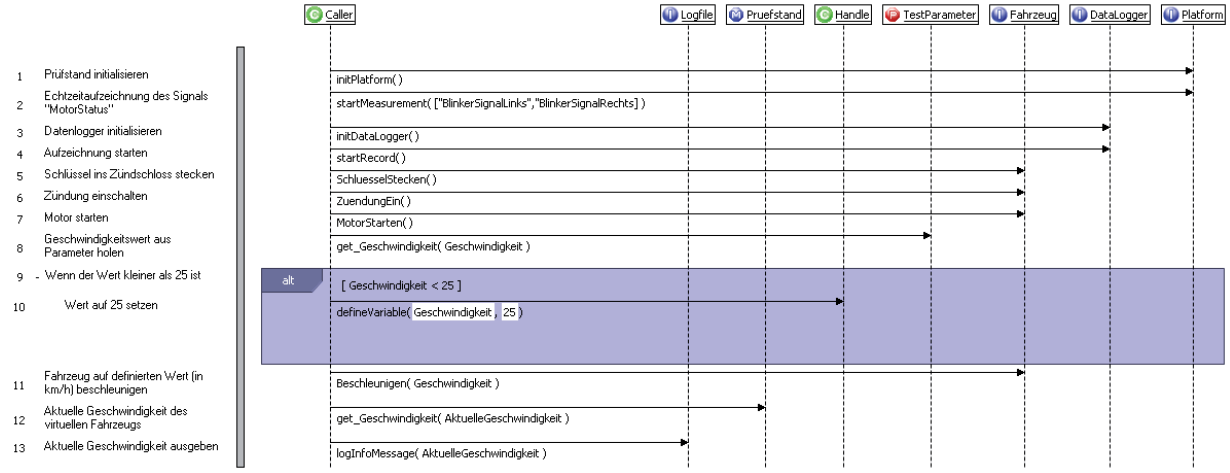

Abbildung 2.18: Operationsaufrufe der *TestSequence PreSequence*

Die eigentliche Stimulation zur Überprüfung der zu testenden Funktion ist in der *ActionSequence* enthalten. In diesem Beispiel wird in dem Sequenzdiagramm (siehe Abbildung 2.19) durch zwei Operationsaufrufe (*Autobahnblinken_Links()* und *Autobahnblinken_Rechts()*) das Autobahnblinken aktiviert. Die Wartezeit von zehn Sekunden (10.000 Millisekunden) ist notwendig, damit sichergestellt werden kann, dass der Blinkzyklus auf der linken Fahrzeugseite nicht durch den Befehl zur Aktivierung des Blinkers auf der rechten Seite unterbrochen wird.

Im Anschluss an die Stimulation erfolgt in der *TestSequence PostSequence* (siehe Abbildung 2.20) die Auswertung der Ergebnisse. Bevor beide Signalaufzeichnungen beendet werden, wird das virtuelle Fahrzeug auf 0 km/h abgebremst und der Motor abgestellt. Die Messdaten werden dabei in die Variable *dMeasure* gespeichert. In den folgenden beiden Operationsaufrufen wird mit Hilfe der Getter-Operationen der Parameterklassen die in dem Parametersatz definierte minimale und maximale Blinkfrequenz abgefragt und entsprechend in die beiden Variablen *minFreq* und *maxFreq* gespeichert. Mit Hilfe der beiden Variablen *minFreq* und *maxFreq* und den

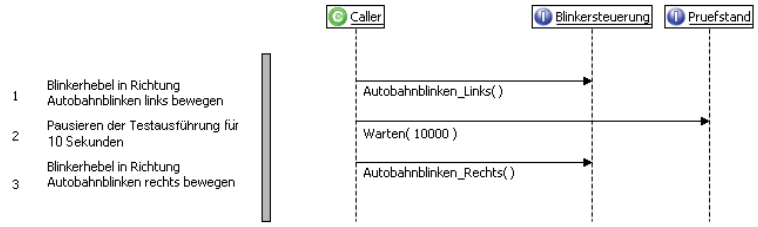

Abbildung 2.19: Operationsaufrufe der *TestSequence ActionSequence*

in *dMeasure* gespeicherten Messdaten wird mit dem Operationsauf *checkFrequence(dMeasure, "BlinkerSignalLinks", minFreq, maxFreq, lReport1)* überprüft, ob die Blinkfrequenz auf der linken Fahrzeugseite den Vorgaben entspricht. Das Ergebnis der Überprüfung wird in die Variable *lReport1* gespeichert und anschließend dem Testreport mit Hilfe des Operationsaufruf *AddResultToReport(...)* hinzugefügt. Entsprechend erfolgt die Überprüfung der Blinkfrequenz auf der rechten Fahrzeugseite analog mit dem *OperationCall* im achten *TestStep*. Die letzten beiden Schritte im Sequenzdiagramm deinitialisieren den Prüfstand und den Datenlogger.

Abbildung 2.20: Operationsaufrufe der *TestSequence PostSequence*

2.4.5 EXAM-Toolsuite und Einsatz von EXAM

EXAM wurde gemeinsam von der AUDI AG und der Volkswagen AG für die Automatisierung von funktionalen Steuergerätetests mit Hilfe von Hardware-in-the-Loop-Prüfständen entwickelt. Ab der Version 2.0 von EXAM basieren die EXAM-Tools[4] auf der Eclipse Rich Client Platform ([Ecl10], [Rei09], [Kün09]).

Das EXAM-Tooling besteht aus vier Komponenten: *EXAM-Modeller*, *EXAM-TestRunner*, *EXAM-Generator* und *EXAM-Reportmanager*. Der *EXAM-Modeller* ermöglicht die grafische Modellierung der Testfälle auf Basis des EXAM-UML-Profil ([ZT10]) und die Verwaltung der EXAM-Modelle. Im Volkswagen-Konzern werden die Modelle zentral durch die IT-Abteilung verwal-

[4]Die EXAM-Tools werden von dem Distributor für EXAM des Volkswagen-Konzerns, der Firma MicroNova AG, vertrieben. Alle Tools und eine Basis-Funktionsbibliothek sind Freeware und können von der EXAM-Webseite (http://www.exam-ta.de) heruntergeladen werden.

tet. Dies ermöglicht die markenübergreifende Zusammenarbeit der verschiedenen Abteilungen bei der Testfallerstellung. Ausgehend von den modellierten Testfällen in den Modellen erzeugt der *EXAM-Generator* inkrementell die für die Testdurchführung am Prüfstand benötigten ablauffähigen Prüfskripte. Der *EXAM-TestRunner* erlaubt die Konfiguration (z. B. die Auswahl der Systemkonfiguration) und Durchführung der Testabläufe und speichert die Testergebnisse in einer Reportdatenbank. Mit Hilfe des *EXAM-Reportmanager* können diese Daten visualisiert, verwaltet und zu Testreporten zusammengefasst werden.

Gegenwärtig wird die Testautomatisierungsmethode EXAM innerhalb des Volkswagen-Konzerns von mehreren hundert Anwendern für die Funktionsabsicherung der Steuergerätefunktionen eingesetzt. Die meisten Anwender verteilen sich dabei auf die Marken Volkswagen, Audi, Skoda, Seat und Bentley.

Alle Anwender eines Bereichs arbeiten dabei auf einem zentralen Modell und können somit Testfragmente austauschen und wiederverwenden. Zurzeit werden von der Konzern-IT rund 50 EXAM-Modelle mit insgesamt mehreren hunderttausend Testfällen betrieben. Die Anzahl der Modelle und der Testfälle wird sich in den nächsten Jahren aufgrund der zukünftigen Herausforderungen bei der Funktionsabsicherung der Steuergeräte deutlich erhöhen. Immer wichtiger wird dann dabei die Zusammenarbeit im Team und die Wiederverwendung bestehender Testfragmente werden.

Neben dem Einsatz der Methode im Volkswagen-Konzern setzen auch weitere Hersteller und eine Vielzahl an Zulieferern der Automobilindustrie EXAM für die Funktionsabsicherung von Steuergeräten ein.

3 Motivation und Problemstellung

Aus Sicht der Test-Ingenieure wird die Funktionsabsicherung der Fahrzeugfunktionen zunehmend komplexer. Zum einen steigt die Anzahl der Funktionen stetig an und zusätzlich werden die Funktionen immer komplexer und verteilen sich auf verschiedene Steuergeräte im ganzen Fahrzeug. Zum anderen wird die effektive Entwicklungszeit eines Fahrzeuges auf Grund kürzerer (Lebens-)Zyklen der Fahrzeugprojekte und einer stärkeren Derivatisierung der Fahrzeugprojekte immer geringer. EXAM versucht, durch den Ansatz einer grafischen Modellierung der Testabläufe, der Rollentrennung zwischen Testfallentwickler und Funktionsbibliotheksentwickler und der Möglichkeit, Testfragmente wiederzuverwenden, den Tester bei seiner täglichen Arbeit zu unterstützen und damit die Komplexität weiterhin beherrschbar zu machen. Dies kann jedoch nur gelingen, wenn die Synergien einer gemeinschaftlichen Testfallentwicklung effizient genutzt werden können und die Testprogramme zum Zeitpunkt der Testausführung möglichst fehlerfrei sind. Daher ist die Qualität der Testfragmente von entscheidender Bedeutung.

In diesem Kapitel stellen wir zuerst den Zusammenhang zwischen der Entwicklung der Fahrzeugfunktionen, den zur Absicherung benötigten Testfällen, der benötigten Testausführungszeit und der zur Verfügung stehenden Prüfstände dar. Im Anschluss betrachten wir, wie die Qualität der Testfälle definiert und bestimmt werden kann. Im letzten Abschnitt beschreiben wir die Auswirkungen von mangelnder Qualität der Testfälle und die sich daraus ergebende Problemstellung der Arbeit.

3.1 Steigende Komplexität bei der Funktionsabsicherung

Die Anzahl der Fahrzeugfunktionen hat sich, wie in den Abbildungen 2.1 und 2.3 auf der Seite 6 bzw. 8 im Kapitel 2 dargestellt, alleine in den letzten zehn Jahren mehr als verdreifacht. Neben dem quantitativen Anstieg der Funktionen ist in dieser Zeit die Komplexität und der Vernetzungsgrad der Funktionen überproportional angestiegen. So waren z. B. im Jahr 1997 an der Motorsteuerung nur drei Steuergeräte beteiligt. Heute sind bei Funktionen wie dem Adaptive Cruise Control (ACC) oder dem Energiemanagement des Fahrzeugs bereits mehr als 25 Steuergeräte beteiligt.

Mit dem Anstieg und der zunehmenden Komplexität der Funktionen ist auch die Anzahl der Testfälle für die Überprüfung der Funktionen in den letzten Jahren deutlich angestiegen. Seit der Einführung der Testautomatisierung EXAM im Jahr 2005 hat sich die Anzahl der Testfälle bis zum Jahr 2010 mehr als vervierfacht. Der Anstieg ist mit der steigenden Anzahl an Funktionen und mit der Zunahme der Komplexität sowie mit der Verteilung der Funktionen auf mehrere Steuergeräte zu begründen. Neben der Anzahl der Testfälle ist auch der Umfang und die Komplexität

der einzelnen Testfälle deutlich angestiegen. Dies hat zur Folge, dass die benötigte Ausführungszeit der Testfälle am Prüfstand signifikant mehr Zeit benötigt. So werden z. B. alleine für die Überprüfung aller Funktionen der Standheizung mehrere Stunden Prüfstandszeit benötigt.

Aus wirtschaftlichen Gründen kann jedoch, aufgrund der hohen Anschaffung und Betriebskosten der Hardware-in-the-Loop-Prüfstände, die Anzahl der Prüfstände nicht in gleichen Umfang wachsen wie die Anzahl der Fahrzeugfunktionen. Die zunehmende Komplexität der Fahrzeugfunktionen hat auch eine direkte Auswirkung auf die Hardware-in-the-Loop-Prüfstände. Durch den Anstieg der Funktionen steigt die Anforderung an die Echtzeithardware der Prüfstände durch die gestiegene Kommunikation der Steuergeräte. Zusätzlich erfordern viele Steuergeräte detailliertere Echtzeitmodelle. So war es z. B. früher für die Absicherung des Motorsteuergeräts ausreichend, die Motortemperatur mit einem konstanten Wert zu simulieren. Heute ist es notwendig, aufgrund einer intelligenteren Eigendiagnose der Steuergeräte, die Motortemperatur in Abhängigkeit der Geschwindigkeit, der Betriebsdauer, der Umweltbedingungen und weiteren Faktoren realitätsnah zu berechnen und ständig zu aktualisieren. Weiterhin müssen für den Test von Funktionen, wie z. B. der ACC oder der PreCrash-Funktion die Prüfstände um realistische Umgebungsmodelle des Fahrzeugs erweitert werden.

Aus diesen Gründen hat sich im Vergleich zu der Entwicklung der Anzahl der Testfälle und deren Ausführungszeiten die Anzahl der Prüfstände in den letzten Jahren gerade einmal verdoppelt. Viele Prüfstände werden aufgrund der hohen Anschaffungs- und Betriebskosten für mehrere Fahrzeugprojekte gleichzeitig verwendet. Dies hat den Effekt, dass die für ein Fahrzeugprojekt zur Verfügung stehende effektive Testzeit weiter reduziert wird, da sich aufgrund der gestiegenen Derivatisierung häufiger die Entwicklung der Fahrzeugprojekte überscheidet. Als Folge davon hat sich die zur Verfügung stehende Testausführungszeit der EXAM Testfälle an den HIL-Prüfständen in den letzten Jahren immer weiter reduziert. Die Ausführung beliebig vieler Testfälle pro Fahrzeugprojekt ist daher nicht möglich. Schon heute kann nur durch eine detaillierte Resourcenplanung der Prüfstandszeit sichergestellt werden, dass alle Funktionen des Fahrzeugs in der zur Verfügung stehenden Zeit abgesichert werden können. Die Wiederholung eines Testfalls, der aufgrund von Systemfehlern oder eines Fehlers in der Modellierung des Testablaufs wiederholt ausgeführt werden muss ist daher nur in Ausnahmefällen möglich und hat zur Folge, dass die Testplanung überarbeitet werden muss. Um einen reibungslosen Testbetrieb sicherstellen zu können muss somit gewährleistet werden können, dass vor der Ausführung eines Testablaufs am Prüfstand dieser eine ausreichende Qualität besitzt. Jeder Fehler bei der Testausführung hat einen direkten Einfluss auf den Testprozess und somit auch indirekt auf die Entwicklungskosten.

Der deutliche Anstieg der Fahrzeugfunktionen und die zunehmende Komplexität der einzelnen Funktionen hat auch Auswirkungen auf die EXAM-Anwender, da für die Absicherung der Funktionen immer mehr und auch komplexere Testfälle erforderlich sind. Durch die Komplexität und die Verteilung der Funktionen über verschiedene Steuergeräte steigt auch die notwendige Anzahl an Operationsaufrufen und Kontrollstrukturen innerhalb der Testfälle und somit auch die Komplexität der Testfälle. Innovationen bei den Fahrzeugfunktionen erfordern neue Techniken und Technologien in der Prüftechnik, die wiederum von EXAM mit Hilfe der Funktionsbibliothek angesteuert werden müssen. Parallel zu dem Anstieg der EXAM-Modellkomplexität ergibt sich

durch die Verkürzung der Entwicklungszeiten und die Zunahme von Derivaten die Notwendigkeit, dass die EXAM-Anwender Testfälle für mehrere Fahrzeugprojekte und verschiedene Funktionen gleichzeitig entwickeln und am Prüfstand absichern müssen. Insgesamt steigt dadurch die Belastung des einzelnen EXAM-Anwenders, der gezwungen wird komplexere Testabläufe in vergleichbarere Qualität in kürzerer Zeit zu entwickeln.

Für eine effiziente Funktionsabsicherung, für die Zusammenarbeit bei der Erstellung der Testfälle und für den Austausch der Testfälle muss trotz der steigenden Anforderungen und der schwierigen Rahmenbedingungen die Qualität der Testfälle sichergestellt werden. Im folgenden Abschnitt definieren wir die Qualität eines EXAM-Testfalls.

3.2 Qualität der Testfälle

Die Erstellung der Testfälle in EXAM ist vergleichbar mit der Erstellung einer Softwarekomponente. Da die Testfälle grafisch modelliert werden kann bei der Erstellung der Testfälle bzw. der Testfallfragmente von einer grafischen Programmierung gesprochen werden. Die EXAM-Testfälle können dabei als eigenständige Programme angesehen werden, die automatisiert die Funktionalität der Fahrzeugfunktionen absichern. Der Anwender ruft in den Sequenz- bzw. Aktivitätsdiagrammen Operationen der Funktionsbibliothek auf, steuert den Kontrollfluss mit Hilfe der Interaktionsfragmente sowie Kontrollknoten und kann über Parameter den Testablauf parametrieren. Dabei enthält ein einzelner Testfall mehrere tausend Operationsaufrufe und eine *TestSuite* kann mehrere hunderttausend Operationsaufrufe enthalten.

Bei der Softwareentwicklung ist die Qualität der Programme (z. B. Einhaltung von Codierungsrichtlinien, möglichst fehlerfreie Programme etc.) die Voraussetzung für die gemeinschaftliche Entwicklung der Komponenten im Team und für die spätere Akzeptanz und Einsetzbarkeit der Software. Die Qualität der Testfälle in EXAM ist von entscheidender Bedeutung für eine effiziente Funktionsabsicherung, die gemeinschaftliche Entwicklung der Testfälle und den Austausch der Testfälle sowohl im Volkswagen-Konzern als auch mit Zulieferern. Mangelnde Qualität der Testfälle, wie z. B. nicht eingehaltene Modellierungsrichtlinen oder syntaktische bzw. semantische Fehler in den Testfällen, hat eine direkte Auswirkung auf die Funktionsabsicherung der Fahrzeugfunktionen und somit auch indirekt auf den gesamten Fahrzeugentwicklungsprozess.

Bevor wir die Qualitätskriterien für die Erstellung der Testfälle in EXAM betrachten ist eine Definition von Qualität notwendig. In der ISO-Norm 8402 [ISO95] sind die wichtigsten Begriffe des Softwarequalitätsmanagement beschrieben. Sie wurde im Jahr 2005 durch die Normen EN ISO 9000:2005 [ISO05] und EN ISO 9001 [ISO08a] ersetzt und durch die Norm ISO/IEC 90003 [ISO04] für den Bereich Softwareentwicklung weiter konkretisiert. In diesen Normen ist Qualität mit Hilfe von Merkmalen und Anforderungen definiert, die eine Einheit erfüllen muss:

> Qualität ist die Gesamtheit von Merkmalen einer Einheit bezüglich ihrer Eignung, festgelegte und vorausgesetzte Erfordernisse zu erfüllen. *ISO 8402* [ISO95]

Übertragen auf die Testfälle in EXAM (Einheit) ist das Ziel (Erfordernisse) die automatisierte Absicherung der Fahrzeugfunktionen. Die dazu notwendigen Merkmale der Einheit sind eine einheitliche Modellierung der Testfälle für die Wiederverwendbarkeit der Testfallfragmente und die gemeinschaftliche Entwicklung der Fragmente im Team. Ebenso darf die Einheit keine syntaktischen oder semantischen Fehler enthalten, die bei der Ausführung der Testfälle am Prüfstand zu einem Abbruch oder zu einem nicht sinnvollen Verhalten der Testausführung führen.

In der Softwareentwicklung beschäftigt sich die Qualitätssicherung bzw. das Qualitätsmanagement entlang des gesamten Entwicklungsprozesses damit, wie die Qualität von Software gemessen und wie mit Hilfe verschiedener Verfahren die Qualität sichergestellt bzw. gesteigert werden kann. Ein Überblick und eine detaillierte Beschreibung der verschiedenen Aufgaben und angewendeten Methoden im Bereich der Softwarequalitätssicherung geben z. B. [Sch07], [Lig09] oder [SL05].

Schneider [Sch07] teilt die Maßnahmen der Qualitätssicherung in die drei Kategorien

- analytisch,
- organisatorisch und
- konstruktiv

ein und beschreibt deren Aufgaben wie folgt:

> Die Qualitätssicherung wird oft systematisch in drei Bereiche eingeteilt: analytische, konstruktive und organisatorische Maßnahmen. Analytische Maßnahmen suchen Fehler, ... Konstruktive versuchen schon bei der Entwicklung, Probleme gar nicht erst entstehen zu lassen. Und organisatorische Maßnahmen bilden den Rahmen um die ersten beiden [...] [Sch07]

Zu den *analytischen Maßnahmen* gehört unter anderem das systematische Testen der Komponenten mit z. B. Black- oder Glass-Box[1]-Testverfahren, Reviews und Inspektionen des Codes und formalen Verfahren wie z. B. Model-Checking ([CGP99], [CW96], [BK08a], [VHBP00]), die statische Analyse des Programmcodes ([EM04], [ECH+01], [BPS00]) oder die symbolische Programmausführung ([BHS07], [Wal01]). Unter die Kategorie *konstruktive Maßnahmen* fallen z. B. Code- bzw. Modellierungsrichtlinien, Design-Patterns oder auch agile Softwareentwicklungsmethoden wie z. B. das eXtreme Programming (XP [BA04]). Damit die Maßnahmen der beiden ersten Kategorien wirksam werden können, muss die Softwarequalität im Unternehmen verankert sein. Die *organisatorischen Maßnahmen* stellen dies sicher.

Zur Sicherstellung qualitativ hochwertiger Testfälle in EXAM müssen die Maßnahmen aus dem Bereich der Softwareentwicklung auf die Entwicklung der Testfragmente in EXAM übertragen werden. Dabei sind in EXAM zwei Fehlertypen zu unterscheiden. Zum einen kann ein Testablauf einen Fehler enthalten, welcher nur eine Auswirkung auf die Zusammenarbeit im Team hat aber keine Auswirkung auf die Testausführung am Prüfstand. Zum anderen kann ein Testablauf einen Fehler enthalten, welcher eine direkte Auswirkung auf die Ausführung der Testfälle am

[1] In [SL10] auch als White-Box bezeichnet.

Prüfstand in Form eines Abbruchs der Testausführung oder in Form einer fehlerhaften Stimulation des zu testenden Objekts hat. Nur durch eine Steigerung der Qualität und die Minimierung der Fehler kann der im vorherigen Abschnitt beschriebene Anstieg der Komplexität und die damit einhergehende Anspannung des Testprozesses entgegengewirkt werden.

Die beiden wichtigsten eingesetzten *konstruktiven Maßnahmen* in EXAM sind zum einen ein agiler Testprozess ([ZT10]), in dem der EXAM-Anwender gemeinsam mit dem Funktionsverantwortlichen in einem Funktionstestteam die Testfälle entwickelt und ausführt. Zum anderen existieren klare Vorgaben zur Modellierung der Modellelemente in Form von Modellierungsrichtlinien. Als *analytische Maßnahme* werden parallel zur Generierung des Testcodes die Diagramme auf syntaktische Fehler, eine Metamodell-konforme Modellierung und auf Verletzungen von Namenskonventionen überprüft. Als *organisatorische Maßnahmen* gibt es einen Modellierungsverantwortlichen und pro Domänenmodell einen Modellverantwortlichen.

Im Moment fehlen noch Maßnahmen, die den Anwender dabei unterstützen, die semantischen Modellierungsfehler in den Testfällen zu erkennen und zu reduzieren. Es geht an dieser Stelle nicht um eine Überprüfung, ob der Testablauf die Steuergerätefunktion richtig überprüft, sondern darum, ob die Aufrufreihenfolge der Operation innerhalb des Testfalls zu keinen Fehlern während der Ausführung des Testfalls am Prüfstand führt. Die richtige Ausführungsreihenfolge der Operationen ergibt sich zum einen durch die kausale Beziehung (Ausführungsreihenfolge) der Operationen und den Vorgaben in den Modellierungsrichtlinien. Im Vergleich zu gängigen Programmiersprachen ist dies z. B. eine Überprüfung, ob vor jedem Aufruf zum Schreiben von Daten in eine Datei, die Datei zum Schreiben geöffnet wurde.

Die Entwicklung einer Methode zur Überprüfung der richtigen Ausführungsreihenfolge der Operationen in den Testabläufen ist Ziel dieser Arbeit. Bevor wir die Ziele der Arbeit im Detail beschreiben, betrachten wir die Auswirkungen von falschen Ausführungsreihenfolgen innerhalb der Testfragmente und teilen die kausalen Beziehungen der Operationen in drei Kategorien ein.

3.3 Modellierungsfehler in den Testabläufen

Für die Überprüfung der kausalen Abhängigkeiten zwischen den EXAM-Bibliotheksfunktionen müssen die kausalen Abhängigkeiten der Operationen in Form von Regeln/Bedingungen definiert werden. Die Abhängigkeiten ergeben sich dabei aus der Logik der Fahrzeugfunktionen, der Logik der benötigten Testhardware oder aus den Modellierungsrichtlinien.

Die Bedingungen können daher in drei Kategorien

- domänenspezifische Bedingungen,
- domänübergreifende Bedingungen und
- Bedingungen basierend auf den Modellierungsrichtlinien

eingeteilt werden, die wir im Folgenden detaillierter betrachten.

Domänenspezifische Bedingungen
Domänenspezifische Bedingungen ergeben sich auf Grund der Logik (Funktionslogik) der zu testenden Fahrzeugfunktion. Sie beschreiben die Besonderheiten in der Bedienung oder Verwendung des Fahrzeugs und sind in der Regel fahrzeugspezifisch. In erster Linie finden sich in der EXAM-Bibliothek dazu Beispiele im Bereich des *stdAbstractCars*, welches dem EXAM-Anwender Operationen zur Verfügung stellt, um aus Sicht des Fahrers oder der Fahrzeugumgebung Aktionen auszuführen.

Beispiele sind:

- Ein Fahrzeug kann erst beschleunigt werden, wenn der Motor zuvor gestartet wurde und ein Gang eingelegt wurde.
- Die Funktion ACC kann erst auf ein Objekt (vorausfahrendes Fahrzeug) reagieren, wenn das ACC-System vom Fahrer aktiviert ist.
- Zum Starten des Motors muss die Zündung aktiviert sein. Dazu muss sich entweder ein Schlüssel im Zündschloss befinden oder bei der Verwendung des Komfortschlüssels muss der Schlüssel im Fahrzeuginnenraum erkannt werden.

Domänenübergreifende Bedingungen
Domänenübergreifende Bedingungen ergeben sich aus der Bedienlogik der benötigten Testhardware wie z. B. Datenlogger, Diagnosetester, Roboter, Kameras etc. Die Bedingungen sind nicht an ein Fahrzeugprojekt gebunden und sind nicht auf die Domäne der Absicherung von Fahrzeugfunktionen beschränkt. Beispiele finden sich dazu in nahezu allen Bibliotheksfunktionen zur Ansteuerung der Prüfstandshardware.

Beispiele sind:

- Starten einer Datenaufzeichnung mit dem Datenlogger ist erst nach der Initialisierung des Datenloggers möglich.
- Nach dem Start einer Signalaufzeichnung muss diese vor Ende des Testablaufes wieder gestoppt werden. Zusätzlich darf vor dem Stoppen der Aufzeichnung keine neue Aufzeichnung gestartet werden.
- Die Bildverarbeitung für ein Bild oder ein Video kann erst aktiviert werden, wenn zuvor ein Bild/Video aufgezeichnet wurde.
- Der Zugriff auf den Fehlerspeicher eines Steuergerätes mit Hilfe eines Diagnosetester kann erst nach der Initialisierung des Diagnosesystems erfolgen. Dazu muss in der Initialisierung das entsprechende Protokoll, das verwendete Bussystem und der Steuergerätetyp definiert werden.

Auf Modellierungsrichtlinien basierende Bedingungen
Neben der Beschreibung von Namenskonventionen enthalten die Modellierungsrichtlinien und die Vorgaben der Modellverantwortlichen auch *Design-Patterns* für die Modellierung der Testabläufe. Die Vorgaben definieren z. B. wie die Signalaufzeichnung von Messdaten durchzuführen ist oder wie zusätzlich benötigte Prüfhardware (Datenlogger, Diagnoseteste etc.) behandelt werden.

Beispiele sind:

- In jedem Testfall soll zu Beginn eine Operation zur Initialisierung aller benötigten Hardware aufgerufen werden und am Ende des Testablaufs ein Operation zur Deinitialisierung der Geräte. Diese beiden Operationen dürfen nur einmal pro Testfall aufgerufen werden, müssen aber zwingend enthalten sein.
- Die Konfiguration der Signalaufzeichnung soll in jedem Testfall, in dem eine Signalaufzeichnung genutzt wird, nur einmal durchgeführt werden.
- Die falsche Ausführungsreihenfolge oder die fehlende Aufteilung eines Testfalls in *Pre-Sequence* (Initialisierung des Prüfstands und von allen benötigten Komponenten), *Action-Sequence* (Durchführung der Stimulation des zu testenden Objekts) und *Post-Sequence* (Auswertung der Messdaten, Generierung des Testreports und Deinitialisierung aller verwendeten Komponenten).

Zur Verdeutlichung der Auswirkungen von falschen Aufrufreihenfolgen der Bibliotheksfunktionen betrachten wir zuerst den heutigen Prozess der Testfallerstellung- und ausführung in Abbildung 3.1. Der Fokus der Betrachtung liegt hierbei auf Modellierungsfehlern aus der Kategorie der domänenspezifischen und domänenübergreifenden Bedingungen, da die Modellierungsrichtlinien in der Regel keine Auswirkungen auf die Testausführung haben, sondern primär auf die Zusammenarbeit oder den Austausch der Testfälle.

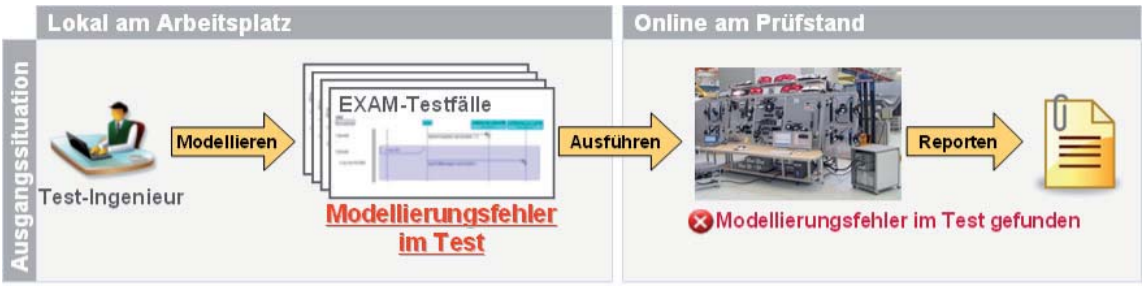

Abbildung 3.1: Fehler bei der Testausführung

Zu Beginn des Testprozesses modelliert der Test-Ingenieur die Testfälle mit Hilfe der EXAM-Funktionsbibliothek lokal am Arbeitsplatz. Nach der automatisierten Generierung des Testcodes bringt der Anwender die Testfälle am Prüfstand zur Ausführung und erhält im Anschluss die entsprechenden Testergebnisse und ein entsprechender Testreport wird erzeugt. Beachtet der Anwender bei der Modellierung der Testfälle die kausalen Beziehungen der verwendeten Bibliotheksfunktionen nicht, kann dies zu Problemen bei der Testausführung führen. In den meisten Fällen führt eine Nichtbeachtung der *domänenübergreifenden Bedingungen* zu einer Fehlermeldung der angesprochenen Prüfhardware oder der entsprechenden Schnittstellenimplementierungen.

Im Falle des zuvor beschriebenen Beispiels des Datenloggers, würde der Aufruf der Operation zum Start der Messung eine Exception zurückgeben, wenn das Gerät zuvor nicht entsprechend parametriert wurde. Der Vorteil dieser Art von Fehler ist, dass der Benutzer den Fehler direkt erkennen (der Fehler wird von dem Prüfstand in Form einer Exception angezeigt) kann und entsprechend den Modellierungsfehler im Testfall korrigieren kann. Jedoch wird auch in diesem Fall

Prüfstandszeit unnötig verwendet, da der Anwender den Fehler erst zur Laufzeit des Testfalls am Prüfstand bemerkt.

Die zweite Art von Fehlern wird in den meisten Fällen durch eine Verletzung einer *domänenspezifischen Bedingung* hervorgerufen. In diesem Fall erhält der Anwender in der Regel vom Laufzeitsystem während der Testausführung keine Fehlermeldung. Der Testfall wird ohne Unterbrechung durchgeführt und auch ein Testreport erzeugt. Ein Beispiel hierzu wäre der Versuch das Fahrzeug zu beschleunigen bevor die Zündung aktiviert ist. Der Prüfstand wird den Befehl zur Beschleunigung verarbeiten und entsprechend an das Echtzeitmodell für die Fahrzeugdynamik weiterleiten, jedoch hätte die Ausführung der Operation keine Auswirkungen auf das Fahrzeug. Die weiteren Stimulationen des Testablaufs gehen jetzt jedoch davon aus, dass das Fahrzeug eine Geschwindigkeit größer als 0 km/h hat und gehen daher von einer falschen Voraussetzung aus. Ohne die Überprüfung der Geschwindigkeit durch den Anwender während der Testausführung führt dies zu einer falschen Bewertung des Tests und zu fehlerhaften Einträgen im Testreport. Somit muss der Benutzer bei Abweichungen der Ergebnisse im Testreport überprüfen, ob der Fehler aufgrund eines gefundenen Fehlers der Steuergerätesoftware entstanden ist oder durch eine falsche Aufrufreihenfolge der EXAM-Operationen. Erst nach der Analyse des Testablaufes und der Testergebnisse kann der Anwender den Modellierungsfehler korrigieren.

Bei beiden Fehlerarten muss nach der Beseitigung der Modellierungsfehler der Testablauf erneut am Prüfstand zur Ausführung gebracht werden. Besonders kritisch ist dies, wenn der Modellierungsfehler an einer zentralen Stelle, z. B. der Konfiguration der Echtzeitaufzeichnung, innerhalb einer *TestSuite* eines Nachts- oder Wochenendlaufs enthalten ist. Dies führt dazu, dass alle in der *TestSuite* enthaltenen Testfälle mit einer fehlerhaft konfigurierten Echtzeitmessung arbeiten. Im schlimmsten Fall muss dann die gesamte *TestSuite* analysiert und erneut zur Ausführung gebracht werden. Ist der Fehler nicht direkt durch eine Fehlermeldung sichtbar, müssen zusätzlich zur Analyse des Fehlers die Testergebnisse mindestens zweimal vom Anwender bewertet werden. Einmal zur Lokalisierung des Fehlers im Testablauf und zum anderen nach der wiederholten Testausführung. Für die Bewertung der Testergebnisse wird aufgrund der Komplexität der Testfälle in etwa noch einmal die Hälfte der Testausführungszeit benötigt.

Im Hinblick auf die wachsenden Anforderungen und die limitierte Prüfstandszeit ist eine Reduzierung der Modellierungsfehler jedoch zwingend erforderlich und würde eine deutliche Verbesserung der Testeffizienz darstellen. Dazu ist es erforderlich, dass der Anwender die Möglichkeit hat, seine Testfälle schon lokal am Arbeitsplatz auf semantische Modellierungsfehler zu überprüfen und vor der Ausführung der Testabläufe am Prüfstand die Fehler zu korrigieren. Eine Simulation der Testausführung ist ohne den Prüfstand nur sehr schwer möglich, da die Testausführung vom Zustand des Prüfstandes während der Ausführung abhängig ist. Die Erstellung eines Verhaltens- und Zustandsmodells des gesamten Prüfstands für eine Simulation ist aufgrund der Komplexität der Prüfstände ebenso nicht wirtschaftlich.

Damit ohne Simulation der Testabläufe sichergestellt werden kann, dass während der Ausführung der Testfälle die Aufrufreihenfolge der Bibliotheksfunktionen nicht verletzt wird, müssen in jedem möglichen Ausführungspfad des Testfalls die kausalen Abhängigkeiten erfüllt sein. In Abhängigkeit der Parametrierung eines Testfalls und der aktuellen Prüfstandskonfigurationen

können bei der Testausführung verschiedene Ausführungspfade des Testfalls durchlaufen werden. Aufgrund der Vielzahl von Ausführungspfaden kann jedoch nicht jeder Pfad des Testablaufs einzeln manuell überprüft werden.

Wir stellen in dieser Arbeit einen vergleichbaren Ansatz wie für die Überprüfung der syntaktischen Modellierungsfehler in EXAM vor. Der Anwender wird dabei über semantische Modellierungsfehler in seinem Testablauf, welche auf einer Missachtung der kausalen Aufrufreihenfolge der Operationen der Funktionsbibliothek beruhen, vor der Ausführung am Prüfstand informiert. Voraussetzung für die Überprüfung der Testabläufe auf Einhaltung der Ausführungsreihenfolge der Operationen ist, dass die Methode eine formale Spezifikation der Abhängigkeiten zwischen den Operationen der Funktionsbibliothek ermöglicht. Die Überprüfung der Bedingungen muss möglichst unabhängig von der Anzahl der möglichen Ausführungspfade in den Testabläufen sein und für eine beliebige Anzahl von Bedingungen funktionieren. Ebenso muss eine einfache Anwendbarkeit der Methode für die EXAM-Anwender sichergestellt werden. Die verwendeten Darstellungs- und Bedienkonzepte sollten sich an den bestehenden Konzepten von EXAM orientieren.

In Abbildung 3.2 ist das prinzipielle Vorgehen der *Offline-Analyse* der Testabläufe dargestellt. Nach der formalen Spezifikation der Abhängigkeiten zwischen den Operationen (Bedingungen) modelliert der Test-Ingenieur seine Testabläufe. Vor der Ausführung der Testfälle am Prüfstand werden alle auszuführenden Testfälle auf Einhaltung der Bedingungen überprüft. Werden Verletzungen der Ausführungsreihenfolge im Testablauf erkannt, erhält der Anwender eine entsprechende Fehlermeldung und kann den Ablauf korrigieren. Werden keine Fehler erkannt, können die Testfälle am Prüfstand zur Ausführung gebracht werden.

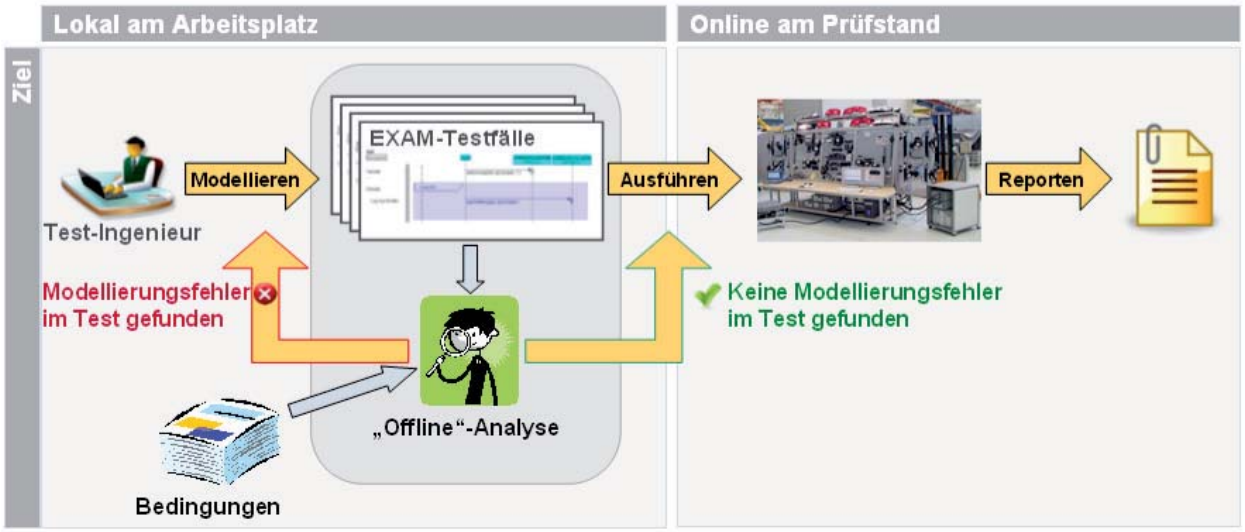

Abbildung 3.2: Analyse der Testfälle vor der Ausführung

Ziel der Arbeit ist

- die Entwicklung einer geeigneten Methode zur formalen Spezifikation und zur Überprüfung der kausalen Abhängigkeiten der Bibliotheksoperationen in EXAM vor der Ausführung der Testfälle am Prüfstand,
- die Integration der Methode in das Testautomatisierungssystem EXAM und
- die Validierung des praktischen Einsatzes der Methode im Testprozess der AUDI AG.

Wir beschreiben die Methode zur Überprüfung der Ablaufreihenfolge der EXAM-Operationen in **Kapitel 6** und demonstrieren die Anwendbarkeit mit Hilfe praktischer Beispiele in **Kapitel 7**. Die Herausforderungen sowie die Erfahrungen der praktischen Anwendung der Methode im Entwicklungsprozess bei der AUDI AG beschreiben wir in **Kapitel 8** im Detail. Zuvor werden in den beiden folgenden Kapiteln **Kapitel 4** und **Kapitel 5** die für das Verständnis der Methode benötigten Grundlagen beschrieben.

4 Mathematische Grundlagen

Das folgende Kapitel beschreibt die für das Verständnis der Arbeit notwendigen und hilfreichen Theorien und Konzepte. Hierbei verzichten wir auf eine ausführliche Darstellung und Beschreibung der zugehörigen Beweise und verweisen an entsprechender Stelle auf die zugehörige Fachliteratur. Nach einer Einführung in die Aussagenlogik stellen wir die wichtigsten Grundlagen der Petri-Netze dar. Im letzten Abschnitt des Kapitels betrachten wir den Gauss-Jordan-Algorithmus als ein Beispiel für ein Lösungsverfahren für lineare homogene Gleichungssysteme. Der Algorithmus wird im weiteren Verlauf der Arbeit zu Berechnung der T-Invarianten der Petri-Netze verwendet.

4.1 Aussagenlogik

Die zentralen Begriffe der Aussagenlogik ([Sch00], [HE01], [MW09]) sind *Aussage* und *Wahrheit*. Eine aussagenlogische Formel (Aussage) besteht aus *Aussagensymbolen* und *Junktoren*. Der *Wahrheitswert* einer aussagenlogischen Formel kann durch Interpretation der Symbole und Junktoren bestimmt werden.

Für die Definition der aussagenlogischen Formel setzen wir die Existenz einer endlichen Menge $\mathbb{A} = \{a, b, c, ...\}$ von *Aussagensymbolen* voraus. Weiterhin verwenden wir die *Junktorsymbole* $\neg$ (nicht), $\vee$ (oder), $\wedge$ (und), $\rightarrow$ (wenn, dann), $\leftrightarrow$ (genau dann, wenn), $\top$ (erfüllbar) und $\bot$ (unerfüllbar).

Definition 4.1 (Aussagenlogische Formel)
Für eine endliche Menge $\mathbb{A} = \{a, b, c, ...\}$ *von* Aussagensymbolen *ist die Menge* Form($\mathbb{A}$) *der aussagenlogischen Formeln wie folgt rekursiv definiert:*

- *Jedes Aussagensymbol* ($a \in \mathbb{A}$) *ist eine atomare aussagenlogische Formel.*
- *Die Junktorsymbole* $\top$ *und* $\bot$ *sind aussagenlogische Formeln.*
- *Ist* F *eine aussagenlogische Formel, so auch* $\neg F$.
- *Sind* F *und* G *aussagenlogische Formeln, so auch* $(F \vee G)$, $(F \wedge G)$, $(F \rightarrow G)$ *und* $(F \leftrightarrow G)$.

Bei komplexen Formeln werden Klammern zur eindeutigen Gruppierung der Formel verwendet. Redundante Klammern können auf Grund der Präzedenz der Operation in der Reihenfolge $\neg$ (nicht), $\vee$ (oder), $\wedge$ (und), $\rightarrow$ (wenn, dann), $\leftrightarrow$ (genau dann, wenn) entfallen.

Nach der Definition der Syntax der Aussagenlogik definieren wir im Folgenden die Semantik der Aussagenlogik.

Ist $\mathbb{D}$ eine Teilmenge der atomaren aussagenlogischen Formeln, so werden diese durch eine Funktion

$$B : \mathbb{D} \rightarrow \{w, f\} \tag{4.1}$$

auf die Wahrheitswerte *wahr* oder *falsch* abgebildet. Somit wird jeder atomaren Formel eine Belegung zugeordnet. Für die Zuordnung einer Belegung zu einer komplexeren Formel erweitern wir die Definition. In der Menge $\mathbb{E}$ sind alle Formeln enthalten, die aus atomaren Formeln aufgebaut sind ($\mathbb{D} \subseteq \mathbb{E} \subseteq \mathit{Form}(\mathbb{A})$). In der nachfolgenden Definition 4.2 erweitern wir die Funktion B zu einer Funktion

$$\widehat{B} : \mathbb{E} \rightarrow \{w, f\} \tag{4.2}$$

Definition 4.2 (Semantik der Aussagenlogik)
Eine Wahrheitsbelegung ist eine Abbildung $B : \mathbb{D} \rightarrow \{w, f\}$, wobei wir die Elemente w (wahr) und f ($falsch$) als Wahrheitswerte bezeichnen.

Für jede atomare Formel $A \in \mathbb{D}$ ist $\widehat{B}(A) = B(A)$.

Daraus ergibt sich die folgende semantische Definition für die Menge $\mathbb{E}$ der aussagenlogischen Formeln:

- $\widehat{B}(\top) = w$
- $\widehat{B}(\bot) = f$
- $\widehat{B}(\neg F) = \begin{cases} w, \textit{falls } \widehat{B}(F) = f \\ f, \textit{sonst} \end{cases}$
- $\widehat{B}(F \wedge G) = \begin{cases} w, \textit{falls } \widehat{B}(F) = w \textit{ und } \widehat{B}(G) = w \\ f, \textit{sonst} \end{cases}$
- $\widehat{B}(F \vee G) = \begin{cases} w, \textit{falls } \widehat{B}(F) = w \textit{ oder } \widehat{B}(G) = w \\ f, \textit{sonst} \end{cases}$
- $\widehat{B}(F \rightarrow G) = \begin{cases} f, \textit{falls } \widehat{B}(F) = w \textit{ und } \widehat{B}(G) = f \\ w, \textit{sonst} \end{cases}$
- $\widehat{B}(F \leftrightarrow G) = \begin{cases} w, \textit{falls } \widehat{B}(F) = \widehat{B}(G) \\ f, \textit{sonst} \end{cases}$

Nach der Definition einer Belegung für eine aussagenlogische Formel ist in der nachfolgenden Definition angegeben, wann eine Belegung zu einer Formel passend ist. In diesem Zusammenhang definieren wir die Begriffe *Modell*, *gültig* und *erfüllbar*.

Definition 4.3 (Modell, gültig und erfüllbar)
Sei F eine aussagenlogische Formel und B eine Belegung.

- *Wenn eine Belegung B für alle in der Formel F vorkommenden atomaren Formeln definiert ist, so heißt B zu F passend.*
- *Ist B eine zu F passende Belegung mit $B(F) = wahr$, dann ist B ein* Modell *von F ($B \vDash F$). Wenn $B(F) = falsch$, so ist B kein Modell der Formel F und wir schreiben $A \nvDash F$.*
- *Besitzt eine Formel F mindestens ein Modell, so ist F* erfüllbar. *Falls es kein Modell für F gibt ist F* unerfüllbar.
- *Ist jede zu der Formel F passende Belegung ein Modell von F, so heißt diese gültig (Tautologie) und wir schreiben $\vDash F$.*

Für den Vergleich zweier Formeln untereinander ist es notwendig, die Variablenmengen (Menge der enthaltenen atomaren Formeln) wechselseitig zu vervollständigen:

Definition 4.4 (Wechselseitige Vervollständigung)
Die Formeln F und G bestehen aus der Menge der Aussagensymbole $\mathbb{A}(F) \subseteq \mathbb{A}$ bzw. $\mathbb{A}(G) \subseteq \mathbb{A}$. Die Vervollständigung $C_G\{F\}$ von F bezüglich G ist die Formel

$$C_G\{F\} = F \wedge (a_1 \vee \overline{a_1}) \wedge ... \wedge (a_k \vee \overline{a_k})$$

für die Aussagensymbole $\{a_1, ..., a_k\} \in \mathbb{A}(G)$.

Die Vervollständigung $C_F\{G\}$ von G bezüglich F ergibt sich analog.

Nach dem Vergleich zweier aussagenlogischen Formeln definieren wir die Äquivalenz zweier Formeln über die Belegung der Formeln mit Wahrheitswerten wie folgt:

Definition 4.5 (Semantische Äquivalenz)
Die Formeln F, G heißen semantisch äquivalent, wenn für alle passenden Belegungen B der Formeln F und G gilt, dass

$$B(F) = B(G).$$

Dann schreiben wir $F \equiv G$.

Für viele Anwendungen der Aussagenlogik ist es notwendig, dass die Formeln in einer definierten Form vorliegen. Für z. B. die Digitaltechnik ist es wichtig, dass die einzelnen Teile der Formeln nur durch eine Konjunktion oder eine Disjunktion verknüpft sind. In der Definition 4.6 führen wir daher die beiden Normalformen aussagenlogischer Formeln ein.

Definition 4.6 (Normalformen)
Besteht eine Formel F nur aus Konjunktionen von Disjunktionen von atomaren Formeln, so ist sie in konjunktiver Normalform *(KNF). Besteht eine Formel F nur aus Disjunktionen von Konjunktionen von atomaren Formeln, so ist sie in* disjunktiver Normalform *(DNF).*

Hornformeln sind ein wichtiger Spezialfall von Formeln in der *konjunktiver Normalform*. *Hornformeln* sind wie folgt definiert:

Definition 4.7 (Hornformeln)
Eine Formel F ist eine Hornformel*, wenn F in* konjunktiver Normalform *ist und jede Disjunktion aus höchstens einem nicht negierten Aussagensymbol besteht.*

Ein Beispiel für eine Hornformel ist die Formel

$$(\neg a \vee \neg b \vee c) \wedge (a \vee \neg b \vee \neg d) \wedge (\neg c \vee \neg d),$$

wobei a, b, c und d atomare Formeln sind.

4.2 Petri-Netze

Petri-Netze werden für die Beschreibung, Simulation und Analyse von nebenläufigen Systemen verwendet. Carl Adam Petri führte die nach ihm benannten Netze 1962 als Bedingungs- und Ereignisnetze in [Pet62] ein.

Eine ausführliche Beschreibung der verschiedenen Netzklassen, deren Eigenschaften und Anwendungen geben z. B. [Bau96], [LP08], [Jen92], [Rei10] oder [Val03].

Im Folgenden beschränken wir uns auf die Einführung der wichtigsten Eigenschaften der Stellen-Transitions-Netze, die für das Verständnis der Arbeit notwendig sind.

4.2.1 Stellen-Transitions-Netze

Wir beginnen die Definition der Stellen-Transitions-Netze mit der Definition eines Netzes. Ein Netz besteht dabei aus einer Menge von Stellen, Transitionen und gerichteten Kanten zwischen diesen beiden Knotentypen.

Definition 4.8 (Netz) *Ein Netz ist ein 3-Tupel* (S, T, F) *mit,*

- S*: eine Menge von Stellen*
- T*: eine Menge von Transitionen*
- $S \cup T$*: die Menge der Knoten, wobei* $S \cap T = \emptyset$
- F*: eine Menge von gerichteten Kanten mit* $F \subseteq (S \times T) \cup (T \times S)$

In einem Stellen-Transitions-Netz ist jeder Stelle des Netzes eine Kapazität und jeder Kante ein Kantengewicht zugeordnet.

Definition 4.9 (Stellen-Transitions-Netz)
Ein Stellen-Transitions-Netz N *ist ein 5-Tupel* $N = (S, T, F, K, W)$ *mit:*

- (S, T, F) *ist ein Netz,*
- $K : S \to \mathbb{N} \cup \{\infty\}$ *ordnet jeder Stelle eine möglicherweise unbeschränkte Kapazität K zu,*
- $W : F \to \mathbb{N}$ *ordnet jeder Kante ein Kantengewicht zu.*

Ein Stellen-Transitions-Netz oder auch Petri-Netz genannt ist somit ein bipartiter Graph, also ein gerichteter Graph (*directed graph* oder kurz *digraph*) mit zwei unterschiedlichen Knotentypen (Stellen und Transitionen). Die gerichteten Kanten des Graphs verlaufen nur zwischen Stellen und Transitionen aber niemals zwischen Knoten vom gleichen Typ ([Sch02], [LP08], [Rei10]).

Jeder Knoten in dem Netz hat eine definierte Menge von Eingangs- und Ausgangsknoten.

Definition 4.10 (Eingangs- und Ausgangsknoten)
Sei (S, T, F, K, W) *ein Petri-Netz und ist* $x \in S \cup T$*, dann gilt:*

- ${}^\bullet x := \{y \in S \cup T \mid (y, x) \in F\}$ *die* Eingangsmenge *oder der* Vorbereich *von x und*
- $x^\bullet := \{y \in S \cup T \mid (x, y) \in F\}$ *die* Ausgangsmenge *oder der* Nachbereich *von x.*

Erweitert auf die ganze Knotenmenge bedeutet dies:
Ist $A \subseteq S \cup T$*, dann gilt:*

- ${}^\bullet A := \bigcup_{x \in A} {}^\bullet x$ *ist der* Vorbereich *von A und*
- $A^\bullet := \bigcup_{x \in A} x^\bullet$ *ist der* Nachbereich *von A.*

Ist die Anzahl der Elemente der Eingangsmenge von x größer als eins ($|{}^\bullet x| > 1$) wird x als *rückwärtsverzweigt* bezeichnet. Analog heißt x *vorwärtsverzweig*t, wenn ($|x^\bullet| > 1$) ist.

Das Netz wird als *schlicht* bezeichnet, wenn keine zwei Knoten im gesamten Netz denselben Vor- und Nachbereich haben und somit muss folgende Bedingung gelten:

Definition 4.11 (Schlicht)
Ein Netz $N = (S, T, F)$ *ist* schlicht*, wenn gilt:*

$$\forall x, y \in S \cup T : {}^\bullet x = {}^\bullet y \wedge x^\bullet = y^\bullet \Rightarrow x = y$$

Ein Teilnetz N' von N besteht aus einer Teilmenge der Stellen, Transitionen und Kanten von N und ist wie folgt definiert:

Definition 4.12 (Teilnetz)
Ein Netz $N' = (S', T', F')$ ist ein Teilnetz eines Netzes $N = (S, T, F)$, wenn folgendes gilt:

- $S' \subseteq S, T' \subseteq T$ *und*
- $F' = F \cap ((S' \times T') \cup (T' \times S'))$.

Nachdem wir Netzgraphen eingeführt haben, definieren wir nun im folgenden Stellen-Transitions-Systeme und stellen wichtige Prinzipien der Systeme, wie die Erreichbarkeitsanalyse, der Zusammenhang mit der Linearen Algebra und Aspekte der Nebenläufigkeit vor.

Definition 4.13 (Stellen-Transitions-System)
Ein Stellen-Transitions-System (markiertes Stellen-Transition-Netz) ist ein 6-Tupel $Y = (S, T, F, K, W, M_0)$ mit

- *einem Stellen-Transitions-Netz $N = (S, T, F, K, W)$ und*
- *der Anfangsmarkierung $M_0 : S \to \mathbb{N}_0$, wobei $\forall s \in S : M_0(s) \leq K(s)$.*

In der graphischen Darstellung eines Stellen-Transitions-Systems werden die Stellen als Kreise, die Transitionen als Rechtecke und die Kanten als gerichtete Pfeile dargestellt. Die Marken (Token), welche die Markierung der Stellen repräsentieren, werden als schwarze Kreise dargestellt. Die Anzahl der Tokens auf einer Stelle wird durch die entsprechende Anzahl an schwarzen Kreisen auf dieser Stelle verdeutlicht.

Abbildung 4.1 auf Seite 53 zeigt die graphische Darstellung eines Stellen-Transitions-Systems mit der Anfangsmarkierung $M_0 = \{(s_1, 1), (s_2, 0), (s_3, 1)\}$.

Der zugehörige Netzgraph $N = (S, T, F)$ des abgebildeten S/T-Systems besteht aus:

- den Stellen $S = \{s_1, s_2, s_3\}$,
- den Transitionen $T = \{s, a, b, c, g\}$ und
- den Kanten $F = \{(s, s_1), (s_1, a), (a, s_2), (s_2, b), (s_2, c), (b, s_3), (c, s_3), (s_3, g)\}$.

Die Kapazitäten

$$\mathrm{K} = \{(s_1, \infty), (s_2, \infty), (s_3, \infty)\}$$

der Stellen und die Kantengewichte

$$\mathrm{W} = \{((s, s_1), 1), ((s_1, a), 1), ((a, s_2), 1), ((s_2, b), 1), ((s_2, c), 1), ((b, s_3), 1), ((c, s_3), 1), ((s_3, g), 1)\}$$

der Kanten des S/T-Systems N=(S,T,F,K,W) sind in der Darstellung nicht abgebildet.

Nach allgemein üblichen Konventionen (vergleiche z. B. [Bau96], [LP08], [Jen92], [Rei10] oder [Val03]) haben Stellen bei fehlender Angabe eine Kapazität von ∞ und Kanten ein Kantengewicht von eins.

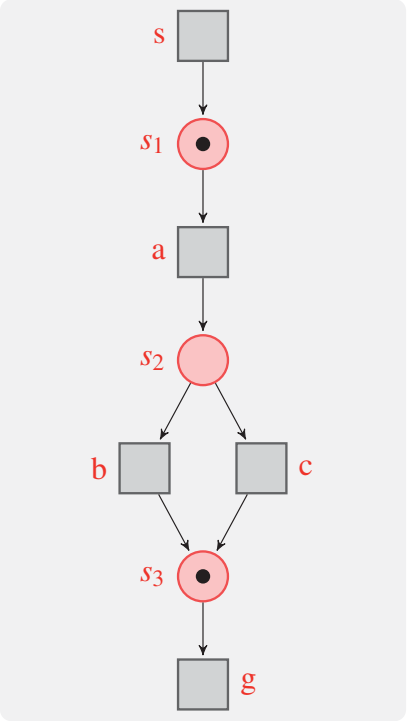

Abbildung 4.1: Stellen-Transitions-System

4.2.2 Dynamik der S/T-Systeme

Im Folgenden ist $N = (S, T, F, K, W)$ ein S/T-Netz und $Y = (N, M_0)$ das zugehörige S/T-System mit der Anfangsmarkierung M_0. In der Definition 4.14 beschreiben wir, wann eine Transition *aktiviert* ist.

Definition 4.14 (Aktivierte Transition)
Sei $N = (S, T, F, K, W)$ ein S/T-Netz und $Y = (N, M_0)$ das zugehörige S/T-System mit der Anfangsmarkierung M_0. Eine Transition $t \in T$ ist aktiviert *unter einer Markierung M, geschrieben $M[t\rangle$, wenn*

- $\forall s \in {}^\bullet t : M(s) \geq W((s,t))$,
- $\forall s \in t^\bullet : M(s) \leq K(s) - W(t,s)$.

Ein Transition ist *aktiv*, wenn die jeweilige Anzahl der Tokens auf den Vorstellen der Transition größer ist als das Kantengewicht der Kanten zwischen den Stellen und der Transition. Für alle Nachstellen der Transition gilt, dass die Anzahl der Tokens auf den Nachstellen nach dem Schalten der Transition nicht größer sein darf als die jeweilige Kapazität der Stellen. Hierbei ist das Kantengewicht der Kanten zwischen der Transition und der jeweiligen Nachstelle zu beachten.

Mit Hilfe der Definition 4.14 führen wir nun die *Schaltregel*, also unter welchen Bedingungen eine Transition schalten kann ein.

Definition 4.15 (Schaltregel)
Sei $N = (S, T, F, K, W)$ ein S/T-Netz und $Y = (N, M_0)$ das zugehörige S/T-System mit der Anfangsmarkierung M_0.

Ist eine Transition t unter der Markierung M aktiviert, so kann t schalten. Durch Entnahme von Tokens aus den Eingangsstellen von t und Ablage der Tokens auf den Nachstellen der Transition t entsteht aus M die Folgemarkierung M' unter Berücksichtigung der Kantengewichte. Wir schreiben dafür $M[t\rangle M'$. Dabei ergibt sich die neue Markierung M' wie folgt:

$$M'(s) := \begin{cases} M(s) - W((s,t)) & \text{wenn } s \in {}^\bullet t \\ M(s) + W((t,s)) & \text{wenn } s \in t^\bullet \\ M(s) - W((s,t)) + W((t,s)) & \text{wenn } s \in {}^\bullet t \cap t^\bullet \\ M(s) & \text{sonst} \end{cases}$$

Die aktuelle Markierung eines S/T-Netzes und seine aktivierten Transitionen können mit Hilfe von Spaltenvektoren dargestellt werden. Eine Markierung M des Netzes N wird durch einen nicht negativen Stellenvektor $V : S \to \mathbb{N}_0$, dessen Index die Elemente der Menge S sind dargestellt, wobei $M(s) \leq K(s)$ für alle $s \in S$ gilt. Hierbei gibt $M(s)$ die Anzahl der Token auf der Stelle $s \in S$ unter der Markierung M an. Die Menge aller Markierungen eines S/T-Netzes wird mit $\mathbb{M}_N$ angegeben. Die aktivierten Transitionen werden durch einen nicht negativen Transitionsvektor mit $U : T \to \mathbb{N}_0$ dargestellt, wobei der Index des Vektors die Elemente der Menge T sind.

Der Stellenverktor für die Anfangsmarkierung M_0 des in Abbildung 4.1 auf Seite 53 dargestellten S/T-System enthält die folgenden Elemente:

$$M_0 = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \begin{bmatrix} s1 \\ s2 \\ s3 \end{bmatrix} \tag{4.3}$$

Der zugehörige Transitionsvektor T gibt an, welche Transition unter der Markierung M_0 aktiviert sind.

$$T = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} \begin{bmatrix} s \\ a \\ b \\ c \\ g \end{bmatrix} \tag{4.4}$$

Die Transition a ist in diesem Beispiel aktiviert, da ${}^\bullet a = \{s_1\}$ und $M_0(s_1) = W(s_1, a) = 1$. Weiterhin ist $a^\bullet = \{s_2\}$ und $M_0(s_2) = 0 \leq K(s_2) - W(a, s_2)$. Somit kann a schalten und wir erhalten die Folgemarkierung M_1, die im Folgenden als Stellenvektor dargestellt ist:

$$M_1 = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} \begin{bmatrix} s1 \\ s2 \\ s3 \end{bmatrix} \tag{4.5}$$

Ausgehend von der Definition 4.15 definieren wir nun die Schaltfolgen.

Definition 4.16 (Schaltfolgen)
Sei $Y = (N, M_0)$ ein S/T-System und $t_1, t_2 \ldots t_n \in T$ die Transitionen des Netzes.

Wir bezeichnen $w = t_1 ... t_n$ als Schaltfolge, wenn Markierungen $M_0, M_1, \ldots, M_n$ existieren, so dass gilt:

$$M_0[t_1\rangle M_1[t_2\rangle M_2 \ldots [t_n\rangle M_n.$$

Für die angegebene Schaltfolge schreiben wir kurz

$$M_0[w\rangle M_n$$

In der folgenden Definition betrachten wir die erreichbaren Markierungen eines Stellen-Transitions-Systems.

Definition 4.17 (Erreichbare Markierungen)
Sei $Y = (N, M_0)$ ein S/T-System.

Eine Markierung M_x, die von einer Markierung M aus erreichbar (d.h. $\exists\ w\ :\ M[w\rangle M_x$) ist, wird als erreichbare Markierung bezeichnet. Die Menge aller erreichbaren Markierungen $[M_x\rangle$ ausgehend von der Markierung M_x ist wie folgt definiert:

$$M_0 \in [M_0\rangle$$
$$M_x \in [M_0\rangle \wedge\ M_x[t\rangle M'_x \Rightarrow M'_x \in [M_0\rangle$$

Nach der Definition von erreichbaren Markierungen, geben wir in der folgenden Definition an, wann eine Transition *lebendig* oder *tot* ist.

Definition 4.18 (Lebendige und tote Transitionen)
Sei $N = (S, T, F, K, W)$ ein Stellen-Transitions-Netz mit der Anfangsmarkierung M_0.

Ein Transition t ist lebendig *unter der Markierung M, wenn*

$$\forall M \in [M\rangle \exists M' \in [M\rangle : M'[t\rangle$$

Ein Transition t ist tot *unter der Markierung M_0, wenn*

$$\forall M \in [M\rangle : \neg M[t\rangle$$

Die Definition 4.19 beschreibt, wann eine Markierung *lebendig*, *schwach lebendig* oder *tot* ist.

Definition 4.19 (Lebendige, schwach lebendige und tote Markierung)
Sei $N = (S, T, F, K, W)$ ein Stellen-Transitions-Netz mit der Anfangsmarkierung M_0.

Ein Markierung M ist lebendig*, wenn*

$$\forall t \in T : \ t \text{ ist lebendig unter } M$$

Ein Markierung M ist schwach lebendig oder deadlockfrei*, wenn*

$$\forall M \in [M\rangle \exists t \in T : M[t\rangle$$

Ein Markierung M ist tot*, wenn*

$$\neg \exists t \in T : M[t\rangle$$

Die Schaltfolgen können auch mit Hilfe eines Häufigkeitsvektors bzw. *Parikh-Vektors* angegeben werden. Hierbei beschreibt der Parikh-Vektor die Häufigkeit des Vorkommens der Transitionen in der Schaltfolge.

Definition 4.20 (Parikh-Vektor)
Ist T die Menge der Transitionen und w eine Schaltfolge über der Transitionsmenge. Dann ist $\Psi(w) : T \to \mathbb{N}$ die Parikh-Abbildung, die jeder Transition aus T die Anzahl ihres Vorkommens in w zuordnet. Das Symbol # definiert die Anzahl der möglichen Schaltfolgen.

Die Schaltfolge w kann durch den Parikh-Vektor (Häufigkeitsvektor) wie folgt dargestellt werden:

$$\overline{w} = \begin{pmatrix} \#(t_1, w) \\ \vdots \\ \#(t_n, w) \end{pmatrix}$$

Eine mögliche Schaltfolge w für das Netz in Abbildung 4.1 auf Seite 53 ist

$$w = (a\ g\ b\ g\ s). \tag{4.6}$$

Der zugehörige Häufigkeitsvektor enthält somit die folgenden Elemente:

$$\overline{w} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 2 \end{pmatrix} \begin{bmatrix} s \\ a \\ b \\ c \\ g \end{bmatrix} \tag{4.7}$$

4.2.3 Petri-Netze und Lineare Algebra

Mit Hilfe von Inzidenzmatrizen (siehe Definition 4.21) können wir die Netze mit den Mitteln der Linearen Algebra mathematisch beschreiben. Die Matrixdarstellung entspricht eineindeutig dem zugehörigen Netz, wenn das Netz keine Schlingen enthält. Schlingen sind Paare von Stellen s und Transition t, wobei sowohl (s,t) in F als auch (t,s) in F vorkommen.

Definition 4.21 (Inzidenzmatrix)
Sei $N = (S, T, F, K, W)$ ein S/T-Netz. Die Inzidenzmatrix von N ist eine Matrix $[N] : S \times T \to \mathbb{Z}$, wobei die Zeilen mit den Stellen (S) und die Spalten mit den Transitionen (T) des S/T-Netzes beschriftet sind.

Die Matrix ist wie folgt definiert:

$$[N](s,t) := \begin{cases} -W((s,t)) & \text{wenn } s \in {}^\bullet t \setminus t^\bullet \\ +W((t,s)) & \text{wenn } s \in t^\bullet \setminus {}^\bullet t \\ -W((s,t)) + W((t,s)) & \text{wenn } s \in {}^\bullet t \cap t^\bullet \\ 0 & \text{sonst} \end{cases}$$

Jede Position der Matrix (i, j) gibt an, wie sich die aktuelle Markenzahl auf der zugehörigen Stelle s_i durch das Schalten der zugehörigen Transition t_j ändert.

Die Darstellung der S/T-Netze als Inzidenzmatrix erlaubt es, strukturelle Merkmale (statische Merkmale) der Netze durch Lösen linearer homogener Gleichungssysteme zu berechnen. Die Merkmale werden als *S-Invarianten* (siehe Definition 4.22) bzw. *T-Invarianten* (siehe Definition 4.23) bezeichnet, welche in [Lau73] das erste Mal beschrieben wurden. Unabhängig von der aktuellen Markierung beschreiben *S-Invarianten* jene Stellenmengen, deren gewichtete Tokenanzahl konstant bleibt. Schalten alle Transitionen mit der in einer *T-Invarianten* angegebenen Häufigkeit, so wird die aktuelle Markierung reproduziert.

Definition 4.22 (S-Invariante)
Eine S-Invariante eines S/T-Netzes N ist ein Spaltenvektor $s : S \to \mathbb{Z}$, welcher eine Lösung des linearen Gleichungssystems $[N]^T * s = 0^T$ ist. $[N]^T$ ist hierbei die transponierte Inzidenzmatrix des S/T-Netzes N.

Definition 4.23 (T-Invariante)
Eine T-Invariante eines S/T-Netzes N ist der Spaltenvektor $t : T \to \mathbb{Z}$, welcher eine Lösung des linearen Gleichungssystems $[N] * t = 0$ ist.

Die beiden Typen von Invarianten werden in Form eines Vektors dargestellt. Alle Transitionen mit einem positiven Eintrag (> 0) in einer T-Invarianten bilden ein Subnetz N' von N, welches von der T-Invariante überdeckt wird. Dabei enthält das Subnetz genau die Transitionen, die einen positiven Eintrag in der T-Invarianten besitzen.

Die Invarianten bilden die Grundlage für die Analyse der dynamischen Eigenschaften der Netze, wie z. B. die Lebendigkeit. Weiterhin können auch Verklemmungen und Fallen in den Netzen bestimmt werden (vergleiche [LR94]).

Die Invarianten werden als kanonisch bezeichnet, wenn der größte gemeinsame Teiler der Elemente gleich eins ist ([Val03]). Weiterhin ist die Linearkombination von S-Invarianten bzw. T-Invarianten wiederum eine S-Invariante bzw. T-Invariante des S/T-Netzes, da diese wiederum eine Lösung des entsprechenden Gleichungssystems ist.

Für verschiedene praktische Anwendungen ist die Frage, ob die aktuelle Markierung eines Petri-Netzes reproduziert werden kann von Interesse. Nach [Lau92] kann die leere Markierung wiederhergestellt werden, falls das Netz mindestens eine T-Invariante besitzt. Dabei darf das entsprechende Subnetz, welches von der Invariante überdeckt ist, keine Verklemmungen oder Fallen enthalten (siehe [Bau96]).

Betrachten wir als Beispiel für den Nachweis der Reproduzierbarkeit der leeren Markierung das Netz mit leerer Markierung in Abbildung 4.2.

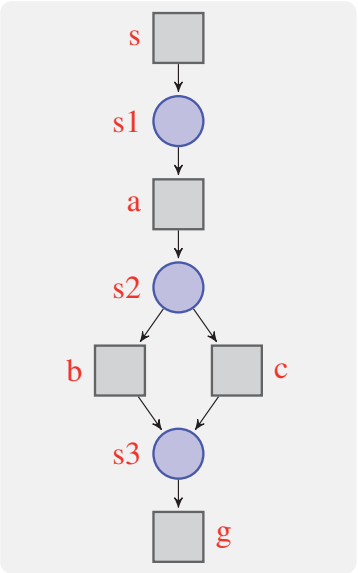

Abbildung 4.2: S/T-Netz mit leerer Markierung

Auch wenn in diesem Beispiel die beiden T-Invarianten direkt ersichtlich sind, wollen wir diese mit Hilfe des homogenen linearen Gleichungssystems $[N] * t_i = 0$ berechnen. Die zugehörige Inzidenzmatrix $[N]$ des abgebildeten Netzes hat folgende Form:

$$[N] = \begin{array}{c} \\ s1 \\ s2 \\ s3 \end{array} \begin{array}{c} \begin{array}{ccccc} s & a & b & c & g \end{array} \\ \begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ 0 & 1 & -1 & -1 & 0 \\ 0 & 0 & 1 & 1 & -1 \end{pmatrix} \end{array} \tag{4.8}$$

Als Lösung des Gleichungssystems ergeben sich die beiden Lösungsvektoren t_1 (siehe 4.9) und t_2 (siehe 4.10). Die sich aus den T-Invarianten ergebenen Subnetze enthalten weder eine Verklemmung noch eine Falle. Somit kann die leere Markierung des Netzes durch die beiden Schaltfolgen (s, a, b, g) und (s, a, c, g) reproduziert werden.

$$t_1 = \begin{pmatrix}1\\1\\1\\0\\1\end{pmatrix}\begin{bmatrix}s\\a\\b\\c\\g\end{bmatrix} \tag{4.9}$$

$$t_2 = \begin{pmatrix}1\\1\\0\\1\\1\end{pmatrix}\begin{bmatrix}s\\a\\b\\c\\g\end{bmatrix} \tag{4.10}$$

4.3 Lösungsverfahren für lineare Gleichungssysteme

Für die Berechnung von S- und T-Invarianten eines Petri-Netzes muss das sich aus der Inzidenzmatrix $[N]$ ergebende lineare homogene Gleichungssystem für S-Invarianten $[N]^t * t_i = 0$ bzw. für T-Invarianten $[N] * t_i = 0$ gelöst werden. Hierbei ist zu beachten, dass nur ganzzahlige positive Lösungen von Interesse sind, da die Transitionen entweder schalten oder nicht schalten können bzw. eine Stelle entweder eine positive Anzahl an Marken besitzen kann oder keine Marke.

Im Folgenden wird der Gauss-Jordan-Algorithmus zur Lösung der Gleichungssysteme vorgestellt und die Anzahl der benötigten Rechenoperationen untersucht.

4.3.1 Gauss-Jordan-Algorithmus

Das Lösungsverfahren für lineare Gleichungssysteme nach Gauss-Jordan basiert darauf, eine gegebene (m, n)-Matrix durch die Anwendung verschiedener Zeilenumformungen, welche die Lösung der Gleichungssysteme nicht verändern (siehe [NEBE05] oder [Fis02]), auf Zeilennormalform zu bringen. Anhand der transformierten Matrix kann die Lösung direkt oder durch Rücksubstitution bestimmt werden.

Die Eigenschaften einer Matrix in Zeilennormalform sind in der Definition 4.24 angegeben (vergleiche [NEBE05]).

Definition 4.24 (Zeilennormalform)
Eine (m,n) Matrix

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}$$

ist in Zeilennormalform, wenn gilt:

1. *Die Matrix enthält unterhalb der Diagonalen nur Nullen, also $a_{i,j} = 0$ für alle $i > j$.*
2. *Das erste von Null verschiedene Element jeder Zeile ist gleich eins.*
3. *Ist a_{ij} das erste von Null verschiedene Element der Zeile i, so gilt für alle andern Zeilen $(k \neq i)$ $a_{kj} = 0$.*

Somit stehen in der Spalte j unterhalb und oberhalb des ersten von Null verschiedenen Elements nur Nullen.

Als Beispiel einer Matrix in Zeilennormalform betrachten wir die folgende Matrix M, wobei die Variablen $x_1, x_2 \dots, x_7$ beliebige Werte annehmen können.

$$M = \begin{pmatrix} 0 & 1 & 0 & 0 & x_1 & 0 & x_2 \\ 0 & 0 & 1 & 0 & x_3 & 0 & x_4 \\ 0 & 0 & 0 & 1 & x_5 & 0 & x_6 \\ 0 & 0 & 0 & 0 & 0 & 1 & x_7 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{4.11}$$

Durch die Anwendung der im Gauss-Jordan-Algorithmus beschriebenen elementaren Zeilenumformungen kann jede (m, n)-Matrix in Zeilennormalform transformiert werden. Der Algorithmus kann dabei in zwei Teile eingeteilt werden. Die drei Schritte des ersten Teils (siehe Algorithmus 4.1) stellen sicher, dass unterhalb der Diagonalen der Matrix nur Nullen vorkommen. Die beiden Schritte des zweiten Teils des Algorithmus (siehe Algorithmus 4.2) sorgen dafür, dass das erste von Null verschiedene Element einer jeden Zeile gleich eins ist und ober- und unterhalb dieses Elements nur Nullen vorkommen.

Algorithmus 4.1 (Gauss-Jordan-Algorithmus Teil 1)
Die Schritte 1 bis 3 des G.-J.-Algorithmus werden jeweils auf den Teil der Matrix wiederholt angewendet, der durch Streichen der aktuellen ersten Zeile und Spalte entsteht.

1. *Schritt: In der ersten Spalte wird das erste von Null verschiedene Element gesucht und durch Zeilenvertauschung mit der ersten Zeile an die erste Position gebracht. Existiert in der ersten Spalte kein von Null verschiedenes Element, wird der Algorithmus beginnend bei der nächsten Spalte wiederholt.*

2. *Schritt: Ist das erste Element der ersten Zeile von eins verschieden, so dividiere die erste Zeile durch das erste Element. Somit ist nun das erste Element eine Eins.*

3. *Schritt: Unterhalb des ersten von Null verschiedenen Elements in der Spalte müssen alle Werte gleich Null sein. Dazu subtrahiere von jeder weiteren Zeile, dessen erstes Element ungleich Null ist, die mit dem ersten Element dieser Zeile multiplizierte erste Zeile.*
   Daraus ergibt sich der Vektor v_1, für die erste Spalte der Matrix:

$$v_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

Nach der Anwendung der ersten drei Schritte des Algorithmus hat die ursprüngliche Matrix (siehe Definition 4.24) die Gestalt:

$$M = \begin{pmatrix} a_{11} & x_1 & x_2 & \dots & \dots & \dots & x_e \\ 0 & a_{22} & x_f & \dots & \dots & \dots & x_g \\ 0 & 0 & a_{33} & \ddots & \dots & \dots & x_h \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 0 & a_{mn-2} & x_{n-1} & x_n \end{pmatrix} \tag{4.12}$$

Hierbei können die Variablen $x_1, x_2 \dots, x_n$ beliebige verschiedene Werte annehmen.

Nachdem unterhalb der Diagonalen alle Werte gleich Null sind, werden nun für die Erstellung der Zeilennormalform die beiden verbleibenden Schritte des Algorithmus angewendet.

Algorithmus 4.2 (Gauss-Jordan-Algorithmus Teil 2)
Die Schritte 4 und 5 des G.-J.-Algorithmus werden auf alle Zeilen der bisher erzeugten Matrix beginnend mit der zweiten Zeile angewendet.

4. *Schritt: In jeder Zeile muss das erste von Null verschiedene Element gleich Eins sein. Ist es ungleich Eins, dividiere die Zeile durch den Wert dieses Elements.*

5. *Schritt: In der Spalte oberhalb des ersten von Null verschiedenen Elements sollen in jeder Zeile nur Nullen stehen. Ist a_{ij} das erste von Null verschiedene Elemente der Zeile und sind a_{kj} die anderen Elemente der j-ten Spalte, so subtrahiere das a_{kj}-fache der i-ten Zeile von der Zeile k, wobei $k \neq i$ ist.*

Nach der Anwendung aller Schritte des Algorithmus hat die ursprüngliche Matrix nun Zeilennormalform:

$$M = \begin{pmatrix} 1 & 0 & 0 & \dots & \dots & \dots & x_1 \\ 0 & 1 & 0 & \dots & \dots & \dots & x_2 \\ 0 & 0 & 1 & 0 & \dots & \dots & x_3 \\ \vdots & \vdots & \ddots & \ddots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & 0 & 1 & x_{n-1} & x_n \end{pmatrix} \tag{4.13}$$

Hierbei können die Variablen $x_1, x_2 \dots, x_n$ wieder beliebige Werte annehmen.

Eine ausführliche Beschreibung des Algorithmus und viele Beispiele geben [NEBE05], [Fis02], [Mei11] und [Ant98].

Für die Berechnung der T-Invarianten eines Petri-Netzes ist es entscheidend, dass die Lösungen des linearen homogenen Gleichungssystems ganzzahlig und positiv sind. Aufgrund der Struktur der in dieser Arbeit verwendeten Petri-Netze eignet sich der vorgestellte G.-J.-Algorithmus zur Berechnung der T-Invariante der Netze.

4.3.2 Benötigte Rechenoperationen

Für die Bewertung des praktischen Einsatzes des Gauss-Jordan-Algorithmus zur Berechnung der *T-Invarianten* ist die Anzahl der benötigten Rechenoperationen zur Berechnung der Lösungsvektoren eines linearen homogenen Gleichungssystems von Bedeutung. Mit Hilfe der Anzahl der benötigten Rechenoperationen kann die von einem Computer benötigte Lösungszeit abgeschätzt werden.

In [Ant98] und [Mei11] wird die benötigte Ausführungszeit zur Lösung einer $n \times n$ Matrix, auf der Basis der benötigten Rechenzeit für Gleitkommaoperationen hergeleitet. Die Anzahl an Gleitkommaoperationen, die eine CPU pro Sekunde ausführen kann, wird als Floating-Point-Operations-Per-Second (FLOPS) bezeichnet. Basierend auf den Angaben der CPU-Hersteller beträgt die durchschnittliche Anzahl der tatsächlich nutzbaren Gleitkommaoperationen einer aktuellen CPU (z. B. *Intel© Core™ i5* oder *i7* Prozessor) pro Sekunde für eine Division bzw. Multiplikation in etwa 6.000 MFLOPS ($6.000 * 10^6$ FLOPS). Dies entspricht in etwa einer Ausführungs-

zeit für eine Operation von $1,67 * 10^{-10}$ Sekunden. Die durchschnittliche Anzahl an Additionen bzw. Subtraktionen benötigt in etwa 9.000 MFLOPS ($9.000 * 10^6$ FLOPS). Dies entspricht einer Ausführungszeit pro Operation von rund $1.1 * 10^{-10}$ Sekunden.

Als Einschränkung für seine Betrachtung geht Howard Anton in [Ant98] davon aus, dass die Matrix ohne Zeilenvertauschungen in die Zeilennormalform transformiert werden kann. Als Begründung gibt er die im Verhältnis zu der Ausführungszeit der arithmetischen Operationen wesentlich geringere Ausführungszeit von Zeilenvertauschungen an.

Für eine $n \times n$-Matrix ergeben sich aus der Betrachtung von Anton:

Benötigte Anzahl an Additionen:

$$\frac{1}{3}n^3 + \frac{1}{2}n^2 - \frac{5}{6}n \tag{4.14}$$

Benötigte Anzahl an Multiplikationen:

$$\frac{1}{3}n^3 + n^2 - \frac{1}{3}n \tag{4.15}$$

In der Praxis ist es notwendig, Gleichungssysteme mit mehreren hundert Gleichungen zu lösen. Hierbei können die Polynome aus (4.14) und (4.15) bei großen n durch den Summanden mit dem höchsten Exponenten approximiert werden. In diesem Fall ergeben sich für die ungefähre Anzahl an Rechenoperationen bei großen $n \times n$-Matrizen die folgenden Abschätzungen:

Benötigte Anzahl an Additionen:

$$\approx \frac{n^3}{3} \tag{4.16}$$

Benötigte Anzahl an Multiplikationen:

$$\approx \frac{n^3}{3} \tag{4.17}$$

Auf diesen Abschätzungen basieren die in Tabelle 4.1 dargestellten maximalen Transformationszeiten für Matrizen unterschiedlicher Größe. Die Werte basieren auf den zuvor angenommenen Ausführungszeiten der arithmetischen Operationen.

Durch den direkten Vergleich der Ergebnisse der beiden Abschätzungen für eine unterschiedliche Anzahl an Gleichungen pro Gleichungssystem wird deutlich, dass die Approximation durch den höchsten Exponenten für die Betrachtung der Laufzeiten ausreichend ist. Die angegebenen Werte sind entsprechend auf zwei bzw. drei Nachkommastellen gerundet.

Bei rund 5.000 verschiedenen Gleichungen im Gleichungssystem bewegt sich die Ausführungszeit im Sekundenbereich, wohingegen schon bei 40.000 Gleichungen die benötigte Zeit größer als eine Stunde ist.

Neben der Größe der Matrix wird die benötigte Lösungszeit dadurch bestimmt, ob die Matrix stark bzw. schwach besetzt ist. Von einer schwach besetzten Matrix wird gesprochen, wenn die

Mehrzahl der Koeffizienten gleich Null ist. Bei einer stark besetzten Gleichung sind die Mehrzahl der Koeffizienten dagegen ungleich Null. Bei einer schwach besetzten Matrix besitzt diese bereits einige Eigenschaften der Zeilennormalform und somit reduziert sich die benötigte Ausführungszeit des Gauss-Jordan-Algorithmus entsprechend. Grund dafür ist, dass weniger Rechenoperationen notwendig sind.

Anzahl der Gleichungen n	Rechenzeit der Additionen in Sekunden nach Formel 4.14	Rechenzeit der Multiplikationen in Sekunden nach Formel 4.15	Gesamt Rechenzeit in Sekunden nach Formel 4.14 und Formel 4.15	Rechenzeit der Additionen in Sekunden nach Formel 4.16	Rechenzeit der Multiplikationen in Sekunden nach Formel 4.17	Gesamt Rechenzeit in Sekunden nach Formel 4.16 und Formel 4.17)
50	$5 * 10^{-6}$	$7 * 10^{-6}$	$12 * 10^{-6}$	$5 * 10^{-6}$	$7 * 10^{-6}$	$12 * 10^{-6}$
100	$38 * 10^{-6}$	$57 * 10^{-6}$	$95 * 10^{-6}$	$37 * 10^{-6}$	$56 * 10^{-6}$	$93 * 10^{-6}$
500	$4,6 * 10^{-3}$	$7 * 10^{-3}$	$11,6 * 10^{-3}$	$4,6 * 10^{-3}$	$6,9 * 10^{-3}$	$11,5 * 10^{-3}$
1000	0,037	0,055	0,092	0,037	0,055	0.092
5000	4,63	6,95	11,58	4,62	6,943	11,57
10.000	37,04	55,57	92,61	37,037	55,55	92,59
20.000	296	444	740	296	444	740
40.000	2.370	3.556	5.926	2.370	3.555	5.925

Tabelle 4.1: Ausführungszeiten des Gauss-Jordan-Algorithmus (Zeitangaben in Sekunden)

Bei der Anwendung des Gauss-Jordan-Algorithmus zur Berechnung der *T-* bzw. *S-Invarianten* eines S/T-Netz können sich die angegebenen Ausführungszeiten in Abhängigkeit der Netzstruktur deutlich reduzieren. Als Beispiel betrachten wir dazu das bereits eingeführte S/T-Netz in Abbildung 4.2 auf Seite 58 und der zugehörigen Inzidenzmatrix 4.8 auf Seite 58. Zum Erreichen der Zeilennormalform ist in dieser Matrix nur noch Schritt 5 des Algorithmus 4.2 anzuwenden, da in jeder Zeile das erste Element bereits gleich eins ist und unterhalb dieser Elemente alle Werte gleich Null sind. Bei großen Petri-Netzen mit einer einfachen Netzstruktur ergeben sich noch weitere Reduktionen der benötigten Ausführungszeit bei der Lösung der linear homogenen Gleichungssysteme, da die Anzahl der Elemente mit dem Wert Null dominiert.

In [Mei07] und [Ant98] sind eine Vielzahl weiterer Verfahren zur Lösung von Gleichungssystemen (LU-Zerlegung, die QR-Zerlegung oder iterative Verfahren wie das Jacobi oder Gauss-Seidel-Verfahren) mit zum Teil geringeren Ausführungszeiten beschrieben. Jedoch kann bei den Ansätzen nicht sichergestellt werden, dass sie konvergieren bzw. für die Berechnung der T-Invarianten der in dieser Arbeit verwendeten Petri-Netze ganzzahlige positive Lösungen ergeben. Aus diesem Grund verwenden wir in dieser Arbeit für die Berechnung der T-Invarianten den G.-J.-Algorithmus.

5 Aktionslogik

Basierend auf der Arbeit von H. J. Genrich [Gen73] führt Fidelak [Fid93] eine zur Aussagenlogik duale Handlungslogik ein. Wie in den weiteren Arbeiten [Sim00], [LS00], [Lau96] dargestellt, erlaubt die Handlungslogik, die auch als *Logic of Action* (LA) bezeichnet wird, mit Hilfe von Prozessen kausale Beziehungen zwischen Aktionen zu definieren. Die Aussagenlogik erlaubt es, eine Situation (Aussage) zu beschreiben (eine Aussage kann wahr oder falsch sein) wohingegen mit Hilfe der Handlungslogik (Aktionslogik) Abläufe beschrieben werden. In dieser Arbeit wird die Handlungslogik bzw. Logic of Action als *Aktionslogik (AL)* bezeichnet.

In diesem Kapitel beschreiben wir im ersten Teil die Syntax und Semantik der Aktionslogik. Im zweiten Teil zeigen wir, wie mit Hilfe eines *direkten* und *indirekten Beweisverfahrens* auf Basis der Anzahl der *Prozesse* einer Formel der Aktionslogik entschieden werden kann, ob eine Formel eine andere Formel erfüllt. In dem darauf folgenden Abschnitt betrachten wir, wie die Elemente der Aktionslogik mit Hilfe von Petri-Netzen dargestellt werden können. Im letzten Teil dieses Kapitel übertragen wir das *direkte* und *indirekte Beweisverfahrens* der Aktionslogik auf die Petri-Netz-Darstellung und zeigen wie mit Hilfe von *T-Invarianten* die Anzahl der Prozesse einer Formel der Aktionslogik berechnet werden kann. Durch den Einsatz der Petri-Netze wird der Vergleich zweier Module der Aktionslogik veranschaulicht und vereinfacht.

5.1 Syntax und Semantik der Aktionslogik

Wir beginnen die Beschreibung der Aktionslogik mit der Definition der atomaren Elemente, den *Aktionen*. Im Anschluss definieren wir die Syntax und Semantik der aktionslogischen Formeln, die wir als *Module* bezeichnen. Das in der Aktionslogik vergleichbare Konzept zu den Modellen in der Aussagenlogik (vgl. Definition 4.3 auf Seite 49) bezeichnen wir als *Prozesse*. Prozesse erlauben den kausalen Zusammenhang der Aktionen zu definieren. Die Tabelle 5.1 stellt den Zusammenhang der Element der Aktionslogik und der Aussagenlogik dar.

Beginnen wir die Betrachtung der Aktionslogik mit den Aktionen. Bei der Definition der Aktionslogik gehen wir von einer endlichen nicht leeren Menge $\mathbb{A} = \{a, b, c, ...\}$ von Aktionen aus.

Aussagenlogik	Aktionslogik
atomare Formel	Aktion
Modell	Prozess
Formel	Modul

Tabelle 5.1: Vergleich Aussagenlogik und Aktionslogik

Eine Aktion ist die atomare Einheit der Aktionslogik und kann entweder ausgeführt oder nicht ausgeführt werden. Gemeinsam mit Operatoren, Konstanten und Klammern bilden die Aktionen das Alphabet der Aktionslogik.

Definition 5.1 (Alphabet der Aktionslogik)
Das Alphabet der Aktionslogik besteht aus:

- *einer endlichen Menge* $\mathbb{A} = \{a, b, c, ..\}$ *von Aktionen,*
- *einer Menge* $\mathbb{K} = \{\bot\}$ *mit der Konstanten* $\bot$*, welche für die Unausführbarkeit steht,*
- *den Operatoren* $\neg$, $\oslash$, $\oslash$, $\oslash$, $\oslash$, $\oplus$ *und* $\ominus$ *sowie*
- *den Klammern (,) zur Gruppierung der Aktionen und Operatoren.*

Für die Operatoren der Aktionslogik vereinbaren wir die folgenden Eigenschaften:
Die Operatoren $\neq$, $\oslash$, $\oslash$, sowie $\ominus$ bezeichnen wir als *elementare Operationen* und die Operatoren $\oslash$, $\oslash$ und $\oplus$ als *zusammengesetzte Operationen*. Hierbei legen wir $\oslash$, $\oslash$, $\ominus$ und $\oplus$ als kommutativ und assoziativ fest. $\oslash$, $\oslash$ sind nur assoziativ. Über $\oplus$ sind die Operationen $\oslash$,$\oslash$, $\ominus$, $\oslash$ und $\oslash$ distributiv.

Kann die Aktion $a \in \mathbb{A}$ ausgeführt werden, schreiben wir dafür kurz a. Kann die Aktion nicht ausgeführt werden, schreiben wir $\neg a$ oder synonym $\bar{a}$. Damit erhalten wir als Literale der Aktionslogik die Elemente der Literalmenge $\mathbb{L}$.

Definition 5.2 (Literalmenge)
Die Literalmenge $\mathbb{L} = \{a, \overline{a}, b, \overline{b}, c, \overline{c}..\}$ *zu der Aktionsmenge* $\mathbb{A}$ *enthält:*

- *die Menge* $\mathbb{A}$ *von Aktionen (Aktionen, die ausgeführt werden können) und*
- *die Menge von negierten Aktionen (Aktionen, die nicht ausgeführt werden können).*

Auf Basis der Grundelemente der Aktionslogik definieren wir nun die Formeln der AL und bezeichnen diese als Module.

Definition 5.3 (Module)
Für eine Aktionsmenge $\mathbb{A}$ *ist die Menge* $\mathbb{M}$ *der Module die kleinste Menge, für die gilt:*

- *Ist* $\bot \in \mathbb{K}$*, dann ist* $[\bot] \in \mathbb{M}$*. Für* $[\bot]$ *schreiben wir kurz* $\bot$*.*
- *Ist* $l \in \mathbb{L}$*, dann ist* $[l] \in \mathbb{M}$*. Für* $[l]$ *schreiben wir kurz* l*.*
- *Ist* $M \in \mathbb{M}$*, dann ist auch* $\overline{M} \in \mathbb{M}$ *mit* $\overline{\overline{M}} := M$
- *Ist* $M_1, M_2 \in \mathbb{M}$ *so auch* $M_1 \circ M_2 \in \mathbb{M}$ *für* $\circ \in \{\oslash, \oslash, \oslash, \oslash, \oplus, \ominus\}$.

Für die Definition der Negation eins Moduls M ($\overline{M}$) fehlen uns an dieser Stelle noch verschiedene Definitionen, die erst im Laufe des Kapitels eingeführt werden. Wir holen die Definition der Negation von Modulen daher am Ende dieses Kapitels in der Definition 5.23 nach.

Das Vorkommen der Konstanten, Aktionen und Literale in einem Modul wird durch die Definition 5.4 beschrieben.

Definition 5.4 (Konstanten, Aktionen und Literale in Modulen)
Seien $M, M_1, M_2 \in \mathbb{M}$ Module, $\perp \in \mathbb{K}$ ein Konstante, $a \in \mathbb{A}$ eine Aktion und $l \in \mathbb{L}$ ein Literal sowie $\circ$ ein elementarer oder zusammengesetzter Operator. Im Folgenden sei $\mathbb{K}[M]$ die Menge der Konstanten des Moduls M, $\mathbb{A}[M]$ die Aktionsmenge des Moduls M und $\mathbb{L}[M]$ die Literalmenge des Moduls M.

- *Für $[\perp] \in \mathbb{M}$ ist $\perp \in \mathbb{K}[\perp]$ und $\perp \in \mathbb{K}[M]$ mit $M = [M_1 \circ M_2]$, wenn $\perp \in \mathbb{K}[M_1] \cup \mathbb{K}[M_2]$.*
- *Für $[a], [\overline{a}] \in \mathbb{M}$ ist $a \in \mathbb{A}[a]$, $a \in \mathbb{A}[\overline{a}]$ und $a \in \mathbb{A}[M]$ mit $M = [M_1 \circ M_2]$, wenn $a \in \mathbb{A}[M_1] \cup \mathbb{A}[M_2]$. Weiterhin gilt, ist $a \in \mathbb{A}[M]$ so ist auch $a \in \mathbb{A}[\overline{M}]$.*
- *Für $[l] \in \mathbb{M}$ ist $l \in \mathbb{L}[l]$ und $l \in \mathbb{L}[M]$ mit $M = [M_1 \circ M_2]$, wenn $l \in \mathbb{L}[M_1] \cup \mathbb{L}[M_2]$.*

Nach der syntaktischen Definition der Formeln der *AL* definieren wir mit Hilfe von Prozessen die Semantik der Module. Die Menge der Prozesse, die ein Modul erfüllt, ist wie bereits erwähnt vergleichbar mit der Menge von Modellen einer aussagenlogischen Formel. Prozesse erlauben die kausale Ordnung von Aktionen zu beschreiben, d.h. in welcher Reihenfolge die Aktionen ausgeführt oder nicht ausgeführt werden. Im Vergleich zu den Modulen werden die Literale in den Prozessen nur mit Hilfe der elementaren Operatoren $\oslash$, $\oplus$ und $\ominus$ verknüpft, welche die kausale Abhängigkeit der Aktionen zueinander ausdrücken.

Wir beginnen die syntaktische Definition von Prozessen mit Hilfe von Vor-Prozessen (in [Fid93] auch als Quasi-Prozesse bezeichnet).

Definition 5.5 (Vor-Prozesse)
Die Menge $\mathbb{V}$ der Vor-Prozesse (Quasi-Prozesse) ist die kleinste Menge, für die gilt:

- *Wenn $l \in \mathbb{L}$, so ist $(l) \in \mathbb{V}$. Hierbei schreiben wir auch kurz l für (l).*
- *Sind $v_1, v_2 \in \mathbb{V}$, so sind auch $(v_1 \oslash v_2)$, $(v_1 \oplus v_2)$ und $(v_1 \ominus v_2) \in \mathbb{V}$.*

Das Auftreten der Aktionen und Literale in den Vor-Prozessen ist wie folgt festgelegt:

Definition 5.6 (Aktionen und Literale in den Vor-Prozessen)
Seien $v_1, v_2, v_3 \in \mathbb{V}$ drei Vor-Prozesse. Weiterhin ist $a \in \mathbb{A}$, $l \in \mathbb{L}$ und $\circ$ eine elementare Operation.

- *Für $(a), (\overline{a}) \in \mathbb{V}$ ist $a \in \mathbb{A}(a)$, $a \in \mathbb{A}(\overline{a})$ und $a \in \mathbb{A}(v)$ mit $v_1 = (v_2 \circ v_3)$, falls $a \in \mathbb{A}(v_2) \cup \mathbb{A}(v_3)$.*
- *Für $(l) \in \mathbb{V}$ ist $l \in \mathbb{L}(l)$ und $l \in \mathbb{L}(v_1)$ mit $v_1 = (v_2 \circ v_3)$, falls $l \in \mathbb{L}(v_2) \cup \mathbb{L}(v_3)$.*

Basierend auf der Definition der Vor-Prozesse definieren wir nun die Prozesse. Dazu schränken wir im Folgenden, wie in [Fid93] beschrieben, die Vor-Prozesse weiter ein.

Definition 5.7 (Prozesse)
Die Menge der Prozesse $\mathbb{P}$ ist die kleinste Menge, für die gilt:

- *Ist das Literal $l \in \mathbb{L}$ und $(l) \in \mathbb{V}$, so ist $(l) \in \mathbb{P}$*
- *Sind die Vor-Prozesse $p_1, p_2 \in \mathbb{V}$ und ist $\mathbb{A}(p_1) \cap \mathbb{A}(p_2) = \emptyset$, so sind auch $(p_1 \oslash p_2)$ und $(p_1 \ominus p_2) \in \mathbb{P}$.*
- *Sind die Vor-Prozesse $p_1, p_2 \in \mathbb{V}$ und es existiert kein Literal $l \in \mathbb{L}$ mit $l \in \mathbb{L}(p_1)$ und $\overline{l} \in \mathbb{L}(p_2)$, dann ist $(p_1 \oplus p_2) \in \mathbb{P}$.*

Prozesse geben somit an, ob eine Aktion *vor*, *nach* oder *gleichzeitig* mit einer anderen Aktion ausgeführt wird. In einem Prozess kann die gleiche Aktion nicht gleichzeitig ausgeführt und nicht ausgeführt (negiert) enthalten sein. Ebenso kann eine Aktion nicht vor oder nach sich selbst ausgeführt werden. Beispiele für gültige Prozesse sind (a), $(a \ominus b)$ oder $(a \oplus b)$. Hingegen sind $(a \oslash \overline{a})$, $(b \oslash \overline{b})$ und $(c \oplus \overline{c})$ keine gültigen Prozesse.

Zur Beschreibung der kausalen Abhängigkeit (*VOR*, *NACH* und *GLEICHZEITIG*) zweier Aktionen verwendet Fidelak [Fid93] Kausalstrukturen. Im Folgenden werden die Abhängigkeiten wie in [LS00] mit Hilfe einer Abhängigkeitsfunktion beschrieben.

Definition 5.8 (Abhängigkeitsfunktion)
Die Abhängigkeitsfunktion $A : \mathbb{L} \times \mathbb{L} \rightarrow \{VOR, NACH, GLEICHZEITIG\}$ beschreibt die kausale Abhängigkeit der Literale l_1 und l_2 der Literalmenge $\mathbb{L}$ wie folgt:

- $A(l_1, l_2)$:= *VOR, wenn* l_1 vor l_2 *ausgeführt oder nicht ausgeführt (verboten) ist,*
- $A(l_1, l_2)$:= *NACH, wenn* l_1 nach l_2 *ausgeführt oder nicht ausgeführt (verboten) ist,*
- $A(l_1, l_2)$:= *GLEICHZEITIG, wenn* l_1 gemeinsam *mit* l_2 *ausgeführt oder nicht ausgeführt (verboten) ist,*

Wird die Beziehung zweier Aktionen, dargestellt durch die Literale (l_1, l_2) der Literalmenge $\mathbb{L}$, durch die Funktion $A(l_1, l_2)$ als *VOR* oder *NACH* ausgewertet, so ist die Ausführung der Aktionen zeitlich voneinander getrennt. Die zweite Aktion kann erst ausgeführt werden wenn die erste Aktion abgeschlossen ist. Wird die Funktion zu *GLEICHZEITIG* ausgewertet, kann keine zeitliche Unterscheidung zwischen der Ausführung beider Aktionen getroffen werden, da beide stark kausal miteinander verknüpft sind und somit die Ausführung zeitlich nicht zu unterscheiden ist.

Ist die Abhängigkeitsfunktion in Bezug auf *GLEICHZEITIG* symmetrisch und in Bezug auf *VOR* und *NACH* transitiv, so bezeichnen wir sie als *konsistent*. Wir beschreiben diese Aussage mit Hilfe der Definition 5.9 formal.

Definition 5.9 (Konsistente Abhängigkeitsfunktion)
Gegeben sei eine Menge $\mathbb{L}$ von Literalen der Aktionslogik.

Die Abhängigkeitsfunktion

$$A : \mathbb{L} \times \mathbb{L} \rightarrow \{VOR, NACH, GLEICHZEITG\}$$

wird als konsistent *bezeichnet, wenn folgende Eigenschaften gelten:*

- *symmetrisch*
  1. $\forall l \in \mathbb{L}$ *gilt:*
     - $A(l, l) = GLEICHZEITIG$
  2. $\forall l_1, l_2 \in \mathbb{L}$ *gilt:*
     - $A(l_1, l_2) = VOR \Rightarrow A(l_2, l_1) = NACH$
     - $A(l_1, l_2) = NACH \Rightarrow A(l_2, l_1) = VOR$
     - $A(l_1, l_2) = GLEICHZEITIG \Rightarrow A(l_2, l_1) = GLEICHZEITIG$
- *transitiv*
  1. $\forall l_1, l_2, l_3 \in \mathbb{L}$ *gilt:*
     - $A(l_1, l_2) = VOR \wedge A(l_2, l_3) = VOR \Rightarrow A(l_1, l_3) = VOR$
     - $A(l_1, l_2) = NACH \wedge A(l_2, l_3) = NACH \Rightarrow A(l_1, l_3) = NACH$
     - $A(l_1, l_2) = GLEICHZEITIG \wedge A(l_2, l_3) = GLEICHZEITIG$
       $\Rightarrow A(l_1, l_3) = GLEICHZEITIG$
  2. $\forall l_1, l_{2,l_3} \in \mathbb{L}$ *gilt:*
     - $A(l_1, l_2) = VOR \wedge A(l_2, l_3) = GLEICHZEITIG \Rightarrow A(l_1, l_3) = VOR$
     - $A(l_1, l_2) = GLEICHZEITIG \wedge A(l_2, l_3) = VOR \Rightarrow A(l_1, l_3) = VOR$
     - $A(l_1, l_2) = NACH \wedge A(l_2, l_3) = GLEICHZEITIG \Rightarrow A(l_1, l_3) = NACH$
     - $A(l_1, l_2) = GLEICHZEITIG \wedge A(l_2, l_3) = NACH \Rightarrow A(l_1, l_3) = NACH$

Für den Beweis der Aussage, dass die Abhängigkeitsfunktion A_p des Prozesses p konsistent ist, wird auf [Sim00] verwiesen.

Mit Hilfe der vorgestellten Definition können wir die Prozesse auch mit Hilfe einer Abhängigkeitsfunktion darstellen. Dies beschreiben wir in der nächsten Definition.

Definition 5.10 (Darstellung von Prozessen als Abhängigkeitsfunktion)
Ist $p \in \mathbb{P}$ ein Prozess und ist a eine Aktion der Aktionsmenge $\mathbb{A}(p)$ des Prozesses.
Dann heißt die Funktion

$$A_p : \mathbb{L}(p) \times \mathbb{L}(p) \rightarrow \{VOR, NACH, GLEICHZEITIG\}$$

die Abhängigkeitsfunktion des Prozesses p, wenn gilt:

- *Ist $p = (\perp)$, dann ist A_p leer.*
- *Ist $p = (a)$, dann ist $A_p(a, a) = GLEICHZEITG.$*
- *Ist $p = (\overline{a})$, dann ist $A_p(\overline{a}, \overline{a}) = GLEICHZEITG.$*
- *Ist $p = (p_1 \ominus p_2)$ und gilt $\forall l_1, l_2 \in \mathbb{L}(p)$, dann ist $A_p(l_1, l_2) = GLEICHZEITIG.$*
- *Ist $p = (p_1 \oslash p_2)$, dann ist* $A_p(p_1, p_2 = \begin{cases} A_{p_1}(l_1, l_2), & wenn\ \forall l_1, l_2 \in \mathbb{L}(p_1) \\ A_{p_2}(l_1, l_2), & wenn\ \forall l_1, l_2 \in \mathbb{L}(p_2) \\ VOR, & wenn\ \forall l_1 \in \mathbb{L}(p_1), l_2 \in \mathbb{L}(p_2) \\ NACH, & wenn\ \forall l_1 \in \mathbb{L}(p_2), l_2 \in \mathbb{L}(p_1) \end{cases}$
- *Ist $p = (p_1 \oslash p_2)$, dann ist* $A_p(p_1, p_2 = \begin{cases} A_{p_1}(l_1, l_2), & wenn\ \forall l_1, l_2 \in \mathbb{L}(p_1) \\ A_{p_2}(l_1, l_2), & wenn\ \forall l_1, l_2 \in \mathbb{L}(p_2) \\ NACH, & wenn\ \forall l_1 \in \mathbb{L}(p_1), l_2 \in \mathbb{L}(p_2) \\ VOR, & wenn\ \forall l_1 \in \mathbb{L}(p_2), l_2 \in \mathbb{L}(p_1) \end{cases}$

Die Darstellung der Prozesse mit Hilfe von Abhängigkeitsfunktionen ermöglicht es, die Prozesse zu vergleichen.

Definition 5.11 (Vergleich von Prozessen)
Seien $p_1, p_2 \in \mathbb{P}$ zwei Prozesse mit $\mathbb{L}(p_1) = \mathbb{L}(p_2)$, dann gilt:

Die beiden Prozesse p_1, p_2 sind genau dann gleich ($p_1 = p_2$), wenn $\forall l_1, l_2 \in \mathbb{L}(p_1)$ gilt, dass $A_{p_1}(l_1, l_2) = A_{p_2}(l_1, l_2)$ ist.

Wie bereits beschrieben, sind die Prozesse eines AL-Moduls vergleichbar mit Modellen einer aussagenlogischen Formel. Wie in der Aussagenlogik die Frage von Interesse ist, welche Modelle (Belegungen) eine Formel erfüllen, sind wir in der Aktionslogik daran interessiert, zu bestimmen, welche Prozesse ein Modul realisieren.

Alle Prozesse, welche eine AL-Modul M realisieren werden zu der Prozessmenge $\mathbb{P}[M]$ zusammengefasst. Wir definieren im Folgenden die Prozessmenge $\mathbb{P}[M]$ in zwei Schritten. Im ersten Schritt betrachten wir nur Module, deren Literale ausschließlich durch elementare Operatoren verknüpft sind. Im zweiten Schritt erweitern wir die Definition auf Module mit elementaren und zusammengesetzten Operatoren.

Definition 5.12 (Prozessmenge von Modulen mit elementaren Operatoren)
Seien $M_1, M_2 \in \mathbb{M}$ zwei Module und $a \in \mathbb{A}$ eine Aktion, dann ergibt sich die Prozessmenge $\mathbb{P}[M]$ für Module ohne Operatoren wie folgt:

- $\mathbb{P}[\bot] := \emptyset$
- $\mathbb{P}[a] := \{(a)\}$
- $\mathbb{P}[\overline{a}] := \{(\overline{a})\}$

Die Prozessmenge für Module mit elementaren Operatoren ergibt sich wie folgt:

- $\mathbb{P}[M_1 \circledcirc M_2] := \{p = (p_1 \circledcirc p_2)\ wobei\ p_1 \in \mathbb{P}[M_1], p_2 \in \mathbb{P}[M_2]\}$
- $\mathbb{P}[M_1 \circledgtr M_2] := \{p = (p_1 \circledgtr p_2)\ wobei\ p_1 \in \mathbb{P}[M_1], p_2 \in \mathbb{P}[M_2]\}$
- $\mathbb{P}[M_1 \circledminus M_2] := \{p = (p_1 \circledminus p_2)\ wobei\ p_1 \in \mathbb{P}[M_1], p_2 \in \mathbb{P}[M_2]\}$

Die Prozessmenge $\mathbb{P}[M_1 \circ M_2]$ mit $\circ \in \{\circledcirc, \circledgtr, \circledminus\}$ ist leer, wenn die Prozessmenge von $\mathbb{P}[M_1]$ oder $\mathbb{P}[M_2]$ leer ist.

Auffällig bei den bisher betrachteten Prozessmengen ist, dass die Prozessmenge jeweils nur einen einzigen oder keinen Prozess enthält. Dies ist mit dem Aufbau der Module, deren Literale nur durch elementare Operatoren verknüpft sind, zu begründen.

Zur Verdeutlichung betrachten wir die Prozessmengen der Module $M_1 = [a]$, $M_2 = [a \circledcirc b]$ und $M_3 = [a \circledgtr (b \circledcirc c)]$.

- $\mathbb{P}[M_1] = \mathbb{P}[a] = \{(a)\}$
- $\mathbb{P}[M_2] = \mathbb{P}[a \circledcirc b] = \{(a \circledcirc b)\}$
- $\mathbb{P}[M_3] = \mathbb{P}[a \circledgtr (b \circledcirc c] = \{(b \circledcirc c \circledcirc a)\}$

Im Gegensatz zu Modulen, welche nur elementare Operatoren enthalten, können die Prozessmengen von Modulen mit zusammengesetzten Operatoren ($\owedge, \owedge, \oplus$) aus mehreren Prozessen bestehen.

Definition 5.13 (Prozessmenge von Modulen mit zusammengesetzten Operatoren)
Seien $M_1, M_2 \in \mathbb{M}$ zwei Module.

Die Prozessmengen der Module ergeben sich wie folgt:

- $\mathbb{P}(M_1 \owedge M_2) := \mathbb{P}[M_1 \circledcirc M_2] \cup \mathbb{P}[M_1 \circledgtr M_2] \cup \mathbb{P}[M_1 \circledminus M_2]$
- $\mathbb{P}(M_1 \owedge M_2) := \mathbb{P}[M_1] \cup \mathbb{P}[M_2] \cup \mathbb{P}[M_1 \owedge \mathbb{P}[M_2]$
- $\mathbb{P}(M_1 \oplus M_2) := \mathbb{P}[(M_1] \uplus \mathbb{P}[M_2]\ also\ (\mathbb{P}[M_1] \cup \mathbb{P}[M_2]) - (\mathbb{P}[M_1] \cap \mathbb{P}[M_2])$

Betrachten wir im Folgenden Beispiele für die Prozessmenge von Modulen mit zusammengesetzten Operationen. Die Prozessmengen der Module $M_4 = [a \oplus b], M_5 = [(a \oslash b) \oslash c]$ und $M_6 = [a \otimes b]$ sollen die vorherige Definition exemplarisch verdeutlichen:

- $\mathbb{P}[M_4] = \mathbb{P}[a \oplus b] = \{(a), (b)\}$
- $\mathbb{P}[M_5] = \mathbb{P}[(a \oslash b) \oslash c] = \{((a \oslash b) \oslash c), ((a \oslash b) \ominus c), ((a \oslash b) \oplus c)\}$
- $\mathbb{P}[M_6] = \mathbb{P}[a \otimes b] = \{(a), (b), (a \oslash b), (a \ominus b), (a \oplus b)\}$

Für die Prozessmengen von zwei Modulen $M_1, M_2 \in \mathbb{M}$ gelten die folgenden Eigenschaften:

- $\mathbb{P}[M_1 \oslash M_2] = \mathbb{P}[M_2 \ominus M_1]$
- $\mathbb{P}[M_1 \oplus M_2] = \mathbb{P}[M_2 \oplus M_1]$
- $\mathbb{P}[M_1 \oslash M_2] = \mathbb{P}[M_2 \oslash M_1]$
- $\mathbb{P}[M_1 \otimes M_2] = \mathbb{P}[M_2 \otimes M_1]$
- $\mathbb{P}[M_1 \oplus M_2] = \mathbb{P}[M_2 \oplus M_1]$

Wir erweitern nun die Definition der AL um Schleifen. Diese ermöglicht die Beschreibung von wiederkehrenden Aktionen in den Prozessen. Damit die Aktionen in den Prozessen jedoch weiterhin eindeutig identifiziert werden können, wird jede Aktion in einer Schleife mit einem Index versehen. Die nachfolgende Definition beschreibt die drei Schleifentypen:

1. a^n für eine n-malige Wiederholung der Aktion a mit $n \in \mathbb{N}_0$
2. a^+ für eine 1 bis n-malige Wiederholung der Aktion a
3. a^* für eine 0 bis n-malige Wiederholung der Aktion a

Definition 5.14 (Schleifen in der Aktionslogik)
Sei $p \in \mathbb{P}_{\mathbb{A}}$ einen Prozess über der Aktionsmenge $\mathbb{A} = \{a, b, c, \ldots\}$.
Die Aktionsmenge $\mathbb{A}_n = \{a_1, \ldots, a_n, b_1, \ldots, b_n, c_1, \ldots, c_n, \ldots\}$ mit $x_i \in \mathbb{A}_n, 1 \leq i \leq n$ für $n \in \mathbb{N}$ enthält eine Kopie für jede Aktion x der Aktionsmenge $\mathbb{A}$ ($x \in \mathbb{A}$).

$S(p, n)$ ist ein Prozess über der Aktionsmenge $\mathbb{A}_n$ wobei jedes Vorkommen des Literals l in p durch das Literal l_n mit entsprechenden Index n ersetzt wird.

Ausgehend von dieser Annahme definieren wir für die Prozessmengen für ein Modul $M \in \mathbb{M}_{\mathbb{A}}$:

- $$\mathbb{P}[M^n] := \begin{cases} \emptyset & \text{wenn } n = 0 \\ \{p_1 | p \in \mathbb{P}[M] \text{ und } p_1 = S(p, 1)\} & \text{wenn } n = 1 \\ \{p_{\leq n} | p \in \mathbb{P}[M], p_{\leq n-1} \in \mathbb{P}[M^{n-1}] \text{ und } p_n = p_{\leq n-1} \oslash S(p, n)\} & \text{wenn } n > 1 \end{cases}$$
- $$\mathbb{P}[M^+] := \bigcup_{i=1}^{\infty} \mathbb{P}[M^i]$$
- $$\mathbb{P}[M^*] := \bigcup_{i=0}^{\infty} \mathbb{P}[M^i]$$

Zur Verdeutlichung der Definition 5.14 betrachten wir zwei Beispiele:

$$\begin{aligned}\mathbb{P}[a^4] &= \{(p \oslash a_4) | p \in \mathbb{P}[a^3]\} \\ &= \{(p \oslash a_3 \oslash a_4) | p \in P[a^2]\} \\ &= \{(p \oslash a_2 \oslash a_3 \oslash a_4) | p \in P[a^1]\} \\ &= \{(a_1 \oslash a_2 \oslash a_3 \oslash a_4)\}\end{aligned}$$

$$\mathbb{P}[[a \oslash b]^3] = \{((a_1 \oslash b_1) \oslash (a_2 \oslash b_2) \oslash (a_3 \oslash b_3))\}$$

In der Aussagenlogik wird eine Formel als Tautologie bezeichnet, wenn jede Belegung der Formel ein Modell der Formel ist (vergleiche Definition 4.3 auf Seite 49). In der Aktionslogik definieren wir die Tautologie als Gegensatz zur Kontradiktion ($\perp$), die für die Unausführbarkeit von Aktionen steht.

Die Prozessmenge der Tautologie enthält in jedem Prozess alle Aktionen der Aktionsmenge. In [Fid93] und [LS00] wird das Modul $M_1 = [a \oplus \overline{a}]$ als elementare Tautologie bezeichnet.

Die Tautologie über einer Aktionsmenge $\mathbb{A}$ besteht aus der Konjunktion von elementaren Alternativen ($\oplus$) aller Aktionen der Aktionsmenge. Daraus ergibt sich folgende Definition für die Tautologie.

Definition 5.15 (Tautologie)
Sei $\mathbb{A} = \{a_1, \ldots, a_n\}$ eine Aktionsmenge. Das Modul

$$[\top_{\mathbb{A}}] := [[a_1 \oplus \overline{a_1}] \oslash \ldots \oslash [a_n \oplus \overline{a_n}]]$$

wird als Tautologie über der Aktionsmenge $\mathbb{A}$ bezeichnet.

Somit ist das Modul $[[a \oplus \overline{a}] \oslash [b \oplus \overline{b}]]$ über der Aktionsmenge $\mathbb{A} = \{a, b\}$ eine Tautologie.

5.2 Vergleich von zwei Aktionslogik-Modulen

Im Abschnitt 5.1 haben wir die Aktionslogik eingeführt und dargestellt, wie mit Hilfe von Prozessen eine kausale Ordnung zwischen Aktionen definiert werden kann. Die Formeln der AL werden dabei als Module bezeichnet. In diesem Kapitel beschäftigen wir uns mit dem Vergleich zweier AL-Module und definieren basierend auf [LS00], [Fid93] bzw. [Sim00] wann ein Modul ein weiteres Modul *erfüllt*, also wann in beiden Modulen die Aktionen in der gleichen kausalen Ordnung zueinander stehen.

Das entsprechende Konstrukt in der Aktionslogik zu einem Modell einer aussagenlogischen Formel ist der Prozess. In der Aussagenlogik stellt sich die Frage, ob ein Modell einer aussagenlogischen Formel auch ein Modell einer zweiten aussagenlogischen Formel ist. Durch die Möglichkeit Abläufe (Prozesse) mit Hilfe der Aktionslogik zu definieren stellt sich nun die Frage, wie

überprüft werden kann, ob die definierten Abläufe zweier Module einander entsprechen. Die Möglichkeit Module zu vergleichen, erlaubt Abläufe mit Hilfe eines Moduls der Aktionslogik zu spezifizieren und zu entscheiden, ob eine Realisierung (ein AL-Modul) das festgelegte Verhalten einer Spezifikation (eines zweiten AL-Moduls) erfüllt.

Für die Überprüfung, ob die Realisierung die Spezifikation erfüllt, müssen beide Module analog zur wechselseitigen Vervollständigung von Formeln der Aussagenlogik (siehe Definition 4.4 auf Seite 49) vergleichbar gemacht werden. Zum einen ist es erforderlich, dass beide Module die gleiche Aktionsmenge besitzen und zum anderen muss die kausale Ordnung der Aktionen betrachtet werden. Für den Vergleich der kausalen Ordnung müssen die Abhängigkeitsfunktionen für gleiche Aktionen die gleichen Ergebnisse liefern.

Bevor wir diese Kriterien formal definieren betrachten wir die Vervollständigung von Modulen. Diese erlaubt es, die Prozesse untereinander vergleichbar zu machen.

Betrachten wir dazu im Vorfeld zwei Beispiele von Prozessen und den zugehörigen Prozessmengen. Die Prozessmenge von $\mathbb{P}[a \oplus b] = \{(a), (b)\}$ bzw. $\mathbb{P}[a \veebar b] = \{(a), (b), (a \circledcirc b), (a \circledgtr b), (a = b)\}$ verdeutlichen, dass nicht in jedem Prozess eines Moduls alle Aktionen der Aktionsmenge vorkommen. Bei $\mathbb{P}[a \oplus b] = \{(a), (b)\}$ ist in einem Prozess der Prozessmenge nur die Aktion a und im anderen Prozess nur die Aktion b enthalten, wobei die Aktionsmenge von $\mathbb{A}_{a \oplus b} = \{a, b\}$ ist. Bei der Aktionsmenge von $\mathbb{P}[a \veebar b]$ enthalten die beiden Prozesse a und b jeweils nur eine Aktion der Aktionsmenge $\mathbb{A}_{a \veebar b} = \{a, b\}$, wohingegen die restlichen drei Prozesse $(a \circledcirc b), (a \circledgtr b), (a \circledS b)$ beide Aktionen der Aktionsmenge enthalten. Die Prozesse, die nicht alle Aktionen der Aktionsmenge des Moduls enthalten, werden als *unvollständige Prozesse* bezeichnet (vergleiche [LS00], [Fid93]) und sind daher gegenseitig schwer zu vergleichen. Module, deren Prozessmenge nur vollständige Prozesse enthalten, werden als *vollständig* bezeichnet.

Generell gilt, dass bei Modulen, die eine Alternative enthalten (Modul mit mindestens einem exklusivem oder ($\oplus$) oder einen einschließenden oder ($\veebar$)) nicht alle Prozesse im Bezug auf die Aktionsmenge des Moduls vollständig sind. Wir geben in der Definition 5.16 an, wann ein Modul vollständig ist.

Definition 5.16 (Vollständige Module)
Ist $M \in \mathbb{M}$ ein Modul mit der Literalmenge $\mathbb{L}[\mathbb{M}]$. M heißt genau dann vollständig, wenn für alle Alternativen Alt (alle Verbindungen zweier Aktionen mit den Operatoren $\oplus$ und $\veebar$) in dem Modul M gilt: $\mathbb{L}[Alt] = \mathbb{L}[M]$.

Für den Vergleich von Prozessen untereinander ist es notwendig, dass die Prozesse vollständig sind, also jeder Prozess alle Aktionen der Aktionsmenge enthält. Zur Vervollständigung der Module wird jede Alternative des Moduls um die fehlenden Literale in Form von einer elementaren Alternative ergänzt.

Definition 5.17 (Vervollständigung von Modulen)
Ein Modul $M \in \mathbb{M}$ mit der Literalmenge $\mathbb{L}$ kann zu einem vollständigen Modul $V[M]$ ergänzt werden, indem jede Alternative Alt von M wie folgt ergänzt wird:

$$[Alt \wedge [a_1 \circledS \overline{a_1}] \barwedge [a_2 \circledS \overline{a_2}] \barwedge \ldots \barwedge [a_n \circledS \overline{a_n}]] \text{ mit } \{a_1 \ldots a_n\} = \mathbb{L}[\mathbb{M}] \backslash \mathbb{L}[\mathbb{A}].$$

In [LS00] ist ein Verfahren zur Vervollständigung der Prozesse und der Prozessmenge eines Moduls auf Basis der Abhängigkeitsfunktionen beschrieben. Die Definitionen 5.18 und 5.19 beschreiben die einzelnen Schritte des Ansatzes.

In der Definition 5.18 ist die Vervollständigung der Prozesse eines AL-Moduls beschrieben.

Definition 5.18 (Vervollständigung der Prozesse eines Moduls)
Sei $M \in \mathbb{M}$ ein Modul über der Aktionsmenge $\mathbb{A}$ und $p \in P[M]$ ein Prozess der Prozessmenge von M. Weiterhin sei die Literalmenge $\mathbb{L} \subseteq \mathbb{A} \cup \{\overline{a} \mid a \in \mathbb{A}\}$ mit $\mathbb{L}(p) \cap \mathbb{L} = \emptyset$.

Die Funktion $\mathrm{g}(p, M)$ *bestimmt die Literale des Moduls M, welche in dem Prozess p nicht enthalten sind wie folgt:*

$$\mathrm{g}(p, M) := \{\overline{a} \mid a \in \mathbb{L}(M), a \notin \mathbb{A}(p)\} \cup \{a \mid a \in \mathbb{L}(M), \overline{a} \notin \mathbb{A}(p)\}$$

Durch die Funktion $\mathrm{e}(p, \mathbb{L})$ *ergibt sich ein neuer Prozess p' durch Ergänzung mit den Literalen der Literalmenge $\mathbb{L}$.*

$$\mathrm{e}(p, \mathbb{L}) := \{p' | p' \textit{ ist ein Prozess mit konsistenter Abhängigkeitsfunktion } A_{p'}\}.$$

Weiterhin gilt $\forall l_1, l_2 \in \mathbb{L}(p')$ *ist*

$$\mathbb{L}(p) \cup \mathbb{L} : A_{p'}(l_1, l_2) \begin{cases} = A_p(l_1, l_2) & \textit{wenn } l_1, l_2 \in \mathbb{L}(p) \\ \{VOR, NACH, GLEICHZEITIG\} & \textit{sonst} \end{cases}$$

In der Definition 5.19 ist die Vervollständigung der Prozessmenge eines AL-Moduls beschrieben.

Definition 5.19 (Vervollständigung der Prozessmenge eines Moduls)
Die Vervollständigung der Prozessmenge $V[M]$ eines Moduls M ist wie folgt definiert:

$$P[V[M]] := \bigcup_{p \in P[M]} \mathrm{e}(p, \mathrm{g}(p, M))$$

Auf Basis der eingeführten Definitionen zur Vervollständigung der Prozessmengen ergibt sich für das Modul $[a \oplus b]$ die Prozessmenge

$$\mathbb{P}[V[a \oplus b]] = \{(a \circledcirc \overline{b}), (a \circledcirc \overline{b}), (a \circledcirc \overline{b}), (b \circledcirc \overline{a}), (b \circledcirc \overline{a}), (b \circledcirc \overline{a})\}.$$

Somit enthält jeder Prozess der Prozessmenge alle Aktionen der Aktionsmenge, was den Vergleich der Prozesse erleichtert bzw. erst ermöglicht.

Neben der Vergleichbarkeit der Prozesse eines Moduls ist es für den Vergleich zweier Module entscheidend, dass beide Module auf der gleichen Aktionsmenge basieren. Dazu übertragen wir den Ansatz zur *wechselseitigen Vervollständigung* von aussagenlogischer Formeln auf die Prozesse und Module der Aktionslogik.

Definition 5.20 (Wechselseitige Vervollständigung von Modulen)
Seien $M_1 \in \mathbb{M}$ und $M_2 \in \mathbb{M}$ zwei Module über der Aktionsmenge $\mathbb{A}[M_1]$ bzw. $\mathbb{A}[M_2]$. Weiterhin sei $\{a_1, \ldots, a_n\} := \mathbb{A}_{M_2} - \mathbb{A}_{M_1}$.

Die wechselseitige Vervollständigung $V_{M_2}[M_1]$ von M_1 im Bezug auf M_2 ist:

$$V_{M_2}[M_1] := \begin{cases} [M_1 \between [[a_1 \oplus \overline{a_1}] \between \ldots \between [a_n \oplus \overline{a_n}]]] & \text{wenn } \mathbb{A}_{M_2} - \mathbb{A}_{M_1} \neq \emptyset \\ V[M_1] & \text{sonst} \end{cases}$$

Basierend auf der Definition 5.20 zur *wechselseitigen Vervollständigung* der Module können wir nun angeben, wann ein Modul (Realisierung) ein zweites Modul (Spezifikation) *erfüllt*. Eine Realisierung (AL-Modul) *erfüllt* eine Spezifikation (AL-Modul), wenn die Prozessmenge der Realisierung eine Teilmenge der Prozessmenge der Spezifikation ist. Dies bedeutet, dass die Aktionen der Prozesse in der Realisierung die gleiche kausale Ordnung haben wie die Aktionen in den Prozessen in der Spezifikationen. Die kausalen Ordnungen dürfen sich nicht widersprechen. Ist der Durchschnitt der beiden Prozessmengen leer, so *widerspricht* die Realisierung der Spezifikation.

Definition 5.21 (Realisierung erfüllt bzw. widerspricht eine(r) Spezifikation)
Seien R und S zwei Module über der Aktionsmenge $\mathbb{A}$, dann gilt:

*Das Modul R **erfüllt** das Modul S ($R \twoheadrightarrow S$), genau dann wenn:*

$$\mathbb{P}[V_S[R]] \subseteq \mathbb{P}[V_R[S]] \text{ bzw.}$$

$$\mathbb{P}[V_S[R]] \cap \mathbb{P}[V_R[S]] = \mathbb{P}[V_S[R]].$$

*Das Modul R **widerspricht** dem Modul S ($R \twoheadrightarrow\!\!\!\!\!\!\cdot\;\; S$) genau dann wenn:*

$$\mathbb{P}[V_S[R]] \cap \mathbb{P}[V_R[S]] = \emptyset.$$

In [Sim00] werden in diesem Zusammenhang die Begriffe *sound* (fehlerfrei) und *complete* (vollständig) eingeführt. Nach der Definition von Simon ([Sim00]) ist eine Realisierung *fehlerfrei* in Bezug auf die Spezifikation, wenn die Realisierung die Spezifikation *erfüllt*. *Vollständig* ist die Realisierung in Bezug auf die Spezifikation, wenn alle definierten Prozesse der Spezifikation in der Realisierung enthalten sind.

Definition 5.22 (Fehlerfrei, Vollständig)
Seien R (Realisierung) und S (Spezifikation) zwei Module über der Aktionsmenge $\mathbb{A}$.

- *R ist **fehlerfrei** (erfüllt) im Bezug auf S wenn $R \twoheadrightarrow S$*
- *R ist **vollständig** im Bezug auf S wenn $S \twoheadrightarrow R$*

Nach der Definition 5.21 ist es für die Entscheidung, ob ein Modul R ein Modul S *erfüllt*, notwendig, alle Prozesse der beiden Module zu bestimmen und zu vergleichen. Dieses Vorgehen kann bei großen Modulen sehr aufwendig werden, zumal durch die *Vervollständigung* und die *wechselseitige Vervollständigung* die Größe der Module noch einmal deutlich erhöht wird. Aufgrund dieser Problematik ist in [LS00] und [Fid93] ein Ansatz beschrieben, welcher es ermöglicht, auf Basis der Anzahl der Prozesse zu entscheiden, ob die Realisierung die Spezifikation erfüllt.

Im Folgenden skizzieren wir diesen Ansatz und beschreiben in der Definition 5.24 das von [LS00], [Sim00] und [Fid93] beschriebene *direkte* und *indirekte Beweisverfahren*.

Ausgangspunkt der Überlegung ist im ersten Schritt, die Anzahl der Prozesse der Realisierung zu berechnen und im zweiten Schritt ein neues Modul, bestehend aus der Konjunktion der Module der Realisierung und der Spezifikation ($R \owedge S$) zu bilden und die Anzahl der Prozesse des neuen Moduls zu bestimmen. Stimmt die Anzahl der Prozesse beider Module (R und $R \owedge S$) überein, so erfüllt die Realisierung die Spezifikation.

Der Grund für die Vergleichbarkeit der Module auf Basis der Anzahl der Prozesse ist, dass wenn gleiche Aktionen in der Realisierung und in der Spezifikation vorkommen, die Ergebnisse der Abhängigkeitsfunktionen, also die kausale Ordnung der Aktionen, gleich sein müssen. Ist die kausale Ordnung der Aktionen in beiden Modulen verschieden, so kann die Prozessmenge der Konjunktion der beiden Module ($(R \owedge S)$), die auf diesen Aktionen basierenden Prozesse nicht enthalten, da sich sonst die Abhängigkeitsfunktionen widersprechen. Für die formalen Beweise wird an dieser Stelle auf [Sim00] und [Fid93] verwiesen.

Bevor wir die Definition des *direkten* und *indirekten Beweisverfahrens* angeben, ist die Definition von *negierten Modulen* erforderlich. Mit Hilfe der Definition der *Tautologie* (siehe Definition 5.15 auf Seite 73) und der Definition zur *Vervollständigung von Modulen* (siehe Definition 5.17 auf Seite 74) ist es nun möglich, die noch fehlende Definition von *negierten Modulen* aus dem Abschnitt 5.1 nachzuholen.

Definition 5.23 (Negation)
Sei $V[M]$ die Vervollständigung eines Moduls $M \in \mathbb{M}$ über der Aktionsmenge $\mathbb{A}$.

Für die Negation $\overline{M}$ gilt $\overline{M} \equiv \overline{V[M]}$ mit:

$$P[\overline{M}] := P[\top_{\mathbb{A}}] - P[V[M]]$$

Mit der Definition von *negierten Modulen* können wir nun für die Verifikation der Aktionslogikmodule das *direkte und indirekte Beweisverfahren* in der Definition 5.24 angeben. Für die formalen Beweise wird an dieser Stelle wieder auf [Sim00] und [Fid93] verwiesen.

Definition 5.24 (Direktes und indirektes Beweisverfahren)
Seien R und S zwei Module über der Aktionsmenge $\mathbb{A}$.

Das Modul R erfüllt das Modul S ($R \twoheadrightarrow S$) *genau dann, wenn die Anzahl der Prozesse in der Prozessmenge von* $(R \oslash S)$ *gleich der Anzahl der Prozesse in der Prozessmenge von* (R) *sind.*

Aus dieser Aussage können wir ein direktes *und ein* indirektes Beweisverfahren *ableiten. Die beiden Verfahren beschreiben, wann eine Realisierung (R) eine Spezifikation (S) erfüllt.*

- *Direktes Beweisverfahren:*

  $R \twoheadrightarrow S$ *wenn* $|P(R \oslash S)| = |P(R)|$
  $R \nrightarrow S$ *wenn* $|P(R \oslash S)| < |P(R)|$

- *Indirektes Beweisverfahren:*

  $R \twoheadrightarrow S$ *wenn* $|P(R \oslash \overline{S})| = 0$
  $R \nrightarrow S$ *wenn* $|P(R \oslash \overline{S})| > 0$

Die Beweisverfahren erlauben auf Basis der Anzahl der Prozesse des Moduls der Realisierung und der Anzahl der Prozesse des aus der Konjunktion der Module der Realisierung und der Spezifikation entstandenen Moduls $((R \oslash S))$ zu entscheiden, ob die Realisierung die Spezifikation erfüllt. Die größte Herausforderung ist hierbei die Anzahl der Prozesse zu bestimmen. Im folgenden Abschnitt beschreiben wir ein Verfahren, basierend auf der Darstellung der Module der Aktionslogik mit Hilfe von standardisierten Petri-Netzen, zur Bestimmung der Anzahl der Prozesse eines Moduls mit Hilfe von T-Invarianten.

5.3 Petri-Netz-Darstellung der Aktionslogik

Im diesem Abschnitt betrachten wir basierend auf der Idee von [Fid93], [Lau96] und [LS00] die Darstellung der Elemente der Aktionslogik mit Hilfe von Petri-Netzen. Die Darstellung der Module als Petri-Netz erlaubt, die Anzahl der Prozesse der Module mit Hilfe von T-Invarianten zu berechnen, da die Prozesse den Schaltfolgen in den Netzen entsprechen, welche die leere Markierung reproduzieren.

In der Petri-Netz-Darstellung $N[M]$ eines AL-Moduls M wird jede Aktion des Moduls durch eine Transition mit dem Namen der Aktion dargestellt. Als zusätzliche Transitionen enthält das Netz die Starttransition s (start-transition) ohne Vorstellen und die Zieltransition (goal-transition) g ohne Nachstellen. Die Prozesse der Prozessmenge $P[M]$ sind in dem Petri-Netz durch die Schaltfolgen des Netzes dargestellt, welche die leere Markierung reproduzieren können. In jeder Schaltfolgen müssen die Transitionen s und g genau einmal vorkommen.

Definition 5.25 (Netzdarstellung der Aktionslogik)
Das Modul $M \in \mathbb{M}$ mit der Prozessmenge $\mathbb{P}[M]$ wird durch die Netzdarstellung $N[M]$ (S/T-Netz $N = \{S, T, F, K, W\}$ mit leerer initialen Markierung ($M_0 = \{\}$)) dargestellt, wenn alle Prozesse von $\mathbb{P}[M]$ durch eine Schaltfolge des Netzes dargestellt werden können, welche die leere Markierung reproduziert.
Die Literale des Moduls M werden dabei auf Transitionen mit dem gleichem Namen abgebildet ($\sigma : \mathbb{L}(M) \rightarrow T$). Zusätzlich enthält die Transitionsmenge T des Netzes genau eine Starttransition s (${}^\bullet s = \emptyset$) und eine Zieltransition g ($g^\bullet = \emptyset$), die in den Schaltfolgen genau einmal vorkommen müssen.

Mit Hilfe der Darstellung eines AL-Module als Petri-Netz können wir entscheiden, ob die Prozessmenge des Moduls leer ist bzw. wie viele Prozesse in der Prozessmenge enthalten sind. Dabei wird die Anzahl der Prozesse durch die Anzahl der Schaltfolgen repräsentiert, welche die leere Markierung reproduzieren und in denen die Transitionen *s* und *g* genau einmal vorkommen. Die Prozesse stellen dabei Schleifen im Erreichbarkeitsgraph der S/T-Netze dar. Dies ermöglicht die Berechnung der Prozesse mit Hilfe von T-Invarianten (siehe Definition 4.23 auf Seite 57), welche durch das Lösen linearer homogener Gleichungssysteme bestimmt werden können. Das Gleichungssystem ergibt sich dabei aus der Inzidenzmatrix des Netzes.

Alle Module der Aktionslogik bestehen aus Literalen, die durch elementare bzw. zusammengesetzte Operatoren verbunden sind. Aus diesem Grund ist es ausreichend, die Netzdarstellungen von Modulen mit jeweils elementaren Operatoren ($\neg$, $\ominus$,$\oslash$ und $\ominus$) in Abbildung 5.2 auf der Seite 80 und zusammengesetzten Operatoren ($\oslash$, $\oslash$ und $\oplus$) in Abbildung 5.3 auf Seite 81 zu betrachten. Weiterhin können die Module auch Schleifen enthalten. Die entsprechenden Netzdarstellungen sind in Abbildung 5.4 auf Seite 82 dargestellt. Die Darstellung von komplexen Modulen mit Hilfe von S/T-Netzen ergibt sich aus der Kombination dieser Darstellungen.

Wir beginnen die Betrachtung mit der Netzdarstellung für das Modul [$\perp$] in der Abbildung 5.1. Das Modul [$\perp$] enthält keine Aktionen und die Prozessmenge ist leer. Entsprechend sind in der Netzdarstellung $N[\perp]$ nur die beiden Transitionen *s* und *g* enthalten und das Netz besitzt keine T-Invariante, da die leere Markierung nicht reproduzierbar ist.

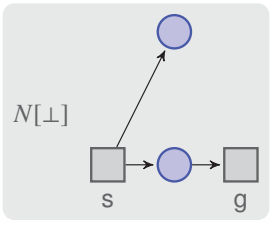

Abbildung 5.1: Netzdarstellung von [$\perp$]

Die Prozessmengen von Modulen mit nur elementaren Operatoren ($\neg$, $\ominus$,$\oslash$ und $\ominus$) bestehen jeweils nur aus einem einzigen Prozess. Entsprechend haben alle Netzdarstellungen dieser Module in Abbildung 5.2 genau eine realisierbare T-Invariante.

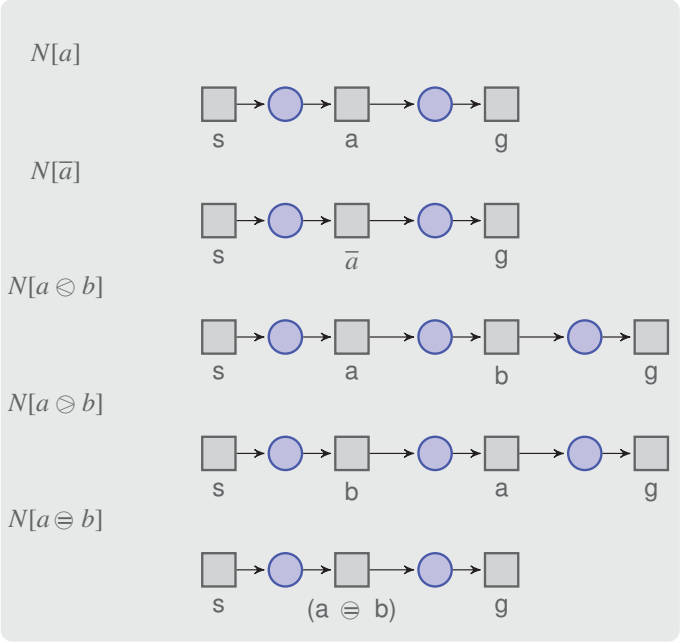

Abbildung 5.2: Netzdarstellung von Modulen mit elementaren Operatoren

Besteht ein AL-Modul nur aus der Aktion a, so enthält die entsprechende Netzdarstellung die Transitionen s, a und g (vergleiche $N[a]$ in Abbildung 5.2). Die leere Markierung wird reproduziert durch das Schalten der Transitionen in der Reihenfolge s, a und g.

Eine negierte Aktion wird auch durch eine Transition in der Netzdarstellung dargestellt. Ein Beispiel dazu gibt das Netz $N[\overline{a}]$ in Abbildung 5.2. Die Prozessmenge des zugehörigen Moduls enthält nur den Prozess $(\overline{a})$ und entsprechend gibt es in dem Netz nur eine Schaltfolge $(s, \overline{a}, g)$, welche die leere Markierung reproduzieren kann.

Das Modul $M_1 = [a \oslash b]$ wird durch den Prozess $(a \oslash b)$ realisiert. In der zugehörigen Schaltfolge der Netzdarstellung $N[a \oslash b]$, welche die leere Markierung reproduziert, schalten die Transitionen in der Reihenfolge s, a, b und g. Das Netz hat die T-Invariante $(1, 1, 1, 1)^T$ mit $(s, a, b, g)^T$.

Die Schaltfolgen und T-Invarianten der beiden anderen Netzdarstellungen $N[a \oslash b]$ und $N[a \ominus b]$ ergeben sich entsprechend.

Bei der Netzdarstellung eines Moduls mit zusammengesetzten Operatoren kann es mehrere Schaltfolgen geben, welche die leere Markierung reproduzieren, da die Prozessmenge mehrere Prozesse enthalten kann. Die Abbildung 5.3 enthält die Netzdarstellung von Modulen mit zusammengesetzten Operatoren.

Die Prozessmenge von $M_2 = [a \oplus b]$ besteht aus zwei Prozessen (a) und (b). Somit muss es auch in $N[a \oplus b]$ zwei Schaltfolgen geben. In der ersten Schaltfolge schaltete s vor a und g. Die zweite Schaltfolge enthält die Transitionen s, b und g in dieser Reihenfolge.

Die zugehörigen T-Invarianten des Netzes $N[a \oplus b]$ sind $(1, 1, 0, 1)^T$ und $(1, 0, 1, 1)^T$, welche durch das Lösen des linearen Gleichungssystems $[a \oplus b] * t = 0$, basierend auf der Inzidenzmatrix (5.1), bestimmt werden.

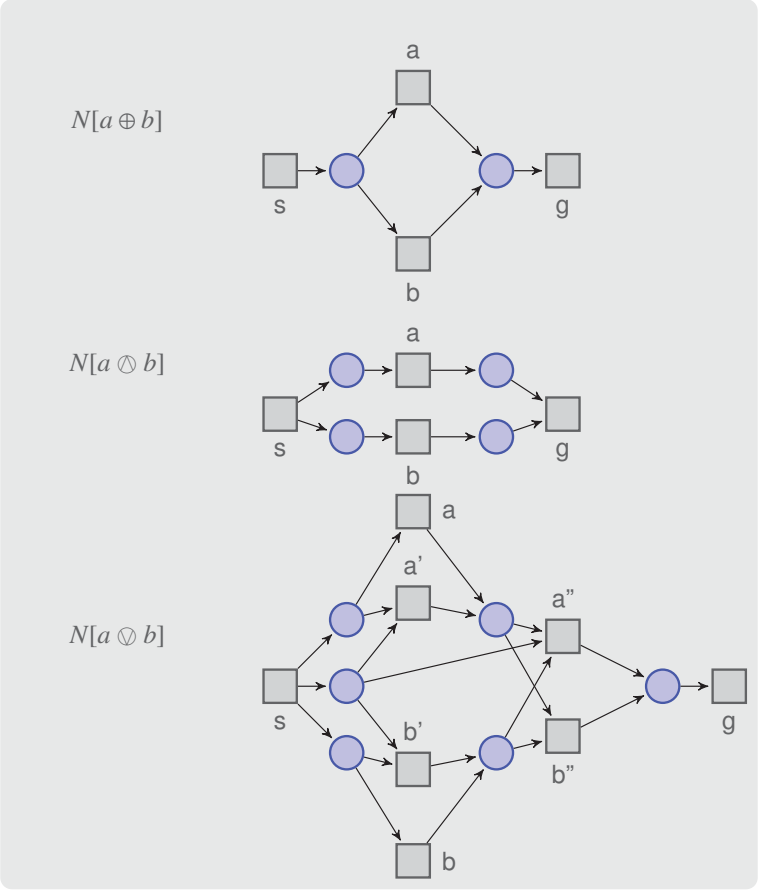

Abbildung 5.3: Netzdarstellung von Modulen mit zusammengesetzten Operatoren

$$[a \oplus b] = \begin{matrix} \\ P_0 \\ P_1 \end{matrix} \begin{matrix} \begin{matrix} s & a & b & g \end{matrix} \\ \begin{pmatrix} 1 & -1 & -1 & 0 \\ 0 & 1 & 1 & 1 \end{pmatrix} \end{matrix} \tag{5.1}$$

Das Netz $N[a \barwedge b]$ hat nur eine minimale T-Invariante $(1,1,1,1)^T$, da in einer T-Invarianten die Reihenfolge in der die Transitionen a und b schalten nicht dargestellt werden kann. Die Prozessmenge von $[a \barwedge b]$ besteht aus drei Prozessen $(\{(a \circledcirc b), (a \circledast b), (a \circledcirc b)\})$, welche auch in der Netzdarstellung enthalten sind. Es kann entweder die Transition a vor b oder a nach b bzw. die Transitionen a und b können gleichzeitig schalten, um die leere Markierung zu reproduzieren.

Die Netzdarstellung von $M_3 = [a \barvee b]$ ist etwas komplexer als die vorherigen, da in dem S/T-Netz noch zusätzliche Transitionen vorkommen (a', a'', b' und b''). $\mathbb{P}[M_3]$ besteht aus den Prozessen $\{(a), (b), (a \circledcirc b), (a \circledast b), (a \circledcirc b)\}$. Damit in dem S/T-Netz auch die leere Markierung durch die Schaltfolgen, welche die Transition a enthält aber nicht b und die Schaltfolge, die die Transition b enthält aber nicht a reproduziert werden kann, sind die zusätzlichen Transitionen notwendig. Für das Netz ergeben sich insgesamt fünf Schaltfolgen, welche die leere Markierung reproduzieren.

Die Abbildung 5.4 zeigt die Netzdarstellung der Module $[a^n]$, $[a^+]$ und $[a^*]$. In $N[a^n]$ ist durch die Angabe des Kantengewichts n ($n \in \mathbb{N}_0$) das Schalten der Transition a auf n limitiert, da nach unserer Definition die Transition s nur einmal schalten darf. Somit hat das Netz die T-Invariante $(1, n, 1)^T$. In der Netzdarstellung von $[a^+]$ muss die Transition a mindestens einmal schalten um die leere Markierung zu reproduzieren. Wohingegen bei $N[a^*]$ die Transition a beliebig oft (auch 0-mal) schalten kann.

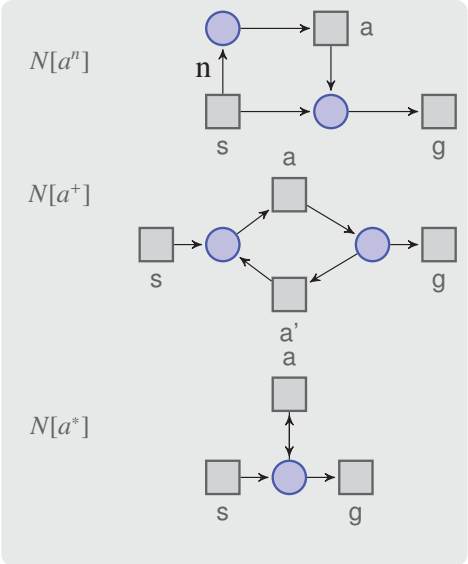

Abbildung 5.4: Netzdarstellung von Modulen mit Schleifen

5.4 Verifikation mit Hilfe der Petri-Netz-Darstellung

Im Abschnitt 5.2 haben wir ein Verfahren vorgestellt, welches es erlaubt, basierend auf der Anzahl der Prozesse von einem AL-Modul zu entscheiden, ob ein Modul (Realisierung) ein zweites Modul (Spezifikation) erfüllt. Dies ist der Fall, wenn die kausalen Abhängigkeiten der Aktionen in den Prozessen der Realisierung den kausalen Abhängigkeiten der Aktionen in den Prozessen in der Spezifikation entsprechen. Wie dargestellt, besteht die Schwierigkeit darin, die Anzahl der Prozesse der Module zu berechnen. Durch die Darstellung der Module mit Hilfe von Petri-Netzen (siehe Abschnitt 5.3) ist es möglich, durch das Lösen von linearen homogenen Gleichungssystemen (basierend auf den Inzidenzmatrizen der Netze) die Anzahl der Prozesse zu bestimmen. In diesem Abschnitt beschreiben wir basierend auf [Fid93], [Lau02] die benötigten Schritte und demonstrieren diese an einem Beispiel.

Nach der Definition 5.24 auf Seite 78 ist für die Verifikation der Module die Anzahl der Prozesse der Realisierung (R) und der Konjunktion der Module der Realisierung und der Spezifikation ($R \owedge S$) zu bestimmen. Für die Verifikation der AL-Module auf Basis von Petri-Netzen muss die Konjunktion der Netzdarstellungen der Realisierung und der Spezifikation erstellt werden. Die dazu notwendigen Schritte sind im Algorithmus 5.1 beschrieben.

Algorithmus 5.1 (Konjunktion von zwei Netzdarstellungen)
Seien M_1 und M_2 zwei Module der Aktionslogik und $N[M_1]$ und $N[M_2]$ die entsprechende Petri-Netz-Darstellung der Module.
Die Konjunktion der beiden Module $M_1 \oslash M_2$ wird wie folgt beschrieben erzeugt:

1. *Schritt: Identifikation der beiden Starttransitionen (s_1 und s_2) und der beiden Zieltransitionen (g_1 und g_2). Die beiden Transitionen werden jeweils zu einer neuen Starttransition s bzw. zu einer neuen Zieltransition g synchronisiert. (Siehe Abbildung 5.5)*
2. *Schritt: Identifikation aller Transitionen a, die in beiden Netzen $N[M_1]$ und $N[M_2]$ vorkommen und synchronisiere diese. (Siehe Abbildung 5.6)*
3. *Schritt: In dem Netz dürfen alle Transitionen a und alle Transitionen $\overline{a}$ nicht im gleichen Prozess vorkommen. Daher muss der gegenseitig Ausschluss der beiden Transitionen garantiert werden. Aus diesem Grund wird zu diesen Transitionen eine gemeinsame Vorstelle hinzugefügt. (Siehe Abbildung 5.7)*
4. *Schritt: Im letzten Schritt werden alle redundanten Stellen des neuen Netzes gelöscht. (Siehe Abbildung 5.8)*

Die folgenden vier Abbildungen (Abbildung 5.5 bis Abbildung 5.8) verdeutlichen die einzelnen Schritte des Algorithmus 5.1.

In der Abbildung 5.5 ist die Identifikation der Start- und Zieltransitionen der Netzdarstellungen $N[M_1]$ und $N[M_2]$ und die Synchronisation zu einer neuen Start- bzw. Zieltransition (s und g) dargestellt.

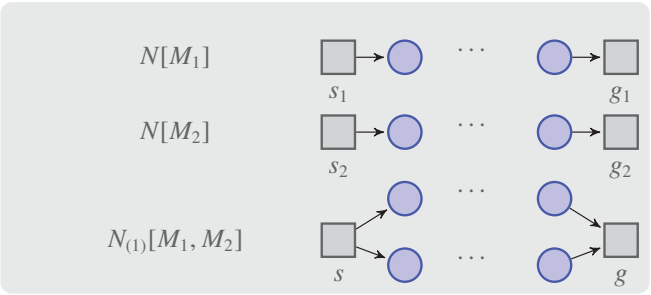

Abbildung 5.5: Identifikation und Synchronisation der Start- und Zieltransitionen

In der Abbildung 5.6 ist die Identifikation von gleichen Transitionen in den beiden Netzdarstellungen und die Synchronisation dieser Transitionen dargestellt. In diesem Beispiel ist in beiden Modulen die Transition a enthalten.

In der Abbildung 5.7 ist die Identifikation der Transitionen und der negierten Transitionen in beiden Netzdarstellungen und der gegenseitige Ausschluss dargestellt. In diesem Beispiel ist in einem Modul die Transition a und in dem anderen Modul die Transition $\overline{a}$ enthalten.

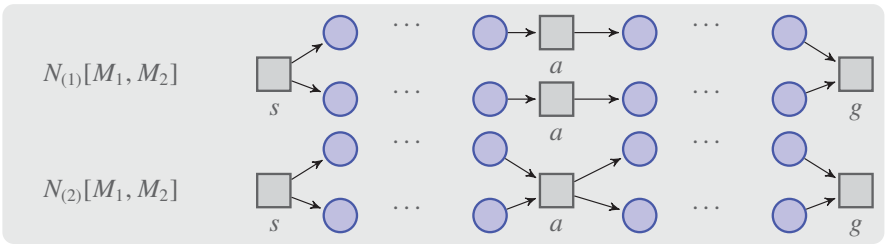

Abbildung 5.6: Identifikation und Synchronisation gleicher Transitionen

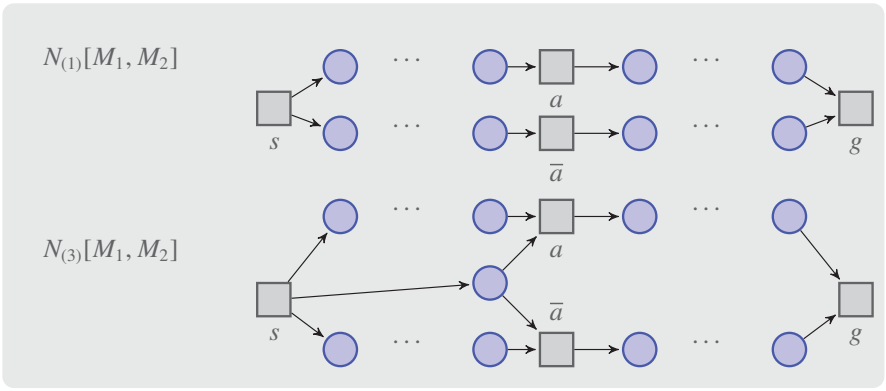

Abbildung 5.7: Wechselseitiger Ausschluss von einer Aktion und deren Negation

In der Abbildung 5.8 ist das Löschen von redundanten Stellen in der Netzdarstellung dargestellt. In diesem Beispiel kann eine Stelle zwischen den Transitionen *a* und *b* entfernt werden, da sowohl die Vortransitionen als auch die Nachtransitionen von beiden Stellen exakt gleich ist.

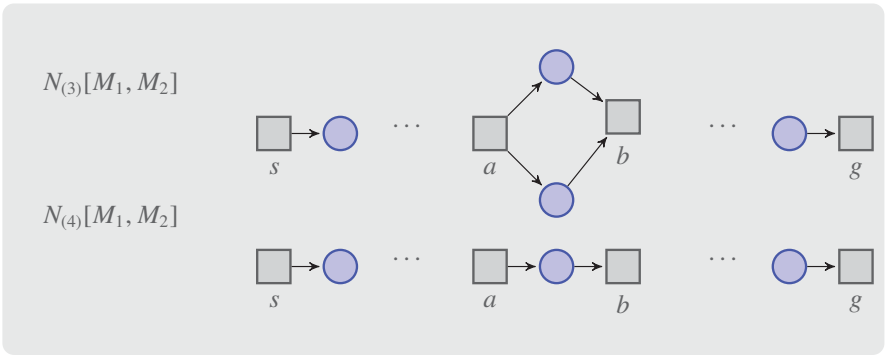

Abbildung 5.8: Löschen von redundanten Stellen

Durch die Darstellung der Module der Aktionslogik mit Hilfe von Petri-Netzen kann die Verifikation auf Basis des Vergleichs der Anzahl der Prozesse (Definition 5.24) anschaulich verdeutlicht werden.

Betrachten wir dazu die beiden Netzdarstellungen der Realisierung R und der Spezifikation S in Abbildung 5.9. Beide Module enthalten die Aktionen a und b und besitzen jeweils einen Prozess, welcher beide Aktionen enthält.

Die kausale Ordnung der Aktionen in der Spezifikation (Aktion b *vor* Aktion a) wurde entsprechend gewählt (genau entgegengesetzt zur Realisierung mit Aktion a *vor* Aktion b), damit die Realisierung die Spezifikation nicht erfüllt. Dieser Gegensatz der kausalen Abhängigkeiten der Aktionen wird in der Konjunktion der beiden Netzdarstellungen ($N[R \owedge S]$) dadurch deutlich, dass die Transition a nicht schalten kann. Damit auf allen Vorstellen $p1$ und $p4$ von a ein Token liegen kann, muss zuvor die Transition b schalten. Das Schalten dieser Transition ist aber abhängig vom Schalten der Transition a, da ein Token auf der Stelle $p2$ erforderlich ist. Somit ergibt sich an dieser Stelle eine Verklemmung und die leere Markierung kann nicht reproduziert werden. Somit ergibt sich in der Netzdarstellung keine Schaltfolge, welche die leere Markierung reproduzieren kann.

Daraus folgt, dass die Realisierung die Spezifikation nicht erfüllt, da

$$|P(R)| = 1 \neq |P(R \owedge S)| = 0.$$

Allgemein ergibt sich bei der Konjunktion der Netzdarstellung der Realisierung und der Spezifikation immer dann eine Verklemmung, wenn die kausale Ordnung der Aktionen in der Realisierung und der Spezifikation unterschiedlich sind.

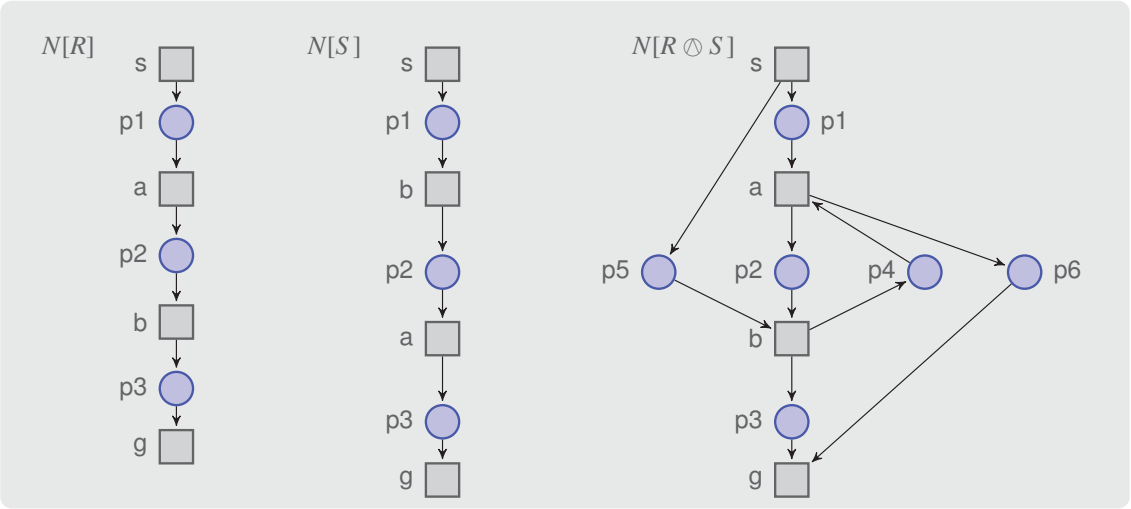

Abbildung 5.9: Netzdarstellung der Realisierung, Spezifikation und deren Konjunktion

Nach der Definition der Konjunktion der Netzdarstellungen und der Illustration der Konjunktion zweier Netzdarstellungen anhand des Beispiels beschreiben wir im Algorithmus 5.2 alle notwendigen Schritte zur Verifikation der LA-Module mit Hilfe der Petri-Netz-Darstellung.

Algorithmus 5.2 (Schritte der Verifikation auf Basis der Netzdarstellungen)
Für die Überprüfung, ob ein Modul R (Realisierung) ein Modul S (Spezifikation) erfüllt, sind die folgenden Schritte notwendig:

1. *Schritt: Definition der Realisierung und Spezifikation als AL-Modul.*
2. *Schritt: Für den Vergleich der Module müssen die Module wie in Definition 5.17 auf Seite 74 beschrieben vervollständigt werden, so dass in jedem Prozess jede Aktion der Aktionsmenge des Moduls vorkommt. Daraus ergeben sich die Module $V[R]$ und $V[S]$.*
3. *Schritt: Nach der Vervollständigung der beiden einzelnen Module sind beide wechselseitig zu vervollständigen, so dass beide Module auf der gleichen Aktionsmenge basieren (siehe Definition 5.20 auf Seite 76).*
   Es ergeben sich daraus die Module $R_V = V_S[R]$ und $S_V = V_R[S]$.
4. *Schritt: Erzeugen der Netzdarstellung von R_V und S_V.*
5. *Schritt: Für die Entscheidung, ob die Realisierung die Spezifikation nach dem direkten Beweisverfahren $R \twoheadrightarrow S$ wenn $|\mathbb{P}(R \oslash S)| = |\mathbb{P}(R)|$ (siehe Definition 5.24 auf Seite 78) erfüllt, wird die Netzdarstellung von $(R_V \oslash S_V)$ benötigt. Diese wird aus den Netzdarstellungen von R_V und S_V mit Hilfe des Algorithmus 5.1 bestimmt.*
6. *Schritt: Bestimme mit Hilfe der realisierbaren T-Invarianten der S/T-Netze der Realisierung und der Vereinigung der Realisierung und der Spezifikation diejenigen Schaltfolgen, welche die leere Markierung reproduzieren. Diese entsprechen genau den Prozessen der zugehörigen Module.*
7. *Schritt: Ist $|\mathbb{P}(R \oslash S)| = |\mathbb{P}(R)|$, so erfüllt die Realisierung die Spezifikation, sonst nicht.*

Als Beispiel für die Verifikation mit Hilfe der Netzdarstellung untersuchen wir, ob das Modul $R = [a \oplus b]$ das Modul $S = [b]$ erfüllt.

Durch die Spezifikation S wird ausgedrückt, dass in jedem Prozess die Aktion b vorkommen muss ($\mathbb{P}_S = \{(b)\}$). Bei der Realisierung R kann entweder die Aktion a oder die Aktion b in einem Prozess vorkommen, aber nie beide gleichzeitig ($\mathbb{P}_R = \{(a), (b)\}$). Aufgrund dieser Betrachtung kann die Realisierung die Spezifikation nie erfüllen, da nicht in jedem Prozess der Realisierung die Aktion b enthalten ist. Wir belegen diese Aussage nun mit Hilfe der Netzdarstellung der beiden Module mit Hilfe der im Algorithmus 5.2 beschriebenen Schritte.

Die Netzdarstellung $N[S]$ der Spezifikation $S = [b]$ ist in Abbildung 5.10 und die Netzdarstellung $N[R]$ der Realisierung $R = [a \oplus b]$ ist in Abbildung 5.11 dargestellt. Für die Überprüfung ob die Realisierung die Spezifikation erfüllt, ist die Anzahl der Prozesse der Realisierung, also jene Schaltfolgen der Netzdarstellung $N[R]$, die die leere Markierung reproduzieren, von Interesse. Zum einen wird die leere Markierung durch die Schaltfolge (s, a, g) und zum anderen durch die Schaltfolge (s, b, g) reproduziert. Somit enthält die Prozessmenge von R zwei Prozesse.

Im nächsten Schritt wird durch die Vervollständigung der Module sichergestellt, dass in jedem Prozess alle Aktionen der zugehörigen Aktionsmenge enthalten sind. Für die Spezifikation ergibt

sich dann das Netz $N[V[S]]$ (siehe Abbildung 5.10) und für die Realisierung die Netzdarstellung $N[V[R]]$ (siehe Abbildung 5.11).

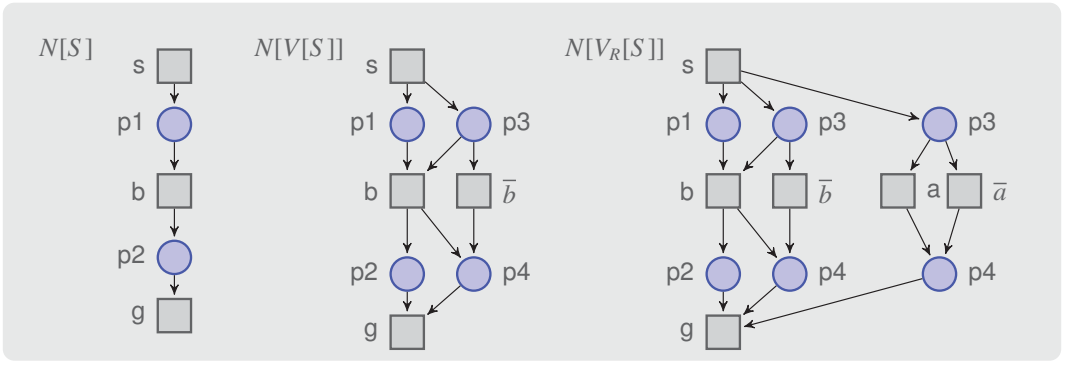

Abbildung 5.10: Netzdarstellung der Spezifikation

Für den Vergleich der AL-Module der Realisierung und der Spezifikation ist es notwendig die beiden Module wechselseitig zu vervollständigen (3. Schritt des Algorithmus 5.2). Die wechselseitige Vervollständigung der Spezifikation in Bezug auf die Realisierung ($N[V_R[S]]$) ist in Abbildung 5.10 dargestellt. Die Netzdarstellung der Vervollständigung der Realisierung in Bezug auf die Spezifikation ($N[V_S[R]]$) unterscheidet sich nicht von der Vervollständigung der Realisierung ($N[V[R]]$), da die Aktionsmenge der Spezifikation eine echte Teilmenge der Aktionsmenge der Realisierung ist. Auch für die Netzdarstellung des wechselseitig vervollständigten Moduls $V_S[R]$ ermitteln wir die Anzahl der Schaltfolgen, welche die leere Markierung reproduzieren. In diesem Falle ergeben sich sechs Schaltfolgen $(s,a,\overline{b},g)$,$(s,\overline{b},a,g)$,$(s,b,\overline{a},g)$, $(s,\overline{a},b,g)$ und jeweils die beiden Schaltfolgen in denen die Transitionen a und $\overline{b}$ bzw. $\overline{a}$ und b gleichzeitig schalten.

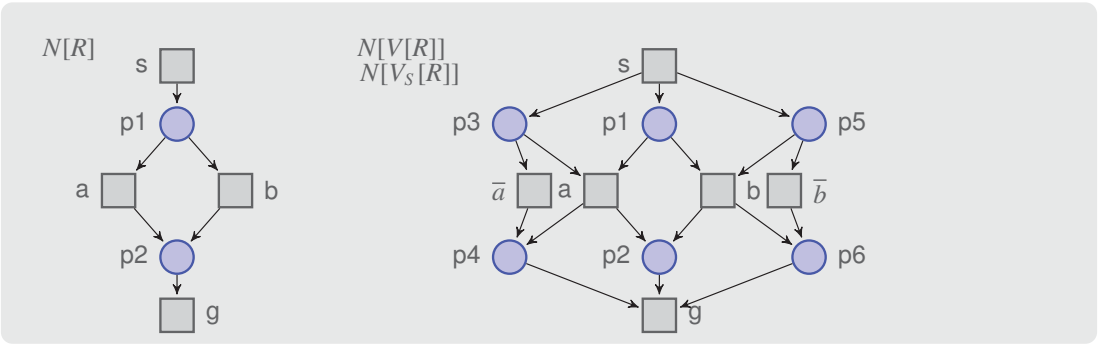

Abbildung 5.11: Netzdarstellung der Realisierung

Für den direkten Beweis ist es notwendig im nächsten Schritt die Netzdarstellung der Konjunktion der Realisierung und der Spezifikation mit Hilfe des Algorithmus 5.1 zu erstellen. In Abbildung 5.12 ist zum einen die Netzdarstellung $N[R \oslash S]$ der Konjunktion der Module R und S sowie die Netzdarstellung $N[V_S[R] \oslash V_R[S]]$ der wechselseitig vervollständigten Module $V_S[R]$ und $V_R[S]$ dargestellt.

An dieser Stelle betrachten wir für beide Netzdarstellungen diejenigen Schaltfolgen, welche die leere Markierung reproduzieren, wobei die Transition s und g genau einmal vorkommen. Für

$N[R \oslash S]$ ergibt sich nur eine Schaltfolge (s, b, g). Bei $N[V_S[R] \oslash V_R[S]]$ sind es vier Schaltfolgen: $(s, b, \overline{a}, g)$, $(s, \overline{a}, b, g)$, $(s, b, \overline{a}, g)$ sowie die Schaltfolge in der b und $\overline{a}$ gleichzeitig schalten.

Ausgehend von dem direkten Beweisverfahren $|\mathbb{P}(R \oslash S)| = |\mathbb{P}(R)|$ ergibt sich, dass die Realisierung die Spezifikation nicht erfüllt, da

$$|\mathbb{P}(R_V \oslash S_V)| = 4 \text{ und } |\mathbb{P}(R_V)| = 6$$

und somit ist

$$|\mathbb{P}(R_V \wedge S_V)| \neq |\mathbb{P}(R_V)|.$$

Sind in einem konkreten Anwendungsfall der Verifikation nur die Ausführung der Aktionen von Interesse (keine negierten Aktionen), so ist eine Betrachtung der negierten Aktionen für die Verifikation nur dann erforderlich, wenn in der Realisierung nicht alle Aktionen der Aktionsmenge der Spezifikation enthalten sind. In diesem Fall kann auf die jeweilige Vervollständigung der Netze verzichtet werden. In diesem Beispiel ergibt sich dann

$$|\mathbb{P}(R \wedge S)| = 1 \text{ sowie } |\mathbb{P}(R)| = 2$$

und somit erfüllt, wie erwartet, die Realisierung die Spezifikation nicht.

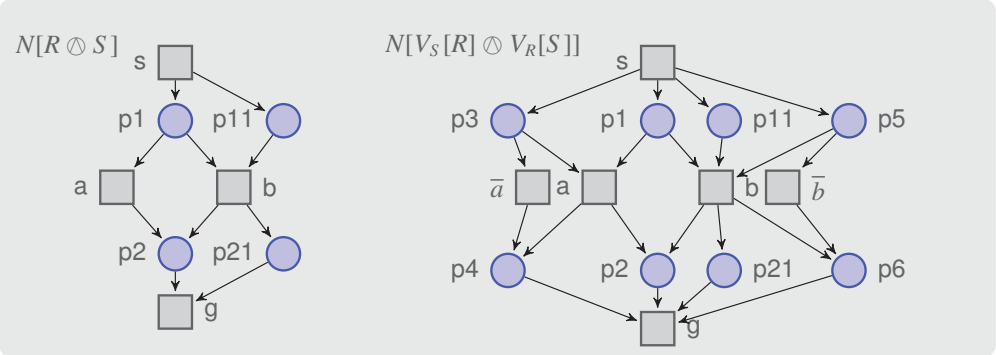

Abbildung 5.12: Vereinigung der Realisierung und Spezifikation $N[R \oslash S]$

In [Sim00], [LS00], [Fid93], [Thi10] sind weitere Beispiele für die Überprüfung zweier Module der Aktionslogik mit Hilfe der Petri-Netz-Darstellung und dem direkten und indirekten Beweisverfahren angegeben.

6 Verifikation der EXAM-Testmodelle

Im Kapitel 3 haben wir gezeigt, welchen Einfluss fehlerhafte Testfälle auf die Testausführung am HIL-Prüfstand und die gemeinsame Testfallentwicklung im Team haben. Durch die steigende Komplexität der Testfälle kann die Qualität der Testfälle manuell nur mit sehr großen Aufwand sichergestellt werden. Die zunehmende Anzahl der zu testenden Funktionen macht es jedoch notwendig die Qualität der EXAM-Testfälle zu erhöhen, sodass eine unnötige Wiederholung der Testfälle am Prüfstand aufgrund falscher Aufrufreihenfolgen der EXAM-Operationen (semantische Fehler) reduziert werden kann. Ebenso wird durch eine einheitliche Modellierung die gemeinsame Entwicklung und Wiederverwendung der Testfälle erleichtert bzw. erst ermöglicht.

Zur Steigerung der Testfallqualität in EXAM stellen wir in diesem Kapitel ein Verfahren basierend auf dem in Kapitel 5 eingeführten Verifikationsverfahren der Aktionslogik mit Hilfe von Petri-Netzen vor. Dazu verdeutlichen wir im ersten Schritt den Zusammenhang zwischen den EXAM-Elementen und den Elementen der Aktionslogik. Im zweiten Schritt stellen wir die notwendigen Schritte für die Überprüfung der Testabläufe in EXAM mit Hilfe der Aktionslogik vor. Dazu beschreiben wir im Abschnitt 6.3 die Transformation der EXAM-Elemente in eine korrespondierende Petri-Netz-Darstellung. Aufgrund der Besonderheiten bei der Funktionsabsicherung mit Hilfe von EXAM sind unterschiedliche Arten von Spezifikationen zu berücksichtigen. Diese definieren wir im letzten Teil des Abschnittes und beschreiben die notwendigen Erweiterungen der Verifikationsmethode sowie der Petri-Netz-Darstellung und demonstrieren die Anwendung anhand einiger Beispiele.

6.1 Verifikation auf Basis der Aktionslogik

Viele Probleme bei der Testausführung bzw. bei der gemeinschaftlichen Entwicklung und Wiederverwendung von Testfragmenten liegen darin begründet, dass bei der Erstellung der Testabläufe die kausalen Beziehungen der Bibliotheksfunktionen nicht berücksichtigt werden.

Im Kapitel 3 haben wir die Bedingungen, welche die Grundlage für die kausalen Abhängigkeiten der Bibliotheksfunktionen sind, in die drei Kategorien

- domänspezifische Bedingungen,
- domänenübergreifende Bedingungen und
- Bedingungen basierend auf den Modellierungsrichtlinien

eingeteilt.

Hierbei beschreiben die *domänenspezifischen Bedingungen* Eigenschaften des zu testenden Objektes, also im Falle von EXAM des Fahrzeugs bzw. der Fahrzeugfunktionen. Unter der Bezeichnung *domänenübergreifende Bedingungen* haben wir alle Eigenschaften der Testumgebung wie z. B. der benötigten Prüfhardware (Diagnosetester, Datenlogger etc.) zusammengefasst. Alle Bedingungen an einen Testablauf in EXAM, welche durch die Modellierungsrichtlinien bzw. die Modellverantwortlichen gestellt werden, sind unter der Kategorie *Bedingungen basierend auf den Modellierungsrichtlinien* zusammengefasst.

Ziel ist es, die durch die Verletzung der kausalen Abhängigkeiten der Bibliotheksfunktionen entstehenden Probleme bei der Testfallerstellung und -ausführung zu reduzieren und somit die Effizienz des gesamten Testprozesses zu steigern. Dazu ist es notwendig, zum einen die kausalen Beziehungen zwischen den EXAM-Operationen formal zu spezifizieren und zum anderen die spezifizierten Bedingungen mit Hilfe eines entsprechenden Verfahrens automatisiert zu prüfen.

In den Artikeln *Verification of Functional ECU Test Cases* [Thi10] und *Petri Net Based Verification of Causal Dependencies in Electronic Control Unit Test Cases* [TD11] haben wir erste Überlegungen vorgestellt, wie mit Hilfe der Aktionslogik die Testabläufe in EXAM überprüft werden können. Basierend auf diesen Vorüberlegungen untersuchen wir im weiteren Verlauf dieser Arbeit, wie die in der Aktionslogik definierten Elemente und die Beweisverfahren auf das Testautomatisierungssystem EXAM übertragbar sind und welche notwendigen Änderungen erforderlich sind.

Die im Kapitel 5 vorgestellte Aktionslogik erlaubt, die kausalen Abhängigkeiten von Aktionen formal zu beschreiben. Durch die Darstellung der Module der Aktionslogik in Form von Petri-Netzen und dem in der Definition 5.24 auf Seite 78 vorgestellten *direkten* bzw. *indirekten Beweisverfahren* kann überprüft werden, ob ein Modul bzw. genauer ob die kausale Ordnung der Aktionen in den Prozessen des Moduls den spezifizierten kausalen Abhängigkeiten entsprechen.

In EXAM bilden die Operationen der Funktionsbibliothek die Schnittstelle zum Prüfstand bzw. zur jeweiligen verwendeten Hard- oder Software des Prüfstands. Aus Sicht der EXAM-Anwender wird durch den Aufruf einer Operation der Funktionsbibliothek innerhalb eines Sequenz- oder Aktivitätsdiagramms genau eine Aktion zur Laufzeit des Testfalls ausgeführt. Die aus Sicht des Anwenders atomaren Aktionen der Funktionsbibliothek können, wie wir im weiterem Verlauf der Arbeit zeigen werden, wiederum durch weitere Operationen der Funktionsbibliothek von EXAM realisiert sein. Die beiden Diagrammtypen (Sequenz- bzw. Aktivitätsdiagramm) erlauben dem Anwender durch Interaktionsfragmente und Kontrollknoten verschiedene Ausführungspfade in einem Testfall zu modellieren, wobei zur Laufzeit immer nur genau ein Ausführungspfad ausgeführt wird. Der Ausführungspfad ist zum einen durch die Parametrierung des Testfalls und zum anderen auch von dem aktuellen Systemzustands des Prüfstands abhängig. Wie im Abschnitt 2.4 beschrieben, wird ein in sich abgeschlossener Testfall in EXAM durch einen *TestCase* repräsentiert, der wiederum eine beliebige Anzahl an Testfallfragmente (*TestSequence* oder *TestActivities*) referenzieren kann. Mehrere Testfälle eines Testthemas werden in einer *TestSuite* gruppiert.

Die Zuordnung der EXAM-Elemente zu den Elementen der Aktionslogik erfolgt vergleichbar zu der im Abschnitt 5.1 beschriebenen Zuordnung zwischen den Elementen der Aktionslogik und den Elementen der Aussagenlogik. Die Tabelle 6.1 enthält eine Gegenüberstellung der Elemente der Aktionslogik und den Elementen von EXAM.

Aktionslogik	EXAM
Aktion	Operation
Prozess	Kontrollpfad bzw. Ausführungspfad eines Testablaufes
Modul	TestSequence, TestActivity, TestCase oder TestSuite

Tabelle 6.1: Vergleich der Elemente der Aktionslogik und der Elemente von EXAM

Die Operationen der Funktionsbibliothek von EXAM werden den Aktionen der Aktionslogik zugeordnet, da die Operationen aus Sicht der Anwender innerhalb des Testablaufes jeweils genau eine abgeschlossene Aktion repräsentieren. Zwischen diesen Aktionen bestehen kausale Abhängigkeiten, die zur Laufzeit des Testfalls nicht verletzt werden dürfen. Im weiteren Verlauf verwenden wir die Begriffe Aktion und Operation daher synonym. Den Modulen der Aktionslogik entsprechen die EXAM-Elemente *TestSequence*, *TestActivity*, *TestCase* und *TestSuite*. In EXAM erlauben diese Elemente den Testablauf zu beschreiben und die Ausführungsreihenfolge der Operationen zu definieren. Der mit Hilfe dieser Elemente definierte Testablauf kann jedoch mehrere mögliche Ausführungspfade enthalten. Bei der Testausführung wird genau ein Kontroll- bzw. Ausführungspfad ausgeführt. Die Ausführungspfade entsprechen somit den Prozessen der Aktionslogik.

Für die Überprüfung, ob ein Testablauf eine spezifizierte Bedingung erfüllt, ist es notwendig, dass die kausalen Beziehungen der Operationen der Funktionsbibliothek zuvor formal spezifiziert sind. Für die Anwendung des *direkten* oder *indirekten Beweisverfahrens* müssen diese Spezifikationen in Form von AL-Modulen dargestellt werden. Nach der Definition der Spezifikationen als AL-Modul und der Darstellung der Module in Form eines Petri-Netzes, kann durch die Anwendung des Algorithmus 5.2 überprüft werden, ob die Testabläufe die spezifizierten Bedingungen erfüllen. Dieser Ansatz ermöglicht uns sicherzustellen, dass die spezifizierten kausalen Beziehungen der Operationen der Funktionsbibliothek im Testablauf eingehalten werden.

In Abhängigkeit des EXAM-Elements (*TestSequence*, *TestActivity*, *TestCase* oder *TestSuite*) kann mit diesem Ansatz nun überprüft werden, ob einzelne Teile des Testablaufs (*TestSequence* oder *TestActivity*) die kausale Ordnung der Operationen nicht verletzen oder ob in dem gesamten Testfall (*TestCase*) die definierten Bedingungen eingehalten sind. Durch die Überprüfung der kausalen Ordnungen in einer *TestSuite* wird sichergestellt, dass die spezifizierten Bedingungen bei der Durchführung aller enthaltener Testfälle nicht verletzt werden.

Nach der Zuordnung der EXAM-Elemente zu den Elementen der Aktionslogik betrachten wir im Folgenden, welche einzelnen Schritte für die Überprüfung notwendig sind.

6.2 Notwendige Schritte der Verifikation

In der Abbildung 6.1 sind die benötigten Schritte für die Überprüfung der EXAM-Testabläufe schematisch dargestellt.

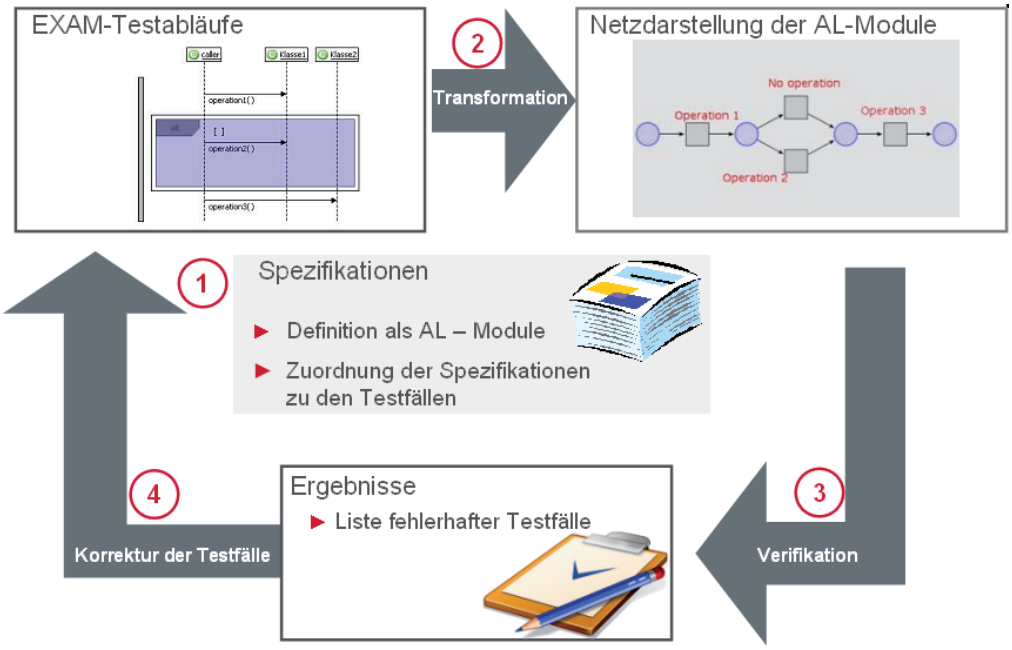

Abbildung 6.1: Vorgehen bei der Verifikation der EXAM-Testabläufe

Im *ersten Schritt* muss die kausale Ordnung zwischen den EXAM-Bibliotheksfunktionen mit Hilfe von Modulen der Aktionslogik spezifiziert und die entsprechenden Netzdarstellungen der Module erstellt werden. Für die Durchführung der Überprüfung der Testabläufe müssen diese Spezifikationen den entsprechenden EXAM-Elementen (*TestSequence*, *TestActivity*, *TestCase* oder *TestSuite*) zugeordnet werden, welche die jeweilige Bedingung erfüllen sollen.

Zur Durchführung der Verifikation müssen alle EXAM-Elemente, denen eine Spezifikation zugeordnet wurde, in Form eines Moduls der Aktionslogik in Petri-Netz-Darstellung vorliegen. Daher sind im *zweiten Schritt* alle betroffenen Testabläufe in eine korrespondierende Petri-Netz-Darstellung zu transformieren. Das Ergebnis des ersten und zweiten Schritts ist, dass nun sowohl die Realisierung (Testabläufe) als auch die Spezifikationen in Form einer Petri-Netz-Darstellung eines Moduls der Aktionslogik vorliegen.

Im *dritten Schritt* kann die eigentliche Verifikation der Testabläufe auf Basis der Anzahl der möglichen Ausführungspfade mit Hilfe des Algorithmus 5.2 auf Seite 86 durchgeführt werden. Dazu wird in der Regel das *direkte Beweisverfahren* (Definition 5.24, Seite 78) verwendet. Wie in dem vorherigen Kapitel beschrieben, basiert das *direkte Beweisverfahren* darauf, die Anzahl der Prozesse der Realisierung ($|\mathbb{P}(R)|$) und die Anzahl der Prozesse der Vereinigung der Realisierung und der Spezifikation ($|\mathbb{P}(R \oslash S)|$) zu vergleichen. Für die Anwendung des indirekten Beweisverfahrens, ist das Modul der Spezifikation zu negieren (vergleiche Definition 5.24 auf Seite 78). In Abhängigkeit der Komplexität der Spezifikation kann die negierte Darstellung des zugehörigen Moduls nur mit großem Aufwand erzeugt werden und daher verwenden wir in dieser Arbeit das *indirekte Beweisverfahren* nicht für die Überprüfung der EXAM-Testabläufe.

Das Ergebnis der Überprüfung mit Hilfe des *direkten Beweisverfahrens* gibt an, welche Testabläufe die spezifizierten Bedingungen erfüllen bzw. verletzen. Somit können im *vierten Schritt*

die entsprechenden Testabläufe durch den Anwender korrigiert und bei Bedarf die Überprüfung beginnend mit Schritt *zwei* wiederholt durchgeführt werden.

Nach der vorgestellten Übersicht der einzelnen Prozessschritte betrachten wir die Transformation der EXAM-Elemente in eine Petri-Netz-Darstellung im folgenden Abschnitt im Detail. Im Anschluss stellen wir dar, welche spezielle Arten von Abhängigkeiten zwischen den Operationen der Funktionsbibliotheken in EXAM bestehen, wie diese mit der vorgestellten Methode überprüft werden können und welche Erweiterung der Verifikationsalgorithmen und der Netzdarstellung der AL-Module dazu notwendig sind.

6.3 Transformation der EXAM-Elemente in ein Petri-Netz

Wie im vorherigen Abschnitt dargestellt müssen für die Überprüfung der EXAM-Elemente mit Hilfe der Aktionslogik die EXAM-Elemente in ein Modul der Aktionslogik in Form einer korrespondierenden Petri-Netz-Darstellung transformiert werden. In den folgenden Abschnitten beschreiben wir die Transformation für die EXAM-Elemente *TestCase*, *TestSequence*, *TestActivity* und *TestSuite*.

6.3.1 Transformation des EXAM-Elements TestSequence

Die *TestSequence* wird in EXAM dazu verwendet einen sequentiellen Testablauf zu beschreiben. Die grafische Modellierung erfolgt dabei mit Hilfe eines Sequenzdiagramms (vgl. Kapitel 2.4.3).

Die Sequenzdiagramme enthalten eine Reihe von Operationsaufrufen (*TestSteps*) und beliebig viele Interaktionsfragmente *alt*, *while*, *for*, *return* und *break* zur Steuerung des Kontrollflusses. In der korrespondierenden Petri-Netz-Darstellung werden die einzelnen *TestSteps* als Transitionen mit dem Namen des Operationsaufrufs dargestellt. Die Interaktionsfragmente werden durch die Struktur der Petri-Netze dargestellt. Im Folgenden geben wir die Transformation für die verschiedenen Interaktionsfragmente an.

Sequenzdiagramm ohne Kontrollstruktur

In der einfachsten Form enthält ein Sequenzdiagramm, wie auf der linken Seite in Abbildung 6.2 dargestellt, nur Operationsaufrufe ohne Kontrollstrukturen oder Verweise auf andere EXAM-Elemente (*UmlUseCases*). Die drei Operationsaufrufe *AktionA*, *AktionB* und *AktionC* werden sequenziell in der dargestellten Reihenfolge hintereinander aufgerufen. Somit enthält das korrespondierende Petri-Netz (Abbildung 6.2, rechts) fünf Transitionen und nur eine Schaltfolge

$$(s, AktionA, AktionB, AktionC, g)$$

welche die leere Markierung reproduziert.

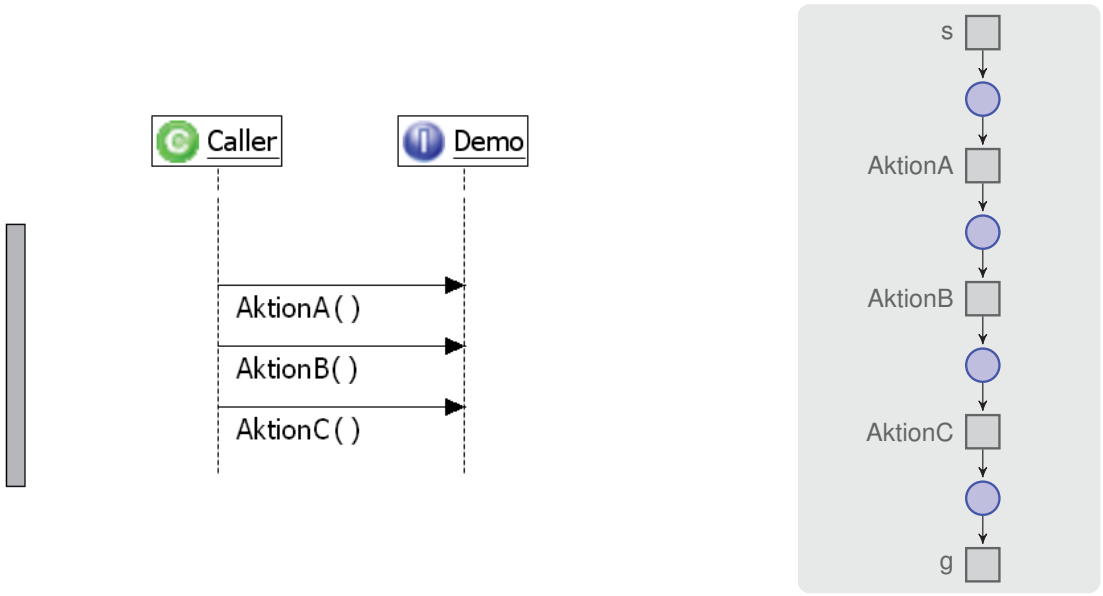

Abbildung 6.2: Sequenz ohne Verzweigung

Sequenzdiagramm mit Alternative

Das Sequenzdiagramm in Abbildung 6.3 enthält ein Interaktionsfragment vom Typ *alt*. Das Fragment ist vergleichbar mit einem *IF-Statement* in einer Programmiersprache. Somit ist in diesem Beispiel die Ausführung des Operationsaufrufs *AktionB* abhängig davon, ob die Bedingung *Bedingung1* des Fragments zu *wahr* oder *falsch* ausgewertet wird.

Im Allgemeinen sind für die Auswertung der Bedingungen in EXAM zwei Alternativen zu unterscheiden:

1. Die Bedingung kann erst zur Laufzeit des Testablaufs am Prüfstand ausgewertet werden.

2. Die Bedingung kann bereits vor der Ausführung des Testablaufs am Prüfstand (*offline*) ausgewertet werden.

Im Folgenden betrachten wir beide Möglichkeiten und geben für beide ein Beispiel. Für die erste Möglichkeit ergibt sich, dass die Auswertung der Bedingung vor der Ausführung des Testablaufes am Prüfstand nicht möglich ist, wenn die zur Auswertung der Bedingung benötigten Werte vom aktuellen Zustand des Prüfplatzes zum Zeitpunkt der Testausführung abhängig sind. Da somit zum Zeitpunkt der Überprüfung nicht entschieden werden kann, ob die Operationen innerhalb der Alternative berücksichtigt werden müssen, gehen wir davon aus, dass die Bedingung einmal zu *wahr* und einmal zu *falsch* ausgewertet werden kann. Kann die Bedingung nicht vor der Ausführung des Testablaufs am Prüfstand eindeutig ausgewertet werden, wird zur Darstellung der Alternative in der Petri-Netz-Darstellung zusätzlich die Operation *Nop* (No Operation) eingefügt.

Für die Transformation des Sequenzdiagramms in Abbildung 6.3 müssen beide Ausführungspfade der Alternative betrachtet werden. Aus dieser Überlegung ergeben sich für das dargestellte Petri-Netz die beiden Schaltfolgen

1. (*s, AktionA, AktionB, AktionC, g*) und
2. (*s, AktionA, Nop, AktionC, g*),

welche die leere Markierung reproduzieren.

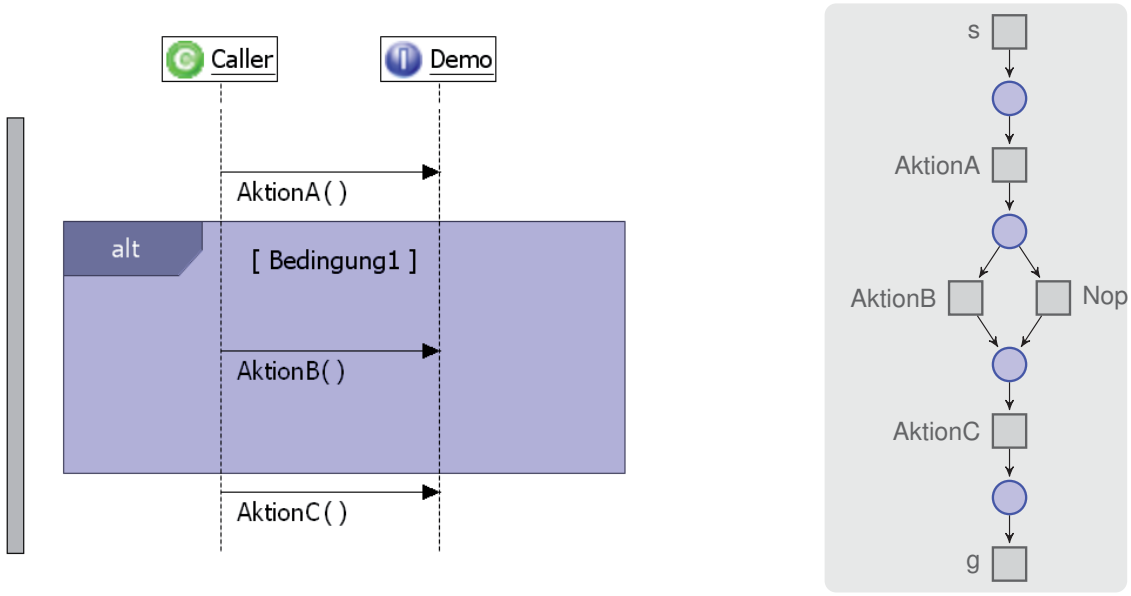

Abbildung 6.3: Sequenz mit If-Fragment

Im zweiten Beispiel (siehe Abbildung 6.5) ist die Bedingung der Alternative von der Parametrierung des Testfalls abhängig und kann somit vor der Ausführung des Testablaufs am Prüfstand (*offline*) ausgewertet werden. In diesem Fall wird nur der entsprechende in der Abbildung 6.5 dargestellte Ausführungspfad in das Petri-Netz übertragen.

Die zugehörige Parametrierung des Testablaufs ist in der Abbildung 6.4 dargestellt. Der Wert des Parameters *ParameterWert1* ist gleich Null. Dieser Wert wird in dem Testablauf mit Hilfe der Operation *get_ParameterWert*1(*parameterWert*) aus dem Parametersatz ausgelesen und in die lokale Variable *parameterWert* gespeichert. Auf Basis dieser Variablen wird in diesem Beispiel die Bedingung der Alternative zu *falsch* ausgewertet ($parameterWert = 0 \neq 1$), sodass die Operationen innerhalb des Interaktionsfragments nicht ausgeführt werden. Daraus ergibt sich die auf der rechten Seite der Abbildung 6.5 dargestellte Netzdarstellung mit nur einer Schaltfolge, welche die leere Markierung reproduziert. Das Netz enthält keine Transition für die *AktionB*.

Abbildung 6.4: Parametrierung für den Testfall in Abbildung 6.5

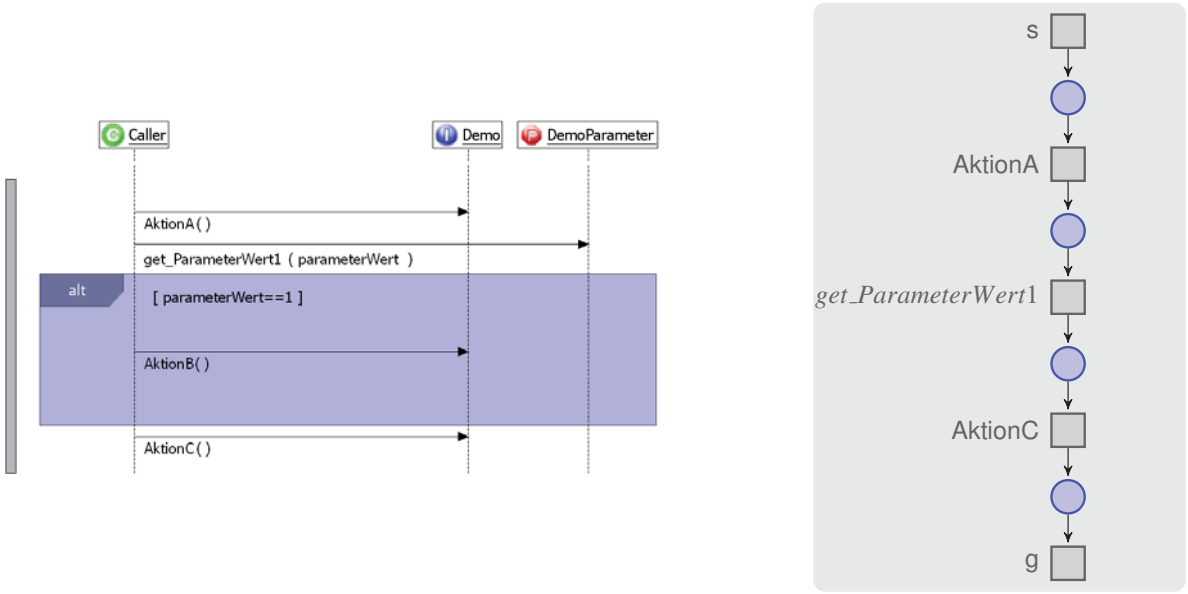

Abbildung 6.5: Sequenz mit If-Fragment und offline auswertbarer Bedingung

Sequenzdiagramm mit kombinierter Alternative

Neben der im vorherigen Sequenzdiagramm dargestellten einfachen Alternative besteht in EXAM die Möglichkeit, zwei oder mehrere Alternativen zu kombinieren. Das dazu verwendete Interaktionsfragment ist vergleichbar mit einem *IF-ELSE-Statement* in einer Programmiersprache. Ist die Bedingung der ersten Alternative erfüllt, werden alle Operationen innerhalb des Fragments ausgeführt und im Anschluss wird der Testablauf nach dem Interaktionsfragments vorgesetzt. Ist die Bedingung der ersten Alternative nicht erfüllt, wird die Bedingung der zweiten Alternative überprüft. Ist diese erfüllt, werden die Operationen der zweiten Alternative ausgeführt. Wenn diese ebenfalls nicht erfüllt ist, wird keine Operation des kombinierten Fragments ausgeführt. Für mögliche weitere Alternativen gilt das gleiche Vorgehen.

Für die Betrachtung der Transformation in eine entsprechende Netzdarstellung ist als Beispiel in der Abbildung 6.6 ein Sequenzdiagramm mit zwei kombinierten Alternativen dargestellt. Wie bei der einfachen Alternative können die Bedingungen erst zur Laufzeit bestimmt werden, wenn diese vom Zustand des Prüfstands abhängig sind. Ist die Bedingung auch *offline* auswertbar, wird entsprechend nur der jeweilige mögliche Ausführungspfad in die Netzdarstellung transformiert.

In diesem Beispiel wird unabhängig von der Auswertbarkeit der Bedingung *Bedingung1* und *Bedingung2* in jedem Fall die Operation *AktionA* zuerst ausgeführt. Abhängig von den beiden Bedingungen *Bedingung1* und *Bedingung2* wird entweder als nächste Operation *AktionB* oder *AktionC* oder gar keine weitere Operation ausgeführt. Dieses Verhalten wird in Analogie zur

einfachen Alternative in der Netzdarstellung wieder mit Hilfe einer Verzweigung an einer Stelle dargestellt. Das entsprechende Netz enthält drei Schaltfolgen (*s*,*AktionA*, *AktionB*, *g*), (*s*, *AktionA*, *AktionC*, *g*) und (*s*, *AktionA*, *Nop*, *g*) welche die leere Markierung reproduzieren. Hierbei wird die Transition *Nop* verwendet um den Fall zu modellieren, dass sowohl die Bedingung *Bedingung1* als auch *Bedingung2* nicht zu wahr ausgewertet wird.

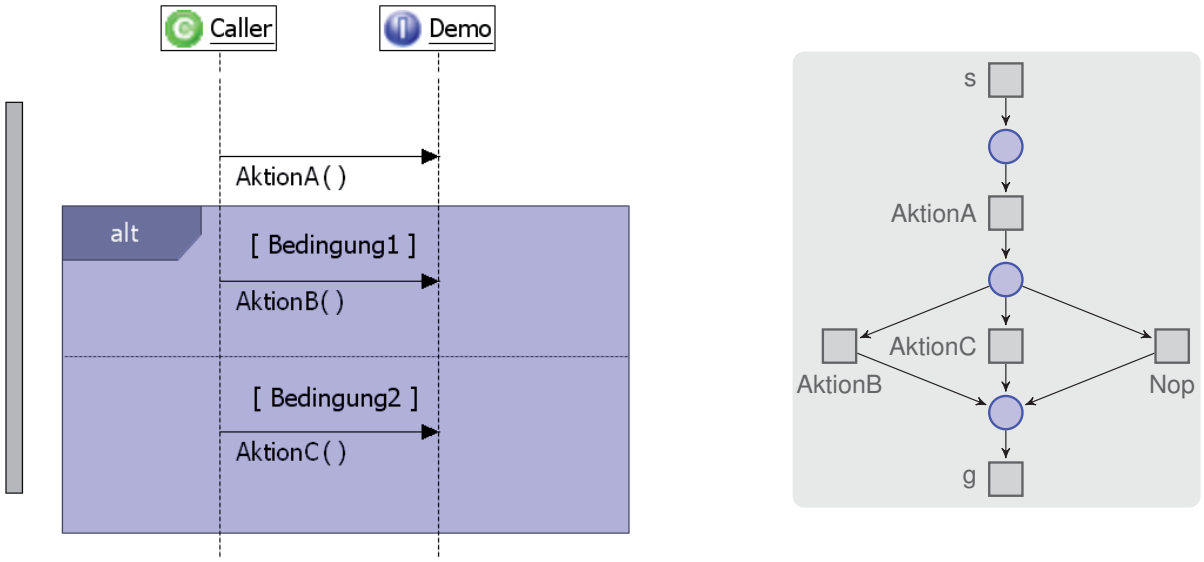

Abbildung 6.6: Sequenz mit If-Else Fragment

Dieses zusammengesetzte Fragment ist nicht auf zwei Alternativen beschränkt. Die Transformation von mehreren kombinierten Alternativen erfolgt analog zu dem beschriebenen Beispiel.

Sequenzdiagramm mit Schleifen

Wie bereits beschrieben werden in EXAM zwei verschiedene Arten von Schleifen (*while* und *for*) unterschieden. Das Fragment *while* ermöglicht eine Abfolge von Operationen mehrfach auszuführen, solange die Schleifenbedingung erfüllt ist. Die *For-Schleife* erlaubt ebenso eine Abfolge von Operationen wiederholt auszuführen, jedoch wird hierbei die Anzahl der Iterationen durch die Angabe eines initialen Wertes, einer Abbruchbedingung und einer Schrittweite in Anlehnung an die Syntax und Semantik einer *For-Schleife* in der Programmiersprache Python vor der Ausführung der Schleife vorgegeben.

Für die Transformation der Schleifenfragmente in eine Petri-Netz-Darstellung und für die anschließende Überprüfung der kausalen Abhängigkeiten der Operation ist die Anzahl der Schleifeniterationen von Bedeutung. Die Abbruchbedingung einer Schleife kann wie bei den Alternativen in der Regel erst zur Laufzeit des Testfalls bestimmt werden. Somit ist die Bestimmung der Anzahl der Iterationen vor der Ausführung des Testfalls größtenteils nicht möglich. Für die Überprüfung ist es jedoch notwendig, jeden möglichen Ausführungspfad zu betrachten. Daher ist für diesen Fall eine besondere Betrachtung der Schleifenelemente notwendig. Ist jedoch die Bestimmung der Anzahl der Iterationen vor der Ausführung des Testablaufs am Prüfstand möglich, wird

die entsprechende Anzahl an Iterationen der Operationen in der Netzdarstellung durch eine wiederholte Ausführung des Schleifenrumpf dargestellt. Diese Darstellung ermöglicht im weiteren Verlauf der Arbeit eine einfachere Berechnung der Schaltfolgen.

Im Gegensatz zu der vorgeschlagenen Modellierung von Schleifen in der Aktionslogik verwenden wir an dieser Stelle eine auf die Verifikation der EXAM-Modelle angepasste Darstellung der Schleifen in den Petri-Netzen. Im Prinzip ist für die Überprüfung der kausalen Abhängigkeiten nur entscheidend, wie oft die Schleife durchlaufen wird. Hierbei sind prinzipiell drei Möglichkeiten zu unterscheiden. Im ersten Fall ist die Schleifenbedingung schon vor dem Schleifendurchlauf nicht erfüllt und somit werden die Operationen im Rumpf der Schleife nie ausgeführt. Im zweiten Fall ist die Schleifenbedingung nach der Ausführung einer ungeraden Anzahl an Wiederholungen nicht mehr erfüllt und im dritten Fall ist die Bedingung nach der Ausführung einer geraden Anzahl an Wiederholungen nicht mehr erfüllt. Hierbei können wir im Hinblick auf die Verifikation die ungerade Anzahl der Wiederholungen mit einer Ausführung approximieren und die gerade Anzahl der Wiederholungen mit der Ausführung von zwei Schleifeniterationen. Wie wir im weiteren Verlauf der Arbeit zeigen werden, ist diese Vereinfachung für die Überprüfung der Testabläufe in EXAM ausreichend.

Basierend auf dieser Überlegung ergibt sich für das Sequenzdiagramm in Abbildung 6.7, dass die Operation *AktionB* innerhalb der Schleife entweder gar nicht, einmal (ungerade Anzahl) oder zweimal (gerade Anzahl) ausgeführt wird. Die entsprechende Petri-Netz-Darstellung enthält drei Schaltfolgen $(s, AktionA, AktionB_1, AktionC, g)$, $(s, AktionA, AktionB_2, AktionB_3, AktionC, g)$ und $(s, AktionA, Nop, AktionC, g)$, wobei der Index der Transition *AktionB* dazu verwendet wird, die einzelnen Transitionen im Netz zu unterscheiden. Die Transition *Nop* wird dazu verwendet, um den Ausführungspfad zu beschreiben, in dem die Schleifenbedingung schon vor der ersten Ausführung ungültig ist.

Für die Transformation des Schleifenfragments vom Typ *for* gehen wir analog vor. Ein Beispiel für die Transformation eines Sequenzdiagramms mit einer *For-Schleife* ist in Abbildung 6.8 dargestellt. Hierbei ist die Anzahl der Schleifendurchläufe vor der Ausführung am Prüfstand eindeutig bestimmbar, daher wird die Aktion zweimal hintereinander ausgeführt.

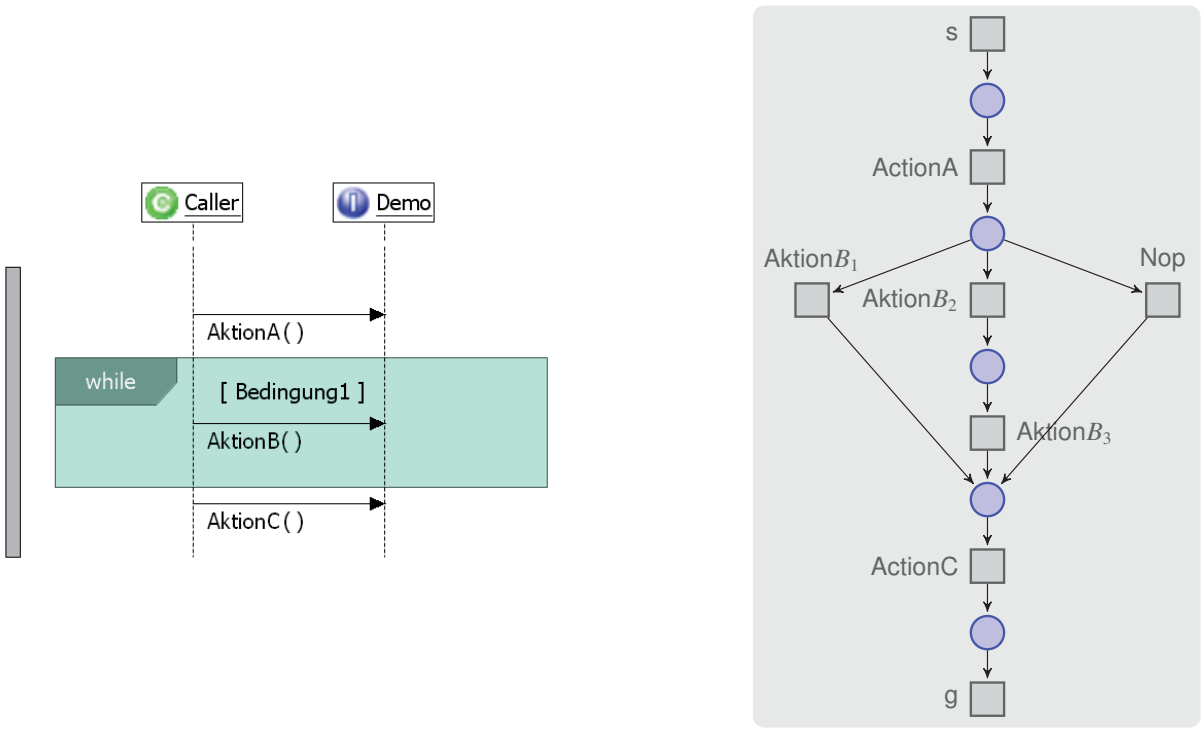

Abbildung 6.7: Sequenz mit While-Fragment

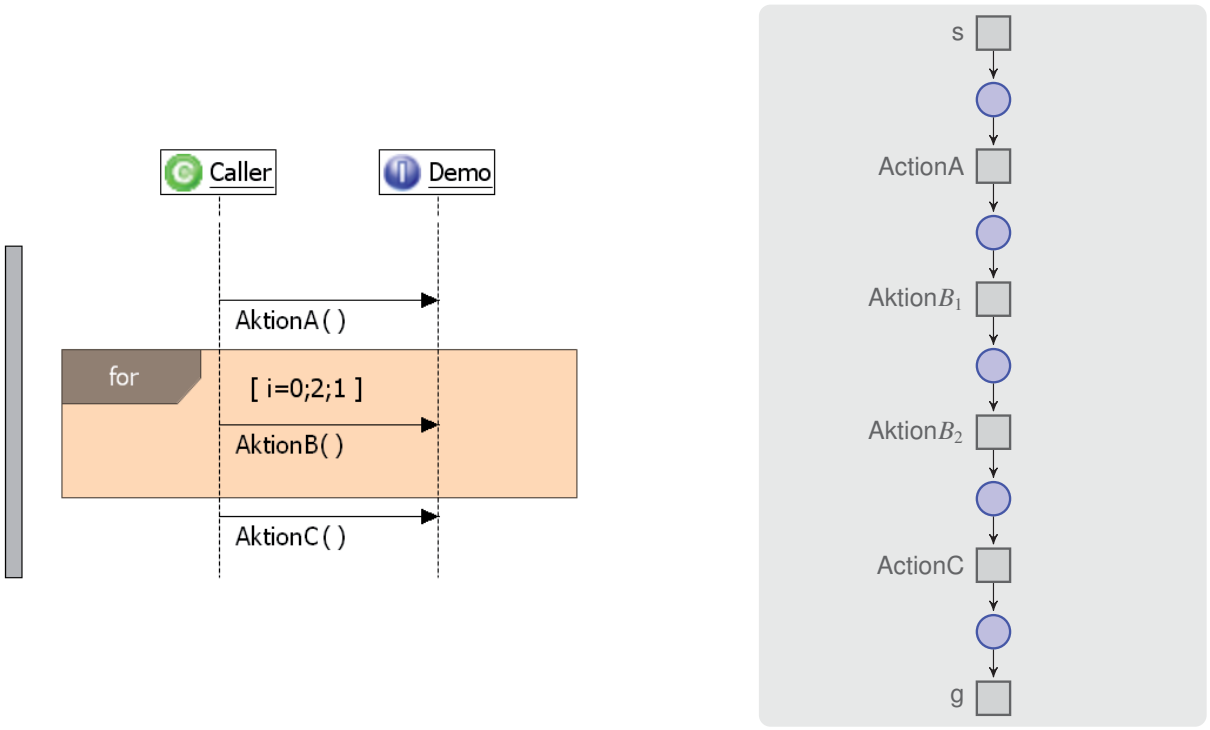

Abbildung 6.8: Sequenz mit For-Fragment

Sequenzdiagramm mit Break-Fragment

Das Fragment vom Typ *break* erlaubt die Interaktionsfragmente vom Typ *alt*, *for* und *while* vorzeitig zu verlassen. Bevor das entsprechende Fragment verlassen und der Testablauf nach dem Fragment fortgesetzt wird, werden alle Operationen innerhalb des *break*-Fragments ausgeführt. Voraussetzung ist natürlich, dass die Bedingung des *break*-Fragments erfüllt ist.

Wir zeigen die Transformation des *break*-Fragments an dem Beispiel des in Abbildung 6.9 dargestellten Sequenzdiagramms mit einem *alt*- und *break*-Fragment. Nach dem Aufruf der Operation *AktionA* wird die Bedingung *Bedingung1* der Alternative überprüft. Wird die Bedingung zu falsch ausgewertet, endet die Testsequenz. Ist die Bedingung wahr, so wird die Operation *AktionB* aufgerufen und im Anschluss überprüft, ob die Bedingung *Bedingung2 wahr* oder *falsch* ist. Ist diese *falsch*, wird *AktionC* aufgerufen. Wenn die Bedingung wahr ist, wird die Operation *AktionD* aufgerufen und danach die Alternative verlassen. In diesen Fall wird die *AktionC* nicht mehr ausgeführt. Das entsprechende Petri-Netz enthält wie in Abbildung 6.9 dargestellt somit drei Schaltfolgen, die die leere Markierung reproduzieren. Wir gehen bei diesem Beispiel davon aus, dass die Bedingungen erst zur Laufzeit des Testablaufs ausgewertet werden können. Wäre eine Auswertung bereits *offline* möglich, könnte die Netzdarstellung entsprechend um die nicht ausführbaren Pfade reduziert werden.

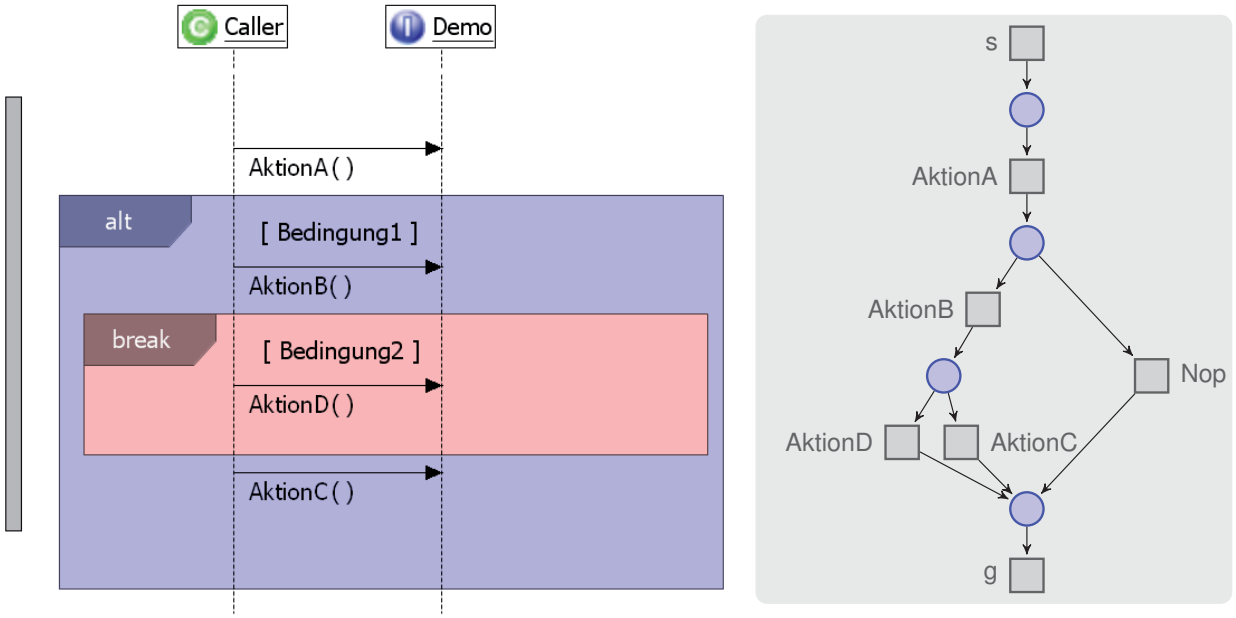

Abbildung 6.9: Sequenz mit If- und Break-Fragment

Sequenzdiagramm mit Return-Fragment

Das *return*-Fragment dient dazu, die aktuelle Testsequenz vorzeitig zu verlassen. Ist die Bedingung des *return*-Fragments erfüllt, wird nach der Ausführung der in dem *return*-Fragment enthaltenen Operationen das aktuelle Sequenzdiagramm sofort verlassen.

Wir zeigen die Transformation des *return*-Fragments an dem Beispiel des in Abbildung 6.10 dargestellten Sequenzdiagramms. Nach dem Aufruf der Operation *AktionA* wird die Bedingung *Bedingung1* des *return*-Fragments überprüft.

Wird die Bedingung zu falsch ausgewertet, so wird die Ausführung des Testablaufs nach dem *return*-Fragment mit dem Aufruf der Operationen *AktionB* und *AktionC* fortgesetzt. Ist die Bedingung wahr, so wird die Operation *AktionD* innerhalb des *return*-Fragments aufgerufen und im Anschluss das Sequenzdiagramm verlassen. Somit werden die Operationen *AktionB* und *AktionC* nicht mehr ausgeführt. Das entsprechende Petri-Netz enthält, wie in Abbildung 6.10 dargestellt, die beiden Schaltfolgen $(s, AktionA, AktionB, AktionC, g)$ und $(s, AktionA, AktionD, g)$, welche die leere Markierung reproduzieren.

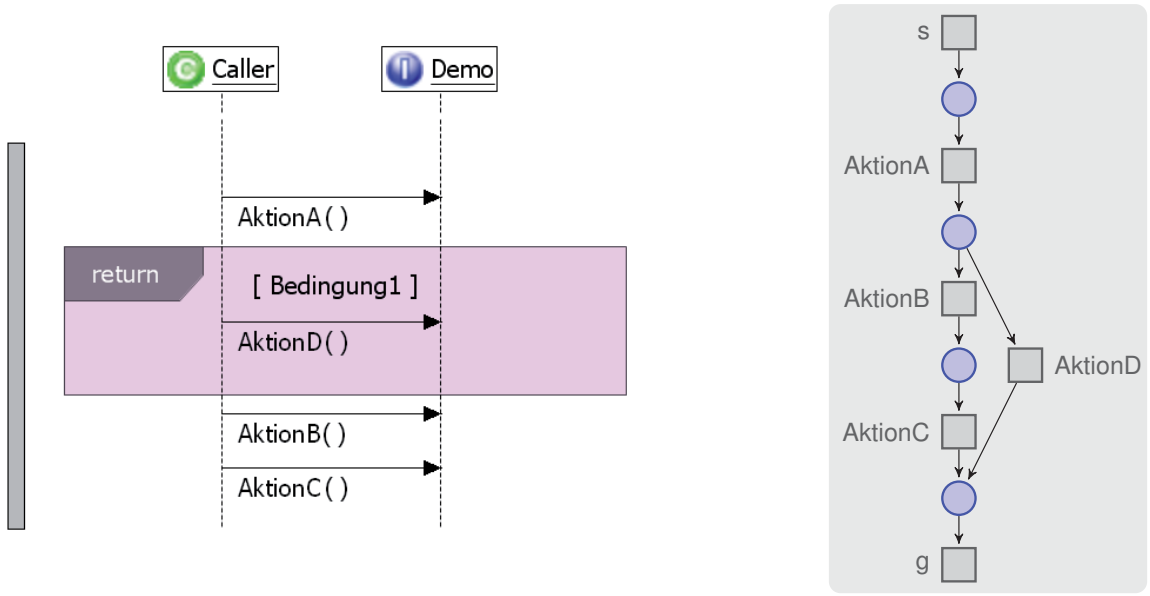

Abbildung 6.10: Sequenz mit Return-Fragment

Sequenzdiagramm mit Ref-Fragment

Das Fragment vom Typ *ref* erlaubt innerhalb eines Sequenzdiagramms eine *TestSequence* oder eine *TestActivity* zu referenzieren (*UseCaseCall*). Während der Testausführung werden anstelle des referenzierten Fragments alle Operationen des referenzierten *TestFlows* ausgeführt. Im Anschluss wird die Ausführung nach dem *UseCaseCall* fortgesetzt. Dieses Interaktionsfragment ist vergleichbar mit einem Methodenaufruf in einer Programmiersprache, jedoch ohne Übergabeparameter und ohne Rückgabewerte.

Das durch den *UseCaseCall* referenzierte EXAM-Element wird in eine separate Netzdarstellung transformiert. Die Netzdarstellung (*Subnetz*) wird eindeutig mit dem Namen des referenzierten EXAM-Elements bezeichnet. In der Netzdarstellung des referenzierenden EXAM-Elements wird diese Referenz durch eine Transition mit dem Namen des Subnetzes dargestellt. Vor der Ausführung der Überprüfung der kausalen Abhängigkeiten wird diese Transition durch das Subnetz wieder ersetzt.

Dieses Vorgehen erlaubt die beiden EXAM-Elemente unabhängig voneinander in eine korrespondierende Netzdarstellung zu transformieren und erst zum Zeitpunkt der Verifikation das zusammengesetzte Netz zu erzeugen. Somit wird die Wiederverwendung der Subnetze in verschiedenen Testabläufen ermöglicht.

In EXAM wird die überwiegende Anzahl der *TestSequences* und *TestActivities* von mehreren Testfällen referenziert. Durch die separate Betrachtung der einzelnen Elemente muss dasselbe EXAM-Element nicht mehrfach in eine Netzdarstellung transformiert werden, nur weil es von unterschiedlichen EXAM-Elementen referenziert wird. Dies ermöglicht auch im Bezug auf die Verifikation von mehreren Testfällen oder Testgruppen (*TestSuites*) eine schnellere Transformation der EXAM-Elemente in die zugehörige Petri-Netz-Darstellung. Wie wir im weiterem Verlauf der Arbeit zeigen werden, ist dieses Vorgehen wichtig für die Beherrschbarkeit der Komplexität und Voraussetzung für den praktischen Einsatz der Methode.

Als Beispiel für eine Transformation eines Sequenzdiagramms mit einer Referenz auf ein weiteres EXAM-Element betrachten wir die beiden Sequenzdiagramme in Abbildung 6.11 und 6.12.

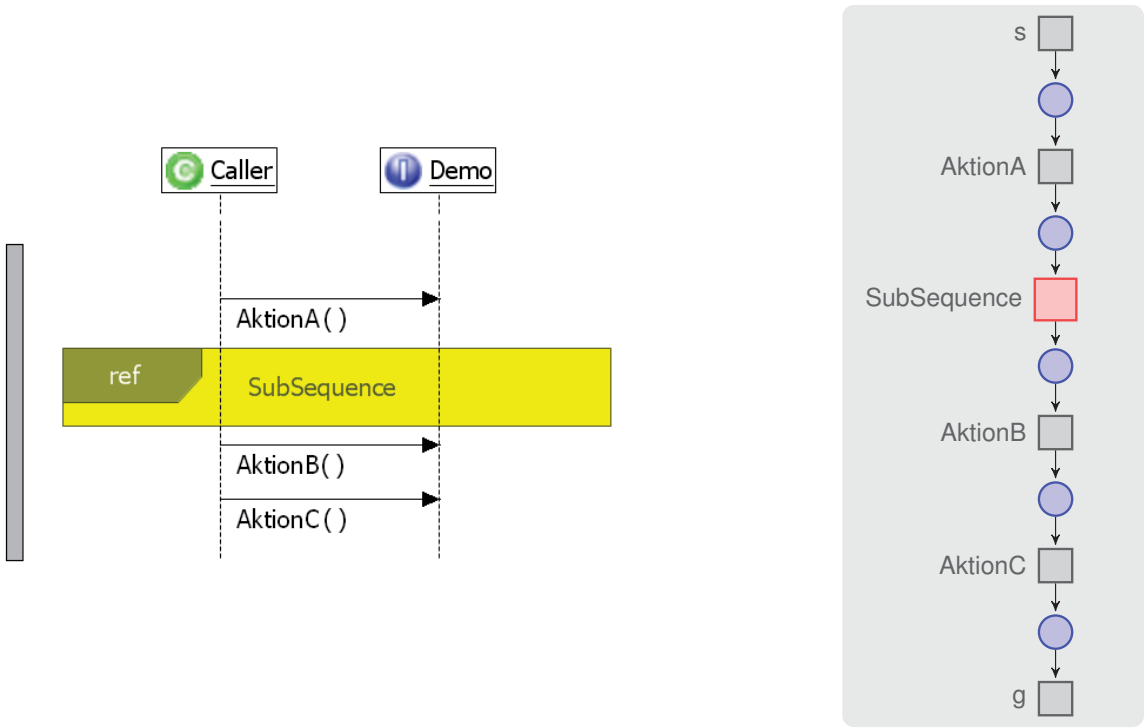

Abbildung 6.11: Sequenz mit Ref-Fragment

Das Sequenzdiagramm in Abbildung 6.11 enthält eine Referenz auf die *TestSequence SubSequence*, deren Sequenzdiagramm in Abbildung 6.12 dargestellt ist. Während der Testausführung werden die Operationen in der Reihenfolge *AktionA*, *AktionD*, *AktionE*, *AktionB*, *AktionC* aufgerufen. Ist die Bedingung *Bedingung1* der Alternative in Abbildung 6.12 nicht erfüllt, werden die Operationen *AktionA*, *AktionD*, *AktionB*, *AktionC* ausgeführt. Hierbei sind die Operationen *AktionA*, *AktionB*, *AktionC* in dem Sequenzdiagramm in Abbildung 6.11 und die Operationen *AktionD* und *AktionE* in der *TestSequence SubSequence* in Abbildung 6.12 enthalten.

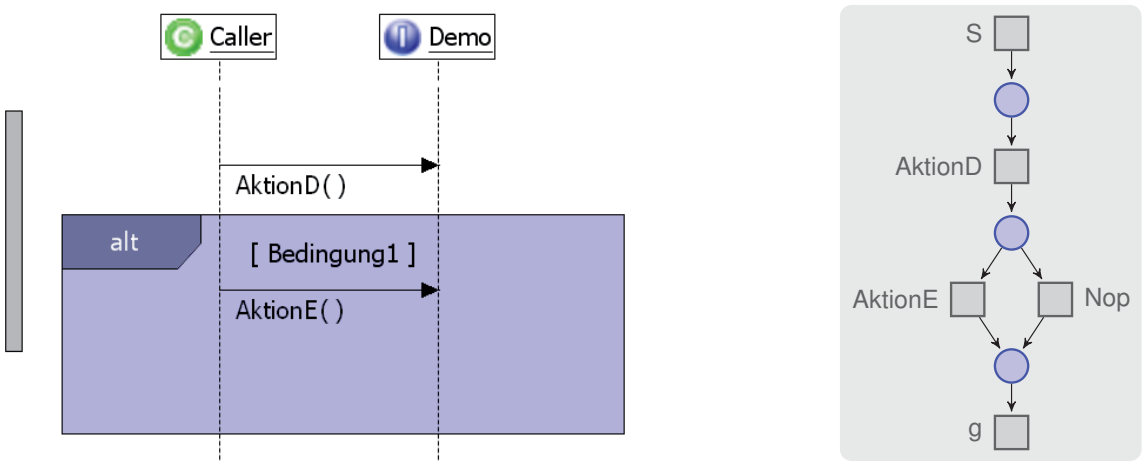

Abbildung 6.12: TestSequence *SubSequence*

Die Netzdarstellung des ersten Sequenzdiagramms (siehe Abbildung 6.11) enthält neben den Transitionen *AktionA*, *AktionB*, *AktionC* die Transition *SubSequence*, welche die Referenz auf die Netzdarstellung der referenzierten *TestSequence* darstellt. Die Netzdarstellung der referenzierten Sequenz ist in Abbildung 6.12 dargestellt. Aus diesen beiden Netzdarstellungen ergibt sich durch Einfügen der referenzierten Netzdarstellung für die Transition *SubSequence* die Netzdarstellung in Abbildung 6.13. Die eingefügten Stellen und Transitionen sind zur Verdeutlichung in der Abbildung grün markiert. Hier ist zu beachten, dass die Start- und Zieltransition (s, g) des Subnetzes nicht in die neue Netzdarstellung übernommen wurden. Die kombinierte Netzdarstellung enthält die beiden Schaltfolgen

- s, *AktionA*, *AktionD*, *AktionE*, *AktionB*, *AktionC*, g und
- s, *AktionA*, *AktionD*, *Nop*, *AktionB*, *AktionC*, g,

welche die leere Markierung reproduzieren und bildet somit genau den Ablauf der Testausführung ab.

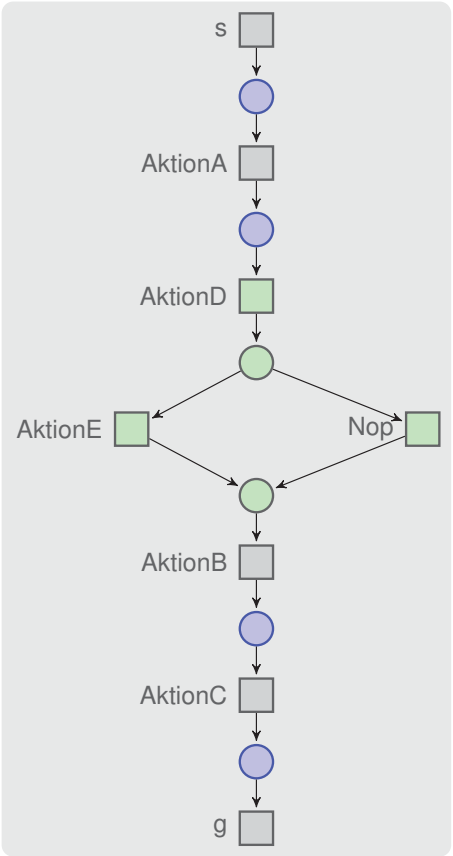

Abbildung 6.13: Sequenz mit inkludierter TestSequence *SubSequence*

6.3.2 Transformation des EXAM-Elements TestActivity

Die *TestActivity* wird in EXAM dazu verwendet, einen sequentiellen bzw. parallelen Testablauf zu beschreiben. Die grafische Modellierung erfolgt dabei mit Hilfe eines Aktivitätsdiagramms. Da in den Aktivitätsdiagrammen der UML2 neben den Notationen der Datenflussdiagramme und der Nassi-Shneidermann-Diagramme einige wesentliche Elemente der Petri-Netze aufgegriffen wurden (z. B. das Token-Konzept von Petri-Netzen - vergleiche z. B. [RHQ+05]) ist die Transformation der in EXAM verwendeten Aktivitätsdiagramme in eine korrespondierende Petri-Netz-Darstellung einfach möglich.

Die Aktionen der Aktivitätsdiagramme in EXAM werden als *TestAction* bezeichnet und in dem korrespondierenden Petri-Netz als Transitionen mit dem Namen der *TestAction* dargestellt. Die verschiedenen Kontrollknoten zur Modellierung von Verzweigungen, parallelen Abläufen, Referenzen auf *TestSequence* oder *TestActivity* und die Verwendung von mehreren Endknoten werden im Folgenden einzeln beschrieben.

Lineare Sequenz im Aktivitätsdiagramm

Wie mit Sequenzdiagrammen kann auch mit Hilfe der Aktivitätsdiagramme in EXAM eine lineare Folge von Operationsaufrufen modelliert werden. Das entsprechende Petri-Netz enthält alle *TestActions* in der gleichen Reihenfolge wie im Aktivitätsdiagramm angegeben, wobei der Startknoten mit Hilfe der Transition *s* und der Zielknoten mit Hilfe der Transition *g* modelliert wird.

Ein Beispiel gibt die Transformation des Aktivitätsdiagramms in Abbildung 6.14.

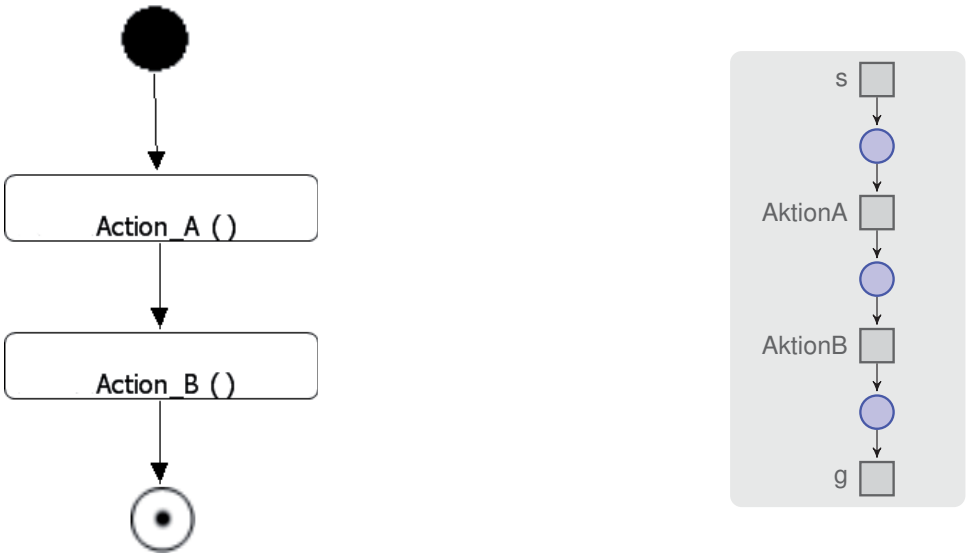

Abbildung 6.14: Lineare Sequenz im Aktivitätsdiagramm

Aktivitätsdiagramm mit Verzweigung

Mit Hilfe der *Verzweigungs-* und *Verbindungsknoten* (auch als *DecisionNode* und *MergeNode* bezeichnet) kann eine Alternative in den Aktivitätsdiagrammen modelliert werden. Die Auswertung der entsprechenden Bedingungen an den ausgehenden Kanten des Verzweigungsknotens sind, wie bei dem *alt*-Fragment der Sequenzdiagramme, in vielen Fällen erst zur Laufzeit des Testfalls möglich. Daher transformieren wir, wenn die Bedingung nicht *offline* auswertbar ist, unabhängig von den Bedingungen jeden möglichen Ausführungspfad in die Petri-Netz-Darstellung.

Für das Aktivitätsdiagramm mit einem *Verzweigungs-* und einem *Verbindungsknoten* in Abbildung 6.15 ergibt sich das dargestellte Petri-Netz mit den beiden Schaltfolgen $(s, AktionA, g)$ und $(s, AktionB, g)$, welche die leere Markierung reproduzieren.

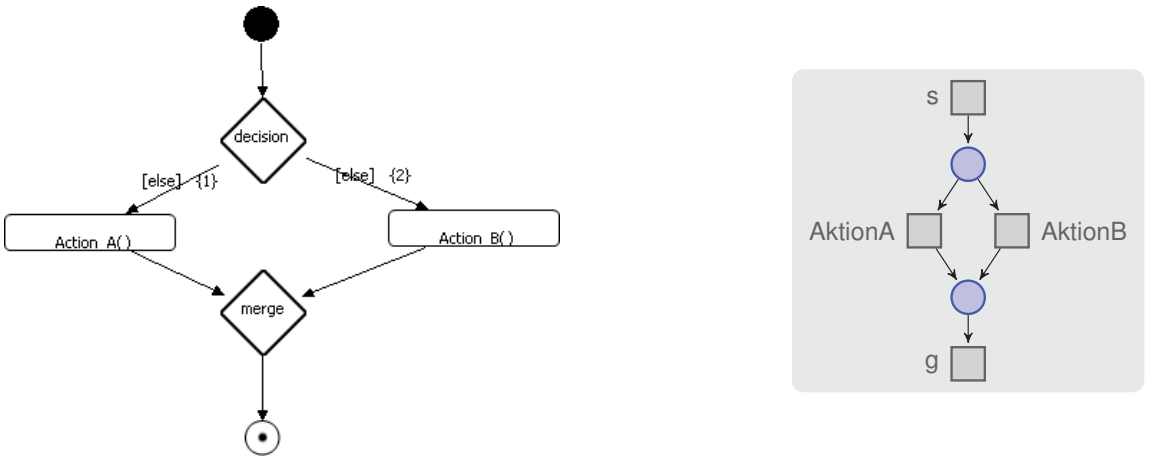

Abbildung 6.15: Verzweigung im Aktivitätsdiagramm

Aktivitätsdiagramm mit parallelen Abläufen

Mit den Aktivitätsdiagrammen ist auch die Modellierung von parallelen Abläufen möglich. Die beiden Kontrollknoten *Synchronisations-* und *Parallelisierungsknoten* erlauben beliebig viele parallele Abläufe zu starten und am Ende wieder zu synchronisieren. Im korrespondierenden Petri-Netz wird die Synchronisation und Parallelisierung mit Hilfe einer Verzweigung bzw. Synchronisation an einer Transition modelliert.

Für das in der Abbildung 6.16 dargestellte Aktivitätsdiagramm mit einem *Synchronisations-* und einem *Parallelisierungsknoten* ergibt sich somit das dargestellte Petri-Netz mit den drei Schaltfolgen $(s, AktionA, AktionB, g)$, $(s, AktionB, AktionA, g)$ und der Schaltfolge in der die Transitionen *AktionA* und *AktionB* gleichzeitig ausgeführt wird, welche die leere Markierung reproduzieren.

Aktivitätsdiagramm mit Referenzaktion

Zur Referenzierung einer *TestSequence* oder einer *TestActivity* wird in den Aktivitätsdiagrammen eine Kontrollaktion vom Typ *ref* (*SubActivity*) vergleichbar zum Interaktionsfragment vom Typ *ref* bei Sequenzdiagrammen verwendet. Während der Testausführung werden anstelle der

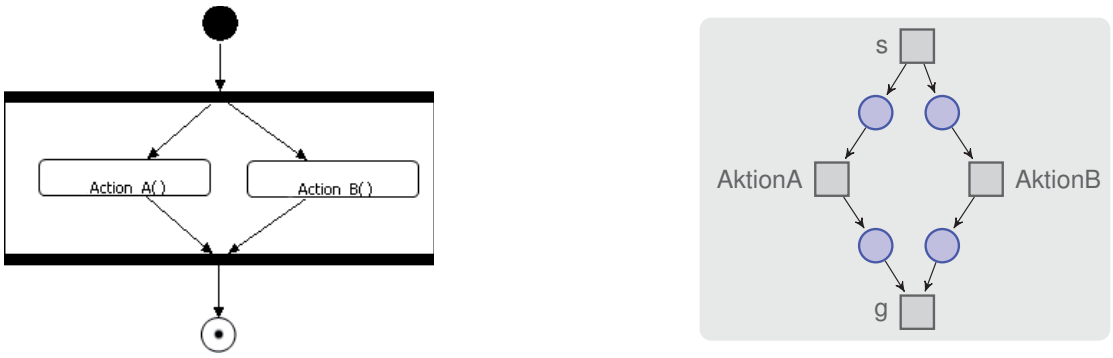

Abbildung 6.16: Parallele Aktionen im Aktivitätsdiagramm

Kontrollaktion alle Aktionen oder *TestSteps* des referenzierten EXAM-Elements (*UmlUseCase*) ausgeführt.

Wir zeigen die Transformation in eine korrespondierende Petri-Netz-Darstellung am Beispiel des in Abbildung 6.17 dargestellten Aktivitätsdiagramms, welches eine *TestActivity* referenziert. Das Vorgehen für die Referenzierung einer *TestSequence* ergibt sich entsprechend.

In Abbildung 6.17 ist ein Aktivitätsdiagramm abgebildet, welches die *TestActivity SubActivity1* referenziert (Abbildung 6.18). In der entsprechenden Netzdarstellung ist die Referenzierung durch ein Transition mit dem Namen der referenzierten *TestActivity* dargestellt (rot hervorgehoben). Wie bei der Transformation von Sequenzdiagrammen werden beide Aktivitätsdiagramme unabhängig voneinander in die Netzdarstellung überführt und erst zum Zeitpunkt der Verifikation zu einem Netz verbunden. Das zusammengesetzte Netz des gesamten Testablaufes ist in Abbildung 6.19 dargestellt.

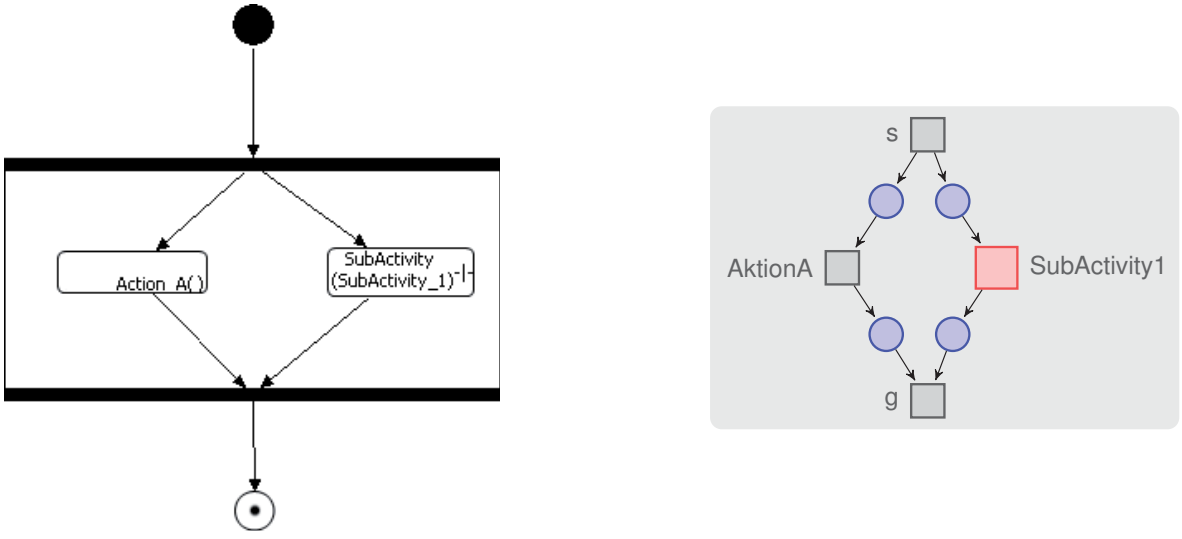

Abbildung 6.17: Parallele Aktionen im Aktivitätsdiagramm mit Referenz auf eine TestActivity

Bei dem Einfügen der Netzdarstellung *SubActivity1* werden die Start- und Zieltransitionen nicht berücksichtigt. Alle anderen Transitionen und Stellen von *SubActivity1* sind in dem vereinigten Netz grün hervorgehoben. Die aggregierte Netzdarstellung stellt genau den durch die beiden

Abbildung 6.18: TestActivity *SubActivity1*

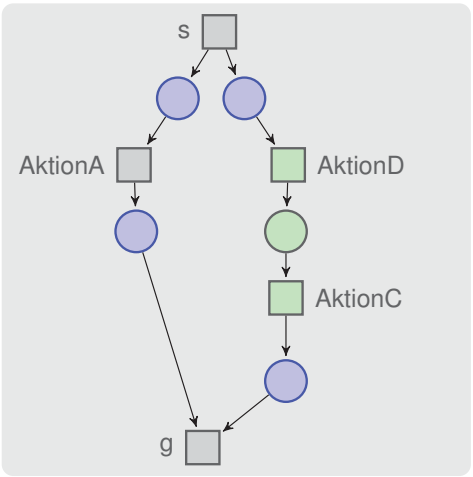

Abbildung 6.19: Parallele Aktionen im Aktivitätsdiagramm mit inkludierter TestActivity

Aktivitätsdiagramme modellierten Testablauf dar. Hierbei kann die Transition *AktionA* entweder gleichzeitig zu den Transitionen *AktionC* und *AktionD* ausgeführt werden oder jeweils vor oder nach den Transition.

Aktivitätsdiagramm mit mehreren Zielknoten

Die Aktivitätsdiagramme in EXAM dürfen nur einen *Startknoten* aber beliebig viele *Zielknoten* enthalten. Wird während der Testausführung ein *Zielknoten* erreicht, wird die Testausführung dieses Aktivitätsdiagramms direkt beendet. Dementsprechend muss das korrespondierende Petri-Netz so modelliert werden, dass beim Erreichen eines Zielknotens im Aktivitätsdiagramm die Zieltransition *g* ausgeführt wird und alle Stellen in dem Netz keine Token mehr enthalten.

Wir betrachten die Transformation am Beispiel des Aktivitätsdiagramms in Abbildung 6.20. Ist die *Bedingung1* erfüllt, wird nach der Ausführung der Operation *AktionA* ein Zielknoten erreicht und somit die Ausführung des Aktivitätsdiagramms beendet. In der Petri-Netz-Darstellung ist

die Nachstelle der Transition *AktionA* direkte Vorstelle der Zieltransition *g*. Die beiden anderen Pfade der Verzweigung werden wie bereits zuvor beschrieben in die Netzdarstellung überführt.

Durch die Verzweigung an der Nachstelle der Transition *s* kann sich jeweils nur ein Token auf einem der drei Pfade in dem Netz befinden. Somit wird sobald die Transition *AktionA* feuert die Transition *g* aktiviert. Nach dem Schalten von *g* befindet sich somit kein Tokens mehr in dem Netz.

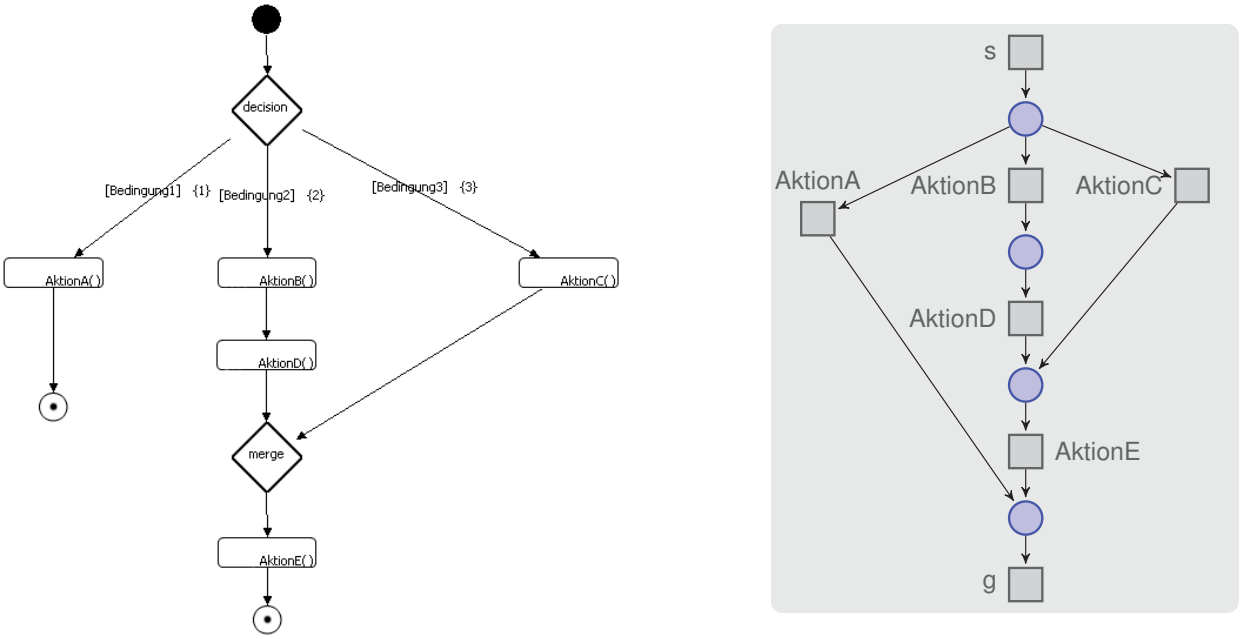

Abbildung 6.20: TestActivity mit mehreren Zielknoten

6.3.3 Transformation des EXAM-Elements TestCase

Nach der Betrachtung der Transformation der EXAM-Elemente *TestSequence* und *TestActivity* betrachten wir nun die Transformation des EXAM-Elements *TestCase* in eine korrespondierende Petri-Netz-Darstellung.

Das EXAM-Element *TestCase* wird in EXAM dazu verwendet einen konkreten, in sich abgeschlossenen Testfall zu beschreiben. Dabei wird der eigentliche Testablauf entweder mit Hilfe eines Sequenz- oder Aktivitätsdiagramms beschrieben. Innerhalb dieser Diagramme können beliebig viele *TestSequences* oder *TestActivities* mit Hilfe eines *UseCaseCalls* referenziert werden.

In Abhängigkeit davon, ob der *TestCase* ein Sequenz- oder Aktivitätsdiagramm für die Modellierung des Testablaufes verwendet, erfolgt die Transformation in die korrespondierende Netzdarstellung basierend auf den Beschreibungen für die Transformation einer *TestSequence* oder *TestActivity* wie in den Abschnitten 6.3.1 bzw. 6.3.2 beschrieben. Die Bezeichnung der Netzdarstellung entspricht dem Namen des *TestCases*.

6.3.4 Transformation des EXAM-Elements TestSuite

Eine *TestSuite* erlaubt es, mehrere Testfälle zu einem bestimmten Themengebiet zu gruppieren. Die Gruppierung der *TestCases* erfolgt dabei durch *TestGroups*.

Jede *TestSuite* in EXAM enthält standardmäßig die drei *TestGroups*

- *InititializeGroup*,
- *ExecuteGroup* und
- *DeInitializeGroup*.

Die *InititializeGroup* dient der Gruppierung der notwendigen EXAM-Elemente zur Initialisierung der Testabläufe. Die *ExecuteGroup* dient zur Gruppierung der eigentlichen Testfälle und die *DeInitializeGroup* zur Gruppierung aller notwendigen Operationen, die nach der Ausführung der Testfälle ausgeführt werden müssen. Für die grafische Modellierung der *TestGroups* werden Sequenzdiagramme eingesetzt, die jedoch keine Interaktionsfragmente enthalten.

Für die Transformation einer *TestSuite* in eine korrespondierende Petri-Netz-Darstellung wird ein Petri-Netz mit dem Namen der *TestSuite* erzeugt, welches neben der Start- und Zieltransition die drei Transitionen *InititializeGroup*, *ExecuteGroup*, *DeInitializeGroup* als Referenz auf die jeweilige Netzdarstellung der drei Elemente enthält. Jede *TestGroup* wird dann separat, wie eine *TestSequence* (vergleiche Abschnitt 6.3.1) in eine entsprechende Netzdarstellung transformiert. Vor der Ausführung der Verifikation werden die entsprechenden Referenzen aufgelöst und alle referenzierten Netzdarstellungen zu einer Netzdarstellung der *TestSuite* integriert.

Als ein Beispiel für eine Transformation einer *TestSuite* in die korrespondierende Petri-Netz-Darstellung betrachten wir die in der Abbildung 6.21 als Baumstruktur dargestellte *TestSuite TestSuiteExample*. Die Kinder eines Elements in der Baumstruktur geben an, dass dieses Element zu dem entsprechenden Elternelement gehört. Neben *OperationCalls* und *UseCaseCalls* sind in der Baumstruktur auch die EXAM-Elemente *TestSuite*, *TestCase* sowie die *TestGroups* dargestellt. In diesem Beispiel enthält die Suite die drei *TestGroups initialization*, *execute* und *deInitialization*, die in dieser Reihenfolge aufgerufen werden. Innerhalb der *TestGroup initialization* wird der *TestCase InitSequence* aufgerufen, der die Operation *Operation1* enthält. Innerhalb der *TestGroup execute* werden zwei weitere *TestGroups*, *testGroup1* und *testGroup2* aufgerufen, die jeweils die Testfälle *TestCase1* und *TestCase2* aufrufen. Die letzte *TestGroup deInitialization* ruft den Testfall *DeInitSequence* auf.

Wie zuvor beschrieben erzeugen wir nun für alle *TestGroups* und alle *TestCases*, die innerhalb der Gruppen der *TestSuite* aufgerufen werden, eine eigene Netzdarstellung und bezeichnen das jeweilige Petri-Netz mit dem Namen des entsprechenden EXAM-Elements. Für dieses Beispiel ergeben sich somit fünf Netzdarstellungen für die enthaltenen *TestGroups* (*initialization, execute, testGroup1, testGroup2, deInitialization*) und vier Netzdarstellungen für die Testfälle (*InitSequence, TestCase1, TestCase2, DeInitSequence*) und natürlich die Netzdarstellung der *TestSuite* mit den Transitionen *s, initialization, execute, deInitialization und g*. Aufgrund der einfachen Struktur der Netzdarstellungen, sind diese an dieser Stelle nicht dargestellt.

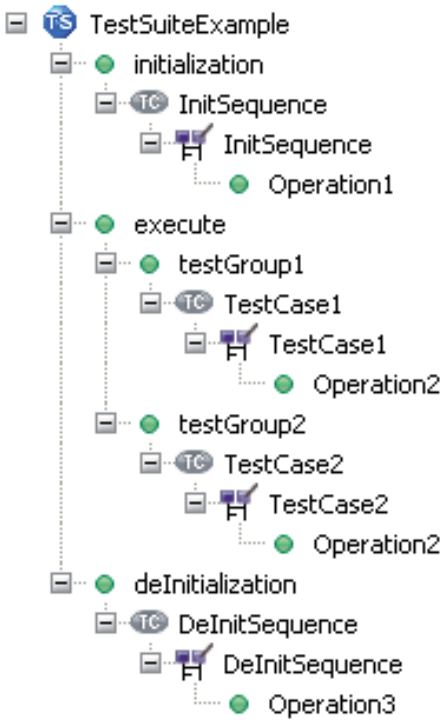

Abbildung 6.21: Baumdarstellung der *TestSuite TestSuiteExample*

6.3.5 Umgang mit Systemkonfigurationen

Bei der bisherigen Transformation der EXAM-Elemente in eine korrespondierende Netzdarstellung haben wir die im Abschnitt 2.4.2 beschriebenen Systemkonfigurationen nicht beachtet. Eine Systemkonfiguration ordnet jeweils einer Interfaceklasse eine Implementierungsklasse zu. Für die Überprüfung der kausalen Abhängigkeiten sind die Implementierungsklassen dann von Interesse, wenn die Operationen der Implementierungsklassen mit Hilfe von Sequenz- oder Aktivitätsdiagrammen modelliert sind. Grund hierfür ist, dass die Sequenz- bzw. Aktivitätsdiagramme wiederum Operationen der Funktionsbibliothek enthalten können, die Auswirkungen auf den Testablauf und die kausale Ordnung der Operationen haben können und somit bei der Überprüfung der kausalen Abhängigkeiten berücksichtigt werden müssen.

In der Abbildung 6.22 ist ein einfacher Testfall dargestellt, welcher zwei Operation des Interface *Fahrzeug* und zwei Operationen des Interface *Datalogger* verwendet. Für das Interface *Fahrzeug* existieren in unserem Beispiel im EXAM-Modell zwei Implementierungsklassen *AudiA4* und *AudiA8*, die in der Abbildung 6.23 dargestellt sind. Für das Interface *Datalogger* existiert nur eine Implementierungsklasse, in der alle Operationen direkt durch Pythoncode realisiert sind und somit keine Auswirkungen auf die Überprüfung der kausalen Abhängigkeiten hat.

Die Implementierung der Operationen der Klasse *AudiA4* erfolgt in Python und übernimmt direkt die Ansteuerung des Prüfstandes. Die Operationen der Klasse *AudiA8* sind mit Hilfe von Sequenzdiagrammen modelliert, die wiederum bestehende Operationen der Funktionsbibliothek verwenden. Die Operation *Beschleunigen* (Abbildung 6.24, oben) ruft die beiden Operationen *initDataLogger(1)* und *Beschleunigen_auf_Wunschgeschwindigkeit*(*Geschwindigkeit*, 1) auf. Für

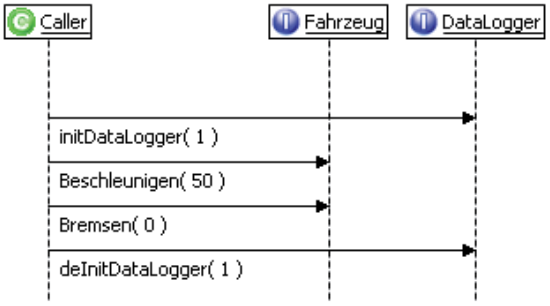

Abbildung 6.22: Beispiel eines Testfalls mit zwei Operationsaufrufen

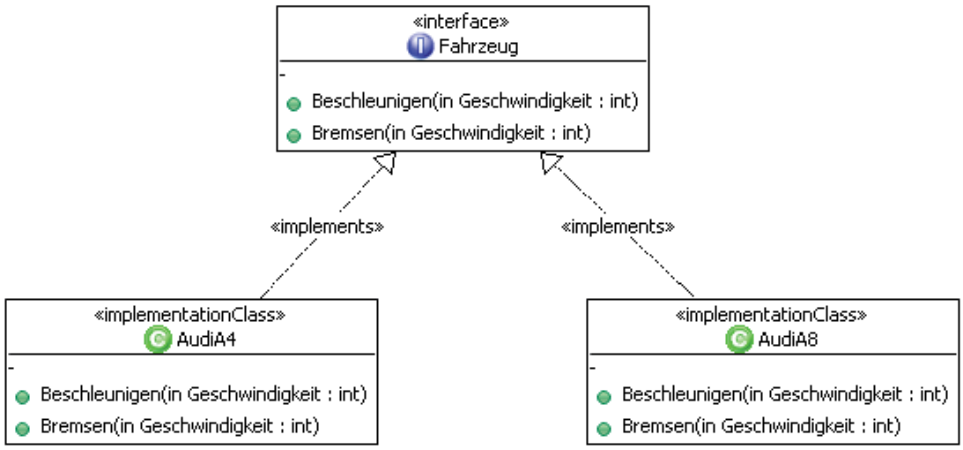

Abbildung 6.23: Zwei Implementierungsklassen des Interfaces Fahrzeug

die Implementierung der Operation *Bremsen* (Abbildung 6.24, unten) werden die beiden Operationen *Abbremsen_auf_Wunschgeschwindigkeit*(*Geschwindigkeit*, 1) und *deInitDataLogger(1)* verwendet.

In Abhängigkeit der Systemkonfiguration unterscheiden sich nun die beiden Testabläufe. Für eine Systemkonfiguration *SysKonf1*, in der der Interfaceklasse *Fahrzeug* die Implementierungsklasse *AudiA4* zugeordnet ist, ergeben sich die im linken Bereich der Abbildung 6.25 dargestellten Operationsaufrufe. In einer zweiten Systemkonfiguration *SysKonf2* ist der Interfaceklasse *Fahrzeug* die Implementierungsklasse *AudiA8* zugeordnet. Während der Testausführung werden, wie im rechten Bereich der Abbildung 6.25 dargestellt, zusätzliche Operationen aufgerufen.

Der Testablauf in diesem Beispiel soll nun mit einer Spezifikation

$$S_1 = (initDataLogger \circledcirc deInitDataLogger)$$

überprüft werden. Die Spezifikation hat als weitere Besonderheit, dass die angegebenen Aktionen nur in einer *eins-zu-eins-Beziehung* vorkommen dürfen. Das heißt, nach jeder Ausführung der Operation *initDataLogger* muss die Operation *deInitDataLogger* ausgeführt werden, bevor wieder *initDataLogger* ausgeführt werden darf. Wird der Testfall nun mit der Systemkonfiguration

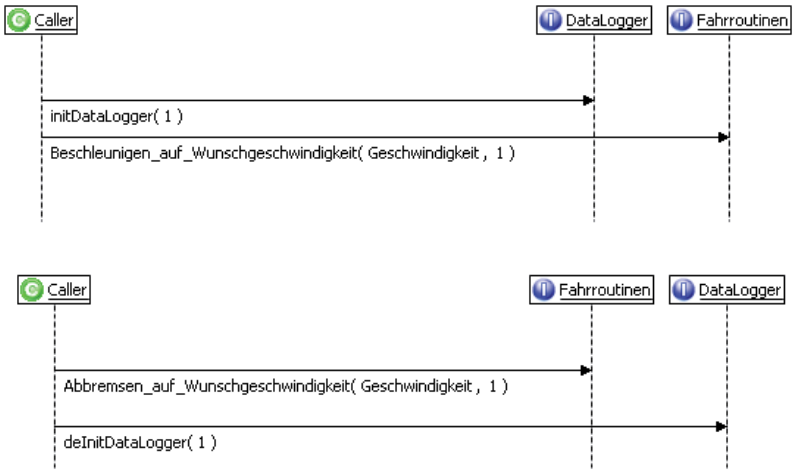

Abbildung 6.24: Modellierung der beiden Operationen der Implementierungsklasse *AudiA8* mit Hilfe von Sequenzdiagrammen

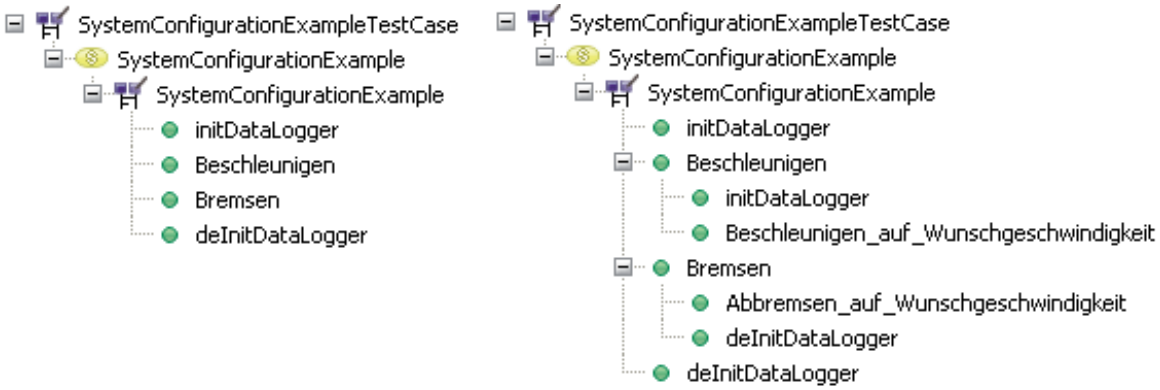

Abbildung 6.25: Testablauf mit der Implementierungsklasse *AudiA4* (links) und *AudiA8* (rechts)

SysKonf1 (AudiA4) ausgeführt, ist die Bedingung erfüllt. Bei einer Ausführung in der Konfiguration *SysKonf2* (AudiA8) wir die Operation *initDataLogger* zweimal in Folge ausgeführt, welches nicht der Spezifikation entspricht. Daher ist die Transformation der EXAM-Elemente nicht unabhängig von den Systemkonfigurationen zu betrachten. Vor jeder Transformation der EXAM-Elemente in eine korrespondierende Petri-Netz-Darstellung muss die für die Ausführung geplante Systemkonfiguration angegeben werden, damit auch die in den Operationen der Funktionsbibliothek enthaltenen Sequenz- bzw. Aktivitätsdiagramme in eine korrespondierende Netzdarstellung transformiert werden.

Im realen Testbetrieb wird vor der Ausführung eines *TestCases* am Prüfstand eine Systemkonfiguration ausgewählt. Soll der Testfall auf unterschiedlichen Prüfständen mit verschiedenen Systemkonfigurationen ausgeführt werden, ist eine Überprüfung des Testablaufes, und somit auch eine Transformation in eine Petri-Netz-Darstellung, für jede verwendete Systemkonfigurationen erforderlich.

Nach der Beschreibung der Transformation der einzelnen EXAM-Elemente in eine korrespondierende Petri-Netz-Darstellung und den Besonderheiten im Umgang mit Systemkonfigurationen betrachten wir im Folgenden die für die Definition der kausalen Abhängigkeiten in EXAM notwendigen verschiedenen Arten von Spezifikationen und die zugehörigen Algorithmen für die Überprüfung der Abhängigkeiten im Testablauf.

6.4 Verschiedene Arten von Spezifikationen

Die verschiedenartigen Beziehungen zwischen den EXAM-Operationen machen es notwendig, verschiedene Typen von Spezifikationen zu unterscheiden. Der Typ der Spezifikation wird durch zwei Attribute (*Voraussetzungsattribut* und *Häufigkeitsattribut*) eindeutig definiert. Das *Voraussetzungsattribut* gibt an, unter welchen Randbedingungen die Spezifikation für die Verifikation der Realisierungen eingesetzt wird. Das *Häufigkeitsattribut* bestimmt, wie oft die spezifizierten kausalen Abhängigkeiten der Aktionen in der Realisierung vorkommen dürfen bzw. müssen. Die beiden Attribute erlauben es zu definieren, unter welchen Bedingungen die Spezifikation überprüft werden muss und welche Besonderheiten im Bezug auf das Vorkommen der kausalen Abhängigkeiten in den Testabläufen gelten.

Eine Spezifikation ist erst dann eindeutig bestimmt, wenn ihr genau ein *Voraussetzungsattribut* und genau ein *Häufigkeitsattribut* zugeordnet ist. Somit darf eine Spezifikation nicht mehrere *Voraussetzungsattribute* bzw. *Häufigkeitsattribute* enthalten, sondern jeweils nur genau eines. Eine Übersicht aller Spezifikationstypen gibt Abbildung 6.26.

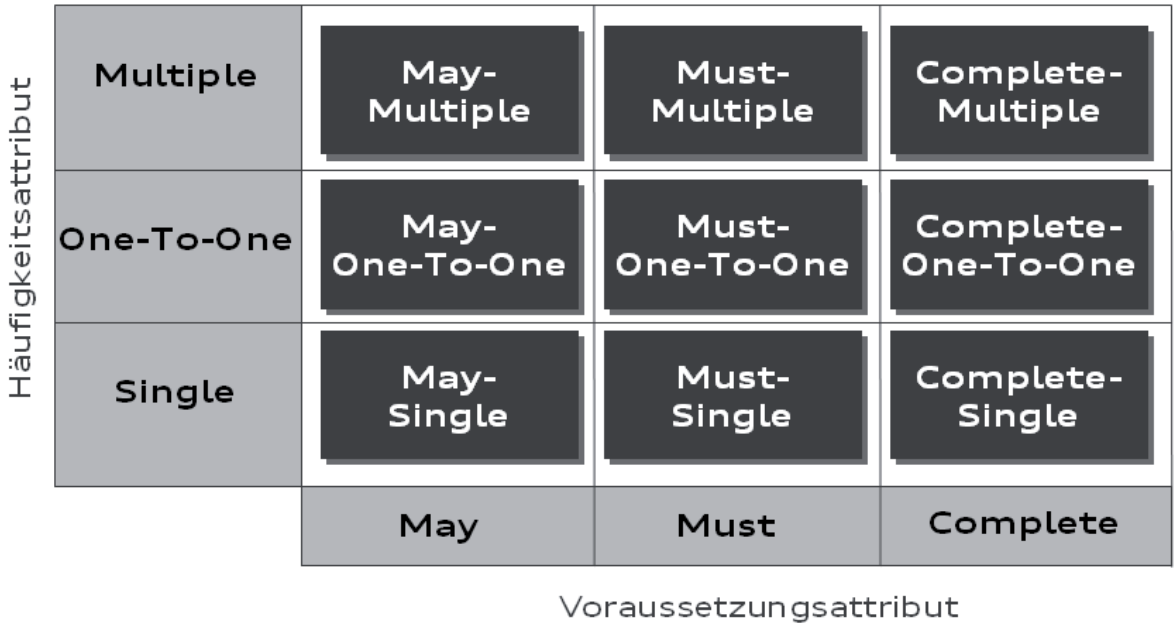

Abbildung 6.26: Arten von Spezifikationen

Im folgenden Abschnitt beschreiben wir die verschiedenen Typen der Spezifikationen sowie die sich daraus ergebenen Besonderheiten bei der Überprüfung der kausalen Abhängigkeiten. An-

hand einiger Beispiele wird die Anwendung der Spezifikationen und der zugehörigen Algorithmen zur Überprüfung der kausalen Abhängigkeiten im Detail verdeutlicht. Dazu beschreiben wir zuerst die *Häufigkeitsattribute* und im Anschluss die *Voraussetzungsattribute* sowie die sich aus den beiden Attributen ergebenen neuen verschiedenen Typen von Spezifikationen.

6.4.1 Häufigkeitsattribute: Single, One-to-One und Multiple

Bisher haben wir nur Testabläufe in EXAM betrachtet, in denen eine Operation (Aktion) nur einmal im Testablauf enthalten ist. Bei den Testabläufen in EXAM gibt es jedoch eine Vielzahl an Operationen, die innerhalb einer Testdurchführung mehrfach ausgeführt werden. So kann zum Beispiel nach dem Starten des Motors (*startMotor()*) das Fahrzeug mehrfach beschleunigt und wieder abgebremst werden oder nach der Initialisierung des Datenloggers kann die Messung mehrfach gestartet und wieder gestoppt werden, bevor die Deinitialisierung des Datenloggers durchgeführt wird.

Ein Auszug aus einem Testfall, in dem gleiche Operationen mehrfach aufgerufen werden, ist in Abbildung 6.27 dargestellt. An diesem Beispiel wird deutlich, dass für die Überprüfung der kausalen Abhängigkeiten der EXAM-Operationen zwei verschiedene Arten der Überprüfung unterschieden werden müssen. Zum einen kann nach der Ausführung der Operation *initDataLogger()* die Operation *startRecord()* mehrfach ausgeführt werden. Zum anderen muss aber nach jeder Ausführung der Operation *initDataLogger()* die Operation *deInitDataLogger()* ausgeführt werden, bevor wieder der Datenlogger initialisiert werden kann. Diese kausalen Abhängigkeiten der Operationen ergeben sich aufgrund der im Abschnitt 3.3 auf Seite 41 definierten *domänenspezifischen* und *domänenübergreifenden Bedingungen*.

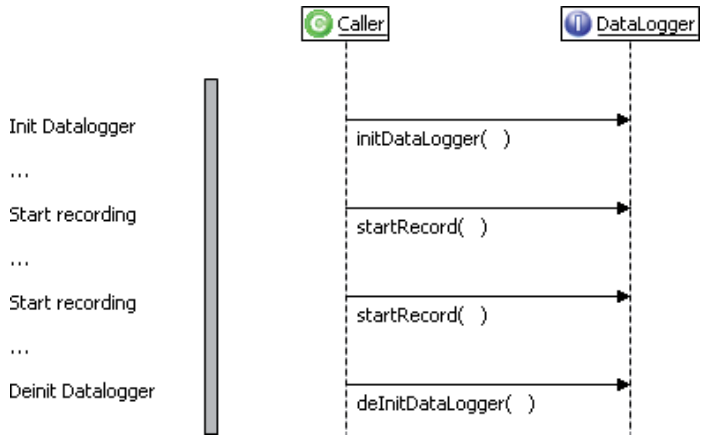

Abbildung 6.27: Auszug aus einem Testfall mit gleichen Operationen

Für diese Anwendungsfälle führen wir zur Beschreibung der Spezifikationen die beiden *Häufigkeitsattribute Multiple* und *One-To-One* ein. Die Möglichkeit, in einem Testablauf mehrere Instanzen der gleichen Operation aufzurufen, erfordert noch ein drittes *Häufigkeitsattribute Single*. Im Bereich der Modellierungsrichtlinien gibt es eine Reihe von Beispielen, in denen überprüft werden muss, ob eine bestimmte Abfolge von Operationen genau einmal im Testablauf enthalten ist. Spezifikationen die dieses Verhalten beschreiben ordnen wir das *Häufigkeitsattribut Single*

zu. Alle drei Attribute schließen sich gegenseitig aus, sodass eine Spezifikation entweder das Attribut *Multiple*, *One-To-One* oder *Single* besitzt.

Bevor wir die drei Attribute *Single*, *Multiple* und *One-To-One*, welche es erlauben die vorgestellten Arten von Bedingungen zu definieren, im Detail betrachten, definieren wir im folgenden Aktionslogik-Module, die es erlauben, mehrere Instanzen einer Aktion zu enthalten.

Nach der bisherigen Definition der Module im Abschnitt 5.1 ist es nicht möglich, verschiedene Instanzen derselben Aktion in einem Modul zu verwenden, da die Abhängigkeitsfunktion zwischen gleichen Aktionen nicht definiert ist. Wie schon bei der Definition von Schleifen in der Aktionslogik (Definition 5.14, Seite 72) wird daher jede Instanz einer Aktion durch einen eindeutigen Index erweitert, sodass die verschiedenen Instanzen eindeutig zu unterscheiden sind. Weiterhin betrachten wir jede Instanz als eigenständige Aktion und somit kann die bestehende Definition der Abhängigkeitsfunktion weiterhin verwendet werden.

Definition 6.1 (Instanzen einer Aktion)
Ist in einem Modul $M \in \mathbb{M}_{\mathbb{A}}$ die Aktion a n-mal für $n \in \mathbb{N}$ enthalten, wird diese Aktion in der Aktionsmenge $\mathbb{A}$ und in dem Modul M durch eine Kopie der Aktion a mit eindeutigem Index ersetzt. Daraus ergeben sich n-Instanzen a_i, $1 \leq i \leq n$, die eindeutig zu unterscheiden sind.
Die Instanzen werden dabei im Bezug auf die Abhängigkeitsfunktion als eigenständige (eindeutig unterscheidbare) Aktionen angesehen.

Beispiele für Module mit mehreren Instanzen sind die beiden Module

- $M_1 = [a_1 \oslash a_2 \oslash b_1 \oslash a_3 \oslash b_3]$ oder
- $M_2 = [(a \oslash b_1) \oslash b_2]$.

Das Modul $M_1 = [a_1 \oslash a_2 \oslash b_1 \oslash a_3 \oslash b_3]$ über der Aktionsmenge $\mathbb{A} = \{a_1, a_2, a_3, b_1, b_2\}$ hat die Prozessmenge $\mathbb{P}[M_1] = \{a_1 \oslash a_2 \oslash b_1 \oslash a_3 \oslash b_3\}$.

Das Modul $M_2 = [(a \oplus b_1) \oslash b_2]$ über der Aktionsmenge $\mathbb{A} = \{a, b_1, b_2\}$ besitzt die Prozessmenge $\mathbb{P}[M_2] = \{((a \oslash b_1) \oslash b_2), ((a \oslash b_1) \oslash b_2), ((a \oslash b_1) \oslash b_2)\}$.

Das Vorkommen verschiedener Instanzen derselben Operationen in den Testabläufen hat Auswirkungen auf die Überprüfung der kausalen Abhängigkeiten. Enthält ein Modul (Testablauf) mehrere Instanzen einer Aktion, so ist das vorgestellte *direkte Beweisverfahren* (vergleiche Definition 5.24 auf Seite 78), basierend auf dem Vergleich der Anzahl der Prozesse des Moduls nicht mehr ausreichend. Daher werden entsprechende Erweiterungen für alle drei *Häufigkeitsattribute* im Folgenden vorgestellt.

Single-Spezifikation

Für die Definition der kausalen Abhängigkeiten zwischen den Operationsaufrufen in EXAM, die *genau einmal* im Testablauf enthalten sein dürfen bzw. müssen, verwenden wir das *Häufigkeitsattribut Single*.

Vorgaben im Rahmen der Modellierungsrichtlinien oder der Modellverantwortlichen geben viele Beispiele für die Anwendung einer Spezifikation mit diesem Attribut. Zum Beispiel ist durch die Modellverantwortlichen im Volkswagen-Konzern definiert, dass genau einmal am Anfang eines Testablaufs der Prüfablauf initialisiert und am Ende des Prüfablaufs deinitialisiert werden soll. In der Initialisierung des Prüfablaufs wird unter anderem definiert, welche Botschaftssignale aufgezeichnet und an welcher Stelle die Messdaten abgespeichert werden. Bei der Deinitialisierung wird die Aufzeichnung der Botschaftssignale beendet, die aufgezeichneten Messdaten aus verschiedenen Quellen für die Auswertung der Testergebnisse zusammengefasst und in eine Datei oder eine Datenbank gespeichert.

Als Einschränkung für *Single-Spezifikationen* legen wir fest, dass in der Spezifikation jede Operation nur einmal vorkommen darf.

Bei der Überprüfung eines Testablaufs auf Einhaltung der in einer Spezifikation mit dem *Häufigkeitsattribut Single* definierten kausalen Abhängigkeit von Operationen sind zwei Möglichkeiten zu unterscheiden.

1. Die erste Möglichkeit ist, dass in dem Testablauf mehrere Instanzen, der in der Spezifikation enthaltenen Aktionen vorkommen.

2. Die zweite Möglichkeit ist, dass der Testablauf nur jeweils eine oder keine Instanz der Operationen der Operationsmenge der Spezifikation enthält.

Für beide Möglichkeiten unterscheidet sich die Überprüfung der kausalen Abhängigkeiten im Testablauf, wie im Algorithmus 6.1 angegeben.

Algorithmus 6.1 (Verifikation von *Single-Spezifikationen*)
Gegeben sei eine Spezifikation S über der Aktionsmenge $\mathbb{A}_S$ und eine Realisierung R über der Aktionsmenge $\mathbb{A}_R$. Dabei gilt, dass $\mathbb{A}_R \cap \mathbb{A}_S \neq \emptyset$. Weiterhin ist in S nur jeweils eine Instanz der Aktionen aus der Aktionsmenge $\mathbb{A}_S$ enthalten, wohingegen in R beliebig viele Instanzen der Aktionen aus der Aktionsmenge $\mathbb{A}_R$ vorkommen dürfen.
Alle Aktionen aus der Menge $\mathbb{A}_R \backslash \mathbb{A}_S$ dürfen beliebig häufig in der Realisierung R enthalten sein.

Für die Verifikation sind zwei Fälle zu unterscheiden:

1. *In mindestens einem Prozess der Realisierung R sind mehrere Instanzen einer Aktion aus der Aktionsmenge $\mathbb{A}_S$ enthalten. Somit kann die Bedingung der* Single-Spezifikationen *nicht erfüllt sein, da spezifizierte Aktionen mehrfach in der Realisierung vorkommen.*

2. *In der Realisierung R (pro Prozess) ist jeweils nur eine oder keine Instanz der Aktionen aus der Aktionsmenge $\mathbb{A}_S$ enthalten. Da somit keine spezifizierten Aktionen mehrfach in der Realisierung vorkommen, kann mit Hilfe des Algorithmus 5.2 auf Basis der Anzahl der Prozesse entschieden werden, ob die Realisierung die Spezifikation erfüllt.*

Multiple-Spezifikation

Bei einer Spezifikation mit dem Attribut *Multiple* können in der Realisierung die Aktionen in der durch die Spezifikation definierten kausalen Abhängigkeit beliebig häufig vorkommen. Nach der Beschreibung der Überprüfung dieser Spezifikationen fassen wir die einzelnen Schritte der Überprüfung im Algorithmus 6.2 zusammen.

Als Einschränkung für Spezifikationen mit dem Attribute *Multiple* gehen wir davon aus, dass mehrere Instanzen der Aktionen nur in der Realisierung erlaubt sind. Diese Einschränkung erlaubt es, die bestehende Verifikation auf Basis der Anzahl der Prozesse der Module zu verwenden. An dieser Stelle muss jedoch für jede der Instanzen überprüft werden, ob die spezifizierte kausale Abhängigkeit erfüllt ist. Bei einer Überprüfung werden alle anderen Instanzen der gleichen Aktion nicht berücksichtigt und somit auch nicht bei der Konjunktion der Netzdarstellungen.

Die Realisierung erfüllt die Spezifikation, wenn die in der Definition 6.2 beschriebene Bedingung erfüllt ist. Ist in der Realisierung nur jeweils eine Instanz der in der Spezifikation definierten Aktionen enthalten, kann wie bisher durch die einmalige Anwendung des Algorithmus 5.2 (Seite 86) überprüft werden, ob die Realisierung die Spezifikation erfüllt.

Definition 6.2 (Verifikation mit mehreren Instanzen)
Sind in der Realisierung mehrere Instanzen der Aktionen aus der Aktionsmenge der Spezifikation enthalten, muss mindestens einmal für jede Instanz der Aktion die spezifizierte Bedingung erfüllt sein. Dabei darf die Spezifikation nur eine Instanz der Aktionen enthalten. Sind mehrere Instanzen der Aktionen in der Realisierung enthalten, muss für die Überprüfung der spezifizierten kausalen Abhängigkeiten die Netzdarstellung der Realisierung erweitert werden.

Für die Überprüfung von Modulen mit mehreren Instanzen einer Aktion betrachten wir zwei Beispiele. Ausgangspunkt der Betrachtung ist die Spezifikation $S_1 = \{a \circledcirc b\}$. Wir überprüfen, ob die Realisierungen $R_1 = \{a_1 \circledcirc a_2 \circledcirc b_1 \circledcirc a_3 \circledcirc b_2\}$ und $R_2 = \{b_1 \circledcirc a_1 \circledcirc a_2 \circledcirc b_2 \circledcirc a_3\}$ die Spezifikation S_1 mit dem Attribute *Multiple* erfüllen. Die Aktionen a_1, a_2 und a_3 sind dabei Instanzen der Aktion a und die Aktionen b_1 und b_2 Instanzen von b. Die Netzdarstellungen der beiden Module R_1 und R_2 sind in Abbildung 6.28 dargestellt.

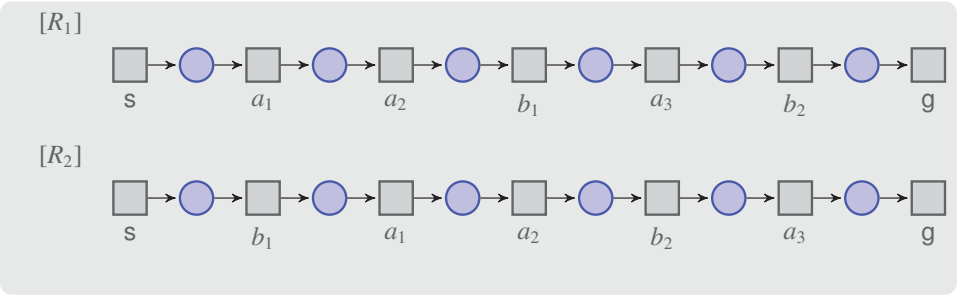

Abbildung 6.28: Netzdarstellung von R_1 und R_2 mit drei Instanzen von a und zwei von b

Betrachten wir zuerst das Modul R_1. Für die Entscheidung, ob die Realisierung R_1 die Spezifikation S_1 erfüllt, muss nach Definition 6.2 für jede Instanz der Aktionen in der Realisierung

die Spezifikation S_1 mindestens einmal erfüllt sein. Daher ist für jede Instanz der Aktionen eine eigene Überprüfung notwendig und somit auch eine Konjunktion der Netzdarstellungen $R_1 \oslash S_1$. Die entsprechenden Konjunktionen der Netzdarstellungen von R_1 und S_1 sind in Abbildung 6.29 für die Instanzen $a_1 \otimes b_1$ in $N_1[R_1 \oslash S_1]$, $a_1 \otimes b_2$ in $N_2[R_1 \oslash S_1]$ und $a_2 \otimes b_1$ in $N_3[R_1 \oslash S_1]$ exemplarisch dargestellt.

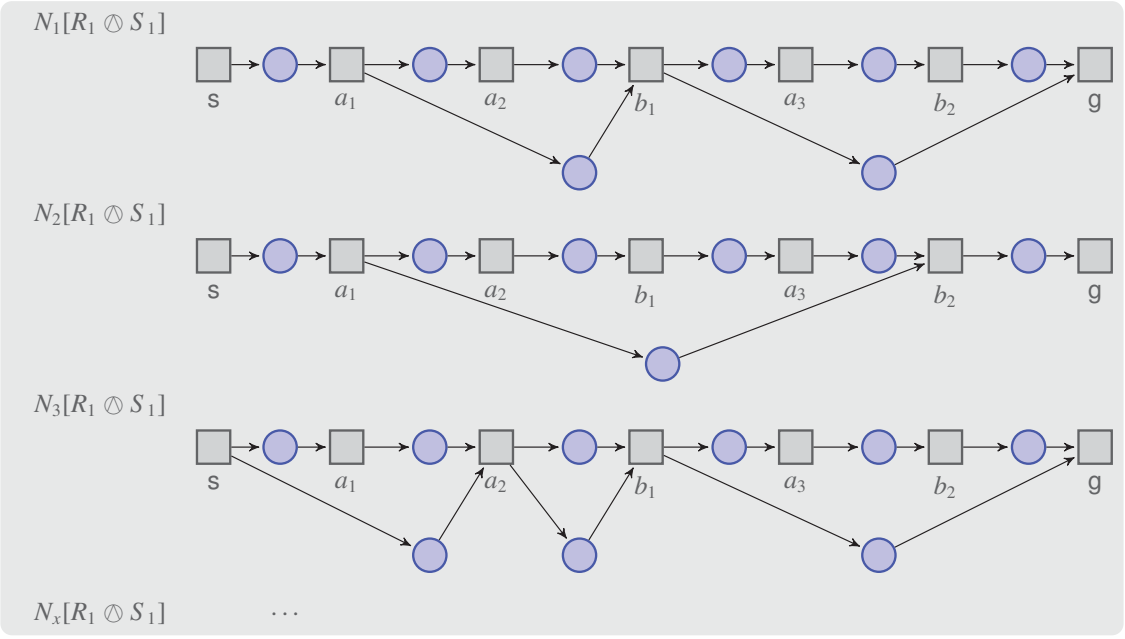

Abbildung 6.29: Beispiele für die Konjunktionen der Netzdarstellung von R_1 und S_1

An den Petri-Netz-Darstellungen $N_1[R_1 \oslash S_1]$ wird deutlich, dass alle anderen Instanzen von a und b, neben den Instanzen a_1 und b_1, in dem konkreten Fall als beliebige Aktionen betrachtet und somit bei der Konjunktion nicht direkt berücksichtigt werden. Die weiteren Netzdarstellungen der verbleibenden Instanzen a_2, a_3 und b_2 ergeben sich entsprechend. Nachfolgend ist für jede Kombination der Aktionen aufgelistet, ob die Realisierung mit dieser Kombination der Instanzen die Spezifikation erfüllt.

Für R_1 ergeben sich somit:

$$\text{Alle Kombinationen der Instanz } a_1 = \begin{cases} \{a_1 \otimes b_1\}; & \{a_1 \otimes b_1\} \twoheadrightarrow S_1 \\ \{a_1 \otimes b_2\}; & \{a_1 \otimes b_2\} \twoheadrightarrow S_1 \end{cases}$$

$$\text{Alle Kombinationen der Instanz } a_2 = \begin{cases} \{a_2 \otimes b_1\}; & \{a_2 \otimes b_1\} \twoheadrightarrow S_1 \\ \{a_2 \otimes b_2\}; & \{a_2 \otimes b_2\} \twoheadrightarrow S_1 \end{cases}$$

$$\text{Alle Kombinationen der Instanz } a_3 = \begin{cases} \{a_3 \otimes b_1\}; & \{a_3 \otimes b_1\} \twoheadrightarrow S_1 \\ \{a_3 \otimes b_2\}; & \{a_2 \otimes b_2\} \twoheadrightarrow S_1 \end{cases}$$

Obwohl die Kombination der beiden Instanzen a_3 und b_1 der spezifizierten Bedingung ($a \circledcirc b$) nicht genügt, erfüllt die Realisierung die Spezifikation. Sowohl a_3 als auch b_1 erfüllen bereits in einer Kombination mit anderen Instanzen die Spezifikation.

Für eine einfache Entscheidung, ob die Realisierung eine Spezifikation mit dem Attribut *Multiple* erfüllt, werden die möglichen Kombinationen der Instanzen in eine (Kombinations-)Tabelle eingetragen. In die Tabelle werden in der ersten Zeile alle Instanzen der Aktionen der Realisierung, welche in der Aktionsmenge der Spezifikation vorkommen, aufgetragen. In der ersten Spalte der Tabelle werden die verschiedenen Kombinationsmöglichkeiten der Instanzen der Aktionen im Bezug zur Spezifikation in je eine eigene Zeile eingetragen. Erfüllen die Instanzen der Aktionen die definierte kausale Abhängigkeit, wird in der entsprechenden Zeile und Spalte der beteiligten Aktionen der Wert X eingetragen. Widerspricht die kausale Abhängigkeit der Instanzen der spezifizierten Abhängigkeit, wird der Wert ◯ eingetragen. In der letzten Zeile der Tabelle werden die Einträge zusammengefasst. Enthält eine Spalte in jeder Zeile nur den Wert ◯, so wird auch in der letzten Zeile in der Zusammenfassung der Wert ◯ eingetragen. Ist mindestens einmal der Wert X vorhanden, so ergibt sich der Wert X.

In Analogie zur Aussagenlogik kann der Wert X als *wahr* und der Wert ◯ als *falsch* interpretiert werden. Der Eintrag in der letzten Tabellenzeile ergibt sich dann aus der Verknüpfung der einzelnen Werte durch ein nicht exklusives oder.

Enthält die letzte Zeile nur Einträge vom Wert X, so erfüllt die Realisierung die Spezifikation. Im anderen Fall geben die Einträge ◯ an, welche Instanzen der Aktionen die Spezifikation nicht erfüllen.

Für das aktuelle Beispiel ergibt sich die Tabelle 6.2. Anhand der Einträge in der Tabelle kann direkt abgelesen werden, dass R_1, wie bereits angegeben, die Spezifikation erfüllt. Jede Instanz genügt mindestens einmal der definierten Bedingung. Daher befinden sich in der letzten Zeile der Kombinationstabelle nur Einträge von Typ X (wahr).

	a_1	a_2	a_3	b_1	b_2
$\{a_1 \circledcirc b_1\}$	X	◯	◯	X	◯
$\{a_1 \circledcirc b_2\}$	X	◯	◯	◯	X
$\{a_2 \circledcirc b_1\}$	◯	X	◯	X	◯
$\{a_2 \circledcirc b_2\}$	◯	X	◯	◯	X
$\{a_3 \circledcirc b_1\}$	◯	◯	◯	◯	◯
$\{a_3 \circledcirc b_2\}$	◯	◯	X	◯	X
Erfüllt	X	X	X	X	X

Tabelle 6.2: Kombination aller Instanzen der Aktionen für R_1

Für das zweite Beispiel R_2 ist aus der Netzdarstellung $N[R_2]$ (Abbildung 6.28, Seite 118) ersichtlich, dass vor der Ausführung der Aktion b_1 keine Instanz der Aktion a ausgeführt wird und nach der Aktion a_3 keine Instanz von b ausgeführt wird. Aus diesem Grund kann R_2 die Spezifikation

nicht erfüllen. Dies ist auch aus der Kombinationstabelle 6.3, welche alle möglichen Kombinationen der Instanzen der Aktionen in der Realisierung enthält, ersichtlich. Die zugehörige Spalte der Aktion b_1 gibt an, dass diese Instanz von b die spezifizierte Bedingung nicht erfüllt. Gleiches gilt für die Aktion a_3.

	a_1	a_2	a_3	b_1	b_2
$\{a_1 \circledcirc b_1\}$	◯	◯	◯	◯	◯
$\{a_1 \circledcirc b_2\}$	X	◯	◯	◯	X
$\{a_2 \circledcirc b_1\}$	◯	◯	◯	◯	◯
$\{a_2 \circledcirc b_2\}$	X	◯	◯	◯	X
$\{a_3 \circledcirc b_1\}$	◯	◯	◯	◯	◯
$\{a_3 \circledcirc b_2\}$	◯	X	◯	◯	X
Erfüllt	X	X	◯	◯	X

Tabelle 6.3: Kombination aller Instanzen der Aktionen für R_2

Nach der Betrachtung der beiden Anwendungsbeispiele fassen wir in dem Algorithmus 6.2 die einzelnen Schritte der Überprüfung einer Realisierung mit verschiedenen Instanzen von Aktionen einer Spezifikation mit dem Attribute *Multiple* zusammen.

Algorithmus 6.2 (Verifikationsalgorithmus für Multiple-Spezifikationen)
Sei S *ein Aktionslogik-Modul mit der zugehörigen Netzdarstellung N(S) über der Aktionsmenge* $\mathbb{A}_S$*. In S sind keine Aktionen der Aktionsmenge* $\mathbb{A}_S$ *mehrfach enthalten. Die Spezifikation S ist das Häufigkeitsattribut* Multiple *zugeordnet.*

Die folgenden Schritte sind für die Überprüfung einer Realisierung R mit mehreren Instanzen von Aktionen der Aktionsmenge der Spezifikation notwendig.

1. *Identifikation aller Instanzen der Aktionen in der Realisierung R aus der Aktionsmenge der Spezifikation.*
2. *Aufstellen der Kombinationstabelle für die identifizierten Instanzen der Aktionen.*
3. *Durchführung der Verifikation auf Basis der Netzdarstellung der Realisierung R nach dem Algorithmus 5.2 (Seite 86) für alle die im Schritt 1. identifizierten Instanzen der Aktionen. Eintragen des jeweiligen Überprüfungsergebnis in die entsprechende Zeile in der Kombinationstabelle.*
4. *Auswertung der Kombinationstabelle. Enthält die letzte Zeile der Tabelle keine negativen Einträge (◯), so erfüllt die Realisierung die Spezifikation. Im anderen Fall zeigen die negativen Einträge in der letzten Zeile an, welche Instanzen der Aktionen die spezifizierte Bedingung verletzen.*

One-To-One-Spezifikation

Bei einer Spezifikation mit dem Attribut *Multiple* gehen wir immer davon aus, dass die spezifizierte Abfolge der Aktionen beliebig oft in der Realisierung enthalten sein darf. Bei diesem Spezifikationstyp ist auch, wie am Beispiel der Realisierung R_1 und der Spezifikation S_1 im vorherigen Abschnitt dargestellt, erlaubt, dass auf die Aktion a_1 zwei Instanzen der Aktion b folgen und in der Zwischenzeit zwei weitere Instanzen von a ausgeführt werden. Nach der Definition 6.2 auf Seite 118 ist nur entscheidend, dass jede Instanz einer Aktion mindestens einmal die spezifizierte Bedingung erfüllt. Dieser Ansatz erlaubt es z. B. die korrekte Verwendung der EXAM-Operation *InitDatalogger()* und *StartRecord()* in einem Testablauf zu überprüfen.

Bei einer Spezifikation mit dem Attribute *One-To-One* dürfen hingegen die spezifizierten Aktionen der Spezifikation auch mehrfach in der Realisierung vorkommen, jedoch darf dabei die kausale Ordnung nicht durch andere Aktionen aus der Aktionsmenge der Spezifikation in einem Prozess unterbrochen werden.

Als Beispiel ist eine Abfolge der EXAM-Operationen in der Realisierung

$$R_a = [InitDatalogger() \circledcirc InitDatalogger() \circledcirc DeInitDatalogger()]$$

bei einer *One-To-One-Spezifikation*

$$S_2 = [InitDatenlogger() \circledcirc DeInitDatalogger()]$$

nicht erlaubt, da die zweite Instanz der Operation *InitDatalogger()* die spezifizierte Abfolge unterbricht. Hingegen erfüllt die Abfolge der EXAM-Operationen in der Realisierung

$$R_b = [InitDatalogger() \circledcirc DeInitDatalogger() \circledcirc InitDatalogger() \circledcirc DeInitDatalogger()]$$

die Spezifikation S_2.

Im Gegensatz zu dem Vorgehen bei der Verifikation von *Multiple-Spezifikationen* müssen bei *One-To-One-Spezifikationen* alle in der Realisierung vorkommenden Instanzen der Aktionen aus der Aktionsmenge der Spezifikation auf einmal berücksichtigt werden. Andernfalls ist es nicht möglich, zu entscheiden, ob die spezifizierte kausale Abhängigkeit der Aktionen durch eine Aktion der Aktionsmenge der Spezifikation unterbrochen wird. Dies wird zum einen durch eine Erweiterung der Netzdarstellung der Spezifikation und zum anderen eine Anpassung des Verifikationsalgorithmus ermöglicht, welche wir im Folgenden vorstellen. Wir gehen wieder von der Einschränkung aus, dass in der Spezifikation jede Aktion nur einmal vorkommen darf.

Enthält die Realisierung nur je eine Instanz der in der Spezifikation enthaltenen Aktionen, so kann wie bei *Multiple-Spezifikationen* durch die Anwendung des Algorithmus 5.2 auf Seite 86 überprüft werden, ob die Realisierung die Spezifikation erfüllt. In diesem Fall kann die kausale Abfolge der Aktionen in der Realisierung nicht durch eine weitere Aktion der Aktionsmenge der Spezifikation unterbrochen werden.

Sind mehrere Instanzen der spezifizierten Aktionen in der Realisierung enthalten, ist im ersten Schritt für die Überprüfung der kausalen Abhängigkeiten die Netzdarstellung der Spezifikation zu erweitern. Bei der Erweiterung darf sich jedoch die Anzahl der Schaltfolgen, welche die leere Markierung reproduzieren, nicht verändern.

Betrachten wir als Beispiel für die Erweiterung der Netzdarstellung die Überprüfung der Realisierung R_1 (siehe Abbildung 6.28 auf Seite 118) mit der Spezifikation S_1. Die Erweiterung der Netzdarstellung der Spezifikation $S_1 = (a \circledcirc b)$ ist in Abbildung 6.30 dargestellt. Neben den Elementen aus der Aktionsmenge des AL-Moduls ist in der Netzdarstellung zusätzlich die Stelle s_1 (gelb markiert) enthalten. Diese hat eine Kapazität von eins, d.h. die Stelle darf/kann maximal ein Token speichern. Die Stelle enthält nach dem Schalten der Transitionen s bzw. b eine Markierung und ihr wird beim Schalten der Transitionen a bzw. g eine Markierung entzogen. Die zusätzliche Stelle verändert nicht die Anzahl der Schaltfolgen des Petri-Netzes, welche die leere Markierung reproduzieren.

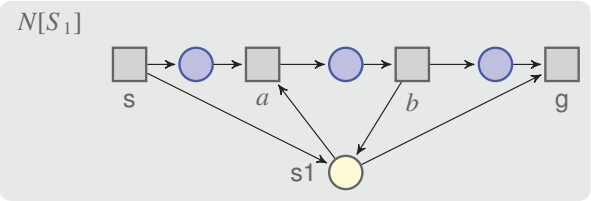

Abbildung 6.30: Erweiterte Netzdarstellung der Spezifikation S_1

Die Stelle ist farblich hervorgehoben, da die zusätzliche Stelle bei der Konjunktion der Realisierung und der Spezifikation gesondert behandelt wird. Mit Hilfe der Stelle $s1$ kann nach der Konjunktion der Netzdarstellungen überprüft werden, ob die kausale Ordnung $(a \circledcirc b)$ der Aktionen a und b in der Realisierung durch eine weitere Instanz der Aktionen a oder b unterbrochen wird. Für diese Überprüfung wird die Netzdarstellung der Realisierung um die zusätzlichen Stellen der Spezifikation erweitert (eingeschränkte Konjunktion), wobei in diesem Beispiel nur die Stelle s_1 und die Kanten zu den Vor- und Nachtransitionen berücksichtigt werden. Würde die spezifizierte Schaltfolge durch weitere Instanzen der Aktionen der Spezifikation unterbrochen, kann die leere Markierung in der sich durch die eingeschränkte Konjunktion ergebende Netzdarstellung nicht mehr reproduziert werden. Daraus ergibt sich die in der folgenden Definition 6.3 beschriebene notwendige aber nicht hinreichende Bedingung für die Überprüfung von *One-To-One-Spezifikationen*.

Definition 6.3 (Notwendige aber nicht hinreichende Bedingung)
Als notwendige aber nicht hinreichende Bedingung für die Überprüfung der kausalen Abhängigkeiten bei einer One-To-One-Spezifikation *muss das Netz der Realisierung nach der Erweiterung (eingeschränkten Konjunktion) mindestens eine Schaltfolge besitzen, welche die leere Markierung reproduziert.*

In der Abbildung 6.31 ist das Ergebnis der eingeschränkten Konjunktion $(N[R_1 \oslash S_1])$ der Realisierung R_1 und der Spezifikation S_1 dargestellt. Die Stelle s_1 ist wieder gelb markiert und es wird deutlich, dass alle Instanzen der spezifizierten Aktionen $(a_1, a_2, a_3, b_1, b_2)$ in der Realisierung bei der Konjunktion berücksichtigt wurden. Die notwendige Bedingung ist in diesem Beispiel nicht erfüllt, da es keine Schaltfolge in dem Petri-Netz gibt, welche die leere Markierung reproduziert. Die Transition a_2 kann Aufgrund der Erweiterung des Netzes durch die Stelle s_1 nicht schalten, da nach dem Schalten der Transition a_1 zuerst eine Instanz der Aktion b hätte schalten müssen. Somit kann die Realisierung die Spezifikation nicht erfüllen.

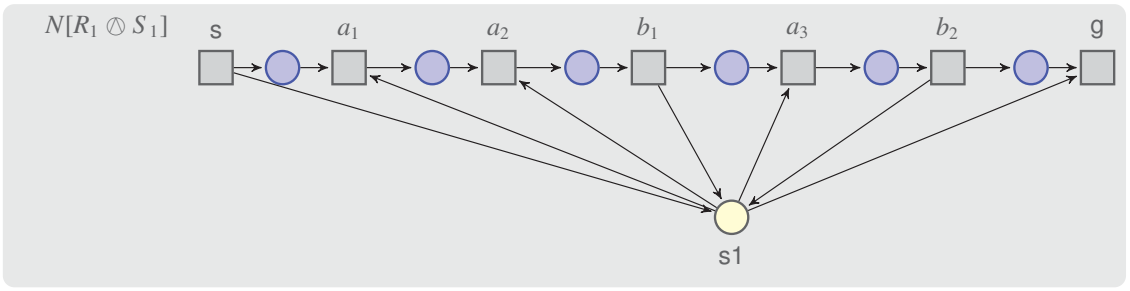

Abbildung 6.31: Konjunktionen der Netzdarstellung von R_1 mit S_1

Betrachten wir als zweites Beispiel die Verifikation der Realisierung

$$R_3 = (a_1 \ominus c \ominus b_1 \ominus a_2 \ominus b_2),$$

deren Netzdarstellung $N[R_3]$ in Abbildung 6.32 auf Seite 124 dargestellt ist.

Die Realisierung R_3 erfüllt die in Spezifikation S_1 definierte Bedingung, da sowohl die kausale Ordnung der Aktionen in der Realisierung mit der Spezifikation übereinstimmen als auch die kausale Ordnung der Aktionen in der Realisierung nicht durch weitere Instanzen der Aktionen aus der Aktionsmenge der Spezifikation unterbrochen werden.

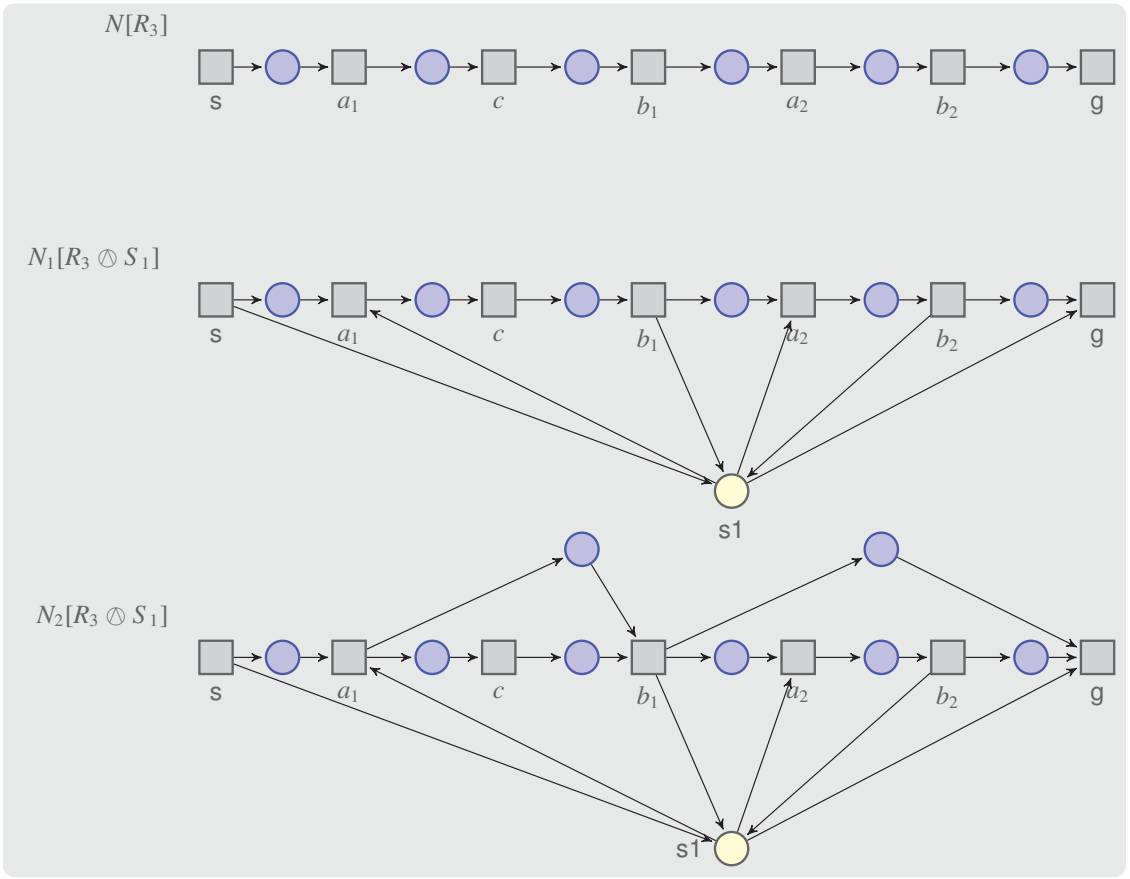

Abbildung 6.32: Netzdarstellung von R_3 und die Konjunktionen mit der Netzdarstellung von S_1

Für den Beweis dieser Aussage prüfen wir im ersten Schritt, ob die notwendige Bedingung (vergleiche Definition 6.3) erfüllt ist. Die Erweiterung der Netzdarstellung der Realisierung durch die Stelle s_1 der Spezifikation $N_1[R_3 \oslash S_1]$ ist in der Abbildung 6.32 dargestellt. Die Bedingung ist erfüllt, da die Schaltfolge $(s, a_1, c, b_1, a_2, b_2)$ die leere Markierung reproduziert.

Im zweiten Schritt muss nun überprüft werden, ob die kausale Ordnung der Aktionen in der Realisierung der spezifizierten Ordnung entspricht. Die kausale Ordnung der Aktionen wird wie bei der Verifikation der *Multiple-Spezifikationen* überprüft, jedoch unter Verwendung der erweiterten Netzdarstellung der Realisierung. Nach dem Algorithmus 6.2 wird für jede Instanz der spezifizierten Aktionen die Konjunktion der erweiterten Netzdarstellung der Realisierung und der Netzdarstellung der Spezifikation durchgeführt und die entsprechende Anzahl an Schaltfolgen bestimmt. Exemplarisch ist die Konjunktion für die Instanzen a_1 und b_1 in der Netzdarstellung $N_2[R_3 \oslash S_1]$ in der Abbildung 6.32 auf Seite 124 dargestellt.

Insgesamt ergibt sich für das Beispiel der Überprüfung der Realisierung R_3 in Bezug auf die Spezifikation S_1 die in der Tabelle 6.4 angegebene Kombinationstabelle für die einzelnen Instanzen der in der Realisierung enthaltenen Aktionen. Da jede Instanz der Aktionen der angegebenen kausalen Abhängigkeit genügt, erfüllt die Realisierung die *One-To-One-Spezifikationen* S_1.

	a_1	a_2	b_1	b_2
$\{a_1 \circledcirc b_1\}$	X	◯	X	◯
$\{a_1 \circledcirc b_2\}$	X	◯	◯	X
$\{a_2 \circledcirc b_1\}$	◯	◯	◯	◯
$\{a_2 \circledcirc b_2\}$	◯	X	◯	X
Erfüllt	X	X	X	X

Tabelle 6.4: Kombination aller Instanzen der Aktionen für R_3

Nach der Betrachtung der Beispiele geben wir nun die Algorithmen für

- die Erweiterung der Netzdarstellung der Spezifikation (Algorithmus 6.3),
- der Erweiterung der Netzdarstellung der Realisierung (Algorithmus 6.4) und
- den erweiterten Verifikationsalgorithmus (Algorithmus 6.5) für Spezifikationen mit dem Attribute *One-To-One*

an.

Wir beginnen die Betrachtung mit dem Algorithmus 6.3 für die Erweiterung der Netzdarstellung eines AL-Moduls für eine Spezifikation mit dem Attribute *One-To-One*. In den vorherigen Beispielen haben wir nur Netzdarstellungen mit einer Schaltfolge, welche die leere Markierung reproduziert, betrachtet. Für eine allgemeingültige Beschreibung für die Überprüfung einer Spezifikation mit dem Attribute *One-To-One* sind auch Aktionslogik-Module zu berücksichtigen, welche mehrere Prozesse enthalten.

Algorithmus 6.3 (Erweiterung der Netzdarstellung für One-To-One-Spezifikationen)
Gegeben sei ein Aktionslogik-Modul S mit der zugehörigen Netzdarstellung $N(S)$ über der Aktionsmenge $\mathbb{A}_S$. In S seien keine Aktionen aus der Aktionsmenge $\mathbb{A}_S$ mehrfach enthalten. Der zugehörigen Spezifikation S ist das Attribut One-To-One *zugeordnet.*

Die Netzdarstellung $N(S)$ wird für die One-To-One-*Spezifikationen durch die folgenden Schritte erweitert:*

1. *Prüfe, ob die Vorstelle der Transition* g *in dem Netz mehr als eine Vortransition enthält. Ist nur eine Vortransition enthalten, bezeichnen wir diese im Folgenden mit* l*. Sind mehrere Vortransitionen enthalten, füge eine neue Transition mit dem Namen* OneToOne *in das Netz ein. Kopiere die Vorstelle der Transition* g*. Die neue Stelle wird dann Vorstelle der Transition* OneToOne*. Verbinde die Transition* OneToOne *mit Hilfe einer neuen Stelle mit der Transition* g*.*

2. *Füge für jede Transition, mit Ausnahme der Transitionen* s, g *und der Transition* l *bzw.* OneToOne*, eine zusätzliche Stelle in das Netz ein. Wir fassen die neuen Stellen zu der Menge $\mathbb{O}$ zusammen.*

3. *Verbinde die Transition* s *mit jeder Stelle aus der Menge $\mathbb{O}$, sodass alle Eingangsmengen dieser Stellen gleich der Transition* s *sind.*

4. *Verbinde die Transition* l *bzw.* OneToOne *mit jeder Stelle aus der Menge $\mathbb{O}$, sodass die Eingangsmenge dieser Stellen um die Transition* l *bzw.* OneToOne *erweitert wird. Die Eingangsmenge jeder dieser Stellen enthält somit die Transition* s *und* l *bzw.* OneToOne*.*

5. *Verbinde die Transition* g *mit jeder Stelle aus der Menge $\mathbb{O}$, sodass die Ausgangsmenge dieser Stellen gleich der Transition* g *ist.*

6. *Erweitere die Ausgangsmenge für je eine Stelle aus der Menge $\mathbb{O}$ um je eine Transition aus der Transitionsmenge* T *von N mit Ausnahme der Transitionen* s, g *und* l *bzw.* OneToOne*. Die Ausgangsmenge einer Stelle aus der Menge $\mathbb{O}$ enthält somit die Transition* g *und eine weitere Transition des Netzes mit Ausnahme von* s, g *und* l *bzw.* OneToOne*.*
   Weiterhin gilt,

   $$\forall x \in T \backslash \{s, g, l, OneToOne\}\ \exists y \in \mathbb{O}, \text{ sodass } y \in {}^\bullet x.$$

Als Beispiel für die Anwendung des Algorithmus 6.3 zur Erweiterung der Netzdarstellung der Spezifikation ist in Abbildung 6.33 die erweitere Netzdarstellung der Spezifikation

$$S_3 = (a \oslash b \oslash c \oslash d)$$

dargestellt. In diesem Fall besitzt die Vorstelle der Transition g nur eine Vortransition.

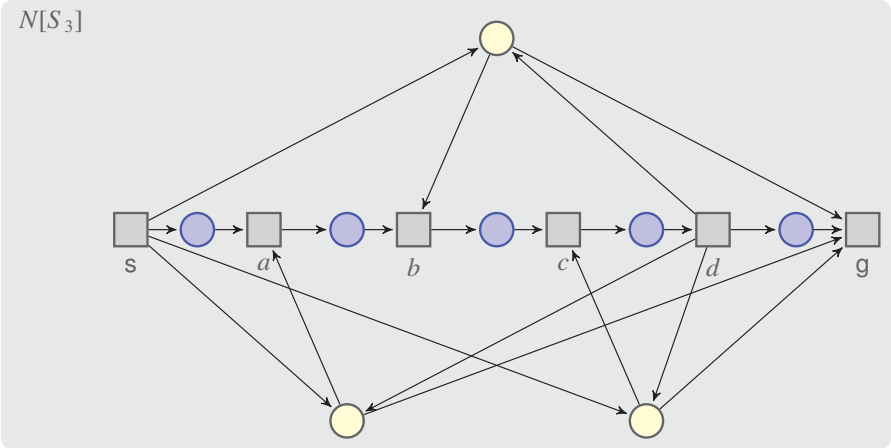

Abbildung 6.33: Netzdarstellung der *One-To-One Spezifikation* S_3

Die in der Abbildung 6.34 darstellte Netzdarstellung der Spezifikation

$$S_4 = (a \oslash (b \oplus c)$$

gibt ein weiteres Beispiel für die Anwendung des Algorithmus 6.3. In diesem Beispiel hat die Vorstelle der Transition g zwei Vortransitionen. Aus diesem Grund wurde die zusätzliche Transition *OneToOne* in das Petri-Netz eingefügt.

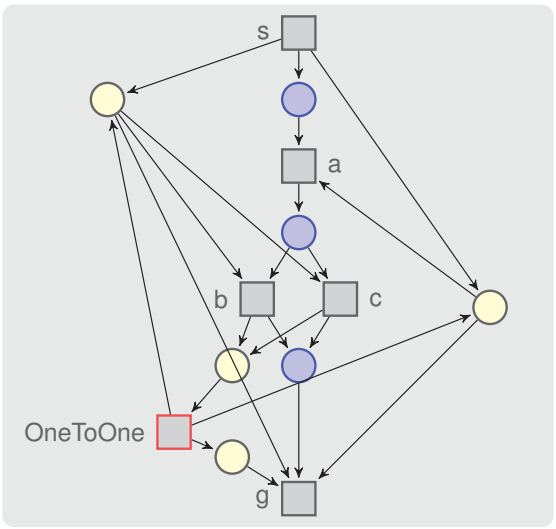

Abbildung 6.34: Erweiterte Netzdarstellung der Spezifikation S_4

Für die Überprüfung, ob die spezifizierte kausale Ordnung in der Realisierung durch Instanzen der Aktionen der Aktionsmenge der Spezifikation unterbrochen wird, muss die Netzdarstellung der Realisierung um die Stellen der Menge $\mathbb{O}$ (siehe Algorithmus 6.3) sowie deren Ein- und Ausgangskanten erweitert werden. Ist in der Spezifikation zusätzlich die Transition *OneToOne* enthalten, sind deren Vor- und Nachstellen und die Transition selbst ebenso zu berücksichtigen.

Die Erweiterung der Netzdarstellung der Realisierung ist im folgenden Algorithmus (Algorithmus 6.4) beschrieben.

Algorithmus 6.4 (Erweiterung der Netzdarstellung der Realisierung)
Wir verwenden für die Erweiterung der Netzdarstellung der Realisierung um die Stellen aus der Menge $\mathbb{O}$ eine Variante des Algorithmus 5.1 zur Konjunktion zweier Netzdarstellungen.

1. *Schritt: Reduktion der erweiterten Netzdarstellung der Spezifikation. Dazu werden in der Netzdarstellung der Spezifikation alle Stellen $st \notin \mathbb{O}$ inklusive aller eingehenden und ausgehenden Kanten gelöscht. Ist in dem Netz die Transition $One-To-One$ enthalten, werden die Vor- und Nachstellen dieser Transition nicht gelöscht.*

2. *Schritt: Erweiterung der Netzdarstellung der Realisierung durch die Anwendung einer Variante des Algorithmus 5.1 mit der reduzierten erweiterten Netzdarstellung der Spezifikation.*

   a) *Identifikation der Starttransitionen (s_1 und s_2) und der Zieltransitionen (g_1 und g_2) in beiden Netzen. Die beiden Transitionen werden jeweils zu einer neuen Starttransition s bzw. Zieltransition g synchronisiert.*

   b) *Identifikation aller Transitionen* a*, welche in beiden Netzen $N[M_1]$ und $N[M_2]$ vorkommen und synchronisiere diese. Sind in der Realisierung mehrere Instanzen der Aktion* a *enthalten, so werden alle Instanzen von* a *mit der Transition a der Spezifikation synchronisiert.*

   c) *In dem neuen Netz dürfen alle Transitionen a und alle Transitionen $\overline{a}$ nicht im gleichen Prozess vorkommen. Daher muss der gegenseitige Ausschluss der beiden Transitionen garantiert werden. Aus diesem Grund wird zu diesen Transitionen eine gemeinsame Vorstelle hinzugefügt.*

   d) *Im letzten Schritt werden alle redundanten Stellen des neuen Netzes gelöscht.*

Basierend auf der erweiterten Netzdarstellung der Spezifikation und der Realisierung können wir nun mit Hilfe des Algorithmus 6.5 überprüfen, ob die spezifizierte Bedingung erfüllt ist.

Algorithmus 6.5 (Verifikationsalgorithmus für One-To-One-Spezifikationen)
Die Überprüfung der Realisierung einer One-To-One-*Spezifikationen ist abhängig von der Anzahl der Instanzen der Aktionen in der Realisierung.*

Sind in der Realisierung keine verschiedenen Instanzen der Aktionen der Aktionsmenge der Spezifikation enthalten, so kann mit Hilfe des Algorithmus 5.2 überprüft werden, ob die Realisierung die Spezifikation erfüllt.

Sind mehrere Instanzen der Aktionen aus der Aktionsmenge der Spezifikation in der Realisierung enthalten, besteht die Verifikation aus den folgenden beiden Schritten:

1. *Als notwendig aber nicht hinreichende Bedingung ist zu überprüfen, ob die erweiterte Netzdarstellung der Realisierung (Algorithmus 6.4) mindestens eine Schaltfolge enthält, die die leere Markierung reproduziert.*

   Existiert keine Schaltfolge, kann die Realisierung die Spezifikation nicht erfüllen, da die kausale Abfolge der Aktionen in der Realisierung durch Instanzen der Aktionen der Spezifikation unterbrochen wird. Dies widerspricht der Definition der Spezifikation mit dem Attribute One-To-One.

2. *Ist die notwendige Bedingung erfüllt, wird mit Hilfe des Algorithmus 6.2 überprüft, ob die kausale Ordnung der Aktionen mit der spezifizierten Abfolge in der Spezifikation übereinstimmt. Für die Konjunktion wird hierbei die erweitere Netzdarstellung der Realisierung und die Netzdarstellung der Spezifikation verwendet.*

Mit Hilfe der vorgestellten Algorithmen soll nun überprüft werden, ob die Realisierung

$$R_3 = (a_1 \oslash b_1 \oslash a_2 \oslash b_2)$$

die Spezifikation S_4 (erweiterte Netzdarstellung ist in Abbildung 6.34 auf Seite 127 dargstellt) erfüllt. In der Spezifikation ist gefordert, dass nach der Aktion a entweder die Aktion b oder c ausgeführt wird. Für die Überprüfung wird die Konjunktion der Netzdarstellung der erweiterten Realisierung und der erweiterten Spezifikation nach dem Algorithmus 6.4 und 6.5 erzeugt.

Die Abbildung 6.35 stellt zum einen die vervollständigte Netzdarstellung der Realisierung um die Transition c und $\overline{c}$ und die Konjunktion der beiden erweiterten Netzdarstellungen der Realisierung R_3 und der Spezifikation S_4 exemplarisch für die Instanzen a_1 und b_1 dar. Die Netzdarstellungen für die weiteren Instanzen ergeben sich entsprechend.

Wir verzichten hier auf die Aufstellung der Kombinationstabelle und leiten das Ergebnis der Überprüfung direkt ab. Nach Betrachtung aller Kombinationen ergibt sich wie erwartet, dass die Realisierung die Spezifikation erfüllt.

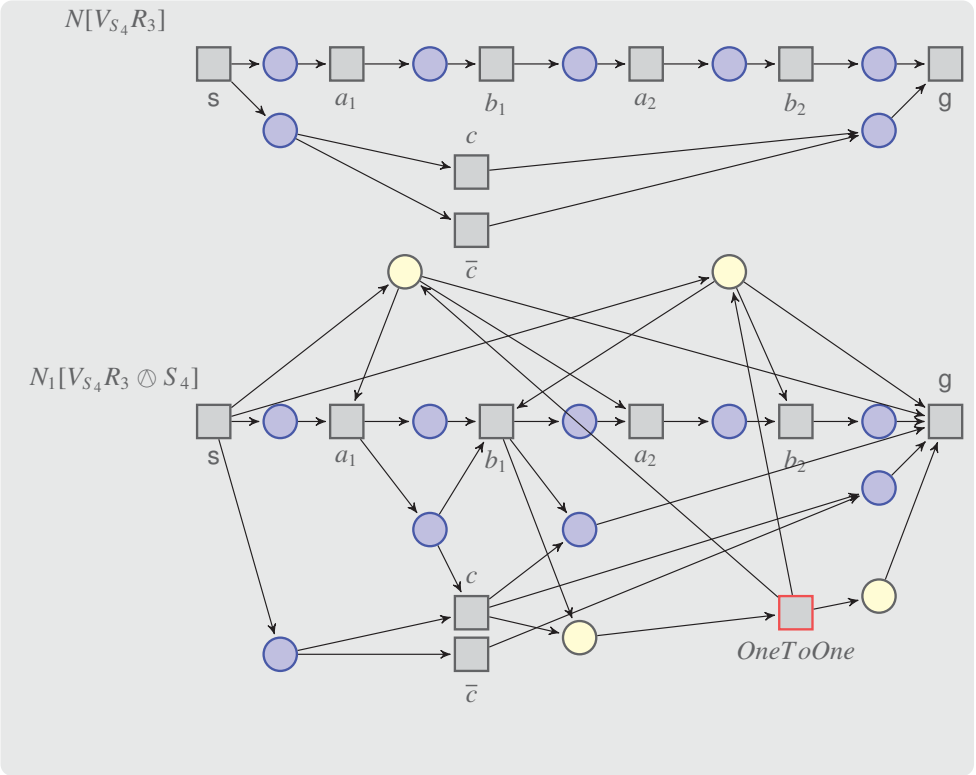

Abbildung 6.35: Konjunktionen der Netzdarstellung von R_3 und S_4

Für die Begründung der im Abschnitt 6.3.1 vorgestellten Transformation der Fragmente vom Typ *while* und *for* der Sequenzdiagramme in die korrespondierende Petri-Netz-Darstellung geben wir für die Verifikation einer Realisierung im Bezug auf eine Spezifikation vom Typ *One-To-One* noch ein weiteres Beispiel.

Für die Überprüfung gehen wir von der Spezifikation $S_5 = [AktionB \oslash AktionC]$ mit dem Attribut *One-To-One* aus und untersuchen, ob die in Abbildung 6.36 auf Seite 131 dargestellte *TestSequence* die Spezifikation S_5 erfüllt. Das Sequenzdiagramm enthält drei Operationsaufrufe und ein Interaktionsfragment vom Typ *while*, welches die Aktion *AktionB* enthält. Nach der Spezifikation muss auf jede Instanz einer Aktion *AktionB* in der Realisierung eine Instanz der Aktion *AktionC* folgen. An dieser Stelle unterscheiden wir drei Möglichkeiten für die Ausführung der While-Schleife. Im ersten Fall ist die Bedingung der Schleife nicht erfüllt und somit wird die Aktion *AktionB* nicht vor der Aktion *AktionC* ausgeführt. Ist die Schleifenbedingung genau einmal wahr, so wird die Aktion *AktionB* genau einmal, wie spezifiziert, vor der Aktion *AktionC* ausgeführt. Wird die Schleife mehr als einmal durchlaufen, werden mehrere Instanzen der Aktion *AktionB* vor der Aktion *AktionC* ausgeführt und somit die Spezifikation S vom Typ *One-To-One* verletzt.

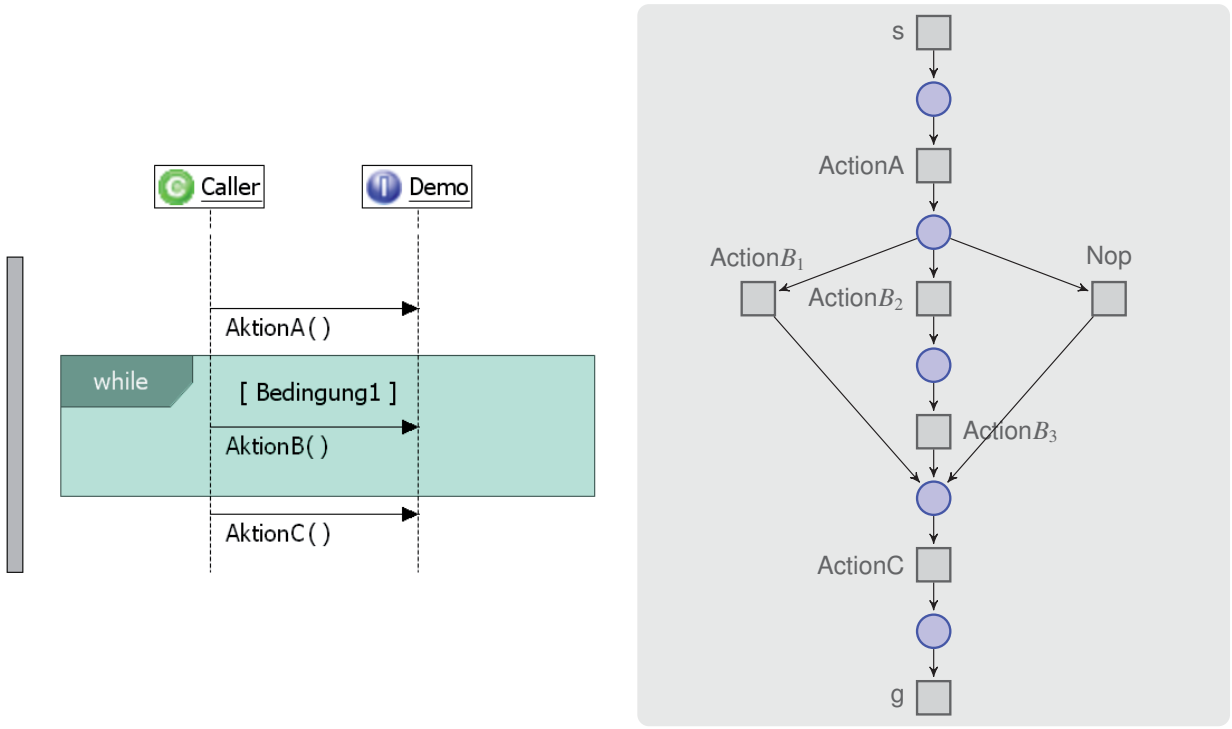

Abbildung 6.36: Sequenz mit While-Fragment

6.4.2 Voraussetzungsattribute: May, Must und Complete

Neben den Häufigkeitsattributen *Single*, *Multiple* und *One-To-One* ist eine weitere Kategorie von Attributen zur Unterscheidung der Spezifikation für die Überprüfung der EXAM-Testabläufe notwendig. Die Attribute in dieser Kategorie bezeichnen wir als *Voraussetzungsattribute*. Eine Vielzahl der kausalen Beziehungen zwischen den Bibliotheksfunktionen in EXAM müssen nur unter bestimmten Bedingungen (Voraussetzungen) erfüllt sein, wohingegen andere spezifizierte Abfolgen immer gültig sein müssen. Ebenso ist die Frage von Interesse, ob *alle* in der Spezifikation definierten Bedingungen im Testablauf eingehalten werden. Für die Unterscheidung, wann eine Überprüfung notwendig ist, ordnen wir den Spezifikationen zusätzlich die Attribute *May*, *Must* und *Complete* zu. Im Folgenden beschreiben wir die verschiedenen Attribute und geben einige Beispiele für ein besseres Verständnis.

May-Spezifikation

Eine Vielzahl der kausalen Abhängigkeiten zwischen EXAM-Operationen müssen nur dann überprüft werden, wenn in dem Testablauf eine bestimmte Operation aufgerufen wird. Als Beispiel betrachten wir die Operationen für die Initialisierung des Datenloggers *initDataLogger()* und das Starten der Datenaufzeichnung mit der Operation *startRecord()*. Aufgrund von Besonderheiten der eingesetzten Datenlogger muss vor jeder Datenaufzeichnung der Datenlogger initialisiert werden. Diese Beziehung impliziert jedoch nicht, dass nach jeder Initialisierung des Datenloggers mit dem Aufruf der Operation *initDataLogger()* die Operation *startRecord()* zum Starten der Signalaufzeichnung aufgerufen werden muss. Aus diesem Grund muss diese Bedingung nur

überprüft werden, wenn in dem Testablauf die Operation *startRecord()* enthalten ist. Operationen wie z. B. *startRecord()* die angeben, dass die *May-Spezifikation* für diesen Testablauf angewendet werden muss, bezeichnen wir als *notwendige Operationen* (*Required-Operation*).

Wir beschreiben die (*Required-Operation*) in der Definition 6.4.

Definition 6.4 (Required-Operation)
Als Required-Operation bzw. Required-Action werden diejenigen Aktionen aus der Aktionsmenge $\mathbb{A}_S$ einer Spezifikation S bezeichnet, die angeben, ob bei deren Vorkommen in einer Realisierung die Spezifikation für die Realisierung überprüft werden muss.

Mit Hilfe der Definition 6.4 können wir nun die Verifikation von Spezifikationen mit dem Attribut *May* beschreiben.

Definition 6.5 (Verifikation von May-Spezifikation)
Gegeben sei eine Spezifikation S über der Aktionsmenge $\mathbb{A}_S$ mit dem Attribut May *und eine Realisierung R über der Aktionsmenge $\mathbb{A}_R$.*

Die Realisierung R muss nur dann auf die in der Spezifikation S definierte Bedingungen überprüft werden, wenn in der Realisierung die als Required-Action *(siehe Definition 6.4) definierten Aktionen der Spezifikation enthalten sind.*

Ist mindestens eine Required-Action *aus der Spezifikation in der Realisierung enthalten, ist der eingesetzte Algorithmus für die Überprüfung abhängig vom Häufigkeitsattribut der Spezifikation (*Single*,* Multiple *oder* One-To-One*).*

Must-Spezifikation

Sind die in der Spezifikation enthaltenen Operationen nicht im Testablauf (Realisierung) enthalten, muss die Verifikation nicht durchgeführt werden, da die Realisierung die definierte Abfolge nicht verletzten kann. Als Beispiel betrachten wir die beiden Operationen *initPlatform()* und *deInitPlatform()* zur Initialisierung und zur Deinitialisierung der Echtzeitplattform. Unabhängig von der Definition der kausalen Abhängigkeit der beiden Operationen in der Spezifikation muss diese Bedingung nur überprüft werden, wenn mindestens eine der beiden Operationen im Testablauf enthalten ist. Sind beide Operationen nicht enthalten, so kann die spezifizierte Bedingung nicht verletzt werden.

Zur Definition dieser Art der Überprüfung verwenden wir Spezifikationen mit dem Attribut *Must*. Bei Spezifikationen mit dem Attribut *Must* muss die Überprüfung immer dann ausgeführt werden, wenn mindestens eine Aktion der Aktionsmenge der Spezifikation in der Realisierung enthalten ist.

Wir beschreiben die Verifikation einer *Must*-Spezifikation in der Definition 6.6.

Definition 6.6 (Verifikation von Must-Spezifikation)
Gegeben sei eine Spezifikation S *über der Aktionsmenge* $\mathbb{A}_S$ *mit dem Attribut* Must *und eine Realisierung* R *über der Aktionsmenge* $\mathbb{A}_R$*.*

Die Realisierung R *muss nur dann auf die Einhaltung der in der Spezifikation* S *definierten Bedingungen überprüft werden, wenn in der Realisierung mindestens eine Aktion aus der Aktionsmenge* $\mathbb{A}_S$ *der Spezifikation enthalten ist. Es muss folglich gelten, dass* $A_S \bigcap A_R \neq \emptyset$*.*

*Der Verifikationsalgorithmus ist dann abhängig von dem Häufigkeitsattribut (*Single*,* Multiple *oder* One-To-One*) der Spezifikation.*

Complete-Spezifikation

Wie in Definition 5.22 auf Seite 76 und der Definition 5.24 auf Seite 78 beschrieben, kann mit dem *direkten Beweisverfahren* in der Aktionslogik überprüft werden, ob die Realisierung die in der Spezifikation definierten Bedingungen erfüllt. Hierbei wird überprüft, ob die Realisierung in Bezug auf die Spezifikation *fehlerfrei* ist. Bei der Übertragung der Aktionslogik auf EXAM bedeutet dies, dass ein Testablauf genau dann *fehlerfrei* in Bezug auf die Spezifikation ist, wenn in jedem möglichen Ausführungspfad die kausale Ordnung der Aktionen mit der Ordnung in der Spezifikation übereinstimmt. Neben der Überprüfung auf die Fehlerfreiheit einer Realisierung im Bezug auf die Spezifikation kann überprüft werden, ob die Realisierung im Bezug auf die Spezifikation *vollständig* ist. Dies bedeutet bei EXAM, dass in dem Testablauf alle in der Spezifikation definierten Beziehungen mindestens einmal enthalten sind.

Soll mit einer Spezifikation überprüft werden, ob alle in der Spezifikation definierten kausalen Abhängigkeiten auch in der Realisierung enthalten sind, verwenden wir in EXAM eine Spezifikation mit dem Attribut *Complete*.

Da mit einer Spezifikation mit dem Attribut *Complete* überprüft werden soll, ob alle in der Spezifikation definierten Abfolgen auch in dem Testablauf enthalten sind, darf vor der Analyse die Struktur des Testablaufs nicht verändert werden. Somit ist die Überprüfung eines Testablaufs im Bezug auf eine Spezifikation mit dem Attribut *Complete* immer unabhängig von der aktuellen Parametrierung des Testablaufs durchzuführen. In Abhängigkeit der Parametrierung werden unter Umständen bei der Transformation des Testablaufs in die zugehörige Petri-Netz-Darstellung Operationen bzw. vollständige Ausführungspfade nicht berücksichtigt. Diese können jedoch für die Entscheidung, ob der Testablauf in Bezug auf die Spezifikation *vollständig* ist, notwendig sein. Es geht an dieser Stelle nicht darum, welcher Ausführungspfad unter der aktuellen Parametrierung ausgeführt wird, sondern es müssen alle (theoretisch) möglichen Ausführungspfade bei der Überprüfung berücksichtigt werden.

Die Definition 6.7 beschreibt die Verifikation einer Spezifikation mit dem Voraussetzungsattribut *Complete*.

Definition 6.7 (Verifikation von Complete-Spezifikation)
Gegeben sei eine Spezifikation S *über der Aktionsmenge* $\mathbb{A}_S$ *mit dem Attribut* Complete *und eine Realisierung* R *über der Aktionsmenge* $\mathbb{A}_R$*.*

Nach der Definition 5.16 auf Seite 74 ist die Realisierung R *in Bezug auf die Spezifikation* S *genau dann vollständig, wenn* S *im Bezug auf* R *fehlerfrei ist.*

*Der in EXAM angewendete Verifikationsalgorithmus ist dabei von dem Häufigkeitsattribut (*Single*,* Multiple *oder* One-To-One*) der Spezifikation abhängig.*

Als Beispiel für die Überprüfung einer Spezifikation mit dem Attribut *Complete* betrachten wir zwei Testabäufe für die Absicherung des Audi Komfortschlüssels[1]. Abbildung 6.37 zeigt einen Auszug aus einem konstruierten Testablauf für die Überprüfung des Motorstarts mit Hilfe des Komfortschlüssels. In dem Testfall wird der Schlüssel zuerst ins Zündschloss gesteckt und anschließend das Fahrzeug (Motor) gestartet. Im zweiten Testfall (T_2), dargestellt in Abbildung 6.38, wird in Abhängigkeit des in einem *ParameterSet* gespeicherten booleschen Werts, entweder der Schlüssel ins Zündschloss gesteckt oder der Schlüssel im Fahrzeuginnenraum abgelegt und anschließend der Motor gestartet. Nach der Ausführung der Operation *SchluesselStecken()* befindet sich der Schlüssel natürlich auch im Fahrzeug, jedoch wird sobald der Schlüssel im Zündschloss steckt, nicht mehr durch die Sensoren überprüft, ob sich ein Schlüssel im Fahrzeug befindet. Somit schließen sich beide Bedingungen gegenseitig aus. Die entsprechenden Netzdarstellungen der beiden Testabläufe T_1 und T_2 sind in der Abbildung 6.39 auf Seite 135 dargestellt.

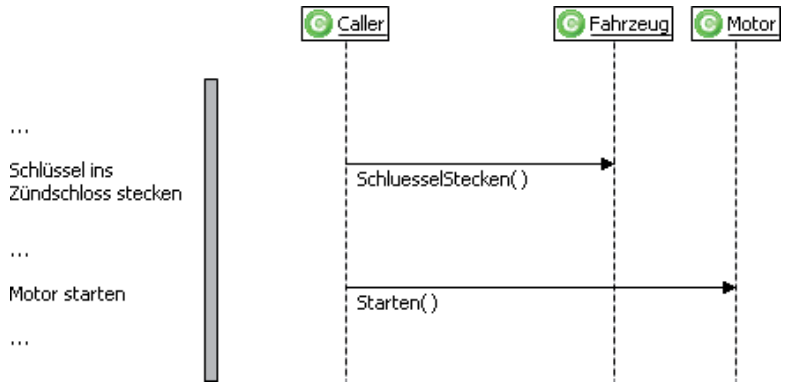

Abbildung 6.37: Testablauf T_1 „Schlüssel ins Zündschloss stecken und Motor starten“

[1]Der Audi Komfortschlüssel ist ein elektronisches Zugangs- und Wegfahrberechtigungssystem, welches es ermöglicht, das Fahrzeug ohne Betätigung der Funkfernbedienung zu ent- bzw. zu verriegeln. Weiterhin erlaubt der Komfortschlüssel das Starten des Fahrzeuges entweder über den Start / Stop-Taster ohne den Schlüssel ins Zündschloss zu stecken oder durch die Verwendung des Zündschlosses. Die Position des Schlüssel im Fahrzeug wird dabei über Sensoren ermittelt. Quelle: Audi Lexikon `http://www.audi.de/de/brand/de/tools/advice/glossary/komfortschluessel.browser.carline_a5sb.html`, Stand vom 21.03.2011

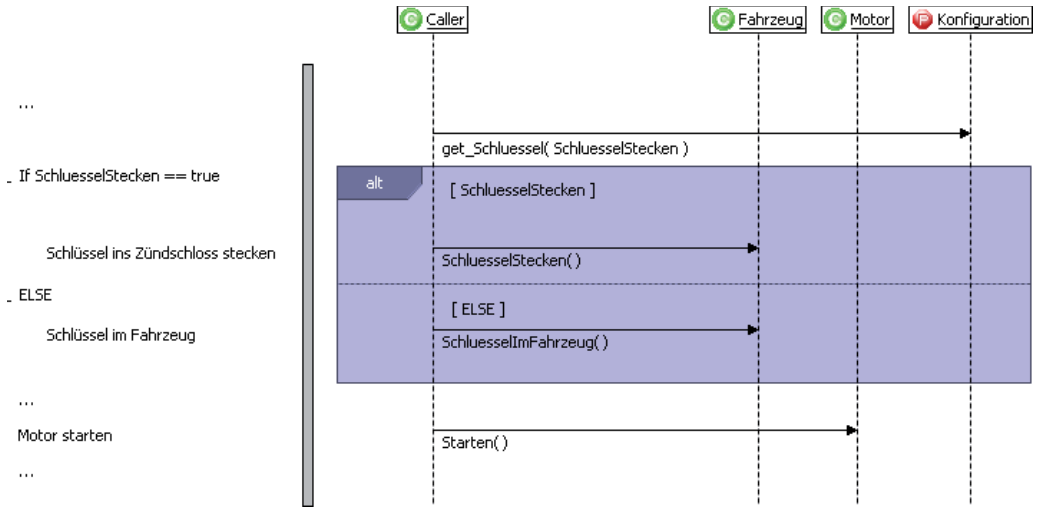

Abbildung 6.38: Testablauf T_2 „Schlüssel ins Zündschloss stecken oder Schlüssel im Fahrzeug und Motor starten“

Wir untersuchen nun, ob die beiden Testabläufe in Bezug auf die Spezifikation

$$S = [\text{SchluesselStecken} \veebar \text{SchluesselImFahrzeug}]$$

fehlerfrei und *vollständig* sind. Die zugehörige Netzdarstellung der Spezifikation S ist in Abbildung 6.40 dargestellt.

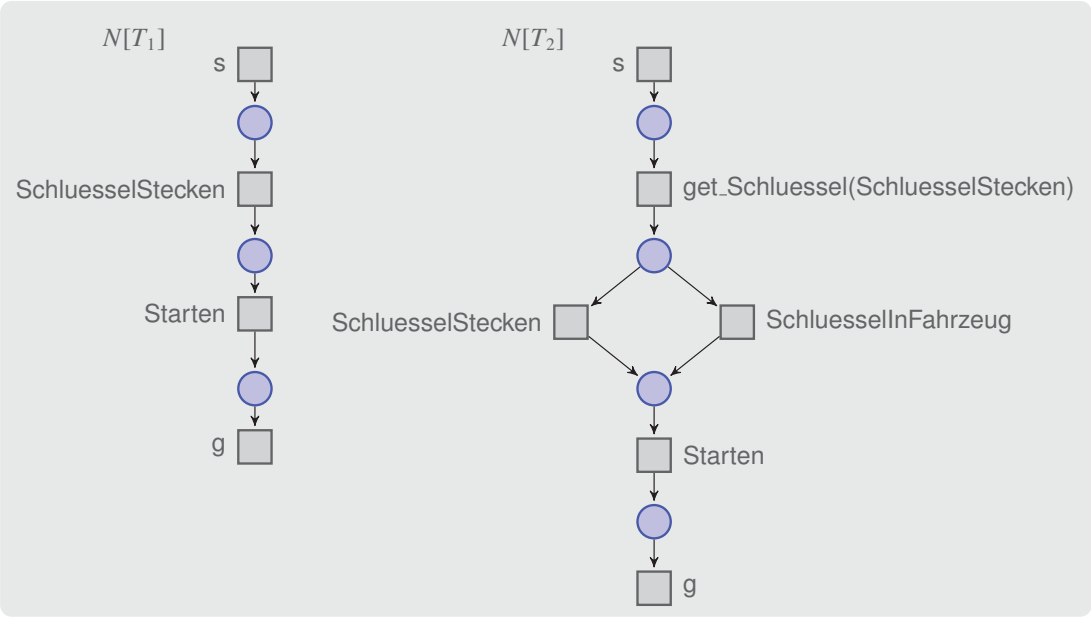

Abbildung 6.39: Netzdarstellung Testabläufe T_1 und T_2

Im *ersten Schritt* gehen wir von der Spezifikation $S_{MultipleMust}$ mit den Attributen *Multiple* und *Must* aus. Da in beiden Realisierungen nur jeweils eine Instanz der Aktionen aus der Aktionsmen-

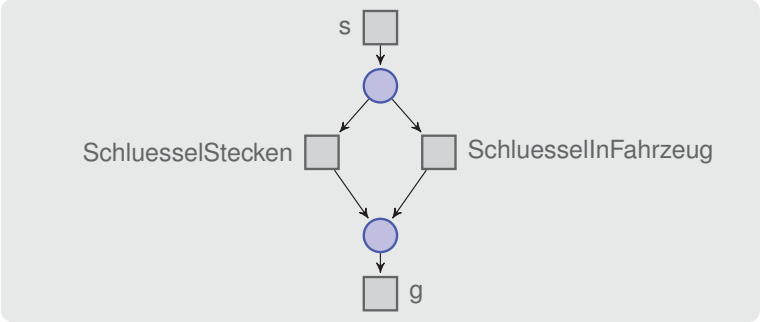

Abbildung 6.40: Netzdarstellung der Spezifikation

ge der Spezifikation vorkommt, kann mit Hilfe des Algorithmus 5.2 auf Seite 86 die Überprüfung durchgeführt werden. Wie auch direkt aus der Netzdarstellung ersichtlich sind T_1 und T_2 im Bezug auf $S_{MultipleMust}$ *fehlerfrei*, da in jedem Ausführungspfad der Testfälle entweder die Operation *SchluesselStecken()* oder *SchluesselImFahrzeug()* enthalten ist, jedoch nie beide.

Im *zweiten Schritt* überprüfen wir, ob beide Realisierungen T_1 und T_2 gegenüber der Spezifikation $S_{CompleteMust}$ mit den Attributen *Complete* und *Must vollständig* sind. Als praktisches Anwendungsbeispiel könnte diese Spezifikation dafür eingesetzt werden, in jedem Testfall für den Komfortschlüssel zu überprüfen, ob immer beide Alternativen zum Starten des Motors berücksichtigt wurden. Mit Hilfe einer entsprechenden Parametrierung könnte somit ein Testablauf beide Anwendungsfälle (Starten des Motors mit Hilfe des Komfortschlüssels via der Start/Stop-Taste oder dem Zündschloss) abdecken.

Nach der Definition 6.7 von Seite 134 testen wir mit Hilfe des Algorithmus 5.2 ob $S_{CompleteMust}$ in Bezug auf T_1 und T_2 fehlerfrei ist. Aus den Netzdarstellungen wird direkt ersichtlich, dass nur T_2 in Bezug auf die Spezifikation vollständig ist, da alle beiden Prozesse der Prozessmenge $\mathbb{P}_{S_{CompleteMust}} = \{(\text{SchluesselStecken}()), (\text{SchluesselImFahrzeug}())\}$ in T_2 enthalten sind. Hingegen ist in T_1 nur ein Prozess mit nur der Aktion *SchluesselStecken()* enthalten.

6.4.3 Kombination der Voraussetzungs- und Häufigkeitsattribute

Nachdem wir im vorherigen Abschnitt die einzelnen Attribute einer Spezifikation betrachtet haben, wollen wir in diesem Abschnitt die Kombination der Attribute zur eindeutigen Bestimmung einer Spezifikation betrachten.

Insgesamt ergeben sich neun verschiedene Arten von Spezifikationen für die Definition der kausalen Abhängigkeiten in EXAM (vergleiche Abbildung 6.26 auf Seite 114). Diese ergeben sich durch die Kombination der drei vorgestellten *Voraussetzungsattribute May*, *Must*, *Complete* und der drei *Häufigkeitsattribute One-To-One*, *Multiple* und *Single*. Die *Voraussetzungsattribute* legen dabei fest, unter welcher Voraussetzung (Bedingung) die Realisierung die in der Spezifikation spezifizierten kausalen Abhängigkeiten erfüllen muss. Die *Häufigkeitsattribute* definieren, wie oft die Aktionen der Spezifikation in der definierten kausalen Ordnung in der Realisierung vorkommen müssen.

Jeder Spezifikation in EXAM muss genau ein *Voraussetzungsattribut* und ein *Häufigkeitsattribut* zugeordnet sein, damit entschieden werden kann, ob und wie eine Realisierung in Bezug auf die Spezifikation überprüft werden muss. Somit ergeben sich die folgenden neun Arten von Spezifikationen:

1. **MayOne-To-One**: Die Spezifikation muss nur dann überprüft werden, wenn in der Realisierung die *Required-Actions* der Spezifikation enthalten sind. Die Realisierung erfüllt die Spezifikation, wenn die kausale Ordnung der Aktionen der Realisierung der spezifizierten Ordnung entspricht und diese nicht durch andere Instanzen von Aktionen aus der Aktionsmenge der Spezifikation unterbrochen werden.

2. **MayMultiple**: Die Spezifikation muss nur dann überprüft werden, wenn in der Realisierung die *Required-Actions* der Spezifikation enthalten sind. Die Realisierung erfüllt die Spezifikation, wenn die kausale Ordnung der Aktionen der Realisierung der spezifizierten Ordnung entspricht. Dabei muss jede Instanz der Aktionen in Kombination mit anderen Instanzen mindestens einmal die spezifizierte Ordnung erfüllen.

3. **MaySingle**: Die Spezifikation muss nur dann überprüft werden, wenn in der Realisierung die *Required-Actions* der Spezifikation enthalten sind. Die Realisierung erfüllt die Spezifikation, wenn die kausale Ordnung der Aktionen der Realisierung der spezifizierten Ordnung entspricht. Dabei darf jede Aktion der Spezifikation nur einmal pro Prozess in der Realisierung enthalten sein.

4. **MustOne-To-One**: Die Spezifikation muss überprüft werden, wenn mindestens eine Aktion der Aktionsmenge der Spezifikation in der Realisierung enthalten ist. Die Realisierung erfüllt die Spezifikation, wenn die kausale Ordnung der Aktionen der Realisierung der spezifizierten Ordnung entspricht und diese nicht durch andere Instanzen von Aktionen aus der Aktionsmenge der Spezifikation unterbrochen werden.

5. **MustMultiple**: Die Spezifikation muss überprüft werden, wenn mindestens eine Aktion der Aktionsmenge der Spezifikation in der Realisierung enthalten ist. Die Realisierung erfüllt die Spezifikation, wenn die kausale Ordnung der Aktionen der Realisierung der spezifizierten Ordnung entspricht. Dabei muss jede Instanz der Aktionen in Kombination mit anderen Instanzen mindestens einmal die spezifizierte Ordnung erfüllen.

6. **MustSingle**: Die Spezifikation muss überprüft werden, wenn mindestens eine Aktion der Aktionsmenge der Spezifikation in der Realisierung enthalten ist. Die Realisierung erfüllt die Spezifikation, wenn die kausale Ordnung der Aktionen der Realisierung der spezifizierten Ordnung entspricht. Dabei darf jede Aktion der Spezifikation nur einmal pro Prozess in der Realisierung enthalten sein.

7. **CompleteOne-To-One**: Es wird überprüft, ob die Realisierung in Bezug auf die Spezifikation vollständig ist. Die Realisierung ist in Bezug auf die Spezifikation *vollständig*, wenn in der Realisierung jeder Prozess der Spezifikation enthalten ist. Dabei muss die kausale Ordnung der Aktionen der Realisierung der spezifizierten Ordnung entsprechen und diese dürfen nicht durch andere Instanzen von Aktionen aus der Aktionsmenge der Spezifikation unterbrochen werden.

8. **CompleteMultiple**: Es wird überprüft, ob die Realisierung in Bezug auf die Spezifikation vollständig ist. Die Realisierung ist in Bezug auf die Spezifikation *vollständig*, wenn in der Realisierung jeder Prozess der Spezifikation enthalten ist. Dabei dürfen die Aktionen der Spezifikation in der Realisierung mehrfach enthalten sein.
9. **CompleteSingle**: Es wird überprüft, ob die Realisierung in Bezug auf die Spezifikation vollständig ist. Die Realisierung ist in Bezug auf die Spezifikation *vollständig*, wenn in der Realisierung jeder Prozess der Spezifikation genau einmal enthalten ist. Dabei dürfen in einem Prozess der Realisierung die Aktionen der Spezifikation nur einmal enthalten sein.

7 Anwendungsbeispiel

In diesem Kapitel verdeutlichen wir die im Kapitel 6 beschriebenen Verfahren zur Überprüfung der kausalen Abhängigkeiten zwischen den Bibliotheksfunktionen in EXAM an einem praktischen Anwendungsbeispiel. Wir betrachten einen Testablauf zur Überprüfung der in Kapitel 2 beschriebenen ACC-Funktion. Der Testablauf ist für ein besseres Verständnis sehr stark vereinfacht. Reale Testabläufe enthalten in der Regel mehrere zehntausend Testschritte, mit mehreren hundert Interaktionsfragmenten bzw. Kontrollknoten. Entsprechend enthält auch die korrespondierende Petri-Netz-Darstellung dieser Elemente mehrere tausend Transitionen, Stellen und Verzweigungen. Eine Betrachtung der dadurch entstehenden Komplexität und die Auswirkungen auf den praktischen Einsatz der Methode diskutieren wir ausführlich in Kapitel 8.

Die Aufteilung dieses Kapitels orientiert sich an den einzelnen Phasen der Verifikation,

- Festlegung der kausalen Abhängigkeiten mit Hilfe von Spezifikationen,
- Transformation der EXAM-Elemente in eine korrespondierende Petri-Netz-Darstellung,
- Durchführung der Verifikation und
- Bewertung der Ergebnisse der Verifikation und Korrektur der EXAM-Elemente,

die wir in Kapitel 6 beschrieben und in der Abbildung 6.1 auf Seite 92 dargestellt haben.

Im ersten Schritt definieren wird in diesem Kapitel die für dieses Anwendungsbeispiel relevanten kausalen Abhängigkeiten zwischen den Operationen der Funktionsbibliothek von EXAM mit Hilfe von sechs Spezifikationen. Im folgenden Schritt beschreiben wir den Testablauf und transformieren die entsprechenden EXAM-Elemente in eine korrespondierende Petri-Netz-Darstellung. Mit Hilfe der in Kapitel 6 vorgestellten Algorithmen überprüfen wir im Anschluss, ob der Testablauf die definierten Spezifikationen erfüllt.

7.1 Festlegung der Spezifikationen

Wir beschreiben die kausalen Abhängigkeiten zwischen den Operationen der Funktionsbibliothek in EXAM mit Hilfe je eines Moduls der Aktionslogik und ordnen jeder Spezifikation je ein Häufigkeits- und ein Voraussetzungsattribut zu. Bei allen Spezifikationen mit dem Attribut *May* geben wir die entsprechenden *Required-Operationen* zusätzlich an.

In der folgenden Übersicht sind alle Spezifikationen enthalten, die in diesem Beispiel von dem Testfall erfüllt werden müssen.

S_1 = {initDataLogger() $\oslash$ startRecord()},
Typ der Spezifikation: May-Multiple,
Required-Operation: (startRecord())

S_2 = {initPlatform() $\oslash$ deInitPlatform()},
Typ der Spezifikation: Must-Single

S_3 = {(SchuesselStecken() $\oplus$ SchluesselImFahrzeug()) $\oslash$ MotorStarten()},
Typ der Spezifikation: May-Multiple,
Required-Operation: (MotorStarten())

S_4 = {startMeasurement() $\oslash$ startRecord()},
Typ der Spezifikation: Must-Multiple

S_5 = {ACCAktivieren() $\oslash$ ACCDeaktivieren()},
Typ der Spezifikation: May-One-To-One,
Required-Operationen: (ACCAktivieren(), ACCDeaktivieren())

S_6 = {(SchuesselStecken() $\oplus$ SchluesselImFahrzeug())},
Typ der Spezifikation: Complete-Multiple

Für ein besseres Verständnis beschreiben wir die einzelnen Spezifikation im Folgenden im Detail.

In der Spezifikation S_1 = {initDataLogger() $\oslash$ startRecord()} wird definiert, dass vor dem Aufruf der Operation *startRecord()* die Operation *initDataLogger()* ausgeführt werden muss. Durch die Attribute *May-Multiple* wird ausgesagt, dass die Spezifikation nur dann erfüllt sein muss, wenn die *Required*-Operation, in diesem Fall *startRecord()*, in dem Testablauf enthalten ist. Durch das Attribut *Multiple* wird angegeben, dass nach dem Aufruf *initDataLogger()* die Operation *startRecord()* n-mal ($n \geq 0$, $n \in \mathbb{N}$) aufgerufen werden darf.

Durch die Spezifikation S_2 = {initPlatform() $\oslash$ deInitPlatform()} ist definiert, dass die Operation *initPlatform()* vor der Aufruf der Operation *deInitPlatform()* ausgeführt werden muss. Das Attribut *Must* gibt an, dass die Bedingung zwingend erfüllt sein muss, wenn eine der Operationen in dem Testablauf enthalten ist. Das Attribute *Single* besagt, dass die beiden Operationen nur einmal im Testablauf vorkommen dürfen.

Die dritte Spezifikation (S_3 = {(SchuesselStecken()$\oplus$SchluesselImFahrzeug())$\oslash$MotorStarten()}) besagt, dass entweder die Operation *SchluesselStecken()* oder (exklusives oder) die Operation *SchluesselImFahrzeug()* vor der Operation *MotorStarten()* aufgerufen werden muss. Wie bei der ersten Spezifikation ist der Typ der Spezifikation *May-Multiple*. Daher muss die Spezifikation nur dann erfüllt sein, wenn die Operation *MotorStarten()* im Testablauf vorkommt. Das Attribut *Multiple* gibt hier an, dass nach dem Aufruf *SchluesselStecken()* bzw. *SchluesselImFahrzeug()* die Operation *MotorStarten()* n-mal ($n \geq 1$, $n \in \mathbb{N}$) aufgerufen werden darf.

Die Spezifikation S_4 = {startMeasurement() $\oslash$ startRecord()} vom Typ *Must-Multiple* beschreibt, dass sowohl die Operation *startMeasurement()* als auch die Operation *startRecord()* im Testablauf

vorkommen muss. Beide Operationen dürfen nach der Spezifikation S_4 beliebig oft vorkommen. Durch den Aufruf von *startMeasurement()* wird die Signalaufzeichnung der simulierten Werte im Prüfstandsmodell gestartet. Die Operation *startRecord()* hingegen startet die Datenaufzeichnung des Datenloggers für die Aufzeichnung des realen Fahrzeugbus. Da der Prüfstand nicht alle Botschaften des Fahrzeugs aufzeichnen kann, ist für eine Auswertung der Testergebnisse eine Kombination der beiden Aufzeichnung notwendig.

Die vorletzte Spezifikation (S_5 = {ACCAktivieren() $\circledcirc$ ACCDeaktivieren()}) gibt an, dass bei der Verwendung der Operationen *ACCAktivieren()* und *ACCDeaktivieren()* der Aufruf von *ACCAktivieren()* immer vor dem Aufruf von *ACCDeaktivieren()* erfolgen muss. Der Typ der Spezifikation gibt dabei an, dass es sich hier um eine eins-zu-eins Beziehung zwischen den Operationen handelt. Hintergrund der Spezifikation ist die Fahrzeugfunktion ACC. Die Funktion kann erst wieder aktiviert werden nachdem sie deaktiviert wurde.

In dem Testablauf soll der Einsatz des Komfortschlüssels abgesichert werden. Damit sichergestellt werden kann, dass in dem Testablauf beide möglichen Alternativen (Schlüssel steckt im Zündschloss oder der Schlüssel ist im Innenraum des Fahrzeugs abgelegt) modelliert sind, ist in der Spezifikation S_6 = {(SchuesselStecken() $\oplus$ SchluesselImFahrzeug())} definiert, dass sowohl der Operationsaufruf *SchluesselStecken()* als auch der Aufruf *SchluesselImFahrzeug()* enthalten ist. Durch den Typ *Complete-Multiple* wird festgelegt, dass beide Elemente enthalten sein müssen, wobei jedes Element beliebig oft vorkommen darf. In einem Prozess des AL-Moduls darf jedoch entweder die Operation *SchluesselStecken()* oder die Operation *SchluesselImFahrzeug()* enthalten sein.

Nach der Beschreibung der einzelnen Spezifikationen ist für die Durchführung der Verifikation die Transformation der Module in die korrespondierende Petri-Netz-Darstellung notwendig. Die entsprechenden Netzdarstellungen der Spezifikationen sind in den Abbildungen 7.1 bis 7.6 dargestellt.

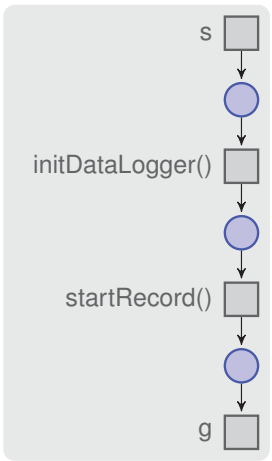

Abbildung 7.1: Netzdarstellung der Spezifikation S_1

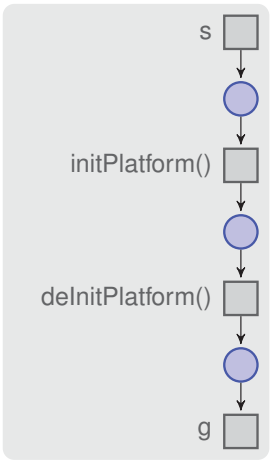

Abbildung 7.2: Netzdarstellung der Spezifikation S_2

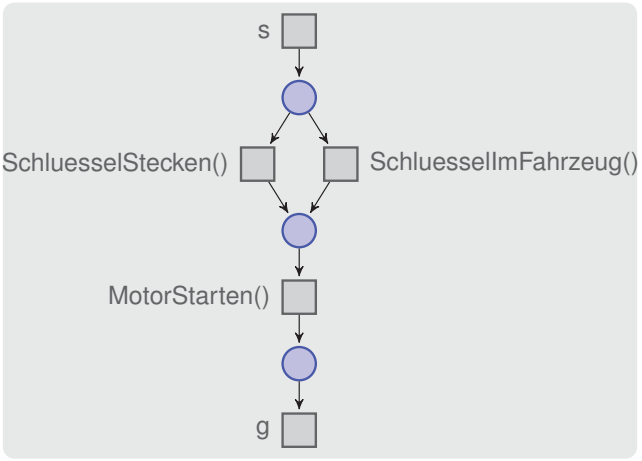

Abbildung 7.3: Netzdarstellung der Spezifikation S_3

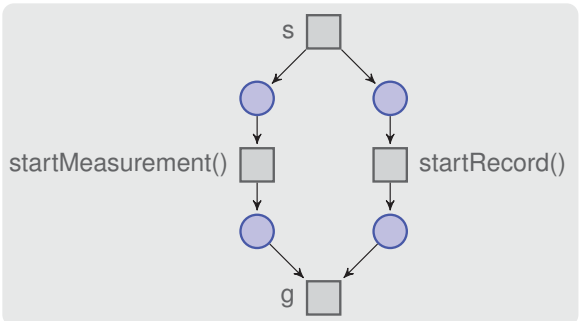

Abbildung 7.4: Netzdarstellung der Spezifikation S_4

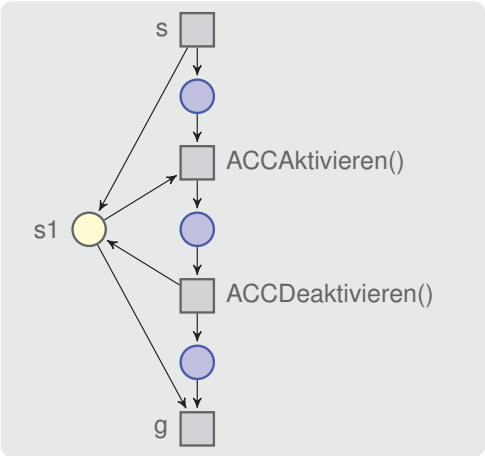

Abbildung 7.5: Netzdarstellung der Spezifikation S_5

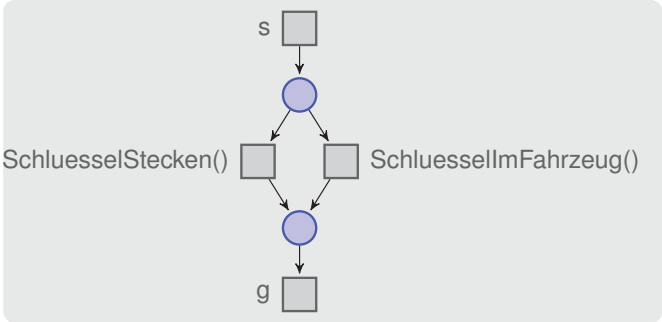

Abbildung 7.6: Netzdarstellung der Spezifikation S_6

7.2 Beispiel eines EXAM-Testablaufs

Wie bereits am Anfang dieses Kapitels beschrieben, verwenden wir für die Demonstration der Anwendbarkeit der vorgestellten Methoden einen Testablauf zur Überprüfung der ACC-Funktion. Der Testfall ist für die Demonstration der Anwendbarkeit der Methode sehr stark vereinfacht. Im Detail wird in dem Testfall geprüft, ob durch eine hintereinander ausgeführte zweimalige Aktivierung der ACC-Funktionen ein Fehlerspeicherwert im Motorsteuergerät eingetragen wird. Zusätzlich zur Auswertung des Fehlerspeichers werden für die Auswertung des Testfalls alle Bussignale mit Hilfe eines Datenloggers aufgezeichnet. Der Testablauf ist generisch für die Verwendung des Komfortschlüssels modelliert, sodass in Abhängigkeit der Parametrierung des Testablaufs sowohl der Schlüssel ins Zündschloss gesteckt als auch im Innenraum des Fahrzeugs positioniert werden kann.

Das in der Abbildung 7.7 dargestellte *UseCaseDiagram* zeigt die Aufteilung des Testfalls *TestCaseExample* in die drei *TestSequences*: *PreSequence*, *ActionSequence* und *PostSequence*. Die *Testsequence PreSequence* wird mit Hilfe des *ParameterSet Schluessel* für die Auswahl der Positionierung des Komfortschlüssels parametriert.

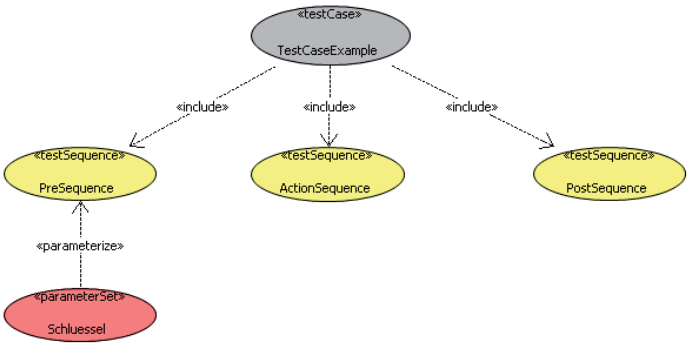

Abbildung 7.7: Darstellung des *TestCases TestCaseExample*

Das Sequenzdiagramm des *TestCases TestCaseExample* (siehe Abbildung 7.8) enthält drei *TestSteps*, die jeweils mit Hilfe eines *UseCaseCall* die *TestSequence PreSequence*, *ActionSequence* und *PostSequence* in der angegebenen Reihenfolge aufrufen.

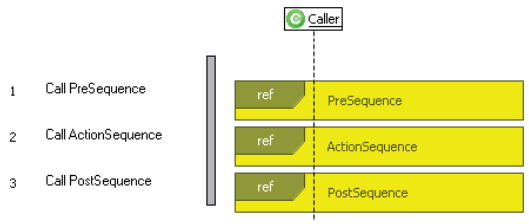

Abbildung 7.8: Testablauf des *TestCases TestCaseExample*

In der *PreSequence* (siehe Abbildung 7.9) werden für die Überprüfung der ACC-Funktion alle notwendigen Initialisierungen und Konfigurationen des Prüfstands, der benötigten Prüfhardware und des virtuellen Fahrzeugs (Echtzeitmodelle) durchgeführt.

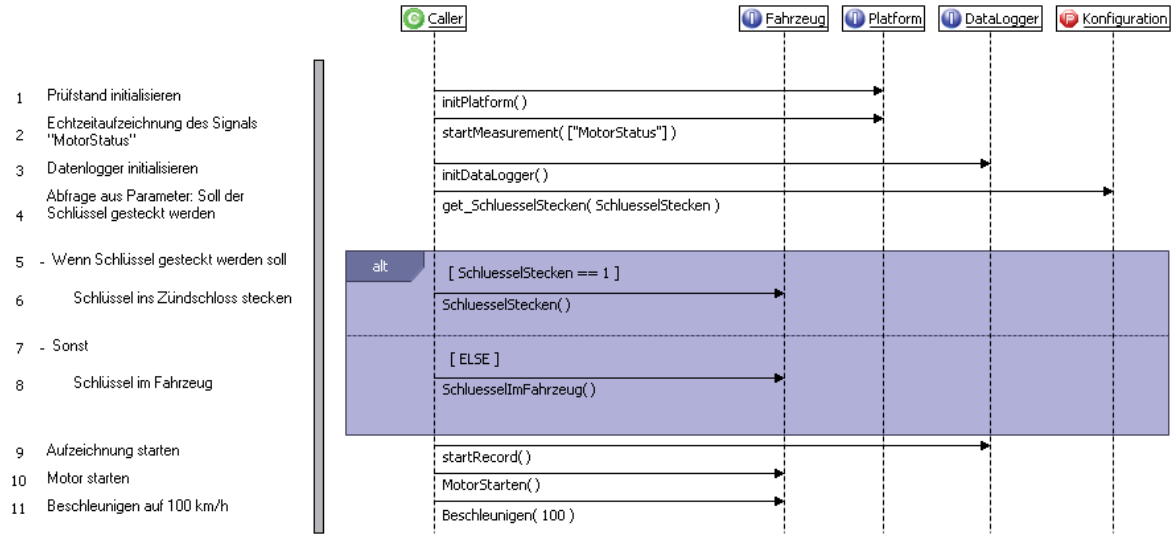

Abbildung 7.9: Operationsaufrufe der *TestSequence PreSequence*

Im ersten Schritt der *PreSequence* wird mit dem Operationsaufruf *initPlatform()* der Prüfstand initialisiert. Mit dem folgenden Operationsaufruf wird definiert, dass das Signal *MotorStatus()* während des Testablaufs aufgezeichnet werden soll. Zusätzlich sollen auch alle Bussignale und Botschaften aufgezeichnet werden. Der dazu notwendige Datenlogger wird im dritten *TestStep* initialisiert.

In dem Testablauf wird der bereits beschriebene Komfortschlüssel als Berechtigungssystem zum Starten des Motors verwendet. Für die Entscheidung, ob der Schlüssel ins Zündschloss gesteckt oder im Fahrzeug positioniert werden soll, wird im vierten Schritt der Parameterwert *Schluessel-Stecken()* aus dem zugeordneten *ParameterSet* ausgelesen und in der Variablen *SchluesselStecken* gespeichert. Die Variable wird im folgenden Interaktionsfragment vom Typ *alt* für die Auswertung der Bedingung verwendet. Ist der Wert der Variablen gleich Eins, wird der Schlüssel im sechsten *TestStep* ins Zündschloss gesteckt. Ist der Wert ungleich Eins, wird der Schlüssel durch den Operationsaufruf *SchluesselImFahrzeug()* im achten *TestStep* im Fahrzeug positioniert. In diesem Beispiel ist der in dem Parameter gespeicherte Wert gleich Eins, sodass der Schlüssel ins Zündschloss gesteckt wird.

Bevor der Motor durch den Operationsaufruf *MotorStarten()* im zehnten *TestStep* gestartet wird, erfolgt der Start der Aufzeichnung der Bussignale und Botschaften mit Hilfe des Datenloggers durch den Aufruf *startRecord()*. Im letzten Schritt wird das virtuelle Fahrzeug auf eine Geschwindigkeit von 100 km/h beschleunigt.

Nach der Ausführung der *PreSequence* ist der für die Überprüfung der ACC-Funktion benötigte Zustand des Prüfstands und des virtuellen Fahrzeugs hergestellt und die eigentliche Überprüfung der ACC-Funktion kann ausgeführt werden.

Die Stimulation der ACC-Funktion erfolgt in der *ActionSequence*. Das zugehörige Sequenzdiagramm ist in Abbildung 7.10 dargestellt.

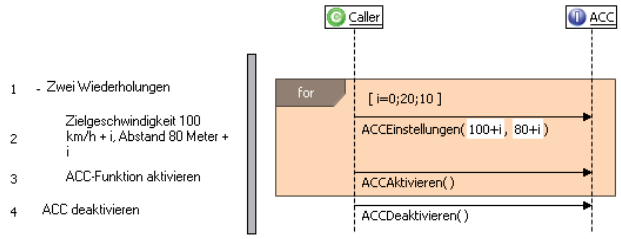

Abbildung 7.10: Operationsaufrufe der *TestSequence ActionSequence*

Im ersten *TestStep* der *ActionSequence* wird ein Interaktionsfragment vom Typ *for* mit der Schleifenvariablen *i* definiert. Insgesamt wird die Schleife zweimal durchlaufen und dabei in jedem Durchlauf die Variable *i* um zehn inkrementiert. Innerhalb der Schleife wird im zweiten Testschritt die Wunschgeschwindigkeit auf 100 km/h + *i* und der Sicherheitsabstand auf 80 Meter + *i* eingestellt. Somit ergibt sich im ersten Schleifendurchlauf eine Wunschgeschwindigkeit von 110 km/h mit einem Sicherheitsabstand von 90 Metern und im zweiten Schleifendurchlauf eine Wunschgeschwindigkeit von 120 km/h mit einem Sicherheitsabstand von 100 Metern. Im Anschluss erfolgt die Aktivierung der ACC-Funktion mit Hilfe der Operation *AccAktivieren()*. Nach der Ausführung des Interaktionsfragments wird die ACC-Funktion deaktiviert.

Die Auswertung der Messdaten und die Deinitialisierung des Prüfstands erfolgt in der *PostSequence* (siehe Abbildung 7.11).

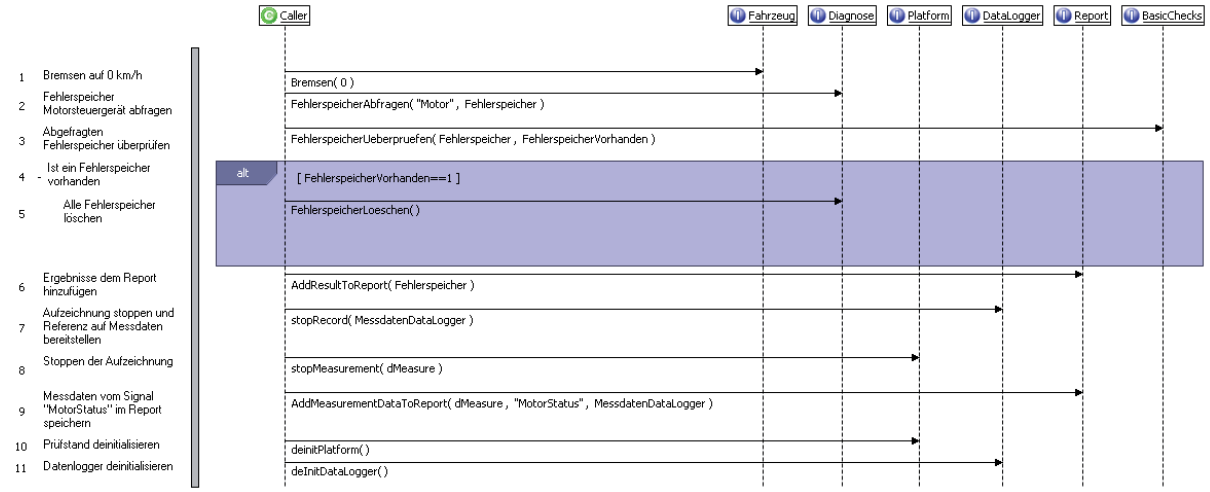

Abbildung 7.11: Operationsaufrufe der *TestSequence PostSequence*

Im ersten *TestStep* der *PostSequence* wird das virtuelle Fahrzeug bis zum Stillstand abgebremst. Im Anschluss wird der Fehlerspeicher des Motorsteuergeräts ausgelesen und im dritten Schritt die Einträge im Fehlerspeicher überprüft. Ist ein Fehlerspeichereintrag vorhanden, wird die Variable *FehlerspeicherVorhanden* auf den Wert Eins gesetzt und im folgenden Interaktionsfragment vom Typ *alt* alle Fehlerspeichereinträge gelöscht. Im sechsten *TestStep* wird der abgefragte Fehlerspeicherwert dem Testreport hinzugefügt. Nach der Überprüfung des Fehlerspeichers wird im siebten und achten Schritt die Aufzeichnung der Bussignale und die Aufzeichnung des Signals *Motor-*

Status gestoppt. Mit dem Operationsaufruf *AddMeasurementDataToReport(dMeasure, „Motor-Status“, MessdatenDataLogger)* werden die aufgezeichneten Messdaten des Signals *MotorStatus* und die Messdaten der Bussignalaufzeichnung für eine spätere manuelle Auswertung dem Report hinzugefügt. Die letzten beiden Operationsaufrufe deinitialisieren den Prüfstand und den verwendeten Datenlogger.

7.3 Transformation in die Netzdarstellung

In diesem Abschnitt beschreiben wir die für die Verifikation des Testablaufs notwendige Transformation der EXAM-Elemente in die korrespondierenden Petri-Netz-Darstellungen.

Wie in Kapitel 6 beschrieben wird jedes an dem Testablauf beteiligte EXAM-Element zuerst separat in eine Netzdarstellung transformiert und vor der Ausführung der Verifikation werden die Netzdarstellungen der einzelnen Elemente zu einem gemeinsamen Netz zusammengefügt. In den Sequenz- und Aktivitätsdiagrammen wird die Referenz auf ein weiteres EXAM-Element mit Hilfe eines *UseCaseCalls* modelliert. Entsprechend verwenden wir in der Netzdarstellung für die Referenzierung einer weiteren Netzdarstellung eine Transition mit dem Namen der referenzierten Netzdarstellung. Diese Transitionen sind in den folgenden Netzdarstellungen durch eine rote Transition mit schwarzem Rahmen dargestellt.

Der in diesem Beispiel beschriebene Testablauf besteht aus den vier EXAM-Elementen *TestCaseExample*, *PreSequence*, *ActionSequence* und *PostSequence*. Im Folgenden betrachten wir die jeweilige Netzdarstellung der einzelnen Elemente und am Ende des Abschnitts das kombinierte Petri-Netz.

In den folgenden Netzdarstellungen der EXAM-Elemente wird dem Namen jeder Transition, welche keine Referenz zu einer weiteren Netzdarstellung repräsentiert, zusätzlich zu dem Namen der Operationen ein Präfix vorangestellt. Dieses Präfix erlaubt in den Netzdarstellungen im folgenden Abschnitt die Transitionen nur mit Hilfe des Präfix zu beschriften und damit eine kompaktere Darstellung der Netze zu ermöglichen.

In der Abbildung 7.12 ist die Netzdarstellung des *TestCase TestCaseExample* dargestellt. In dem Testfall werden die entsprechenden *TestSequences PreSequence*, *ActionSequence* und *PostSequence* mit Hilfe von *UseCaseCalls* aufgerufen. Daher enthält die korrespondierende Netzdarstellung neben den beiden Transitionen *s* und *g* die drei Transitionen *PreSequence*, *ActionSequence* und *PostSequence*.

Die Abbildung 7.13 auf Seite 150 stellt die entsprechenden Netzdarstellung der Elemente *PreSequence*, *ActionSequence* und *PostSequence* dar.

Zur optischen Unterscheidung sind diejenigen Transitionen, die eine Operation innerhalb eines Interaktionsfragments repräsentieren, in einer anderen Farbe dargestellt. Für die Operationen in einem Interaktionsfragment vom Typ *alt* werden grüne Transitionen und für Operationen, die in einem Interaktionsfragment vom Typ *for* enthalten sind, gelbe Transitionen verwendet.

Betrachten wir zuerst die Netzdarstellung der *PreSequence*. In dem EXAM-Element ist ein Interaktionsfragment vom Typ *alt* enthalten. Die Entscheidung, welcher Teil der Alternative während

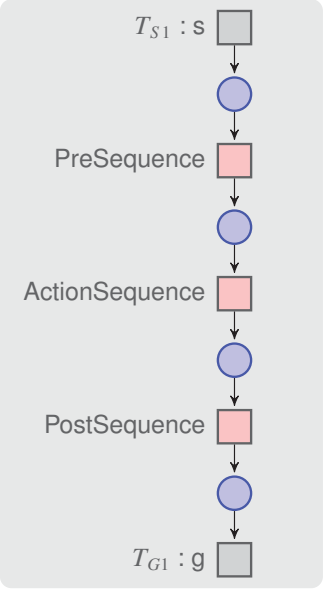

Abbildung 7.12: Netzdarstellung des *TestCases TestCaseExample*

der Testausführung ausgeführt wird, ist abhängig von der Parametrierung des Testfalls. In diesem Beispiel könnte schon vor der Ausführung des Testfalls am Prüfstand mit Sicherheit festgestellt werden, dass zur Laufzeit nur der Ausführungspfad mit der Operation *SchluesselStecken()* möglich ist. Da jedoch eine der Spezifikation vom Typ *Complete* ist, müssen in der entsprechenden Netzdarstellung alle Ausführungspfade der Alternative enthalten sein. Wie im Kapitel 6.4.2 beschrieben, wird die Struktur des Testablaufs bei der Transformation in die Netzdarstellung verändert, wenn aufgrund eines Parameters in einem Interaktionsfragment eine oder mehrere Ausführungspfade ausgeschlossen werden. Ziel einer Spezifikation vom Typ *Complete* ist es zu überprüfen, ob die Struktur des Testablaufs der spezifizierten Struktur entspricht. Also ob alle spezifizierten kausalen Abhängigkeiten der Spezifikation auch in der Realisierung enthalten sind. Daher muss eine Spezifikation mit dem Attribute *Complete* immer auf die unveränderte Struktur des Testablaufs angewendet werden. Aus diesem Grund werden für das folgende Beispiel in der Netzdarstellung beide (theoretisch) möglichen Ausführungspfade (Operation *SchluesselStecken()* oder Operation *SchluesselimFahrzeug()*) berücksichtigt.

Wäre unter den Spezifikationen keine Spezifikation mit dem Attribute *Complete* enthalten, müsste auf Basis der aktuellen Parametrierung des Testablaufs nur der Ausführungspfad mit der Operation *SchluesselStecken()* betrachtet werden.

In der *ActionSequence* ist ein Interaktionsfragment vom Typ *for* enthalten. Die Schleife wird während der Testausführung zweimal durchlaufen. Da die Anzahl der Schleifendurchläufe aufgrund der angegebenen Schleifenbedingung vor der Ausführung des Testfalls am Prüfstand bestimmt werden kann, wird diese bei der Transformation in die Netzdarstellung berücksichtigt. In diesem Beispiel werden die Operationen *ACCEinstellungen()* und *ACCAktivieren()* zweimal hintereinander ausgeführt. Zur eindeutigen Unterscheidung der beiden Instanzen werden die Transitionen durch einen Index erweitert. Die entsprechenden Transitionen sind in Gelb visualisiert.

Die Netzdarstellung der TestSequence *PostSequence* enthält eine Verzweigung. Für das Interaktionsfragment vom Typ *alt* in dem EXAM-Element *PostSequence* kann die Bedingung erst zur Laufzeit ausgewertet werden. Somit ist bei der Transformation in die Netzdarstellung zu beachten, dass sowohl die Operation *FehlerspeicherLoeschen()* ausgeführt bzw. nicht ausgeführt werden kann. Die an der Alternative beteiligten Transitionen sind in der Netzdarstellung entsprechend grün markiert.

Die für die Verifikation des Testfalls *TestCaseExample* notwendige Kombination der einzelnen Netzdarstellungen ist in der Abbildung 7.14 auf Seite 151 dargestellt. In dieser Darstellung werden für die Namen der Transitionen nur die eingeführten Präfixe verwendet.

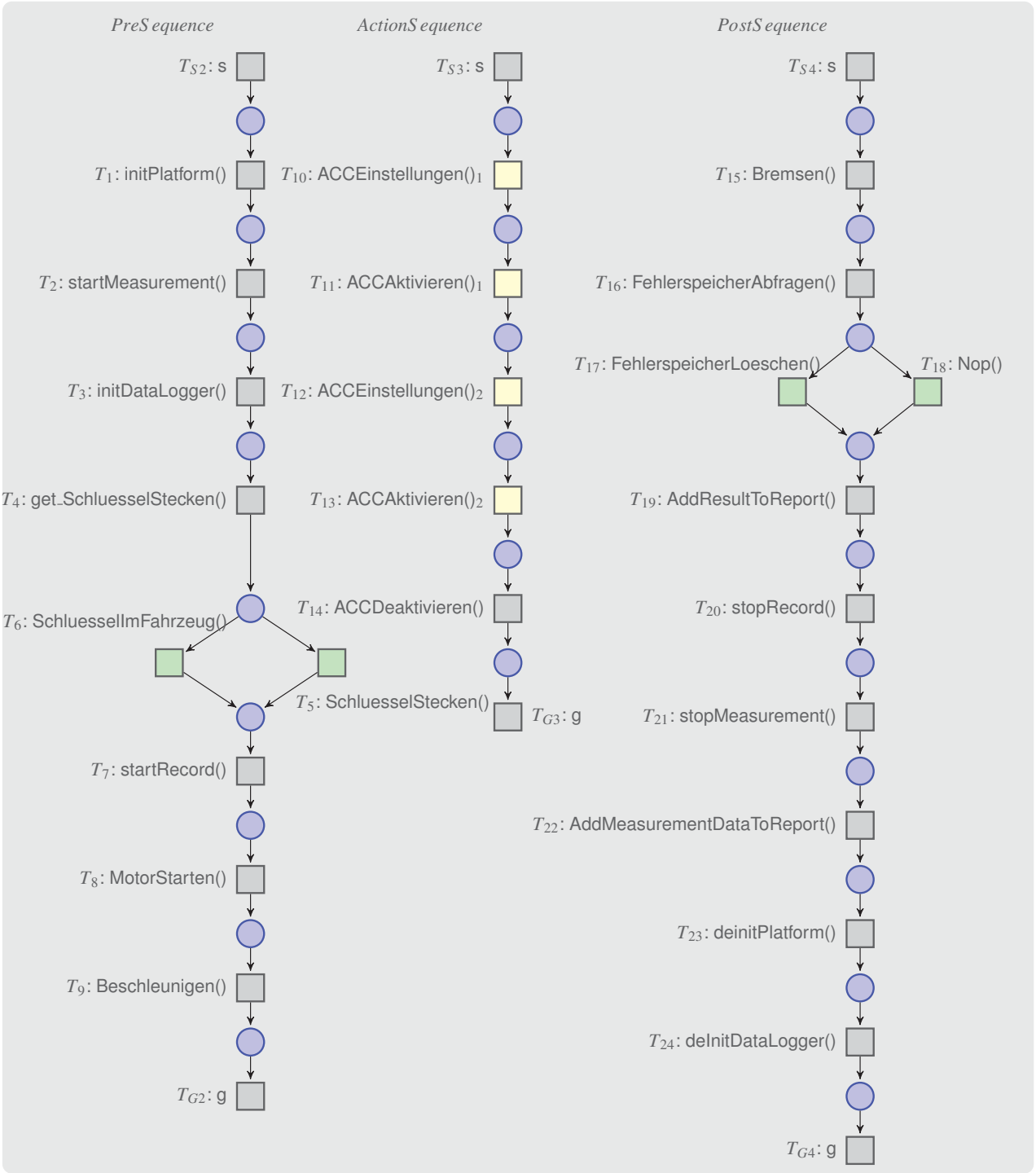

Abbildung 7.13: Netzdarstellungen der *TestSequence PreSequence, ActionSequence und PostSequence*

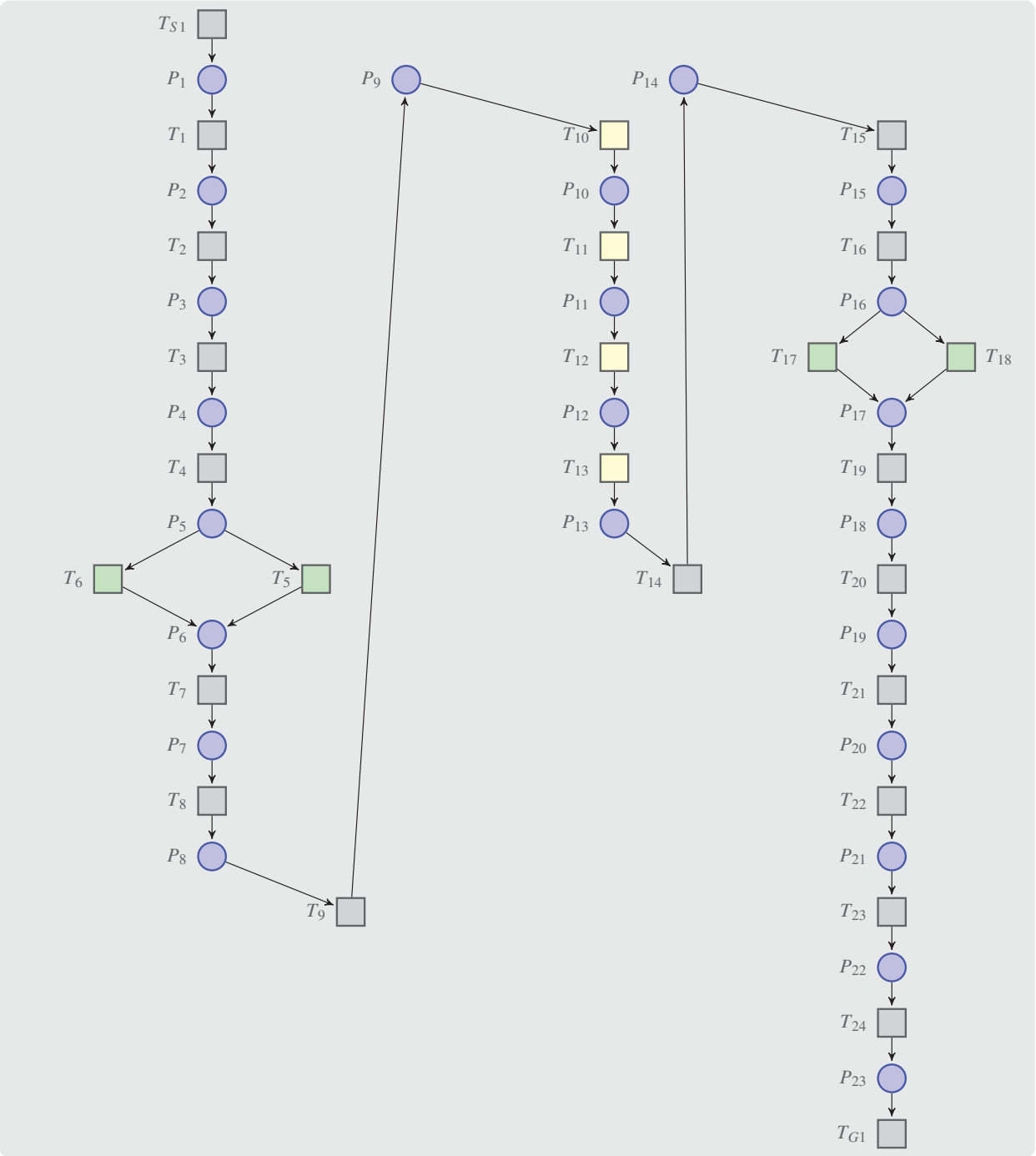

Abbildung 7.14: Kombinierte Netzdarstellung $N[TestCaseExample]$ des *TestCases TestCase-Example*

7.4 Durchführung der Verifikation

Für die Überprüfung, ob der Testfall *TestCaseExample* die angegebene Spezifikationen erfüllt, verwenden wir das in der Definition 5.24 auf Seite 78 beschriebene direkte Beweisverfahren. Das Beweisverfahren basiert darauf, die Anzahl der Prozesse der Realisierung ($|\mathbb{P}(R)|$) mit der Anzahl der Prozesse der Konjunktion der Realisierung und der Spezifikation ($|\mathbb{P}(R \owedge S)|$) zu vergleichen. Die Realisierung erfüllt dann die Spezifikation, wenn die Anzahl der Prozesse in den beiden Prozessmengen gleich ist ($|\mathbb{P}(R \wedge S)| = |\mathbb{P}(R)|$). Mit diesem Ansatz wird sichergestellt, dass die kausale Ordnung der Aktionen in der Realisierung der kausalen Ordnung der Aktionen in der Spezifikation nicht widerspricht. In Abhängigkeit der Attribute (*Voraussetzungs-* und *Häufigkeitsattribut*) der Spezifikation sind die im Kapitel 6 beschriebenen Algorithmen zu verwenden.

Für die Berechnung der Anzahl der Prozesse in den Prozessmengen verwenden wir die entsprechende Petri-Netz-Darstellung der Realisierung und der Spezifikation, wie im Abschnitt 5.4 beschrieben. Dabei ist Anzahl der Prozesse gleich der Anzahl der Schaltfolgen der zugehörigen Petri-Netz-Darstellung, welche die leere Markierung reproduzieren. Hierbei müssen die Transition *s* und *g* in jeder Schaltfolge genau einmal vorkommen.

Im ersten Schritt bestimmen wir die Anzahl der Prozesse der Realisierung. Dazu verwenden wir die sich aus der Netzdarstellung $N[TestCaseExample]$ (Abbildung 7.14 auf Seite 151) ergebende Inzidenzmatrix $[TestCaseExample]$:

	T_{S1}	$T1$	$T2$	$T3$	$T4$	$T5$	$T6$	$T7$	$T8$	$T9$	$T10$	$T11$	$T12$	$T13$	$T14$	$T15$	$T16$	$T17$	$T18$	$T19$	$T20$	$tT21$	$T22$	$T23$	$T24$	T_{G1}
$P1$	1	−1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$P2$	0	1	−1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$P3$	0	0	1	−1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$P4$	0	0	0	1	−1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$P5$	0	0	0	0	1	−1	−1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$P6$	0	0	0	0	0	1	1	−1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$P7$	0	0	0	0	0	0	0	1	−1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$P8$	0	0	0	0	0	0	0	0	1	−1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$P9$	0	0	0	0	0	0	0	0	0	1	−1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$P10$	0	0	0	0	0	0	0	0	0	0	1	−1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$P11$	0	0	0	0	0	0	0	0	0	0	0	1	−1	0	0	0	0	0	0	0	0	0	0	0	0	0
$P12$	0	0	0	0	0	0	0	0	0	0	0	0	1	−1	0	0	0	0	0	0	0	0	0	0	0	0
$P13$	0	0	0	0	0	0	0	0	0	0	0	0	0	1	−1	0	0	0	0	0	0	0	0	0	0	0
$P14$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	−1	0	0	0	0	0	0	0	0	0	0
$P15$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	−1	0	0	0	0	0	0	0	0	0
$P16$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	−1	−1	0	0	0	0	0	0	0
$P17$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	−1	0	0	0	0	0	0
$P18$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	−1	0	0	0	0	0
$P19$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	−1	0	0	0	0
$P20$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	−1	0	0	0
$P21$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	−1	0	0
$P22$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	−1	0
$P23$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	−1

(7.1)

Die Anzahl der Schaltfolgen, welche die leere Markierung reproduzieren, ergibt sich aus der Anzahl der realisierbaren T-Invarianten. Diese können auf Basis der Lösungen des homogenen Gleichungssystems $[TestCaseExample] * tr = 0$ bestimmt werden. In diesem Fall ergeben sich durch die Anwendung, des in Kapitel 4.3 vorgestellten Gauss-Jordan-Algorithmus die vier T-Invarianten tr_1,tr_2,tr_3 und tr_4.

$$tr_1 = \begin{pmatrix}1\\1\\1\\1\\1\\1\\0\\1\\1\\1\\1\\1\\1\\1\\1\\1\\1\\0\\1\\1\\1\\1\\1\\1\\1\\1\end{pmatrix} \quad tr_2 = \begin{pmatrix}1\\1\\1\\1\\1\\1\\0\\1\\1\\1\\1\\1\\1\\1\\1\\1\\1\\0\\1\\1\\1\\1\\1\\1\\1\\1\end{pmatrix} \quad tr_3 = \begin{pmatrix}1\\1\\1\\1\\1\\0\\1\\1\\1\\1\\1\\1\\1\\1\\1\\1\\0\\1\\1\\1\\1\\1\\1\\1\\1\end{pmatrix} \quad tr_4 = \begin{pmatrix}1\\1\\1\\1\\1\\0\\1\\1\\1\\1\\1\\1\\1\\1\\1\\1\\0\\1\\1\\1\\1\\1\\1\\1\\1\end{pmatrix} \tag{7.2}$$

Die vier T-Invarianten repräsentieren dabei die vier Schaltfolgen F_1, F_2, F_3 und F_4 mit

$$F_1 = (T_{S_1}, T_1, T_2, T_3, T_4, T_5, T_7, T_8, T_9, T_{10}, T_{11}, T_{12}, T_{13}, T_{14}, T_{15}, T_{16}, T_{18}, T_{19}, T_{20}, T_{21}, T_{22}, T_{23}, T_{24}, T_{G_1})$$

$$F_2 = (T_{S_1}, T_1, T_2, T_3, T_4, T_5, T_7, T_8, T_9, T_{10}, T_{11}, T_{12}, T_{13}, T_{14}, T_{15}, T_{16}, T_{18}, T_{19}, T_{20}, T_{21}, T_{22}, T_{23}, T_{24}, T_{G_1})$$

$$F_3 = (T_{S_1}, T_1, T_2, T_3, T_4, T_6, T_7, T_8, T_9, T_{10}, T_{11}, T_{12}, T_{13}, T_{14}, T_{15}, T_{16}, T_{18}, T_{19}, T_{20}, T_{21}, T_{22}, T_{23}, T_{24}, T_{G_1})$$

$$F_4 = (T_{S_1}, T_1, T_2, T_3, T_4, T_6, T_7, T_8, T_9, T_{10}, T_{11}, T_{12}, T_{13}, T_{14}, T_{15}, T_{16}, T_{17}, T_{19}, T_{20}, T_{21}, T_{22}, T_{23}, T_{24}, T_{G_1})$$

Als Ergebnis der Betrachtung ergibt sich, dass die Realisierung (Testfall *TestCaseExample*) vier Prozesse (Ausführungspfade) ($|\mathbb{P}(TestCaseEXAM)| = 4$) besitzt.

Nach der Berechnung der Anzahl der Prozesse für die Realisierung betrachten wir im Folgenden die Überprüfung jeder Spezifikation im Detail. In Abhängigkeit der Spezifikationsattribute müssen hier die im Kapitel 6 beschriebenen Verifikationsalgorithmen angewendet werden. Dazu ist im nächsten Schritt die Konjunktion der Netzdarstellung der Realisierung und der Netzdarstellung der Spezifikation zu erzeugen. Im Anschluss kann mit Hilfe der realisierbaren T-Invarianten die Anzahl der Prozesse der Module bestimmt werden.

Als erste Spezifikation betrachten wir die Spezifikation $S_1 = \{initDataLogger() \oslash startRecord()\}$. Die zugehörige Netzdarstellung ist in der Abbildung 7.1 auf Seite 141 dargestellt. Die Spezifikation ist vom Typ *May-Multiple*. Daher muss die Spezifikation nur überprüft werden, wenn die als *Required-Operation* gekennzeichneten Operationen der Spezifikation in dem Testablauf enthalten sind. Für die Spezifikation S_1 ist die *Required-Operation startRecord()* definiert. Da die Operation in der Realisierung enthalten ist, muss der Testablauf auf die Einhaltung der Spezifikation überprüft werden.

In der Abbildung 7.15 auf Seite 159 ist die Konjunktion der Netzdarstellung des Testfalls *TestCaseExample* und der Netzdarstellung der Spezifikation S_1 auf Basis des Algorithmus 5.1 dargestellt. Dabei sind die durch die Konjunktion entstandenen Stellen dunkelblau markiert.

Basierend auf der Netzdarstellung $N[TestCaseExample \oslash S_1]$ ergibt sich folgende Inzidenzmatrix $[TestCaseExample \oslash S_1]$:

$$
\begin{array}{c}
 \\
P1 \\ P2 \\ P3 \\ P4 \\ P5 \\ P6 \\ P7 \\ P8 \\ P9 \\ P10 \\ P11 \\ P12 \\ P13 \\ P14 \\ P15 \\ P16 \\ P17 \\ P18 \\ P19 \\ P20 \\ P21 \\ P22 \\ P23 \\ P24 \\ P25 \\ P26
\end{array}
\begin{array}{c}
\begin{array}{cccccccccccccccccccccccc}
T_{S1} & T1 & T2 & T3 & T4 & T5 & T6 & T7 & T8 & T9 & T10 & T11 & T12 & T13 & T14 & T15 & T16 & T17 & T18 & T19 & T20 & iT21 & T22 & T23 & T24 & T_{G1}
\end{array} \\
\left(\begin{array}{cccccccccccccccccccccccc}
1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \\
1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1
\end{array}\right)
\end{array}
\tag{7.3}
$$

Auch hier bestimmen wir die Anzahl der Schaltfolgen, welche die leere Markierung reproduzieren, auf Basis der Anzahl der realisierbaren T-Invarianten. Als Lösungen des homogenen Gleichungssystems $[TestCaseExample \oslash S_1] * trs1 = 0$ ergeben sich vier realisierbare T-Invarianten $trs1_1$, $trs1_2$, $trs1_3$ und $trs1_4$.

$$
trs_1 = \begin{pmatrix}
1\\1\\1\\1\\1\\1\\0\\1\\1\\1\\1\\1\\1\\1\\1\\0\\1\\1\\1\\1\\1\\1\\1\\1\\1\\1\\1\\1
\end{pmatrix}
\quad
trs_2 = \begin{pmatrix}
1\\1\\1\\1\\1\\1\\0\\1\\1\\1\\1\\1\\1\\1\\1\\0\\1\\1\\1\\1\\1\\1\\1\\1\\1\\1\\1\\1
\end{pmatrix}
\quad
trs_3 = \begin{pmatrix}
1\\1\\1\\1\\1\\0\\1\\1\\1\\1\\1\\1\\1\\1\\0\\1\\1\\1\\1\\1\\1\\1\\1\\1\\1\\1\\1\\1
\end{pmatrix}
\quad
trs_4 = \begin{pmatrix}
1\\1\\1\\1\\0\\1\\1\\1\\1\\1\\1\\1\\1\\1\\0\\1\\1\\1\\1\\1\\1\\1\\1\\1\\1\\1\\1\\1
\end{pmatrix}
\tag{7.4}
$$

Die vier T-Invarianten repräsentieren dabei die vier Schaltfolgen F_1, F_2, F_3 und F_4

$F_1 = (T_{S_1}, T_1, T_2, T_3, T_4, T_5, T_7, T_8, T_9, T_{10}, T_{11}, T_{12}, T_{13}, T_{14}, T_{15}, T_{16}, T_{18}, T_{19}, T_{20}, T_{21}, T_{22}, T_{23}, T_{24}, T_{G_1})$

$F_2 = (T_{S_1}, T_1, T_2, T_3, T_4, T_5, T_7, T_8, T_9, T_{10}, T_{11}, T_{12}, T_{13}, T_{14}, T_{15}, T_{16}, T_{18}, T_{19}, T_{20}, T_{21}, T_{22}, T_{23}, T_{24}, T_{G_1})$

$F_3 = (T_{S_1}, T_1, T_2, T_3, T_4, T_6, T_7, T_8, T_9, T_{10}, T_{11}, T_{12}, T_{13}, T_{14}, T_{15}, T_{16}, T_{18}, T_{19}, T_{20}, T_{21}, T_{22}, T_{23}, T_{24}, T_{G_1})$

$F_4 = (T_{S_1}, T_1, T_2, T_3, T_4, T_6, T_7, T_8, T_9, T_{10}, T_{11}, T_{12}, T_{13}, T_{14}, T_{15}, T_{16}, T_{17}, T_{19}, T_{20}, T_{21}, T_{22}, T_{23}, T_{24}, T_{G_1})$

Somit besitzt das AL-Modul $TestCaseExample \oslash S_1$ vier Prozesse (Ausführungspfade)

$$|\mathbb{P}(TestCaseExample \oslash S_1)| = 4.$$

Nach dem *direkten Beweisverfahren* (vergleiche Definition 5.24 auf Seite 78) erfüllt somit der Testfall die Spezifikation S_1, da

$$|\mathbb{P}(TestCaseExample \oslash S_1)| = 4 = |\mathbb{P}(TestCaseExample)|$$

ist.

Die zweite zu überprüfende Spezifikation $S_2 = \{initPlatform() \circledcirc deInitPlatform()\}$ ist vom Typ *Must-Single*. Die Netzdarstellung der Spezifikation ist in der Abbildung 7.2 auf Seite 142 dargestellt.

Nach dem Algorithmus 6.1 auf Seite 117 ist bei einer Spezifikation mit dem Attribute *Single* im ersten Schritt zu prüfen, ob die Operationen der Operationsmenge der Spezifikation mehr als einmal in der Realisierung vorkommen. In diesem Fall ist sowohl die Operation *initPlatform()* (dargestellt durch die Transition T_1) als auch die Operation *deInitPlatform()* (dargestellt durch die Transition T_{22}) genau einmal in dem Testfall *TestCaseExample* enthalten. Somit ist die erste Bedingung des Attributes *Single* erfüllt und die Spezifikation kann nun mit Hilfe des direkten Beweisverfahrens überprüft werden.

Betrachten wir zur Bestimmung der Anzahl der Schaltfolgen die Netzdarstellung der Konjunktion der Realisierung und der Spezifikation S_2 in der Abbildung 7.16 auf der Seite 160. Die Berechnung der Schaltfolgen, welche die leere Markierung reproduzieren und in denen die Transition T_{S_1} und T_{G_1} nur einmal schalten, erfolgt analog zu dem ersten Beispiel mit Hilfe der Inzidenzmatrix und der Lösung des entsprechenden linearen Gleichungssystems. An dieser Stelle verzichten wir auf die Aufstellung der Inzidenzmatrix und bestimmen die Anzahl der Schaltfolgen direkt aus der Netzdarstellung.

Für die Netzdarstellung $N[TestCaseExample \oslash S_2]$ ergeben sich vier Schaltfolgen, die die leere Markierung reproduzieren und somit ist

$$|\mathbb{P}(TestCaseExample \oslash S_2)| = 4.$$

Somit erfüllt der Testfall nach dem direkten Beweisverfahren die Spezifikation S_2, da

$$|\mathbb{P}(TestCaseExample \oslash S_2)| = 4 = |\mathbb{P}(TestCaseExample)|$$

ist.

Die dritte Spezifikation

$$S_3 = \{(SchluesselStecken() \oplus SchluesselImFahrzeug()) \otimes MotorStarten()\}$$

vom Typ *May-Multiple* gibt an, dass vor dem Starten des Motors der Schlüssel im Zündschloss stecken oder im Fahrzeuginnenraum positioniert sein muss. Aufgrund des Attributes *May* ist nach der Definition 6.5 im ersten Schritt zu überprüfen, ob die in der Spezifikation als *Required-Operation* gekennzeichneten Operationen in dem Testfall enthalten sind. In diesem Fall ist die als *Required-Operation* gekennzeichnete Operation *MotorStarten()* in dem Testfall enthalten und somit muss die Realisierung auf die Einhaltung der definierten kausalen Abhängigkeiten in der Spezifikation S_3 überprüft werden. Die für die Überprüfung notwendige Kombination der beiden Netzdarstellungen ist in Abbildung 7.17 auf Seite 161 dargestellt.

Für die Verifikation gehen wir analog zu den ersten Beispielen vor und berechnen mit Hilfe der Inzidenzmatrix die Anzahl der realisierbaren T-Invarianten und können somit auf die Anzahl der Prozesse schließen. Die Anzahl der Schaltfolgen des Netzes $N[TestCaseExample \oslash S_3]$, welche die leere Markierung reproduzieren, ist wie aus der Abbildung 7.17 ersichtlich, gleich vier.

Somit erfüllt der Testfall nach dem direkten Beweisverfahren die Spezifikation S_3, da

$$|\mathbb{P}([TestCaseExample] \oslash S_3)| = 4 = |\mathbb{P}([TestCaseExample])|$$

ist.

Wie bereits beschrieben, soll mit der Spezifikation $S_4 = \{startMeasurement() \oslash startRecord()\}$ überprüft werden, ob in dem Testablauf sowohl die Simulationsgrößen in den Echtzeitmodellen als auch die Buskommunikation am Prüfstand aufgezeichnet werden. Die Attribute der Spezifikation (*Must-Multiple*) geben an, dass zum einen die Spezifikation immer erfüllt sein muss, wenn mindestens eine Operation der Spezifikation in der Realisierung enthalten ist und zum anderen, dass die beiden Operationsaufrufe beliebig oft in dem Testablauf enthalten sein dürfen.

Bei einer Spezifikation mit dem Attribute *Multiple* ist nach der Beschreibung im Kapitel 6.4 im ersten Schritt zu prüfen, ob mehrere Instanzen der spezifizierten Operationen in der Realisierung enthalten sind. In diesem Beispiel sind in dem Testfall beide Operationen nur jeweils einmal vorhanden. Somit kann die Überprüfung direkt mit Hilfe des direkten Beweisverfahrens durchgeführt werden.

Die Kombination der Netzdarstellungen ($[TestCaseExample \oslash S_4]$) der Realisierung und der Spezifikation S_4 ist in der Abbildung 7.18 auf Seite 162 dargestellt. Wir berechnen wieder auf Basis der Inzidenzmatrix die Lösungen des homogenen Gleichungssystems

$$[TestCaseExample \oslash S_4] * tr = 0.$$

Es ergeben sich aus der Lösungsmenge des Gleichungssystems vier Schaltfolgen, welche die leere Markierung reproduzieren. Nach dem direkten Beweisverfahren erfüllt der Testfall die Spezifikation S_4 somit, da

$$|\mathbb{P}([TestCaseExample] \oslash S_4)| = 4 = |\mathbb{P}([TestCaseExample])|$$

ist.

Die Spezifikation $S_5 = \{ACCAktivieren() \circledcirc ACCDeaktivieren())\}$ beschreibt, dass die Operation *ACCAktivieren()* immer vor der Operation *ACCDeaktivieren()* ausgeführt werden muss. Dabei darf zwischen den beiden Aufrufen keine weitere Instanz der beiden Operationen im Testablauf aufgerufen werden. Dies ist durch das Attribut *One-To-One* der Spezifikation festgelegt. Das Attribut *May* beschreibt, dass der Testfall auf Einhaltung der Spezifikation nur dann überprüft werden muss, falls die als notwendig gekennzeichneten Operationen der Spezifikation (*Required-Operation*) in der Realisierung enthalten sind. In diesem Fall sind beide als notwendig gekennzeichneten Operationen in dem Testfall enthalten.

Nach dem Algorithmus 6.5 auf Seite 129 muss bei einer Spezifikation vom Typ *One-To-One* im ersten Schritt die notwendige, aber nicht hinreichende Bedingung überprüft werden. Diese besagt, dass die erweiterte Netzdarstellung der Realisierung (siehe Algorithmus 6.4 auf Seite 128) mindestens eine Schaltfolge enthält, welche die leere Markierung reproduziert.

Für die Überprüfung dieser Bedingung ist in Abbildung 7.19 auf Seite 163 die erweiterte Netzdarstellung des Testfalls *TestCaseExample* dargestellt. Als zusätzliches Element ist in der Netzdarstellung die Stelle *P*24 hinzugefügt worden. Die Stelle ist mit jeder Instanz der Operationen *ACCAktivieren()* und *ACCDeaktivieren()* nach der Vorgabe in der erweiterten Netzdarstellung der Spezifikation (siehe Abbildung 7.5 auf Seite 143) verbunden. Die Anzahl der Schaltfolgen kann nun auf Basis der Inzidenzmatrix berechnet werden. Dabei ergibt sich, dass aufgrund der Struktur des Petri-Netzes die leere Markierung durch keine Schaltfolge reproduziert werden kann. Nach dem Schalten der Transition T_{11} befindet sich kein Token mehr auf der Stelle P_{24}. Daher kann die Transition T_{13} nicht schalten, ohne das die Transition T_{S_1} zweimal schaltet.

Nach dem Algorithmus 6.4 auf Seite 128 ist die notwendige Bedingung nicht erfüllt und somit kann der Testfall die Spezifikation nicht erfüllen. Dies entspricht auch dem erwarteten Ergebnis, da innerhalb der Schleife die Operation *ACCAktivieren()* zweimal hintereinander aufgerufen wird. Nach der Spezifikation hat jedoch nach jeder Aktivierung der ACC-Funktion eine Deaktivierung der Funktion zu erfolgen, bevor die Funktion erneut wieder aktiviert werden darf.

Die letzte Spezifikation (S_6) in diesem Anwendungsbeispiel ist vom Typ *Complete-Multiple* und beschreibt, dass in der Struktur des Testfalls in mindestens einem Ausführungspfad die Operation *SchluesselStecken()* und in einem zweiten Ausführungspfad die Operation *SchluesselImFahrzeug()* enthalten sein muss. Nach der Definition 5.22 auf Seite 76 ist dies die Frage, ob die Realisierung im Bezug auf die Spezifikation *vollständig* ist.

Wie in Kapitel 6.4.2 an einem ähnlichen Beispiel bereits im Detail gezeigt, erfüllt der Testfall dann die Spezifikation (der Testfall ist im Bezug auf die Spezifikation *vollständig*), wenn die in der Definition 5.24 auf Seite 78 angegebene Bedingung $|\mathbb{P}(S \wedge R)| = |\mathbb{P}(S)|$ erfüllt ist.

Wie aus der Abbildung 7.14 auf Seite 151 ersichtlich ist, sind beide Ausführungspfade in dem Testfall enthalten. Somit ist der Testfall in Bezug auf die Spezifikation vollständig.

Da wir bereits im Kapitel 6.4.2 die Überprüfung der Spezifikation im Detail vorgestellt haben und aufgrund der sich durch die Konjunktion der beiden Netzdarstellungen ergebenen komplexen Netzstruktur, wird an dieser Stelle auf die Darstellung des Petri-Netzes und auf die Berechnung der Schaltfolgen verzichtet.

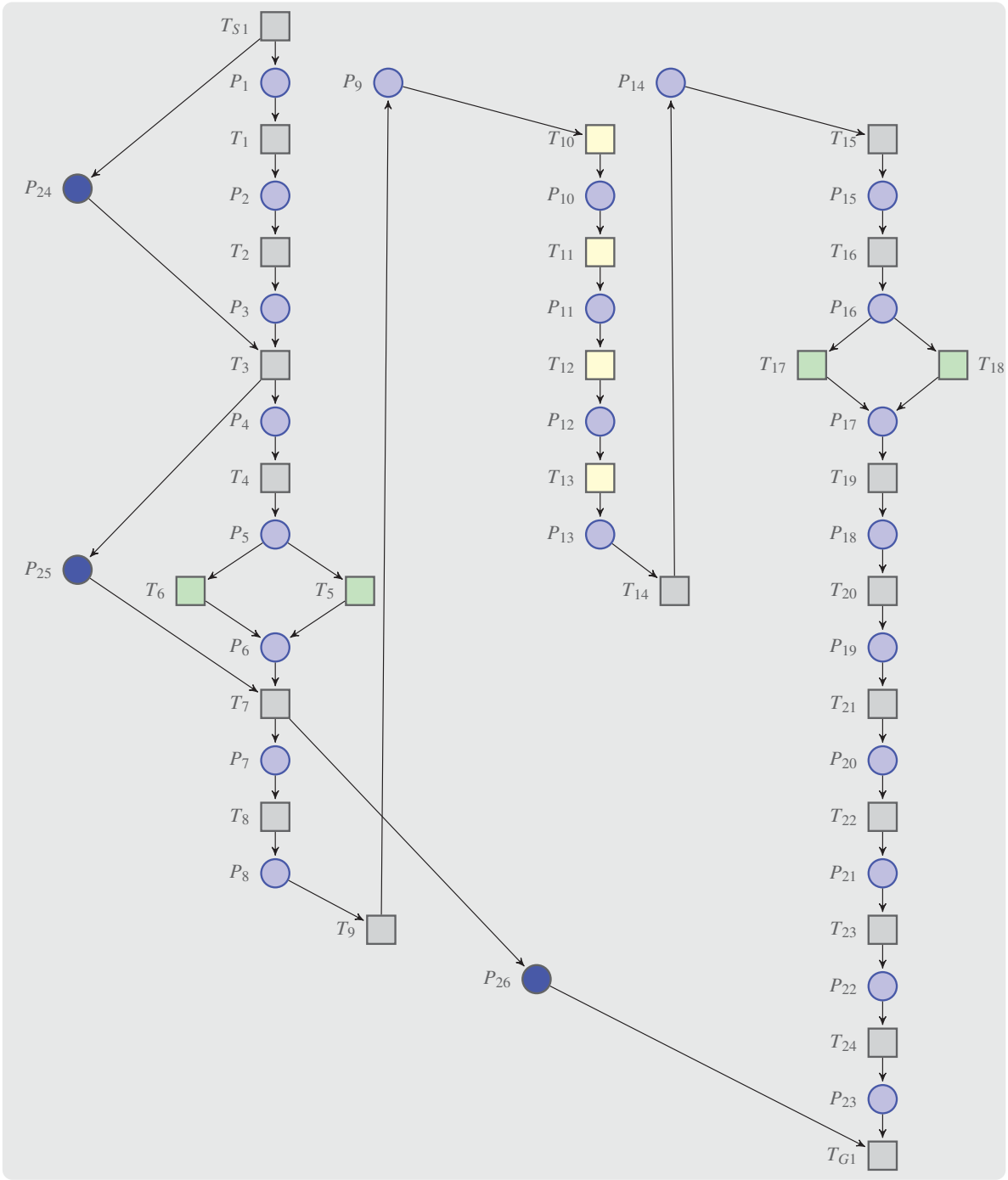

Abbildung 7.15: Netzdarstellung von $[TestCaseExample \oslash S_1]$

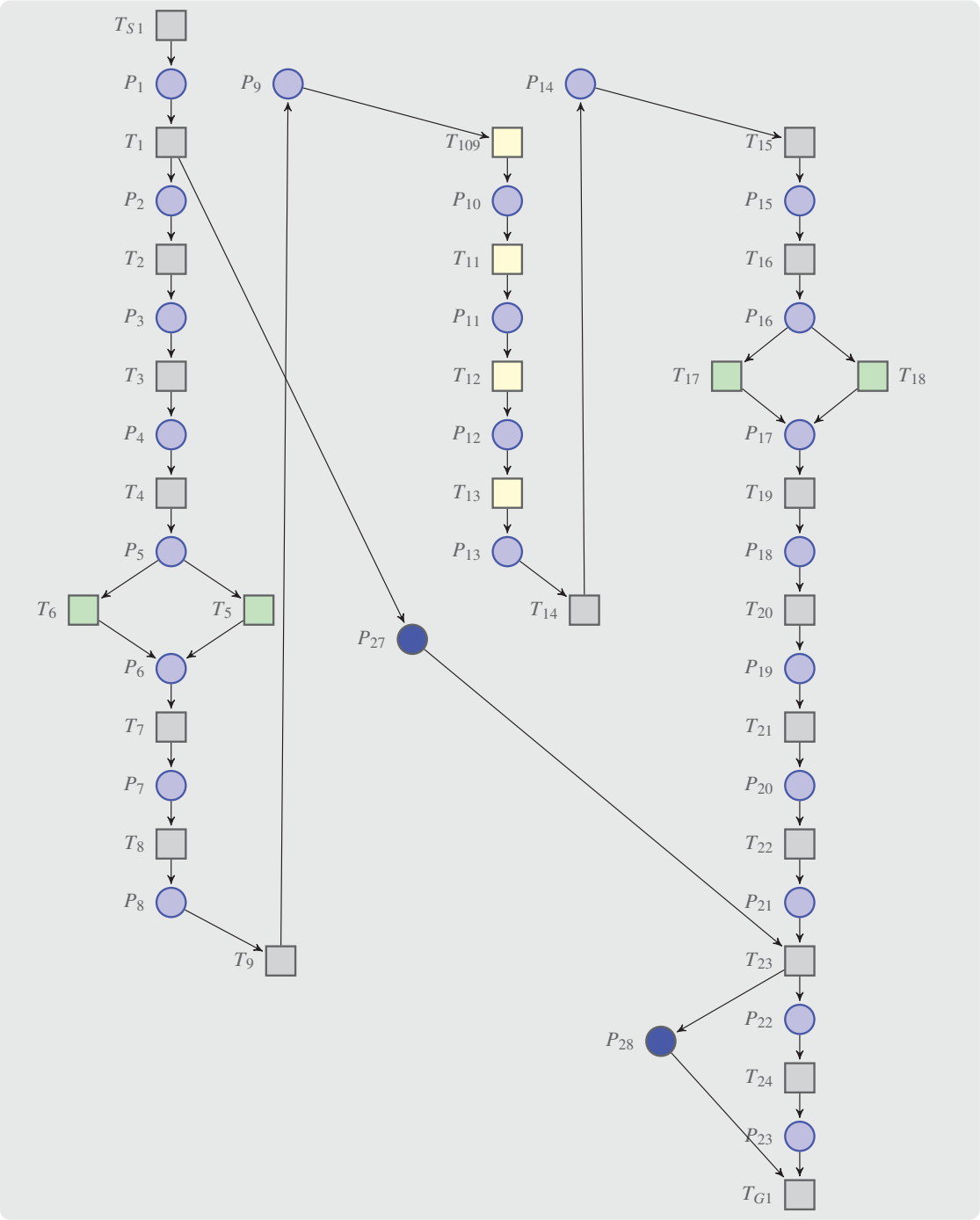

Abbildung 7.16: Netzdarstellung von $[TestCaseExample \oslash S_2]$

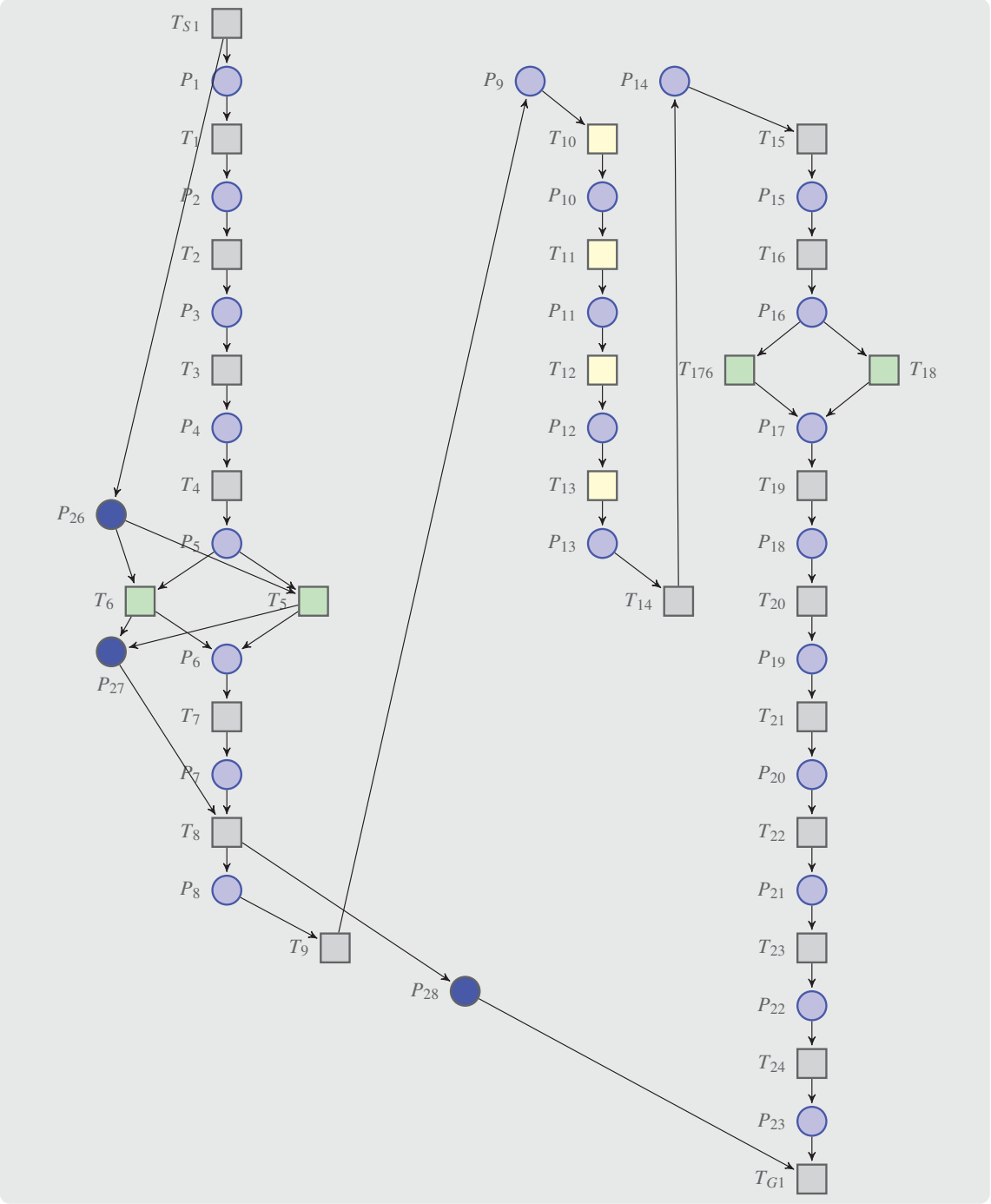

Abbildung 7.17: Netzdarstellung von $[TestCaseExample \oslash S_3]$

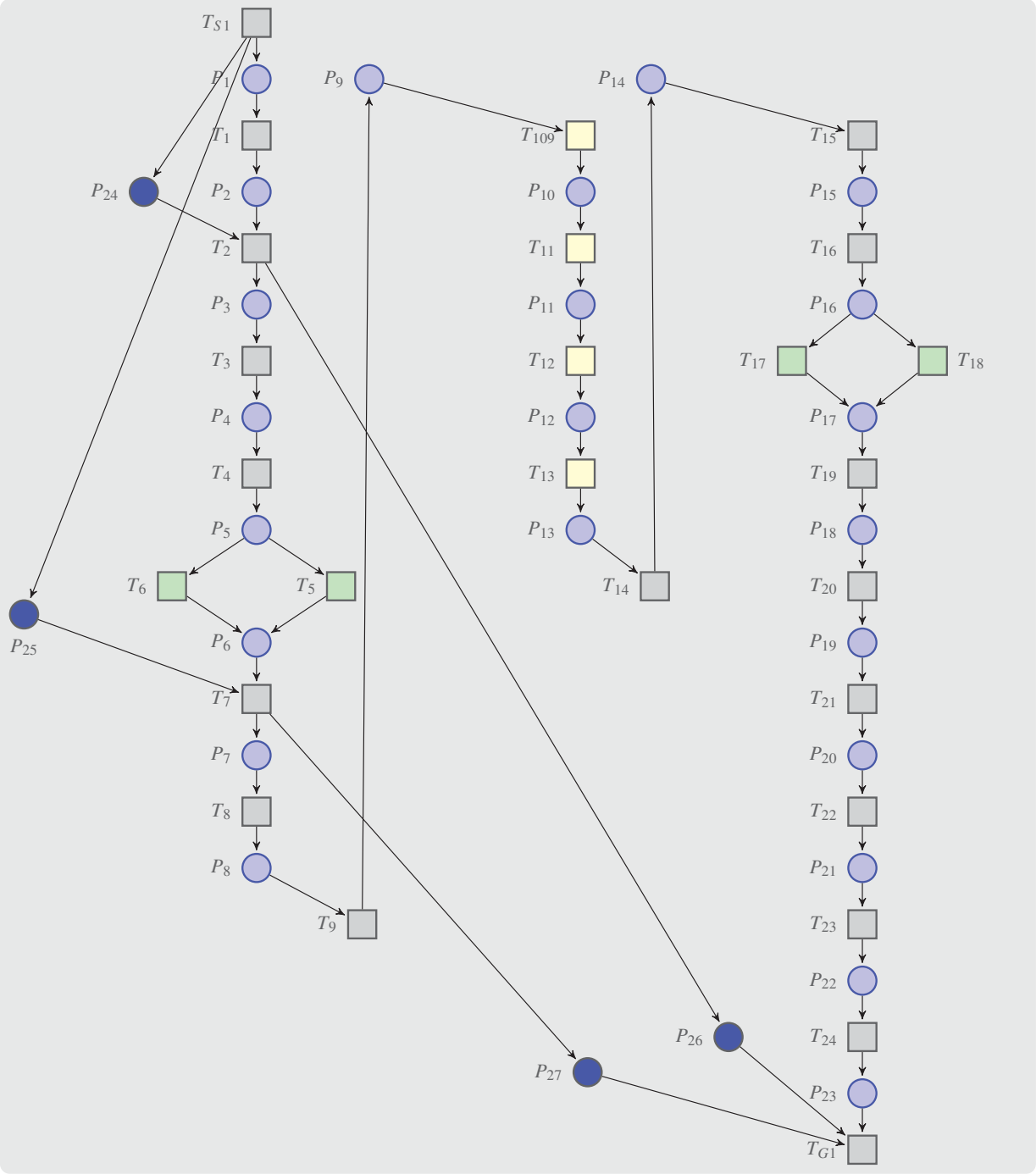

Abbildung 7.18: Netzdarstellung von $[TestCaseExample \oslash S_4]$

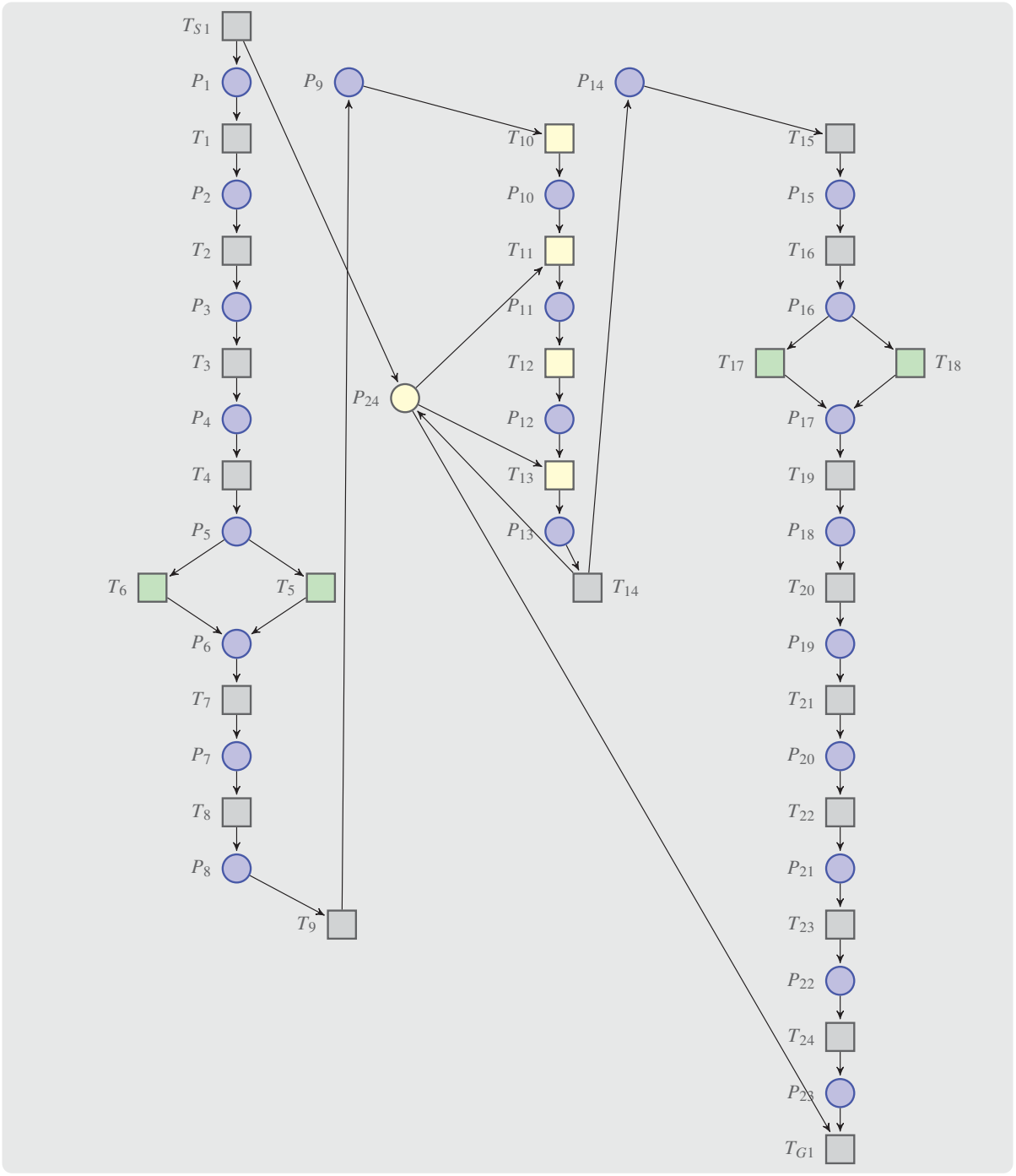

Abbildung 7.19: Netzdarstellung von $[TestCaseExample \oslash S_5]$

8 Praktischer Einsatz

Nach der Vorstellung der Methode zur Überprüfung der kausalen Abhängigkeiten in EXAM in Kapitel 6 und der Darstellung eines ausgewählten Anwendungsbeispiels in Kapitel 7 beschreiben wir in diesem Kapitel, wie die Methode in der Praxis für die Erhöhung der Qualität der EXAM-Testfälle eingesetzt werden kann.

Als Grundlage zur Validierung der Einsetzbarkeit der Methode wurde ein eigenständiges Eclipse RCP Plugin als Erweiterung für den EXAM-Modeller prototypisch umgesetzt und mit Hilfe von Key-Usern im Testbetrieb bei der AUDI AG erprobt.

Wir beschreiben im ersten Teil dieses Kapitels wie mit Hilfe der Erweiterungen von EXAM die kausalen Abhängigkeiten zwischen den Operationen der Funktionsbibliothek in EXAM definiert werden können und wie die Anwender bei der Erstellung der Spezifikationen unterstützt werden. Im folgenden Abschnitt betrachten wir die benötigten Ausführungszeiten der Verifikationsmethode eingehend und entwickeln ein Reduktionsverfahren zur Optimierung der benötigten Lösungszeit. Im letzten Teil des Kapitels stellen wir vor, wie die gefundenen Verletzungen der kausalen Abhängigkeiten der Operationen in den Testabläufen dem Anwender dargestellt werden und wie dieser bei der Fehlerbehebung unterstützt wird.

8.1 Definition der Spezifikationen in EXAM

Bevor wir mit Hilfe der im Kapitel 6 vorgestellten Algorithmen die kausalen Abhängigkeiten zwischen den EXAM-Operationen in den Testabläufen überprüfen können, müssen die Abhängigkeiten mit Hilfe von Spezifikationen (AL-Module) formal definiert werden. Im Folgenden beschreiben wir, wie die Erweiterung des EXAM-Modeller den Anwender bei der Erstellung der Spezifikationen und der Zuordnung der Spezifikationen zu den zu überprüfenden EXAM-Elementen unterstützt.

Für die Überprüfung der EXAM-Elemente auf Einhaltung der kausalen Abhängigkeiten zwischen den einzelnen Operationen müssen den zu überprüfenden Elementen in EXAM eine oder mehrere Spezifikationen zugeordnet werden (vgl. Abbildung 6.1 auf Seite 92). In EXAM kann eine Spezifikation mehreren EXAM-Elementen zugeordnet werden. Damit die Wiederverwendbarkeit einer einzelnen Spezifikation sichergestellt werden kann, unterscheiden wir zum einen die Erstellung der Spezifikation und zum anderen die Zuordnung der Spezifikation zu einem EXAM-Element. Im Folgenden beschreiben wir die dazu notwendigen neuen Elemente von EXAM und zeigen wie der Anwender bei der Erstellung und der Zuweisung der Spezifikationen zu den Testabläufen unterstützt wird.

Die kausalen Abhängigkeiten zwischen den Operationen werden im EXAM-Modell mit Hilfe des neuen EXAM-Elements *Specification-Object* beschrieben. Dieses Objekt speichert zum einen die kausalen Abhängigkeiten zwischen den Operationen in Form eines AL-Moduls und zum anderen die Art der Spezifikation (vgl. Abschnitt 6.4), also welches *Häufigkeits-* und *Voraussetzungsattribut* der Spezifikation zugeordnet ist sowie zusätzliche Informationen wie z.B. die *Required-Operation*.

Bei der Erstellung der Spezifikationen wird der Anwender durch einen neuen Editor als Erweiterung der bestehenden EXAM-Benutzeroberfläche unterstützt. Der Editor erlaubt dem Anwender die jeweiligen EXAM-Operationen durch einen Operator der Aktionslogik ($\oslash, \ominus, \oplus$) zu verbinden und bei Bedarf die entsprechenden *Required-Operation* zu definieren. Alternativ ist eine grafische Beschreibung des AL-Moduls mit Hilfe eines Aktivitätsdiagramms möglich, welches dem entsprechenden *Specification-Object* zugeordnet wird. Die Verwendung eines Aktivitätsdiagramms anstelle eines Petri-Netzes liegt zum einen darin begründet, dass die Anwender mit der Modellierung von Aktivitätsdiagrammen vertraut sind und sich somit nicht an einen neuen Diagrammtyp gewöhnen müssen. Zum anderen würde die Einführung eines neuen Diagrammtyps zu einer Erweiterung der Datenbankschemata zur Speicherung der EXAM-Modelle führen. Wie für die Transformation der Testabläufe kann an dieser Stelle die im Abschnitt 6.3.2 beschriebene Transformation der Aktivitätsdiagramme in eine entsprechende Petri-Netz-Darstellung auch für die Transformation der Spezifikationen verwendet werden.

Für die Erstellung der Spezifikationen wird in EXAM die neue Rolle des *Qualitätsbeauftragten* eingeführt. Ergibt sich die Spezifikation aus den Abhängigkeiten von Operationen der Funktionsbibliothek wird die Rolle durch den Verantwortlichen der Funktionsbibliothek besetzt. Parallel zur Erstellung neuer Operationen erstellt dieser die entsprechenden Spezifikationen. Basiert die Spezifikation auf Abhängigkeiten, die sich durch die Modellierungsrichtlinien ergeben, wird die Rolle von dem *Modellierungskoordinator* oder dem jeweiligen *Modellverantwortlichen* besetzt. Ebenso kann jeder EXAM-Anwender eigene benutzerspezifische Spezifikationen erstellen.

Neben der Erstellung einzelner Spezifikationen besteht der Bedarf, die Spezifikationen zu gruppieren. Für eine logische Gruppierung der Spezifikationen wird das EXAM-Element *SpecificationGroup* verwendet. Dies ermöglicht, z. B. alle Spezifikationen für eine Anwendergruppe oder für ein Fahrzeugprojekt in einer logischen Gruppe zu speichern. Ziel sollte es sein, dass neben der Erstellung der einzelnen Spezifikationen der *Qualitätsbeauftragte* diese zu entsprechenden Gruppen zusammenfasst.

Nach der Erstellung und der Gruppierung der Spezifikationen zu *SpecificationGroups* müssen die Spezifikationen für die Überprüfung den entsprechenden EXAM-Elementen zugeordnet werden. Die Zuordnung erfolgt dabei über das Element *SpecificationConfiguration*. In einer *SpecificationConfiguration* werden jedem zu überprüfenden EXAM-Element (*TestSuite*, *TestSequence*, *TestCase*, *Package*) eine oder mehrere Bedingungen (*Specifications*) zugeordnet. Bei der Zuordnung der Spezifikationen zu einem Ordner (*Package*) wird die Spezifikation allen EXAM-Elementen innerhalb des Ordners inklusive aller Unterordner zugewiesen.

Der Zusammenhang zwischen den beteiligten Elementen ist in der Abbildung 8.1 dargestellt. In dem Klassendiagramm ist dargestellt, dass in einer *SpecificationGroup* beliebig viele Spezifikationen (*Specifications*) enthalten sein können. Eine *SpecificationConfiguration* ordnet eine oder

mehrere *SpecificationGroups* einem oder mehreren EXAM-Elementen (*Package*, *TestSequence*, *TestActivity*, *TestCase* und *TestSuite*) zu.

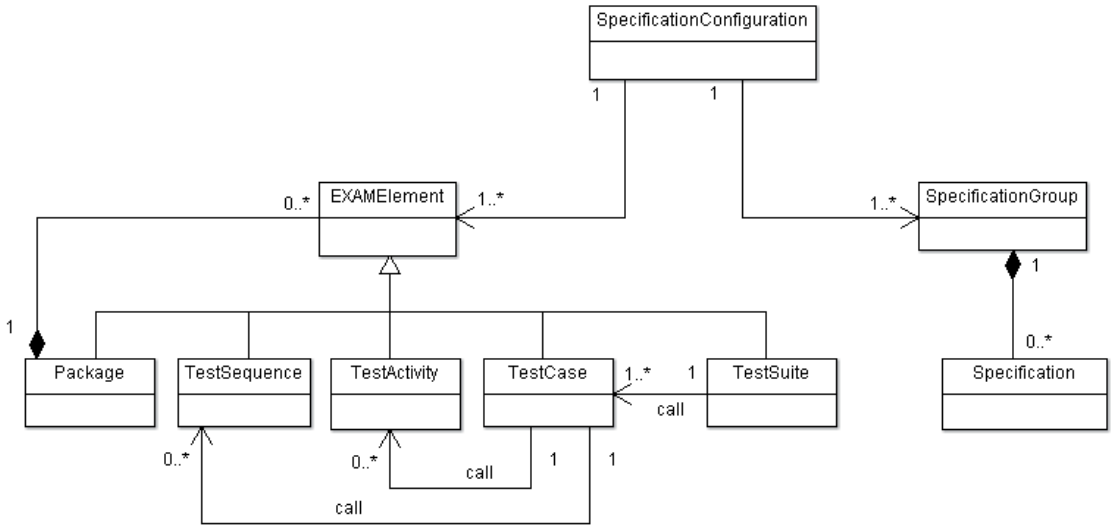

Abbildung 8.1: Zuordnung der Spezifikationen zu den EXAM-Objekten

Zur Konfiguration einer *SpecificationConfiguration* wird der Anwender in EXAM durch den *Specification-Editor* unterstützt. In Abbildung 8.2 ist als Beispiel der *Specification-Editor* für die *SpecificationConfiguration SampleSpecificationConfiguration* dargestellt.

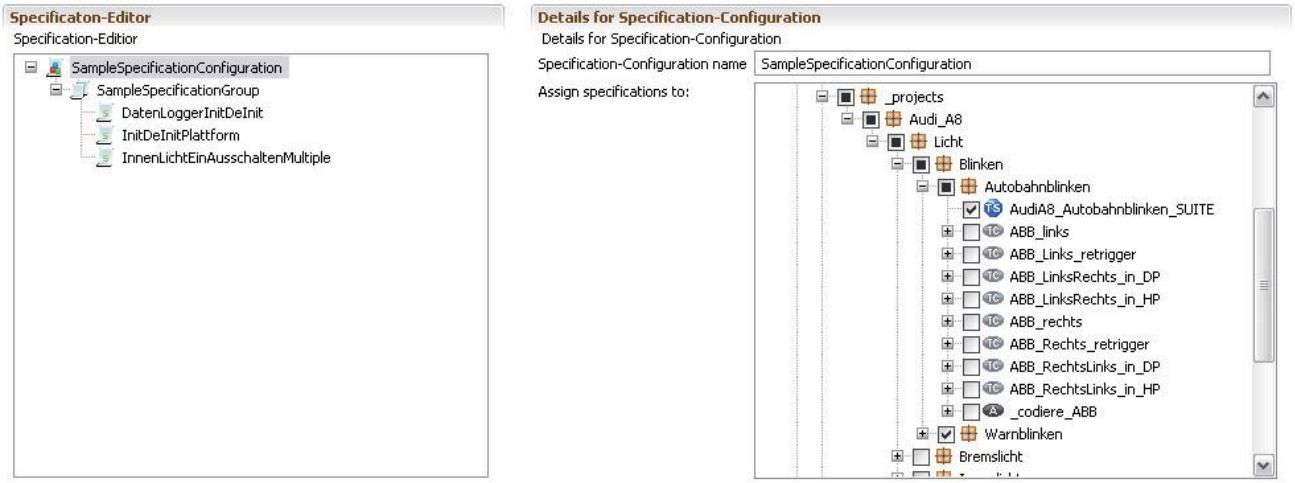

Abbildung 8.2: Beispiel des SpecificationEditor im EXAM-Modeller

Der Editor ist dabei in zwei Bereiche unterteilt. Im linken Bereich ist angegeben, welche *SpecificationGroups* den im rechten Bereich ausgewählten EXAM-Elementen zugeordnet sind. In diesem Beispiel gibt es eine *SpecificationGroup* (*SampleSpecificationGroup*) die drei Spezifikationen (*DatenLoggerInitDeInit*, *InitDeInitPlattform*, *InnenLichtEinAusschaltenMultiple*) enthält. Im rechten Bereich des Editors ist ein Auszug aus einem EXAM-Modell für den Bereich der Komfortelektronik in Form einer Baumstruktur dargestellt. In diesem Beispiel erfolgt die Zuord-

nung der drei Spezifikationen zum einen zu der *TestSuite AudiA8_Autobahnblinken_SUITE* und zum anderen zu allen EXAM-Elementen im Ordner *Warnblinken*.

8.2 Notwendige Optimierung

Für den praktischen Einsatz der Methode und der im Kapitel 6 vorgestellten Algorithmen ist die benötigte Ausführungszeit zur Berechnung der Ergebnisse von Bedeutung. Die Ergebnisse der Überprüfung müssen dem Anwender in akzeptabler Zeit zur Verfügung stehen. Aus Sicht des Anwenders ist die benötigte Zeit, die für die Berechnung des Ergebnisses benötigt wird dann akzeptabel, wenn diese Zeit kleiner ist als die benötigte Testausführungszeit am Prüfstand.

In diesem Abschnitt betrachten wir im ersten Schritt die Auswirkungen der Komplexität der EXAM-Elemente auf die Ausführungsdauer der Verifikationsalgorithmen. Im zweiten Schritt wird ein Verfahren zur Reduktion der Stellen und Transitionen in den Petri-Netz-Darstellungen der EXAM-Elemente beschrieben. Die Betrachtung der Auswirkungen des vorgestellten Reduktionsverfahrens auf die benötigte Ausführungszeit und ein Erfahrungsbericht mit der praktischen Anwendung der Methode bei der AUDI AG schließen den Abschnitt ab.

8.2.1 Komplexität der Testabläufe

Im ersten Teil des Abschnitts beschreiben wir zwei Metriken zur Beschreibung der Komplexität der EXAM-Elemente und zeigen im zweiten Teil des Abschnittes, welchen Einfluss die Komplexität auf die benötigte Ausführungszeit der von uns entwickelten Methode hat. Die dafür ermittelten statistischen Werte basieren auf Daten der EXAM-Modelle im Volkswagen-Konzern von November 2010.

In der Softwaretechnik gibt es eine Reihe von Metriken für die Bestimmung der Komplexität eines Programms. Eine detaillierte Beschreibung verschiedener Softwaremetriken geben z. B. [Sch07], [SL05] oder [Lig09]. Für die Bewertung der Komplexität der einzelnen EXAM-Elemente übertragen wir zwei aus der Softwaretechnik bekannte Metriken, die Anzahl der Programmzeilen (engl. Lines of Code) und zum anderen die von Thomas J. McCabe im Jahr 1976 eingeführte zyklomatische Komplexität (auch als *McCabe-Metrik* bezeichnet, vergleiche [McC76]) auf die Testabläufe in EXAM.

Die Anzahl der Programmzeilen (Lines of Code) gibt an, aus wie vielen einzelnen Zeilen ein Programm besteht. In der Literatur gibt es unterschiedliche Definitionen dieser Metrik. In einigen Definitionen werden nur die Zeilen mit Operationsaufrufen als Programmzeilen berücksichtigt. In anderen Definitionen werden zusätzlich dazu auch Zeilen mit Kommentaren und Leerzeilen bei der Erstellung der Metrik berücksichtigt. Bei der Übertragung auf EXAM betrachten wir jeden *TestStep* bzw. jede *TestAction* als eigene Programmzeile.

Die zyklomatische Komplexität gibt die Anzahl der Ablaufmöglichkeiten in einem Programm an. McCabe geht von der Annahme aus, dass ein Programm ab einer bestimmten Komplexität (Anzahl der theoretisch möglichen Ausführungspfade) für den Anwender schwerer zu verstehen

ist. Mit Hilfe der Metrik wird jedem Programm eine natürliche Zahl (zyklomatische Komplexität) zugeordnet. Die Abbildung der Komplexität eines Programms auf eine natürliche Zahl ermöglicht es, die Komplexität verschiedener Programme zu vergleichen.

Nach der Definition der zyklomatischen Komplexität in [Sch07] berechnet sich die *McCabe-Metrik* wie folgt:

Definition 8.1 (Zyklomatische Komplexität)
Die zyklomatische Komplexität eines beliebigen Programms ergibt sich wie folgt:
Zyklomatische Komplexität =
+ Anzahl der Verzweigungen (if)
+ Anzahl der Schleifen (for, while, repeat, usw.)
+ je Switch-Anweisung Anzahl der Zweige
+ 1 (für den Hauptprogrammpfad)

Betrachten wir nun, wie die beiden Metriken zur Bestimmung der Komplexität der Testabläufe nach EXAM übertragen werden können. Dabei ist zu beachten, dass die Struktur der Testabläufe in EXAM abhängig von der jeweiligen Domäne (Komfort, Antrieb, Fahrwerk und Infotainment) und der Art des Prüfstands (Komponentenprüfstand oder Systemprüfstand) ist. Für eine unabhängige Betrachtung ist für jede Metrik jeweils ein Durchschnittswert der Modelle und der jeweils maximale Wert in den Modellen angegeben.

Im ersten Schritt betrachten wir nur die Testabläufe ohne die Berücksichtigung einer *SystemConfiguration*. Die Betrachtung der *SystemConfiguration* diskutieren wir im Anschluss anhand eines Beispiels. Diese Unterscheidung haben wir getroffen, da für eine Reihe von Überprüfungen die Testabläufe unabhängig von der gewählten Systemkonfiguration betrachtet werden können. Viele Beispiele finden sich dabei in den durch die Modellierungsrichtlinien vorgegebenen Spezifikationen.

Beginnen wir die Betrachtung mit der Anzahl der einzelnen Operationsaufrufe in einen Testablauf. Die Anzahl der Programmzeilen entspricht in EXAM der Anzahl aller in einem Testablauf verwendeten *TestSteps* bzw. *TestAction*. Hierbei spielt es keine Rolle, ob der einzelne Testschritt einen *OperationCall* oder einen *UseCaseCall* enthält. Die durchschnittliche und die maximale Anzahl an Testschritten für die EXAM-Elemente *TestSuite*, *TestCase* und *TestSequence* ist in der Tabelle 8.1 dargestellt (ohne Berücksichtigung von *SystemConfigurations*). Die geringen Unterschiede zwischen den Werten bei *TestCase* und *TestSequence* ist damit zu erklären, dass in der Regel (laut Modellierungsrichtlinien) in einem *TestCase* nur *TestSequence* als *UseCallCall* referenziert werden.

Die Anzahl der Testschritte in einem Testablauf in EXAM kann aufgrund der Auswahl einer *SystemConfiguration* sehr stark variieren. Wie in den Abschnitt 2.4 und 6.3.5 beschrieben, können die Operationen der Funktionsbibliothek wiederum mit Hilfe von Sequenz- oder Aktivitätsdiagrammen modelliert werden. In diesen Diagrammen werden wieder Operationen der Funktionsbibliothek aufgerufen. Die Anzahl der Testschritte in einem Testablauf ist somit abhängig von der ausgewählten Systemkonfiguration. Aufgrund der Vielzahl der unterschiedlichen Systemkonfigurationen (typischerweise mehr als 50 Konfigurationen pro EXAM-Modell) können wir hier nicht

EXAM-Element	Durchschnittliche Anzahl an *OperationCalls*	Durchschnittliche Anzahl an *UseCaseCalls*	Durchschnittliche Anzahl an Testschritten	Maximale Anzahl an *UseCaseCalls*	Maximale Anzahl an *OperationCalls*	Maximale Anzahl an Testschritten
TestSequence	116	4	120	38	386	386
TestCase	150	8	158	44	431	445
TestSuite	2.361	145	2.506	3.061	37.779	40.840

Tabelle 8.1: Durchschnittliche und maximale Anzahl an Testschritten pro EXAM-Element

auf alle Varianten eingehen. Daher betrachten wir exemplarisch die Anzahl der Testschritte einer *TestSuite* bei der Auswahl einer Systemkonfiguration, in der die Mehrzahl der Operationen in der Funktionsbibliothek durch Sequenz- oder Aktivitätsdiagramme modelliert sind. Hierbei ergibt sich für die *TestSuite* mit der ausgewählten Systemkonfiguration eine Anzahl der Testschritte von circa zwei Millionen. Diesen Wert betrachten wir als Worst-Case-Szenario. Der Wert von zwei Millionen Testschritten ist nach unserer Analyse der EXAM-Modelle der aktuell längste Testablauf in den Modellen im Volkswagen-Konzern.

Für die Übertragung der *McCabe-Metrik* muss für die verschiedenen Interaktionsfragmente der Sequenzdiagramme und den Kontrollknoten in den Aktivitätsdiagrammen festgelegt werden, um welchen Wert diese Elemente den Komplexitätswert erhöhen. Jedes Interaktionsfragment vom Typ *alt*, *for*, *while*, *break* und *return* erhöht die zyklomatische Komplexität um eins. Bei den Kontrollknoten in den Aktivitätsdiagrammen ist die Erhöhung davon abhängig, wie viele Zweige von dem jeweiligen Kontrollknoten ausgehen. Die zyklomatische Komplexität wird dabei um die Anzahl der ausgehenden Kanten eines Kontrollknotens erhöht.

Basierend auf dieser Annahme ergeben sich die in der Tabelle 8.2 angegeben durchschnittlichen zyklomatischen Komplexitäten für die EXAM-Elemente. Neben den durchschnittlichen Werten sind die jeweiligen minimalen und maximalen Werte für die zyklomatische Komplexität in der Tabelle enthalten. Unter Berücksichtigung der Systemkonfigurationen ist die angegebene zyklomatische Komplexität je nach Systemkonfiguration im Schnitt um den Faktor zehn größer.

Neben der Komplexität eines einzelnen Testablaufs ist für die Bewertung des praktischen Einsatzes der Methode die Anzahl der zu überprüfenden EXAM-Elemente von Bedeutung. Wie in Abschnitt 2.4 beschrieben, wird in EXAM der Testablauf mit Hilfe der Elemente *TestSuite*, *TestCase* und *TestSequence* modelliert. In der Tabelle 8.3 ist die durchschnittliche Anzahl der EXAM-Elemente pro Modell (gerundet auf hundert Elemente) dargestellt.

Basierend auf den Betrachtungen zu der Komplexität der EXAM-Elemente können wir nun die benötigte Ausführungszeit des Verifikationsalgorithmus abschätzen.

Zur Bestimmung der Modellierungsfehler müssen im ersten Schritt die EXAM-Elemente in eine korrespondierende Netzdarstellung transformiert werden. Die für die Transformation benötigte Zeit ist direkt abhängig von der Anzahl der EXAM-Elemente und der Anzahl der enthaltenen

EXAM-Element	Durchschnittliche zyklomatische Komplexität	Minimale zyklomatische Komplexität	Maximale zyklomatische Komplexität
TestSequence	10	1	175
TestCase	19	1	218
TestSuite	331	1	4.988

Tabelle 8.2: Zyklomatische Komplexität der EXAM-Elemente

EXAM-Element	Durchschnittliche Anzahl pro Modell
TestSequence	15.000
TestCase	38.000
TestSuite	2.500

Tabelle 8.3: Durchschnittliche Anzahl der EXAM-Elemente pro Modell

TestSteps. Für jede *SpecificationConfiguration* ist die Transformation der benötigten EXAM-Elemente einmal auszuführen.

Im Anschluss an die Transformation der Elemente erfolgt die Berechnung der T-Invarianten mit Hilfe des im Abschnitt 4.3 beschriebenen Gauss-Jordan-Algorithmus. Die Größe des linearen homogenen Gleichungssystems ist direkt von der Anzahl der Stellen und Transitionen der entsprechenden Petri-Netz-Darstellung abhängig und somit indirekt von der Anzahl der *TestSteps* im Testablauf.

Wie im Kapitel 4.3.2 beschrieben sind die benötigten Rechenoperationen und somit die Ausführungszeit des Gauss-Jordan-Algorithmus abhängig von der Anzahl der Gleichungen (n) im Gleichungssystem. Als Abschätzung für die benötigten Rechenoperationen verwenden wir die Formel $\frac{2*n^3}{3}$.

Im Gegensatz zu der Transformation der EXAM-Elemente in die zugehörige Netzdarstellung muss die Berechnung der Schaltfolgen für die Überprüfung der Testabläufe mehrfach durchgeführt werden. Aus diesem Grund konzentrieren wir uns im Folgenden auf die Abschätzung der benötigten Lösungszeit zur Bestimmung der realisierbaren T-Invarianten. Die einmalig benötigte Zeit für die Transformation der EXAM-Elemente, welche einer *SpecificationConfiguration* zugeordnet sind, schätzen wir im Folgenden auf Basis von Messwerten und Erfahrungen mit Hilfe des Prototypen ab.

Betrachten wir zuerst die benötigte Zeit für die Transformation der einzelnen EXAM-Elemente in eine Petri-Netz-Darstellung. Die EXAM-Elemente, denen eine Spezifikation zugeordnet ist,

müssen in eine Netzdarstellung transformiert werden. Enthalten die Netze Referenzen auf Subnetze, müssen alle referenzierten Netze inkludiert werden und im Anschluss die zugehörige Inzidenzmatrix aufgestellt werden. Wie bereits erwähnt ist dieser Schritt nur einmal zu Beginn der Überprüfung für alle EXAM-Elemente in der *SpecificationConfiguration* notwendig.

Die benötigte Zeit für die Transformation der EXAM-Elemente in eine korrespondierende Petri-Netz-Darstellung und der Erstellung der Inzidenzmatrix ist abhängig von der Anzahl der zu transformierenden Elemente, der Anzahl der *TestSteps* und der Anzahl der referenzierten *TestCases*, *TestSequences* oder *TestActivities*.

Betrachten wir zuerst die notwendige Transformationszeit ohne die wiederverwendeten referenzierten EXAM-Elemente zu berücksichtigen. Für die Transformation einer *TestSequence* oder eines *TestCase* werden im Durchschnitt etwa fünfzehn Sekunden benötigt. Die benötigte Zeit für die Transformation einer *TestSuite* liegt bei etwa drei Minuten.

Werden mehrere EXAM-Elemente, z. B. alle Elemente die einer *SpecificationConfiguration* zugeordnet sind, für die Überprüfung transformiert, kann sich die benötigte Transformationszeit der Elemente reduzieren. Der Grund dafür ist, dass für jedes EXAM-Element eine eigene Netzdarstellung erzeugt wird (vergleiche Abschnitt 8.1). Somit wird bei jedem *UseCaseCall* überprüft, ob das referenzierte Element bereits in eine Netzdarstellung transformiert wurde.

Besonders deutlich ist der Effekt bei der Transformation von *TestSuiten*. Eine *TestSuite* kapselt *TestCases* für ein Prüfthema. Dadurch ist in der Regel der Anteil der gemeinschaftlich genutzten *TestSequences* sehr hoch, was zu einer deutlichen Reduktion der Transformationszeit führt. Ein weiterer Grund für die Reduktion der Transformationszeit, die sich aus der Wiederverwendung bestehender Netzdarstellungen ergibt, ist die Client-Server-Architektur von EXAM. Bei jedem *UseCaseCall* muss das referenzierte Element über ein Netzwerk bei der Datenbank angefragt werden und die Daten vom Server zum Client übertragen werden. Wird z. B. bei der Transformation einer *TestSuite* in den enthaltenen Testfällen eine *TestSequence* 25-mal referenziert, müssen die Daten nur einmal vom Server auf den Client übertragen werden.

Mit Hilfe dieses Ansatzes ergibt sich für eine durchschnittliche *SpecificationConfiguration* mit mehreren *TestSuiten*, *TestCases* und *TestSequences* eine Transformationszeit der EXAM-Elemente von etwa zehn Minuten. Die Transformationszeit ist natürlich auch wieder abhängig von der verwendeten Systemkonfiguration.

Nach der Transformation der EXAM-Elemente in eine korrespondierende Petri-Netz-Darstellung und der Erstellung der benötigten Inzidenzmatrizen kann die Anzahl der T-Invarianten berechnet werden. Wie im Abschnitt 4.3.2 dargestellt ist die Berechnungszeit zur Lösung der Gleichungssysteme abhängig von der Anzahl der Gleichungen. Hierbei kann die Lösungszeit für mehrere hunderttausend Gleichungen (bei einer vollbesetzten Matrix) mit Hilfe des Gauss-Jordan-Algorithmus mehrere Stunden betragen. Der zweite Einflussfaktor auf die Ausführungszeit ist die Anzahl der von Null verschiedenen Elemente in der Matrix. Bei einer schwach besetzten Matrix reduziert sich die Ausführungszeit deutlich, da in diesem Fall wesentlich weniger Additionen und Multiplikationen zur Transformation der Matrix in Zeilennormalform notwendig sind.

Bei der Überprüfung der EXAM-Elemente ist die Größe der Gleichungssysteme von der Anzahl der Operationsaufrufe innerhalb des Testablaufs abhängig. Durch die Transformation der Testa-

bläufe in Petri-Netze werden in Abhängigkeit der Interaktionsfragemente bzw. Kontrollknoten zusätzliche Transitionen und Stellen eingefügt, die jedoch bei der Betrachtung von großen Testabläufen vernachlässigt werden können. Für die Abschätzung der Laufzeit gehen wir an dieser Stelle davon aus, dass die Größe der Inzidenzmatrix direkt abhängig von der Anzahl der Operationsaufrufe ist.

Als Basis für die Abschätzung der Lösungszeiten verwenden wir die in der Tabelle 8.1 auf Seite 170 angegebene durchschnittliche und maximale Anzahl an Testschritten für die EXAM-Elemente. Die Berechnung der in der Tabelle 8.4 angegeben Lösungszeiten basiert auf den im Abschnitt 4.3.2 getroffenen Annahmen.

Die Abschätzung beruht an dieser Stelle auf der Annahme, dass die zugehörige Matrix vollständig besetzt ist. Später werden wir zeigen, dass aufgrund der Struktur der Testfälle in EXAM die Matrix in der Regel sehr schwach besetzt ist und daher sich die Lösungszeiten deutlich reduzieren.

EXAM-Element	Lösungszeit basierend auf der durchschnittlichen Anzahl an Testschritten	Lösungszeit basierend auf der maximalen Anzahl an Testschritten
TestSequence	< 1 Sekunde	< 1 Sekunde
TestCase	< 1 Sekunde	< 1 Sekunde
TestSuite	≈ 2 Sekunden	1,64 Stunden

Tabelle 8.4: Benötigte Zeit zur Berechnung der T-Invariante in Abhängigkeit der Testschritte der einzelnen EXAM-Elemente

Die in der Tabelle angegeben Lösungszeiten sind ohne die Berücksichtigung einer *SystemConfiguration* ermittelt worden. In dem von uns ermittelten Worst-Case-Szenario enthält die TestSuite bei Berücksichtigung der *SystemConfiguration* zwei Millionen Testschritte. Somit ergeben sich im Worst-Case bei der Berücksichtigung der *SystemConfiguration* auf Basis dieser Annahmen Lösungszeiten von mehreren Wochen (bei einer vollbesetzten Matrix) und somit für die praktische Anwendung der Methode nicht mehr akzeptabel.

Neben der Anzahl der Operationen hat auch die zyklomatische Komplexität der EXAM-Elemente einen Einfluss auf die Lösungszeit. Je mehr Verzweigungen in den Testabläufen enthalten sind, desto stärker ist die Matrix besetzt. In Abhängigkeit davon, wie stark bzw. wie schwach die Inzidenzmatrix besetzt ist, reduziert sich die Anzahl der benötigten Rechenoperationen und damit die benötigte Ausführungszeit deutlich. Grund dafür ist, dass bei einer schwach besetzten Matrix die Anzahl der benötigten Rechenoperationen zur Transformation der Matrix in Zeilennormalform deutlich geringer ist.

Bei einer Inzidenzmatrix bestimmt die Anzahl der Kanten in dem Petri-Netz wie stark bzw. wie schwach die Matrix besetzt ist. Übertragen auf EXAM bedeutet dies, dass eine niedrige *McCabe-Zahl* der EXAM-Elemente angibt, dass es sich um eine schwach besetzte Matrix handelt, d.h. ein

Großteil der Koeffizienten der Matrix gleich Null ist. Eine hohe zyklomatische Komplexität gibt eine stark besetzte Matrix an.

Wie bei der Anzahl der Testschritte ist für die Betrachtung der Auswirkung der zyklomatischen Komplexität das EXAM-Element *TestSuite* besonders interessant, da hier die höchsten Komplexitäten vorhanden sind. Wie in der Tabelle 8.2 auf Seite 171 angegeben ist die durchschnittliche *McCabe-Zahl* einer *TestSuite* 331. Im Verhältnis zu der durchschnittlichen Anzahl an Testschritten von 2.506 Testschritten kann in diesem Fall von einem schwach besetzten Gleichungssystem gesprochen werden. In etwa das gleiche Verhältnis besteht zwischen der höchsten zyklomatischen Komplexität einer *TestSuite* von 4.988 zu der maximalen Anzahl an Testschritten von 40.840. Somit kann auch hier von einer (sehr) schwach besetzten Matrix gesprochen werden. Bei der Berücksichtigung der Systemkonfiguration ergibt sich ein ähnliches Verhältnis, sodass auch hier von einer (sehr) schwach besetzten Matrix gesprochen werden kann. Die Auswertung von mehreren Inzidenzmatrizen von unterschiedlichen EXAM-Elementen mit verschiedenen *SystemConfigurations* hat ergeben, dass im Schnitt weniger als 8% der Werte in der Inzidenzmatrix ungleich Null sind.

Trotz der Reduktion der benötigten Lösungszeiten durch schwach besetzten Matrizen gestaltet sich der praktische Einsatz der Methode als schwierig. Weiterhin haben wir uns bei der bisherigen Betrachtung nur auf ein einziges EXAM-Element beschränkt. Im Schnitt enthält ein EXAM-Modell in Summe rund 55.500 *TestSequences*, *TestCases* und *TestSuites*, die im schlimmsten Fall alle überprüft werden müssen. Aus diesem Grund ist eine weitere Optimierung der Lösungszeit für den praktischen Einsatz der Methode notwendig.

8.2.2 Reduktion der Netzdarstellungen

Die beschriebene Komplexität der zu überprüfenden EXAM-Elemente und die sich daraus ergebenden Lösungszeiten der linearen Gleichungssysteme macht es für den praktischen Einsatz der Methode notwendig, die Anzahl der Transitionen und Stellen in der Netzdarstellung der Testabläufe zu reduzieren. Die Reduktion darf jedoch keine Auswirkungen auf das Ergebnis der Überprüfung haben. Daher muss sichergestellt werden, dass alle benötigten Operationsaufrufe (mindestens die Operationen, die in der Aktionsmenge aller zugeordneten Spezifikationen enthalten sind) weiterhin in der Netzdarstellung enthalten sind.

Die für die Überprüfung notwendigen Operationsaufrufe ergeben sich aus der Vereinigung aller Aktionsmengen der Spezifikationen, die dem zu überprüfenden EXAM-Element mit Hilfe einer *SpecificationConfiguration* zugeordnet sind. Alle weiteren Operationsaufrufe innerhalb des EXAM-Elements haben keine Auswirkungen auf das Ergebnis der Überprüfung und müssen daher nicht als Transition in der Netzdarstellung dargestellt werden.

Auf Grund der Zuordnung der Spezifikationen zu den EXAM-Elementen muss die Reduktion der zugehörigen Netzdarstellung für Elemente mit und ohne enthaltenen *UseCaseCalls* (Referenzen auf weitere EXEM-Elemente) unterschieden werden. Diese Unterscheidung ist notwendig, da die Spezifikationen immer nur dem jeweiligen zu überprüfenden Element zugeordnet werden und nicht automatisch allen referenzierten Elementen. Daher betrachten wir im Folgenden zuerst die

Reduktion von EXAM-Elementen ohne enthaltene *UseCaseCalls* und im Anschluss von EXAM-Elementen mit *UseCaseCalls*.

Reduktion für EXAM-Elemente ohne UseCaseCalls

Sind in dem zu überprüfenden EXAM-Element keine *UseCaseCalls* enthalten, müssen bei der Transformation des Elements in die korrespondierende Netzdarstellung nur diejenigen Operationsaufrufe berücksichtigt werden, die in einer der zugeordneten Spezifikationen enthalten sind. Für alle Kontrollstrukturen gilt, dass solange innerhalb der Kontrollstrukturen mindestens eine spezifizierte Operation enthalten ist, muss die Kontrollstruktur bei der Transformation berücksichtigt werden. Ist in der Kontrollstruktur keine spezifizierte Operation enthalten, hat die Kontrollstruktur keinen Einfluss auf die Verifikation und muss somit nicht in die Netz-Darstellung übertragen werden.

Betrachten wir zur Verdeutlichung den in Abbildung 8.3 auf Seite 175 dargestellten Testablauf. Der Testablauf soll das automatische Anklappen der Außenspiegel überprüfen. Für die Überprüfung der kausalen Abhängigkeit der verwendeten Operationen sind dem EXAM-Element die beiden Spezifikationen

$$S_1 = (ZuendungEin() \circledcirc ZuendungAus())$$

und

$$S_2 = (initDataLogger() \circledcirc deInitDataLogger())$$

zugeordnet. Die Art der Spezifikation spielt für die Reduktion der Netzdarstellung keine Rolle und ist aus diesem Grund an dieser Stelle nicht angegeben.

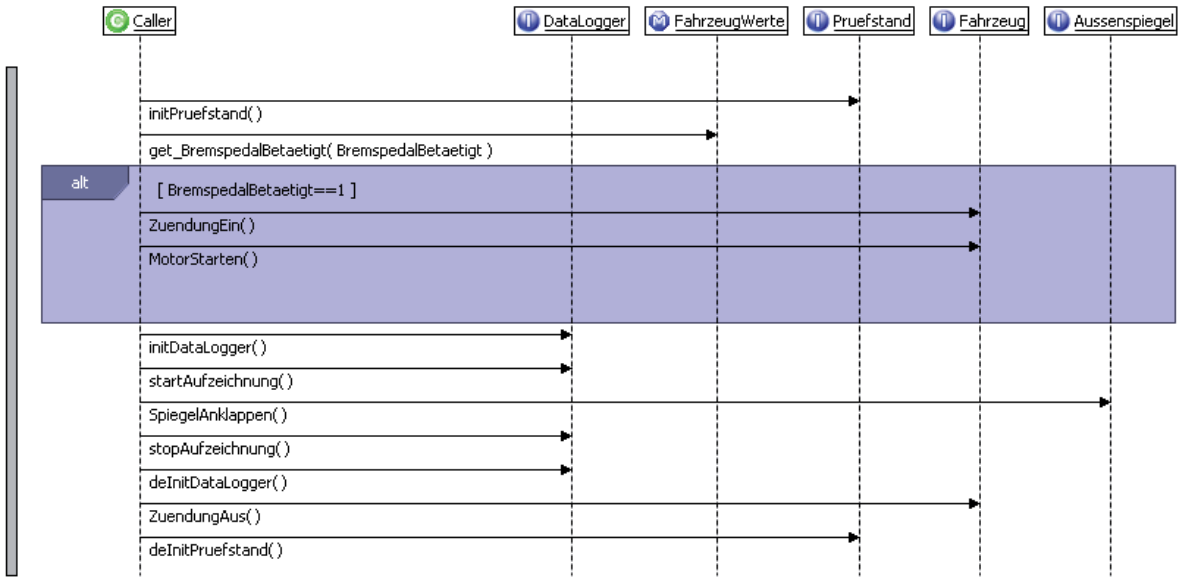

Abbildung 8.3: Testfall als Beispiel für die Reduktion der Netzdarstellung

Aus der Vereinigung der beiden Operationsmengen der Spezifikationen S_1 und S_2 ergibt sich die Operationsmenge

$$M_{S_1S_2} = \{initDataLogger(), deInitDataLogger(), ZuendungEin(), ZuendungAus()\}.$$

Basierend auf der Operationsmenge $M_{S_1S_2}$ kann nun die Transformation des Testfalls in eine korrespondierende Petri-Netz-Darstellung durchgeführt werden. Alle Operationsaufrufe die nicht in der Operationsmenge $M_{S_1S_2}$ enthalten sind, werden bei der Transformation nicht berücksichtigt. Die resultierende Netzdarstellung ist in Abbildung 8.4 auf Seite 176 dargestellt. Im Vergleich zur der nicht reduzierten Netzdarstellung konnte durch die Reduktion die Anzahl der Transitionen von 14 um die Hälfte auf 7 Transitionen reduziert werden.

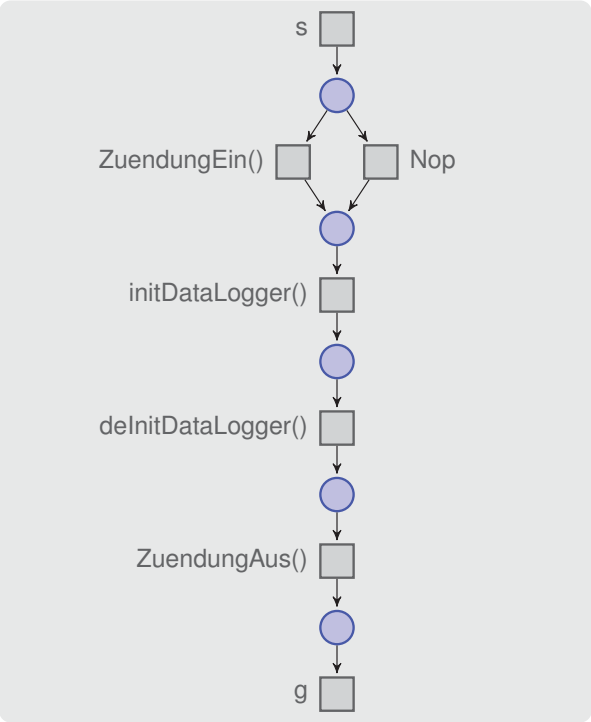

Abbildung 8.4: Reduzierte Netzdarstellung basierend auf der Operationsmenge $M_{S_1S_2}$

Die geschilderte Reduktion hat keinen Einfluss auf die Transformation des Interaktionsfragments vom Typ *alt*, da das Fragment eine Operation aus der Operationsmenge $M_{S_1S_2}$ enthält.

Wird der Testablauf nur auf die Einhaltung der Spezifikation S_2 mit der Operationsmenge

$$M_{S_2} = \{initDataLogger() \circledcirc deInitDataLogger()\}$$

überprüft, enthält das Fragment *alt* in diesem Fall keinen Operationsaufruf aus der Operationsmenge M_{S_2} und muss daher nicht in die Netzdarstellung transformiert werden. Die sich daraus ergebende Netzdarstellung ist in Abbildung 8.5 auf Seite 177 dargestellt. Hier konnte die Anzahl der benötigten Transitionen von 14 Transitionen auf 4 Transitionen reduziert werden.

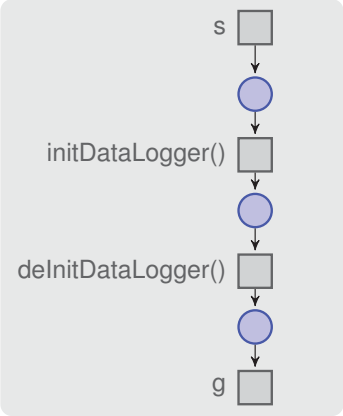

Abbildung 8.5: Reduzierte Netzdarstellung basierend auf der Operationsmenge M_{S_2}

Reduktion für EXAM-Elemente mit UseCaseCalls

Wie zuvor beschrieben ist die Reduktion der EXAM-Elemente abhängig von den zugeordneten Spezifikationen. Durch eine *SpecificationConfiguration* werden den EXAM-Elementen eine oder mehrere Spezifikationen zugeordnet. Bei der Ausführung der Verifikation werden alle Elemente im EXAM-Modell überprüft, denen mindestens eine Spezifikation zugeordnet ist. Für die Durchführung der Überprüfung erfolgt im ersten Schritt, wie im Kapitel 6 beschrieben, die Transformation der EXAM-Elemente in die korrespondierende Netzdarstellung.

Damit zum Beispiel eine *TestSequence*, die von mehreren anderen EXAM-Elementen verwendet wird, nur einmal transformiert werden muss, erfolgt die Transformation für jedes EXAM-Element separat. Welche Elemente transformiert werden müssen ist zum einen dadurch bestimmt, welchen Elementen direkt eine Spezifikation zugeordnet ist und zum anderen welche Elemente innerhalb des zu überprüfenden Testablaufs per *UseCaseCall* referenziert werden. Den referenzierten Elementen ist in diesem Fall keine Spezifikation zugeordnet, da eine Überprüfung dieser Elemente nur im Kontext z. B. eines *TestCases* oder einer *TestSuite* sinnvoll ist.

Wir geben ein Beispiel für ein besseres Verständnis der Aussage. Wird bei der Ausführung einer *TestSuite* eine *TestSequence* für die Berechnung von Messwerten referenziert (per *UseCaseCall*), ist eine Überprüfung, ob in dem Testablauf der Prüfstand richtig initialisiert und deinitialisert wurde, nur auf Ebene der *TestSuite* sinnvoll und nicht für jede *TestSequence* separat. Die einzelne *TestSequence* muss keine Initialisierung bzw. Deinitialisierung des Prüfstands durchführen, da dies zentral in der referenzierenden *TestSuite* erfolgt. Aus diesem Grund kann eine Spezifikation, die einem EXAM-Element zugeordnet ist, nicht direkt auf alle referenzierten Elemente übertragen werden, da es sonst zu falschen Überprüfungsergebnissen kommt.

Für die Reduktion der Netzdarstellung ist es jedoch erforderlich, die Spezifikationen auch allen referenzierten Elementen zuzuordnen. In diesem Fall würden jedoch diese Elemente unabhängig von ihrem Kontext auf Einhaltung der Spezifikation überprüft. Dies kann zu falschen Ergebnissen führen, da die Elemente nur im Kontext die Bedingung erfüllen können bzw. müssen. Aus diesem Grund ist es notwendig zwei Arten der Zuordnung einer Spezifikation zu einem Testablauf zu

unterscheiden: *direkte* und *indirekte Zuordnung*. Synonym zu diesen Begriffen verwenden wir die Begriffe *direkte Spezifikation* und *indirekte Spezifikation*.

Eine *direkte Zuordnung* erfolgt durch eine *SpecificationConfiguration*. Jedem referenzierten Element (*UseCaseCall*) eines EXAM-Elements mit einer *direkten Spezifikation* wird automatisch die jeweilige Spezifikation *indirekt* zugeordnet. Eine *indirekte Spezifikation* gibt dabei nur an, dass bei der Reduktion der Netzdarstellung alle Operationen aus der Vereinigungsmenge aller Operationsmengen der Spezifikationen (sowohl *direkt* als auch *indirekt* zugeordnete Spezifikationen) betrachtet werden müssen, jedoch erfolgt für EXAM-Elemente mit einer *indirekten Spezifikation* keine eigenständige Überprüfung der definierten kausalen Abhängigkeiten.

Dieser Ansatz erlaubt es, die EXAM-Elemente separat in eine Netzdarstellung zu transformieren. Durch die Vereinigung aller Operationsmengen der zugeordneten Spezifikationen wird sichergestellt, dass in der entsprechenden Netzdarstellung alle benötigten Operationen enthalten sind und zwar unabhängig davon, von welchem EXAM-Element das jeweilige Element referenziert wird.

Betrachten wir als Beispiel den in Abbildung 8.6 in Form einer Baumstruktur dargestellten Testfall *TestCaseExample*. In dem *TestCase* werden die *TestSequences InitSequence1*, *TestCaseSequence1*, *SubSequence*, *TestCaseSequence2* und *DeInitSequence1* referenziert. Zur Vereinfachung des Beispiels sind im Testablauf keine Verzweigungen enthalten.

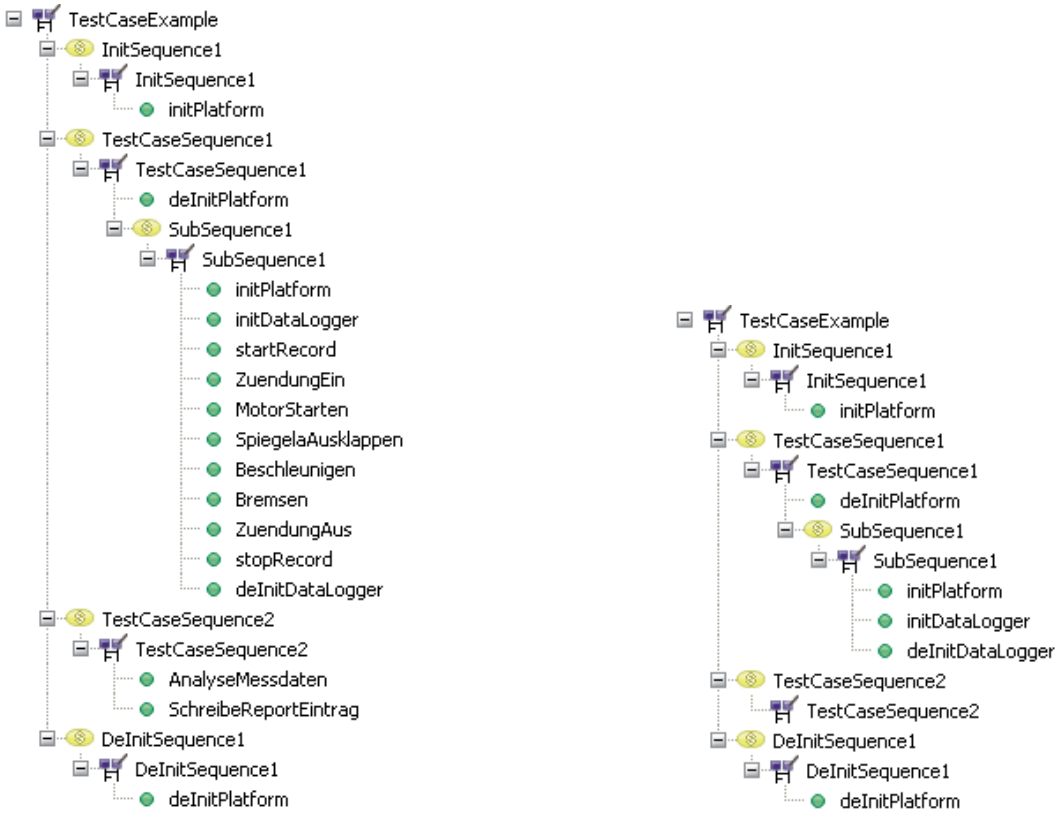

Abbildung 8.6: *TestCase TestCaseExample* mit allen referenzierten *TestSequences* (links ohne Reduktion, rechts mit Reduktion)

Wir ordnen dem *TestCase* die Spezifikation

$$S_1 = \{initPlatform() \oslash deInitPlatform()\}$$

und der Sequenz *TestCaseSequence1* die Spezifikation

$$S_2 = \{initDataLogger() \oslash deInitDataLogger()\}$$

zu.

Durch die Übertragung der Spezifikationen auf die referenzierten Elemente in Form einer *indirekten Spezifikation* ergeben sich für die einzelnen Elemente die in der Tabelle 8.5 auf Seite 179 dargestellten Zuordnungen der direkten und indirekten Spezifikationen. Ebenso ist für jeder Zeile in der Tabelle die sich aus der Vereinigung alle Operationsmengen der zugeordneten Spezifikationen ergebende Operationsmenge *M* angegeben. Auf Basis der jeweiligen Operationsmenge *M* kann jedes in der Tabelle dargestellte EXAM-Element unabhängig voneinander in eine korrespondierende reduzierte Netzdarstellung transformiert werden.

EXAM-Element	Direkte Spezifikation	Indirekte Spezifikation	Operationsmenge M
TestCaseExample	*S1*	-	{*initPlatform*(), *deInitPlatform*()}
InitSequence1	-	*S1*	{*initPlatform*(), *deInitPlatform*()}
TestCaseSequence1	*S2*	*S1*	{*initDataLogger*(), *deInitDataLogger*(), *initPlatform*(), *deInitPlatform*()}
TestCaseSequence2	-	*S1*	{*initPlatform*(), *deInitPlatform*()}
DeInitSequence1	-	*S1*	{*initPlatform*(), *deInitPlatform*()}
SubSequence1	-	*S1*,*S2*	{*initDataLogger*(), *deInitDataLogger*(), *initPlatform*(), *deInitPlatform*()}

Tabelle 8.5: Zuordnung der direkten und indirekten Spezifikationen

Die Petri-Netz-Darstellung für die einzelnen Elemente sind in den Abbildungen 8.7 bis 8.9 dargestellt. Die *UseCaseCalls* in den Sequenzdiagrammen sind in der Netzdarstellung als rote Transitionen mit dem Namen des referenzierten Elements hervorgehoben. Das für die Verifikation des Testfalls *ExampleTestCase* notwendige kombinierte Petri-Netz ist in Abbildung 8.10 auf Seite 182 dargestellt.

Durch die Reduktion konnte die Anzahl der benötigten Transitionen in der Netzdarstellung von 18 Transitionen auf 8 Transitionen reduziert werden. Hervorzuheben ist an dieser Stelle die *TestSequence SubSequence1*. Der Sequenz *SubSequence1* ist keine direkte Spezifikation zugeordnet.

Sie erbt aber durch die Aufrufhierarchie sowohl die Spezifikation S_1, die dem Testfall *TestCaseExample* zugeordnet ist, als auch die Spezifikation S_2 der *TestSequence TestCaseSequence1*. Auf Basis dieser beiden Spezifikationen kann durch die Reduktion die Anzahl der Transitionen in dem entsprechenden Petri-Netz für die *TestSequence SubSequence1* von 13 Transitionen auf 5 Transitionen reduziert werden.

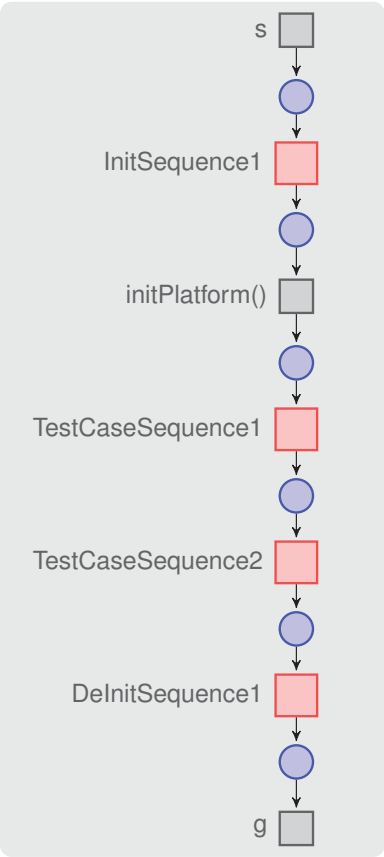

Abbildung 8.7: Reduzierte Netzdarstellung für den *TestCase TestCaseExample* mit *UseCaseRefs*

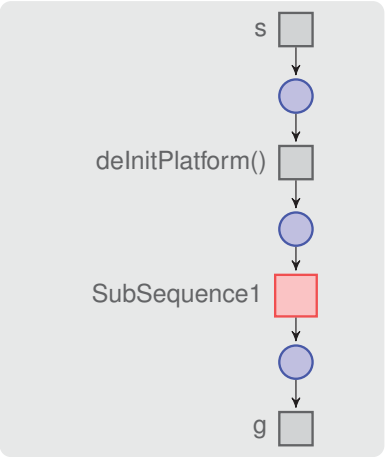

Abbildung 8.8: Reduzierte Netzdarstellung für die *TestSequence TestCaseSequence1*

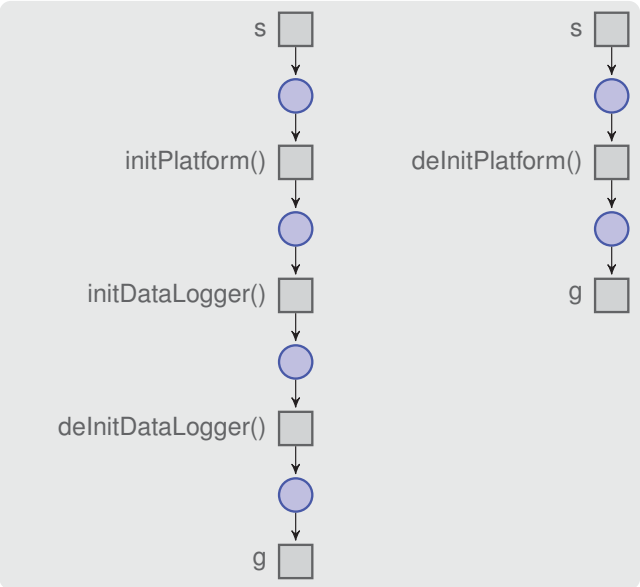

Abbildung 8.9: Reduzierte Netzdarstellung für die *TestSequence SubSequence1* und *DeInitSequence1*

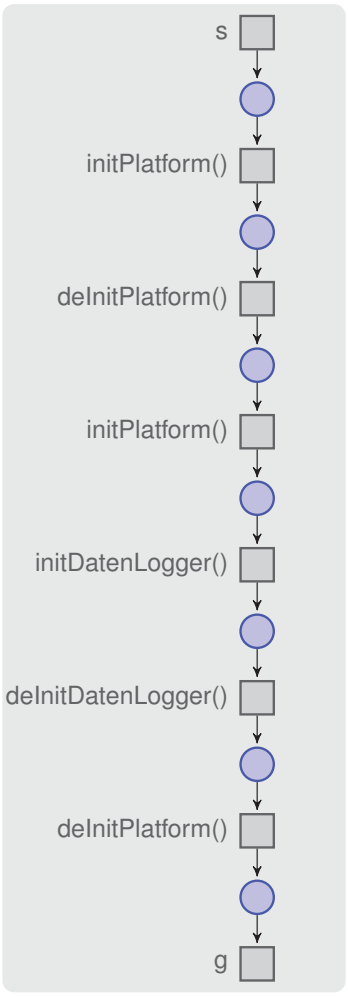

Abbildung 8.10: Reduzierte Netzdarstellung des *TestCases TestCaseExample*

8.2.3 Ergebnis der Reduktion

Wie im Abschnitt 8.2.2 gezeigt kann durch eine Reduktion der Netzdarstellung auf Basis der Operationsmenge der *direkten* und *indirekten Spezifikationen* die Anzahl der Transitionen und Stellen in den Petri-Netzen deutlich verringert werden. Der Grad der Reduktion ist abhängig von der Anzahl der Operationen in der vereinigten Operationsmenge der zugeordneten Spezifikationen. Neben der Anzahl der Operationen kann durch die Reduktion die Anzahl der Verzweigungen in den Netzen verkleinert werden, was zu einer geringeren zyklomatischen Komplexität führt. Dies hat eine direkte Auswirkung darauf, wie stark bzw. wie schwach die Inzidenzmatrix besetzt ist und damit auf die benötigte Lösungszeit.

In der EXAM-Funktionsbibliothek gibt es eine Reihe von Operationen, die vollständig unabhängig von anderen Operationen aufgerufen werden können. Diese Operationen spielen somit für die Überprüfung der Ausführungsreihenfolge keine Rolle und sind daher auch in keiner Spezifikation enthalten. Die Operation *defineVariable()* für die Definition einer lokalen Variablen oder die Operation *logInfoMessage()* für die Ausgabe einer Nachricht in die Konsole des EXAM-TestRunners

sind Beispiele für solche Funktionen, die keine kausale Abhängigkeit zu anderen Operationen haben. Wir bezeichnen diese Operationen im Folgenden als *Unterstützungsoperationen*.

In der Tabelle 8.6 ist die durchschnittliche und maximale Anzahl an Testschritten der EXAM-Elemente und der jeweilige Anteil an *Unterstützungsoperationen* dargestellt (ohne die Berücksichtigung der *SystemConfiguration*). Rund 50 Prozent aller Operationen in den Testabläufen sind *Unterstützungsoperationen*. Eine *Unterstützungsoperationen* hat keine zwingende kausale Abhängigkeit zu weiteren Operationen in dem Testablauf, d. h. sie kann an einer beliebigen Stelle im Testablauf aufgerufen werden. Aus diesem Grund müssen diese Operationen bei der Verifikation nicht berücksichtigt werden und müssen daher nicht zwingend in den Netzdarstellungen enthalten sein.

EXAM-Element	Durchschnittliche Anzahl an Testschritten	Durchschnittliche Anzahl an Unterstützungs-operationen	Maximale Anzahl an Testschritten	Maximale Anzahl an Unterstützungs-operationen
TestSequence	120	59	386	245
TestCase	158	72	445	295
TestSuite	2.506	1.135	40.840	20.810

Tabelle 8.6: Durchschnittliche und maximale Anzahl an Testschritten pro EXAM-Element mit Unterstützungsoperationen

Betrachten wir die Anzahl der *Unterstützungsoperationen* unter Berücksichtigung der *SystemConfiguration* für die im Abschnitt 8.2.1 dargestellte *TestSuite* mit circa zwei Millionen Testschritten. Hier wird deutlich, dass der Anteil der *Unterstützungsoperationen* deutlich mehr als 50 Prozent aller Operationen ausmacht. In der *TestSuite* sind alleine rund 1,3 Millionen Aufrufe der Operationen *defineVariable()* und *logInfoMessage()* enthalten. Insgesamt sind in der Suite ca. 1,8 Millionen *Unterstützungsoperationen* enthalten. Der Grund für die große Anzahl an *Unterstützungsoperationen* ist der Aufbau der Operationen der Funktionsbibliothek, welche mit Hilfe von Sequenz- bzw. Aktivitätsdiagrammen modelliert sind. In der Regel enthalten diese Operationen eine Vielzahl an *Unterstützungsoperationen*.

Die Tabelle 8.7 stellt die Auswirkungen der Reduktion der Testabläufe (mit und ohne Berücksichtigung der *Unterstützungsoperationen*) auf die Ausführungszeit der Verifikation dar. Hierbei ist zu beobachten, dass aufgrund des Zusammenhangs zwischen der Anzahl der Gleichungen (n) und der benötigten Anzahl an Rechenoperationen ($\frac{2*n^3}{3}$), die benötigte Lösungszeit deutlich geringer ist.

Bei der Berücksichtigung der *SystemConfiguration* sind in dem vorgestellten Worst-Case-Szenario in der Netzdarstellung noch rund 200.000 Testschritte enthalten. Somit beträgt die Ausführungszeit immer noch mehrere Stunden unter der Annahme einer vollbesetzten Matrix.

EXAM-Element	Lösungszeit basierend auf der durchschnittlichen Anzahl an Testschritten	Lösungszeit basierend auf der durchschnittlichen Anzahl an Testschritten mit Reduktion	Lösungszeit basierend auf der maximalen Anzahl an Testschritten	Lösungszeit basierend auf der maximalen Anzahl an Testschritten mit Reduktion
TestSequence	< 1 Sekunde	< 1 Sekunde	< 1 Sekunde	< 1 Sekunde
TestCase	< 1 Sekunde	< 1 Sekunde	< 1 Sekunde	< 1 Sekunde
TestSuite	1,46 Sekunden	0,2 Sekunden	1,64 Stunden	0,2 Stunden

Tabelle 8.7: Vergleich der benötigten Zeit zur Berechnung der Lösungsmenge des Gleichungssystems mit und ohne Reduktion der Netzdarstellung

Die vorgestellten Betrachtungen der benötigten Lösungszeit der linearen Gleichungssysteme gehen von einer vollbesetzten Matrix aus, also von einer Matrix, in der alle Elemente ungleich Null sind. In Bezug auf die Anzahl der benötigten Rechenoperationen zur Erstellung der Zeilennormalform ist dies selbstverständlich der schlimmste Fall (Worst-Case-Betrachtung), da in diesem Fall die meisten Rechenoperationen notwendig sind. Dieser wird aufgrund der Struktur der Testabläufe in EXAM jedoch nie erreicht. Bei einer schwach besetzten Matrix ist die Anzahl der benötigten Rechenoperationen zur Umformung der Matrix in Zeilennormalform und für die Berechnung der T-Invarianten deutlich geringer.

Um ein Gefühl für den Grad der Reduktion der benötigten Rechenoperationen bei einer schwach besetzten Matrix zu bekommen, betrachten wir als Beispiel die beiden schwach besetzten Matrizen 7.1 und 7.3 im Kapitel 7.

In der Matrix 7.1 sind bereits eine Vielzahl der Bedingungen der Zeilennormalform (z. B. das erste von Null verschiedene Element in jeder Zeile muss gleich Eins sein oder unterhalb des ersten von Null verschiedenen Elements jeder Zeile stehen nur Nullen) erfüllt. Für die Transformation der Matrix in Zeilennormalform sind noch weniger als 100 Operationen notwendig. Im Vergleich, bei einer vollbesetzten Matrix gleicher Größe wären maximal $\frac{23^3}{3} \approx 4.055$ Operationen notwendig.

Auch in der zweiten Matrix 7.3 auf Seite 154 sind bereits viele Bedingungen der Zeilennormalform erfüllt. Bei dieser Matrix sind in etwa 100 Rechenoperationen für die Transformation der Matrix in Zeilennormalform notwendig. Hier geben wir auch wieder zum Vergleich die benötigte Anzahl an Operationen bei einer vollbesetzten Matrix gleicher Größe an. Für die Transformation sind dabei rund $\frac{26^3}{3} \approx 5.858$ Operationen notwendig.

Wie bereits beschrieben ist die Anzahl der Operationen, welche in die Netzdarstellung transformiert werden müssen, abhängig von der Operationsmenge aller Spezifikationen einer *SpecificationConfiguration*. In der vorherigen Betrachtung der benötigten Ausführungszeit des Verifikationsalgorithmus sind wir davon ausgegangen, dass in der Operationsmenge der *SpecificationConfiguration* alle Operationen bis auf die Unterstützungsoperationen enthalten sind. Die

Erfahrungen aus der Key-User-Testphase haben gezeigt, dass in der Regel in einer *Specification-Configuration* in etwa nur 50 verschiedene Operationen in der vereinigten Operationsmenge aller Spezifikationen enthalten sind. Die Operationen können natürlich mehrfach in den Testabläufen enthalten sein, sodass im Durchschnitt die Petri-Netze, welche eine *TestSuite* repräsentieren, in etwa 500 Transitionen und Stellen enthalten. Die Anzahl der Transitionen und Stellen ist natürlich abhängig von der ursprünglichen Größe der *TestSuite* und der Anzahl der Verzweigungen im Testablauf. Bei 500 Stellen und Transitionen ergibt sich ungefähr eine theoretische Zeitdauer für die Umformung der Inzidenzmatrix des Petri-Netzes auf Zeilennormalform von etwa einer Sekunde. Bei der Berücksichtigung der *SystemConfigurationen* erhöht sich der Wert auf etwa 6.000 Transitionen und Stellen. Im Vergleich: die *TestSuite* enthält in unserem betrachteten Worst-Case-Szenario bis zu 200.000 Operationen (Anzahl der Operationen ohne die *Unterstützungsoperationen*). Bei 6.000 Transitionen und Stellen beträgt die benötigte Lösungszeit im Worst-Case bei einer stark besetzten Inzidenzmatrix in etwa 20 Sekunden.

Wie dargestellt kann durch die Reduktion der Netzdarstellungen die benötigte Ausführungszeit des Verifikationsalgorithmus deutlich reduziert werden. In Abhängigkeit der Auslastung der EXAM-Modelldatenbank wurden bei den Key-User-Tests für die vollständige Überprüfung des angesprochenen Worst-Case-Szenario, mit etwa 50 verschiedenen Operationen in der Operationsmenge der Spezifikationen, Ausführungszeiten von etwa 10 Minuten bis 15 Minuten benötigt. Für die vollständigen Überprüfung wurden dabei die folgenden Schritte durchgeführt:

- Transformation der Testabläufe in die korrespondierende Petri-Netz-Darstellung,
- Vereinigung der Netzdarstellungen,
- Aufstellung der Inzidenzmatrizen,
- Berechnung der Lösung der linearen homogenen Gleichungssysteme und
- Bestimmung der realisierbaren T-Invarianten

In Abhängigkeit der ausgewählten *SystemConfiguration* kann die Überprüfung auch mehr als eine Stunde betragen. Da die Überprüfung vollständig automatisiert abläuft und die Anwender in ihrer Arbeit nicht beeinträchtigt werden, ist die benötigte Ausführungszeit von einer Stunde im Vergleich zu der Ausführungszeit am Prüfstand von bis zu 24 Stunden akzeptabel. Wird ein Fehler erst am Prüfstand während der Testausführung festgestellt, muss im schlimmsten Fall der gesamten Ablauf (Ausführungszeit bis zu 24 Stunden) wiederholt werden. Die Überprüfung der kausalen Abhängigkeiten kann z. B. über Nacht vor der Ausführung des Testablaufs am realen Prüfstand erfolgen. Vor dem Start des Testablaufs am nächsten Morgen können die gefunden Fehler ausgebessert werden.

Die Ausführungszeiten der Überprüfung sind zusätzlich sehr stark abhängig vom Typ der Spezifikationen. Wie im Kapitel 6 beschrieben kann für bestimmte Typen der Spezifikationen ohne die Berechnung der T-Invarianten bestimmt werden, ob der Testablauf die Spezifikation erfüllt. Dadurch kann die benötigte Dauer der gesamten Überprüfung der *SpecificationConfiguration* nochmals deutlich reduziert werden.

Mit Hilfe des Prototypen konnten wir anhand einer Vielzahl von praktischen Anwendungsbeispielen zeigen, dass die Ausführungszeiten der Methode für unterschiedliche Testabläufe und unterschiedliche Arten von Spezifikationen für den praktischen Einsatz geeignet ist.

8.3 Darstellung der Modellierungsfehler

Aus Sicht der EXAM-Anwender ist neben der benötigten Ausführungszeit die Darstellung der gefundenen Fehler in den Testabläufen von entscheidender Bedeutung.

In diesem Abschnitt zeigen wir anhand eines Beispiels, wie dem Benutzer die Ergebnisse der Überprüfung angezeigt werden und wie er bei der Korrektur der Testabläufe unterstützt wird. Dazu erweitert der Prototyp die Benutzeroberfläche des EXAM-Modeller um die beiden Fenster *VerificationProblemView* und *DependencyView*. Die gefundenen Fehler werden direkt in den entsprechenden Sequenz- bzw. Aktivitätsdiagrammen angezeigt.

Ausgangspunkt unserer Betrachtung ist der bereits im Abschnitt 8.2.2 eingeführte Testfall *TestCaseExample*. Zur Darstellung eines Fehlers haben wir die *TestSequence InitSequence1* um den Operationsaufruf *initDataLogger()* ergänzt. Der sich nun daraus ergebende neue Testablauf ist in der Baumdarstellung auf der linken Seite der Abbildung 8.11 dargestellt.

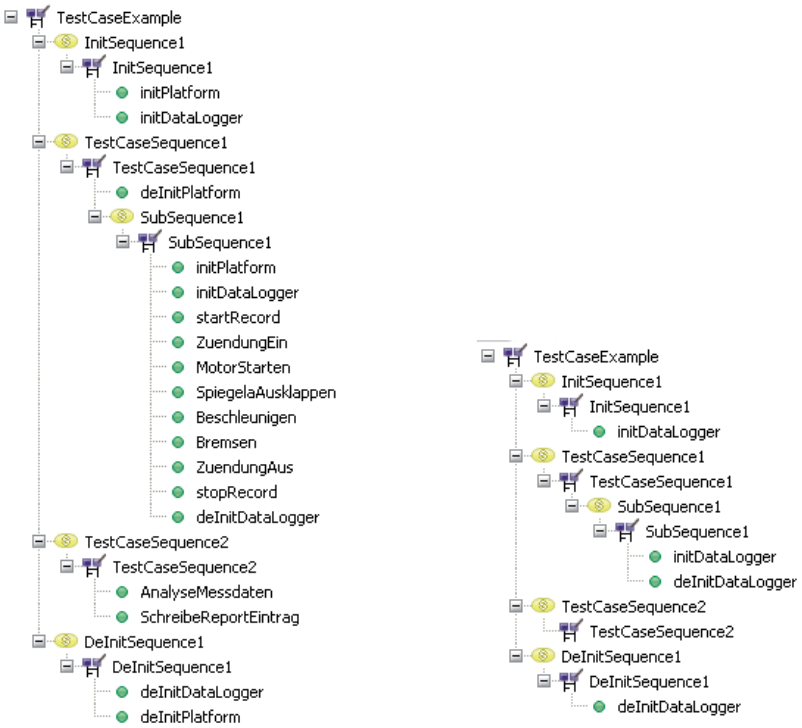

Abbildung 8.11: Testfall als Beispiel für die Anzeige von Modellierungsfehlern (links ohne Reduktion, rechts mit Reduktion)

Wir wollen nun überprüfen ob in dem Testablauf der Datenlogger nach jeder Initialisierung auch wieder deinitialisiert wird. Dazu definieren wir die Spezifikationen

$$S_1 = \{initDataLogger() \oslash deInitDataLogger()\}.$$

Der Spezifikation S_1 sind hierbei die Attribute *Must-One-To-One* zugeordnet. Bei einer *Must-One-To-One* Spezifikation müssen die Elemente der Aktionsmenge der Spezifikation in der spe-

zifizierten Reihenfolge im Testablauf vorkommen und dürfen nicht von weiteren Operationen aus der Aktionsmenge der Spezifikation unterbrochen werden.

Basierend auf der zugeordneten Spezifikation S_1 ist auf der rechten Seite der Abbildung 8.11 der reduzierte Testablauf dargestellt. Anhand der reduzierten Darstellung ist ersichtlich, dass der Testablauf die Spezifikation nicht erfüllt. Nach der Initialisierung des Datenloggers in *InitSequence1* wird versucht, den Datenlogger ohne vorherige Deinitialisierung in der *TestSequence SubSequence1* erneut zu initialisieren.

Im EXAM-Tool bekommt der Anwender die Ergebnisse der Überprüfung in dem *VerificationProblemsView* (Abbildung 8.12) angezeigt. Dieses Fenster ist eine Komponente des umgesetzten Prototypen und erweitert die standardmäßige Benutzeroberfläche des EXAM-Modellers.

In diesem Fenster werden alle Verletzungen einer Überprüfung in Form einer Tabelle angezeigt. Jede Verletzung einer Spezifikation wird dabei in einer eigenen Tabellenzeile eingetragen. Jede Zeile enthält neben der Angabe des fehlerhaften EXAM-Elements den Namen des Autors und das Datum der letzten Modifikation des Elements. Zusätzlich wird angezeigt, welche Spezifikation verletzt worden ist und ein entsprechender Fehlertext wird ausgegeben.

Für das aktuelle Beispiel ist der *VerificationProblemsView* in Abbildung 8.12 dargestellt. Im oberen Bereich der Abbildung ist das Sequenzdiagramm des Testfalls *TestCaseExample* und im unteren Bereich der *VerificationProblemsView* enthalten.

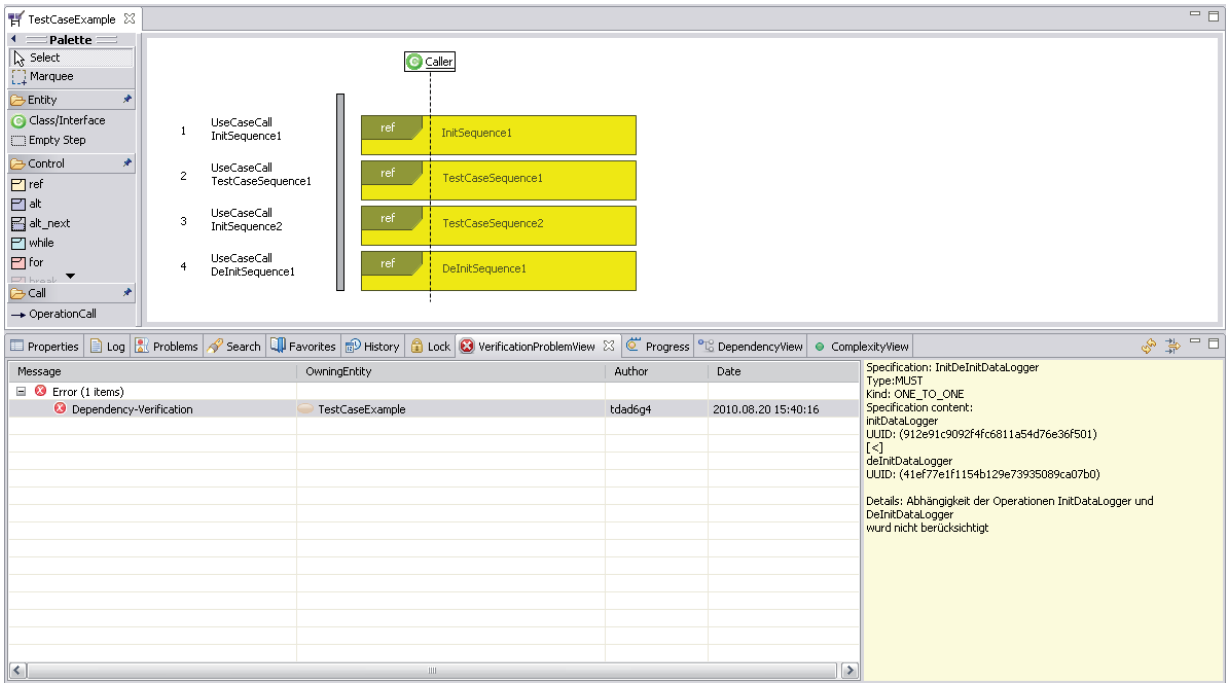

Abbildung 8.12: Ansicht des VerificationProblemView

Wie beschrieben verletzt der Testfall die Spezifikation S_1. Entsprechend ist in dem *VerificationProblemsView* ein Fehler eingetragen. Die Spalte *Message* sagt dabei aus, dass es sich um eine Verletzung der Ablaufreihenfolge handelt. Das betroffene Element ist *TestCaseExample* und wurde von dem Autor *tdad6g4* am 20.08.2010 um 15:40 Uhr zuletzt geändert. Im rechten Bereich des

Fensters werden die Details zu der verletzten Spezifikation angegeben. In diesem Fall ist der Name der Spezifikation *InitDeInitDataLogger*. Die Art der Spezifikation ist *Must-One-To-One* und das zugehörige Aktionslogik-Modul ist (*initDataLogger* $\oslash$ *deinitDataLogger*). Jedes Element in EXAM besitzt zur eindeutigen Identifikation einen eindeutigen Bezeichner. Dies ist in EXAM mit Hilfe eines *Universally Unique Identifier* (UUID) realisiert. Diese beiden UUIDs werden zusätzlich in dem *VerificationProblemsView* als weitere Information mit ausgegeben. Die letzte Angabe ist ein Fehlertext, der bei der Erstellung der Spezifikation vom *Qualitätsbeauftragten* für jede Spezifikation zum besseren Verständnis im Fehlerfall angegeben werden kann.

Durch einen Doppelklick auf eine Zeile in der Tabelle des *VerificationProblemsView* wird zum einen das Sequenz- oder Aktivitätsdiagramm des betroffenen EXAM-Elements im oberen Bereich des Tools geöffnet (in diesem Beispiel das Sequenzdiagramm von *TestCaseExample*) und zum anderen wird ein weiteres Fenster mit dem Namen *DependencyView* geöffnet.

Die Darstellung der EXAM-Elemente in dem *DependencyView* soll den Anwender bei dem Erkennen der Fehler unterstützen. Indirekt haben wir im Verlauf der Arbeit den *DependencyView* schon mehrfach kennen gelernt. Das Fenster stellt, in Abhängigkeit zu dem im *VerificationProblemsView* selektierten Eintrag, den Testablauf in Form einer Baumstruktur dar. Dabei werden nur diejenigen Operationen in der Darstellung angezeigt, die in der Operationsmenge der Spezifikation enthalten sind. Dies hat den Vorteil, dass nur die für den Anwender relevanten Informationen in der Darstellung enthalten sind.

Die im Kapitel 6 vorgestellten Verifikationsalgorithmen erlauben, in einem Fehlerfall die Operationsaufrufe anzugeben, die die spezifizierte kausale Abhängigkeit verletzten. Diese Operationen werden in der Baumstruktur zusätzlich markiert.

In unserem Beispiel ergibt sich somit die in Abbildung 8.13 enthaltene Darstellung des EXAM-Elements *TestCaseExample*. In der Baumstruktur werden nur die Operationen der Operationsmenge der Spezifikation

$$S_1 = (initDataLogger \oslash deInitDataLogger)$$

unterhalb des *TestCases* und der jeweiligen *TestSequences* angezeigt. Weiterhin sind in diesem Beispiel alle vier Operationsaufrufe markiert, da alle vier Operationen die spezifizierte Bedingung verletzen.

Als zusätzliche Hilfe für den Anwender können in der Baumstruktur beliebig viele Operationen durch den Anwender selektiert werden. Für jedes selektierte Element wird mit Hilfe eines Kontextmenüs das entsprechende Sequenz- oder Aktivitätsdiagramm geöffnet und die selektierten Operationen in dem Diagramm markiert.

Für dieses Beispiel ist dies in der Abbildung 8.14 für die Operationen *initDataLogger()* in *InitSequence1* und *initDataLogger* in *SubSequence1* dargestellt.

Nach der Korrektur der Testabläufe hat der Anwender entweder die Möglichkeit, die Überprüfung auf Basis der *SpecificationConfiguration* erneut auszuführen oder er kann nur die bei der ersten Überprüfung als fehlerhaft erkannten Elemente erneut überprüfen. Dabei müssen nur die geänderten Elemente in die entsprechende Petri-Netz-Darstellung transformiert werden.

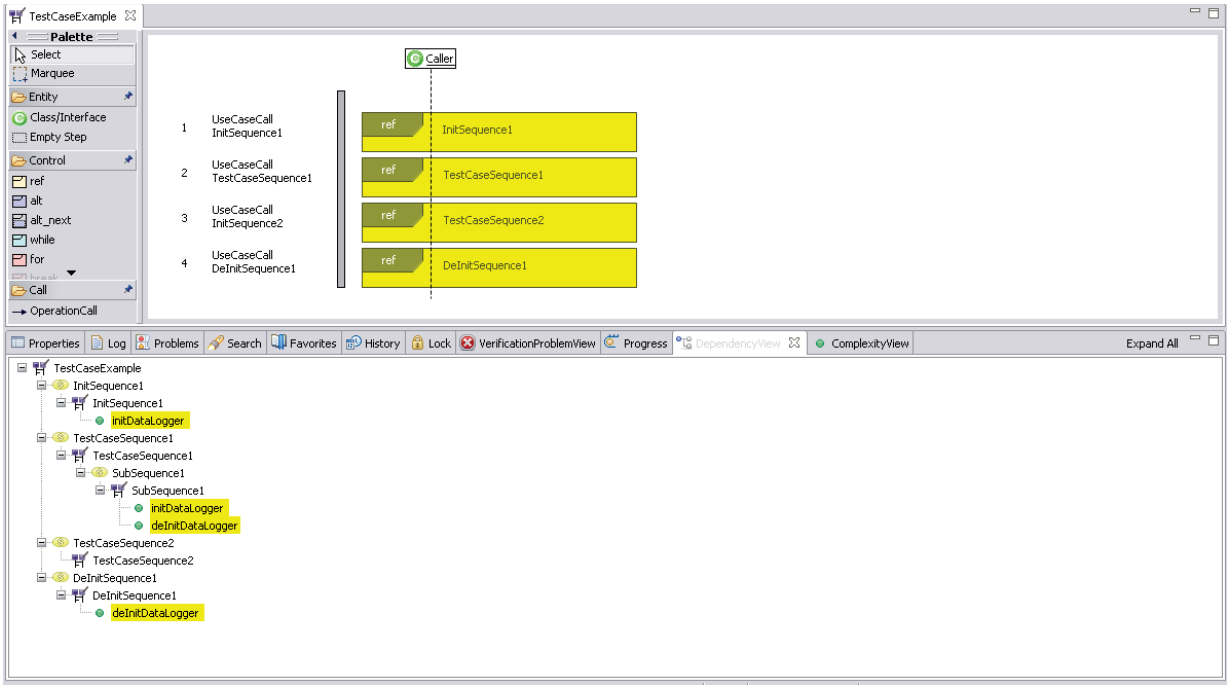

Abbildung 8.13: Ansicht des DependencyViews

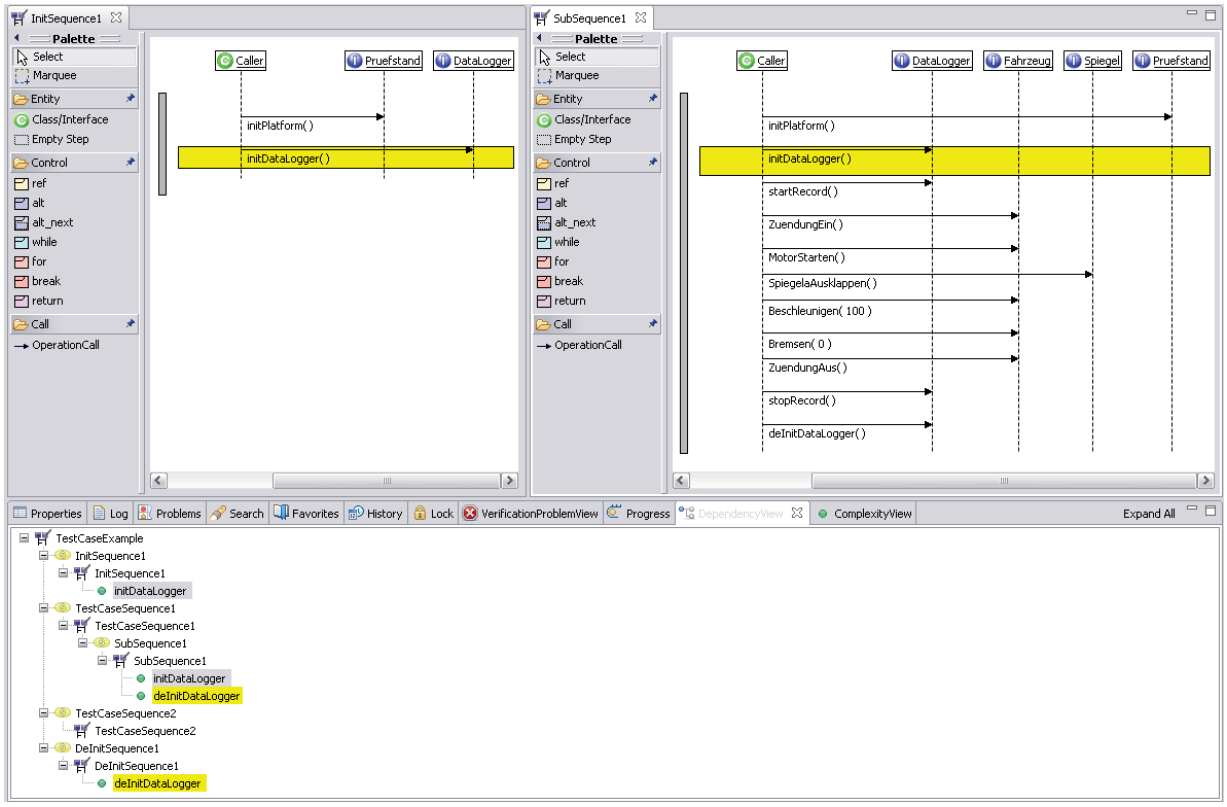

Abbildung 8.14: Hervorhebung der Modellfehler in mehreren Diagrammen

Die dargestellten Erweiterungen des EXAM-Modellers ermöglichen eine einfache Anwendung der Methode im produktiven Einsatz. Mit Hilfe des Prototypen konnten wir die praktische Anwendbarkeit der Methode im Testprozess bei der AUDI AG nachweisen.

9 Zusammenfassung und Ausblick

In der Automobilindustrie hat sich die Entwicklung neuer Fahrzeugfunktionen in den letzten Jahren deutlich gewandelt. Durch die gestiegenen Anforderungen an die Fahrzeugfunktionen, getrieben durch wachsende Kundenansprüche, gesetzliche Vorgaben (z. B. Reduktion der CO_2-Emissionen oder Normen, wie die DIN EN 61508 [ISO09] und die entstehende ISO/DIS 26262 *Road vehicles - Functional safety* [LPP10]) sowie Innovationen in der Unterhaltungselektronik ist die Anzahl und Komplexität der Fahrzeugfunktionen in den letzten Jahren zunehmend angestiegen.

Die Automobilindustrie steht vor der Herausforderung, die Qualität der neuen hochgradig vernetzten Fahrzeugfunktionen in vertretbarer Zeit und zu vertretbaren Kosten abzusichern. Zusätzlich zur steigenden Anzahl und Komplexität der Fahrzeugfunktionen ist die Entwicklung der Fahrzeuge heute durch immer kürzere Entwicklungszyklen, einen hohen Kostendruck, eine zunehmende Parallelisierung der Fahrzeugprojekte und einer Reduzierung der zur Verfügung stehenden Prototypen geprägt. Zur Beherrschung dieser Herausforderungen ist zum einen ein effizienter Testprozess und zum anderen der Einsatz von Simulationstechniken im Entwicklungs- und Absicherungsprozess notwendig.

Im Volkswagen-Konzern hat sich für die Absicherung der Steuergerätefunktionen die Hardware-in-the-Loop-Simulation in Verbindung mit dem Testautomatisierungssystem EXAM etabliert. Die Hardware-in-the-Loop-Simulation erlaubt die Kombination von Simulationsmodellen und realen Steuergeräten. Durch die Simulationsmodelle werden alle nicht als Echtteil verbauten Komponenten möglichst realistisch dargestellt. Dadurch kann die Methode zum einen für das entwicklungsbegleitende Testen der Funktionen als auch für die Komponenten- und Systemtests der Funktionen eingesetzt werden.

Die Steuerung der HIL-Prüfstände im Volkswagen-Konzern erfolgt mit Hilfe des Testautomatisierungssystems EXAM. Die Testfälle werden dabei basierend auf einem UML-Dialekt grafisch modelliert und in einer zentralen Datenbank abgespeichert. EXAM erlaubt es, den Testablauf grafisch zu beschreiben, die Testfälle im Anschluss automatisiert am Prüfstand auszuführen und die Ergebnisse teilautomatisiert zu bewerten.

Der hohe Zeit- und Kostendruck und die zunehmende Parallelisierung der Fahrzeugprojekte bei einer gleichzeitigen Reduzierung der Prototypen erfordert einen verstärkten Einsatz der Hardware-in-the-Loop-Prüfstände für die Funktionsabsicherung. Aufgrund der hohen Anschaffungs- und Betriebskosten der Prüfstände können jedoch nicht für jedes Fahrzeugprojekt beliebig viele HIL-Prüfstände aufgebaut und betrieben werden. In der Regel werden die Prüfstände parallel von mehreren Fahrzeugprojekten verwendet. Nur eine detaillierte Prüfstandszeitplanung ermöglicht die zeitgerechte Ausführung alle benötigten Testfälle. Jede Störung bei der Testausführung, wie

z.B. ein irrtümlicher Abbruch des Testfalls oder eine unnötige Wiederholung, führt zu einer nutzlosen Verwendung der ohnehin sehr limitierten und sehr teuren Prüfstande und hat eine direkte Auswirkung auf den gesamten Funktionsabsicherungsprozess. Die Störung der Testausführung kann zum einen durch einen fehlerhaften Aufbau der Prüfstandshardware, einen Fehler in den echtzeitfähigen Simulationsmodellen oder einem syntaktischen oder semantischen Fehler in den EXAM-Testfällen ausgelöst werden.

Damit auch bei einer weiter steigenden Anzahl und einer zunehmenden Komplexität der Fahrzeugfunktionen die Funktionsabsicherung noch gewährleistet werden kann, müssen die Störungen an den Prüfständen reduziert werden. Anderenfalls können die Fahrzeugfunktionen in der vorgegebenen Zeit nicht in der notwendigen Testtiefe abgesichert werden. Um die Absicherung trotzdem gewährleisten zu können, müsste die Anzahl der Prüfstände und die Anzahl der Prüfingenieure deutlich erhöht werden. Dies würde zu einer nicht unwesentlichen Erhöhung der Kosten für die Absicherung führen. Dies wiederum würde eine Erhöhung des Entwicklungsbudget und somit auch zu einer Preiserhöhung der Fahrzeuge für den Kunden führen.

Analog zur Komplexität der Fahrzeugfunktionen steigt die Komplexität der in EXAM modellierten Testfälle. Dies hat zur Folge, dass die Anzahl der syntaktischen und semantischen Fehler pro Testfall deutlich zugenommen hat. Die häufigsten syntaktischen Fehler in den EXAM-Testfällen sind z.B. eine falsche Variablenbezeichnung mit ungültigen Zeichen oder fehlerhafte Referenzen auf Operationen der Funktionsbibliothek. Unter semantischen Fehlern in den EXAM-Testabläufen haben wir in dieser Arbeit Verletzungen der Aufrufreihenfolge (kausalen Abhängigkeiten) der EXAM-Operationen verstanden. Die kausalen Abhängigkeiten der Operationen ergeben sich zum einen aus der Logik der Fahrzeugfunktionen (z. B. Beschleunigung des Fahrzeugs ist nur nach dem Start des Motors möglich oder die Funktion ACC kann erst auf ein Objekt (vorausfahrendes Fahrzeug) reagieren, wenn das ACC-System vom Fahrer aktiviert wurde) und zum anderen aus der Bedienlogik des Prüfstands (z. B. Starten einer Datenaufzeichnung mit dem Datenlogger ist erst nach der Initialisierung des Datenloggers möglich oder der Zugriff auf den Fehlerspeicher eines Steuergerätes mit Hilfe eines Diagnosetester kann erst nach der Initialisierung des Diagnosesystems erfolgen). Bei der Ausführung fehlerhafter Testfälle sind die Testergebnisse in jedem Fall unbrauchbar und die Testfälle müssen wiederholt werden.

Aufgrund der Größe der Testfälle (mehrere zehntausend Testschritte) können die Fehler nicht mehr ausschließlich durch manuelle Reviews der Testfälle gefunden werden. Eine automatisierte Überprüfung der Testfälle auf syntaktische Fehler ist bereits heute in EXAM integriert. Mit dem in dieser Arbeit vorgestellten Ansatz wird die automatisierte Erkennung von semantischen Fehlern (basierend auf den kausalen Abhängigkeiten der Operationen der Funktionsbibliothek) in den Testfällen ermöglicht.

Wir haben in dieser Arbeit ein Verfahren entwickelt, welches die Störungen an den HIL-Prüfständen durch eine Erkennung und Behebung der semantischen Fehler in den EXAM-Testfällen vor der Ausführung der Testfälle an den Prüfständen fast vollständig beseitigt. Durch die Entkopplung der Überprüfung der Testfälle von den HIL-Prüfstanden entsteht durch das von uns entwickelte Verfahren keine zusätzliche Belastung der Prüfstände. Die von uns entwickelte Methode erlaubt die automatisierte Erkennung semantischer Fehler in den EXAM-Testfällen. Die semantischen Fehler basieren auf fehlerhaften Ausführungsreihenfolgen der EXAM-Operationen in den Testfällen. Der Ansatz basiert auf einer Erweiterung der von [Fid93], [Sim00] und [LS00]

eingeführten Aktionslogik und Petri-Netzen. Durch eine Abbildung der EXAM-Elemente auf Module der Aktionslogik in Form von (standardisierten) Petri-Netzen kann mit diesem Ansatz die kausale Ordnung der Operationen der EXAM-Funktionsbibliothek formal spezifiziert und die Testabläufe auf die Einhaltung der kausalen Abhängigkeiten überprüft werden. Die Überprüfung der Testabläufe erfolgt dabei ohne die Verwendung der Prüfstandshardware und erlaubt somit die Testfälle vor der Ausführung am Prüfstand zu überprüfen.

Mit Hilfe einer prototypischen Umsetzung der beschriebenen Methode als eine Erweiterung des EXAM-Modellers haben wir die praktische Anwendbarkeit der Methode im Entwicklungsprozess bei der AUDI AG nachgewiesen. Durch den praktischen Einsatz der von uns entwickelten Methode konnten wir zeigen, dass diese entscheidend zur Entlastung in der Funktionsabsicherung beiträgt. Die Anzahl der Störungen im Testablauf (bedingt durch semantische Modellierungsfehler) kann deutlich reduziert werden. Durch die automatisierte Überprüfung der Testablaufe vor der Ausführung am Prüfstand kann die teure und limitierte Prüfstandszeit effektiv für die eigentliche Funktionsabsicherung eingesetzt werden.

Bei der Entwicklung der Methode standen wir vor den folgenden Herausforderung:

- Identifikation der Arten von Abhängigkeiten zwischen den Operationen in der Funktionsbibliothek in EXAM.
- Formale Definition der kausalen Abhängigkeiten zwischen den Operationen in der Funktionsbibliothek in EXAM.
- Entwicklung einer Methode und Algorithmen zur Überprüfung der kausalen Abhängigkeiten.
- Überprüfung der Anwendbarkeit der Methode bei der gegebenen Komplexität der EXAM-Testfälle.
- Darstellung der gefundenen Verletzungen der kausalen Abhängigkeiten in den Testfällen für den Anwender.
- Validierung der praktischen Anwendbarkeit und den praktischen Nutzen der Methode.

Mit der von uns in dieser Arbeit entwickelten Methode, haben wir die zuvor aufgezählten Herausforderung gelöst. Ebenso konnten wir die praktische Anwendbarkeit und den praktischen Nutzen der Methode nachweisen.

Die kausalen Abhängigkeiten zwischen den Operationen in der Funktionsbibliothek von EXAM ergeben sich aufgrund unterschiedlicher Bedingungen. Durch eine Analyse der Funktionsbibliothek konnten wir drei Kategorien für die Einordnung der kausalen Abhängigkeiten identifizieren. Die kausalen Abhängigkeiten ergeben sich basierend auf

- domänenspezifischen Bedingungen,
- domänenübergreifenden Bedingungen und
- Bedingungen basierend auf den Modellierungsrichtlinien.

Die einzelnen Bedingungen haben wir im Abschnitt 3.3 im Detail beschrieben.

Viele der kausalen Abhängigkeiten müssen nur unter definierten Voraussetzungen und Rahmenbedingungen in den EXAM-Testfällen erfüllt sein. Daher haben wir als Erweiterung zu den drei zuvor erwähnten Kategorien die drei *Häufigkeitsattribute*

- *Single*,
- *Multiple* und
- *One-To-One*

sowie die drei *Voraussetzungsattribute*

- *May*,
- *Must* und
- *Complete*

definiert. Die Attribute beschreiben unter welchen Voraussetzungen die kausalen Abhängigkeiten in den EXAM-Testfällen einzuhalten sind und wie oft die einzelnen Operationen in den Testfällen enthalten sein dürfen bzw. müssen. Im Abschnitt 6.4 ist jedes Attribut im Detail beschrieben.

Für eine automatisierte Überprüfung der kausalen Abhängigkeiten müssen diese formal spezifiziert werden. In den Arbeiten von [Fid93], [Sim00] und [LS00] ist eine Aktionslogik beschrieben, die es erlaubt kausale Beziehungen zwischen atomaren Aktionen formal zu definieren. Wir haben diesen Ansatz in dieser Arbeit um die *Voraussetzungsattribute* und *Häufigkeitsattribute* erweitert und können damit die kausalen Abhängigkeiten der Operationen in der Funktionsbibliothek formal spezifizieren.

Mit Hilfe des in dieser Arbeit vorgestellten *direkten* und *indirekten Beweisverfahrens* (Definition 5.24 auf Seite 78) ist es möglich, auf Basis der Anzahl der Prozesse der AL-Module zu bestimmen, ob ein Modul ein weiteres Modul *erfüllt*. In diesem Zusammenhang haben wir die Begriffe *erfüllt* (*sound*) und *vollständig* (*complete*) in der Definition 5.22 auf Seite 76 eingeführt.

Den in der Aktionslogik beschriebenen Ansatz haben wir auf die Überprüfung der Testfälle in EXAM übertragen. Durch die Darstellung der Testfälle in Form von AL-Modulen und der von uns beschriebenen Erweiterungen der Aktionslogik können wir die kausalen Abhängigkeiten in den EXAM-Testfällen überprüfen.

Durch die Darstellung der Formeln (Module) der Aktionslogik mit Hilfe von S/T-Petri-Netzen kann die Anzahl der Prozesse mit Hilfe von T-Invarianten bestimmt werden. In Kapitel 6 haben wir im Detail gezeigt, wie mit Hilfe der von uns definierten Erweiterung der Aktionslogik und mit Petri-Netzen überprüft werden kann, ob in den Testabläufen in EXAM, die definierten kausalen Abhängigkeiten zwischen den Operationen der Funktionsbibliothek eingehalten werden.

Die Anwendbarkeit der vorgestellten Methode haben wir in Kapitel 7 mit Hilfe von sechs Spezifikationen und einem vereinfachten realen Testablauf demonstriert. Anhand des Beispiels haben wir gezeigt, wie die kausalen Abhängigkeiten zwischen den Operationen in der Funktionsbibliothek mit Hilfe von AL-Modulen beschrieben werden. Ebenso haben wir gezeigt, wie die Transformation der Testabläufe in eine korrespondierende Petri-Netz-Darstellung erfolgt. Im Anschluss haben wir im Detail dargestellt, wie die Testfälle auf Einhaltung der spezifizierten

Abhängigkeiten mit Hilfe der beschriebenen Algorithmen in Abhängigkeit der *Voraussetzungs-* und *Häufigkeitsattribute* überprüft werden. Alle Verfahren basieren dabei auf der Berechnung der möglichen Schaltfolgen der Netzdarstellungen der Testabläufe, welche die leere Markierung reproduzieren. Die Berechnung dieser Schaltfolgen basiert dabei auf der Bestimmung der realisierbaren T-Invarianten des entsprechenden Petri-Netz. Die T-Invarianten werden dabei durch das Lösen eines linearen homogenen Gleichungssystems, basierend auf der Inzidenzmatrix des Petri-Netzes bestimmt. Aufgrund der Struktur der Netzdarstellungen der Testabläufe konnten wir in dieser Arbeit den in Kapitel 4 angegebenen Gauss-Jordan-Algorithmus einsetzen.

Für die Validierung der Anwendbarkeit der Methode im Testprozess bei der AUDI AG haben wir als Erweiterung des Testautomatisierungssystem EXAM die beschriebene Methode mit Hilfe einer prototypischen Implementierung umgesetzt und produktiv im Testprozess bei der AUDI AG erprobt. In Kapitel 8 haben wir beschrieben wie der EXAM-Anwender bei der Erstellung von Spezifikationen durch den Prototypen unterstützt wird und wie die Ergebnisse der Überprüfung dem Anwender direkt in den Testabläufen in EXAM mit Hilfe des *VerificationProblemsView* und des *DependencyView* dargestellt werden.

Für die Abschätzung des praktischen Einsatzes der Methode spielt die benötigte Ausführungszeit der Überprüfung eine entscheidende Rolle. Dazu haben wir im Abschnitt 8.2 die benötigte Ausführungszeit der Methode anhand verschiedener Beispiele bewertet und ein Verfahren zur Reduktion der benötigten Ausführungszeit vorgestellt. Als Ergebnis können wir zusammenfassend sagen, dass durch die Reduktion der Operationen in den Testabläufen eine Überprüfung von Testabläufen mit mehr als einer Million Elemente effizient möglich ist.

Mit Hilfe dieser Betrachtung konnten wir zeigen, dass die Methode für den praktischen Einsatz im Testprozess geeignet ist. Durch die Analyse der Testabläufe vor der Ausführung am Prüfstand kann die Anzahl der Störungen während des Testablaufs entscheidend reduziert werden. Dadurch wird wertvolle Prüfstandszeit eingespart, welche somit für die Durchführung weiterer Testabläufe zur Verfügung steht. Damit konnten wir die von uns im Abschnitt 3.3 gesteckten Ziele der Arbeit

- die Entwicklung einer geeigneten Methode zur formalen Spezifikation und zur Überprüfung der kausalen Abhängigkeiten der Bibliotheksoperationen in EXAM,
- die Integration der Methode in das Testautomatisierungssystem EXAM und
- die Validierung des praktischen Einsatzes der Methode im Testprozess der AUDI AG

erreichen.

In den nächsten Schritten wird die vorgestellte Methode und der entwickelte Prototyp in das EXAM-Release aufgenommen und im Entwicklungsprozess im Volkswagen-Konzern produktiv eingesetzt. In diesem Zusammenhang wäre eine Ausweitung des Ansatzes auf die Testfallgenerierung mit Hilfe von Benutzungsmodellen (vergleiche [SDGK09]) oder der Generierung von Testfällen mit Hilfe von Testfallschablonen auf Basis einer Wissensdatenbank denkbar. Damit kann sichergestellt werden, dass auch bei generierten Testabläufen die kausalen Abhängigkeiten der Operation der EXAM-Funktionsbibliothek berücksichtigt werden.

Neben dem praktischen Einsatz der Methode sehen wir auch noch weitere wissenschaftliche Fragestellungen. Zum einen ist zu untersuchen, ob die kausalen Abhängigkeiten zwischen den Operationen der Funktionsbibliothek in EXAM auch automatisch ermittelt werden können. Im Bereich der statischen Analyse von Programmen hat Dawson Engler et al. in [EM04] und [ECH+01] Ansätze auf Basis einer statistischen Auswertung der Operationsaufrufe in Softwareprogrammen vorgestellt. Es wäre denkbar, einen ähnlichen Ansatz auch auf den in dieser Arbeit entwickelten Ansatz und auf EXAM zu übertragen. Zum anderen ergibt sich die Fragestellung, ob die von [Fid93], [Sim00] und [LS00] eingeführte Aktionslogik und der von uns entwickelte Ansatz mit entsprechenden Erweiterungen auch in weiteren Anwendungsfällen wie z. B. im Bereich der statischen Analyse von Programmen oder in Verbindung mit klassischen Model-Checking-Verfahren eingesetzt werden kann. Ebenso ist ein Einsatz der vorgestellten Methode für die Überprüfung von kausalen Abhängigkeiten im Bereich der Geschäftsprozessmodellierung denkbar. Die auf Petri-Netzen basierende Modellierungssprache für Geschäftsprozesse *YAWL* (Yet Another Workflow Language, vergleiche [AH02], [RH09] oder [HAAR09]) würde sich z. B. für die Analyse der Anwendbarkeit der vorgestellten Methode eignen.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[AH02] Aalst, W.M.P. van d. ; Hofstede, A.H.M. ter: YAWL: Yet Another Workflow Language / QUT Technical Report, FIT-TR-2002-06, Queensland University of Technology, Brisbane. 2002. – Forschungsbericht

[Ant98] Anton, Howard: *Lineare Algebra: Einführung, Grundlagen, Übungen*. 3. korr. Nachdruck, 1. Auflage. Spektrum Akademischer Verlag, 1998. – ISBN 9783827403247

[aut10a] AUTOSAR: *Automotive Open System Architecture*. `http://www.autosar.org`. Version: 2010. – Letzter Zugriff: 11.04.2011

[aut10b] SPICE User Group: *Automotive SPICE*. `http://www.automotivespice.com/`. Version: 2010. – Letzter Zugriff: 11.04.2011

[BA04] Beck, Kent ; Andres, Cynthia: *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004. – ISBN 9780321278654

[Bau96] Baumgarten, Bernd: *Petri-Netze : Grundlagen und Anwendungen*. 2. Auflage. Spektrum Akademischer Verlag, 1996

[BHS07] Beckert, Bernhard (Hrsg.) ; Hähnle, Reiner (Hrsg.) ; Schmitt, Peter H. (Hrsg.): *Verification of Object-Oriented Software: The KeY Approach*. Springer-Verlag, 2007 (LNCS 4334)

[BK08a] Baier, Christel ; Katoen, Joost-Pieter: *Principles of Model Checking*. The MIT Press, 2008. – ISBN 9780262026499

[BK08b] Bringmann, Eckard ; Krämer, Andreas: Model-Based Testing of Automotive Systems. In: *ICST '08: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*. Washington, DC, USA : IEEE Computer Society, 2008. – ISBN 978–0–7695–3127–4, S. 485–493

[Bor10] Borgeest, Kai: *Elektronik in der Fahrzeugtechnik: Hardware, Software, Systeme und Projektmanagement*. 2., überarbeitete und erweiterte Auflage. Vieweg+Teubner, 2010. – ISBN 9783834805485

[Bos07] Bosch ; Robert Bosch GmbH (Hrsg.): *Autoelektrik / Autoelektronik*. 5. vollst. überarb. u. erw. Auflage. Vieweg+Teubner, 2007. – ISBN 9783528238728

[BPS00] Bush, William R. ; Pincus, Jonathan D. ; Sielaff, David J.: A static analyzer for finding dynamic programming errors. In: *Softw. Pract. Exper.* 30 (2000), June, S. 775–802. – ISSN 0038–0644

[CGP99] Clarke, Edmund M. ; Grumberg, Orna ; Peled, Doron A.: *Model Checking*. The MIT Press, 1999. – ISBN 9780262032704

[CW96] Clarke, Edmund M. ; Wing, Jeannette M.: Formal Methods: State of the Art and Future Directions. In: *ACM Computing Surveys* 28 (1996), S. 626–643

[ECH+01] Engler, Dawson ; Chen, David Y. ; Hallem, Seth ; Chou, Andy ; Chelf, Benjamin: Bugs as deviant behavior: a general approach to inferring errors in systems code. In: *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles* Bd. 35. New York, NY, USA : ACM Press, December 2001. – ISSN 0163–5980, S. 57–72

[Ecl10] Eclipse: *Rich Client Platform*. `http://wiki.eclipse.org/index.php/Rich_Client_Platform`. Version: 2010. – Letzter Zugriff: 11.04.2011

[EM04] Engler, Dawson ; Musuvathi, Madanlal: Static analysis versus software model checking for bug finding. In: *VMCAI*, Springer, 2004, S. 191–210

[Fid93] Fidelak, Manfred: *Integritätsbedingungen in Petri-Netzen*, Universität Koblenz-Landau, Diss., 1993

[Fis02] Fischer, Gerd: *Lineare Algebra: Eine Einführung für Studienanfänger*. 13. Vieweg, 2002

[Gen73] Genrich, H.J.: *Formale Eigenschaften des Entscheidens und Handelns*. Interner Bericht 09/73-11-29, GMD, St. Augustin, 1973

[HAAR09] Hofstede, Arthur H. M. (Hrsg.) ; Aalst, Wil M. P. d. (Hrsg.) ; Adams, Michael (Hrsg.) ; Russell, Nick (Hrsg.): *Modern Business Process Automation: YAWL and its Support Environment*. Springer, 2009. – ISBN 9783642031205

[Haj08] Hajji, Farid: *Das Python-Praxisbuch: Der große Profi-Leitfaden für Programmierer*. 1. Auflage. Addison-Wesley, Muenchen, 2008. – ISBN 9783827325433

[HE01] Hartmut Ehrig, Martin Große-Rhode Felix Cornelius und Philip Z. Bernd Mahr M. Bernd Mahr: *Mathematisch-strukturelle Grundlagen der Informatik*. 2. Auflage. Spektrum Akademischer Verlag, 2001. – ISBN 3540419233

[ISO95] Norm DIN EN ISO 8402 08 1995. *Qualitätsmanagement und Qualitätssicherung - Begriffe*

[ISO04] Norm DIN EN ISO 9001 04 2004. *Software engineering – Guidelines for the application of ISO 9001:2000 to computer software*

[ISO05] Norm DIN EN ISO 9000 12 2005. *Qualitätsmanagementsysteme - Grundlagen und Begriffe*

[ISO06] Norm DIN EN ISO/IEC 15504-5 07 2006. *Information technology – Process Assessment – Part 5: An exemplar Process Assessment Model*

[ISO08a] Norm DIN EN ISO 9001 12 2008. *Qualitätsmanagementsysteme - Anforderungen*

[ISO08b] Norm DIN EN ISO/IEC 12207 02 2008. *Systems and software engineering – Software life cycle processes*

[ISO09] Norm DIN EN ISO/IEC 61508 06 2009. *Funktionale Sicherheit sicherheitsbezogener elektrischer/elektronischer/programmierbarer elektronischer Systeme*

[Jen92] Jensen, Kurt: *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*. Springer, 1992

[KABV07] Koomen, Tim ; Aalst, Leo van d. ; Broekman, Bart ; Vroon, Michiel: *TMap Next - Ein praktischer Leitfaden für ergebnisorientiertes Softwaretesten*. dpunkt Verlag, 2007. – ISBN 9783898644617

[KF09] Kindel, Olaf ; Friedrich, Mario: *Softwareentwicklung mit AUTOSAR: Grundlagen, Engineering, Management in der Praxis*. 1. Auflage. dpunkt Verlag, 2009. – ISBN 9783898645638

[Kif09] Kiffe, Gerhard: *EXAM-Konzeptpapier, Version 3.0*. www.exam-ta.de. Version: 2009. – Letzter Zugriff: 11.04.2011

[Kün09] Künneth, Thomas: *Einstieg in Eclipse 3.5: RCP-, Web- und AJAX-Anwendungen entwickeln, Ant, Refactoring, Debugging, Subversion, CVS, Plug-ins, Subversion, CVS u.v.m (Galileo Computing)*. Galileo Computing, 2009. – ISBN 9783836214285

[Lau73] Lautenbach, Kurt: Exakte Bedingungen der Lebendigkeit für eine Klasse von Petri-Netzen / Berichte der GMD 82, GMD, Bonn, 1973. 1773. – Forschungsbericht

[Lau92] Lautenbach, Kurt: The Reproducibility of the Empty Marking in Place/Transition Nets / Fachbereich Informatik 16/92, Univerität Koblenz-Landau. 1992. – Forschungsbericht

[Lau96] Lautenbach, Kurt: Action Logical Correctness Proving / Gelbe Reihe, Universität Koblenz-Landau, Fachberichte Informatik. 1996. – Forschungsbericht

[Lau02] Lautenbach, Kurt: Reproducibility of the Empty Marking. In: *ICATPN* Bd. 2360, Springer, 2002 (Lecture Notes in Computer Science). – ISBN 3–540–43787–8, S. 237–253

[Lig09] Liggesmeyer, Peter: *Software-Qualität: Testen, Analysieren und Verifizieren von Software*. 2. Auflage. Spektrum Akademischer Verlag, 2009. – ISBN 9783827420565

[LP08] Lutz Priese, Harro W.: *Petri-Netze*. 2. Auflage. Springer Verlag, Berlin, 2008

[LPP10] Löw, Peter ; Pabst, Roland ; Petry, Erwin: *Funktionale Sicherheit in der Praxis: Anwendung von DIN EN 61508 und ISO/DIS 26262 bei der Entwicklung von Serienprodukten*. 1. Auflage. dpunkt Verlag, 2010. – ISBN 9783898645706

[LR94] Lautenbach, Kurt ; Ridder, Hanno: Liveness in Bounded Petri Nets Which Are Covered by T-Invariants. In: *Application and Theory of Petri Nets* Bd. 815, Springer, 1994 (Lecture Notes in Computer Science). – ISBN 3–540–58152–9, S. 358–375

[LS00] Lautenbach, Kurt ; Simon, Carlo: Verification in a Logic of Action. In: *7th Workshop on Algorithms and Tools for Petri Nets*, 2000, S. 19–24

[McC76] McCabe, Thomas J.: A Complexity Measure. In: *IEEE Trans. Software Eng.* 2nd edition (1976), Nr. 4, S. 308–320

[Mei07] Meister, Andreas: *Numerik linearer Gleichungssysteme: Eine Einführung in moderne Verfahren. Mit MATLAB-Implementierung von C. Vömel*. 3., überarbeitete Auflage. Vieweg+Teubner, 2007. – ISBN 9783834804310

[Mei11] Meister, Andreas: *Numerik linearer Gleichungssysteme: Eine Einführung in moderne Verfahren. Mit MATLAB-Implementierungen von C. Vömel*. 4., überarbeitete Auflage. Vieweg+Teubner, 2011

[MHDZ07] Müller, Markus ; Hörmann, Klaus ; Dittmann, Lars ; Zimmer, Jörg: *Automotive SPICE in der Praxis: Interpretationshilfe für Anwender und Assessoren*. 1. Auflage. Dpunkt Verlag, 2007. – ISBN 9783898644693

[MSK+09] Miegler, Maximilian ; Schieber, Reinhard ; Kern, Andreas ; Ganslmeier, Thomas ; Nentwig, Mirko: Hardware-in-the-Loop-Test von vorausschauenden Fahrerassistenzsystemen. In: *ATZelektronik* 04 (2009), Nr. 05, S. 14–19

[MW09] Manfred Wolff, Peter Hauck und Wolfgang K.: *Mathematik für Informatik und Bio-Informatik*. 1. Auflage. Spektrum Akademischer Verlag, 2009. – ISBN 3540205217

[NEBE05] Neunzert, Helmut ; Eschmann, Winfried G. ; Blickensdörfer-Ehlers, Arndt: *Analysis 2: Mit einer Einführung in die Vektor- und Matrizenrechnung. Ein Lehrbuch- und Arbeitsbuch (Springer-Lehrbuch)*. 3. Auflage. Springer, Berlin, 2005. – ISBN 9783540641186

[Pet62] Petri, Carl A. (Hrsg.): *Kommunikation mit Automaten*. Bonn : Schriften des Rheinisch-Westfälischen Institutes für instrumentelle Mathematik an der Universität Bonn, 1962

[Pre03] Pretschner, Alexander: *Zum modellbasierten funktionalen Test reaktiver Systeme*, Universität München, Diss., 2003

[RBGW10] Rossner, Thomas ; Brandes, Christian ; Götz, Helmut ; Winter, Mario: *Basiswissen Modellbasierter Test*. 1. Auflage. dpunkt Verlag, 2010. – ISBN 9783898645898

[Rei08] Reif, Konrad: *Automobilelektronik: Eine Einführung für Ingenieure*. 3., überarbeitete Auflage. Vieweg+Teubner, 2008. – ISBN 9783834804464

[Rei09] Reichert, Stefan: *Eclipse RCP im Unternehmenseinsatz: Verteilte Anwendungen entwerfen, entwickeln, testen und betreiben*. 1. Auflage. dpunkt Verlag, 2009. – ISBN 9783898645737

[Rei10] Reisig, Wolfgang: *Petrinetze: Modellierungstechnik, Analysemethoden, Fallstudien*. Vieweg+Teubner, 2010

[RH09] Russell, Nick ; Hofstede, A.H.M. ter: newYAWL: Towards Workflow 2.0. In: Jensen, K. (Hrsg.) ; Aalst, W. van d. (Hrsg.): *Special issue of LNCS Transactions on Petri Nets and Other Models of Concurrency (ToPNoC) II on Concurrency in Process-Aware Information Systems volume 5460 of Lecture Notes in Computer Science*, pages 79-97, 2009

[RHQ+05] Rupp, Chris ; Hahn, Jürgen ; Queins, Stefan ; Jeckle, Mario ; Zengler, Barbara: *UML2 glasklar: Praxiswissen für die UML-Modellierung und -Zertifizierung*. Hanser Verlag, 2005

[Sax08] Sax, Eric: *Automatisiertes Testen Eingebetteter Systeme in der Automobilindustrie*. Hanser Fachbuch, 2008. – ISBN 9783446416352

[Sch00] Schöning, Uwe: *Logik für Informatiker*. 5. Auflage. Spektrum Akademischer Verlag, 2000. – ISBN 3827410053

[Sch02] Schneeweiss, Winfrid G.: *Petri-Netz-Bilder-Buch: Eine elementare Einführung in die beste bildliche Darstellung zeitlicher Veränderungen*. LiLoLe-Verlag, 2002. – ISBN 9783934447059

[Sch07] Schneider, Kurt: *Abenteuer Softwarequalität: Grundlagen und Verfahren für Qualitätssicherung und Qualitätsmanagement*. 1. Auflage. Dpunkt Verlag, 2007. – ISBN 9783898644723

[Sch08] Schol, Oliver: *Der neue Audi Q5: Entwicklung und Technik*. 1. Auflage. Vieweg+Teubner, 2008. – ISBN 9783834806048

[SD04] Schieber, Reinhard ; Derichsweiler, Frank: Testautomatisierung in der Automobilindustrie. In: *SIGS-DATACOM, Objekt Spektrum*, April 2004

[SDGK09] Siegl, Sebastian ; Dulz, Winfried ; German, Reinhard ; Kiffe, Gerhard: Model-Driven Testing based on Markov Chain Usage Models in the Automotive Domain. In: *LAAS-CNRS 7 (Hrsg.) : Proc. of the 12th European Workshop on Dependable Computing*, EWDC 2009, Toulouse, France, 14-15 May 2009

[Sim00] Simon, Carlo: *A Logic of Actions and Its Application to the Development of Programmable Controllers*, Universität Koblenz-Landau, Diss., 2000

[SL05] Spillner, Andreas ; Linz, Tilo: *Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester - Foundation Level nach ISTQB-Standard*. 3., überarb. Auflage. Dpunkt Verlag, 2005. – ISBN 9783898643580

[SL10] Spillner, Andreas ; Linz, Tilo: *Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester - Foundation Level nach ISTQB-Standard*. 4., überarb. u. akt. Auflage. dpunkt Verlag, 2010. – ISBN 9783898646420

[SVEH07] Stahl, Thomas ; Völter, Markus ; Efftinge, Sven ; Haase, Arno: *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. 2., aktualisierte und erweiterte Auflage. Dpunkt Verlag, 2007. – ISBN 9783898644488

[SZ04] Schäuffele, Jörg ; Zurawka, Thomas: *Automotive Software Engineering*. Vieweg, 2004. – ISBN 3528110406

[SZ10] Schäuffele, Jörg ; Zurawka, Thomas: *Automotive Software Engineering: Grundlagen, Prozesse, Methoden und Werkzeuge effizient einsetzen*. 4., überarbeitete und erweiterte Auflage. Vieweg+Teubner, 2010. – ISBN 9783834803641

[TD11] Thiel, Sebastian ; Derichsweiler, Frank: Petri Net Based Verification of Causal Dependencies in Electronic Control Unit Test Cases. In: *Proceedings of the 35th Annual IEEE International Computer Software and Applications Conference, COMPSAC 2011, Munich, Germany, 18-21 July 2011*, IEEE Computer Society, 2011

[Thi09] Thiel, Sebastian: *Testautomatisierung mit EXAM und Hardware-in-the-Loop-Simulation zur automatisierten Funktionserprobung von Steuergeräten*. TEMEA Workshop - Test eingebetteter Steuergeräte-Software der Automobilindustrie. `http://www.temea.org/media/workshop/thiel_audi_exam.pdf`. Version: 09 2009. – Letzter Zugriff: 11.04.2011

[Thi10] Thiel, Sebastian: Verification of Functional ECU Test Cases. In: *Gesellschaft für Informatik, Petri Net Newsletter* 77 (2010), S. 11–20

[TZ08] Thiel, Sebastian ; Zitterell, Dirk: EXtended Automation Method (EXAM) zur automatisierten Funktionserprobung von Steuergeräten in der Automobilindustrie. In: *Lecture Notes in Informatics, GI Jahrestagung Band 2*, 2008, S. 625–630

[Val03] Valk, Rüdiger Claude Girault und (Hrsg.): *Petri Nets for Systems Engineering: A Guide to Modelling, Verification, and Application*. 1. Auflage. Springer-Verlag, 2003

[VHBP00] Visser, Willem ; Havelund, Klaus ; Brat, Guillaume ; Park, Seungjoon: Model checking programs. In: *Automated Software Engineering Journal*, Press, 2000, S. 3–12

[vmo05] V-Modell: *Entwicklungsstandard für IT-Systeme des Bundes, Vogehensmodell Kurzbeschreibung*. `http://www.v-modell.iabg.de`. Version: 1997, 2005. – Letzter Zugriff: 11.04.2011

[Wal01] Wallmüller, Ernest: *Software-Qualitätsmanagement in der Praxis: Software-Qualität durch Führung und Verbesserung von Software-Prozessen*. 2., völlig überarb. Auflage. Hanser Fachbuch, 2001. – ISBN 9783446213678

[WHW09] Winner, Hermann (Hrsg.) ; Hakuli, Stephan (Hrsg.) ; Wolf, Gabriele (Hrsg.): *Handbuch Fahrerassistenzsysteme Grundlagen, Komponenten und Systeme für aktive Sicherheit und Komfort*. 1. Auflage. Vieweg+Teubner, 2009. – ISBN 9783834802873

[ZS08] Zimmermann, Werner ; Schmidgall, Ralf: *Bussysteme in der Fahrzeugtechnik: Protokolle und Standards. Praxis/ATZ/MTZ-Fachbuch*. 3., akt. u. erw. Auflage. Vieweg+Teubner, 2008. – ISBN 9783834804471

[ZT10] Zitterell, Dirk ; Thiel, Sebastian: Automatisierter funktionaler Steuergerätetest mit der EXtended Automation Method (EXAM). In: *Lecture Notes in Informatics, GI Jahrestagung Band 2*, 2010, S. 351–356