Konsistenzsicherung von Anforderungen und Architekturen

Dissertation

zur Erlangung des Grades eines Doktors
der Naturwissenschaften (Dr. rer. nat.)

vorgelegt von
Björn Schindler

Hauptberichterstatter:
Prof. Dr. Andreas Rausch
Berichterstatter:
Prof. Dr. Jörg Müller

Kurzfassung

Die Anforderungserhebung und der Architekturentwurf eines Softwareentwicklungsprojekts sind für die erfolgreiche Entwicklung hochqualitativer Softwaresysteme von besonderer Wichtigkeit. Das Ziel des Entwurfs ist die Entwicklung einer Architektur, die die gestellten Anforderungen an das Softwaresystem erfüllt. Anforderungen und Architekturen werden in der Realität zumeist iterativ und evolutionär entwickelt, da beispielsweise viele Anforderungen erst während des Architekturentwurfs aufgedeckt werden. Ein fundamentales Problem hierbei ist die Entstehung von Inkonsistenzen, die zu einer fehlerhaften Berücksichtigung von Anforderungen und folglich zu unerfüllten Anforderungen führen. Aktuelle modellbasierte Ansätze erlauben eine eindeutige und formale Beschreibung von Anforderungen und Architekturen. Eine Automatisierung der Konsistenzsicherung dieser Modelle würde das Problem der Entstehung von Inkonsistenzen lösen. Für Strukturmodelle kann dies auf einfache Weise durch ein Metamodell mit zusätzlichen formalen Konsistenzbedingungen erfolgen. Eine Automatisierung der Konsistenzsicherung der Verhaltensmodelle ist eine Herausforderung. Szenarienbasierte Verhaltensbeschreibungen wie beispielsweise Sequenzdiagramme sind für die Beschreibung von Anforderungen geeignet. Für den Architekturentwurf sind zustandsbasierte Verhaltensbeschreibungen wie beispielsweise höhere Petrinetze geeignet. Eine Automatisierung der Konsistenzsicherung von Anforderungen und Architekturen kann folglich durch eine Automatisierung der Konsistenzsicherung szenarienbasierter und zustandsbasierter Modelle erreicht werden. Eine Konsistenzsicherung derartiger Modelle ist problematisch, da zwischen den Verhaltensbeschreibungen eine Verfeinerungsbeziehung besteht und häufig eine Turing-vollständige Sprache für die Beschreibung der Architektur erforderlich ist.

Die in dieser Arbeit vorgestellte Lösung dieses Problems im Kontext der Konsistenzsicherung von Anforderungen und Architekturen besteht darin, eine entscheidbare und effiziente Konsistenzüberprüfung durch Syntaxeinschränkungen der Verhaltensmodelle zu ermöglichen. Durch die Einschränkung der Syntax wird die Ausdrucksmächtigkeit der Modelle nicht verringert. Vielmehr werden die Variationen zur Beschreibung eines Sachverhalts eingeschränkt und Regeln für den Zusammenhang der szenarienbasierten und zustandsbasierten Modelle festgelegt. Ein Ergebnis dieser Arbeit ist die Definition eines allgemeinen szenarienbasierten Modells für die Beschreibung von Anforderungen und eines zustandsbasierten Modells für die Beschreibung von Architekturen. Diese Modelle haben die erforderliche Ausdrucksmächtigkeit für die Beschreibung von Anforderungen und Architekturen sowie der Automatisierung der Konsistenzsicherung im Detail. Zudem werden Regeln für den Zusammenhang dieser Modelle definiert. Ein weiteres wichtiges Ergebnis ist die Definition der Syntaxeinschränkungen und Konsistenzbedingungen, die eine entscheidbare und effiziente Konsistenzüberprüfung bei vollständiger Berücksichtigung der Ausführungssemantik der Modelle ermöglichen. Eine abschließende Betrachtung zeigt, dass dieser Ansatz zur Automatisierung der Konsistenzsicherung von Anforderungen und Architekturen geeignet ist.

Danksagung

Hiermit möchte ich mich bei allen herzlich bedanken, die mir direkt oder indirekt bei der Erstellung dieser Arbeit geholfen haben.

An erster Stelle bedanke ich mich bei Prof. Dr. Andreas Rausch, der mir die Möglichkeit gab, bei interessanten und industrienahen Projekten mitzuwirken. Im Rahmen dieser Projekte gelang es mir letztendlich, die in dieser Arbeit behandelte und grundlegende Problemstellung zu erkennen. Seine zahlreichen Tipps, Anregungen und kritischen Kommentare halfen mir sehr, die Ergebnisse meiner Arbeit zu entwickeln und zu verbessern. Durch große Freiräume in der Erledigung gestellter Aufgaben sowie durch viele Aktionen am Lehrstuhl wie Kickerturniere und gemeinsame Frühstücke mit anschließender Küchendiskussion wurde ein angenehmes Arbeitsklima geschaffen.

Auch möchte ich mich herzlich bei den Mitstreitern in den Projekten ReqBw und GloSE bedanken, da in diesen Projekten die Idee dieser Arbeit entstand. Begonnen bei Michael Deynet und Sabine Niebuhr mit denen ich zusammen die ersten Jahre im Projekt ReqBw bestritten habe. Und weitergeführt bei Marcel Ibe, Joachim Schramm und Martin Vogel, die im weiteren Verlauf dazu stießen. Die gemeinsamen Fahrten nach Koblenz zu den Projekttreffen am IT-AmtBw und die Fahrten nach Mannheim zu den Lehrgängen am BAkWVT bleiben mir in besonderer Erinnerung. In den Projekttreffen von ReqBw kam es häufig zu ausufernden Diskussionen. An dieser Stelle auch vielen Dank an Eric Saas (IT-AmtBw), der diese Diskussionen gelassen hinnahm.

Einen ganz besonderen Dank spreche ich meiner Frau Ketevan Schindler aus, die mich durchgängig unterstützt hat. So ging sie nicht nur ihrem Studium nach, sondern übernahm während der Erstellung dieser Dissertation auch zu großen Teilen die Kinderbetreuung unseres Sohnes Konstantin. Noch einmal einen ganz besonderen Dank für die hierdurch geschaffenen Freiräume.

Abschließend bedanke ich mich bei allen übrigen Angestellten und Kollegen des Lehrstuhls von Prof. Dr. Andreas Rausch, die während der Erstellung dieser Dissertation stets für eine angenehme sowie unterhaltsame Arbeitszeit gesorgt haben und mir immer mit Rat und Tat zur Seite standen.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Für die erfolgreiche Entwicklung hochqualitativer Softwaresysteme sind die Phasen Anforderungserhebung und Architekturentwurf im Entwicklungsprojekt entscheidend [40][74]. Beide Aktivitäten hängen eng miteinander zusammen und werden in der Realität zumeist iterativ durchgeführt [56]. Das Ziel des Entwurfs ist die Entwicklung einer Architektur, die die gestellten Anforderungen an das Softwaresystem erfüllt. Bei der Entwicklung der Architektur kann beispielsweise der Bedarf zusätzlicher Anforderungen festgestellt werden. Diese können wiederum Änderungen an der Architektur erfordern. Umgekehrt wirken sich Einschränkungen in der Architektur häufig auf die Umsetzbarkeit der Anforderungen aus. Eine Änderung einer früh in diesem iterativen und evolutionären Entwicklungsprozess getroffenen Entscheidung ist mit hohen Kosten verbunden, wenn die Änderung in späten Phasen erfolgt. Daher sind frühe Entscheidungen für den Projekterfolg von besonderer Wichtigkeit.

Ein Entwicklungsprozess, der diese iterative und evolutionäre Entwicklung von Anforderungen und Architekturen realitätsnah beschreibt, ist das *Twin Peaks*-Modell. Dieses Modell wurde erstmals von Nuseibeh [56] eingeführt. Bei diesem Entwicklungsmodell werden die Aktivitäten der Anforderungserhebung und des Architekturentwurfs in einen iterativen und evolutionären Softwareentwicklungsprozess integriert. Anforderungen und Architekturen haben in diesem Modell den selben Stellenwert und werden iterativ sowie evolutionär weiterentwickelt. Dies wird durch zwei Bergspitzen (*Twin Peaks*) symbolisiert (siehe Abb. 1.1). Ein fundamentales Problem bei der iterativen und evolutionären Entwicklung von Anforderungen und Architekturen ist die Entstehung von Inkonsistenzen, die zu einer fehlerhaften Berücksichtigung von Anforderungen des zu entwickelnden Systems und folglich zu unerfüllten Anforderungen führen [40][74]. Eine Automatisierung der Konsistenzsicherung würde dieses Problem lösen. Hierfür müssen die Konsistenzbedingungen zwischen Anforderungen und Architekturen formal definiert werden. Dies macht eine formale Definition einer konkreten Technik für die Beschreibung von Anforderungen und Architekturen erforderlich.

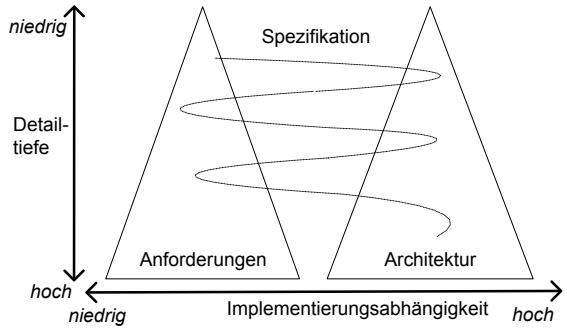


Abbildung 1.1: Twin Peaks-Modell nach Nuseibeh [56]

Ein konkreter Ansatz zur Beschreibung von Anforderungen und Architekturen sollte die folgenden allgemeinen Richtlinien einhalten [40]:

- Anforderungsbeschreibungen sollten eindeutig und vollständig sein, um Fehlinterpretationen zu vermeiden. Bei der Erhebung sollten die Interessenvertreter integriert werden. Hierfür sollten die Anforderungsbeschreibungen eindeutig und vollständig sowie leicht verständlich sein. Diese Richtlinien werden auch von Nuseibeh genannt [57].

- Die Architektur eines Softwaresystems sollte beschreiben, wie Anforderungen von der Architektur erfüllt werden. Beim Architekturentwurf werden abstrakte Beschreibungen fortlaufend verfeinert. Die zusätzlichen Details müssen hierbei von den Anforderungen getrennt werden, um eine Prüfung der Erfüllung von Anforderungen zu ermöglichen. Architekturbeschreibungen müssen zudem auch vollständig und detailliert genug für eine Implementierungsvorgabe sein.

Im Rahmen des Forschungsprojekts *Seamless Requirements Engineering Approach with Respect to the NATO Architectural Framework (NAF) and the V-Modell XT Bw* (ReqBw) haben wir den modellbasierten Ansatz CREATE [40] entwickelt, der die Entwicklung von Anforderungen und Architekturen im Sinne von *Twin Peaks* unterstützt und die genannten Richtlinien erfüllt. Der Ansatz wurde iterativ weiterentwickelt und im Rahmen von realen Entwicklungsprojekten im praktischen Einsatz getestet. Die Beschreibungstechnik des Ansatzes ist formal definiert. Des Weiteren wurden Konsistenzbedingungen zwischen Anforderungen und Architekturen definiert, die die Konsistenzsicherung erleichtern. Doch aktuell existiert kein Verfahren zur Automatisierung der Konsistenzsicherung von Anforderungs- und Architekturmodellen. Eine automatische Konsistenzsicherung der Strukturmodelle kann auf eine einfache Weise durch ein Metamodell mit zusätzlichen formalen Konsistenzbedingungen erfolgen, da die Strukturmodelle auf Typebene miteinander in Beziehung gesetzt werden können [74]. Dies ist beispielsweise der Fall beim modellbasierten Ansatz CREATE. Eine Automatisierung der Konsistenzsicherung der Verhaltensmodelle, insbesondere mit der vollständigen Berücksichtigung der Ausführungssemantik, ist eine Herausforderung [74].

Szenarienbasierte Verhaltensbeschreibungen im Sinne von Liang [51] wie beispielsweise Sequenzdiagramme sind für die Beschreibung von Anforderungen geeignet [40] und beschreiben Interaktionen zwischen Objekten. Für den Architekturentwurf sind zustandsbasierte Verhaltensbeschreibungen im Sinne von Liang [51] wie beispielsweise höhere Petrinetze [64][42] geeignet [74] und beschreiben das Verhalten eines Objekts vollständig. Eine Automatisierung der Konsistenzsicherung von Anforderungen und Architekturen kann folglich durch eine Automatisierung der Konsistenzsicherung von szenarien- und zustandsbasierten Modellen erreicht werden. Die Konsistenzsicherung szenarienbasierter Anforderungsmodelle und zustandsbasierter Architekturmodelle ist problematisch, da zwischen den Verhaltensbeschreibungen eine Verfeinerungsbeziehung besteht und häufig eine Turing-vollständige Sprache für die Beschreibung der Architektur erforderlich ist. Eine einfache Automatisierung der Konsistenzsicherung kann über Konsistenzbedingungen auf Syntaxebene erreicht werden [74]. Eine Herausforderung ist jedoch die Automatisierung der Konsistenzsicherung, die die Ausführungssemantik der Verhaltensmodelle vollständig berücksichtigt.

1.2 Ziele und Ergebnisse

Ziel dieser Arbeit ist die Entwicklung eines Ansatzes, der eine Automatisierung der Konsistenzsicherung von szenarienbasierten Anforderungsmodellen und zustandsbasierten Architekturmodellen ermöglicht. Der Ansatz muss bei Einhaltung der genannten Richtlinien für die iterative und evolutionäre Entwicklung von Anforderungen und Architekturen einsetzbar sein. Hierfür muss der Ansatz den folgenden Anforderungen genügen [74]:

- Die Erfüllung von Anforderungen muss überprüfbar sein. Daher müssen Lösungsdetails der Architektur von den Anforderungen getrennt werden. Anforderungen und Architekturen haben folglich eine Verfeinerungsbeziehung.
- In den wichtigen frühen Entwicklungsphasen sind die Architekturbeschreibungen abstrakt und Abhängigkeiten zwischen Architekturbestandteilen sind nur informell beschrieben. Abhängigkeiten zwischen Architekturbestandteilen können daher im Allgemeinen nicht automatisiert erkannt werden. Der Ansatz zur Automatisierung der Konsistenzsicherung muss an diesen Stellen folglich manuelle Entscheidungen zulassen.
- Um eine detaillierte Beschreibung der Architektur für Implementierungsvorgaben zu ermöglichen, muss der modellbasierte Ansatz eine Turing-vollständige Sprache zur Beschreibung der Architektur zur Verfügung stellen. Die automatische Konsistenzsicherung muss trotz der Turing-vollständigen Verhaltensbeschreibung berechenbar sein. Für die Einsetzbarkeit bei großen Softwaresystemen sollte die automatische Konsistenzsicherung zudem effizient durchgeführt werden können.

Die Lösung dieser Arbeit besteht darin, eine entscheidbare und effiziente Konsistenzüberprüfung durch Syntaxeinschränkungen der szenarienbasierten und zustandsbasierten Modelle zu ermöglichen. Durch die Einschränkung der Syntax wird die Aus-

drucksmächtigkeit der Modelle nicht verringert. Vielmehr werden die Variationen zur Beschreibung eines Sachverhalts eingeschränkt und Regeln für den Zusammenhang der Modelle festgelegt. Die Konsistenzüberprüfung berücksichtig die Ausführungssemantik der Verhaltensmodelle vollständig. Aus diesem Ansatz ergeben sich die folgenden wesentlichen Ergebnisse dieser Arbeit:

- Im Rahmen der Arbeit wird ein allgemeines szenarienbasiertes Modell für die Beschreibung von Anforderungen und ein zustandsbasiertes Modell für die Beschreibung von Architekturen definiert. Diese Modelle haben die erforderliche Ausdrucksmächtigkeit für eine formale Beschreibung von Anforderungen und Architekturen sowie der Automatisierung der Konsistenzsicherung im Detail. Zudem werden Regeln für den Zusammenhang dieser Modelle definiert.
- Syntaxeinschränkungen werden definiert, die eine entscheidbare und effiziente Konsistenzüberprüfung ermöglichen. Hierbei wird die Ausdrucksmächtigkeit der Modelle nicht verringert. Vielmehr werden die Variationen zur Beschreibung eines Sachverhalts eingeschränkt.
- Konsistenzbedingungen zwischen dem szenarienbasierten Anforderungsmodell und dem zustandsbasierten Architekturmodell werden definiert, die die Ausführungssemantik dieser Modelle vollständig berücksichtigen und effizient geprüft werden können.

1.3 Struktur der Ausarbeitung

In Kapitel 2 werden die verwendeten Begriffe aus dem Bereich der Softwareentwicklung definiert und die für diese Arbeit relevanten Techniken und Verfahren betrachtet. Kapitel 3 beinhaltet eine ausführliche Beschreibung des in [40] kurz eingeführten modellbasierten Ansatzes CREATE, der sich zur Automatisierung der Konsistenzsicherung von Anforderungen und Architekturen eignet. Hierbei wird die Beschreibungstechnik in allen Details eingeführt. Kapitel 4 zeigt die Problemstellung im Kontext der Automatisierung der Konsistenzsicherung anhand eines Fallbeispiels. Anschließend wird die Problemstellung auf ein allgemeines Modell abgebildet. Bestehende Ansätze zur Konsistenzsicherung von Modellen werden in Hinblick ihrer Eignung zur Konsistenzsicherung von Anforderungen und Architekturen betrachtet. Hieraus wird die resultierende Aufgabenstellung abgeleitet. In Kapitel 5 wird die Lösung vorgestellt. Diese besteht darin, eine entscheidbare und effiziente Konsistenzüberprüfung von szenarienbasierten Anforderungsmodellen und zustandsbasierten Architekturmodellen über eine Einschränkung der Syntax des zustandsbasierten Modells zu erreichen. In Kapitel 6 wird eine mögliche Implementierung des Lösungsansatzes durch ein *Ecore*-Metamodell [84] und Syntaxeinschränkungen in der *Epsilon Validation Language* [48] beschrieben. Kapitel 7 beinhaltet die Beschreibung der Evaluation des entwickelten Ansatzes anhand dreier Anwendungsszenarien. In Kapitel 8 erfolgt eine Zusammenfassung der erreichten Ergebnisse und eine Beschreibung der offenen Punkte für zukünftige Arbeiten.

Kapitel 2

Grundlegende Verfahren und Techniken

Der in dieser Arbeit beschriebene Ansatz basiert auf etablierten Ansätzen und Begriffen des *Software Engineering* [80]. Dieses Kapitel beinhaltet eine Beschreibung der für diese Arbeit wesentlichen Ansätze und Begriffe. Zuerst erfolgt eine Beschreibung grundlegender Softwareentwicklungsprozesse anhand von Vorgehensmodellen. Anschließend werden Modelle zur Beschreibung von Softwaresystemen und das Konzept der modellbasierten Softwareentwicklung vorgestellt. Abschließend erfolgt eine Beschreibung gängiger Verfahren zur Konsistenzsicherung von Modellen.

2.1 Vorgehensmodelle der Softwareentwicklung

In dieser Arbeit wird die Definition von Sommerville [80] für die Beschreibung von Softwareentwicklungsprozessen verwendet. Für die Entwicklung eines Softwaresystems sind eine Menge von Tätigkeiten erforderlich, die mit Ereignissen zusammenhängen. Die Menge dieser Tätigkeiten und Ereignisse wird als Softwareentwicklungsprozess bezeichnet. Ein Vorgehensmodell beschreibt ein Softwareentwicklungsprozess vereinfacht. Ein Vorgehensmodell kann hierbei unter anderem folgende Aspekte eines Softwareentwicklungsprozesses beschreiben: Arbeitsablauf, Datenfluss, Aktivitäten, Rollen und Aktionen. Im Folgenden werden allgemeine Typen von Vorgehensmodellen beschrieben. Auf konkrete Ausprägungen von Vorgehensmodellen wie beispielsweise das V-Modell-XT [67] wird an dieser Stelle nicht eingegangen. Allgemeine Vorgehensmodelltypen sind hierbei das Wasserfallmodell, die evolutionäre Entwicklung, die komponentenbasierte Entwicklung und die iterative Entwicklung. Anschließend werden das *Spiral Life Cyle*-Modell und das *Twin Peaks*-Modell vorgestellt. Diese sind Varianten der iterativen Entwicklung.

2.1.1 Wasserfallmodell

Das Wasserfallmodell ist ein Vorgehensmodell bei dem die Phasen der Softwareentwicklung aufeinander folgend durchlaufen werden [71]. Im Modell beginnt eine neue Phase, wenn die vorangehende abgeschlossen wurde [80]. Die einzelnen Phasen des Modells werden daher stufenförmig dargestellt (siehe Abb. 2.1). Hierdurch entspricht die Darstellung der eines Wasserfalls. Die typischen Phasen eines Softwareentwicklungsprojekts lassen sich hierbei wie folgt zusammenfassen [80]:

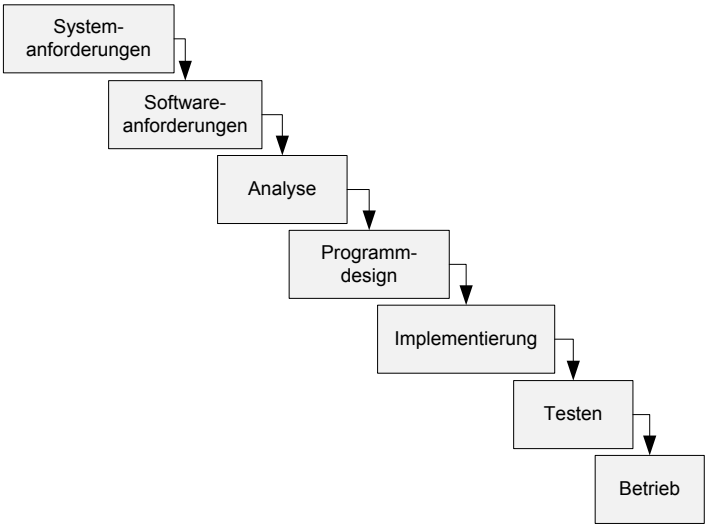


Abbildung 2.1: Das Wasserfallmodell nach Royce [71]

1. Anforderungsdefinition - Diese Phase wird auch als Requirements Engineering (RE) [57] oder auch Anforderungserhebung bezeichnet. Interessenvertreter und deren Bedürfnisse werden identifiziert und als Anforderungen dokumentiert [56]. Die Dokumentation der Anforderungen erfolgt in einer Form, die leicht analysierbar ist und mit Interessenvertretern abgestimmt werden kann.

2. System- und Softwareentwurf - Anforderungen werden hierbei auf die Hard- und Softwaresysteme aufgeteilt und es wird eine allgemeine Systemarchitektur festgelegt . Diese Phase wird häufig auch als Design (Entwurf) [9] oder Architekturentwurf bezeichnet. Beim Architekturentwurf werden große Systeme in Subsysteme unterteilt. Die grundlegenden abstrakten Softwaresysteme und deren Beziehungen untereinander werden identifiziert und dokumentiert.

3. Implementierung und Komponententest - Die im System- und Softwareentwurf dokumentierten Softwaresysteme werden entsprechend der Architektur als Programme und Programmeinheiten umgesetzt. In Form von Komponententests wird geprüft, ob die Einheiten ihre Spezifikation erfüllen.

4. Integration und Systemtest - Einzelne Programme und Programmeinheiten werden integriert. Das Zusammenspiel der integrierten Systeme wird in Form von Integrationstests geprüft. Hierbei wird geprüft, ob die Systeme ihre Spezifikation erfüllen.

5. Betrieb und Wartung - Das System wird installiert und in Betrieb genommen. Zum Betrieb gehört häufig die Wartung des Systems, bei der auftretende Fehler behoben werden.

In jeder Phase wird ein Dokument erstellt, welches nach Abschluss abgenommen wird [80]. Dies stellt eine durchgängige Dokumentation sicher. In der Praxis werden die Phasen häufig parallel durchgeführt und es erfolgen iterative Wiederholungen. Beispielsweise können beim Architekturentwurf Fehler in den Anforderungen festgestellt werden, welche in Folge dessen behoben werden. Oder es können Änderungswünsche der Interessenvertreter geben, die berücksichtigt werden. Änderungen an den Anforderungen haben wiederum Auswirkungen auf die Architektur. Die Kosten für Korrekturen und Änderungen sind hoch [9]. Daher wird die Anzahl der Iterationen im Wasserfallmodell meist begrenzt [80]. Da hierbei Fehler und Änderungswünsche ignoriert werden müssen, führt dies häufig zu nicht erfüllten Wünschen der Interessenvertreter.

2.1.2 Evolutionäre Entwicklung

Bei einer evolutionären Entwicklung wird eine Anfangsversion der Implementierung entwickelt und in Zusammenarbeit mit Interessenvertretern validiert und in Form von Zwischenversionen weiterentwickelt [80]. Die Spezifikation erfolgt hierbei parallel zu der Entwicklung und Validierung (siehe Abb. 2.2). Die Entwicklung wird weitergeführt bis eine Endversion fertig gestellt wurde. Bei der evolutionären Entwicklung kann zwischen zwei grundlegenden Typen unterschieden werden [80]:

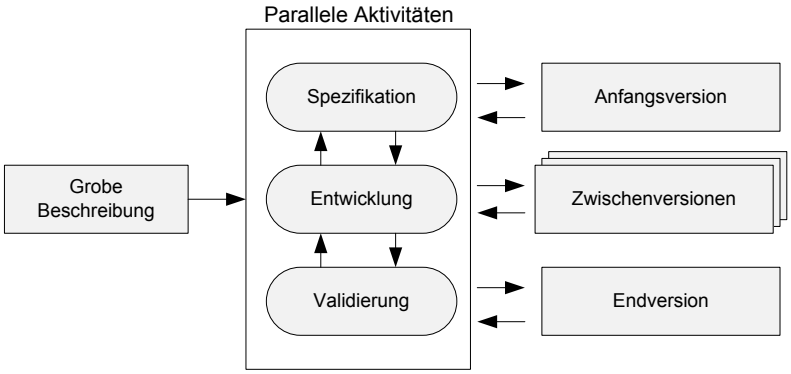


Abbildung 2.2: Evolutionäre Entwicklung nach Sommerville [80]

1. Explorative Entwicklung - Bei dieser Entwicklungsart erfolgt die Entwicklung im Zusammenarbeit mit dem Interessenvertreter. Hierbei wird mit einem abgegrenzten Teil des Systems begonnen und sukzessiv neue Funktionen hinzugefügt.

2. Entwicklung mit Wegwerf-Prototyp - Wegwerf-Prototypen werden entwickelt, welche anschließend durch Interessenvertreter validiert werden. Hierdurch können unklare Anforderungen erhoben und besser verstanden werden.

Durch den evolutionären Ansatz können Änderungswünsche der Interessenvertreter jederzeit berücksichtigt werden. Hierdurch steigt die Wahrscheinlichkeit eines Projekterfolgs. Für große und komplexe Systeme ist dieser Ansatz jedoch nicht geeignet, da die Festlegung einer stabilen Systemarchitektur problematisch ist [80]. Die Software, die Architektur und die Anforderungen werden zusammen entwickelt. Die Anforderungen an die Software und der entgültige Umfang sind folglich erst spät im Projektverlauf bekannt. Die Planung der Ressourcen und die Überprüfung des Projektfortschritts sind damit schwierig [62].

2.1.3 Komponentenbasierte Entwicklung

Die Wiederverwendung bestehender Software wird bei Softwareprojekten zur Beschleunigung der Systementwicklung angewendet [80]. Des Weiteren hilft die Wiederverwendung bei der Vermeidung von Fehlern, da die bestehende Software bereits getestet wurde. In den letzten Jahren wurde für die Wiederverwendung die komponentenbasierte Softwareentwicklung (CBSE) [91] fortlaufend verbessert und erfolgreich eingesetzt [55]. Sie erlaubt eine Zusammensetzung von Systemen aus bestehenden Software-Teilen, die Komponenten genannt werden. Käufliche Komponenten werden hierbei häufig als *Commercial Of-The-Shelf* (COTS)-Systeme bezeichnet. Ein allgemeines Prozessmodell der komponentenbasierten Softwareentwicklung kann aus sechs Phasen bestehen (siehe Abb. 2.3) Innerhalb der Phasen der komponentenbasierten Soft-

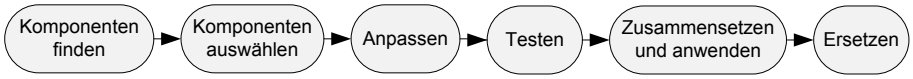


Abbildung 2.3: Komponentenbasierte Entwicklung nach Crnkovic [20]

wareentwicklung werden die folgenden Tätigkeiten durchgeführt [20]:

1. Komponenten finden - In dieser Phase werden bestehende Komponenten identifiziert, mit denen die Anforderungen potentiell erfüllt werden können.

2. Komponenten auswählen - Komponenten werden ausgewählt, die im Zusammenspiel die Anforderungen erfüllen. Die Anforderungen sind individuell. Bestehende Komponenten erfüllen diese Anforderungen meistens nicht vollständig. Sofern eine Anpassung der Anforderungen möglich ist, erfolgt dies in dieser Phase. Alternativ kann eine Eigenentwickelte Komponente verwendet werden.

3. Anpassen - In dieser Phase werden die ausgewählten Komponenten für die Integration in das System angepasst. Die Anpassung kann beispielsweise durch Parametrierung erfolgen.

4. Testen - Es wird geprüft, ob die ausgewählten Komponenten ihre Spezifikation erfüllen.

5. Zusammensetzen und anwenden - Die ausgewählten Komponenten werden innerhalb eines bestehenden Frameworks verwendet.

6. Ersetzen - Im Betrieb werden im Rahmen der Wartung alte Komponentenversionen durch neuere Versionen ersetzt. Hierdurch werden Fehler behoben oder neue Funktionalitäten hinzugefügt.

Die komponentenbasierte Softwareentwicklung beschleunigt den Entwicklungsprozess und hilft bei der Vermeidung von Fehlern. Bei der Entwicklung müssen jedoch meistens Kompromisse bei den Anforderungen eingegangen werden [80]. Hierdurch können Wünsche der Interessenvertreter häufig nicht erfüllt werden. Des Weiteren werden Weiterentwicklungen der Komponenten nicht durch ein einzelnes System bestimmt. Dies kann zu Inkompatiblitäten zu neueren Komponentenversionen oder zu einem Entwicklungsstillstand führen.

2.1.4 Iterative Entwicklung

Durch die genau festgelegten Phasen im Wasserfallmodell ist der Aufwand und die Dauer des Entwicklungsprojekts planbar. Bei Entwicklungsprojekten treten häufig Änderungswünsche der Interessenvertreter auf oder es werden Fehler entdeckt, die sich auf frühere Phasen auswirken. Der evolutionäre Ansatz erlaubt durch die parallele Spezifikation, Entwicklung und Validierung eine Anpassung durch Änderungswünsche über den gesamten Entwicklungsprozess. Die iterative Entwicklung vereint die Vorteile des Wasserfallmodells und der evolutionären Entwicklung. Bei der iterativen Entwicklung werden die Phasen des Entwicklungsprozesses regelmäßig wiederholt. Änderungswünsche können hierbei berücksichtigt werden. Die Wiederholungen unterliegen genau festgelegten Regelungen, die die Korrekturkosten reduzieren. Beispiele für iterative Vorgehensmodelle sind die inkrementelle Entwicklung, das *Spiral Life Cycle*-Modell und das *Twin Peaks*-Modell. Im Folgenden werden das *Spiral Life Cycle*-Modell und das *Twin Peaks*-Modell vorgestellt.

2.1.5 Spiral Life Cycle-Modell

Das *Spiral Life Cycle*-Modell ist eine spezielle Variante der iterativen Entwicklung [80], welche erstmals von Böhm [10] eingeführt wurde. Die Iterationen werden in Form einer Spirale dargestellt (siehe Abb. 2.4). Jede Iteration kann als Phase eines Wasserfallmodells betrachtet werden, bei der ein Produkt entwickelt wird. Das *Spiral Life Cycle*-Modell sieht die folgenden Iterationen vor:

1. Marbarkeit des Systems bestimmen - In dieser Iteration wird das Produkt Betriebskonzept entwickelt

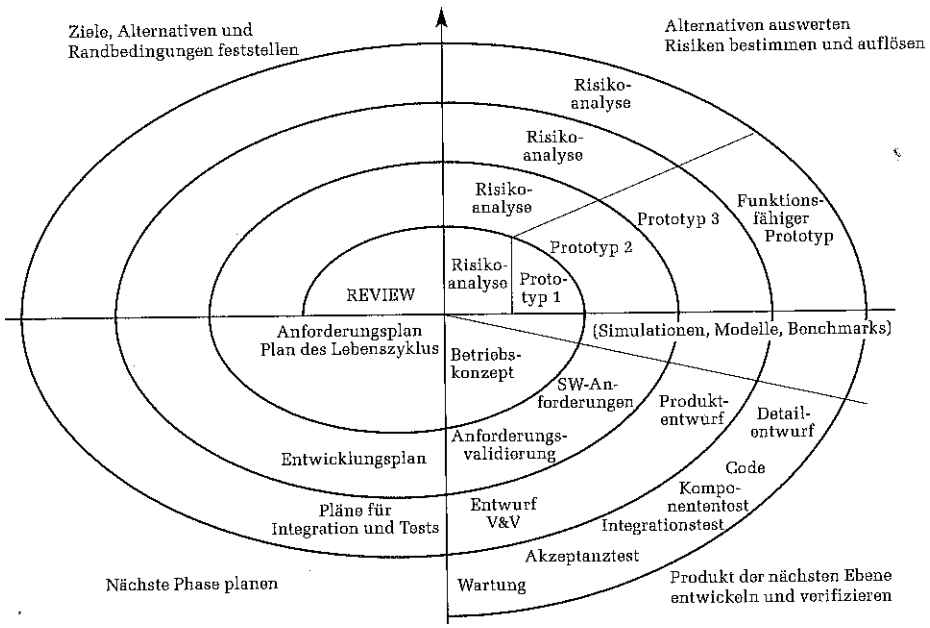


Abbildung 2.4: Spiral Life Cycle-Modell nach Böhm [80]

2. Definition der Anforderungen - Das Ziel ist die Entwicklung der Anforderungsdokumentation

3. Systementwurf - Das Produkt dieser Iteration ist die Spezifikation des Architekturentwurfs.

4. System - In dieser Iteration wird das entgültige System entwickelt.

Bei jeder Iteration werden vier Phasen durchlaufen. Das *Spiral Life Cycle*-Modell sieht hierfür die folgenden Phasen vor:

1. Ziele, Alternativen und Randbedingungen identifizieren - Ziele sind beispielsweise die Funktionalität und Performanz, Alternativen können beispielsweise durch unterschiedliche Architekturen bestehen. Randbedingungen sind beispielsweise Kosten.

2. Alternativen auswerten, Risiken bestimmen und auflösen - Die Auflösung von Risiken kann beispielsweise durch Prototypen und Simulation erfolgen.

3. Entwicklung und Verifizierung - In dieser Phase wird das Produkt der entsprechenden Ebene entwickelt.

4. Planung der nächsten Phasen - Bei einer Prüfung des Projekts wird über die Durchführung der nächsten Iteration entschieden. Wird die nächste Iteration durchgeführt, werden die Phasen geplant.

Durch die spiralförmige Entwicklung und die wiederholte Auflösung von Risiken durch beispielsweise Prototypen wird die Wahrscheinlichkeit des Projekterfolgs erhöht. Ähnlich zu dem Wasserfallmodell ist jedoch für jede Iteration die Entwicklung eines Produkts vorgesehen. Änderungen an vorangegangenen Produkten werden in diesem Modell nicht berücksichtigt. Vielmehr wird durch die Auflösung der Risiken davon ausgegangen, dass vorangegangene Produkte nicht erneut geändert werden müssen.

2.1.6 Twin Peaks-Modell

Das *Twin Peaks*-Modell ist eine spezielle Variante der iterativen Entwicklung und basiert auf einer Anpassung des *Spiral Life Cycle*-Modells [56]. Architekturentscheidungen beeinflussen die Wahl von Anforderungen und umgekehrt beeinflusst die Wahl von Anforderungen die Architekturentscheidungen. Im Gegensatz zum *Spiral Life Cycle*-Modell werden die Wechselwirkungen von Anforderungen und Architekturen berücksichtigt. Hierdurch sind Änderungen im gesamten Entwicklungsprozess möglich. Anforderungen und Architekturen werden daher im *Twin Peaks*-Modell parallel zueinander entwickelt. Die *Twin Peaks* des Modells stellen hierbei den selben Stellenwert von Anforderungen und Architekturen dar (siehe Abb. 2.5). Anforderungen und de-

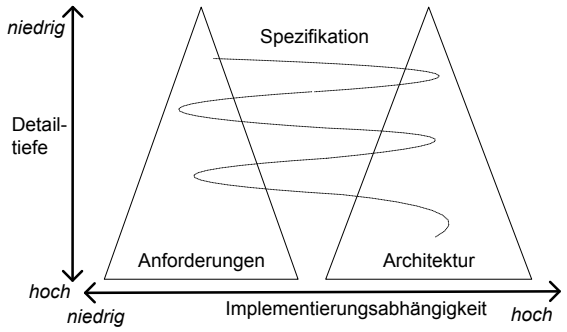


Abbildung 2.5: Twin Peaks-Modell nach Nuseibeh [56]

ren Spezifikation werden fortlaufend strikt von der Architektur und dessen Spezifikation getrennt. Iterativ werden zunehmend detailliertere Anforderungen und Architekturen entwickelt. Durch die iterative Entwicklung in Verbindung mit Prototypen werden Änderungswünsche der Interessenvertreter berücksichtigt. Durch den gleichen Status von Anforderungen und Architekturen wird zudem die Anpassung von Anforderungen an bestehende COTS-Produkte entsprechend der komponentenbasierten Entwicklung adressiert.

2.2 Modelle und Modellierungssprachen

In den letzten Jahren werden Modelle im Rahmen der Softwareentwicklung zunehmend zur Dokumentation verwendet [81]. Ein Modell ist in diesem Zusammenhang eine Datensammlung mit einer festgelegten Struktur und Bedeutung. Modelle erlauben eine abstrakte Beschreibung, bei der Details nach genau definierten Regelungen weg gelassen werden. Dies ist bei der Softwareentwicklung beispielsweise in Entwicklungsstufen erforderlich, in denen die Software nicht in allen Details beschrieben wird. Modelle sind damit zur Beschreibung von Anforderungen und Architekturen geeignet. Sie erhöhen die Verständlichkeit und Überschaubarkeit der Dokumentationen großer Softwaresysteme. Zudem können Sie zur Generierung von Quellcode, Testfällen und Dokumenten wie beispielsweise ein Lastenheft verwendet werden. Modelle können hierbei mehr oder weniger formal sein. Ein formales Modell beschreibt festgelegte Aspekte des Systems vollständig. Durch Abstraktion kann auch bei formalen Modellen die Verständlichkeit und Übersichtlichkeit von Dokumentationen beibehalten werden. Eine Sprache, die sich zur Definition von Anforderungen und Architekturen eignet, ist beispielsweise eine *Requirements Definition Language* (RDL), eine *Architecture Definition Language* (ADL) oder auch eine allgemeine Modellierungssprache. RDLs und ADLs sind beispielsweise COMPASS [14] sowie ArTek [92], AADL [72], Wright [100] und CODE [17]. Weit verbreitete Modellierungssprachen sind beispielsweise die grafischen Sprachen *Unified Modeling Language* (UML) [58] und *Systems Modeling Language* (SysML) [59] sowie die textuelle Sprache Ruby [81]. Grafische Modellierungssprachen sehen Diagramme für die Beschreibungen vor. Allgemein können RDLs, ADLs und Modellierungssprachen dazu verwendet werden ein Modell der Anforderungen und der Architektur des zu entwickelnden Systems zu erstellen. Bei Modellen kann grundlegend zwischen Strukturmodellen und Verhaltensmodellen unterschieden werden. Strukturmodelle beschreiben die Struktur eines Systems und Verhaltensmodelle das Verhalten eines Systems und dessen Bestandteile.

Die Konzepte aktueller Sprachen zur Beschreibung von Systemen basieren häufig auf grundlegenden Modellen wie *Message Sequence Charts* [53], Zustandsautomaten [33] und Petrinetze [63]. Im Folgenden werden die für diese Arbeit relevanten grundlegenden Modelle eingeführt. Der in dieser Arbeit vorgestellte modellbasierte Ansatz wird anhand von UML-Diagrammen erklärt. Daher wird in diesem Abschnitt die UML eingeführt und die verwendeten Struktur- und Verhaltensdiagramme der UML im Detail betrachtet.

2.2.1 Grundlegende Modelle

Message Sequence Chart (MSC)

Das *Message Sequence Chart* (MSC) [53] wurde erstmals 1990 von der *International Telecommunication Union* (ITU) standardisiert und wird häufig in der Telekommunikation und der Softwareentwicklung eingesetzt. Das MSC ist eine Modellierungssprache mit einer grafischen Darstellung zur Beschreibung von Interaktionen zwischen konkreten Ausprägungen von Systembestandteilen. Ausprägungen von Systembestandteilen werden auch Instanzen oder Objekte genannt. Instanzen werden innerhalb eines

MSC durch eine senkrechte Linie visualisiert, die zugleich einen zeitlichen Verlauf darstellt. Die Zeit schreitet hierbei an der Linie abwärts voran. Interaktionen erfolgen in Form von Nachrichten zwischen den Instanzen, die durch Pfeile dargestellt werden. Das MSC *msc1* beschreibt beispielsweise die Interaktion der Instanzen *Person1* und *Person2* (siehe Abb. 2.6), bei der die Nachrichten *Frage* und *Antwort* nacheinander ausgetauscht werden. Die Reihenfolge der auszutauschenden Nachrichten ent-

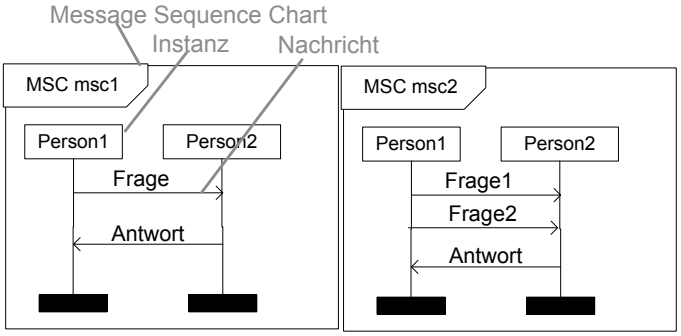


Abbildung 2.6: Beispiel MSC

spricht hierbei der Reihenfolge der Nachrichten des zeitlichen Verlaufs, welcher durch die senkrechten Linien dargestellt wird. Die Nachrichten sind hierbei stets asynchron. Die Modellierung mehrerer aufeinander folgender Anfragen ist demnach möglich. Das MSC *msc2* stellt beispielsweise eine Interaktion dar, bei der eine *Person1* zwei Fragen nacheinander an *Person2* sendet. Erst nach der zweiten Frage erfolgt die Antwort. Der Standard MSC stellt vielfältige Beschreibungselemente zur Modellierung von beispielsweise alternativen und nebenläufigen Nachrichten zur Verfügung [41]. Auch besteht die Möglichkeit große Interaktionen in mehrere kleine Interaktionen über Referenzen zu dekomponieren.

Zustandsübergangsdiagramm

Das Zustandsübergangsdiagramm oder auch *Statechart* (SC) [33] wurde 1987 von David Harel eingeführt. Das SC ist eine grafische Darstellung eines endlichen Automaten, der zur Beschreibung von Systemzuständen und Übergängen bzw. Transitionen dient. Im SC werden Zustände durch Kreise und Transitionen durch Pfeile dargestellt. Das SC *sc1* beschreibt beispielsweise einen Zustandsautomaten mit zwei Zuständen und zwei Transitionen (siehe Abb. 2.7). Bei Bestätigung des Schalters erfolgt jeweils ein Wechsel vom Zustand *Licht an* zu *Licht aus* und umgekehrt. Das SC stellt vielfältige Beschreibungselemente zur Verfügung. Diese ermöglichen beispielsweise die Modellierung bedingter Zustandsübergänge oder die Modellierung eines Sachverhalts, bei dem sich ein System in mehreren Zuständen parallel befinden kann [33]. Zudem besteht die Möglichkeit Zustände zu strukturieren, indem ein Zustand aus mehreren weiteren Zuständen zusammengesetzt wird.

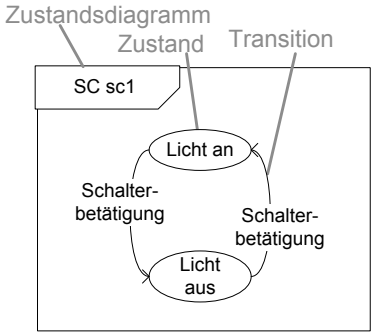


Abbildung 2.7: Beispiel Zustandsdiagramm

Petrinetz

Das Petrinetz [63] [66] ist ein formales Modell für die Beschreibung von Kontrollflüssen mit einer grafischen Repräsentation. Es wurde 1962 erstmals von Carl Adam Petri eingeführt und dient insbesondere der Beschreibung von Systemen, die paralleles und asynchrones Verhalten aufweisen. Ein einfaches Petrinetz wird als Stellen-Transitions-Netz (S-T-Netz) bezeichnet. Für Petrinetze existieren Erweiterungen, die die Ausdrucksmächtigkeit einfacher S-T-Netze erhöhen. Weit verbreitete Petrinetztypen sind Objekt-Netze (OPN) [50], rekursive Netze (RPN) [76], Prädikats-Transitions-Netze (PrT-Netz) [30], gefärbte Petrinetze (CPN) [64][42] und hierarchisch gefärbte Petrinetze (HCPN) [39]. Der in dieser Arbeit vorgestellte modellbasierte Ansatz verwendet für die Beschreibung von Systemen Verhaltensmodelle. Die Semantik des Verhaltensmodells der Architektur basiert auf der Semantik von Petrinetzen und den erweiternden Petrinetztypen. Im Folgenden werden das einfache S-T-Netz sowie die für die Definition der Verhaltensmodelle relevanten Petrinetztypen vorgestellt.

Stellen-Transitionsnetz (S-T-Netz)

Ein einfaches Petrinetz besteht aus Stellen und Transitionen und wird daher auch als Stellen-Transitions-Netz (S-T-Netz) bezeichnet. Stellen und Transitionen sind über gerichtete Kanten miteinander verbunden. Kanten führen immer von einer Stelle zu einer Transition oder von einer Transition zu einer Stelle. Wenn eine Kante von einer Stelle zu einer Transition führt, wird diese als Eingabestelle bezeichnet. Wenn eine Kante von einer Transition zu einer Stelle führt, wird diese als Ausgabestelle bezeichnet. Das Petrinetz *pn1* besteht beispielsweise aus drei Stellen und einer Transition (siehe Abb. 2.8 links). Petrinetze erlauben die Beschreibung des aktuellen Zustands über sogenannte Marken. Im Petrinetz *pn1* ist beispielsweise die Stelle *Licht zentral aus* markiert. Transitionen im Petrinetz können schalten, wenn jede Eingabestelle markiert ist. Beim Schalten wird eine Marke aus jeder Eingabestelle entfernt und zu jeder Ausgabestelle wird eine Marke hinzugefügt. Beim Schalten einer Transition können folglich mehrere Stellen markiert werden. Auch kann eine Stelle mehrfach markiert werden. Im Petrinetz *pn2* kann beispielsweise die Stelle *d* zwei Marken enthalten, nachdem die Transi-

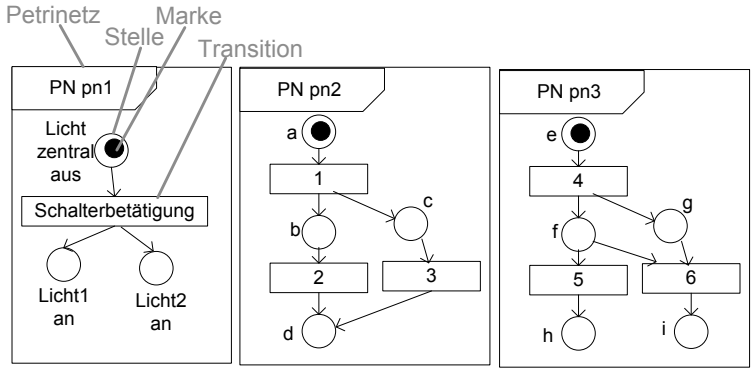


Abbildung 2.8: Beispiel Petrinetz

tionen *1*, *2* und *3* nacheinander geschaltet haben (siehe Abb. 2.8 mitte). Die Schaltung von Transitionen kann nicht-deterministisch sein. Nach der Schaltung der Transition *4* des Petrinetzes *pn3* sind die Stellen *f* und *g* markiert (siehe Abb. 2.8 rechts). Durch diese Markierung sind die Bedingungen für die Schaltung der Transitionen *5* und *6* erfüllt. Da sich jedoch nur eine Marke in der Stelle *f* befindet, kann nur eine der beiden Transitionen Schalten. Die Auswahl der zu schaltenden Transition ist nicht-deterministisch. Durch die Modellierung mehrerer Eingabestellen kann die Synchronisierung paralleler Abläufe beschrieben werden. Die Transition *6* des Petrinetzes *pn3* kann nur schalten, wenn sowohl die Stelle *f* als auch die Stelle *g* markiert ist (siehe Abb. 2.8 rechts). Der Kontrollfluss wird durch diese Transition synchronisiert.

Rekursives Petrinetz (RPN)

Das rekursive Petrinetz (RPN) [76] [32] ist ein erweitertes Petrinetz, dessen Transitionen in drei Kategorien eingeteilt werden können. Die drei Kategorien sind elementare, abstrakte und finale Transitionen. Die rekursiven Petrinetze *rpn1* und *rpn2* zeigen beispielsweise Netze mit elementaren, abstrakten und finalen Transitionen (siehe Abb. 2.9). Im Gegensatz zum einfachen S-T-Netz existiert im RPN nicht nur ein Ausführungsstrang mit einer Markierung, sondern ein dynamischer Baum von Ausführungssträngen mit jeweils einer Markierung bestehend aus einer Menge von Marken. Beim schalten einer abstrakten Transition wird ein neuer Ausführungsstrang mit einer initialen Markierung gestartet und im Baum den aktuellen Ausführungsstrang untergeordnet. Marken sind hierbei immer genau einem Ausführungsstrang zugeordnet. Die abstrakten Transitionen *1* und *2* der Petrinetze *rpn1* und *rpn2* können beispielsweise die initiale Markierung der Stelle *c* des *rpn2* festlegen. Beim Schalten der Transitionen wird folglich ein neuer Ausführungsstrang gestartet, bei dem alleinig die Stelle *c* initial markiert ist. Abb. 2.10 zeigt einen mögliche Ausführungssequenz der Petrinetze *rpn1* und *rpn2*. Beim Schalten der Transition *1* erfolgt der Wechsel vom Zustand *1* zum Zustand *2*. Hierbei wird der Ausführungsstrang *2* dem Baum hinzugefügt. Die Schaltregeln elementarer Transitionen entsprechen denen der einfachen Transitionen eines S-T-Netzes.

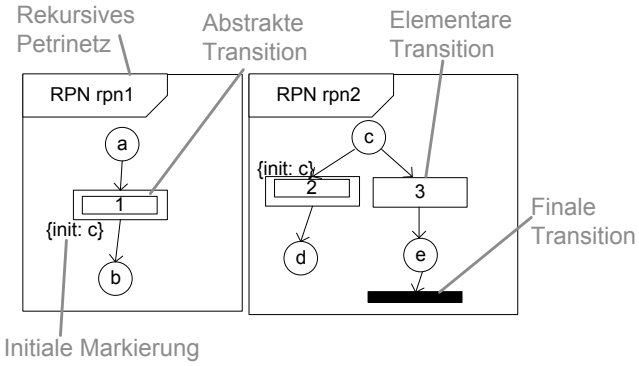


Abbildung 2.9: Beispiel rekursives Petrinetz

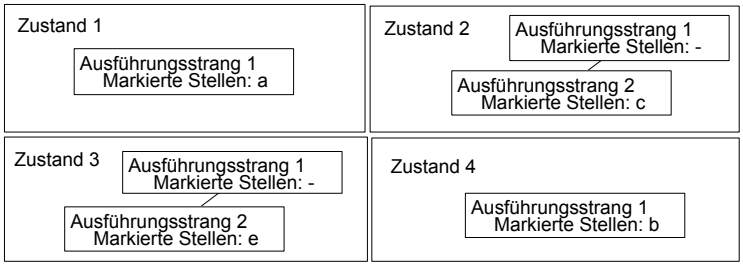


Abbildung 2.10: Ausführungssequenz der Petrinetze *rpn1* und *rpn2*

Die Marken der Eingabestellen müssen zum Schalten dem selben Ausführungsstrang zugeordnet sein. Aus den Eingabestellen wird jeweils eine Marke entfernt und den Ausgabestellen wird jeweils eine Marke hinzugefügt. Die hinzugefügten Marken werden dem Ausführungsstrang der entfernten Marken zugeordnet. Schaltet beispielsweise die Transition *3* im Zustand *2* wird eine Marke aus der Stelle *c* des Ausführungsstrangs *2* entfernt und eine Marke der Stelle *e* hinzugefügt. Die neue Marke der Stelle *e* wird dem Ausführungsstrang *2* zugeordnet (siehe Zustand 3 in Abb. 2.10). Schaltet eine finale Transition wird der aktuelle Ausführungsstrang und alle im Baum untergeordneten Ausführungsstränge entfernt. Den Ausgabestellen der abstrakten Transition, die den Ausführungsstrang gestartet hat, wird jeweils eine Marke hinzugefügt. In der Ausführungssequenz der Abb. 2.10 wird beispielsweise der Ausführungsstrang *2* durch die Schaltung der finalen Transition des Petrinetzes *rpn2* im Zustandswechsel vom Zustand *3* zum Zustand *4* entfernt. Zudem wird die Stelle *b* im Ausführungsstrang *1* markiert.

Gefärbtes Petrinetz (CPN)

Das gefärbte Petrinetz (CPN) [64][42] ist ein höheres Petrinetz. Bei einem höheren Petrinetz können der Netzstruktur Beschriftungen und Deklarationen hinzugefügt werden, die eine abgekürzte Beschreibung des Verhaltens ermöglichen. Beispielsweise

können einfache Datenmanipulationen wie die Addition zweier Zahlen durch eine Kantenbeschriftung beschrieben werden. In einem einfachen Petrinetz hingegen ist hierfür eine komplexe Menge von Stellen und Transitionen erforderlich. Ein CPN besteht aus einem einfachen S-T-Netz, einer Deklaration und den Netzbeschriftungen. Innerhalb einer Deklaration werden Farbmengen und Variablen definiert. Das gefärbte Petrinetz *cpn1* beinhaltet beispielsweise eine Deklaration der Farbmengen *I* und *R*, die einen Ganzzahlwert (Integer) und einen Tupel dreier Ganzahlwerte repräsentieren (siehe Abb. 2.11). Des Weiteren werden die Variablen *i*, *j*, und *r* der Farbmenge *I* deklariert.

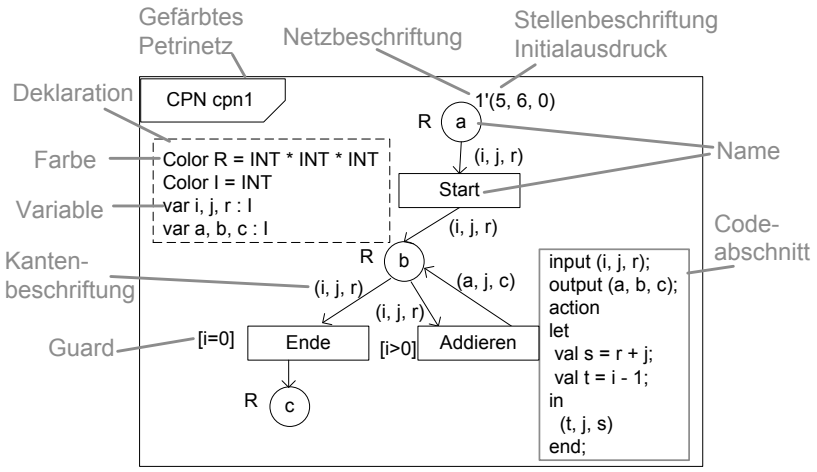


Abbildung 2.11: Beispiel gefärbtes Petrinetz

Mit der Netzbeschriftung können die Stellen, Transitionen und Kanten beschriftet werden. Stellen und Transitionen können hierbei einen Namen erhalten. Zu einer Stelle muss eine Farbmenge angegeben werden. Diese bestimmt die Farbmenge, die die Marken innerhalb dieser Stelle tragen dürfen. Zudem können Stellen einen Initialausdruck erhalten, der die Menge der Marken bei Initialisierung des Netzes bestimmt. Die Stelle *a* des CPN *cpn1* kann beispielsweise Marken der Farbmenge *R* enthalten und enthält initial eine Marke mit der Färbung *(5, 6, 0)* (siehe Abb. 2.11). Kantenbeschriftungen können die Variablen angeben, deren Werte mit einem Token an die nächste Stelle bzw. Transition übergeben werden. Der Transition *Start* werden beispielsweise die Variablen i, j, und r mit den Werten 5, 6 und 0 übergeben (siehe Abb. 2.11). Transitionen können mit einem *Guard* versehen werden, der eine Bedingung für die Schaltung der Transition festlegt. Die Transition *Addieren* des CPN *cpn1* kann beispielsweise nur schalten, wenn die Variable *i* einen Wert größer Null enthält (siehe Abb. 2.11). In modernen Werkzeugen besteht zudem häufig die Möglichkeit einer Transition einen Codeausschnitt hinzuzugefügen [43]. Dieser beschreibt Eingabewerte und Ausgabewerte der Transition sowie die Angabe der durchzuführenden Operationen. Die Transition *Addieren* hat beispielsweise einen Codeausschnitt (siehe Abb. 2.11). Dieser Codeausschnitt legt fest, dass der Wert der Variable *r* um den Wert der Variable *j* erhöht wird. Des Weiteren wird der Wert der Variable *i* um eins verringert. Die Ausgabe der veränderten

Werte erfolgt über die Variablen *a* und *c*. Zur Ausführung eines CPNs benötigen die verwendeten Ausdrücke eine genau festgelegte Syntax und Semantik. Hierfür wird in einem CPN häufig die Sprache *Standard Meta Language* (SML) verwendet [35]. SML unterstützt zudem die Definition komplexer Funktionen. Bei Ausführung des CPN *cpn1* wird letztendlich die Multiplikation der Zahlen *5* und *6* durchgeführt, indem in einer Schleife der Wert der Variable *j* genau *i*-mal der Variable *r* hinzugefügt wird.

Hierarchisches gefärbtes Petrinetz (HCPN)

Das hierarchische gefärbte Petrinetz (HCPN) [39] ist ein CPN, welches die Strukturierung von Petrinetzen erlaubt. In einem HCPN wird ein einzelnes Petrinetz Seite genannt. Innerhalb einer Seite können Aufruftransitionen verwendet werden. Wenn eine Aufruftransition schaltet, wird eine Aufrufinstanz einer Seite erzeugt. Seiten sind hierbei vergleichbar mit Schablonen für Aufrufinstanzen. Die Seite *hcpn1* beinhaltet beispielsweise die Auftruftransition *1* (siehe Abb. 2.12). Stellen an den eingehenden

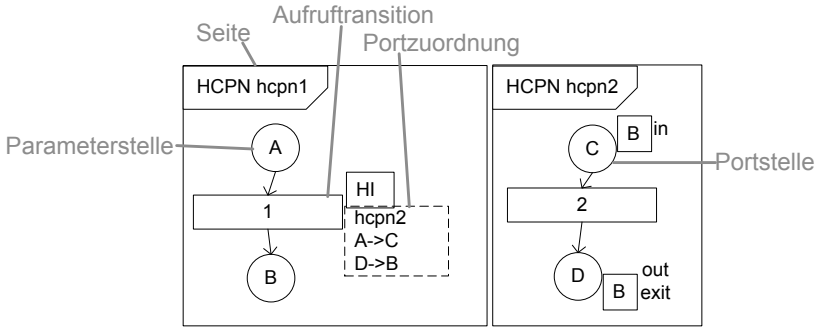


Abbildung 2.12: Beispiel hierarchisches gefärbtes Petrinetz

und ausgehenden Kanten von Aufruftransitionen werden Parameterstellen genannt. Die Stellen *A* und *B* des *hcpn1* sind beispielsweise Parameterstellen (siehe Abb. 2.12). Seiten können Eingabe- und Ausgabeportstellen haben, welche durch ein *B-Tag in* oder *B-Tag out* gekennzeichnet werden. Die Stellen *C* und *D* der Seite *hcpn2* sind beispielsweise Portstellen (siehe Abb. 2.12). Die Stelle *C* ist eine Eingabeportstelle und *D* ist eine Ausgabeportstelle. Die Beziehung zwischen Parameter- und Portstellen wird durch eine Portzuordnung beschrieben. Innerhalb der Portzuordnung wird der Aufruftransition die aufzurufende Unterseite zugeordnet. Zudem werden Parameterstellen den Portstellen zugeordnet. Der Aufruftransition *1* der Seite *hcpn1* ist beispielsweise die Unterseite *hcpn2* zugeordnet, der Parameterstelle *A* ist die Portstelle *C* zugeordnet und der Parameterstelle *B* die Portstelle *D* (siehe Abb. 2.12). Bei Erzeugung einer Aufrufinstanz werden die Marken der Parameterstellen entfernt und den zugeordneten Portstellen hinzugeügt. Wird eine Ausgabeportstelle markiert, die mit *exit* gekennzeichnet ist, werden alle Marken der Ausgabeportstellen auf die zugewiesenen Parameterstellen kopiert und es werden die Aufrufinstanz und Unterinstanzen vom HCPN entfernt. Bei hierarchischen gefärbten Petrinetzen besteht auch unter anderem die Möglichkeit Stel-

len und Transitionen durch Seiten zu substituieren [39]. Bei einer Substitution ist im Gegensatz zu den beschriebenen Aufruftransitionen keine Rekursion möglich.

2.2.2 Unified Modeling Language (UML)

Die Sprache *Unified Modeling Language* (UML) [58][73] ist eine standardisierte Modellierungssprache, die im Jahr 1997 in der Version 1.0 eingeführt wurde. Die UML ist geeignet für die Dokumentation, Spezifikation und für die grafische Darstellung komplexer Softwaresysteme. Seit der Version 1.2 wurde die UML von der *Object Management Group* (OMG) zur UML 2.0 weiterentwickelt, welche im Jahr 2005 eingeführt wurde. Bei der Weiterentwicklung wurde unter anderem die Komplexität der Diagramme reduziert und die Ausführbarkeit der Modelle durch die Verwendung erprobter Konzepte wie Petri-Netze [63] verbessert [73]. Die UML 2.0 stellt Struktur- und Verhaltensdiagramme zur Verfügung (siehe Abb. 2.13). Die Struktur kann durch Ty-

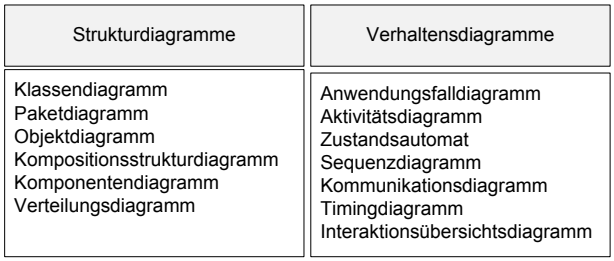


Abbildung 2.13: Die Diagrammtypen der UML2 [73]

pen beschrieben werden, die für eine Menge möglicher Ausprägungen von Systembestandteilen stehen. Die Beschreibung von Typen erfolgt beispielsweise über das Klassendiagramm oder das Komponentendiagramm. Zudem können mit Hilfe der UML beispielhaft konkrete Ausprägungen der Systembestandteile modelliert werden. Diese werden auch als Objekt oder Instanz bezeichnet. Die Beschreibung konkreter Ausprägungen kann beispielsweise durch ein Objektdiagramm erfolgen. Im Kapitel 2.2 werden Strukturmodelle im Detail betrachtet. Bei den Verhaltensdiagrammen kann ebenfalls zwischen zwei Arten unterschieden werden. Eine Art der Diagramme beschreibt konkrete Ausprägungen von Abläufen in Form von Interaktionen zwischen Objekten. Die andere Art beschreibt die Menge aller möglichen Abäufe. Konkrete Ausprägungen von Interaktionen zwischen Objekten werden beispielsweise über das Kommunikationdiagramm und das Sequenzdiagramm beschrieben. Die Beschreibung der Menge aller möglichen Abläufe erfolgt über Diagramme wie das Aktivitätsdiagramm und dem Zustandsautomat.

2.2.3 UML Strukturdiagramme

UML Objektdiagramm

Das UML Objektdiagramm [58] beschreibt konkrete Objekte oder auch Instanzen. Bei der Modellierung der Struktur eines Softwaresystems können Objekte konkrete

Systembestandteile zur Laufzeit darstellen. Objekte haben einen Namen, einen Typ und Attribute. Attribute im Objektdiagramm sind mit konkreten Werten belegt. Des Weiteren haben Attribute einen Namen und einen Typ. In einem Objektdiagramm können zudem Verknüpfungen von Objekten beschrieben werden. Das Objekt *hans* vom Typ *Person* des Objektdiagramms *od1* hat beispielsweise die Attribute *größe* und *gewicht* mit der Wertebelegung 178 und 78 (siehe Abb. 2.14). Diese Werte sind ganze

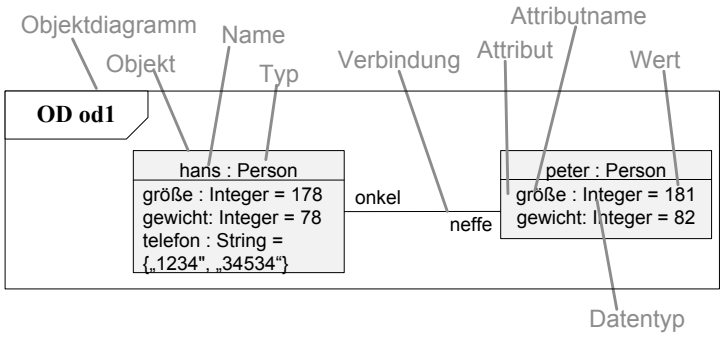


Abbildung 2.14: Beispiel Objektdiagramm

Zahlen und haben damit den Typ *Integer*. Ein Attribut kann auch mit einer Menge von Werten belegt sein. Das Attribut *telefon* ist mit mit einer Menge von Zeichenketten belegt. Zwischen dem Objekt *peter* und dem Objekt *hand* existiert eine Verküpfung, die eine Onkel-Neffe-Beziehung ausdrückt.

UML Klassendiagramm

Das UML Klassendiagramm [58] beschreibt Typen in Form von Klassen, die für eine Menge von Objekten mit gleichen Eigenschaften stehen. Dies erlaubt eine abstrakte Beschreibung der Struktur eines Softwaresystems. Die Objekte *hans* und *peter* haben die Attribute *größe* und *gewicht* vom Typ *Integer* (siehe Abb. 2.14). Das Klassendiagramm *kd1* beschreibt beispielsweise für diese Objekte ein Typ *Person* in Form einer Klasse (siehe Abb. 2.15). Die Klasse *Person* hat ein Attribut *größe* vom Typ *Integer* (siehe Abb. 2.15). Jede Klasse hat einen Bezeichner. Attribute werden durch eine Linie getrennt vom Bezeichner dargestellt. Attribute haben grundlegend einen Namen (eine Rolle) und einen Typ. Der Typ kann primitiv sein wie eine ganze Zahl oder eine Zeichenkette. Zudem kann der Typ komplex sein und durch eine Klasse beschrieben werden. Für ein Attribut, welches mit einer Menge von Werten belegt werden kann, wird im Klassendiagramm eine Multiplizität festgelegt. Eine Multiplizität gibt vor, wie viele Werte sich in der Menge befinden müssen bzw. dürfen. Das Attribut *telefon* der Klasse *Person* hat beispielsweise die Multiplizität *0...** (siehe Abb. 2.15). Das Zeichen *0* steht hierbei für die untere Grenze und das Zeichen * für die obere Grenze. Das Zeichen * bedeutet hierbei eine unbegrenzte Anzahl. Sofern nur eine Angabe innerhalb einer Multiplizät angegeben wird, ist diese sowohl die obere als auch die untere Grenze. Eine Verknüpfung von Objekten kann im Klassendiagramm über Beziehungen wie

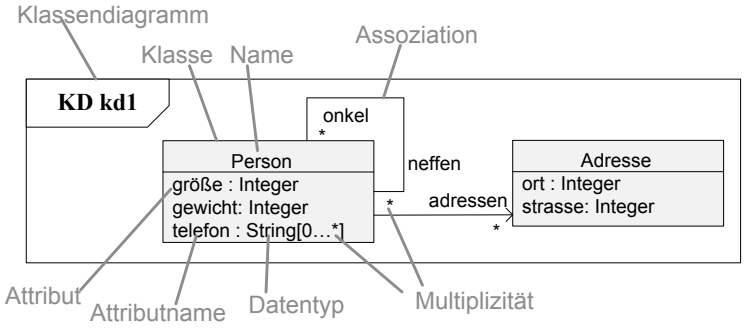


Abbildung 2.15: Beispiel Klassendiagramm

beispielsweise Assoziationen zwischen Klassen modelliert werden. Die Klasse *Person* ist beispielsweise mit sich selbst assoziiert (siehe Abb. 2.15). Diese Assoziation drückt die Onkel-Neffe-Beziehung aus. Die Namen der Assoziationsenden werden als Rollen bezeichnet. Assoziationen können hierbei sowohl gerichtet als auch ungerichtet sein. Die Assoziation für die Onkel-Neffe-Beziehung ist ungerichtet und die Assoziation der Klasse *Person* mit der Klasse *Adresse* ist gerichtet. Die Bedeutung der Multiplizität an den Assoziationsenden ist analog zu der Bedeutung der Multiplizität an den Attributen. Ein Objekt der Klasse *Person* kann beispielsweise beliebig viele Onkels und Neffen haben sowie beliebig viele Adressen. Die UML stellt eine Menge spezieller Beziehungstypen zwischen Klassen zur Verfügung [58]. Häufig verwendete Beziehungstypen sind neben der Assoziation die Generalisierung, die Komposition und die Aggregation. Bei einer Generalisierung ist eine Klasse die Verallgemeinerung einer anderen [73] und bei der Komposition ist ein Objekt einer Klasse ein Teil eines Objekts einer anderen Klasse [73]. Bei einer Aggregation kann ein Objekt einer Klasse Teil mehrerer Objekte sein. Für Klassen können zudem Operationen definiert werden [58]. Diese dienen zur allgemeinen Beschreibung des Verhaltens von Objekten und können mit detaillierteren Verhaltensbeschreibungen verknüpft sein. Des Weiteren können mit einem Klassendiagramm spezielle Klassen wie Schnittstellen und Aufzählungen modelliert werden [73].

UML Komponentendiagramm

Das UML Komponentendiagramm [58] beschreibt Komponenten, die für Bestandteile des Systems stehen. Komponenten beschreiben Typen vergleichbar mit den Klassen des UML Klassendiagramms. Sowohl im Klassendiagramm als auch im Komponentendiagramm können Schnittstellen definiert werden. Schnittstellen sind spezielle Klassen, die einen Kontrakt spezifizieren. Dies erfolgt durch die Definition verpflichtender Eigenschaften wie beispielsweise das zur Verfügung zu stellende Verhalten. Ein Objekt eines Typs, der eine Schnittstelle realisiert bzw. implementiert, muss diesen Kontrakt erfüllen. Im Gegensatz zum Klassendiagramm stehen im Komponentendiagramm die Schnittstellen im Vordergrund. Komponenten verfügen über klar de-

finierte Schnittstellen und können gegen andere Komponenten ausgetauscht werden, die die selben Schnittstellen realisieren. Daher bestehen Beziehungen zwischen Komponenten ausschließlich über Schnittstellen. Die Komponente *Personenverwaltung* der Komponentendiagramms *kompD1* implementiert beispielsweise die Schnittstelle *PersonIF* (siehe Abb. 2.16). Komponenten können andere Komponenten über Schnitt-

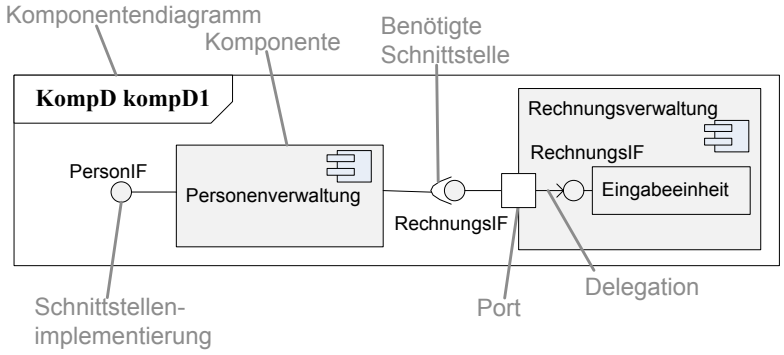


Abbildung 2.16: Beispiel Komponentendiagramm

stellen verwenden. Zu einer Kompenente werden hierfür die benötigten Schnittstellen definiert. Die Komponente *Personenverwaltung* benötigt beispielsweise die Schnittstelle *RechnungsIF*. Diese Schnittstelle wird von der Komponente *Rechnungsverwaltung* über einen Port implementiert. Ein Port stellt einen Kommunikationspunkt einer Komponente mit ihrer Außenwelt dar. Die Komponente *Rechnungsverwaltung* kann durch eine andere Komponente ausgetauscht werden, die den Kontrakt der Schnittstelle *RechnungsIF* erfüllt. Mit einem Komponentendiagramm kann des Weiteren der interne Aufbau einer Komponente durch weitere Komponenten beschrieben werden. Die Komponente *Rechnungsverwaltung* beinhaltet beispielsweise die Komponente *Eingabeeinheit*, die ebenfalls die Schnittstelle *RechnungsIF* implementiert. Aufrufe an eine Schnittstelle, können an interne Komponenten weitergeleitet bzw. delegiert werden. Dies wird über einen Delegationsverbinder modelliert. Zwischen dem Port und der Schnittstelle *RechnungsIF* befindet sich beispielsweise ein Delegationsverbinder.

UML Kompositionsstrukturdiagramm

Das UML Kompositionsstrukturdiagramm [73] ermöglicht die Beschreibung von Kompositionsbeziehungen bzw. Teile-Ganzes-Beziehungen. Im Gegensatz zum Klassendiagramm können im Kompositionsstrukturdiagramm die Kompositionsbeziehungen genauer beschrieben werden. Im Kompositionsstrukturdiagramm besteht die Möglichkeit eine Verknüpfungen von Objekten in Abhängigkeit ihres Kontexts zu beschreiben. Die Klasse *Auto* im Klassendiagramm *kd2* besteht beispielsweise aus zwei Hinterreifen und einem Motor (siehe Abb. 2.17 rechts). Die Assoziation *Achse* zwischen den Klassen *Reifen* und *Motor* beschreibt die generell mögliche Existenz einer Verknüpfung von Hinterreifen- und Motorobjekten. Da die Assoziation allgemein für alle Reifen und Motoren gilt, können die Multiplizitäten nicht genauer beschrieben werden. Im

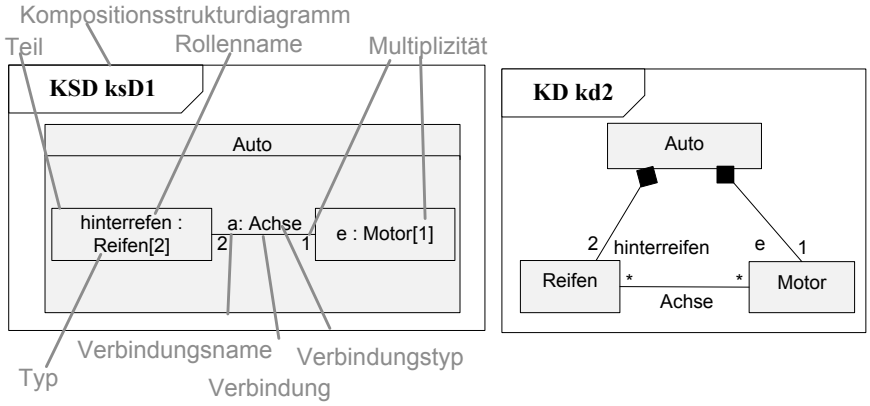


Abbildung 2.17: Beispiel Kompositionsstrukturdiagramm [58]

Gegensatz hierzu beschreibt die Verbindung *a* vom Typ *Achse* des Kompositionsstrukturdiagramms *ksD1* die Verknüpfungsmöglichkeit im Kontext der Klasse *Auto* genauer (siehe Abb. 2.17 links). Die Hinterreifen sind mit genau einem Motor verknüpft und der Motor mit genau zwei Hinterreifen. Da diese Beschreibung nur für die Teile im Kontext *Auto* gilt, stellt diese keine Einschränkung der allgemeinen Klassen *Reifen* und *Motor* dar.

2.2.4 UML Verhaltensdiagramme

UML Kommunikationsdiagramm

Das UML Kommunikationsdiagramm [73] dient der Beschreibung konkreter Abläufe in Form von Kommunikationen, bei denen Objekte Nachrichten in einer festgelegten Reihenfolge austauschen. Objekte werden hierbei durch Lebenslinien modelliert. Die Reihenfolge wird über Sequenznummern festgelegt. Das Kommunikationsdiagramm *kmD1* beschreibt beispielsweise ein Szenario bei dem das Objekt *peter* vom Typ *Person* eine Frage an das Objekt *hans* vom Typ *Person* sendet (siehe Abb. 2.18 links). Das Objekt *hans* sendet daraufhin eine Antwort zurück. Die Beschreibung nebenläufiger Nachrichten wird über die Sequenznummern ermöglicht. Das Kommunikationsdiagramm *kmD2* beschreibt beispielsweise ein Szenario, bei dem ein Sensor ein Signal an einen PC sendet (siehe Abb. 2.18 rechts). Daraufhin werden nebenläufig vom PC die Bestätigung an den Sensor und die Reaktion an den Aktuator gesendet. Das UML Kommunikationsdiagramm unterstützt zudem die Beschreibung iterativ wiederholter Nachrichten sowie bedingte Nachrichten [73].

UML Aktivitätsdiagramm

Das UML Aktivitätsdiagramm [73] eignet sich zur Beschreibung einer Menge möglicher Abläufe. Bei der Beschreibung eines Softwaresystems kann das Aktivitätsdia-

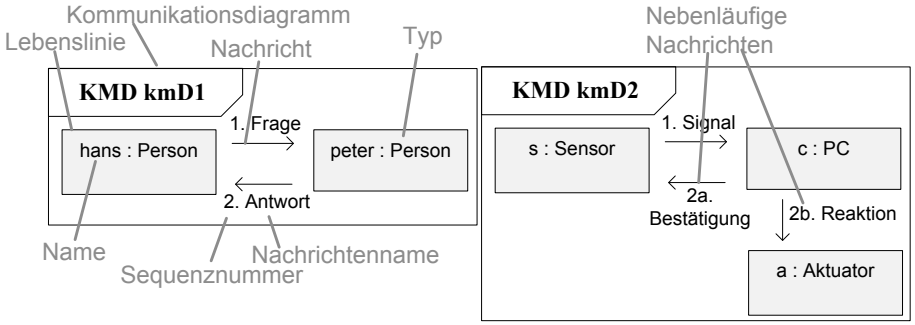


Abbildung 2.18: Beispiel Kommunikationsdiagramm

gramm beispielsweise alle unterstützten Abläufe einer Funktion beschreiben. Ablaufsvarianten werden hierbei durch Entscheidungen beschrieben. UML Aktivitätsdiagramme der Version 2.0 unterstützen neben der Modellierung des Kontrollflusses auch die Möglichkeit der Modellierung eines Datenflusses. Im Folgenden wird zunächst die Modellierung des Kontrollflusses und anschließend die Modellierung des Datenflusses beschrieben.

UML Aktivitätsdiagramm Kontrollfluss

Ein UML Aktivitätsdiagrammm besteht grundlegend aus einer Menge von Knoten, die über gerichtete Kanten miteinander verbunden sind. Knoten können Aktionen darstellen. Die gerichteten Kanten geben die Richtung des Kontrollflusses vor und bestimmen damit die Reihenfolge der im Rahmen der Aktivität auszuführenden Aktionen. Für die Beschreibung des Kontrollflusses stehen mehrere Knotentypen zur Verfügung. Die Aktivitätsdiagramme *ad1* und *ad2* stellen beispielsweise Aktivitäten dar, die die meist verwendeten Knotentypen beinhalten (siehe Abb. 2.19). Der Kontrollfluss einer Aktivität startet immer bei den initialen Knoten. Parallele Abläufe werden durch Parallelisierungsknoten gestartet. Innerhalb eines Ablaufs einer Aktivität können Aktionen erfolgen. Nach Beginn der Aktivität *ad1* wird beispielsweise parallele Abläufe gestartet, bei denen die Aktionen *Wasser aufkochen* und *Gemüse schneiden* parallel zueinander durchgeführt werden (siehe Abb. 2.19). Parallele Abläufe können über Vereinigungsknoten synchronisiert werden. Der Kontrollfluss wird in der Aktivität *ad1* beispielsweise erst fortgesetzt, nachdem die Aktionen *Wasser aufkochen* und *Gemüse schneiden* beendet wurden.

UML Aktivitätsdiagramme erlauben die Dekomposition großer Abläufe in mehrere kleinere Aktivitäten. Hierfür können Aufrufaktionen innerhalb einer Aktivität verwendet werden. Diese starten eine andere Aktivität. In der Aktivität *ad1* wird beispielsweise nach der Synchronisierung die Aktivität *ad2* durch eine Aufrufaktion gestartet (siehe Abb. 2.19). Mehrere Varianten der Durchführung einer Aktivität werden durch Entscheidungsknoten beschrieben. Die Bedingungen für die Durchführung einer Variante werden durch *Guards* an den ausgehenden Kanten aus dem Entscheidungsknoten festgelegt. Nach Durchführung der Aktion *Gemüse kochen* werden beispielsweise

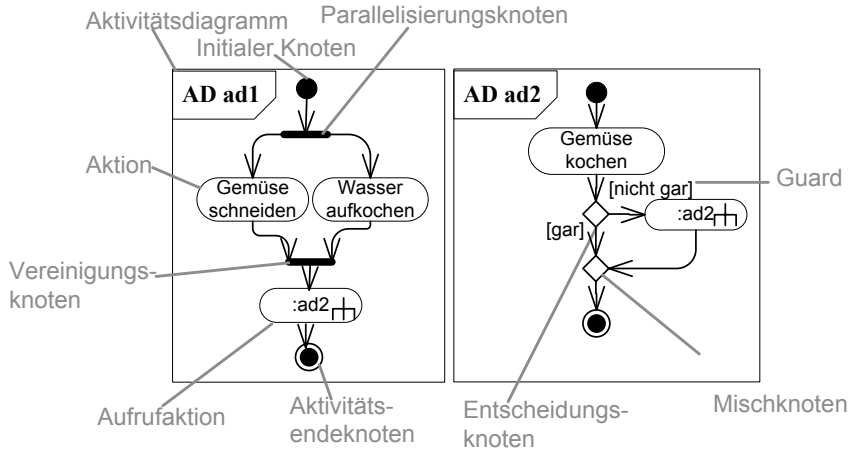


Abbildung 2.19: Beispiel Aktivitätsdiagramm

zwei Varianten der Durchführung der Aktivität mit den Bedingungen *gar* und *nicht gar* beschrieben (siehe Abb. 2.19). Kontrollflüsse werden über Mischknoten zusammengeführt. Durch den Mischknoten der Aktivität *ad2* wird beispielsweise ausgedrückt, dass an dieser Stelle sowohl der Kontrollfluss mit der Variante *nicht gar* als auch mit der Variante *gar* fortgesetzt wird (siehe Abb. 2.19). Das Ende einer Aktivität wird über Aktivitätsendeknoten modelliert. Hierbei werden auch alle Aktivitäten beendet, die durch Aufrufaktionen innerhalb der beendeten Aktivität aufgerufen wurden. Die Aktivität *ad1* ist beispielsweise nach Abschluss der Aufrufaktion *:ad2* beendet.

UML stellt zudem weitere Beschreibungselemente für Aktivitätsdiagramme zur Verfügung, um beispielsweise Ausnahmebehandlungen zu modellieren [73]. Des Weiteren können den Aktionen über sogenannte Schwimmbahnen Verantwortlichkeiten (z.B. Personen) zugeordnet werden.

Semantik des Kontrollflusses

Seit Version UML 2.0 basiert die Semantik von Aktivitätsdiagrammen auf tokenbasierten Kontrollflüssen, welche von Petrinetzen inspiriert ist [89]. Die Semantik der Aktivitätsdiagramme ist hierbei informell definiert. Für die Formalisierung existieren mehrere Ansätze. Bei den Ansätzen [89] und [82] wird die Semantik über die Abbildung auf Petri-Netze beschrieben. Bei der Abbildung der Semantik auf Petrinetze nach Störrle [89] entsprechen die Knoten eines Aktivitätsdiagramms Stellen und Transitionen eines Petrinetzes. Je nach Typ des Knotens wird eine bestimmte Kombination von Stellen und Transitionen verwendet, um die Semantik zu beschreiben. Beispielsweise wird ein Parallelisierungsknoten durch eine Transition mit mehreren Ausgabestellen beschrieben.

Eine formale Beschreibung der Semantik des Kontrollflusses mit Aufrufaktionen erfolgt beispielsweise in [87]. Hierbei werden Aufrufaktionen durch Prozeduraufruf-

transitionen beschrieben. Beim Schalten einer Prozeduraufruftransition wird ein Zustandselement erzeugt. Ein Zustandselement beschreibt unter anderem die aufrufende Transition und eine Markierung. Ein Zustandselement ist damit vergleichbar mit dem Ausführungsstrang eines RPN. Durch die aufrufende Transition wird der Vorgänger im Baum der Ausführungsstränge beschrieben. Ein RPN kann damit analog zur formalen Beschreibung der Semantik von Aufrufaktionen verwendet werden. Das RPN *adrpn* beschreibt beispielsweise die Semantik der Aktivität *ad3* (siehe Abb. 2.20). Eine Auf-

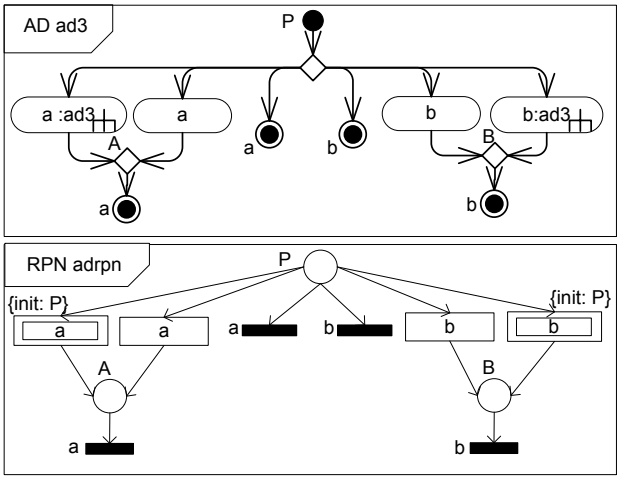


Abbildung 2.20: Beispiel Aktivitätsdiagramm-Semantik

rufaktion entspricht hierbei einer abstrakten Transition und ein Aktivitätsendeknoten einer finalen Transition. Die abstrakte Transition muss immer die initiale Markierung der Stellen festlegen, die die initialen Knoten der aufgerufenen Aktivität bestimmen. Die abstrakte Transition *a* des RPN *adrpn* steht beispielsweise für die Aufrufaktion *a: ad3* der Aktivität *ad3* und legt die initiale Markierung der Stelle *P* fest, die für den initialen Knoten der Aktivität *ad3* steht (siehe Abb. 2.20). Der Aktivitätsendeknoten *a* der Aktivität *ad3* wird beispielsweise durch die finale Transition *a* des RPN *adrpn* beschrieben. Die Namen der Knoten der Aktivität *ad3* können beispielsweise den Beschriftungen des RPN *adrpn* entsprechen.

Das beschriftete RPN *adrpn* beschreibt genau die Menge aller Palindrome mit den zugelassenen Zeichen *a* und *b* [32]. Ein einfaches S-T-Netz kann die Menge aller Palindrome nicht beschreiben [32]. Der Kontrollfluss eines Aktivitätsdiamms mit den vorgestellten Kontrollknotentypen und Aufrufaktionen hat folglich eine größere Mächtigkeit als ein S-T-Netz. Ein RPN kann hingegen die Semantik dieser Beschreibungselemente des Aktivitätsdiagramms abbilden.

UML Aktivitätsdiagramm Datenfluss

Das UML 2.0 Aktivitätsdiagramm ermöglicht neben der Modellierung des Kontrollflusses auch die Beschreibung des Datenflüsses [73]. Datenflüsse werden über spezielle

Kanten von und zu Objektknoten beschrieben. Objektknoten modellieren den Fluss von Objekten und Werten indem sie beispielsweise den Namen und den Typ der übergebenen Objekte festlegen. Sie stellen selbst kein Objekt dar. Das Aktivitätsdiagramm *ad4* erweitert beispielsweise das Aktivitätsdiagramm *ad1* mit einer Beschreibung des Datenflusses dar (vgl. Abb. 2.19 und 2.21). Der Ausgabewert der Aktion *Gemüse schnei-*

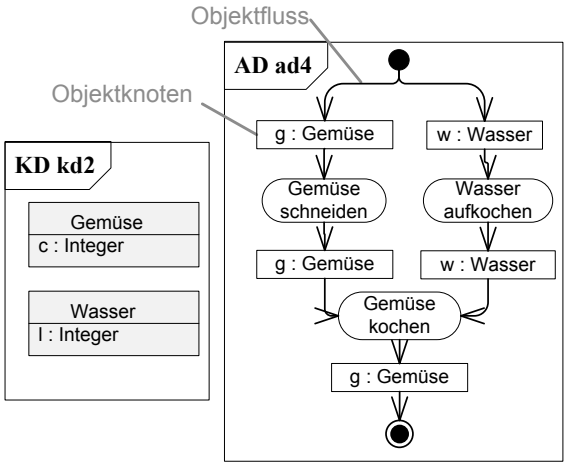


Abbildung 2.21: Beispiel Aktivitätsdiagramm mit Objektfluss

den wird beispielsweise durch einen Objektknoten beschrieben. Dieser legt den Ausgabewert *g* vom Typ *Gemüse* fest. Die verwendeten Typen können beispielsweise über ein Klassendiagramm beschrieben werden. Das Klassendiagramm *kd2* beschreibt beispielsweise den Typ *Gemüse* (siehe Abb. 2.21). Dieser verfügt über ein Attribut *c* vom Typ einer ganzen Zahl. Der Objektknoten *g* modelliert zugleich einen der Eingabewerte der Aktion *Gemüse kochen* (siehe Abb. 2.21).

Semantik mit Datenfluss

Die Semantik des Datenflüsses der UML Aktivitätsdiagramme ist ebenfalls informell beschrieben. Für die Formalisierung existieren Ansätze, die die Semantik beispielsweise über das CPN beschreiben [88]. Die in einer Aktivität über Objektknoten verwendeten Datentypen werden hierbei durch die Farben des CPN modelliert und die Namen der Objektknoten werden zu Variablen. Das CPN *cpn3* beschreibt beispielsweise die Semantik der Aktivität *ad4* (siehe Abb. 2.22). Der Datentyp *Gemüse* wird zum Beispiel durch eine Farbe beschrieben. Ein Objektfluss entspricht hierbei einer gefärbten Marke, die durch Transitionen im Netz bewegt wird. Die Eingabestelle der Aktion *Gemüse schneiden* kann Marken mit der Farbe *Gemüse* beinhalten. Der Aktion wird mit einer eingehenden Marke der Wert der Variable *g* übergeben. Die Variable *g* hat die Farbe *Gemüse*. Die Bedeutung von Aktionen und *Guards* kann durch Netzbeschriftungen des CPN beschrieben werden. In dem Ansatz von Störrle [88] wird eine angepasste Variante der Beschriftungssprache SML für *Guards* und Kantenbeschriftungen verwendet.

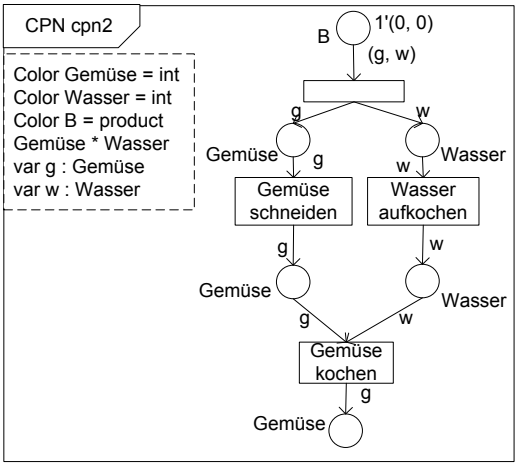


Abbildung 2.22: Beispiel CPN-Semantik für den Datenfluss

2.3 Modellbasierte Softwareentwicklung

Eine Softwareentwicklung, bei der die Dokumentation mit Hilfe von Modellen erfolgt, wird als *modellbasiert* bezeichnet [81]. Die *modellgetriebene* Softwareentwicklung geht einen Schritt weiter. Bei der modellgetriebenen Softwareentwicklung wird aus Modellen automatisiert Software erzeugt [81]. Die Modelle sind bei der modellgetriebenen Softwareentwicklung formal und damit gleichzusetzen mit Quellcode. Die Struktur und die Bedeutung der Modelle muss sowohl für die modellbasierte als auch für die modellgetriebene Softwareentwicklung eindeutig definiert sein. Diese Definition erfolgt durch ein Metamodell, das auch als abstrakte Syntax bezeichnet wird. Die abstrakte Syntax legt die möglichen Konstrukte einer Modellierungssprache und ihre Beziehungen untereinander fest.

Bestehende modellbasierte Ansätze für die Entwicklung von Anforderungen und Architekturen können kategorisiert werden in modellbasierte Ansätze für die Anforderungserhebung, modellbasierte Ansätze für den Architekturentwurf und kombinierte Ansätze. Die Metamodellierung ist ein grundlegendes Konzept zur Definition modellbasierter Ansätze. Im Folgenden wird das Konzept der Metamodellierung eingeführt. Anschließend werden die bestehenden modellbasierten Ansätze zur Anforderungserhebung und zum Architekturentwurf sowie die kombinierten Ansätze vorgestellt.

2.3.1 Metamodellierung

Ein Metamodell oder auch die abstrakte Syntax legt die Struktur und die Bedeutung eines Modells fest. Hierbei werden die grundlegenden Konstrukte und ihre Beziehungen zueinander definiert. Im Gegensatz hierzu beschreibt eine konkrete Syntax wie diese Konstrukte und Beziehungen aufgeschrieben oder dargestellt werden. Die abstrakte Syntax kann beispielsweise ein Konstrukt *Klasse* definieren, welche für einen Datentyp

steht und einen Bezeichner hat. Die konkrete Syntax definiert hingegen beispielsweise, dass die Klassendefinition nach dem Wort *class* im Text beginnt und das nächste Wort ein Bezeichner sein muss. Bei einer grafischen Modellierungssprache kann die konkrete Syntax die Darstellung der Klasse durch beispielsweise ein Rechteck festlegen.

Bei der Metamodellierung kann grundlegen zwischen den Ebenen *die zu beschreibende Domäne*, *Modell* und *Metamodell* unterschieden werden [73] (siehe Abb. 2.23). Die zu beschreibende Domäne besteht beispielsweise aus Personen der realen Welt.

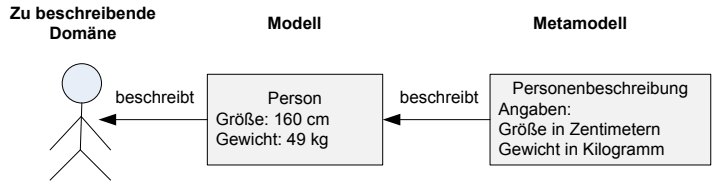


Abbildung 2.23: Modell und Metamodell am Beispiel nach [73]

Das Modell kann eine abstrakte Beschreibung der Personen beinhalten mit den Informationen, die für die gegebene Aufgabe relevant sind. Eine Information kann beispielsweise die konkrete Größenangabe einer Person von 160 cm sein. Das Metamodell beschreibt nun wie diese abstrakte Beschreibung der Person definiert ist. Beispielsweise kann das Metamodell festlegen, dass zu jeder Personenbeschreibung eine Größenangabe in Zentimetern gehört. Zu einem Metamodell kann immer auch ein Meta-Metamodell definiert werden, die die Elemente des Metamodells beschreiben [81]. Bei dem Begriff der Metamodellierung ist nicht immer eindeutig, ob ein Meta-Metamodell auch ein einfaches Metamodell oder ein Modell ist. Beispielsweise kann auch ein Metamodell potenziell die zu beschreibende Domäne sein.

Die OMG entwickelte zur Meta-Metamodellierung die *Meta Object Facility* (MOF) [61]. Diese beschreibt beispielsweise sich selbst und das Metamodell der UML. Die verschiedenen Ebenen der Beschreibung werden von der OMG als M0 bis M3 bezeichnet. M0 entspricht hierbei der Ebene der zu beschreibenden Domäne, M1 dem Modell, M2 dem Metamodell und M3 dem Meta-Metamodell. Das Klassendiagramm *m3* beschreibt skizzenhaft einen Ausschnitt der MOF (siehe Abb. 2.24).

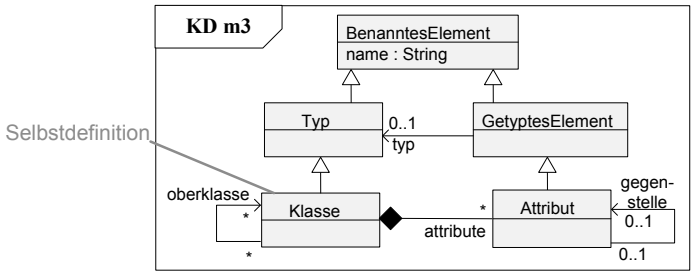


Abbildung 2.24: Skizze der MOF M3

Diese beschreibt skizzenhaft das Metamodell eines Klassendiagramms und damit auch sich selbst. Ein Attribut steht hierbei auch für Beziehungen zwischen Klassen wie beispielsweise Assoziationen. In diesem Fall hat dieses Attribut eine Gegenstelle. Eine weitere Beziehung zwischen Klassen wird durch die Assoziation *oberklasse* beschrieben. Diese steht für die Generalisierungsbeziehung beispielsweise zwischen den Klassen *BenanntesElement* und *Typ*.

2.3.2 Modellbasierte Anforderungserhebung

Modellbasierte Ansätze zur Anforderungserhebung stellen Modelle zur Beschreibung von Anforderungen zur Verfügung. Modelle zur Beschreibung von Anforderungen unterstützen zumeist die Validierung der Anforderungen. Zudem eignen sie sich häufig zur automatisierten Generierung von Artefakten, die im Rahmen der Softwareentwicklung und Dokumentation verwendet werden können. Zur Validierung von Anforderungen stellen einige Ansätze geeignete Modelle zur Verfügung [24] [79] [94] oder unterstützen die Visualisierung der Modelle durch Mockups [75] [99] sowie durch prototypische Ausführung und Simulation [24] [47]. Zur automatisierten Weiterverarbeitung erlauben einige Ansätze die automatische Generierung von beispielsweise Testfällen aus Anforderungsspezifikationen [70] [54]. Im Folgenden werden repräsentative Ansätze, die die Validierung der Anforderungen unterstützen, im Detail vorgestellt.

Szenarienbasierte Anforderungsbeschreibung

Modelle zur Beschreibung von Szenarien wie ein MSC oder ein UML Kommunikationsdiagramm werden als szenarienbasierte Modelle bezeichnet und werden häufig zur Beschreibung von Anforderungen verwendet [51] [24] [94] [90]. Ein repräsentativer Ansatz, der szenarienbasierte Modelle zur Definition von Anforderungen verwendet, wird von Elkoutbi et al. [24] beschrieben. In diesem Ansatz werden UML Kollaborationsdiagramme (seit UML 2.0 Kommunikationsdiagramme) verwendet, um Anforderungen szenarienbasiert zu beschreiben. Die Typen der in den Kollaborationsdiagramm beschriebenen Instanzen werden durch UML Klassendiagramme beschrieben. Das Klassendiagramm ATM stellt beispielsweise eine Klasse ATM dar, die einen Geldautomaten beschreibt (siehe Abb. 2.25). Das Kommunikationsdiagramm *Withdraw* beschreibt ein Szenario, bei dem ein Benutzer (User) versucht Geld vom Automaten abzuheben (siehe Abb. Abb. 2.25). Hierfür gibt der Benutzer unter anderem die Karte und die Identifikationsnummer (PIN) ein (siehe Nachricht *1*). In diesem Szenario ist der Betrag vom Konto nicht gedeckt und der Versuch schlägt fehl. Der Automat antwortet daher mit der Nachricht “Insufficient funds“ (siehe Nachricht *7*).

Konkrete Szenarien sind für Interessenvertreter weniger komplex als vollständige Beschreibungen mit Schleifen und alternativen Ausführungssequenzen. Zudem werden bei diesen Szenarien Instanzen aus der realen Welt beschrieben wie beispielsweise Geldautomaten und Benutzer. Dies erhöht die Verständlichkeit für Interessenvertreter. Hierdurch kann eine eindeutige formale sowie leicht verständliche Beschreibung der Anforderungen für eine Validierung durch Interessenvertreter erreicht werden.

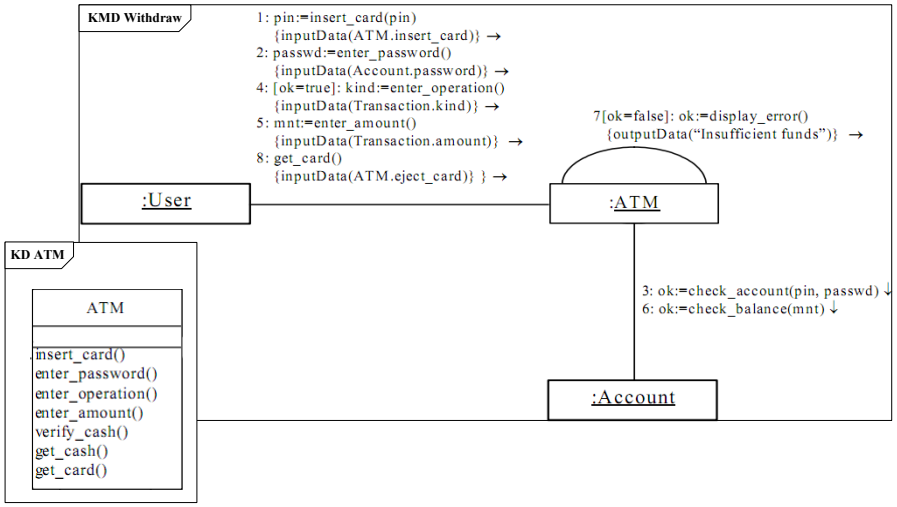


Abbildung 2.25: Beispiel szenarienbasierte Anforderungsbeschreibung nach [24]

Mockups

Ein geeigneter Ansatz Anforderungsmodelle mit Interessenvertretern abzustimmen ist die Visualisierung der Modelle mittels Mockups [75] [99]. Ein repräsentativer Ansatz, der Mockups zur Visualisierung von Anforderungsmodellen verwendet, wird von Zhang et al. [99] beschrieben. In diesem Ansatz wird eine Methode vorgestellt, die es erlaubt auf schnelle Weise ein Mockup eines Internetseiten-basierten Informationssystems zu erstellen. Dieser kann dann für ein frühe Abstimmung des Systems mit den Interessenvertretern verwendet werden. Die Erstellung eines Mockups erfolgt in neun

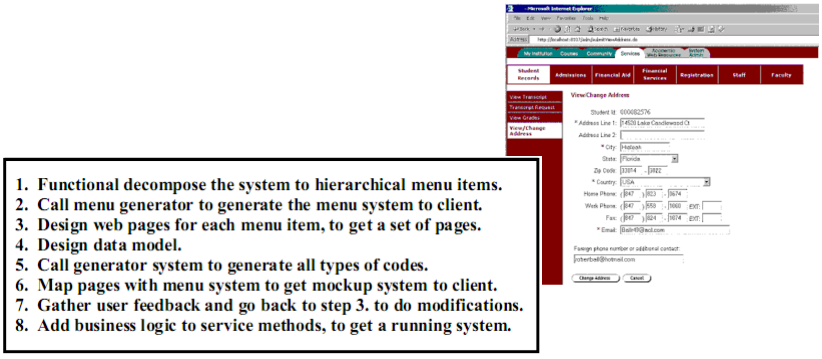


Abbildung 2.26: Beispiel szenarienbasierte Anforderungsbeschreibung nach [99]

Schritten (siehe Abb. 2.26). Im ersten Schritt wird beispielsweise die Menüstruktur aus den Funktionen des Systems abgeleitet und formal durch eine Baumstruktur beschrieben. Mit Hilfe eines Generators wird anschließend automatisch ein auf einem Internets-

erver ausführbares und aufrufbares Menü erzeugt. Dieses Menü wird mit den Interessenvertretern abgestimmt. Anschließend erfolgt die Ableitung der zu verarbeitenden Daten sowie der Entwurf der Seiten der einzelnen Menüeinträge. Die Datenstrukturen und die Seitenentwürfe werden ebenfalls in einem formalen Modell beschrieben. Aus diesem Modell wird ein Mockup der Internetseite generiert, das auf einem Internetserver ausführbar und aufrufbar ist (siehe Abb. 2.26 Schritt 6). Dieses Mockup wird anschließend für eine frühe Abstimmung des Systems mit den Interessenvertretern verwendet.

Prototypische Ausführung und Simulation

Ein weiterer Ansatz Anforderungsmodelle mit Interessenvertretern abzustimmen, ist die prototypische Ausführung und Simulation der Modelle [24] [47]. Ein repräsentativer Ansatz dieser Art wird von Knieke et al. [47] beschrieben. Bei diesem Ansatz werden UML Aktivitätsdiagramme zur Beschreibung von Anforderungen verwendet. Es wird eine Ausführungssemantik der Aktivitätsdiagrammspezifikationen formal definiert. Beispielsweise wird der Zustand eines Kontrollflusses innerhalb eines Aktivitätsdiagramms über eine *Location* bestimmt. Am Aktivitätsdiagram *AD Y* sind beispielsweise Locations an den Kontrollknoten visualisiert (siehe Abb. 2.27). Die Fortsetzung

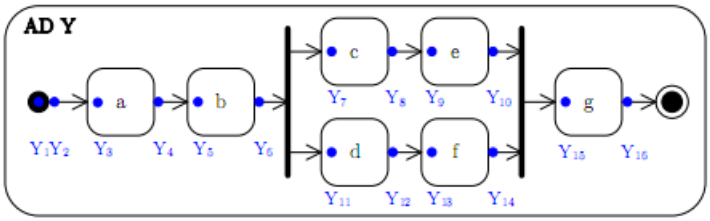


Abbildung 2.27: Aktivitätsdiagramm-Semantik mit Locations nach Knieke [47]

des Kontrollflusses bei einem bestimmten Zustand wird über einen Schrittalgorithmus definiert, der die Semantik aller gängigen Konstrukte von UML Aktivitätsdiagrammen beschreibt. Diese ermöglicht eine prototypische Ausführung der Spezifikation. Da unter anderem der Datenfluss durch ein internes Objektmodell vollständig beschrieben werden kann, ist eine unterbrechungsfreie Ausführung möglich. Daher eignet sich dieser Ansatz insbesondere für die Abstimmung eines Modells, welches Anforderungen an ein reaktives System wie beispielsweise eines Roboters festlegt.

2.3.3 Modellbasierter Architekturentwurf

Modellbasierte und modellgetriebene Ansätze zum Architekturentwurf stellen Modelle zur Beschreibung von Architekturen zur Verfügung. Modelle zur Beschreibung einer Architektur unterstützen häufig die Beschreibung des Softwaresystems in mehreren Abstraktionsebenen [65][46]. Zudem unterstützen die Modelle zumeist eine automatisierte Weiterverarbeitung beispielsweise zur Generierung von konkreteren Modellen oder von Quellcode. In den letzten Jahren wurden verstärkt Ansätze entwickelt,

die speziell die Entwicklung einer komponentenbasierten Architektur durch Modelle unterstützen. Diese Modelle werden auch Komponentenmodelle genannt [68][5]. Im Folgenden werden repräsentative Ansätze zum modellbasierten und modellgetriebenen Architekturentwurf vorgestellt.

Model Driven Architecture

Model-driven Architecture (MDA) [65][46] ist ein konkreter Ansatz zur standardisierung des modellgetriebenen Architekturentwurfs. MDA wurde von der OMG entwickelt und wird seit Ende 2000 von dieser kommuniziert. Die grundlegende Idee der MDA ist die Architekturmodelle präzise genug zu definierten, um eine automatisierte Unterstützung des Entwurfs zu ermöglichen [65]. Konkretere Modelle werden hierbei aus abstrakteren Modellen generiert. Die Generierung erfolgt durch eine Modelltransformation, die durch ein Transformationswerkzeug realisiert wird. Eine schematische Darstellung des MDA-Prozesses kann der Abb. 2.28 entnommen werden. Um eine der-

Abbildung 2.28: MDA-Prozess nach Kleppe et al. [46]

artige Generierung zu automatisieren muss die Struktur der Modelle eindeutig und formal beschrieben sein. Hierfür wird das Konzept der Metamodellierung verwendet. Ein weiterer Vorteil dieser Vorgehensweise ist die Wiederverwendbarkeit der abstrakten Modelle, um konkretere Modelle für verschiedene Zielplatformen generieren zu können.

MDA stellt für die Entwicklung der Architektur drei Modelle zur Verfügung. Diese Modelle werden *Computation Independent Model* (CIM), *Platform Independent Model* (PIM) und *Platform Specific Model* (PSM) genannt. Mit Hilfe des CIM wird das System unabhängig von konkreten Hard- oder Softwaresystemen beschrieben [65]. Beispielsweise kann ein CIM die Struktur der Domäne durch ein UML Klassendiagramm beschreiben. Die Beschreibung des CIM ist nicht vollständig formal und dient als Grundlage zur manuellen Erstellung des PIM. Das PIM dient der formalen Beschreibung der rein fachlichen Aspekte des zu entwickelnden Systems weiterhin unabhängig von konkreten Hard- und Softwaresystemen. Durch die Formalisierung wird eine automatisierte Weiterverarbeitung durch beispielsweise die Generierung des PSM ermöglicht. Das PSM ist ein Modell des zu entwickelnden Systems, welches sich auf konkrete Hard- und Softwaresysteme bezieht. Das PSM legt sich damit auf diese Systeme fest. Hierbei kann es mehrere Ebenen der Spezifizierung geben. Konkreter als die Festlegung auf eine Programmiersprache ist beispielsweise die Festlegung des Betriebssystems oder der konkreten Hardware. Nach MDA wird abschließend aus dem PSM automatisch Code generiert.

Komponentenmodelle

Komponentenmodelle sind modellbasierte oder modellgetriebene Ansätze, die speziell die Entwicklung einer komponentenbasierten Architektur unterstützen[68]. In den letzten Jahren wurden Komponentenmodelle für vielfältige Einsatzzwecke entwickelt. Das Ziel des Komponentenmodells *Component-Interaction Automata Approach* (CoIn) [101] ist beispielsweise die Unterstützung der formalen Analyse des Verhaltens großer Systeme. Das Verhalten des zu untersuchenden Systems wird durch Automaten beschrieben, deren Transitionen mit Eingabe- und Ausgabeaktionen verbunden sind. Die Analyse erfolgt durch den Model-Checker DiVinE [6], der die Zustände des Komponentenmodells auf verschiedene Eigenschaften prüfen kann. Bei der Analyse nach CoIn werden beispielsweise die Auswirkungen von Serviceaufrufen und die Kommunikationsgerechtigkeit geprüft. Das Komponentenmodell DisCComp [4] stellt unter anderem eine formale Semantik der verwendeten Modelle zur Verfügung, um den Bruch zwischen Komponentenmodellen und komponentenbasierten Programmiermodellen zu schließen. Die Definition der formalen Semantik erfolgt hierbei durch die Beschreibung von Systemzuständen mittels *Snapshots* und Übergangsregeln. Die Semantik von DisCComp berücksichtigt hierbei alle Konzepte der Programmiermodelle für Komponenten wie asynchrone Kommunikation und sich dynamisch verändernde Strukturen. Einige Komponentenmodelle unterstützen auch den MDA-Ansatz. Im Folgenden werden die zur Verfügung gestellten Modelle eines repräsentatives Komponentenmodell dieser Art vorgestellt.

Das Komponentenmodell KobrA

KobrA [5] ist ein Komponentenmodell, welches eine UML-basierte Methode zur Beschreibung von Komponenten und komponentenbasierten Systemen zur Verfügung stellt. Das Hauptziel von KobrA ist den Entwurf eines komplexen Systems zu ermöglichen, welches in Komponenten zerlegt werden kann. Diese Komponenten stellen wiederum komplexe Systeme dar, die einzeln entworfen werden können. Im KobrA-Ansatz wird entsprechend des MDA-Ansatzes zwischen verschiedenen Abstraktionsebenen durch das CIM, PIM und PSM unterschieden. Des Weiteren erfolgt eine Unterscheidung zwischen Spezifikation und Realisierung. Eine Spezifikation definiert hierbei welche Tätigkeiten ein System durchführt. Eine Realisierung definiert wie das System diese Tätigkeit durchführt. Zudem wird zwischen Struktur-, Funktions- und Verhaltensprojektionen unterschieden, die verschiedene Sichten auf das Komponentenmodell darstellen. Jede Spezifikation und Realisierung wird in KobrA durch diese drei Projektionen beschrieben.

Auf der Ebene des CIM wird der Kontext des Systems beschrieben. Diese Beschreibung stellt eine Spezifikation des Systems dar. Auf der Ebene des PIM und PSM wird jeweils eine Realisierung und eine Spezifikation beschrieben. Das PIM beschreibt hierbei die Realisierung der Spezifikation des CIM und das PSM die Realisierung der Spezifikation des PIM. Das Aktivitätsdiagramm *cim* stellt beispielsweise die Verhaltensprojektion des Kontextes zu einem Handelssystem auf der Ebene des CIM dar, welches den Ablauf eines Geschäftsprozesses definiert (siehe Abb. 2.29). Unter anderem wird der Ablauf eines Geschäftsprozesses nach Eintreffen einer Bestellung beschrie-

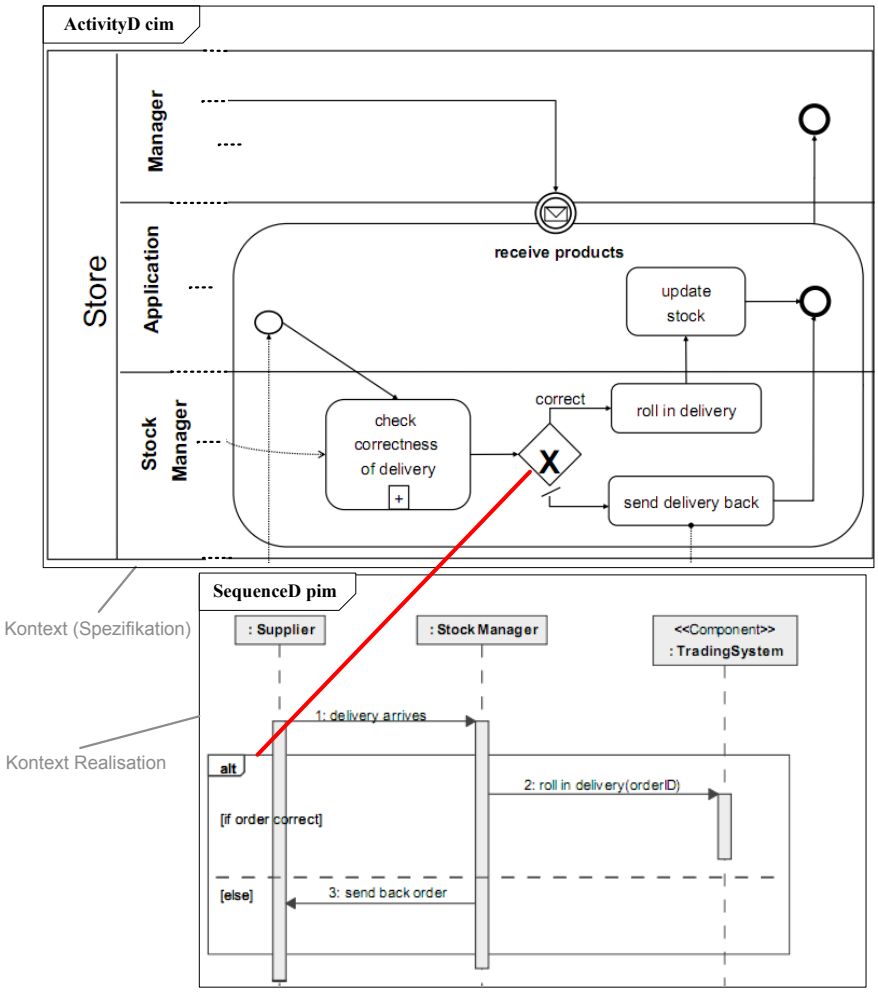


Abbildung 2.29: KobrA Spezifikation und Realisierung nach Atkinson [5]

ben. Die Bestellung wird im Rahmen der Aktion *check correctness of delivery* geprüft. Anschließend wird abhängig vom Prüfungsergebnis die Bestellung eingelagert oder zurückgesendet. Das UML Sequenzdiagramm *pim* stellt die Funktionsprojektion der Kontextrealisierung auf der Ebene des PIM dar. Diese beschreibt die Realisierung des Geschäftsprozesses mit Zuhilfenahme des Handelssystems. Bei Einlagerung der Bestellung wird diese im Handelssystem eingetragen. Die zugehörigen Struktur-, Funktions- und Verhaltensprojektionen des Handelssystems werden in [5] genauer beschrieben. Bei der Beschreibung der Realisierung werden die Funktionen der verwendeten Systeme oder Komponenten definiert. Beispielsweise wird im Sequenzdiagramm *pim* die Funktion *roll in delivery* des Handelssystems definiert (siehe Abb. 2.29). Im weiteren Entwicklungsverlauf wird aus dieser Beschreibung die Spezifikationen der

Systeme und Komponenten abgeleitet. In der jeweils folgenden Detaillierungsebene wie beispielsweise das PSM wird wiederum die Realisierung dieser Spezifikationen festgelegt. Das vollständige Beispiel des Handelsystems kann der Beschreibung in [5] entnommen werden.

2.3.4 Kombinierte modellbasierte Ansätze

Kombinierte modellbasierte Ansätze stellen sowohl Modelle für die Anforderungsspezifikation als auch für den Architekturentwurf zur Verfügung. Zudem werden Beziehungen zwischen den Modellen definiert. Beziehungen werden beispielsweise direkt oder durch Zwischenmodelle beschrieben. Bei einigen Ansätzen dient die Definition der Beziehungen unter anderem der Abstimmung und Verfeinerung von Anforderungen und Architekturen mit Interessenvertretern [31]. Bei anderen Ansätzen ist das Hauptziel der Definition der Beziehungen die Nachvollziehbarkeit und Überprüfung der Zusammenhänge zwischen Anforderungen und Architekturen [14][44]. Im Folgenden werden repräsentative kombinierte modellbasierte Ansätze kurz vorgestellt.

CBSP

Component Bus System Property (CBSP) stellt einen Ansatz zur systematischen Abstimmung und Verfeinerung von Anforderungen und Architekturen mit Interessenvertretern zur Verfügung. Für die Beschreibung der Beziehung zwischen Anforderungen und Architekturen wird ein Zwischenmodell verwendet, welches in der Lage ist komplexe Zusammenhänge abzubilden. Im Ansatz wird zudem ein Prozess zur Erstellung des Modells definiert, der die Zusammenarbeit mit den Interessenvertretern zur Abstimmung berücksichtigt.

In einem ersten Schritt werden Anforderungen für die nächste Iteration selektiert, um den Umfang in großen Entwicklungsvorhaben zu begrenzen. Die Selektion kann beispielsweise auf einer Priorisierung der Interessenvertreter beruhen. In einem zweiten Schritt werden Anforderungen entsprechend einer Taxonomie klassifiziert. Klassen können hierbei beispielsweise Anforderungen an die Komponenten (Component) des Systems und Anforderungen an die Verbindung von Komponenten (Bus) sein. Zudem wird die Relevanz der Anforderungen für diese Klasse bestimmt. Die Beschreibung der Zuordnung einer Anforderung zu einer Klasse erfolgt nach den definierten Regeln des CBSP-Metamodells (siehe Abb. 2.30). Eine Anforderung wird hierbei über ein CBSP-Element einem Architekturelement zugeordnet. Der spezielle Typ des CBSP-Elements bestimmt die Klassifizierung. Die Klassifizierung wird durch mehrere Architekten vorgenommen. Widersprüche werden in einem dritten Schritt identifiziert und behoben, da diese oftmals Missverständinisse darstellen. Im vierten Schritt werden Anforderungen, die mehreren Klassen zugeordnet sind aufgeteilt und verfeinert. Beispielsweise kann eine Anforderung sowohl eine Anforderung an die Komponenten als auch an die Verbindungen darstellen. Dies ist beispielsweise der Fall bei der Anforderung: Das System soll eine Schnittstelle zu einem Webbrowser bereitstellen [31]. Die Architektur, die diese Anforderungen erfüllen soll, wird im fünften Schritt entwickelt. Die Anforderungen dürfen hierfür keine überlappenden oder widersprüchliche Klassifizierungen

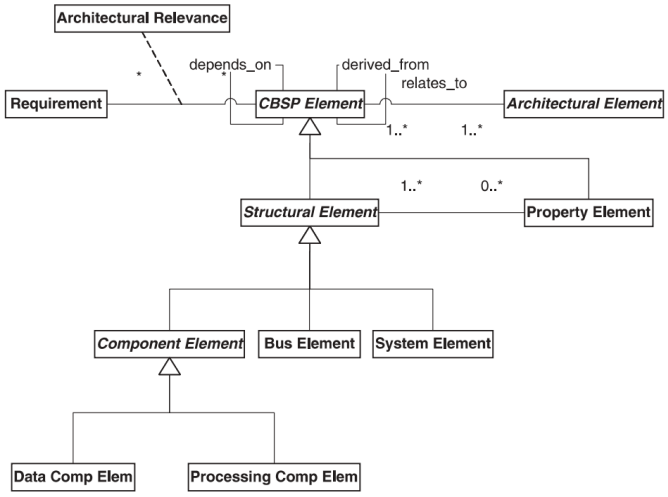


Abbildung 2.30: CBSP Metamodell nach Grünbacher [31]

aufweisen. Eine Anwendung des CBSP-Ansatzes in einem realen Projekt wird in [96] beschrieben.

COMPASS

Composition-Centric Mapping of Aspectual Requirements to Architectures (COMPASS) ist ein kombinierter modellbasierter Ansatz zur Entwicklung von Anforderungen und Architekturen [14]. Ein Ziel des Ansatzes ist die Nachvollziehbarkeit und Überprüfung der Zusammenhänge zwischen Anforderungen und Architekturen. Im Rahmen dieses Ansatzes wird eine Beschreibungstechnik für Anforderungen und Architekturen definiert. Zudem wird die Beziehung zwischen Anforderungen und Architekturen durch eine Abbildung von Anforderungselementen auf Architekturelemente beschrieben. Der Prozess zur Entwicklung von Anforderungen und Architekturen erlaubt einen Informationsaustausch zwischen Anforderungserhebung und Architekturentwurf in beide Richtungen.

Anforderungen an ein System werden nach COMPASS mit Hilfe einer RDL beschrieben. Ein Metamodell definiert die zur Verfügung stehenden Sprachkonstrukte der RDL. Beispielsweise kann eine Anforderung mehrere Subjekte (Subject) und Objekte (Object) beschreiben (siehe Abb. 2.31). Die Architektur eines Systems werden nach COMPASS mit Hilfe einer ADL beschrieben. Ein Metamodell der ADL definiert hierbei die zur Verfügung stehenden Sprachkonstrukte. Beispielsweise können Komponenten (Component) aus weiteren Komponenten zusammengesetzt werden (Composite) und einen Zustand haben (State) (siehe Abb. 2.31). Mit Hilfe einer Abbildung von Elementen des RDL-Metamodells auf Elemente des ADL-Metamodells werden die Beziehungen zwischen Anforderungen und Architekturen definiert. Das Metamodell-Element Subjekt der RDL wird beispielsweise auf das Metamodell-Element Komponente abgebildet (siehe Abb. 2.31). Diese Beziehungen können dann im Entwicklungs-

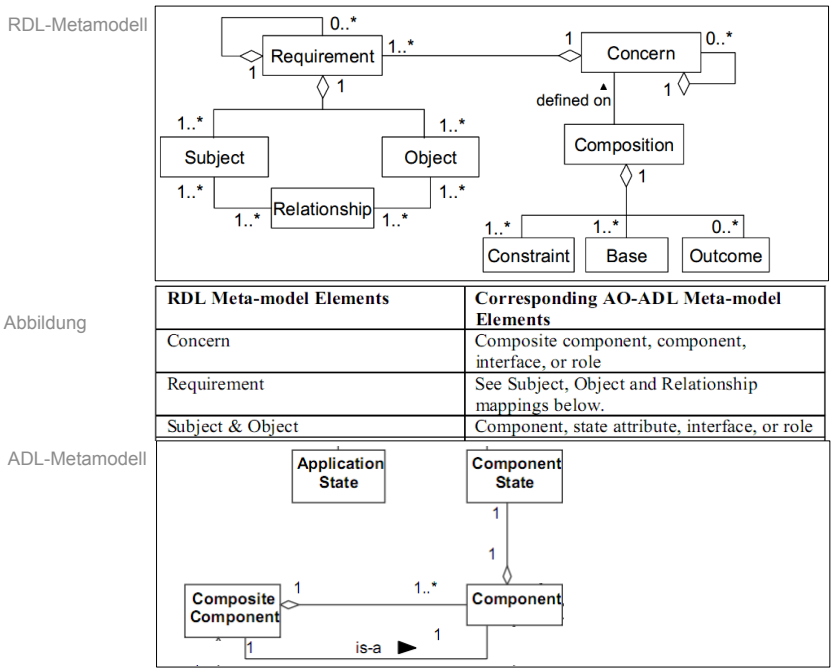


Abbildung 2.31: Abbildung RDL auf ADL nach [14]

verlauf zur Rückverfolgbarkeit und zur Überprüfung der Anforderungen und Architekturen verwendet werden.

2.4 Konsistenzsicherungsverfahren

Verfahren zur Konsistenzsicherung von Modellen können kategorisiert werden in Modelltransformationsverfahren und Konsistenzüberprüfungsverfahren. Die Modelltransformation ist ein grundlegendes Verfahren der MDA [77] und ermöglicht unter anderem die Transformation abstrakter Modelle zu initialen konkreteren Modellen. Zur Transformationen von Modellen stellt die OMG den *Queries/ Views/ Transformation* (QVT) [60] Standard zur Verfügung. Für eine Modelltransformation existieren sowohl unidirektionale Verfahren als auch bidirektionale Verfahren [21]. Im Folgenden werden diese Verfahren beschrieben und repräsentative Ansätze vorgestellt.

2.4.1 Unidirektionale Modelltransformation

Eine unidirektionale Modelltransformation transformiert ein Quellmodell zu einem Zielmodell. Das Zielmodell ist im Gegensatz zum bidirektionalen Fall kein Eingabeparameter der Transformation [86]. Hierdurch wird das Zielmodell nicht bei der Transformation berücksichtigt. Czanecki et al. beschreibt in einer Arbeit [21] mögliche Klassifizierungen für Modelltransformationsverfahren, die überwiegend unidirektional sind.

In [1] beschreiben Agrawal et al. beispielsweise ein unidirektionales Modelltransformationsverfahren zwischen domänenspezifischen Modellen wie beispielsweise hierarchische und parallele Zustandsdiagramme und endliche Automaten.

Szenarienbasierte Verhaltensbeschreibungen im Sinne von Liang [51] wie beispielsweise UML Sequenzdiagramme sind für die Beschreibung von Anforderungen geeignet [40] und beschreiben Interaktionen zwischen Objekten. Für den Architekturentwurf sind zustandsbasierte Verhaltensbeschreibungen im Sinne von Liang [51] wie beispielsweise höhere Petrinetze geeignet. Speziell für die Transformation von szenarien- zu zustandsbasierten Modellen stehen Syntheseverfahren zur Verfügung [51]. Diese Ansätze ermöglichen eine automatisierte Generierung vollständiger zustandsbasierter Modelle aus Szenarien. Mehrere dieser Ansätze synthetisieren zustandsbasierte Modelle aus MSCs. Diese werden beispielsweise in [49], [78] und [12] beschrieben. Im Ansatz [49] werden Zustandsautomaten synthetisiert. In der MSC-Spezifikation werden hierbei Zustände integriert. Anschließend werden aus der MSC-Spezifikation Zustandsautomaten generiert, indem unter anderem Nachrichten zu Transitionen transformiert werden. Das Zustandsdiagramm *sc* wird beispielsweise aus dem MSC *msc* synthetisiert (siehe Abb. 2.32). In das MSC werden hierfür die Zustände *z1* und *z2* vor der

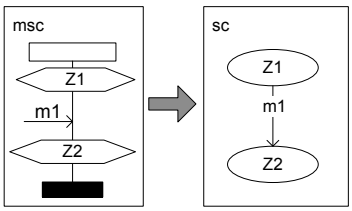


Abbildung 2.32: Synthese eines Zustandsautomaten aus einem MSC nach [49]

Nachricht *m1* integriert. Anschließend wird das Zustandsdiagramm sc1 synthetisiert, indem unter anderem die Nachricht *m1* zu der Transition *m1* transformmiert wird. In [78] wird ein Verfahren zur Synthese von Petrinetzen aus MSCs vorgestellt. Eine weitere große Teilmenge der Syntheseansätze synthetisieren zustandsbasierte Modelle aus UML Sequenzdiagrammen. Diese werden beispielsweise in [97], [25] und [52] vorgestellt. Whittle und Schuhmann beschreiben beispielsweise ein Verfahren zur Synthese von Zustandsdiagrammen aus UML Sequenzdiagrammen [97]. Für jedes Sequenzdiagramm wird hierbei zu jedem Objekt ein Zustandsdiagramm generiert. Die Nachrichten des Sequenzdiagramms an das Objekt werden zu Ereignissen des Zustandsdiagramms und Nachrichten vom Objekt werden zu Aktionen. In [25] wird ein Verfahren zur Synthese von Petrinetzen aus Sequenzdiagrammen vorgestellt.

2.4.2 Bidirektionale Modelltransformation

Eine bidirektionale Modelltransformation berücksichtigt das Zielmodell bei der Transformation eines Quellmodells zu einem Zielmodell. Das Zielmodell ist hierfür ebenfalls ein Eingabeparameter der Transformation [86]. In der Arbeit von Czanecki et al. [21] werden die Verfahren [2] und [19] genannt, die eine bidirektionale Modelltrans-

formation unterstützen. In der Arbeit von Stevens [86] werden bidirektionale Transformationsverfahren in QVT allgemein betrachtet. Nach Stevens ist die Transformation im Sinne einer Bijektion ein spezialfall einer bidirektionalen Transformation [86]. Bei einer Bijektion beinhaltet das eine Quellmodell die selben Informationen wie das Zielmodell. Konkrete bidirektionale Transformationsverfahren im Sinne einer bijektion werden beispielsweise in [82] und [29] beschrieben. In [82] erfolgt beispielsweise eine bidirektionale Modelltransformation zwischen UML-Aktivitätsdiagrammen und Petrinetzen. Ein bijektive Abbildung zwischen UML Sequenz- und Aktivitätsdiagrammen wird in [29] beschrieben.

Die Regel ist nach Stevens [86] jedoch eine bidirektionale Modelltransformation, bei der ein Modell mehr Informationen beinhalten kann als das andere. Das eine Modell ist dann eine Abstraktion des anderen Modells. In diesem Falle ist die Transformation nicht bijektiv. Bidirektionale Modelltransformationsverfahren, die Abstaktion unterstützen, werden in [28], [36], [22], [23] und [7] beschrieben. Diese Verfahren verwenden zur Abbildung eines konkreten Modells auf ein abtraktes Modell überwiegend sogenannte Linsen (lenses) [28]. Das konkrete Modell kann hierbei beispielsweise Klassen beschreiben (siehe Abb. 2.33). Bei einer Linse wird ein Quellmodell mit einem

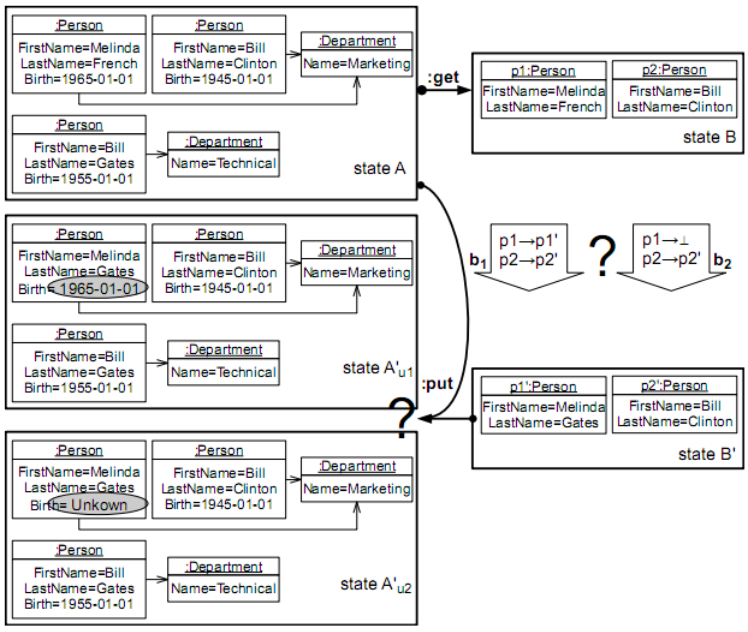


Abbildung 2.33: Bidirektionale Modelltransformation mit Linsen nach [22]

Modell synchronisiert, welches eine Sicht auf das Quellmodell darstellt. Eine Linse unterstützt zwei Operationen. Die erste Operation wird *get* genannt und berechnet die Sicht auf das Quellmodell. Die zweite Operation wird *put* genannt und berechnet das Quellmodell, wenn Änderungen an der Sicht vorgenommen wurden. In der Arbeit von Diskin et al. [22] wird beispielsweise ein Verfahren zur bidirektionalen Transformation

zwischen zwei Objektmodellen mittels Linsen beschrieben. Das Quellmodell des verwendeten beispiels beschreibt Personen mit Vor-, Nachnamen und Geburtsdatum (siehe Abb. 2.33). Mit Hilfe der Funktion *get* wird die Sicht auf das Modell berechnet, in der lediglich die Vor- und Nachnamen der Personen enthalten sind. Wird in der Sicht der Nachname der Person *Melinda* auf *Gates* geändert, kann mit Hilfe der Funktion *put* die Änderung in das Zielmodell übernommen werden. Unterschiedliche Interpretationen der selben Änderung können verschiedene Auswirkungen auf das Zielmodell haben. Beispielsweise könnte die neue Sicht auch entstanden sein, indem die Person *Melinda* gelöscht und eine neue Person mit dem selben Vornamen eingeführt wurde (siehe Abb. 2.33). Diese Problematik wird in [22] durch den Einsatz von *delta-lenses* gelöst. Diese ermöglichen Änderungen eindeutig zu beschreiben. Die Beschreibungen werden in die Modelltransformation einbezogen. In [23] wird auch der Fall betrachtet in dem ein Modell eine Abstraktion eines anderen Modells ist und umgekehrt. Häufig existieren auch mehrere Anpassungsmöglichkeiten des Zielmodells der Transformation. In vielen Fällen ist dann eine manuelle Entscheidung des Benutzers erforderlich. Ein derartiger Ansatz wird in der Arbeit von Becker et al. [7] vorgestellt.

2.4.3 Konsistenzüberprüfung

Konsistenzüberprüfungsverfahren prüfen zwei Modelle hinsichtlich ihrer Konsistenz. Zwei Modelle sind konsistent oder widerspruchsfrei, wenn festgelegte Bedingungen erfüllt sind. Konsistenzüberprüfungsverfahren können als bidirektionale Modelltransformationen angesehen werden [86]. Die Ergebnisse der Konsistenzüberprüfung werden für eine manuelle Anpassung des Quell- oder Zielmodells verwendet. Anschließend wird wieder automatisiert die Konsistenz überprüft. Der Einsatz einer Konsistenzüberprüfung für eine bidirektionale Modelltransformation bietet sich an, wenn viele manuelle Entscheidungen getroffen werden müssen [86].

Konsistenzüberprüfungsverfahren können in Verfahren eingeteilt werden, die auf Konsistenz im Sinne einer Bijektion oder mittels *Model Checking* [16] prüfen. Verfahren der ersten Kategorie werden beispielsweise in mehreren Ansätzen zur Synthese von zustandsbasierten zu szenarienbasierten Modellen vorgestellt [51]. In [45] werden UML-Kollaborationsdiagramme zu partiellen Objektspezifikationen in Form von Zustandsdiagrammen transformiert. Anschließend werden diese zu vollständigen Objektspezifikationen zusammengefügt. Hierbei erfolgen Konsistenzüberprüfungen. Ein Zustandsdiagramm ist inkonsistent, wenn es nicht-deterministisch ist. Die Kollaborationsdiagramme sind inkonsistent, wenn diese bezüglich des zustandsbasierten Modells nicht vollständig sind. In den Ansätzen [97] und [34] erfolgt die Konsistenzüberprüfung zwischen den Szenarien. Darauf folgend wird das zustandsbasierte Modell generiert. Im Ansatz [97] werden UML-Zustandsdiagramme aus Sequenzdiagrammen generiert. Durch das Hinzufügen semantischer Informationen zu den Sequenzdiagrammen in Form von Zustandsvektoren können gleiche Zustände identifiziert und Inkonsistenzen festgestellt werden. Die Ansätze [3], [11] und [15] basieren auf einer Konsistenzüberprüfung von szenarienbasierten und zustandsbasierten Modellen mittels *Model Checking*. In der Arbeit von Aoki et al. [3] wird ein Konsistenzüberprüfungsverfahren von UML Aktivitätsdiagrammen und Zustandsdiagrammen vorgestellt. Die Konsistenzüberprüfung erfolgt durch den Model Checker UPPAAL [8]. UML-

Diagramme werden hierfür zu UPPAAL-Modellen transformiert. In Abb. 2.34 wird eine Transformation von einem UML Aktivitätsdiagramm zu ein UPPAAL-Modell gezeigt. Aktivitätsknoten werden hierbei zu Locations im UPPAAL-Modell transfor-

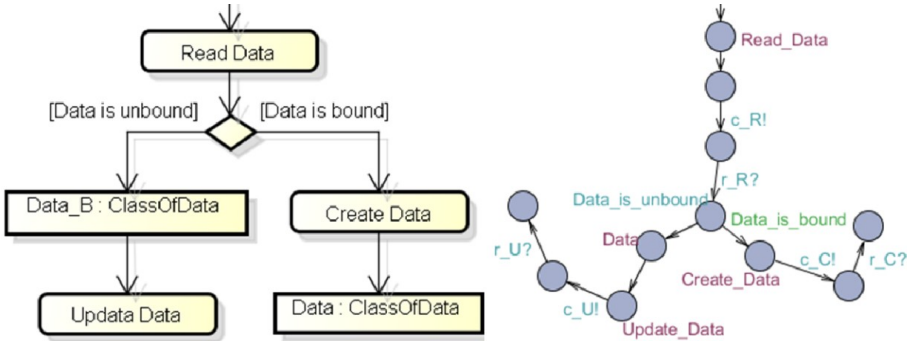


Abbildung 2.34: Transformation eines UML-Aktivitätsdiagramms zu UPPAAL [3]

miert und Kanten zu Transitionen [3]. Aktionen, die Daten verändern, werden beispielsweise zu drei Locations transformiert, die den Zustand vor, während und nach der Aktion beschreiben. Die Aktion *Create Data* wird beispielsweise zu der Location *Create_Data* sowie den Locations mit den eingehenden Kanten $c_C!$ und $r_C?$ transformiert. $c_C!$ und $r_C?$ stehen hierbei für Kanäle, die den Aufruf resp. die Rückkehr beschreiben. Diese Kanäle werden mit einem anderen UPPAAL-Modell synchronisiert, welches für das UML Zustandsdiagramm steht. Bei der Konsistenzüberprüfung wird beispielsweise die Durchführbarkeit von Aktionen nach der Synchronisierung geprüft. In der Arbeit von Choi und Bunse [15] wird hingegen ein Verfahren zur Konsistenzüberprüfung für den Modellierungsansatz KobrA [5] beschrieben. Die Konsistenzüberprüfung erfolgt zwischen Aktionssystemen und reaktiven Systemen, die beispielsweise aus UML Aktivitäts- oder Sequenzdiagrammen generiert werden. Ein reaktives System führt hierbei eine Folge von Aktionen aus. Zwischen den Aktionen erfolgt jeweils ein Ereignis. Beim Aktionssystem erfolgt kein Ereignis zwischen den Aktionen. Ein reaktives System ist konsistent zu einem Aktionssystem, wenn das reaktive System verklemmungsfrei bei den durch das Aktionssystem ausgelösten Ereignissen ist. Die Konsistenzüberprüfung erfolgt durch den Model-Checker SPIN [37]. SPIN transformiert hierfür die Modelle in endliche Automaten. In [11] ist das szenarienbasierte Modell eine Menge von *universal Life Sequence Charts* (uLSCs). Das zustandsbasierte Modell wird durch Input-Output-Automaten (I/O-Automaten) beschrieben. Mittels *Model Checking* wird geprüft, ob die I/O-Automaten den uLSCs genügen.

Kapitel 3

Modellbasierter Ansatz zur Automatisierung der Konsistenzsicherung

Für eine Automatisierung der Konsistenzsicherung von Anforderungen und Architekturen müssen Konsistenzbedingungen formal definiert werden. Eine Voraussetzung hierfür ist eine konkrete Beschreibungstechnik. Im Rahmen des Forschungsprojekts *ReqBw* haben wir den modellbasierten Ansatz CREATE [40] entwickelt, der die iterative und evolutionäre Enwicklung von Anforderungen und Architekturen unterstützt. Der Ansatz wurde iterativ weiterentwickelt und im Rahmen von realen Entwicklungsprojekten im praktischen Einsatz getestet. CREATE definiert sowohl eine konkrete Technik für die Beschreibung von Anforderungen und Architekturen als auch Querbezüge zwischen den Beschreibungen. Der Ansatz erfüllt zudem die in Kapitel 1 aufgeführten Richtlinien. CREATE bietet daher eine konkrete Beschreibungstechnik, die als Grundlage zur Automatisierung der Konsistenzsicherung von Anforderungen und Architekturen geeignet ist. Das grundlegende Konzept von CREATE wird in [40] beschrieben. Im Folgenden wird dieses Konzept aus der Arbeit in [40] kurz vorgestellt. Anschließend werden die für diese Arbeit relevanten Diagramme der Beschreibungstechnik und die Querbezüge vollständig und im Detail eingeführt.

3.1 Grundlegendes Konzept

Der CREATE-Prozess für die iterative und evolutionäre Entwicklung von Anforderungen und Architekturen basiert auf dem *Twin Peaks*-Modell [56][40]. Bei diesem Modell haben Anforderungen und Architekturen den gleichen Stellenwert und werden iterativ weiterentwickelt. Dieser Sachverhalt wird im Modell durch die zwei Bergspitzen (*Twin Peaks*) dargestellt (siehe Abb. 3.1). CREATE konkretisiert das *Twin Peaks*-Modell, indem eine geeignete und konkrete Beschreibungstechnik für Anforderungen und Architekturen definiert wird. Der Ansatz ist domänenspezifisch für interaktive In-

formationssysteme wie webbasierte Systeme oder moderne Kommunikationssysteme. Der CREATE-Prozess beginnt mit der Entwicklung einer initialen Version des Anforderungsmodells. Anschließend wird die Architektur entwickelt. Bei der Entwicklung der Architektur erfolgt eine stetige Konsistenzsicherung mit den Anforderungen.

Der wesentliche Bestandteil des Ansatzes ist die konkrete Beschreibungstechnik für Anforderungen und Architekturen, welche eindeutige Querbezüge zwischen den Beschreibungen definiert [40]. Anforderungen und Architekturen werden durch Diagramme beschrieben. Die Diagramme des Ansatzes werden innerhalb der Rauten an den Bergspitzen für Anforderungen und Architekturen des *Twin Peaks*-Modells dargestellt (siehe Abb. 3.1). Eine eindeutige Beschreibung der erhobenen Anforderun-

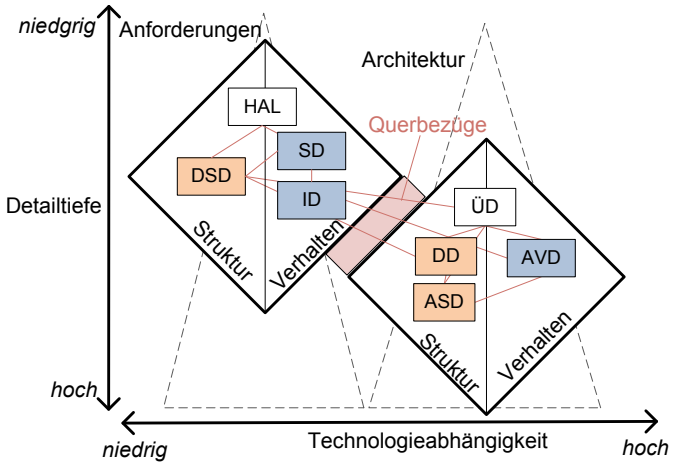


Abbildung 3.1: Überblick über den modellbasierten Ansatz [40]

gen erfolgt in CREATE durch die Modellierung von Szenarien. Hierbei werden lediglich repräsentative und konkrete Szenarien beschrieben, um die Verständlichkeit der Modelle für Interessenvertreter zu erhöhen. Die Validierung der Anforderungen wird weiterhin verbessert durch die Kombination dieser Modelle mit Modellen, die eine Visualisierung der Anforderungen durch Dialogskizzen erlauben. Eine Architektur in CREATE beschreibt das Verhalten und die resultierende Struktur des Softwaresystems eindeutig. Hierfür definiert CREATE ein Komponentenmodell. Die Diagramme des modellbasierten Ansatzes werden verwendet, um Struktur- und Verhaltensaspekte der Anforderungen und Architekturen zu beschreiben. Dies wird durch die Einteilung der Rauten in zwei Teile für Struktur- und Verhaltensdiagramme dargestellt (siehe Abb. 3.1). Beispielsweise werden Prozesse in der Domäne (z.B. Geschäftsprozesse) durch ein Szenariendiagramm (SD) beschrieben. Daher ist das SD dem Teil der Anforderungsraute für Verhaltensdiagramme zugeordnet (siehe Abb. 3.1). Einige Diagramme wie beispielsweise das Systemüberblicksdiagramm (ÜD) können verwendet werden, um sowohl strukturelle Aspekte als auch Verhaltensaspekte zu beschreiben. Im Abschnitt 3.1.1 wird die Beschreibungstechnik von CREATE aus unserer Arbeit in [40] kurz vorgestellt und in den Abschnitten 3.2 bis 3.7 werden die für diese Arbeit relevan-

ten Diagramme im Detail erklärt.

Die Konsistenzsicherung wird durch die genaue Definition von Querbezügen zwischen den Anforderungs- und Architekturbeschreibungen unterstützt. Zudem werden auch Querbezüge innerhalb dieser Beschreibungen definiert. Die Definition erfolgt durch Assoziationen und zusätzliche Syntaxeinschränkungen. Das Konzept der Querbezüge in CREATE aus unserer Arbeit in [40] wird im Abschnitt 3.1.2 kurz vorgestellt. In den Abschnitten 3.2 bis 3.7 werden alle Modelle und deren Querbezüge des modellbasierten Ansatzes CREATE im Detail beschrieben, die für diese Arbeit relevant sind.

3.1.1 Beschreibungstechnik

Die Beschreibungstechnik von CREATE wird in [40] anhand eines Fallbeispiels beschrieben. Im Folgenden wird die Beschreibungstechnik kurz schematisch erklärt.

Beschreibung der Anforderungen

Hauptgegenstand der Anforderungsbeschreibung mit CREATE sind die Szenarien. In vielen Fällen werden Anforderungen jedoch initial in Textform gestellt. Textbasierte Anforderungen werden in CREATE durch die hierarchische Anforderungsliste (HAL) (siehe Abb. 3.1) beschrieben. Textbasierte Anforderungen werden hierbei für eine Verfeinerung hierarchisch strukturiert. Die Anforderungsliste für das System kann beispielsweise aus einer Anforderung *1* und einer verfeinerten Anforderung *1.1* bestehen (siehe Abb. 3.2). Szenarien beschreiben konkrete Abläufe in einer Domäne (z.B. die Domäne Bibliothek). Die Struktur der Domäne (z.B. die Geschäftsstruktur) legt wichtige Anforderungen an das zu entwickelnde System fest. Diese Struktur wird in diesem Ansatz durch das Domänenstrukturdiagramm (DSD) beschrieben. Das DSD basiert auf dem UML Kompositionsstrukturdiagramm [58]. Eine Domänenstruktur besteht aus Systemen, Personen und Entitäten, welche miteinander kommunizieren können. Dies wird im DSD durch Teile und Verbindungen beschrieben. Die Domänenstruktur kann beispielsweise ein System, eine Menge von Benutzern und eine Menge von Produkten beinhalten (siehe Abb. 3.2). Ein wichtiger Aspekt in der Anforderungserhebung ist die Beschreibung von Prozessen in der Domäne (z.B. Geschäftsprozesse), die vom zu entwickelnden System unterstützt werden sollen. Prozesse in der Domäne werden durch das Szenariendiagramm (SD) eindeutig beschrieben, welches auf dem UML Kommunikationsdiagramm basiert [58]. Das SD beschreibt hierbei repräsentative Szenarien in der Domäne. Es beschreibt Nachrichtensequenzen zwischen Instanzen, die im DSD definiert wurden. Das Szenario *sd1* beschreibt beispielsweise eine Interaktion zwischen einer Instanz *b* vom Typ *Benutzer* und dem System (siehe Abb. 3.2). Die Nachrichten des SD werden zur Visualisierung für Interessenvertreter durch Interaktionsskizzen des Interaktionsskizzendiagramms (ID) genauer beschrieben. Die Visualisierung unterstützt die Validierung durch die Interessenvertreter. Interaktionsskizzen sind Dialogskizzen, die allgemein die Interaktionen des Systems beschreiben. Jede Nachricht des SD wird durch genau eine Interaktionsskizze des ID beschrieben. Das beschriebene Szenario kann visualisiert werden, indem die Interaktionsskizzen Schritt für Schritt angezeigt werden. Der Schritt *1* des ID *sd1_skizzen* zeigt beispielsweise die Skizze *Produktdaten*, die die Inhalte der Nachricht *1* beschreibt (siehe Abb. 3.2).

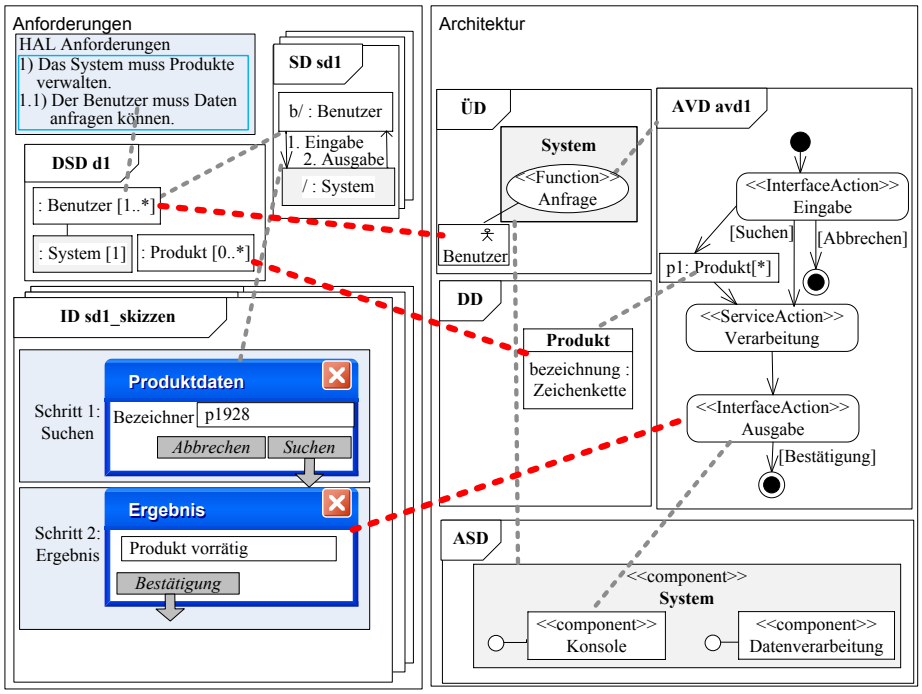


Abbildung 3.2: Beschreibungstechnik mit Querbezügen

Beschreibung der Architektur

Eine Architektur in CREATE beschreibt, wie die gegebenen Anforderungen durch das Softwaresystem erfüllt werden. Die Architektur beinhaltet hierfür eine vollständige Beschreibung des Verhaltens des Systems und der resultierenden Struktur. Ein erster wichtiger Schritt im Architekturentwurf ist die Definition der Systemgrenze und der vom System zur Verfügung gestellten Funktionen. Die Struktur und die Funktionen werden durch das Systemüberblicksdiagramm (ÜD) beschrieben, welches auf UML Anwendungsfalldiagrammen [58] basiert. Die Akteure und der Systemkontext werden vom DSD abgeleitet. Die Anwendungsfälle werden Funktionen genannt und vom SD abgeleitet. Ein ÜD kann beispielsweise den Akteur *Benutzer* und ein System mit der Funktion *Anfrage* beschreiben (siehe Abb. 3.2). Das Verhalten des Softwaresystems wird durch das architektonische Verhaltensdiagramm (AVD) beschrieben, welches auf UML Aktivitätsdiagrammen [58] inklusive Datenfluss basiert. Das AVD beschreibt den Prozess der Funktionen, die im ÜD definiert werden, vollständig. Die Funktion *Anfrage* wird beispielsweise durch die Aktivität *avd1* beschrieben (siehe Abb. 3.2). Innerhalb des AVD werden verschiedene Aktionstypen wie Schnittstellenaktion und Serviceaktion verwendet. Eine Serviceaktion ist eine Aktion, die vom System durchgeführt wird (z.B. ein Datenbankaufruf). Eine Schnittstellenaktion beschreibt eine Interaktion des Systems mit seiner Umgebung und steht hierfür mit einer Interaktionsskizze des ID in

Beziehung. Die Schnittstellenaktion *Ausgabe* ist beispielsweise mit der Interaktionsskizze *Ergebnis* verknüpft (siehe Abb. 3.2). Innerhalb eines AVD können Datenflüsse beschrieben werden. Die in den Datenflußbeschreibungen verwendeten Datentypen werden im Datendiagramm (DD) in Form von Entitäten beschrieben. Das DD basiert auf UML Klassendiagrammen [58]. Entitäten werden vom DSD abgeleitet. Das DD beschreibt Entitäten und deren Beziehungen zueinander detaillierter. Die Entität *Produkt* aus dem DSD kann beispielsweise durch das Datenobjekt *Produkt* mit dem Attribut *bezeichner* vom Typ einer Zeichenkette genauer beschrieben werden (siehe Abb. 3.2). Der interne Aufbau des zu entwickelnden Systems wird durch Das architektonische Strukturdiagramm (ASD) beschrieben. Das ASD basiert auf dem UML Komponentendiagramm [58] und beschreibt den internen Aufbau durch Komponenten und deren angebotene Schnittstellen als eine Blackbox. Anschließend werden die Komponenten weiter dekomponiert, um den internen Aufbau zu detaillieren. Der interne Aufbau wird hierbei von den Aktionen des AVD abgeleitet. Daher steht jede Komponente mit einer Aktion eines AVD in Beziehung. Die Komponente *System* wird beispielsweise durch eine Komponente *Konsole* verfeinert, die mit der Aktion *Ausgabe* in Beziehung steht (siehe Abb. 3.2).

3.1.2 Konsistenzbedingungen und Querbezüge

Die Konsistenzsicherung von Anforderungen und Architekturen wird in CREATE durch die Definition von Querbezügen unterstützt (siehe graue und rote Linien in Abb. 3.2). Ein Querbezug wird durch Assoziationen und zusätzliche Syntaxeinschränkungen definiert. Beispielsweise haben die Teile des DSD einen Querbezug zu den Akteuren des ÜD [40]: Wenn ein Teil über eine Verbindung mit dem zu entwickelnden System verbunden ist, muss der Typ des Teils ein Akteur im ÜD sein. Der Akteur muss in der Architektur berücksichtigt werden. Der Einsatz dieser Konsistenzbedingung wird am schematischen Beispiel gezeigt, indem ein neues Teil vom Typ *Datenbank* mit dem zu entwickelnden System im DSD verbunden wird (siehe Abb. 3.3). Die Diagramme sind

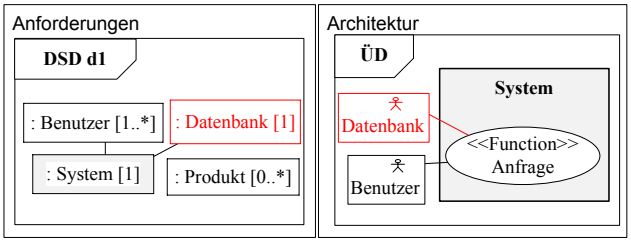


Abbildung 3.3: Konsistenzbedingungen am Beispiel [40]

in diesem Fall so lange inkonsistent bis dem ÜD ein entsprechender Akteur hinzugefügt wird. Umgekehrt muss bei Einführung eines neuen Akteurs *Datenbank* zur Wiederherstellung der Konsistenz ein Teil im DSD hinzugefügt werden. Dieses muss einen Typ haben, der einen Querbezug zu dem neuen Akteur hat. In der Arbeit in [40] wurden die wichtigsten Querbezüge beschrieben. In den folgenden Abschnitten werden die für diese Arbeit relevanten Diagramme und Querbezüge im Detail vorgestellt.

3.2 Hierarchische Anforderungsliste (HAL)

Hauptgegenstand der Anforderungsbeschreibung in CREATE sind die Szenarien. In vielen Fällen werden Anforderungen jedoch initial in Textform gestellt. Dies kann im Rahmen von Protokollen oder Anforderungsdokumenten erfolgen. Die hierarchische Anforderungsliste (HAL) erlaubt die strukturierte Verwaltung textbasierter Anforderungen und ermöglicht damit auch deren Rückverfolgung.

Für die Unterscheidung verschiedener Anforderungstypen besteht die Möglichkeit mehrere Anforderungslisten zu beschreiben. Beispielsweise kann jeweils eine Anforderungsliste für funktionale und nicht-funktionale Anforderungen angelegt werden (siehe Abb. 3.4). Eine HAL enthält eine Sequenz von Anforderungen. Eine Anforde-

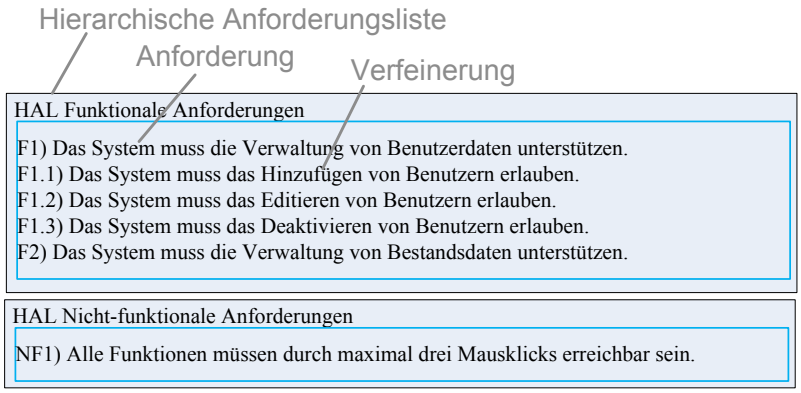


Abbildung 3.4: HAL Struktur

rung kann durch mehrere weitere Anforderungen verfeinert werden. Die Anforderung *F1* wird beispielsweise durch die Anforderung *F1.1* verfeinert (siehe Abb. 3.4). Anforderungen können im Bedarfsfall weitere Attribute hinzugefügt werden wie beispielsweise Prioritäten oder eine Kritikalität.

3.3 Domänenstrukturdiagramm (DSD)

Die Struktur der Domäne (z.B. die Geschäftsstruktur) legt wichtige Anforderungen an das zu entwickelnde System fest. Die Domänenstruktur wird in CREATE durch das Domänenstrukturdiagramm (DSD) beschrieben. Das DSD basiert auf dem UML Kompositionsstrukturdiagramm [58]. Eine Domänenstruktur besteht aus Systemen, Personen und Entitäten, welche miteinander kommunizieren können. Diese werden durch Teile beschrieben, die durch Verbindungen miteinander verknüpft sind. Teile sind Mengen von Objekten und haben eine Multiplizität. Des Weiteren sind Teile typisiert. Der interne Aufbau dieser Typen kann rekursiv durch Kompositionsstrukturdiagramme beschrieben werden. Ein Typ im DSD ist das zu entwickelnde System. Das DSD *dsd1* zeigt beispielsweise eine Domäne mit mehreren Teilen (siehe Abb. 3.5).

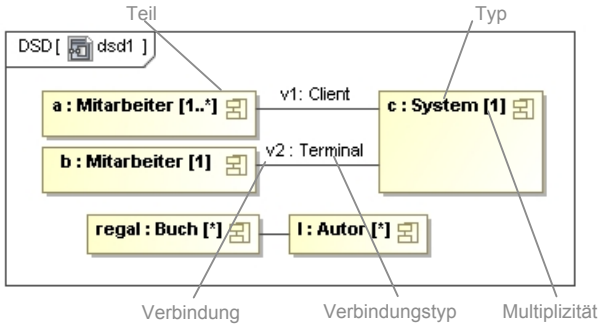


Abbildung 3.5: DSD Domänenstruktur

Die Teile *a* und *b* sind vom Typ *Mitarbeiter* und das Teil *c* vom Typ *System*. Das Teil *a* beschreibt eine Menge mit einem oder mehreren Objekten. Die Teile *b* und *c* bestehen jeweils aus genau einem Objekt. Kommunikationsmöglichkeiten zwischen den Teilen werden durch Verbindungen beschrieben. Zur Vereinfachung des Modells drücken Verbindungen eine Kommunikationsmöglichkeit zwischen allen Objekten des einen Teils und allen Objekten des anderen Teils aus. Verbindungen können sowohl uni- als auch bidirektional sein. Des Weiteren können Verbindungen typisiert sein. In *dsd1* bestehen sowohl zwischen den Teilen *a* und *c* als auch zwischen den Teilen *b* und *c* Kommunikationsmöglichkeiten über die Verbindungen *v1* und *v2*. Die Verbindung *v1* ist hierbei vom Typ *Client* und *v2* vom Typ *Terminal*. Jedes Objekt des Teils *a* kann beispielsweise mit dem System *c* kommunizieren. Die Objekte der Teile *a* und *b* können hingegen nicht miteinander kommunizieren.

Querbezüge

Die durch das DSD beschriebene Struktur sollte nicht den textbasierten Anforderungen der HAL widersprechen und die Berücksichtiung der Angaben im HAL sollte rückverfolgbar sein. Zur Rückverfolgung hat das DSD Querbezüge zur HAL. Worte aus den Anforderungsbeschreibungen können mit Typen des DSD verknüpft sein. Beispielsweise ist der Typ *Buchhaltungssystem* des DSD *dsd2* mit dem Wort Buchhaltungssystem der HAL verknüpft (siehe Abb. 3.6).

Dieser Querbezug dient der Rückverfolgung von Anforderungen und ermöglicht die schnelle Auffindung relevanter Texte des HAL. Die relevanten Texte können dann auf Konsistenz geprüft werden. Zudem kann festgestellt werden, wie die Anforderungen im DSD berücksichtigt wurden. Umgekehrt besteht die Möglichkeit festzustellen, wegen welcher Anforderung ein bestimmter Typ erstellt wurde.

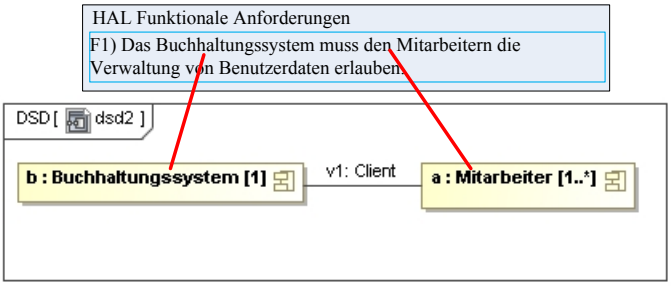


Abbildung 3.6: Querbezug DSD zu HAL

3.4 Szenariendiagramm (SD) und Interaktionsskizzendiagramm (ID)

Ein wichtiger Aspekt in der Anforderungserhebung ist die Beschreibung von Prozessen in der Domäne (z.B. Geschäftsprozesse), die vom zu entwickelnden System unterstützt werden sollen. Prozesse in der Domäne werden in CREATE durch das Szenariendiagramm (SD), welches auf dem UML Kommunikationsdiagramm [58] basiert, eindeutig beschrieben. Bei dem Kommunikationsdiagramm tauschen Objekte Nachrichten in einer festgelegten Reihenfolge aus. SD beschreibt hierbei repräsentative Szenarien in der Domäne, welche vom zu entwickelnden System unterstützt werden sollen. Nachrichten zwischen Kommunikationspartnern werdem im SD Interaktionen genannt. Das Interaktionsskizzendiagramm (ID) visualisiert die Interaktionen des SD mittels Interaktionsskizzen und legt zudem die Reihenfolge fest, in der die Interaktionen aktiv sind. Das SD *sd1* und das ID *id1* modellieren beispielsweise ein Szenario (siehe Abb. 3.7). Hierbei beschreibt *sd1* eine Kommunikation zwischen zwei Kommunikationspartnern vom Typ *A* und Typ *B*. Im Rahmen der Kommunikation *sd1* erfolgen die Interaktionen

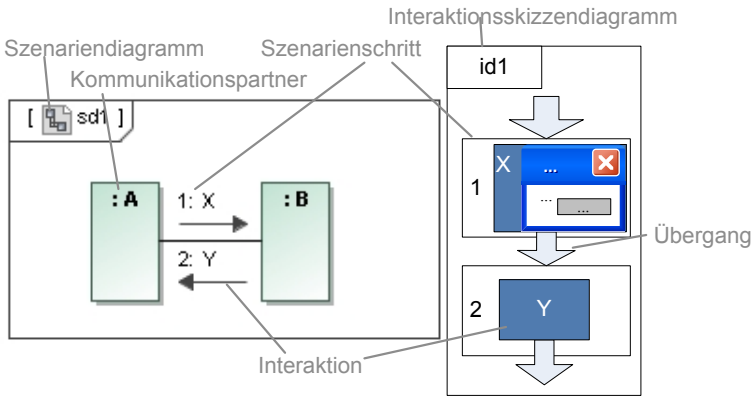


Abbildung 3.7: Szenario im SD und ID

X und *Y*. Das ID *id1* visualisiert diese Interaktionen. Die Interaktion *X* ist hierbei im Szenarienschritt *1* und *Y* im Szenarienschritt *2* aktiv. Die Übergänge legen die Reihenfolge der Szenarienschritte fest. Aus der Reihenfolge ergibt sich die Nummerierung der Szenarienschritte. Eine Interaktion des SD trägt immer die Nummer des Szenarienschritts aus dem ID. Die Interaktion *X* ist im Szenarienschritt *1* aktiv und trägt im SD damit ebenfalls die Nummer *1*. Ein Übergang kann für den Start oder das Ende eines Szenarios stehen. Des Weiteren findet ein Übergang beim Beenden einer Interaktion statt. Diese Interaktion ist der Auslöser des Übergangs. Für die aus den Szenarienschritten *1* und *2* ausgehenden Übergänge sind beispielsweise die Interaktionen *X* und *Y* die Auslöser (siehe Abb. 3.7).

Parallele Abläufe werden modelliert, indem in einem Szenarienschritt mehrere Interaktionen aktiv sind. Zudem können Interaktionen in mehreren aufeinander folgenden Szenarienzuständen aktiv sein. Folglich kann eine Interaktion im SD eine Menge von Nummern tragen und eine Nummer kann von mehreren Interaktionen getragen werden. An dieser Stelle unterscheidet sich die Nummerierung von der eines UML-Kommunikationsdiagramms. Das SD *sd2* und das ID *id2* modellieren beispielsweise ein Szenario mit einem parallelen Ablauf (siehe Abb. 3.8). Die Interaktion *X* aus *sd2* ist

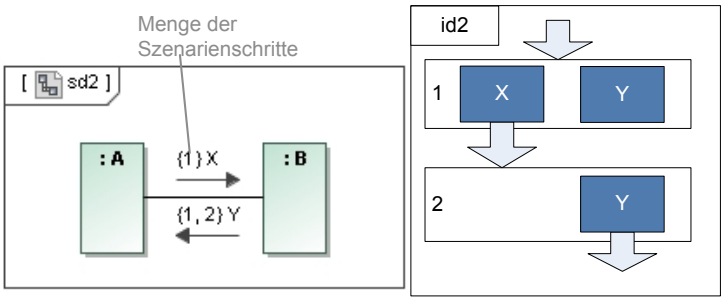


Abbildung 3.8: Szenario im SD und ID mit parallelem Ablauf

im Szenarienschritt *1* aktiv und trägt damit die Nummer *1*. Die Interaktion *Y* ist in den Szenarienschritten *1* und *2* aktiv und trägt damit die Nummern *1* und *2*. Im Szenarienschritt *1* sind beide Interaktionen aktiv. Damit tragen beide Interaktionen die Nummer *1*. Durch diese Art der Nummerierung wird im SD ein eindeutiges Zeitinterleaving der aktiven Interaktionen festgelegt.

Mit Hilfe des SD und ID besteht des Weiteren die Möglichkeit alternative Abläufe zu modellieren. Hiermit können mehrere Szenarien mit einem SD und ID modelliert werden. Eine mehrfache Modellierung gleicher Szenarienschritte ist damit nicht erforderlich. Das SD *sd3* und das ID *id3* beschreiben beispielsweise ein Szenario mit einem alternativen Ablauf (siehe Abb. 3.9). Nach Szenarienschritt *1* kann entweder der Szenarienschritt *2a* oder *2b* folgen. In *2a* ist die Interaktion *X* aktiv und in *2b* die Interaktion *Y*. Die Interaktion *X* ist damit in den Szenarienschritten *1* und *2a* aktiv. Die Interaktion *Y* ist hingegen in den Szenarienschritten *1* und *2b* aktiv. Im SD *sd3* tragen die Interaktionen *X* und *Y* damit die Nummern *1* und *2a* bzw. *1* und *2b*.

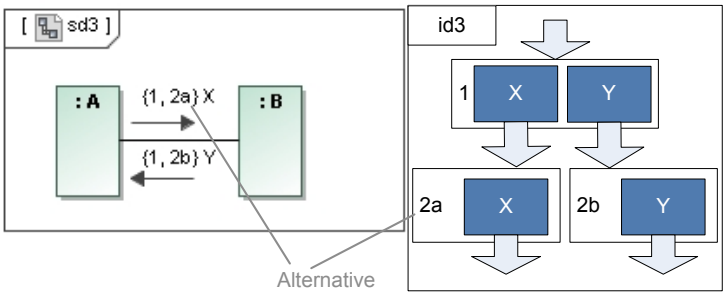


Abbildung 3.9: Szenario mit alternativem Ablauf

Querbezüge

Das SD beschreibt Prozesse in der vom DSD modellierten Domäne. Das SD hat folglich Querbezüge zum DSD. Die Teile der Domäne sind Mengen von Objekten. Ein Kommunikationspartner des SD ist ein Element dieser Objektmenge. Beispielsweise ist das Objekt *m1* ein Element des Teils *a* vom Typ *Mitarbeiter* aus *dsd1* (siehe Abb. 3.10).

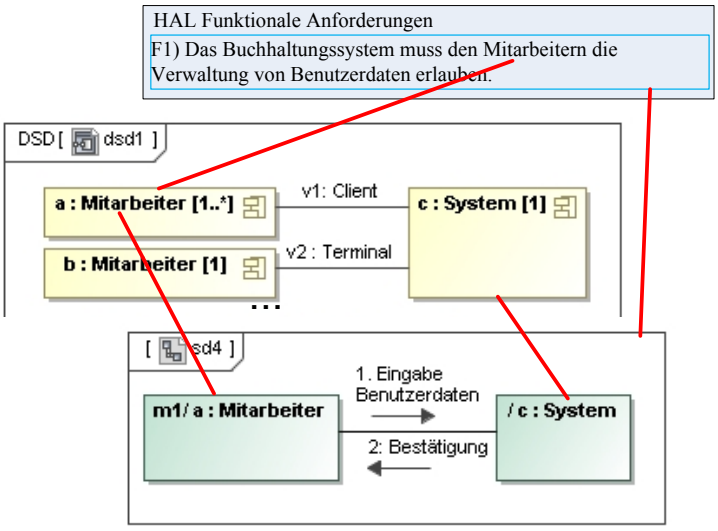


Abbildung 3.10: Querbezug DSD und HAL zu SD

Jeder Kommunikationspartner eines SD ist demnach im DSD definiert. Die Anzahl der Kommunikationspartner innerhalb einer Kommunikation muss hierbei die obere Grenze der Multiplizität der Teile im DSD einhalten. Beispielsweise hat das Teil *c* vom Typ *System* die Multiplizität *1*. Das Szenario *sd4* ist zulässig, da nur ein Kommunikationspartner aus *c* an der Kommunikation teilnimmt (siehe Abb. 3.10). Ein SD ist

hingegen nicht zulässig, wenn zwei Kommunikationspartner aus dem Teil *c* vom Typ *System* vorkommen. Eine Interaktion zwischen Kommunikationspartnern darf zudem nur im SD beschrieben werden, wenn auch eine Kommunikationsmöglichkeit zwischen den entsprechenden Teilen im DSD existiert. Die Kommunikation zwischen den Teilen *a* und *c* ist beispielsweise zulässig. Eine Interaktion zwischen einem Kommunikationspartner aus *a* und einem Kommunikationspartner aus *b* ist hingegen nicht zulässig.

Die durch das SD und ID beschriebenen Szenarien müssen zu den textbasierten Anforderungen der HAL passen. Jede Anforderung der HAL, die nicht weiter verfeinert wird, kann mit einem Szenario aus dem SD verknüpft sein. Das SD *sd4* ist beispielsweise mit der Anforderung *F1* verknüpft (siehe Abb. 3.10). Dieser Querbezug dient der Rückverfolgung von Anforderungen und ermöglicht die schnelle Auffindung relevanter Texte des HAL. Die Texte und Szenarien können anschließend auf Konsistenz geprüft werden. Zudem kann festgestellt werden, wie die Anforderungen in den Szenarien berücksichtigt wurden. Umgekehrt besteht die Möglichkeit festzustellen, wegen welcher textbasierten Anforderung eine bestimmtes Szenario beschrieben wurde.

3.5 Datendiagramm (DD)

Beim Architekturentwurf werden unter anderem die vom System zu verarbeitenden Datenobjekte spezifiziert. Diese Datenobjekte werden durch das DD beschrieben. Das DD basiert auf UML Klassendiagrammen [58], bei denen Datentypen in Form von Klassen modelliert werden. Klassen haben Attribute und Operatoren. Klassen können des Weiteren untereinander in Beziehung stehen. Das Architekturmodell des hier beschriebenen modellbasierten Ansatzes verwendet zur besseren Zugriffssteuerung auf Datenobjekte eine Kopiersemantik. Bei dieser Semantik werden Datenobjekte bei Parameterübergaben kopiert. Zyklische Abhängigkeiten führen beim Kopieren von Objektgeflechten zu einer erhöhten Komplexität. Im Gegensatz zu UML-Klassendiagrammen verwendet das DD daher lediglich Sequenzen und Generalisierungen zur Beschreibung von Beziehungen zwischen Datenobjekten. Eine Sequenz ist eine Kompositionsbeziehung, die keine zyklischen Abhängigkeiten und keine mehrfachen Zugehörigkeiten von Objekten zulässt. Ein einfaches Attribut eines Datenobjekts hat daher stets einen primitiven Typ. Operatoren sind zur Beschreibung der Eigenschaften der Datenobjekte nicht erforderlich und werden im DD nicht beschrieben. Das DD *dd1* beschreibt beispielsweise ein Modell mit drei Datenobjekten (siehe Abb. 3.11). Das Modell *dd1* beschreibt die Datenobjekte *A*, *B* und *C*. Das Datenobjekt *A* hat das Attribut *titel* vom Typ einer Zeichenkette. Das Datenobjekt *A* generalisiert das Datenobjekt *B*, welches ein zusätzliches Attribut *text* vom Typ einer Zeichenkette hat. *A* verweist überdies auf eine Sequenz von Objekten vom Typ *C*. Ein Datenobjekt vom Typ *C* hat ein Attribut *anzahl* vom Typ einer ganzen Zahl.

Durch die Einschränkung der Beziehungen zwischen Datenobjekten auf Sequenzen, kann beispielsweise das Datenobjekt *C* nicht auf eine Sequenz von Objekten vom Typ *A* verweisen. Zyklische Abhängigkeiten zwischen Datenobjekten erlauben in einigen Fällen eine schnelleres Suchen in Objektgeflechten. Mehrfache Zugehörigkeiten verringern die Anzahl der erforderlichen Kopien. Diese Nachteile werden zugunsten einer geringeren Komplexität bei Anwendung der Kopiersemantik toleriert.

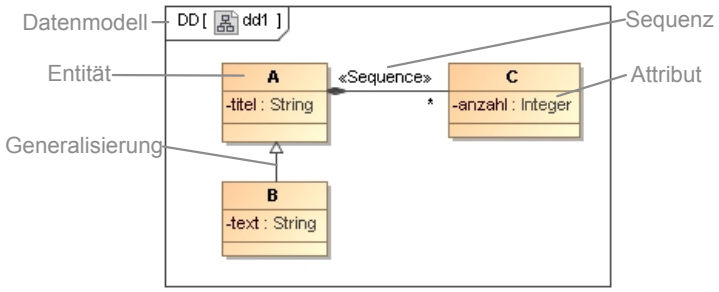


Abbildung 3.11: DD Datenobjekte und deren Beziehungen

Querbezüge

Die Datenobjekte des DD werden aus der Struktur der Domäne abgeleitet. Das DD hat demnach einen Querbezug zu DSD. Das Datenobjekt *Autor* aus *dd2* ist beispielsweise vom Typ *Autor* aus *dsd1* abgeleitet. Das Datenobjekt und der Typ *Autor* haben damit einen Querbezug (siehe Abb. 3.12). Analog haben auch das Datenobjekt und der

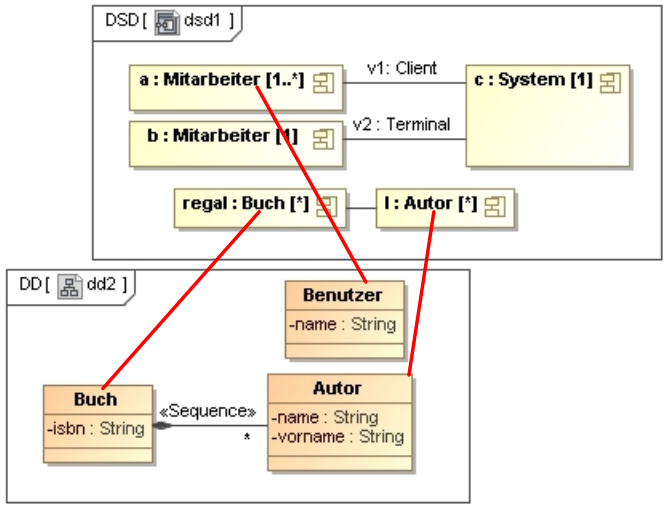


Abbildung 3.12: Querbezug DSD zu DD

Typ *Buch* sowie das Datenobjekt *Benutzer* und der Typ *Mitarbeiter* einen Querbezug. Zudem müssen auch die Beziehungen zwischen den Datenobjekten den Verbindungen der Teile in der Domänenstruktur entsprechen. Die Teile *regal* vom Typ *Buch* und *l* vom Typ *Autor* aus *dsd1* sind miteinander verbunden (siehe Abb. 3.12). Folglich muss es auch eine Beziehung zwischen den Datenobjekten *Buch* und Autor im *dd2* geben.

3.6 Systemüberblicksdiagramm (ÜD)

Ein erster wichtiger Schritt des Architekturentwurfs ist die Bestimmung der Systemgrenze sowie die Festlegung der grundlegenden Funktionen des Systems. Das Systemüberblicksdiagramm (ÜD) erlaubt die Beschreibung der Systemgrenze sowie der externen Systeme und Benutzer in Form von Akteuren und basiert auf UML Anwendungsfalldiagrammen [58]. Die grundlegenden Funktionen des Systems werden in Form von Anwendungsfällen beschrieben. Das ÜD beschreibt hierbei die Akteure, die direkt an den Funktionen des Systems beteiligt sind. Das ÜD *üd1* beschreibt beispielsweise die Systemgrenze und die grundlegenden Funktionen eines Systems (siehe Abb. 3.13).

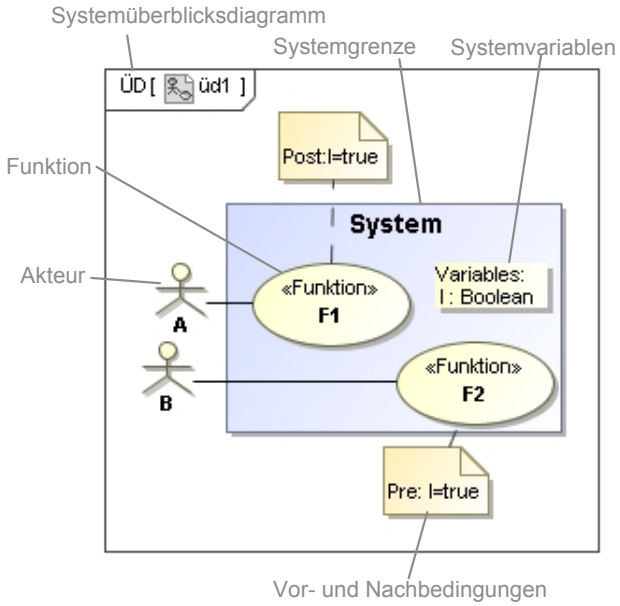


Abbildung 3.13: ÜD Systemkontext und Funktionen

Das System in *üd1* hat die Akteure *A* und *B* und die Funktionen *F1* und *F2*. Der Akteur *A* ist an der Durchführung der Funktion *F1* beteiligt. Der Akteur *B* ist an der Durchführung der Funktion *F2* beteiligt.

Funktionen können zueinander in Beziehung stehen. Anwendungsfalldiagramme der UML stellen hierfür Beziehungen wie beispielsweise *includes* und *extends* zur Verfügung. In einigen Fällen sind diese jedoch nicht zur Beschreibung der Beziehung zwischen Funktionen von Systemen geeignet. Beispielsweise soll sich ein Akteur vor der Durchführung einer Funktion anmelden. Eine *includes*-Beziehung würde ausdrücken, dass sich der Akteur bei jeder Durchführung der Funktion anmelden muss. Dies ist nicht zutreffend, wenn nur eine einmalige Anmeldung erforderlich ist. Bei Anwendung einer *extends*-Beziehung mit *extension points* wird die Anmeldung unter bestimmten Bedingungen durchgeführt. Es kann aber nicht ausgedrückt werden, dass die vorheri-

ge und erfolgreiche Anmeldung zwingend erforderlich ist. Das ÜD verwendet daher zur Beschreibung von Beziehungen zwischen Funktionen Vor- und Nachbedingungen in Kombination mit Systemvariablen. Diese werden beispielsweise durch die *Object Constraint Language* (OCL) [69] beschrieben. Das System des ÜD *üd1* hat beispielsweise die Variable *l*, welche ein Wahrheitswert beinhaltet. Die Funktion *F1* hat eine Nachbedingung. Diese legt fest, dass *l* nach Durchführung der Funktion der Wert *true* zugewiesen ist. Die Funktion *F2* hat eine Vorbedingung. Durch diese Bedingung wird die Funktion nur durchgeführt, wenn der Variable *l* der Wert *true* zugewiesen ist.

Querbezüge

Die Systemgrenze und die Akteure müssen Teil der Domänenstruktur sein. Das ÜD hat daher Querbezüge zum DSD. Ein System kann innerhalb einer Domänenstruktur mehrfach verwendet werden. Die Systemgrenze und die Akteure des ÜD entsprechen damit den Typen des DSD. Die Akteure und das System müssen damit mit einem Typ des DSD in Beziehung stehen. Der Akteur *Mitarbeiter* aus *üd2* hat beispielsweise einen Querbezug zum Typ *Mitarbeiter* aus *dsd1* (siehe Abb. 3.14). Analog stehen das System aus *üd2* und der Typ *System* aus *dsd1* miteinander in Beziehung. Die Funktionen des

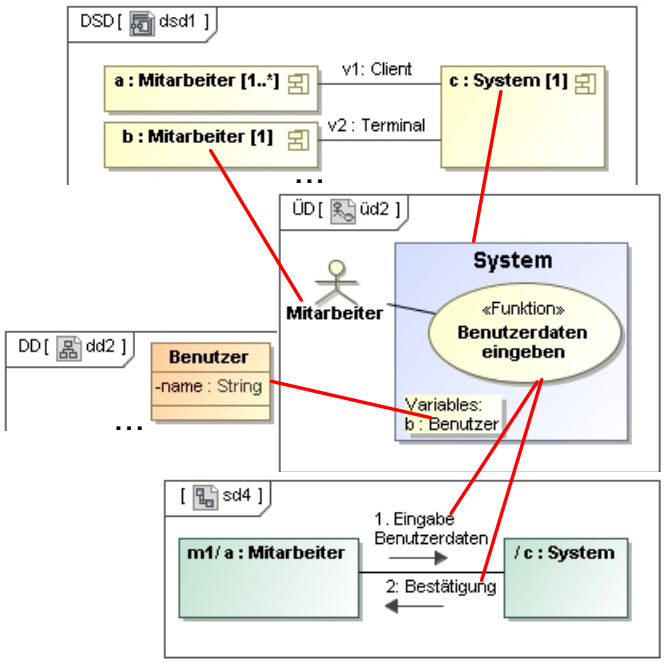


Abbildung 3.14: ÜD Querbezüge zu DSD und SD

ÜD werden aus SD abgeleitet. Die Funktionen des ÜD haben damit einen Querbezug zu den Interaktionen eines SD. Die Funktion *Benutzerdaten eingeben* des ÜD *üd2* wird

beispielsweise aus den Interaktionen *Eingabe der Benutzerdaten* und *Bestätigung* des SD *sd4* abgeleitet (siehe Abb. 3.14). Diese Funktion und die Interaktionen haben damit einen Querbezug. Die Vor- und Nachbedingungen der Funktionen des ÜD können auch über Variablen eines komplexen Datentyps ausgedrückt werden. Diese Datentypen werden im DD in Form von Datenobjekten beschrieben. Das ÜD hat damit auch einen Querbezug zum DD. Die Variable *b* des Systems aus *üd2* ist beispielsweise vom Typ *Benutzer* (siehe Abb. 3.14). Der Datentyp *Benutzer* ist in *dd2* beschrieben.

3.7 Architektonisches Verhaltensdiagramm (AVD)

Das Verhalten des Softwaresystems wird durch das architektonische Verhaltensdiagramm (AVD) beschrieben, welches auf UML Aktivitätsdiagrammen [58] inklusive Datenfluss basiert. Das AVD beschreibt den Prozess der Funktionen vollständig, die im ÜD definiert werden. Ein Aktivitätsdiagramm beschreibt einen Ablauf in Form einer Aktivität, in der Aktionen erfolgen. Der Ablauf kann über Kontrollflussknoten gesteuert werden. Kontrollflussknoten erlauben unter anderem die Beschreibung paralleler und alternativer Abläufe (siehe Kapitel 2.2.4). Ein UML Aktivitätsdiagramm stellt zudem Aufrufaktionen zur Verfügung, die innerhalb eines Ablaufs die Durchführung einer anderen Aktivität beschreiben. Dies ermöglicht eine Strukturierung umfangreicher Ablaufbeschreibungen durch Zusammenfassung zusammengehöriger Aktionen. Des Weiteren erlaubt dies die Wiederverwendung von Abläufen in anderen Beschreibungen.

Innerhalb eines AVD werden verschiedene Aktionstypen verwendet. Das AVD *avd1* beschreibt beispielsweise eine Funktion, bei der die Aktionen *Eingabe* vom Typ *InterfaceAction* und *Verarbeitung* vom Typ *ServiceAction* nacheinander durchgeführt werden (siehe Abb. 3.15). Die Bedeutung dieser Aktionstypen ist wie folgt definiert: Bei

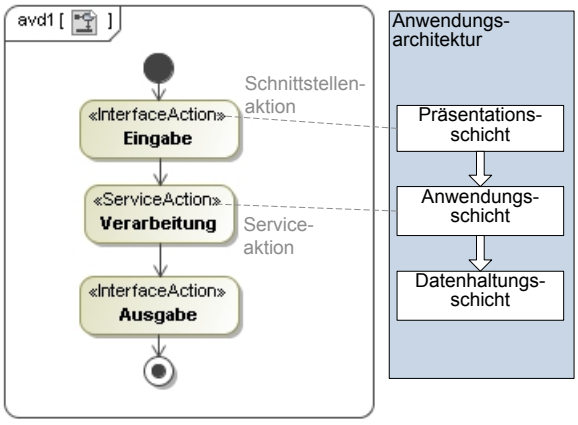


Abbildung 3.15: Aktionstypen des AVD

Durchführung einer Aktion vom Typ *InterfaceAction* interagiert das System mit den umliegenden Systemen oder Benutzern. Bei Durchführung einer Aktion vom Typ *ServiceAction* erfolgt ein systeminterner Verarbeitungsschritt. Eine Aktion vom Typ *InterfaceAction* wird im Folgenden als Schnittstellenaktion und eine Aktion vom Typ *ServiceAction* als Serviceaktion bezeichnet. Aktionen werden durch Komponenten der komponentenbasierten Architektur realisiert. Diese Komponenten können in Schichten eingeteilt werden. Mit der Festlegung der Schichtenarchitektur, können die zur Verfügung stehenden Aktionstypen definiert werden. Zu jeder Schicht der Architektur kann in diesem Fall ein Aktionstyp gehören. Der Typ *InterfaceAction* kann beispielsweise der *Präsentationsschicht* und der Typ *ServiceAction* der *Anwendungsschicht* zugeordnet werden (siehe Abb. 3.15). Kommunikationen zwischen Komponenten werden im AVD nicht beschrieben. Die Schichtenarchitektur schränkt damit nicht die Wahl der Reihenfolge der Aktionen ein.

Das AVD verwendet Variablen für die Beschreibung des Objektflusses. Aktionen können auf Variablen über Eingabe- und Ausgabepins lesend und schreibend zugreifen. Variablen können sowohl einzelne Objekte als auch Mengen von Objekten beinhalten. Das AVD *av2* beschreibt beispielsweise einen Ablauf, bei der die Variable *i* ein Objekt vom Typ *Integer* und *j* eine Menge von Integer-Objekten beinhaltet (siehe Abb. 3.16). Die Aktion *a* hat einen Ausgabepin, der auf die Variable *i* verweist.

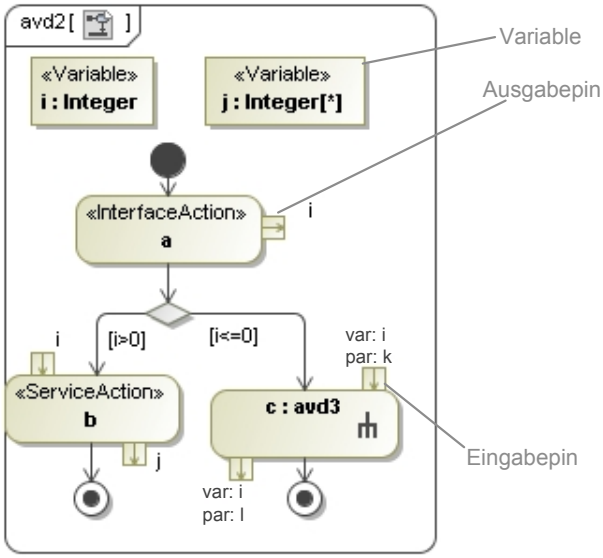


Abbildung 3.16: Variablen und Pins

Nach Durchführung wird die Variable *i* mit der Ausgabe der Aktion *a* beschrieben. Die Aktion *b* hat einen Eingabe- und einen Ausgabepin, die auf die Variable *i* resp. *j* verweisen. Bei Beginn der Durchführung wird der Wert der Variable *i* gelesen und als Eingabewert für die Aktion verwendet. Nach Durchführung der Aktion *b* wird die Variable *j* mit dem Ausgabewert beschrieben. Bei einem Aufruf eines weiteren AVDs

können Parameter übergeben werden. Eine Aktivität hat eine Menge von Variablen für Eingabe- und Ausgabeparameter (Parametervariablen). Parametervariablen werden innerhalb des zugehörigen AVDs wie andere Variablen verwendet. Aufrufaktionen haben für jeden Eingabeparameter des aufgerufenen AVDs einen Eingabepin und für jeden Ausgabeparameter einen Ausgabepin. Ein Pin einer Aufrufaktion verweist sowohl auf eine Variable des eigenen AVD als auch auf eine Parametervariable des aufgerufenen AVDs. Das AVD *avd3* hat beispielsweise eine Variable für den Eingabeparameter *k* und eine Variable für den Ausgabeparameter *l* (siehe Abb. 3.17). Die Aktion *c* des

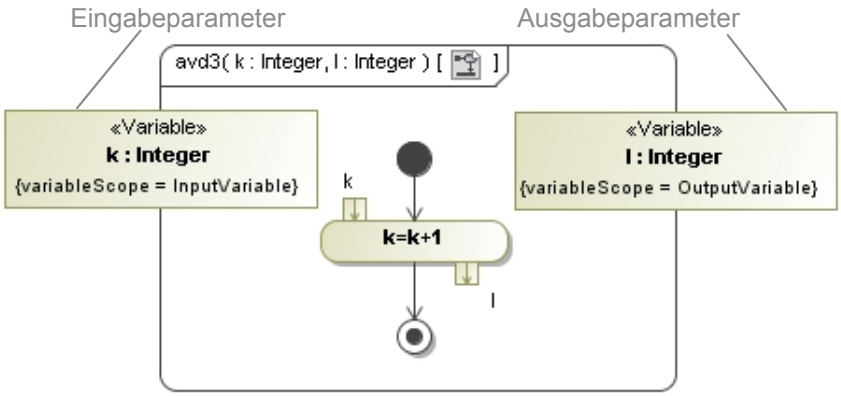


Abbildung 3.17: Parameterübergaben

AVD *avd2* ruft *avd3* auf (siehe Abb. 3.16). Der Eingabepin der Aktion *c* verweist sowohl auf die Variable *i* als auch auf die Parametervariable *k* des aufgerufenen AVDs *avd3*. Bei Durchführung einer Aufrufaktion wird eine Aufrufinstanz der aufgerufenen Aktivität erzeugt. Der Inhalt der Variablen, auf die die Eingabepins verweisen, werden auf die Variablen für Eingabeparameter kopiert. Bei Abschluss eines AVD wird auch die aufrufende Aktion beendet. Der Inhalt der Parametervariablen, auf die die Ausgabepins verweisen, werden auf die referenzierten Variablen kopiert. Beispielsweise wird bei Durchführung der Aktion *c* der Inhalt der Variable *i* auf die Parametervariable *k* kopiert. Bei Abschluss des AVD *avd3* wird der Inhalt der Parametervariable *l* auf die Variable *i* kopiert.

Querbezüge

Die Abläufe der Funktionen des ÜD werden jeweils durch ein AVD beschrieben. Zudem können die Systemvariablen des ÜD innerhalb eines AVD verwendet werden. Das AVD hat demnach Querbezüge zum ÜD. Der Ablauf des Anwendungsfalls *Benutzerdaten eingeben* wird beispielsweise durch das AVD *Benutzerdaten eingeben* beschrieben (siehe Abb. 3.18). Die Aktion *Eingabe* hat beispielsweise einen Ausgabepin, der die Systemvariable *b* referenziert. Nach Durchführung wird demnach der Wert der Variable *b* mit der Ausgabe der Aktion beschrieben. Die Variablen des AVD können einen komplexen Typ haben, der im DD beschriebenen ist. Das AVD hat folglich einen

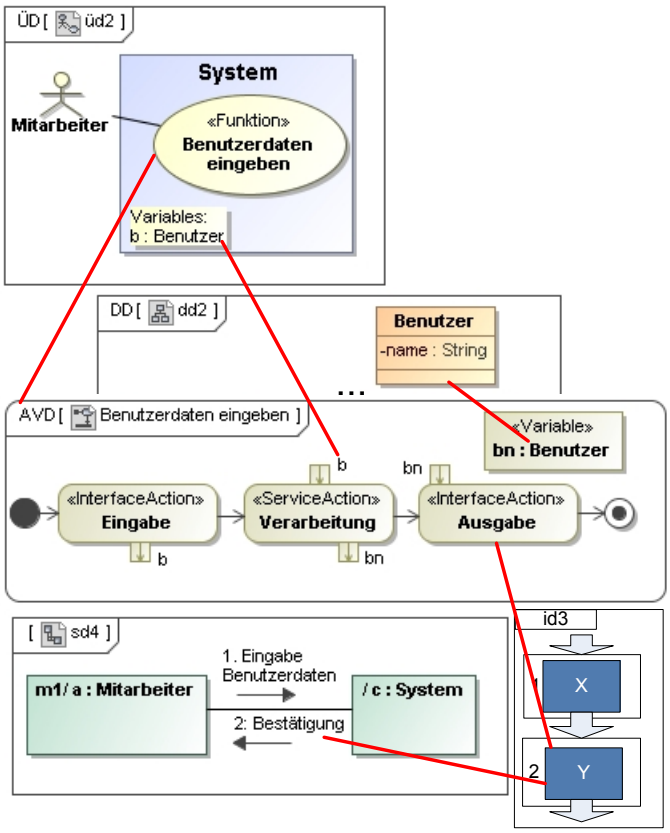


Abbildung 3.18: Querbezüge AVD zu ÜD und ID

Querbezug zum DD. Die Variable *bn* des AVD *Benutzerdaten eingeben* ist beispielsweise vom Typ *Benutzer* des DD *dd2* (siehe Abb. 3.18). Schnittstellenaktionen stellen Interaktionen des Systems mit den umliegenden Systemen und Benutzern dar. Die Visualisierung der Daten erfolgt über die Interaktionsskizzen des ID. Eine Schnittstellenaktion kann in mehreren Szenarien verwendet werden. Jede Schnittstellenaktion kann daher mit einer Menge von Interaktionsskizzen eines ID verknüpft sein. Die Aktion *Ausgabe* ist beispielsweise mit der Interaktionsskizze *Y* verknüpft, die die Interaktion *Bestätigung* des SD visualisiert (siehe Abb. 3.18).

Kapitel 4

Problemstellung im Kontext der Konsistenzsicherung

Modellbasierte Ansätze unterstützen die Konsistenzsicherung von Anforderungen und Architekturen. Der im Kapitel 3 vorgestellte modellbasierte Ansatz CREATE [40] stellt beispielsweise geeignete Struktur- und Verhaltensmodelle für die Beschreibung von Anforderungen und Architekturen zur Verfügung. Des Weiteren werden Querbezüge zwischen Anforderungs- und Architekturmodellen definiert, die die Konsistenzsicherung der Modelle unterstützen. In diesem Kapitel wird die Problematik einer Automatisierung der Konsistenzsicherung von Modellen beschrieben, die sich zur Beschreibung von Anforderungen und Architekturen eignen. Im Abschnitt 4.1 wird der vorgestellte modellbasierte Ansatz anhand eines Fallbeispiels angewendet. Die Herausforderung der Automatisierung der Konsistenzsicherung wird im Abschnitt 4.2 anhand dieses Beispiels beschrieben. Des Weiteren beinhaltet dieses Kapitel eine Abbildung der Problematik auf ein allgemeines Modell. Anhand dieses Modells werden die Anforderungen an ein Verfahren zur Automatisierung der Konsistenzsicherung erläutert. Abschnitt 4.3 beinhaltet eine Betrachtung bestehender Ansätze. Im Abschnitt 4.4 werden die Ergebnisse dieser Betrachtung zusammengefasst. Hieraus wird die resultierende Aufgabenstellung zur Lösung der Problematik der Automatisierung der Konsistenzsicherung von Anforderungen und Architekturen abgeleitet.

4.1 Fallbeispiel Bibliothekssystem

Im hier betrachteten Fallbeispiel werden die Anforderungen und die Architektur eines Bibliothekssystems mit Hilfe des in Kapitel 3 eingeführten modellbasierten Ansatzes CREATE beschrieben. Für die Beschreibung der Anforderungen wird das DSD und das SD in Verbindung mit ID verwendet (siehe Abb. 3.1). Die Szenarien des SD und ID umfassen das Verwalten, Ausleihen und Einchecken von Büchern. Für die Beschreibung der hieraus abgeleiteten Architektur wird das ÜD, DD und AVD verwendet.

4.1.1 Anforderungsmodell

DSD Bibliothek

Das DSD *Bibliothek* beschreibt die Struktur der Domäne Bibliothek (siehe Abb. 4.1). Die Domäne besteht aus einem Leiter, einer Menge von Angestellten, einer Menge von Kunden, einem Mailserver, dem Bibliothekssystem und einer Menge von Büchern mit den zugehörigen Autoren. Der Leiter, die Kunden und die Angestellten sind Personen. Das Bibliothekssystem und der Mailserver sind Systeme. Verbindungen bestehen zwischen Leiter und Angestellte, Angestellte und Bibliothekssystem, Kunden und Bibliothekssystem, Buch und Autor sowie Mailserver und Bibliothekssystem. Die Art der Kommunikationsmöglichkeit (z.B. Sprache oder Text) und die Art der Systeme (z.B. Hardware oder Software) wird an dieser Stelle offen gelassen. Der Mailserver ist in diesem Beispiel ein System außerhalb des zu entwickelnden Bibliothekssystems. Die Beziehung zwischen Kunde und Mailserver wird daher vernachlässigt.

Szenarien innerhalb der Bibliothek

Die Anforderungen an das Bibliothekssystem werden in diesem Beispiel durch die folgenden drei Szenarien beschrieben:

1. *Buch hinzufügen*
2. *Bücher ausleihen*
3. *Bücher einchecken*

Die Szenarien werden in CREATE durch das SD und ID beschrieben.

Szenario 1 Beim Szenario *Buch hinzufügen* interagiert ein Angestellter mit dem Bibliothekssystem (siehe Abb. 4.1). Das Szenario besteht aus den folgenden zwei Schritten:

1. Der Angestellte wählt das Schaltelement „Buch hinzufügen“ auf einer Interaktionsskizze aus. Die Interaktionsskizze zeigt hierbei eine Auflistung der vorhandenen Bücher und die angebotenen Schaltelemente. Als Schaltelement steht in dieser initialen Version der Anforderungen neben dem hinzufügen eines Buchs lediglich das Schaltelement „Beenden“ zur Verfügung.
2. Der Angestellte gibt die Daten des neuen Buchs ein. Die Interaktionsskizze zeigt hierbei die Möglichkeit zur Eingabe des Namens, der ISBN-Nummer und der Autoren. Als Schaltelement steht hier neben der Bestätigung der Daten das *Abbrechen* und das Verwalten von Autoren zur Verügung. Der Angestellte bestätigt die Daten.

Nach der Eingabe und Bestätigung der Daten ist das Szenario aus Sicht der Domänenstruktur beendet. Eine Rückmeldung vom System nach der Eingabe der Daten wird in dieser initialen Version der Anforderungen nicht gefordert.

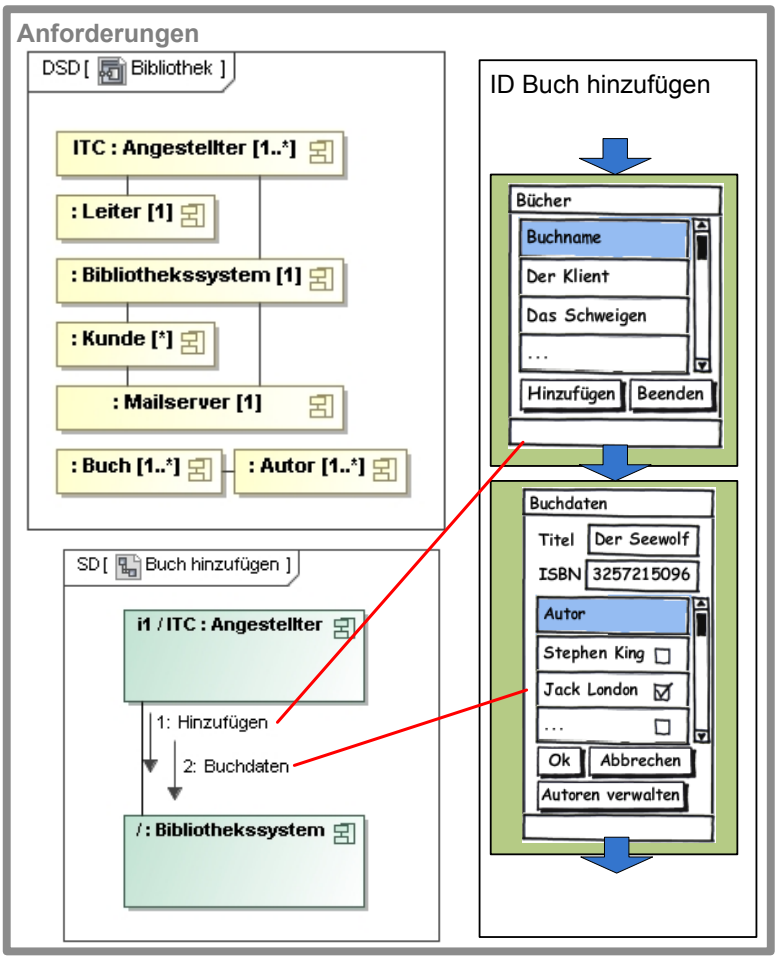


Abbildung 4.1: Buch hinzufügen

Szenario 2 Das Szenario *Bücher ausleihen* beschreibt eine Kommunikation zwischen einem Kunden, dem Bibliothekssystem und dem Mailserver (siehe Abb. 4.2). Das Szenario besteht aus den folgenden zwei Schritten:

1. Der Kunde wählt auf der Interaktionsskizze *Bücher auswählen* eine Menge von Büchern. Hierfür zeigt die Interaktionsskizze die zur Verfügung stehenden Bücher an. Des Weiteren besteht die Möglichkeit eine Bestätigung per Email zu selektieren. In diesem Szenario wird dies nicht gewünscht.

2. Der Kunde erhält eine Auflistung der aktuell ausgeliehenen Bücher. Der Kunde bestätigt die angezeigte Auflistung.

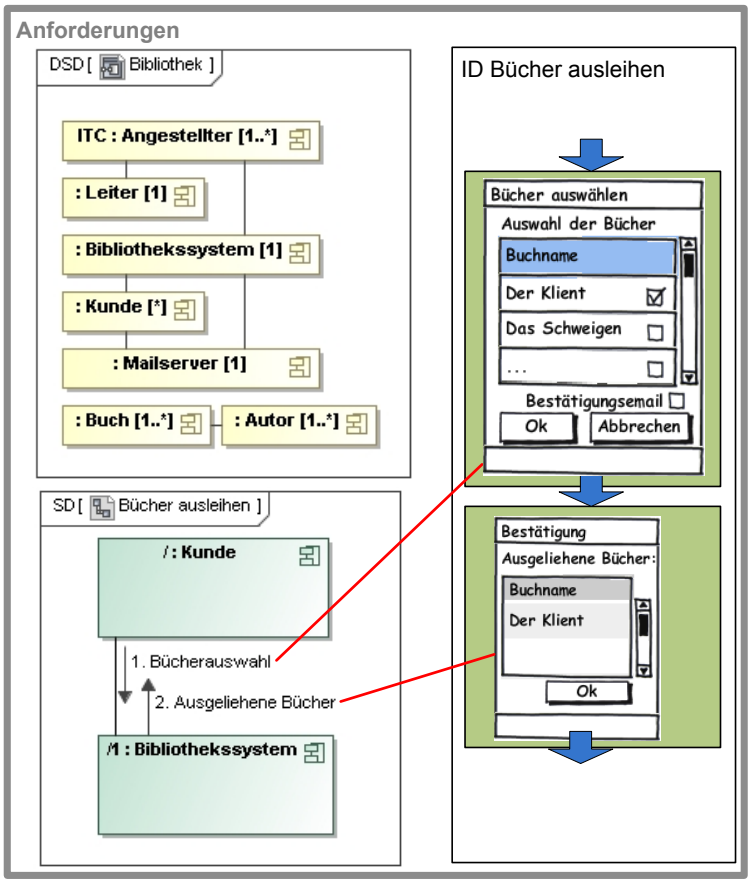


Abbildung 4.2: Bücher ausleihen

Szenario 3 Beim Szenario *Bücher einchecken* interagieren ein Kunde, das Bibliothekssystem und der Mailserver miteinander (siehe Abb. 4.3). Das Szenario besteht aus den folgenden drei Schritten:

1. Der Kunde wählt die Bücher, die eingecheckt werden sollen. Die Interaktionsskizze *Bücher einchecken* zeigt hierfür eine Liste der ausgeliehenen Bücher. Eine Emailbestätigung wird nicht gewünscht. Die Auswahl wird in diesem Szenario mit *OK* bestätigt.

2. Dem Kunden wird eine Auflistung der eingecheckten Bücher angezeigt. Diese Auflistung wird vom Kunden bestätigt.

3. Kunden können ausgeliehene Bücher vormerken. Wird ein vorgemerktes Buch eingecheckt, wird der Kunde per Email benachrichtigt. In diesem Szenario ist das Buch vorgemerkt. Hierfür sendet das Bibliothekssystem die Benachrichtigung als Email an den Mailserver.

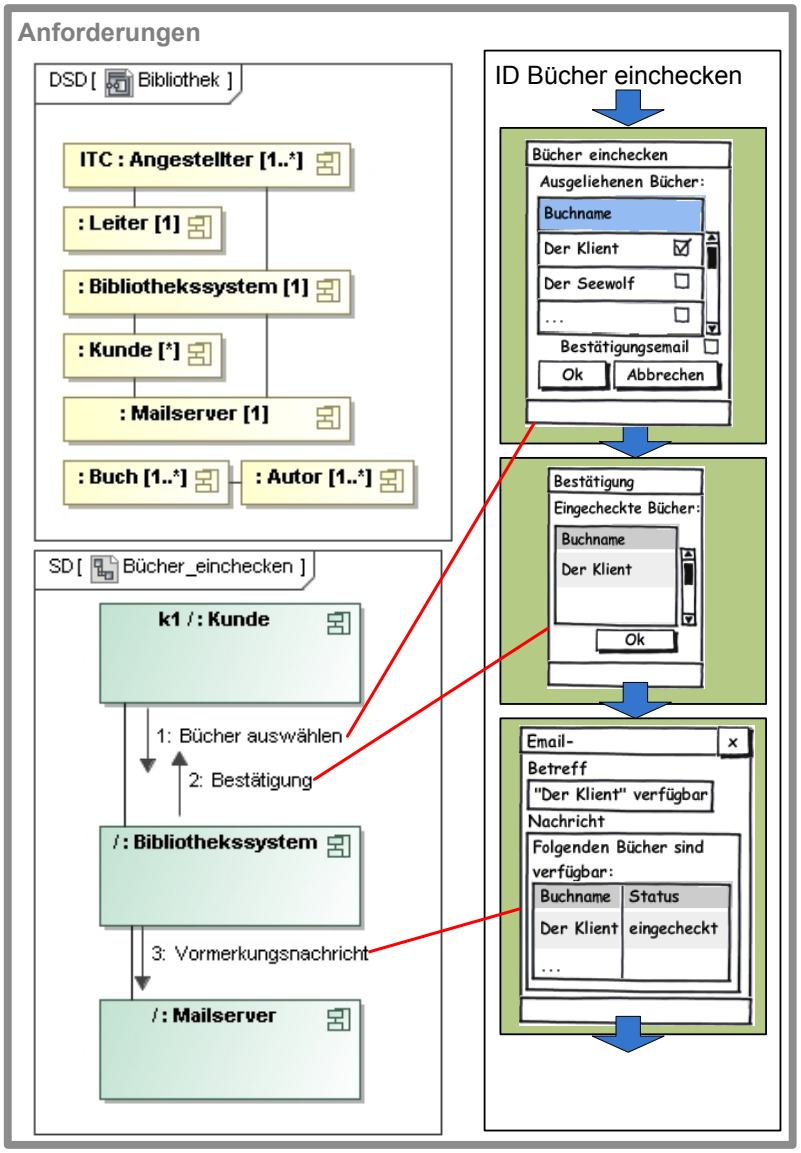


Abbildung 4.3: Bücher einchecken

4.1.2 Architekturmodell

Überblick über das Bibliothekssystem

Die Systemgrenze und die Akteure des ÜD der Architektur werden aus dem DSD abgeleitet und haben einen Querbezug zu den Typen des DSD. Im Bibliothekssystemmodell haben beispielsweise der Akteur *Angestellter* und der Typ *Angestellter* einen Querbezug (siehe Abb. 4.4). Analog stehen die Systemgrenze und der Typ *Bibliothekssystem* miteinander in Beziehung.

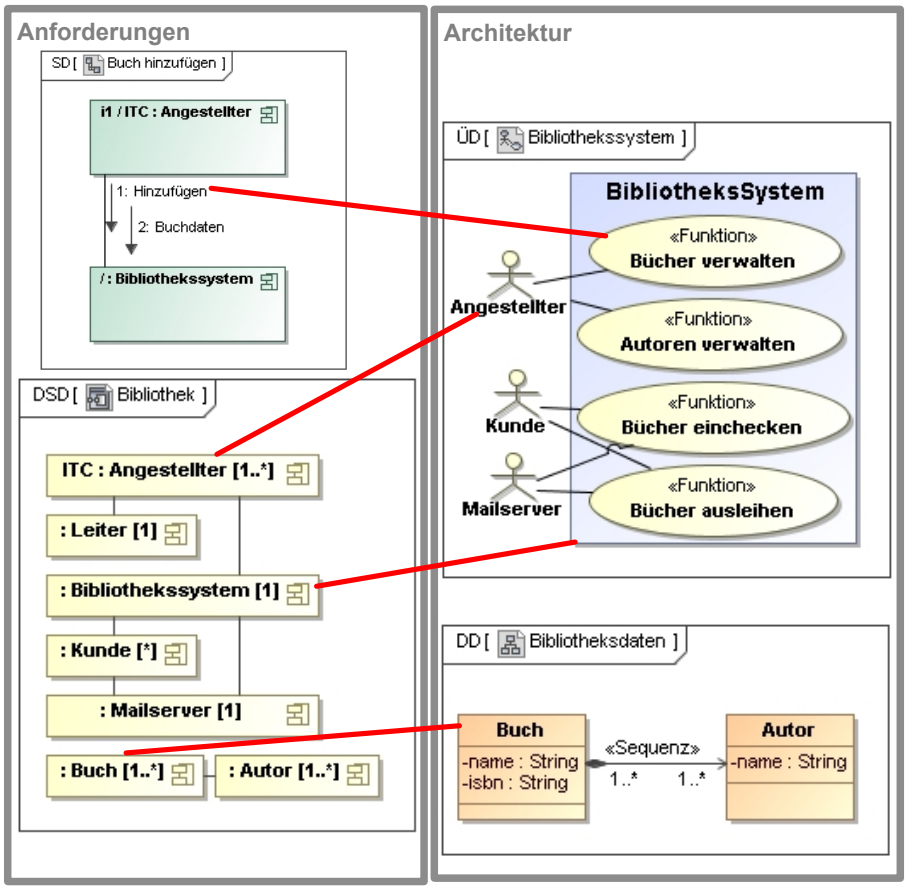


Abbildung 4.4: Strukturmodelle der Architektur

Aus den Szenarien der Anforderungen werden die Funktionen des Systems abgeleitet. Im Fallbeispiel sind dies die Szenarien *Buch hinzufügen*, *Bücher ausleihen* und *Bücher einchecken*. Die Funktion *Bücher verwalten* wird beispielsweise aus der Interaktion *Hinzufügen* des SD *Buch hinzufügen* abgeleitet (siehe Abb. 4.4). Diese Funktion und die Interaktion haben daher einen Querbezug. Aus einem Szenarios können dabei mehrere Funktionen des Systems abgeleitet werden. Aus dem Szenario *Buch hinzufü-*

gen des Fallbeispiels werden die Funktionen *Bücher verwalten* und *Autoren verwalten* abgeleitet. Aus den Szenarien *Bücher ausleihen* und *Bücher einchecken* werden die Funktionen *Bücher ausleihen* bzw. *Bücher einchecken* abgeleitet.

Datenobjekte des Bibliothekssystems

Die vom System verarbeiteten Datenobjekte werden aus dem DSD abgeleitet. Die Datenobjekte haben damit einen Querbezug zu den Typen des DSD. Im Bibliothekssystemmodell stehen beispielsweise der Typ *Buch* mit dem Datenobjekt *Buch* miteinander in Beziehung (siehe Abb. 4.4). Analog haben auch der Typ und das Datenobjekt *Autor* einen Querbezug.

Buch und *Autor* haben das Attribut *name* vom Typ *String*. *Buch* hat zusätzlich das Attribut *isbn* und beinhaltet einen Sequenz von Autoren.

AVD

Das AVD beschreibt die vollständigen Abläufe der Funktionen des Bibliothekssystems, die in den Szenarien der Anforderungen verwendet werden. Das Fallbeispiel umfasst die architektonischen Verhaltensbeschreibungen der folgenden Funktionen:

1. Bücher verwalten
2. Bücher ausleihen

Die Schnittstellenaktionen des AVD stehen mit den Interaktionsskizzen der Szenarien in Beziehung. Für die Darstellung dieser Querbezüge werden einem ID jeweils die relevanten AVDs gegenübergestellt (siehe Abb. 4.5).

Funktion Bücher verwalten Der Ablauf der Funktion *Bücher verwalten* wird durch die AVDs *Bücher verwalten* und *Buch hinzufügen* beschrieben (siehe Abb. 4.5).

Das AVD *Bücher verwalten* beschreibt den folgenden Ablauf:

1. Die vorhandenen Bücher werden geladen und in der Variablen *bücher* abgelegt.
2. Der Inhalt der Variable *bücher* dient als Eingabe für die Schnittstellenaktion *Zeige Verwaltungsdialog*, die mit der Interaktionsskizze *Bücher* in Beziehung steht.
3. Die Wahl zwischen *Beenden* oder *Hinzufügen* bestimmt den weiteren Ablauf. Bei Auswahl von *Beenden* wird die Funktion beendet. Bei Auswahl von *Hinzufügen* wird vor dem Beenden der Ablauf des AVD *Buch hinzufügen* durchgeführt.

Das AVD *Buch hinzufügen* beschreibt den folgenden Ablauf:

1. Die vorhandenen Autoren werden geladen und in der Variablen *autoren* abgelegt.
2. Der Inhalt dieser Variable dient als Eingabe für die Schnittstellenaktion *Buchdaten eingeben*, die mit der Interaktionsskizze *Buchdaten* in Beziehung steht. Die eingegebenen Daten werden in der Variablen *buch* abgelegt.

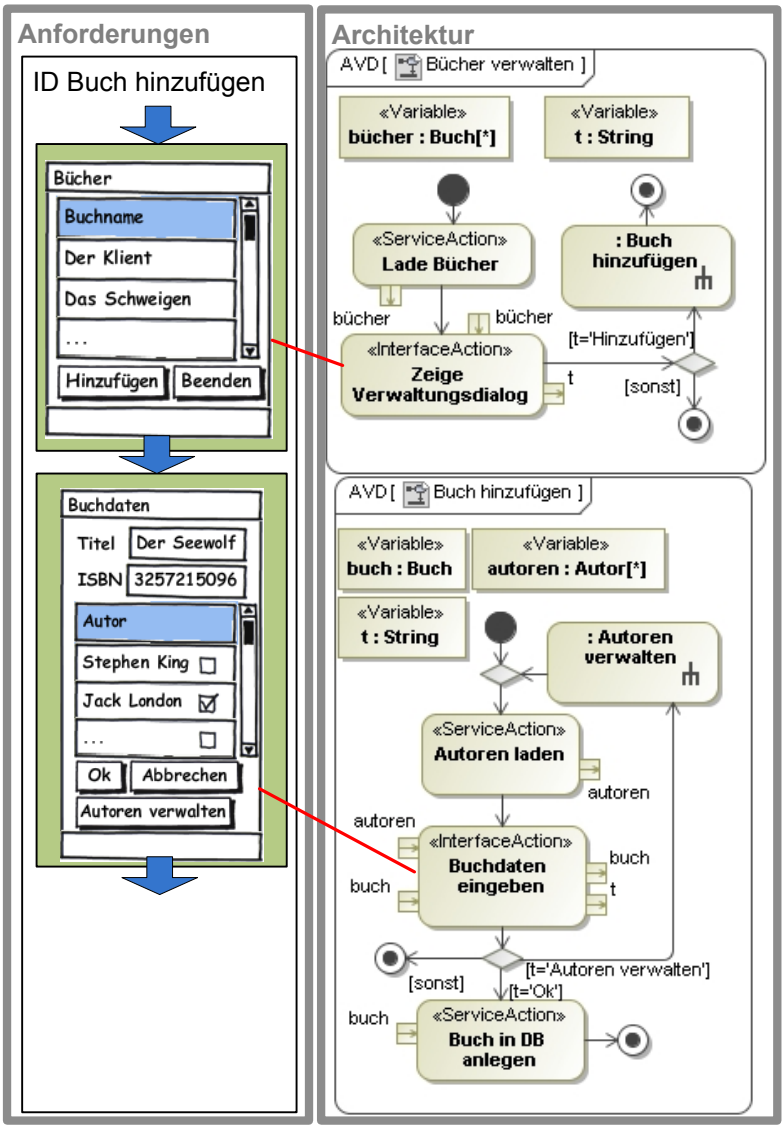


Abbildung 4.5: AVD - Bücher verwalten und Buch hinzufügen

3. Die Wahl zwischen *Abbrechen*, *Ok* oder *Autoren verwalten* bestimmt den weiteren Ablauf.

4. Bei Auswahl von *Abbrechen* ist die Funktion beendet. Die Auswahl von *Autoren verwalten* führt zur Durchführung des in AVD *Autoren verwalten* beschriebenen Ablaufs. Anschließend beginnt der Ablauf mit dem Laden der Autoren erneut.

5. Bei Auswahl von *Ok* dienen die Buchdaten der Variable *buch* als Eingabeparameter für die Serviceaktion *Buch in DB anlegen*. Nach dieser Aktion ist die Funktion beendet.

Funktion Bücher ausleihen Der Ablauf der Funktion *Bücher ausleihen* wird durch das AVD *Bücher ausleihen* beschrieben (siehe Abb. 4.6).

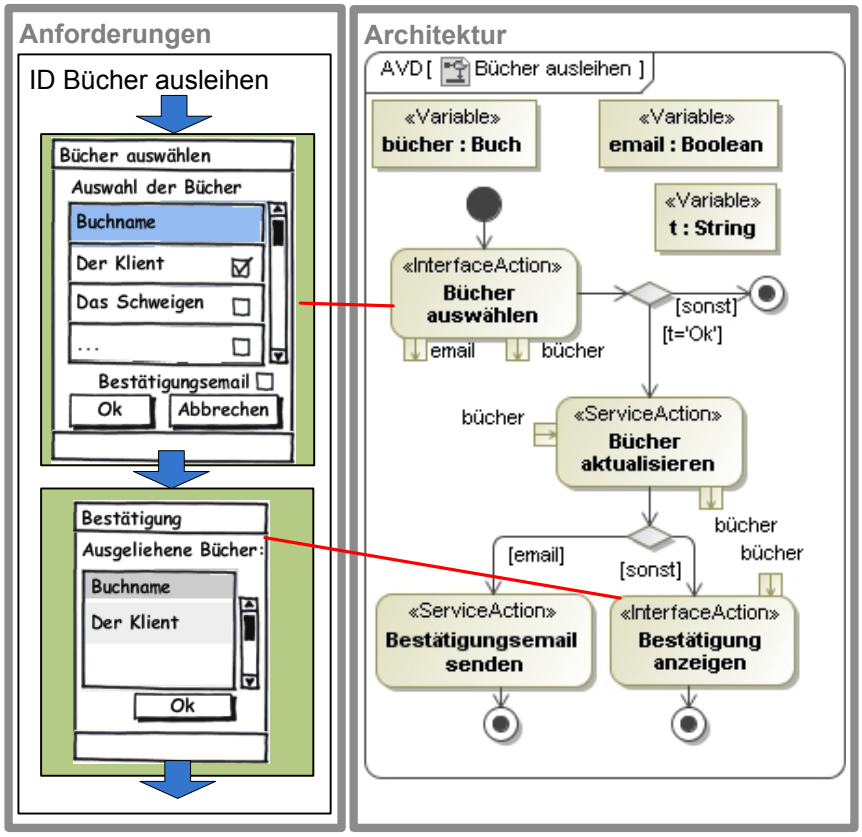


Abbildung 4.6: Aktivität - Bücher ausleihen

Die Funktion *Bücher ausleihen* beschreibt den folgenden Ablauf:

1. Die Aktion *Bücher auswählen* wird durchgeführt. Diese Aktion hat einen Querbezug zu der Interaktionsskizze *Bücher auswählen*.
2. Die selektierten Bücher und der boolsche Wert für die Auswahl der Emailbestätigung werden in den Variablen *bücher* und *email* abgelegt.
3. Die Wahl zwischen *Abbrechen* oder *Ok* bestimmt den weiteren Ablauf.
4. Bei der Wahl von *Abbrechen* wird die Funktion beendet.

5. Wird *Ok* gewählt wird die Aktion *Bücher aktualisieren* durchgeführt.

6. Abhängig von der Auswahl der Emailbestätigung wird die Aktion *Bestätigungsemail senden* oder *Bestätigung anzeigen* durchgeführt.

7. Nach Durchführung der Aktion ist die Funktion beendet.

4.2 Problembeschreibung

4.2.1 Problemstellung am Beispiel

Die Anforderungen und die Architektur entwickeln sich iterativ und evolutionär weiter. Wie im Kapitel 1 beschrieben, können hierbei leicht Inkonsistenzen entstehen. Inkonsistenzen verursachen fehlerhaft berücksichtigte und folglich nicht erfüllte Anforderungen. Daher sind Inkonsistenzen ein schwerwiegendes Problem. Sie müssen aufgedeckt und behoben werden. Inkonsistenzen können sowohl zwischen den Strukturmodellen als auch zwischen den Verhaltensmodellen der Anforderungen und Architekturen bestehen. Sei folgender Fall gegeben: Im Verlauf der Anforderungserhebung wird festgestellt, dass die Verbindung zwischen dem Teil *ITC:Angestellter* und dem Bibliothekssystem entfernt werden muss (siehe Abb. 4.4). Der Akteur *Angestellter* innerhalb der Anforderungen hat folglich keine Kommunikationsmöglichkeit mit dem Bibliothekssystem. Im ÜD ist dieser jedoch an der Durchführung einer Funktion beteiligt. Dies ist eine Inkonsistenz zwischen den Strukturmodellen der Anforderungen und der Architektur, die aufgedeckt und behoben werden muss.

Eine Inkonsistenz zwischen den Verhaltensmodellen mit Berücksichtigung der Ausführungssemantik ergibt sich beispielsweise, wenn das Szenario *Buch ausleihen* wie folgt geändert wird: Der Emailserver ist ein externes System. Die Versendung der Email kann damit über eine Interaktion des Systems mit dem Emailserver erfolgen. Die Versendung wird überdies direkt nach der Dialogbestätigung durchgeführt (vgl. Abb. 4.6 und Abb. 4.7). Die Interaktionsskizzen *Bestätigung* und *Email Bestätigung* des ID *Bücher ausleihen* werden nacheinander angezeigt. Innerhalb des AVD *Bücher ausleihen* können die in Beziehung stehenden Aktionen *Bestätigungsemail senden* und *Bestätigung anzeigen* jedoch nur alternativ durchgeführt werden (siehe Abb. 4.7). Das ID ist inkonsistent zum AVD und die Architektur erfüllt damit die Anforderungen nicht. Im Rahmen der Konsistenzsicherung müssen diese Inkonsistenzen aufgedeckt und behoben werden.

4.2.2 Konsistenz der Strukturen und des Verhaltens

Die Querbezüge zwischen den Strukturmodellen der Anforderungen und Architekturen bestehen auf der Typebene. Beispielsweise sind die Typen des DSD mit den Akteuren des ÜD und den Datenobjekten des DD verknüpft. Dieser Zusammenhang erlaubt die Definition einfacher Konsistenzbedingungen. Beispielsweise kann gefordert werden, dass ein Akteur des ÜD immer ein Teil der Domäne realisiert, das mit dem zu entwickelnden System verbunden ist. Diese einfachen Konsistenzbedingungen erlauben eine automatisierte Konsistenzüberprüfung und damit auch eine automatisierte

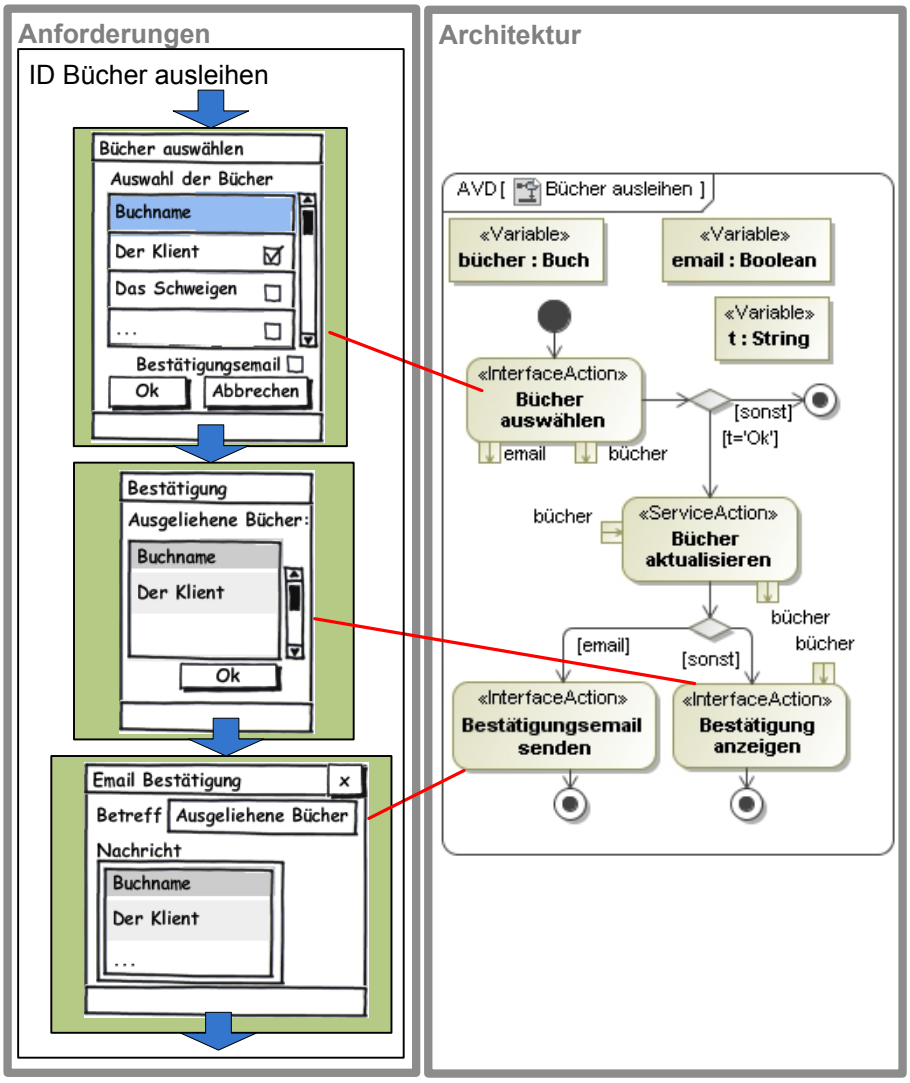


Abbildung 4.7: Inkonsistenz am Beispiel

Konsistenzsicherung [74], indem Änderungen erst akzeptiert werden, wenn alle Inkonsistenzen behoben wurden. Beispielsweise kann ein Akteur dem ÜD nur hinzugefügt werden, wenn dieser ein mit dem System verbundenes Teil der Domäne realisiert. Eine besondere Herausforderung ist die Konsistenzsicherung zwischen den Verhaltensmodellen bei Berücksichtigung der Ausführungssemantik, die im Folgenden betrachtet wird.

4.2.3 Allgemeines Modell

Szenarienbasiertes Modell

Das SD und ID beschreiben szenarienbasiert die Interaktionen zwischen Objekten. Die Typen des DSD werden hierfür instanziert. Diese Art der Beschreibung ist vergleichbar mit der eines MSC. Die Kommunikationspartner und Interaktionen des SD können durch Instanzen resp. Nachrichten des MSC beschrieben werden. Eine Interaktionsskizze des ID ist eine detaillierte Beschreibung einer Interaktion des SD und damit auch einer Nachricht des entsprechenden MSC. Das Szenario *Bücher einchecken* kann beispielsweise auch durch ein entsprechendes MSC beschrieben werden (vgl. Abb. 4.2 und Abb. 4.8).

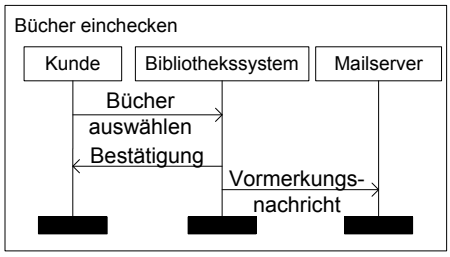


Abbildung 4.8: Szenario - Buch einchecken

Das SD ist im Zusammenhang mit dem ID folglich ein szenarienbasiertes Modell im Sinne von Liang [51].

Zustandsbasiertes Modell

Das AVD basiert auf UML 2.0 Aktivitätsdiagrammen und beschreibt das Verhalten eines Objekts, welches das System darstellt, vollständig. Die Semantik von Aktivitätsdiagrammen kann durch Petrinetze beschrieben werden (siehe [89] und [82] sowie Kapitel 2.2.4). Das AVD ist damit ein zustandsbasiertes Modell im Sinne von Liang [51]. Die Ausdrucksmächtigkeit einfacher Stellen-Transitionsnetze ist zu gering, um die Hierarchien des AVD mit Rekursion sowie Datenflüsse und Entscheidungsbedingungen abzubilden. Höhere Petrinetze wie CPNs, HCPNs und Objektenetze sowie Petrinetze mit *inhibitor arcs* haben die erforderliche Ausdrucksmächtigkeit. Die Semantik des AVD Datenflusses wird analog zu den UML 2.0 Aktivitätsdiagrammen durch CPNs beschrieben. Ein AVD hat demnach die selbe Ausdrucksmächtigkeit. Eine Beschreibung des UML 2.0 Datenflusses durch das CPN erfolgt in [88] (siehe Kapitel 2.2.4). Der Datenfluss, Entscheidungsbedingungen sowie die Ausgaben der Aktionen werden hierbei durch die Farben und Prädikate der CPN-Palette [42] beschrieben. Das CPN *Bücher ausleihen* beschreibt beispielhaft einen Datenfluss mit der Variable *t* des AVD *Bücher ausleihen* durch die entsprechenden Variablen und Farben (vgl. Abb. 4.6 und Abb. 4.9). Die Entscheidungsbedingungen *t=Ok* und *t=Abbrechen* werden durch die entsprechenden *Guards* der Transitionen beschrieben (siehe Kapitel 2.2.1). Die Beschreibung der Ausgabe der Transition *Bücher auswählen* erfolgt durch einen Codeab-

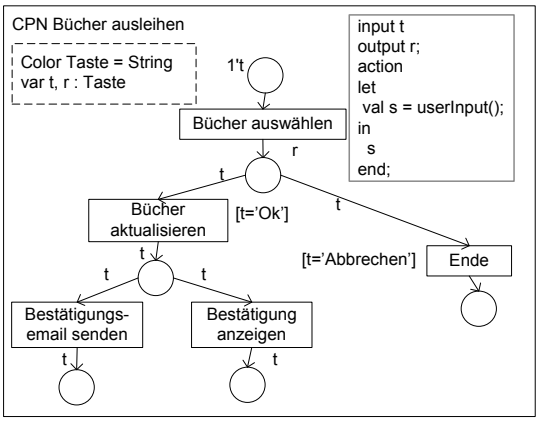


Abbildung 4.9: Petri-Netz - Bücher ausleihen

schnitt. Die Ausgabe der Schnittstellenaktion hängt von der Benutzereingabe ab. Daher wird der Rückgabewert der Transition auch durch eine Benutzereingabe bestimmt. Die Serviceaktionen sind in diesem Beispiel noch nicht implementiert. Die Ausgabe ist damit nicht-deterministisch und entspricht damit einer zufälligen Variablenbelegung oder ebenfalls einer Benutzereingabe.

Konsistenz

Eine Interaktionsskizze des ID kann einen Querbezug zu einer Schnittstellenaktion haben. Übertragen auf das allgemeine Modell hat eine Nachricht des szenarienbasierten Modells einen Querbezug zu einer Transition des zustandsbasierten Modells. Das MSC *msc1* stellt beispielsweise das szenarienbasierte Modell dar und das Petrinetz *pn1* das zustandsbasierte Modell (siehe Abb. 4.10). Die Nachrichten 1 und 2 haben einen Querbezug zu den Transitionen 1 und 2. Ein geeigneter semantischer Zusammenhang szena-

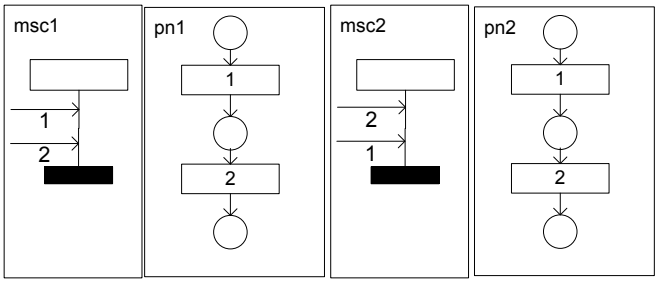


Abbildung 4.10: Querbezug der Modelle

rienbasierter und zustandsbasierter Modelle kann informell wie folgt definiert werden:

Genau dann wenn in einem Szenario eine Nachricht ausgetauscht wird, muss die in Beziehung stehende Transition schalten.

Die Transition *T1* muss beispielsweise schalten, wenn die Nachricht *1* ausgetauscht wird. Das szenarienbasierte Modell beschreibt eine Menge *S* von Szenarien. Jedes Szenario $s \in S$ ist eine Folge von Nachrichten. Das Szenario *msc*1 beschreibt beispielsweise die Folge <1,2> (siehe Abb. 4.10). Das zustandsbasierte Modell beschreibt eine Menge *P* von Schaltsequenzen der Transitionen im Petrinetz. Das Petrinetz *pn*1 beschreibt die Schaltsequenz-Menge {<1,2>}. Durch den Zusammenhang von Nachrichten und Transitionen, werden durch die Szenarien in S Schaltsequenzen in P gefordert. P legt fest, welche Schaltsequenzen möglich sind. Wenn sich die Sequenzen nicht widersprechen, sind die Modelle konsistent. Durch das MSC *msc1* wird durch die Nachrichtenfolge <1,2> die aufeinanderfolgende Schaltung der Transitionen *1* und *2* im Petrinetz gefordert (siehe Abb. 4.10). Im *pn1* ist die Schaltsequenz {<1, 2>} möglich. Die Modelle widersprechen sich nicht und die Modelle sind konsistent. Durch MSC *msc2* ist die Nachrichtenfolge <2, 1> beschrieben, die die Schaltsequenz <2, 1> in *pn2* fordert (siehe Abb. 4.10). Im *pn2* ist die Schaltsequenz {<1, 2>} möglich. Die geforderte Schaltsequenz ist nicht möglich. Die Modelle widersprechen sich und sind inkonsistent.

4.2.4 Anforderungen an die Konsistenzsicherung

Konsistenzbeziehung

Anforderungen müssen im Entwicklungsverlauf verifizierbar sein und müssen daher separat von der Architektur spezifiziert werden. Die Anforderungen müssen zudem klar erkennbar sein und dürfen nicht mit Lösungsdetails angereichert werden. Das Architekturmodell muss daher für die Lösungsdetails im Allgemeinen mehr Schaltsequenzen beschreiben als das Anforderungsmodell. Zwischen den Modellen besteht folglich eine Verfeinerungsbeziehung. Zu einem Anforderungsmodell existieren damit mehrere konsistente Architekturmodelle. Die von einem Architekturmodell über das Anforderungsmodell geforderten Schaltsequenzen können hierbei variieren. Zu einem Architekturmodell existieren folglich auch mehrere konsistente Anforderungsmodelle. Das Petrinetz *pn3* beschreibt beispielsweise eine Schleife in der die Transitionen *1* und *V* abwechselnd schalten können (siehe Abb. 4.11). Die Schleife wird beendet, wenn nach Transition *1* die Transition *2* schaltet. Vom MSC *msc3* wird durch die Nachrichten *1* und *2* das einmalige Schalten der Transitionen *1* und *2* gefordert. Die Transition *V* stellt einen Verarbeitungsschritt des Systems dar, der das interne Verhalten des Systems detaillierter beschreibt. *V* realisiert keine Interaktion mit externen Systemen und den Benutzern. Damit sind diese Transitionen für Anforderungen irrelevant und stellen Verfeinerungen dar. Da das Architekturmodell mehr Schaltsequenzen als das Anforderungsmodell beschreiben darf, ist das szenarienbasierte Modell *msc3* konsistent zu *pn3*. Dies ist unabhängig davon, ob die Transition *V* im zustandsbasierten Modell enthalten ist. Umgekehrt sei folgender Fall gegeben: Das szenarienbasierte Anforderungsmodell fordert die Schaltsequenz <1, 1, 2>. Die Transition *V* ist für die Anforderungen irrelevant. Durch *V* kann die Transition *1* jedoch mehrfach vor der Transition *2* schalten. Die Modelle sind damit auch in diesem Fall konsistent. Zu einem zustandsbasierten Modell existieren folglich mehrere konsistente szenarienbasierte Modelle.

Seien die Mengen N und M Metamodelle der Modelle S sowie P und $R \subseteq M \times N$ die

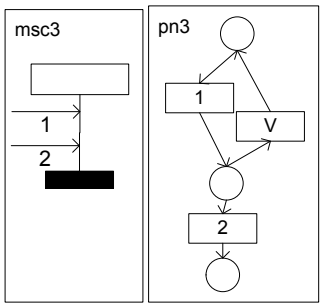


Abbildung 4.11: Konsistenzbeziehung

Konsistenzbeziehung [86]. Sei $(a,b) \in R$, wenn die Modelle a und b konsistent sind. Da durch die Verfeinerungsbeziehung für ein Anforderungsmodell mehrere konsistente Architekturmodelle existieren und umgekehrt ist die Konsistenzbeziehung weder bijektiv noch injektiv.

Manuelle Entscheidungen

Die zustandsbasierte Verhaltensbeschreibung der Architektur ist in den frühen Phasen abstrakt. Die Transitionen sind demnach nicht implementiert und die Auswirkungen der Schaltung einer Transition sind teilweise nur informell beschrieben. Zwischen Transitionen können demnach Abhängigkeiten bestehen, die nicht formal beschrieben sind und damit nicht automatisiert erkannt werden können. Eine automatisierte Konsistenzsicherung muss daher manuelle Entscheidungen beinhalten. Das Petrinetz *pn4* beschreibt beispielsweise ein zustandsbasiertes Modell bei der eine Transition *X* nach der Transition *1* schaltet (siehe Abb. 4.12). Die Schaltung der Transition *1* wird vom szenarienbasierten Modell *msc4* gefordert und die Transition *X* ist ein systeminterner Verarbeitungsschritt, der für die Anforderungen irrelevant ist. Das MSC *msc4'* stellt ei-

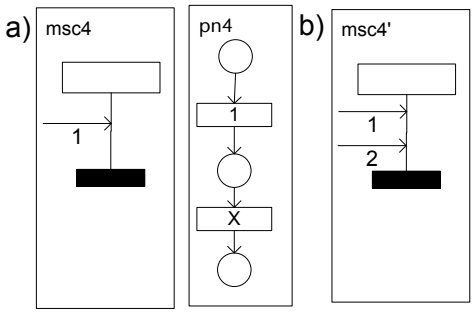


Abbildung 4.12: Manuelle Entscheidung

ne Weiterentwicklung des Anforderungsmodells dar. Das *msc4'* fordert zusätzlich die Schaltung einer Transition *2*. Für Herstellung der Konsistenz bestehen die folgenden

Anpassungsmöglichkeiten am zustandsbasierten Modell:

1. Transition *2* schaltet vor *X*.
2. Transition *2* schaltet nach *X*.
3. Transition *2* schaltet alternativ zu *X*.
4. Transition *2* schaltet parallel zu *X*.

Da die Transitionen *1*, *2* und *X* noch nicht implementiert sind, können informelle Abhängigkeiten existieren. Beispielsweise kann *2* ggf. nur alternativ zu *X* schalten, da sonst redundante Daten produziert würden. Diese Abhängigkeiten können insbesondere in den frühen Phasen des Architekturentwurfs nicht automatisiert erkannt werden. Das Verfahren zur Konsistenzsicherung muss daher manuelle Entscheidungen beinhalten. Eine Anpassung des zustandsbasierten Modells hat gegebenenfalls grundlegende Auswirkungen auf die Architektur und erhebliche Aufwände zur Folge. Daher existiert gegebenenfalls keine geeignete Anpassungsmöglichkeit. In diesem Fall muss das szenarienbasierte Modell geändert werden. Das Verfahren zur Konsistenzsicherung muss daher Änderungsmöglichkeiten sowohl am Anforderungs- und Architekturmodell berücksichtigen.

Ausdrucksmächtigkeit

Viele modellbasierte Ansätze für den Architekturentwurf verwenden Turing-vollständige Modelle. CREATE verwendet zur Verhaltensbeschreibung von Architekturen beispielsweise UML Aktivitätsdiagramme mit Objektflüssen. Diese haben die Ausdrucksmächtigkeit höherer Petrinetze wie das CPN [88]. CPNs sind Turing-vollständig. Das Verfahren zur Konsistenzsicherung von Anforderungs- und Architekturmodellen muss daher auch praktisch anwendbar sein, wenn das architektonische Verhaltensmodell Turing-vollständig ist.

4.3 Bestehende Ansätze

Verfahren zur Konsistenzsicherung von Modellen können kategorisiert werden in Modelltransformationsverfahren und Konsistenzüberprüfungsverfahren. Für eine Modelltransformation existieren sowohl unidirektionale Verfahren als auch bidirektionale Verfahren. Im Folgenden werden die Verfahren zur Konsistenzsicherung von Modellen auf ihre Eignung zur Konsistenzsicherung von Anforderungs- und Architekturmodellen untersucht.

4.3.1 Unidirektionale Modelltransformation

Ein Ansatz zur Konsistenzsicherung von Modellen ist die unidirektionalen Modelltransformation. Speziell für die Transformation von szenarien- zu zustandsbasierten Modellen stehen Syntheseverfahren zur Verfügung [51]. Diese Ansätze ermöglichen

eine automatisierte Generierung vollständiger zustandsbasierter Modelle aus Szenarien. Im Ansatz [49] werden beispielsweise Zustandsautomaten aus szenarienbasierten MSCs synthetisiert. Neben diesen Verfahren existieren Ansätze zur Transformation von Szenarien zu Petrinetzen [27].

Die Modelle S und P werden bei der Weiterentwicklung der Anforderungen und der Architektur manuell geändert. Die Szenarien sind hierdurch potentiell inkonsistent zum zustandsbasierten Modell, so dass gilt $(S,P) \notin R$. Um Konsistenz herzustellen, ist eine erneute unidirektionale Transformation erforderlich. Sei die Transformation vom Quell- zum Zielmodell und umgekehrt durch die Funktionen $f : N \rightarrow M$ resp. $g : M \rightarrow N$ gegeben [86]. Wenn $P = f(S)$ gilt auch $(S,P) \in R$, da die unidirektionale Transformation von S zu einer konsistenten Menge P führt. Sei nun P' eine Weiterentwicklung von P. Für die Weiterentwicklung gilt auch $(S,P') \in R$. Dies ist möglich, da zu einem Anforderungsmodell mehrere konsistente Architekturmodelle existieren. Bei einer erneuten unidirektionalen Transformation gilt $f(S) = P \neq P'$. Die Weiterentwicklungen werden von f nicht berücksichtigt. Die Weiterentwicklung des zustandsbasierten Modells ist ein Ziel der Architekturerstellung und kann mit erheblichem Aufwand verbunden sein. Der Verlust der Informationen über die Weiterentwicklung ist daher gravierend. Abb. 4.13 a) zeigt beispielsweise die Synthese des MSC msc4 zu einem Zustandsautomaten sc1 mit dem in [49] beschriebenen Syntheseverfahren. Das Szenario msc4 beschreibt die Nachrichtensequenz $\{< m1 >\}$. Vor und nach der Nachricht m1 werden die Zustände z1 und z2 eingeführt. Die Nachricht m1 stellt eine Transition im Zustandsautomaten dar. Die Menge P ist damit $\{< m1 >\}$. Die Stelle m1 ist mit der Transition m1 verknüpft. Nach der Konsistenzbedingung von S und P gilt $(S,P) \in R$. Bei Weiterenwicklung des zustandsbasierten Modells zu sc1' kann eine neue Transition

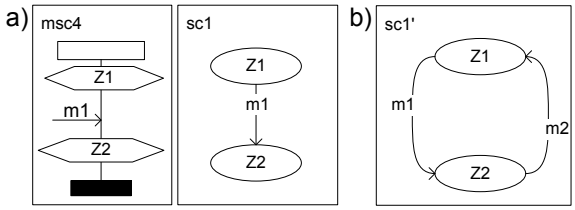


Abbildung 4.13: Änderung der Zustandsbeschreibung

m2 eingeführt werden (siehe 4.13 b)). Die Menge P wird damit auf die Menge P' geändert. In dieser befinden sich alle Schaltsequenzen, in denen m1 und m2 abwechselnd schalten. Da zu einem Anforderungsmodell mehrere konsistente Architekturmodelle existieren (siehe Kapitel 4.2.4), kann weiterhin $(S,P') \in R$ gelten. Bei einer erneuten Transformation ergibt f von S wieder P. Die Transformation von msc4 ergibt wieder sc1. Die Weiterentwicklung kann von f nicht berücksichtigt werden.

Bei der Transformation $g : M \rightarrow N$ ausgehend vom zustandsbasierten Modell besteht dieses Problem ebenfalls. Sei nun S' eine Weiterentwicklung von S. Für die Weiterentwicklung gilt $(S',P) \in R$. Dies ist möglich, da es zu einem Architekturmodell mehrere konsistente Anforderungsmodelle geben kann. Bei einer erneuten unidirektionalen Transformation gilt $g(P) = S \neq S'$. Die Weiterentwicklungen können von f nicht

berücksichtigt werden. Die unidirektionale Transformation von sc1' ergibt beispielsweise msc4. Die Menge S von msc4 kann daraufhin auf die Menge S' $= \{< 1,2 >\}$ geändert werden. Da zu einem Architekturmodell mehrere konsistente Anforderungsmodelle existieren, kann weiterhin $(S',P) \in R$ gelten. Bei einer erneuten unidirektionalen Modelltransformation ergibt g wieder msc4. Die Weiterentwicklung kann von g nicht berücksichtigt werden. Die Szenarien sind von den Anwendern formulierte Anforderungen. Der Verlust der Informationen über die Weiterentwicklung ist daher gravierend. Folglich muss das Zielmodell bei einer Transformation vom Quellmodell berücksichtigt werden.

4.3.2 Bidirektionale Modelltransformation

Eine bidirektionalen Modelltransformation ermöglicht die Berücksichtigung des Zielmodells bei einer Modelltransformation und kann durch die Funktionen $f : M \times N \rightarrow M$ und $g : M \times N \rightarrow N$ beschrieben werden [86]. Ansätze zur bidirektionalen Modelltransformation können kategorisiert werden in Ansätze, die eine vollständige Konsistenz im Sinne einer Bijektion sichern, und Ansätze, die Abstraktion unterstützen. Eine gewünschte Eigenschaft bidirektionaler Modelltransformation ist *check-then-enforce* [86]. Eine Anpassung des Zielmodells erfolgt bei Einhaltung von *check-then-enforce* nur, wenn eine Anpassung erforderlich ist. Eine Anpassung ist erforderlich, wenn die Modelle inkonsistent sind. Inkonsistenzen werden hierbei durch eine Konsistenzüberprüfung aufgedeckt. Der Bearbeiter des angepassten Modells wird so nicht in der Arbeit gestört. Einige Ansätze nehmen jedoch auch Anpassungenen am Zielmodell vor, ohne zuvor eine Konsistenzüberprüfung durchzuführen.

Ansätze zur bidirektionalen Modelltransformation, die eine Konsistenz im Sinne einer Bijektion sichern, werden beispielsweise in [82][29] beschrieben. In [82] erfolgt beispielsweise eine bidirektionale Modelltransformation zwischen UML Aktivitätsdiagrammen und Petrinetzen. Ein bijektive Abbildung zwischen UML Sequenz- und Aktivitätsdiagrammen wird in [29] beschrieben. Da die Konsistenzbeziehung zwischen dem szenarienbasierten Modell und dem zustandsbasierten Modell nicht bijektiv ist, können diese Verfahren zur Konsistenzsicherung nicht eingesetzt werden.

Ansätze, die nicht-bijektive Konsistenzbeziehungen unterstützen, werden beispielsweise in [28], [36], [22] und [23] beschrieben. Hierbei erfolgt die Modelltransformation über sogenannte Linsen (lenses). Eine Linse transformiert ein konkretes in ein abstraktes Modell [28]. Die Modelltransformation kann sowohl asymmetrisch als auch symmetrisch sein [22]. Asymmetrisch bedeutet, dass N eine Abstraktion von M ist. Wenn M auch eine Abstraktion von N ist, liegt ein symmetrischer Fall vor. Eine bidirektionale Modelltransformation kann über die Funktionen $fPpg(N',M,N) = N'$ und $gPpg(M',M,N) = N'$ beschrieben werden [23]. Unterschiedliche Interpretationen der selben Änderung können verschiedene Auswirkungen auf das Zielmodell haben. Diese Problematik wird unter anderem in [23] behandelt und wird durch *delta-lenses* gelöst. Diese ermöglichen Änderungen eindeutig zu beschreiben. Die Beschreibungen werden in die Modelltransformation einbezogen.

Wie im Abschnitt 4.2.4 beschrieben, existieren meistens mehrere Anpassungsmöglichkeiten zur Herstellung der Konsistenz. In den frühen Phasen des Architekturentwurfs sind die Beschreibungen abstrakt und zwischen den Transitionen können infor-

mell beschriebene Abhängigkeiten existieren. Die Abbildung $fPpg(msc5', msc5, hcpn5')$ muss daher manuelle Entscheidungen beinhalten. Zur Herstellung der Konsistenz zwischen msc4' und hcpn4 muss beispielsweise manuell zwischen den Anpassungsmöglichkeiten 1-4 entschieden werden (siehe Abschnitt 4.2.4, Abb. 4.12). Einige Ansätze erlauben manuelle Entscheidungen bei der bidirektionalen Modelltransformation [7]. Das zustandsbasierte Architekturmodell ist bei vielen Modellierungsansätzen wie CREATE jedoch Turing-vollständig. Die Anzahl der Anpassungsmöglichkeiten steigt damit stark mit der Größe des Modells. Die manuelle Prüfung jeder dieser Anpassungsmöglichkeiten auf ihre Eignung hin ist nicht praktikabel. Im Abschnitt 4.2.4 ist zudem aufgeführt, dass eine Anpassung gegebenenfalls grundlegende Auswirkungen auf die Architektur und erhebliche Aufwände zur Folge haben kann. Daher existiert gegebenenfalls keine geeignete Anpassungsmöglichkeit. In diesem Fall muss das Quellmodell geändert werden. Dies ist durch eine bidirektionale Modelltransformation mit der Funktion fPpg nicht zu erreichen. Umgekehrt besteht dieses Problem auch für die Funktion gPpg, da Änderungen an der Architektur Anpassungen an den Szenarien der Anforderungen zur Folge haben können. Die Anpassung von geforderten Geschäftsprozessen ist gegebenenfalls nicht möglich.

Nach [85] ist eine automatisierte Konsistenzüberprüfung im Falle mehrerer Anpassungsvarianten eine geeignete bidirektionalen Modelltransformation. Liegen Inkonsistenzen vor, kann der Benutzer mit den Informationen der automatisierten Konsistenzüberprüfung geeignete Lösungen zur Behebung erarbeiten.

4.3.3 Konsistenzüberprüfung

Konsistenzüberprüfungsverfahren können kategorisiert werden in Verfahren, die auf Konsistenz im Sinne einer Bijektion oder mittels *Model Checking* prüfen. Verfahren der ersten Kategorie werden beispielsweise in mehreren Ansätzen zur Synthese von zustandsbasierten zu szenarienbasierten Modellen vorgestellt [51]. In [45] werden UML Kollaborationsdiagramme zu partiellen Objektspezifikationen in Form von Zustandsdiagrammen transformiert. Anschließend werden diese zu vollständigen Objektspezifikationen zusammengefügt. Hierbei erfolgen Konsistenzüberprüfungen. Ein Zustandsdiagramm ist inkonsistent, wenn es nicht-deterministisch ist. Die Kollaborationsdiagramme sind inkonsistent, wenn diese bezüglich des zustandsbasierten Modells nicht vollständig sind. Das zustandsbasierte Modell der Architektur basiert auf Petrinetzen und ist damit nicht-deterministisch. Zudem ist die Konsistenzbeziehung nicht bijektiv und in der Architektur werden Zustände beschrieben, die für das Anforderungsmodell nicht relevant sind (siehe Kapitel 4.2.4). Dieser Ansatz ist daher zur Konsistenzüberprüfung von Anforderungs- und Architekturmodellen nicht geeignet. In den Ansätzen [97] und [34] erfolgt die Konsistenzüberprüfung zwischen den Szenarien. Darauf folgend wird das zustandsbasierte Modell generiert. Im Ansatz [97] werden UML-Zustandsdiagramme aus Sequenzdiagrammen generiert. Durch das Hinzufügen semantischer Informationen zu den Sequenzdiagrammen in Form von Zustandsvektoren können gleiche Zustände identifiziert und Inkonsistenzen festgestellt werden. Eine Konsistenzüberprüfung von manuell weiterentwickelten szenarienbasierten und zustandsbasierten Modellen wird in diesen Arbeiten nicht beschrieben.

Die Ansätze in [11] und [15] basieren auf einer Konsistenzüberprüfung von szena-

rienbasierten und zustandsbasierten Modellen mittels *Model Checking*. In [11] ist das szenarienbasierte Modell eine Menge von *Universal Life Sequence Charts* (uLSCs). Das zustandsbasierte Modell wird durch *Input-Output-Automaten* (I/O-Automaten) beschrieben. Mittels Model Checking wird geprüft, ob das zustandsbasierte Modell dem szenarienbasierten Modell genügt. In [15] erfolgt die Konsistenzüberprüfung zwischen Aktionssystemen und reaktiven Systemen. Ein reaktives System führt hierbei eine Folge von Aktionen aus. Zwischen den Aktionen erfolgt jeweils ein Ereignis. Beim Aktionssystem erfolgt kein Ereignis zwischen den Aktionen. Ein reaktives System ist konsistent zu einem Aktionssystem, wenn das reaktive System verklemmungsfrei bei den durch das Aktionssystem ausgelösten Ereignissen ist. Die Konsistenzüberprüfung erfolgt durch den Model-Checker SPIN [37]. SPIN transformiert hierfür die Modelle in endliche Automaten.

Bei einer Konsistenzüberprüfung des szenarienbasierten Modells der Anforderungen und des zustandsbasierten Modells der Architektur muss geprüft werden, ob die von den Szenarien geforderte Schaltsequenz im zustandsbasierten Modell realisiert ist. Eine Transition schaltet, wenn ein bestimmter Zustand erreicht ist. Das Problem der Konsistenzüberprüfung kann damit auf das Problem der Erreichbarkeit eines Zustands im zustandsbasierten Modell reduziert werden. Geeignete zustandsbasierte Architekturmodelle wie das AVD haben die Mächtigkeit eines CPN, welches Turing-vollständig ist [64][89]. Das Erreichbarkeitsproblem eines bestimmten Zustands in einer Turingmaschine ist nicht entscheidbar [93]. Entsprechend ist auch das Erreichbarkeitsproblem in einem CPN nicht entscheidbar [64]. Hieraus folgt die Nicht-Entscheidbarkeit der Konsistenzüberprüfung des szenarienbasierten Modells der Anforderungen und des zustandsbasierten Modells der Architektur. In der Arbeit [74] wird ein Ansatz vorgestellt, bei dem die Konsistenzüberprüfung auf Syntaxebene erfolgt. Die Ausführungssemantik wird hier jedoch nicht vollständig berücksichtigt und Konstrukte wie Hierarchien werden nicht betrachtet.

4.4 Resultierende Aufgabestellung

Im Verlauf der Softwareentwicklung können die Anforderungen und die Architektur unabhängig voneinander weiterentwickelt werden. Inkonsistenzen zwischen den Spezifikationen führen zu Fehlentwicklungen. Anforderungen und Architekturen müssen folglich während des gesamten Entwicklungsprozesses konsistent zueinander gehalten werden. Die Strukturmodelle des betrachteten modellbasierten Ansatzes CREATE sind auf der Typebene miteinander verknüpft. Dieser Zusammenhang erlaubt die Definition einfacher Konsistenzbedingungen, die eine automatisierte Konsistenzsicherung erlauben. Eine besondere Herausforderung ist die Konsistenzsicherung der Verhaltensmodelle. Nach [51] beschreiben szenarienbasierte Modelle Interaktionen zwischen Objekten. Zustandsbasierte Modelle beschreiben das vollständige Verhalten von Systemen. Szenarienbasierte Modelle wie das SD und ID des betrachteten modellbasierten Ansatzes sind für die Beschreibung von Anforderungen geeignet. Zustandsbasierte Modelle wie das AVD des betrachteten modellbasierten Ansatzes sind für die Beschreibung von Architekturen geeignet. Zur Konsistenzsicherung dieser Modelle können die grundlegenden Ansätze unidirektionalen Modelltransformation, bidirektionale Modell-

transformation und Konsistenzüberprüfung angewendet werden. Anforderungs- und Architekturmodelle müssen eine Verfeinerungsbeziehung haben, womit eine Architektur auch nach Weiterentwicklungen konsistent zu den Anforderungen sein kann. Bei einer unidirektionalen Modelltransformation wird das Zielmodell nicht berücksichtigt. Weiterentwicklungen des Zielmodells werden daher im Allgemeinen nicht beibehalten werden. Eine bidirektionale Modelltransformation berücksichtigt das Zielmodell. Das zustandsbasierte Modell der Architektur ist jedochh abstrakt und Abhängigkeiten zwischen den Transitionen können nicht automatisiert erkannt werden. Eine Transformation muss daher manuelle Entscheidungen beinhalten. Bei einer bidirektionalen Modelltransformation muss folglich jede einzelne Anpassungsmöglichkeit des Zielmodells analysiert werden. Die Menge der Anpassungsmöglichkeiten steigt stark mit der Größe des Turing-vollständigen Architekturmodells. Eine Modelltransformation stellt demnach keine geeignete Lösung dar.

Eine Konsistenzüberprüfung der Modelle erscheint durch manuelle Entscheidungen bei der automatisierten Konsistenzsicherung am besten geeignet. Die Konsistenzüberprüfung szenarienbasierter und zustandsbasierter Modelle durch Prüfung des gesamten Zustandsraums ist jedoch im Allgemeinen nicht entscheidbar. In der Arbeit [74] wird ein Ansatz vorgestellt, bei dem die Konsistenzüberprüfung auf Syntaxebene erfolgt. Die Ausführungssemantik wird hier jedoch nicht vollständig berücksichtigt und Konstrukte wie Hierarchien werden nicht betrachtet. Eine entscheidbare oder sogar effiziente Konsisistenzüberprüfung mit vollständiger Berücksichtigung der Ausführungssemantik der Modelle kann durch die Einschränkung der Syntax erreicht werden. Für die praktische Einsetzbarkeit des modellbasierten Ansatzes sind jedoch Konstrukte wie beispielsweise Schleifen, Hierarchien mit Rekursion und parallele Abläufe im zustandsbasierten Modell erforderlich. Zum Beispiel wäre die Beschreibung eines Loginvorgangs mit mehreren Versuchen ohne Schleifen nicht praktikabel. Ein genereller Ausschluss dieser Konstrukte ist damit keine geeignete Lösung. Die Herausforderung der Realisierung einer entscheidbaren und effizienten Konsistenzüberprüfung mit vollständiger Berücksichtigung der Ausführungssemantik ist die Entwicklung geeigneter Syntaxeinschränkungen. Diese müssen sowohl eine entscheidbare und effiziente Konsistenzüberprüfung ermöglichen als auch die Ausdrucksmächtigkeit der Modelle beibehalten.

Ziel der Arbeit ist die Entwicklung einer entscheidbaren und effizienten Konsistenzüberprüfung durch geeignete Einschränkungen des szenarienbasierten Anforderungsmodells und des zustandsbasierten Architekturmodells. Die Ausdrucksmächtigkeit der Modelle soll hierbei für den praktischen Einsatz beibehalten werden. Ein genereller Auschluss von beispielsweise Schleifen ist demnach keine geeignete Lösung.

Kapitel 5

Lösungskonzept

Im Verlauf der Softwareentwicklung werden Anforderungen und Architekturen iterativ und evolutionär weiterentwickelt. Inkonsistenzen zwischen den Spezifikationen führen zu Fehlentwicklungen. Anforderungen und Architekturen müssen folglich während des gesamten Entwicklungsprozesses konsistent zueinander gehalten werden. Die Konsistenz von Strukturmodellen zur Beschreibung von Anforderungen und Architekturen kann zwischen Typen definiert werden. Dieser Zusammenhang erlaubt die Definition einfacher Konsistenzbedingungen, die eine automatisierte Konsistenzsicherung erlauben. Eine besondere Herausforderung ist die Konsistenzsicherung der Verhaltensmodelle. Wie in Kapitel 4 gezeigt, sind szenarienbasierte Modelle für die Anforderungserhebung und zustandsbasierte Modelle für den Architekturentwurf geeignet. Zudem wird gezeigt, dass eine automatisierte Konsistenzüberprüfung szenarienbasierter und zustandsbasierter Modelle für die Automatisierung der Konsistenzsicherung von Anforderungen und Architekturen wegen der Vielzahl an manuellen Entscheidungen am geeignetsten erscheint. Eine derartige Konsistenzüberprüfung ist durch Prüfung des gesamten Zustandsraums im Allgemeinen nicht entscheidbar. Eine entscheidbare und effiziente Konsisistenzüberprüfung mit vollständiger Berücksichtigung der Ausführungssemantik kann durch die Einschränkung der Syntax der verwendeten Modelle erreicht werden. Für die praktische Einsetzbarkeit des Ansatzes sind jedoch Konstrukte wie Schleifen und Hierarchien mit Rekursion im zustandsbasierten Modell erforderlich. In diesem Kapitel werden Syntaxeinschränkungen und Konsistenzbedingungen vorgestellt, die eine effiziente Konsistenzüberprüfung ermöglichen und die Ausdrucksmächtigkeit der Modelle nicht verringern. Kapitel 5.1 beinhaltet die Beschreibung des grundlegenden Ansatzes. Im Abschnitt 5.2 wird für die Konsistenzüberprüfung ein geeignetes szenarienbasiertes und zustandsbasierten Modell in Form einer abstrakten Syntax definiert. Eine Abbildung der Modelle des modellbasierten Ansatzes CREATE auf diese Syntax erfolgt im Abschnitt 5.3. Abschnitt 5.4 beinhaltet eine Definition der Zusammenhänge des szenarienbasierten und zustandsbasierten Modells. In Abschnitt 5.5 werden die Syntaxeinschränkungen hergeleitet, die eine effiziente Konsistenzüberprüfung ermöglichen. Die formalen Konsistenzbedingungen werden in Kapitel 5.6 eingeführt.

5.1 Grundlegender Ansatz

Der grundlegende Ansatz zur Realisierung einer entscheidbaren und effizienten Konsistenzüberprüfung szenarienbasierter Anforderungsmodelle und zustandsbasierter Architekturmodelle ist die Einschränkung der Syntax. Bei der Einschränkung der Syntax wird die Ausdrucksmächtigkeit der Modelle nicht verringert. Vielmehr werden die Variationen zur Beschreibung eines Sachverhalts eingeschränkt und Regeln für den Zusammenhang der szenarienbasierten und zustandsbasierten Modelle festgelegt. Hierfür wird eine abstrakte Syntax eingeführt, die eine genauere Differenzierung zwischen Konstrukten wie beispielsweise verschiedenen Schleifentypen ermöglicht. Durch diese Differenzierung können Syntaxeinschränkungen definiert werden, die eine entscheidbare und effiziente Konsistenzüberprüfung erlauben und die Ausdrucksmächtigkeit der Modelle für die praktische Einsetzbarkeit des modellbasierten Ansatzes beibehalten.

Die abstrakte Syntax des szenarienbasierten Modells basiert auf Bäumen [83] und des zustandsbasierten Modells auf Graphen [83]. Ein Szenario wird jeweils durch einen Baum beschrieben. Die Knoten und Kanten repräsentieren die Szenarienschritte und Übergänge des szenarienbasierten Modells. Nachrichten zwischen Objekten werden durch Interaktionen repräsentiert, die an einer Menge von Szenarienschritten aktiv sind. Interaktionen werden daher den Knoten des Szenarios zugewiesen. Der Baum des Szenarios *Bücher ausleihen* beschreibt beispielsweise drei Szenarienschritte. Im zweiten Szenarienschritt ist die Interaktion *Bücher auswählen* aktiv (vgl. Abb. 4.2 und Abb. 5.1). Die Wurzel und die Blätter des Baums stellen hierbei den Start und das Ende des

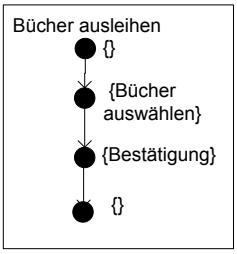


Abbildung 5.1: Szenario-Baum Bücher ausleihen

Szenarios dar, an denen keine Interaktion aktiv ist. Die Abbildung anderer szenarienbasierter Modelle auf diese Struktur kann analog erfolgen. Die abstrakte Syntax der Baumstruktur wird im Abschnitt 5.2 formal definiert.

Ein Teil eines zustandsbasierten Modells wie beispielsweise eine Aktivität wird jeweils durch einen Graph beschrieben. Die Knoten und Kanten des Graphs repräsentieren die Knoten und Kanten des zustandsbasierten Modells. Der Graph zur Aktivität *Bücher ausleihen* besteht beispielsweise aus den Knoten *Bücher auswählen*, *Bücher aktualisieren*, *Bestätigungsemail senden* und *Bestätigung anzeigen* (vgl. Abb. 4.6 und Abb. 5.2). Die Abbildung anderer zustandsbasierter Modelle auf diese Struktur kann analog erfolgen. Die abstrakte Syntax der Graphenstruktur wird im Abschnitt 5.2 formal definiert. Im Abschnitt 5.3 wird die Abbildung der Modelle des modellbasierten Ansatzes CREATE auf die abstrakte Syntax beschrieben. Die für die Konsistenzüber-

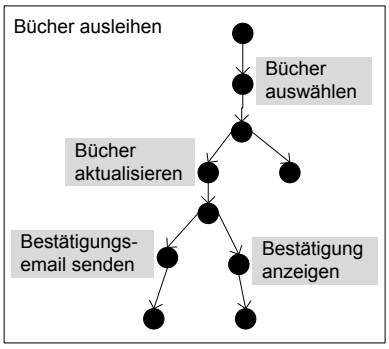


Abbildung 5.2: Zustandsmodell-Graph Bücher ausleihen

prüfung erforderliche Ausdrucksmächtigkeit des szenarienbasierten und zustandsbasierten Modells von CREATE bleibt hierbei erhalten.

Wie in Kapitel 4 gezeigt, kann eine Nachricht des szenarienbasierten Modells beispielsweise mit einer Transition des zustandsbasierten Modells in Beziehung stehen. Eine Interaktion eines Szenarios ist entsprechend mit einem Knoten des zustandsbasierten Modells verknüpft. Diese werden als Interaktionsknoten bezeichnet und beschreiben eine Eingabe in das System oder eine Ausgabe des Systems. Beim Szenario *Bücher ausleihen* sind beispielsweise die Interaktionen *Bücher auswählen* und *Bestätigung* mit Interaktionsknoten des zustandsbasierten Modells verknüpft (siehe Abb. 5.3).

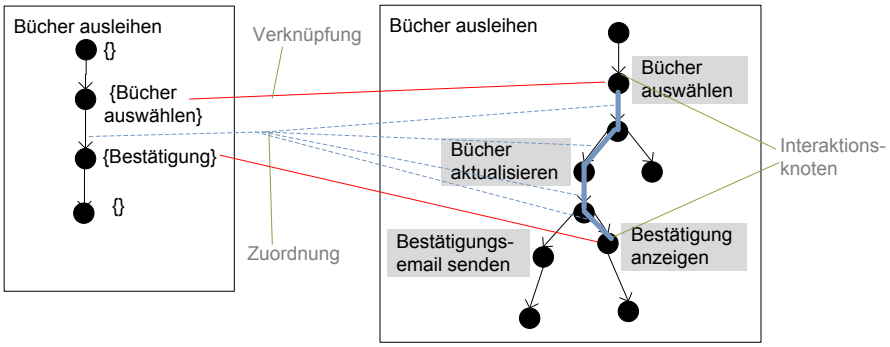


Abbildung 5.3: Zuordnung der Syntaxelemente

Auf Basis dieser Verknüpfungen werden die Zusammenhänge des szenarienbasierten und zustandsbasierten Modells definiert. Wenn eine Interaktion beendet wird, erfolgt ein Übergang zum nächsten Szenarienschritt. Eine Interaktion ist genau dann aktiv, wenn sich der Kontrollfluss am verknüpften Knoten des zustandsbasierten Modells befindet. Der Kontrollfluss wird demnach bei Abschluss einer Interaktion am verknüpften Interaktionsknoten fortgesetzt. Bei Abschluss der Interaktion *Bücher auswählen* wird beispielsweise der Kontrollfluss am Interaktionsknoten *Bücher auswählen* fortge-

setzt (siehe Abb. 5.3). Die Kanten des zustandsbasierten Modells bilden unter anderem eine Menge von Pfaden [83], die vom verknüpften Interaktionsknoten zu weiteren Interaktionsknoten führen. Die verbindenden Kanten können beim Übergang zum nächsten Szenarienschritt vom Kontrollfluss durchlaufen werden. Die durchlaufenen Kanten werden der Kante des szenarienbasierten Modells zugeordnet. Die Kanten zwischen den Interaktionsknoten *Bücher auswählen* und *Bestätigung anzeigen* des zustandsbasierten Modells *Bücher ausleihen* werden beispielsweise der Kante zwischen den Szenarienschritten mit den Interaktionen *Bücher auswählen* und *Bestätigung* im Szenario *Bücher ausleihen* zugeordnet (siehe Abb. 5.3). Die Menge der Kanten, die einer einzelnen Kante des Szenarios zugeordnet sind, wird im Folgenden Zuordnung genannt. Durch die Definition der Zuordnungen sind die Zusammenhänge der Modelle genau festgelegt. Im Kapitel 5.4 erfolgt eine formale Definition der Zuordnungen.

Zuordnungen erlauben die Definition differenzierender Syntaxeinschränkungen, die eine entscheidbare und effiziente Konsistenzüberprüfung ermöglichen und die Ausdrucksmächtigkeit der Modelle beibehalten. Die zustandsbasierten Modelle *z2* und *z3* sind beispielsweise Graphen mit Schleifen (siehe Abb. 5.4). Diese Schleifen sollten voneinander differenziert werden. Die Zuordnungen zu den Kanten des Szenarios *s2*

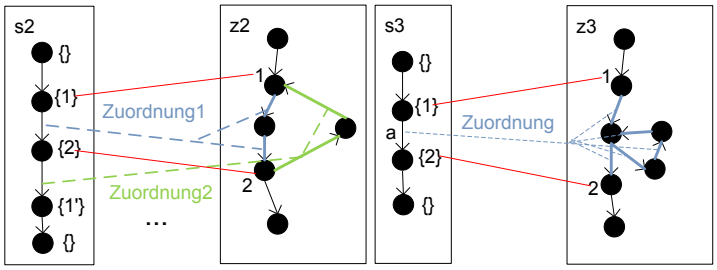


Abbildung 5.4: Grundlegendes Konzept der Syntaxeinschränkung

enthalten beispielsweise keine Schleife. Die Schleife im zustandsbasierten Modell *z2* ausgehend vom Interaktionsknoten *1* wird vom Szenario *s2* genau einmal durchlaufen. Die Ursache der Nicht-Entscheidbarkeit der Konsistenzüberprüfung sind Schleifen innerhalb von Zuordnungen, wie beispielsweise die Zuordnung zur Kante *a* in *s3* (siehe Abb. 5.4). Schleifen innerhalb von Zuordnungen sind jedoch zur Beschreibung der Architektur nicht relevant. Die Architektur muss beschreiben, wie Anforderungen umgesetzt werden. Anforderungen werden im szenarienbasierten Modell durch Interaktionen mit dem zu entwickelnden System beschrieben. Diese werden vom System mit den in Beziehung stehenden Interaktionsknoten realisiert. Die Interaktionen *1* und *2* des Szenarios *s3* werden beispielsweise durch die Interaktionsknoten *1* und *2* realisiert (siehe Abb. 5.4). In einer Zuordnung befinden sich Kanten, die Pfade zwischen Interaktionsknoten bilden. Beispielsweise befindet sich in der Zuordnung ein Pfad zwischen den Interaktionsknoten *1* und *2*. Andere Knoten auf diesen Pfaden realisieren keine Interaktion und stellen Verarbeitungsschritte des Systems dar. Die Anzahl und die Reihenfolge der Verarbeitungsschritte ist aus Sicht der Anforderungen irrelevant, sofern die Ein- und Ausgaben in Form der Interaktionen korrekt realisiert sind. Innerhalb der Zuordnungen können damit Pfade zu einem Knoten zusammengefasst werden.

Dies gilt auch für Pfade mit Schleifen. Die Zuordnung zur Kante *a* des Szenarios *s3* beinhaltet beispielsweise eine Schleife, die in einen Verarbeitungsschritt zusammengefasst werden kann (siehe Abb. 5.4). Durch Syntaxeinschränkungen können Schleifen innerhalb von Zuordnungen ausgeschlossen werden. Da andere Schleifentypen durch diese differenzierende Syntaxeinschränkung weiterhin zugelassen sind, bleibt die Ausdrucksmächtigkeit der Modelle erhalten. Beispielsweise ist die Modellierung eines Loginvorgangs mit Schleifen weiterhin möglich. In Kapitel 5.5 werden die differenzierenden Syntaxeinschränkungen hergeleitet und formal definiert. Diese ermöglichen eine entscheidbare und effiziente Konsistenzüberprüfung. Zudem werden die Auswirkungen der Syntaxeinschränkungen auf die Art der Beschreibung im Detail betrachtet.

Durch die eingeführten Syntaxeinschränkungen bilden die Kanten einer Zuordnung eine Baumstruktur. Hierdurch wird eine vereinfachte Formulierung geeigneter Konsistenzbedingungen ermöglicht. Die Konsistenzbedingungen können über die Blätter der Zuordnungen und den aktiven Interaktionen der Szenarienschritte definiert werden. In Kapitel 5.5 werden die Konsistenzbedingungen zur abschließenden Prüfung formal definiert. Hierbei werden insbesondere auch Hierarchien im zustandsbasierten Modell berücksichtigt.

5.2 Abstrakte Syntax

5.2.1 Allgemeine Eigenschaften der Graphen

Die Graphen des szenarienbasierten und zustandsbasierten Modells haben gemeinsame Eigenschaften, welche im Folgenden beschrieben werden. Die Graphen der Modelle sind gerichtet. Das Universum aller möglicher Graphen wird durch die Menge *G* beschrieben (siehe Definition 1).

Definition 1.

$$G =_{DEF} ND \times ED$$

hierbei ist

- ND die endliche Menge aller Knoten und
- $ED \subseteq ND \times ND$ die endliche Menge aller Kanten.

Folgen von Knoten werden in bestehenden Arbeiten überwiegend durch n-Tupel definiert. Das Universum aller Folgen in den Graphen ist durch N^* (siehe [1] für die Notationskonvention) definiert.

Ein Weg in einem Graph ist nach [83] eine Folge von Knoten, bei der zwei aufeinander folgende Knoten jeweils durch eine Kante verbunden sind. Die Abbildung *isWay* beschreibt, in welchem Fall eine Folge von Knoten bei einer gegebenen Menge von Kanten ein Weg ist (siehe Definition 2 sowie [2] und [3] für die Notationskonvention).

[1] $M^* = \bigcup_{n\in\mathbb{N}} M^n$ mit $M^n = \times_{\{1,\ldots,n\}} M$, M ist eine Menge

[2] $\mathcal{P}(M)$ ist die Potenzmenge von M, M ist eine Menge

[3] $\rightarrow$ bezeichnet eine Funktion, $\rightharpoonup$ eine partielle Funktion und $\leftrightarrow$ eine bijektive Funktion

Definition 2.

$$isWay : ND^* \times \mathcal{P}(ED) \rightarrow B \mid$$
$$isWay(w,E) = \begin{cases} \text{true, wenn } len(w) = 1 \\ \text{false, wenn } len(w) <= 0 \\ \forall x \in \{1,..,len(w)-1\} : \exists (s,t) \in E : s = get(w,x) \wedge \\ \quad t = get(w,x+1), \text{ andernfalls} \end{cases}$$

Die Funktion *len* definiert hierbei die Länge einer Folge (siehe Definition 3).

Definition 3.

$$len : ND^* \rightarrow \mathbb{N} \mid len((v_1,..,v_n)) = n, n \in \mathbb{N}$$

Die Funktion *get* gibt zu einer gegebenen Folge und einem Index einen Knoten der Folge zurück (siehe Definition 4 sowie [4] für die Notationskonvention).

Definition 4.

$$get : ND^* \times \mathbb{N} \rightarrow ND \cup \{\square\} \mid$$
$$get((v_1,..,v_n),i) = \begin{cases} v_i, \text{ wenn } 1 \leq i \leq n \in \mathbb{N} \\ \square, \text{ andernfalls} \end{cases}$$

Ein Weg, in dem jeder Knoten maximal einmal enthalten ist, wird als Pfad bezeichnet [83]. Die Abbildung *isPath* definiert, ob eine Folge von Knoten bei einer gegebenen Menge von Kanten ein Pfad ist (siehe Definition 5).

Definition 5.

$$isPath : ND^* \times \mathcal{P}(ED) \rightarrow B \mid$$
$$isPath(p,E) = isWay(p,E) \wedge \forall x,y \in \{1,..,len(w)\} :$$
$$x \neq y \Rightarrow get(p,x) \neq get(p,y)$$

Die Menge aller möglicher Pfade mit einem vorgegebenen Beginn und einer gegebenen Kantenmenge ist durch die rekursive Funktion *pathsFrom* definiert (siehe Definition 6). Der Beginn eines Pfades ist das erste Argument der Funktion. Die Kantenmenge ist das zweite Argument. Beispielsweise ist die Menge der Pfade ausgehend vom Knoten *1* eines gegebenen Graphs durch $pathsFrom(<1>,\{a,b\})$ definiert (siehe Abb. 5.5).

Definition 6.

$$pathsFrom : ND^* \times \mathcal{P}(ED) \rightarrow \mathcal{P}(ND^*) \mid$$
$$pathsFrom(p,E) = \begin{cases} \emptyset, \text{ wenn nicht } isPath(p,E) \\ \{p\} \bigcup_{\{(s,t) \in E \mid s = get(p,len(p))\}} pathsFrom(\\ \quad append(p,t),E), \text{ andernfalls} \end{cases}$$

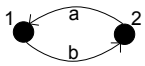


Abbildung 5.5: Einfacher Graph mit Pfaden

Wenn die Kantenfolge des ersten Arguments kein Pfad ist, wird die leere Menge zurückgegeben. Andernfalls wird eine Vereinigungsmenge des vorgegebenen Pfades mit den weiterführenden Pfaden gebildet. Die Knotenfolge $< 1 >$ ist ein Pfad. Die weiterführenden Pfade werden ermittelt, indem über alle Kanten iteriert wird, die vom letzten Knoten des vorgegebenen Pfades ausgehen. Nur die Kante *b* geht beispielsweise vom Knoten *1* aus (siehe Abb. 5.5). Der Knoten, zu dem die jeweilige Kante führt, wird dem Pfad mit der Funktion *append* angehängt (siehe Definition 7). Hieraus resultiert beispielsweise der Pfad $< 1,2 >$. Durch die Selektion der vom letzten Knoten ausgehenden Kanten ist sichergestellt, dass die neue Knotenfolge ein Weg ist. Die neue Knotenfolge ist das erste Argument des rekursiven Aufrufs der Funktion *pathsFrom*. Die Funktion gibt die leere Menge zurück, wenn die neue Knotenfolge kein Pfad ist. Die Knotenfolge $< 1,2,1 >$ ist beispielsweise kein Pfad. Damit ist sichergestellt, dass die Knotenfolge der Vereinigungsmenge Pfade sind. Durch die Iteration über alle ausgehenden Kanten des letzten Knotens eines Pfades ist die Vollständigkeit aller Pfade sichergestellt. Die Vereinigungsmenge ist das Ergebnis der Funktion. Die Menge aller Pfade ausgehend vom Knoten *1* ist beispielsweise die Menge $\{< 1 >, < 1,2 >\}$.

Definition 7.

$$append : ND^* \times ND \rightarrow ND^* \mid$$
$$append((v_1,..,v_n),v_x) = (v_1,..,v_n,v_x), n \in \mathbb{N}$$

5.2.2 Szenarienbasiertes Modell

Die Menge der Knoten und Kanten des Universums aller Szenarien ist als Teilmenge aller Knoten und Kanten definiert (siehe Definition 8).

Definition 8.

$$ND^{SG} \subseteq_{DEF} ND,$$
$$ED^{SG} \subseteq_{DEF} ED$$

Die Menge *SG* aller möglicher Szenarien ist eine Menge endlicher Bäume (siehe Definition 9).

[4] $\square$ ist das leere Element

Definition 9.

$$SG =_{DEF} \begin{cases} (Nd^{SG} \subseteq ND^{SG}, Ed^{SG} \subseteq ED^{SG}, r \in Nd^{SG}, L \subseteq Nd^{SG}) \mid \\ \forall (s,t) \in Ed^{SG} : \{s,t\} \subseteq Nd^{SG} \\ r \text{ ist die Wurzel des Baumes} \\ L \text{ ist die Menge der Blätter des Baumes} \end{cases}$$

Das Szenario *sg* zeigt eine graphische Darstellung der Syntax anhand eines Beispiels (siehe Abb. 5.6). Knoten und Kanten eines Szenarios müssen eindeutig einem Graph

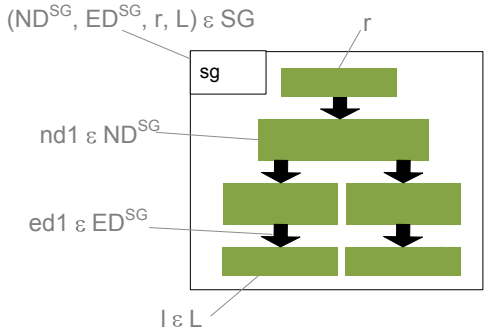


Abbildung 5.6: Szenarienbasiertes Modell

zugeordnet sein. Diese Bedingung wird durch den Ausdruck in Definition 10 beschrieben (siehe [5] für die Notationskonvention).

Definition 10.

$$\forall sg1 = (Nd^{SG}, Ed^{SG}, r, L), sg2 = (Nd'^{SG}, Ed'^{SG}, r', L') \in SG :$$
$$sg1 \neq sg2 \implies Nd^{SG} \cap Nd'^{SG} = \emptyset$$

Ein Baum hat keine zyklen [83]. Damit existiert kein Pfad dessen Anfangs- und Endknoten über eine Kante miteinander verbunden sind (siehe Definition 11).

Definition 11.

$$\forall (Nd^{SG}, Ed^{SG}, r, L) \in SG : \forall nd \in Nd^{SG}, p \in pathsFrom((nd), Ed^{SG}) :$$
$$\nexists (s,t) \in Ed^{SG} : s = get(p, len(p)) \wedge t = nd$$

Die Wurzel eines Baumes hat keine einlaufende Kante [83] (siehe Definition 12).

Definition 12.

$$\forall (Nd^{SG}, Ed^{SG}, r, L) \in SG : \nexists (s,t) \in Ed^{SG} : t = r$$

Ein Beispiel einer Wurzel zeigt das Szenario *sg* (siehe Abb. 5.6).

[5] $\Rightarrow$ bezeichnet eine Implikation und $\Leftrightarrow$ eine Implikation in beide Richtungen

Ausgehend von der Wurzel existiert genau ein Pfad zu jedem Knoten [83] (siehe Definition 13 sowie [6] für die Notationskonvention).

Definition 13.

$$\forall (Nd^{SG}, Ed^{SG}, r, L) \in SG, nd \in Nd^{SG} :$$
$$\exists! p \in pathsFrom((r), Ed^{SG}) : get(p, len(p)) = nd$$

Blätter sind Knoten ohne ausgehende Kante [83] (siehe Definition 14).

Definition 14.

$$\forall (Nd^{SG}, Ed^{SG}, r, L) \in SG :$$
$$L = \{nd' \in Nd^{SG} \mid \nexists (s,t) \in Ed^{SG} : s = nd'\}$$

Im Szenario *sg* ist beispielsweise der Knoten *l* ein Blatt (siehe Abb. 5.6).

5.2.3 Zustandsbasiertes Modell

Die Menge der Knoten und Kanten des Universums aller zustandsbasierter Modelle ist als Teilmenge aller Knoten und Kanten definiert (siehe Definition 15).

Definition 15.

$$ND^{ZG} \subseteq_{DEF} (ND_{id} \times ND_{type}) \subseteq (ND \setminus ND^{SG}),$$
$$ED^{ZG} \subseteq_{DEF} (ED \setminus ED^{SG})$$

hierbei ist

- ND_{id} eine Menge von Knoten-Identifikatoren,
- $ND_{type} =_{DEF} \{y_{init}, y_{final}, y_{fork}, y_{join}, y_{merge}, y_{dec}, y_{cba}, y_{ia}, y_{sa}\}$ die endliche Menge aller Knotentypen. Die Zustandsmodellgraphen *z4* und *z5* zeigen ihre Visualisierung (siehe Abbildung 5.7).

Die Menge aller möglicher zustandsbasierter Modelle wird durch die Menge ZG beschrieben (siehe Definition 16).

Definition 16.

$$ZG =_{DEF} \begin{cases} (Nd^{ZG} \subseteq ND^{ZG}, Ed^{ZG} \subseteq ED^{ZG}) \mid \\ \forall (s,t) \in Ed^{ZG} : \{s,t\} \subseteq Nd^{ZG} \end{cases}$$

Jeder konkrete Knoten hat einen eindeutigen Identifikator (siehe Definition 17).

Definition 17.

$$\forall a = (Nd^{ZG}, Ed^{ZG}), b = (Nd'^{ZG}, Ed'^{ZG}) \in ZG, u = (i,y) \in Nd^{ZG},$$
$$v = (i',y') \in Nd'^{ZG}) : i = i' \Rightarrow a = b \wedge u = v$$

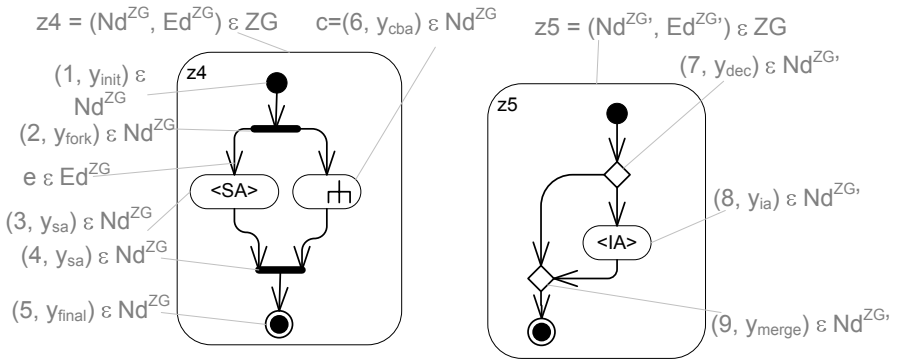


Abbildung 5.7: Zustandsbasiertes Modell

Die Knoten jedes zustandsbasierten Modells müssen eindeutig einem Graphen zugeordnet sein. Diese Bedingung wird durch den Ausdruck in Definition 18 beschrieben.

Definition 18.

$$\forall zg1 = (Nd^{ZG}, Ed^{ZG}), zg2 = (Nd'^{ZG}, Ed'^{ZG}) \in ZG :$$
$$zg1 \neq zg2 \implies Nd^{ZG} \cap Nd'^{ZG} = \emptyset$$

Knoten im zustandsbasierten Modell haben je nach Typ eine zulässige Anzahl eingehender und ausgehender Kanten (siehe Definition 19 sowie Tabelle 5.1 für die Werte *in* und *out*).

Definition 19.

$$\forall (Nd^{ZG}, Ed^{ZG}) \in ZG, nd \in Nd^{ZG} : out_{min}(nd) \leq$$
$$|\{(u,v) \in Ed^{ZG} : u = n\}| \leq out_{max}(nd)$$

$$\forall (Nd^{ZG}, Ed^{ZG}) \in ZG, nd \in Nd^{ZG} : in_{min}(nd) \leq$$
$$|\{(u,v) \in Ed^{ZG} : v = n\}| \leq in_{max}(nd)$$

Des Weiteren ist nur ein initialer Knoten innerhalb eines zustandsbasierten Modells zulässig (siehe Definition 20).

Definition 20.

$$\forall (Nd^{ZG}, Ed^{ZG}) \in ZG : nd = (i,y), nd' = (j,y') \in Nd^{ZG} :$$
$$y = y' \wedge y = y_{init} \Rightarrow i = j$$

[6] $\exists!$ bedeutet es existiert genau ein Element

nType(n)	out_{min}(n)	out_{max}(n)	in_{min}(n)	in_{max}(n)
y_{cba}	1	1	1	1
y_{ia}	1	1	1	1
y_{sa}	1	1	1	1
y_{init}	1	1	0	0
y_{final}	0	0	1	1
y_{dec}	2	$m \in \mathbb{N}$	1	1
y_{merge}	1	1	2	$m \in \mathbb{N}$
y_{fork}	2	$m \in \mathbb{N}$	1	1
y_{join}	1	1	2	$m \in \mathbb{N}$

Tabelle 5.1: Zulässige Anzahl ein- und ausgehender Kanten

Hierarchien im zustandsbasierten Modell

Hierarchien werden im zustandsbasierten Modell ebenfalls durch Kanten beschrieben. Hierbei wird zwischen Aufrufkanten und Rückkehrkanten unterschieden. Aufrufkanten werden durch die Menge ED_{call} definiert (siehe Definition 21).

Definition 21.

$$ED_{call} =_{DEF} \begin{cases} ((i,y) \in ND^{ZG}, (i',y') \in ND^{ZG}) \in ED^{ZG} \mid \\ \exists (Nd^{ZG}, Ed^{ZG}), (Nd'^{ZG}, Ed'^{ZG}) \in ZG : \\ (i,y) \in Nd^{ZG}, (i',y') \in Nd'^{ZG} \wedge y = y_{cba} \wedge y' = y_{init} \end{cases}$$

Hierbei muss für jeden Knoten vom Typ y_{cba} genau eine Kante aus ED_{call} existieren (siehe Definition 22).

Definition 22.

$$\forall (Nd^{ZG}, Ed^{ZG}) \in ZG, a = (i,y) \in Nd^{ZG} :$$
$$y = y_{cba} \Leftrightarrow \exists! (s,t) \in ED_{call} : s = a$$

Die zustandsbasierten Modelle *zg4*, *zg5* und *zg6* zeigen ein Beispiel mit zwei Aufrufkanten von zwei verschiedenen Knoten *6* und *12* vom Typ y_{cba} zu einem Knoten *10* vom Typ y_{init} (siehe Abb. 5.8).

Rückkehrkanten werden durch die Menge ED_{return} definiert (siehe Definition 23).

Definition 23.

$$ED_{return} =_{DEF} \begin{cases} ((i,y) \in ND^{ZG}, (i',y') \in ND^{ZG}) \in ED^{ZG} \mid \\ \exists (Nd^{ZG}, Ed^{ZG}), (Nd'^{ZG}, Ed'^{ZG}) \in ZG : \\ (i,y) \in Nd^{ZG}, (i',y') \in Nd'^{ZG} \wedge y = y_{final} \wedge y' = y_{cba} \end{cases}$$

Hierbei muss für jedes Paar (Knoten vom Typ y_{final}, Aufrufkante) eine Rückkehrkante existieren, wenn sich der Knoten vom Typ y_{final} im selben zustandsbasierten Modell befindet wie der initiale Knoten der Aufrufkante (siehe Definition 24).

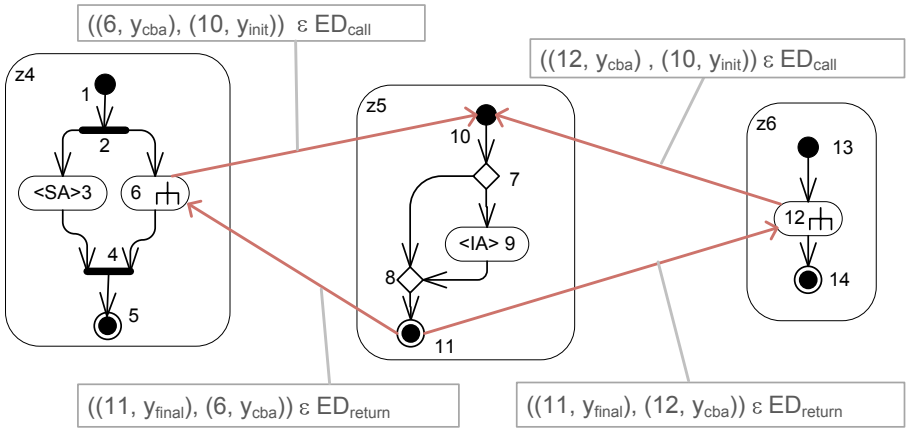


Abbildung 5.8: Aufruf- und Rückkehrkanten

Definition 24.

$$\forall (Nd^{ZG}, Ed^{ZG}) \in ZG, a = (i,y) \in Nd^{ZG}, (s,t) \in ED_{call} :$$
$$y = y_{final} \wedge t \in Nd^{ZG} \Leftrightarrow \exists (a,s) \in ED_{return}$$

Die zustandsbasierten Modelle *zg4*, *zg5* und *zg6* zeigen ein Beispiel mit zwei Rückkehrkanten von einem Knoten *11* vom Typ y_{final} zu zwei verschiedenen Knoten *6* und *12* vom Typ y_{cba} (siehe Abb. 5.8).

5.2.4 Interaktionen im szenarienbasierten Modell

In UML Kommunikationsdiagrammen und MSCs werden Nachrichten beschrieben, die zwischen Objekten ausgetauscht werden. In den hier definierten szenarienbasierten Modell werden Eingaben ins System und Ausgaben vom Systemen durch Interaktionen beschrieben. Interaktionen werden durch die Menge I_{ACT} beschrieben (siehe Definition 25).

Definition 25.

$$I_{ACT} =_{DEF} \begin{cases} (A \subseteq ND^{SG}, E \subseteq ED^{SG} \cup \emptyset) \mid \\ \exists (Nd^{SG}, Ed^{SG}, r, L) \in SG : A \subseteq Nd^{SG} \wedge E \subseteq Ed^{SG} \end{cases}$$

hierbei ist

- A die Menge aller Knoten eines Szenarios, an dem die Interaktion aktiv ist. Im Szenario *sg2* ist beispielsweise die Interaktion *i* an den Knoten *nd2* und *nd4* aktiv (siehe Abb. 5.9).
- E ist eine Menge von Kanten eines Szenarios, mit der die Interaktion beendet wird. Mit einer Kante können mehrere Interaktionen beendet werden. Maximal

eine Interaktion ist jedoch der Auslöser des Übergangs zum nächsten Szenarienschritt. Die Menge E enthält nur die Kanten für die die Interaktion der Auslöser ist. Wenn eine Interaktion bei keiner Kante der Auslöser ist, bleibt E leer. Im Szenario *sg2* ist beispielsweise die Interaktion *i* der Auslöser der Übergänge *ed2* und *ed5* (siehe Abb. 5.9). Die Interaktion *k* ist hingegen bei keinem Übergang der Auslöser.

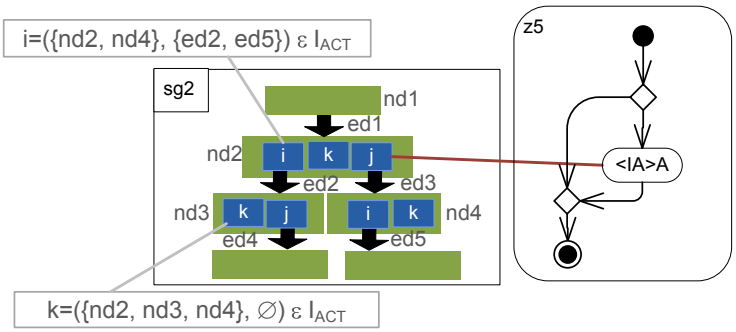


Abbildung 5.9: Fachliche Inhalte in den Graphen

Eine Interaktion darf im Szenario nicht ein zweites Mal aktiviert werden. Eine Interaktion ist damit nur in aufeinander folgenden Szenarienschritten aktiv. Durch alternative Abläufe können hierbei mehrere dieser Pfade ausgehend von dem Knoten existeren, an dem die Interaktion erstmals aktiv ist. In der Menge A muss es folglich einen Knoten geben, von dem aus jeder andere Knoten in A über einen Pfad erreichbar ist (siehe Definition 26).

Definition 26.

$$\forall (A,E) \in I_{ACT} : \exists (Nd^{SG}, Ed^{SG}, r, L) \in SG, r' \in A :$$
$$\forall nd \in A \setminus \{r'\} : \exists p \in pathsFrom((r'), Ed^{SG}) : nd = get(p, len(p)) \wedge$$
$$\forall i \in \{1, .., len(p)\} : get(p,i) \in A$$

Der Wurzel des Szenarios sind keine Interaktionen zugeordnet (siehe Definition 27).

Definition 27.

$$\forall (Nd^{SG}, Ed^{SG}, r, L) \in SG, (A,E) \in I_{ACT} : r \notin A$$

Den Blättern des Szenarios sind ebenfalls keine Interaktionen zugeordnet (siehe Definition 28).

Definition 28.

$$\forall (Nd^{SG}, Ed^{SG}, r, L) \in SG, (A,E) \in I_{ACT} : A \cap L = \emptyset$$

Es darf maximal eine Interaktion der Auslöser eines Übergangs zum nächsten Szenarienschritt sein (siehe Definition 29).

Definition 29.

$$\forall a = (A,E), b = (A',E') \in I_{ACT} : \\ a \neq b \Rightarrow E \cap E' = \emptyset$$

Wenn eine Interaktion der Auslöser eines Übergangs im Szenario ist, muss folgendes gelten: Die Interaktion ist dem Knoten zugeordnet, aus dem die Kante des Übergangs ausgeht. Die Interaktion ist dem Knoten nicht zugeordnet, in dem die Kante eingeht (siehe Definition 30).

Definition 30.

$$\forall (A,E) \in I_{ACT}, (s,t) \in E : s \in A \wedge t \notin A$$

Wenn eine Kante eines Szenarios bei keiner Interaktion in der Menge E enthalten ist, müssen alle aktiven Interaktionen auch aktiv bleiben (siehe Definition 31).

Definition 31.

$$\forall (Nd^{SG}, Ed^{SG}, r, L) \in SG, (s,t) \in Ed^{SG} : \\ (\forall (A,E) \in I_{ACT} : (s,t) \notin E) \Rightarrow (\forall (A,E) \in I_{ACT} : s \in A \Rightarrow t \in A)$$

Interaktionen innerhalb eines Szenarios können mit dem System oder zwischen zwei externen Systemen bzw. Personen stattfinden. Die Menge der Interaktionen mit dem System I_{ACT}^{SYS} ist eine Teilmenge aller Interaktionen (siehe Definition 32).

Definition 32.

$$I_{ACT}^{SYS} \subseteq_{DEF} I_{ACT}$$

Jede Interaktion mit dem System ist mit einer Schnittstellenaktion verknüpft. Entsprechend werden diese Interaktionen auf einen Knoten des zustandsbasierten Modells vom Typ y_{ia} abgebildet (siehe Definition 33).

Definition 33.

$$iaNode : I_{ACT}^{SYS} \rightarrow ND^{ZG} \mid iaNode(i) = \begin{cases} (i,y) \in Nd^{ZG} \mid \\ (Nd^{ZG}, Ed^{ZG}) \in ZG \\ \quad \wedge y = y_{ia} \end{cases}$$

Im Szenario *sg2* ist beispielsweise die Interaktion *j* mit der Schnittstellenaktion A verknüpft und isNode(j)=A (siehe Abb. 5.9).

5.3 Abbildung auf die Graphen-Syntax

5.3.1 Abbildung des szenarienbasierten Modells

Das szenarienbasierte Modell des modellbasierten Ansatzes CREATE kann beispielsweise wie folgt auf die Graphensyntax abgebildet werden: Jedes Szenario wird durch einen Baum beschrieben. Das SD *Buch hinzufügen* kann z.B. zusammen mit dem ID auf den Baum *sg1* abgebildet werden (siehe Abb. 5.10). Die Szenarienschritte und Übergänge sind die Knoten und Kanten des Baums. Eine Interaktion des SD wird auf eine Interaktion des Baums abgebildet. Im vorgestellten modellbasierten Ansatz wird jede Interaktion des SD durch genau eine Interaktionsskizze des ID beschrieben und umgekehrt beschreibt jede Interaktionsskizze genau eine Interaktion des SD. Die Szenarienschritte mit den Interaktionen *1* und *2* des Szenarios *Buch hinzufügen* werden daher auf die Knoten *1* und *2* mit den Interaktionen *a* und *b* abgebildet (siehe Abb. 5.10). Für den Start und für das Ende des Szenarios werden zusätzliche Knoten eingefügt. Das zustandsbasierte Modell beschreibt im Allgemeinen nicht die Quelle und das

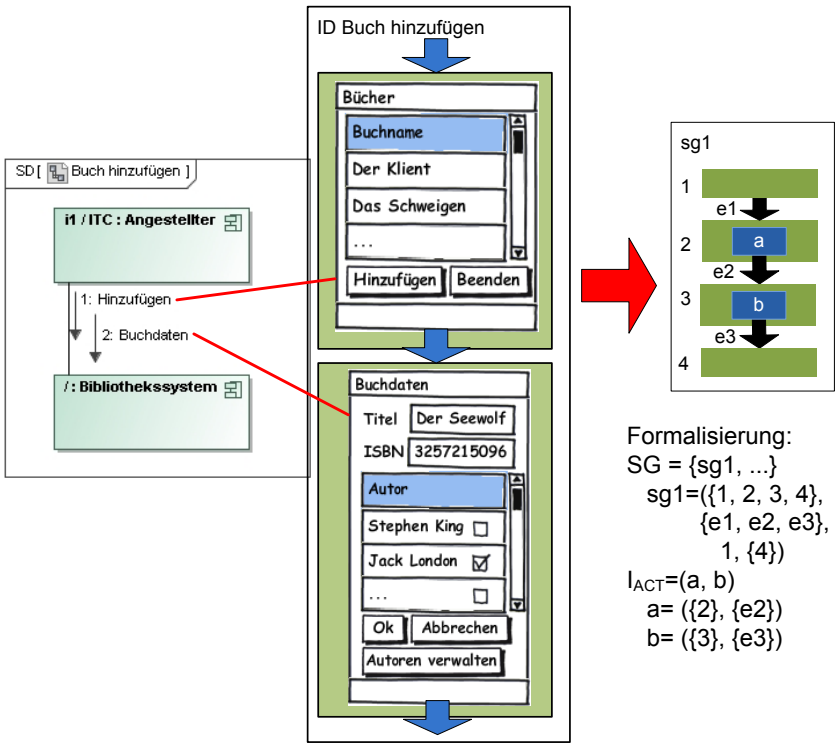


Abbildung 5.10: Abbildung des SD und ID

Ziel von Daten, die im Rahmen einer Interaktion ausgetauscht werden. Zur Konsistenzsicherung des szenarienbasierten und zustandsbasierten Modells müssen demnach die Instanzen des szenarienbasierten Modells nicht auf den Baum abgebildet werden. In

einem Szenario sind keine Schleifen erlaubt. Diese werden nicht benötigt, da durch eine Schleife ein Szenarienschritt erneut erreicht werden würde. Ausgehend von diesem Szenarienschritt kann jeder nachfolgende Szenarienschritt erreicht werden. Eine mögliche Zustandsänderung durch die Schleife wird an keiner Stelle des Szenarios sichtbar. Damit sind Schleifen im szenarienbasierten Modell irrelevant.

5.3.2 Abbildung des zustandsbasierten Modells

Das zustandsbasierte Modell des modellbasierten Ansatzes CREATE kann beispielsweise wie folgt auf die Graphensyntax abgebildet werden: Jeder Teil des zustandsbasierten Modells wie beispielsweise eine Aktivität wird durch ein Graph beschrieben. Die Kanten und Knoten des zustandsbasierten Modells (z.B. Kanten und Transitionen) werden zu den Kanten und Knoten des Graphs. Die Abbildung der Knotentypen des AVD wird durch die Tabelle 5.2 definiert. Ein zustandsbasiertes Modell ist ein ein-

AVD-Knoten	**Graph-Symbol**	**Knotentyp**
Aufrufaktion	y_{cba}	Aufrufknoten
Schnittstellenaktion	y_{ia}	Interaktionsknoten
Serviceaktion	y_{sa}	Verarbeitungsknoten
Initialer Knoten	y_{init}	Initialer Knoten
Aktivitätsendeknoten	y_{final}	Endeknoten
Entscheidungsknoten	y_{dec}	Entscheidungsknoten
Mischknoten	y_{merge}	Mischknoten
Parallelisierungsknoten	y_{fork}	Parallelisierungsknoten
Vereinigungsknoten	y_{join}	Vereinigungsknoten

Tabelle 5.2: Abbildung der Knotentypen

facher gerichteter Graph, bei dem nicht mehrere Kanten mit dem selben Quell- und Zielknoten zugelassen sind. Dies ist für das AVD keine Einschränkung der Ausdrucksmächtigkeit. Ein Sachverhalt, der durch mehrere Kanten von einem Knoten zum anderen ausgedrückt wird, kann stets in einer anderen Art und Weise modelliert werden. Beispielsweise können hierfür Schleifen verwendet werden.

Das Verhalten eines Aufrufknotens wird durch ein weiteres zustandsbasiertes Modell (z.B. einer aufgerufenen Aktivität) beschrieben. Diese Aufrufbeziehung wird bei der Graphensyntax über Aufrufkanten modelliert. Zudem werden für jedes Paar (Endeknoten, Aufrufkante) Rückkehrkanten eingeführt, wenn sich der initiale Knoten der Aufrufkante in dem selben Graph wie der Endeknoten befindet. Die AVDs *Bücher verwalten* und *Buch hinzufügen* können beispielsweise auf die Graphen *zg1* und *zg2* abgebildet werden (siehe Abb. 5.11). Die Abbildung der Knotentypen erfolgt hierbei nach der Vorgabe der Tabelle 5.2 und die Aufrufbeziehung der Aktion *:Buch hinzufügen* erfolgt über die Aufruf- und Rückkehrkanten. Die Variablen des zustandsbasierten Modells sowie die Beschriftungen der Kanten und Knoten (z.B. der Transitionen eines Petrinetzes) werden nicht auf die Graphensyntax abgebildet. Die Abbildung erfolgt nicht, weil eine Konsistenzüberprüfung auf der Graphensyntax durch die zusätzlichen Informationen nicht an Aussagekraft gewinnt. Das zustandsbasierte Modell

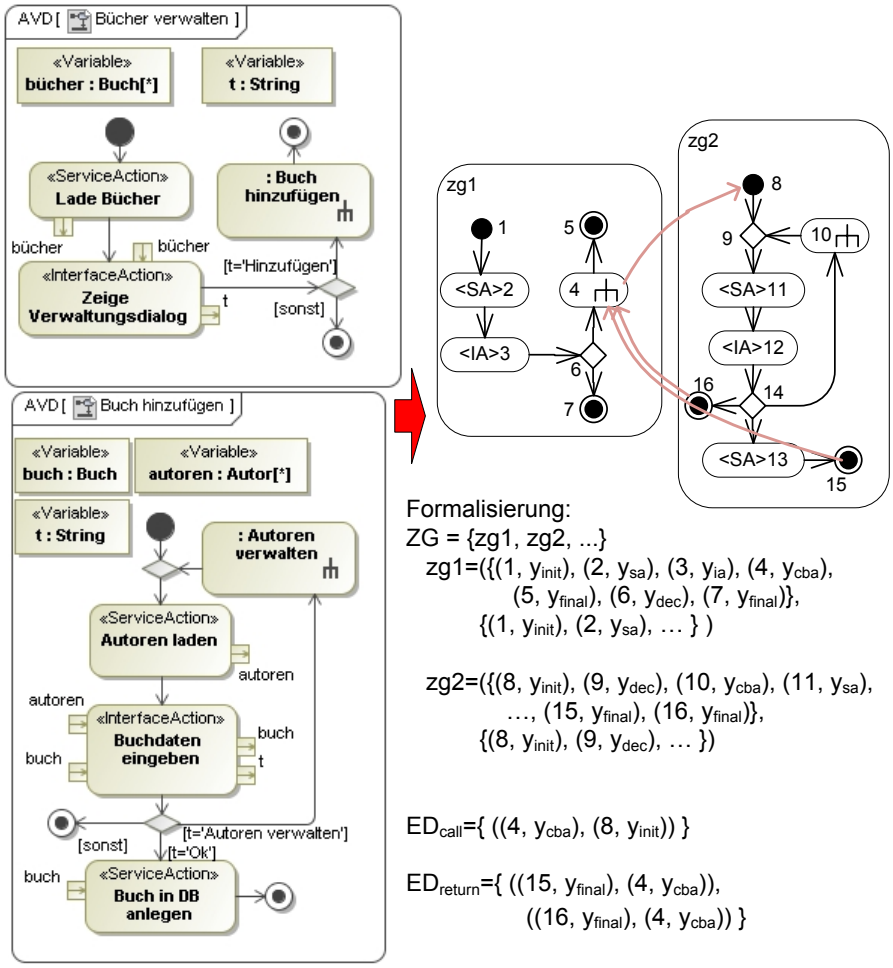


Abbildung 5.11: Abbildung des AVD

ist insbesondere in den frühen Phasen abstrakt. Die Knoten (z.B. Transitionen) sind nicht implementiert und deren Wirkung kann im Allgemeinen nicht berechnet werden. Als Ausgabe muss daher bei einer Konsistenzüberprüfung jeder mögliche Wert angenommen werden. Durch die Annahme jedes möglichen Wertes wird jede mögliche Ausführungssequenz des zustandsbasierten Modells durchsucht. Zudem haben die Variablenzustände keine Aussagekraft über die Konsistenz. Das Durchsuchen jeder möglichen Ausführungssequenz kann auch ohne die Abbildung der Variablen auf die Graphensyntax erfolgen. Wie in Kapitel 2.2.4 gezeigt, haben Aktivitätsdiagramme und damit auch das AVD mit den in Tabelle 5.2 angegebenen Knotentypen ohne Objektfluss die Mächtigkeit eines RPN. Ein RPN ist nicht Turing-vollständig. Das Problem der Erreichbarkeit einer Stelle in einem RPN ist entscheidbar [32]. Durch Schleifen und parallele Abläufe ist das Erreichbarkeitsproblem jedoch bereits in einfachen Petrinetzen

EXPSpace-vollständig [26]. Durch die Hierarchien mit Rekursion ist eine noch höhere Komplexität des Erreichbarkeitsproblems im RPN anzunehmen. Eine Verringerung dieser Komplexität wird in dieser Arbeit durch Syntaxeinschränkungen erreicht (siehe Kapitel 5.5). Diese Syntaxeinschränkungen verringern nicht die Ausdrucksmächtigkeit der Modelle. Beispielsweise sind Schleifen, parallele Abläufe und Hierarchien mit Rekursion weiterhin zugelassen. Vielmehr werden die Varianten zur Beschreibung eines Sachverhalts eingeschränkt.

5.4 Syntaxzuordnung

In den vorangegangenen Abschnitten wurde die abstrakte Syntax des allgemeinen szenarienbasierten und zustandsbasierten Modells eingeführt. Im Folgenden werden Regeln für den Zusammenhang der Modelle definiert.

5.4.1 Definition

Die Definition der Zusammenhänge des szenarienbasierten und des zustandsbasierten Modells erfolgt durch die Zuordnung von Kanten des zustandsbasierten Modells zu den Kanten des szenarienbasierten Modells. Die Menge aller möglicher Zuordnungen wird durch die Menge ED^{assign} beschrieben (siehe Definition 34).

Definition 34.

$$ED^{assign} =_{DEF} \begin{cases} (ed \in ED^{SG}, E \subseteq ED^{ZG}, s \in E, F \subseteq E) \mid \\ \exists (Nd^{SG}, Ed^{SG}, r, L) \in SG : ed \in Ed^{SG}, \\ \forall e \in E : e \in ED_{call} \cup ED_{return} \vee \\ \quad \exists (Nd^{ZG}, Ed^{ZG}) \in ZG : e \in Ed^{ZG} \end{cases}$$

hierbei ist

- *ed* eine Kante eines Szenarios,
- E die Menge der zugeordneten Kanten des zustandsbasierten Modells,
- s eine Startkante der Zuordnung und
- F eine Menge der Endkanten der Zuordnung.

Für das Szenario *sg* sind beispielsweise die Zuordnungen zu den Kanten *e1* und *e2* angegeben (siehe Abb. 5.12). Der Kante *e1* ist beispielsweise die Kante *a* des zustandsbasierten Modells *zg* zugeordnet. Die Kante *a* ist auch gleichzeitig die Startkante und eine Endkante der Zuordnung.

Allgemein existiert für jede Kante eines Szenarios maximal eine Zuordnung (siehe Definition 35).

Definition 35.

$$\forall a = (ed, E, s, F), b = (ed', E', s', F') \in ED^{assign} :$$
$$ed = ed' \Rightarrow a = b$$

Die Kanten der Zuordnung müssen zusammenhängend und jeweils von der Startkante *s* aus erreichbar sein (siehe Definition 36).

Definition 36.

$$\forall (ed, E, (s,t), F) \in ED^{assign}, (s', t') \in E :$$
$$\exists! p \in pathsFrom((s,t), E) : get(p, len(p)) = t'$$

5.4.2 Zuordnungsvorschriften

Mit Hilfe von Zuordnungen können Regeln für den Zusammenhang szenarienbasierter und zustandsbasierter Modelle definiert werden. Die Zuordnung von Kanten erfolgt nach genau festgelegten Vorschriften.

Startkante

Bei einem Übergang von einem Szenarienschritt zum nächsten können Interaktionen mit dem zu entwickelnden System aktiviert und beendet werden. Neue Interaktionen mit dem System können aktiviert werden, nachdem eine Funktion des Systems gestartet wurde oder eine vorherige Interaktion beendet wurde. Das Starten und Beenden von Interaktionen im Architekturmodell erfolgt durch den Kontrollfluss im zustandsbasierten Modell. Beim Start einer Funktion beginnt der Kontrollfluss bei einem initialen Knoten des zustandsbasierten Modells. Beim Beenden einer Interaktion wird der Kontrollfluss bei einem Interaktionsknoten fortgesetzt.

Dieser Zusammenhang des szenarienbasierten und zustandsbasierten Modells wird durch die folgende Zuordnungsvorschrift definiert: Jeder Übergang von einem Szenarienschritt zum nächsten eines Szenarios muss eine auslösende Interaktion zugeordnet sein, genau dann wenn eine Zuordnung existert, die vom verknüpften Knoten der Interaktion vom Typ y_{ia} ausgeht (siehe Definition 37).

Definition 37.

$$\forall (Nd^{SG}, Ed^{SG}, r, L) \in SG, esg \in Ed^{SG} : i = (A, E) \in I_{ACT}^{Sys} :$$
$$esg \in E \Leftrightarrow \exists (ed, E', (s,t), F) \in ED^{assign} : ed = esg \wedge iaNode(i) = s$$

Im Szenario *sg* ist beispielsweise die Interaktion *i* der Auslöser des Übergangs *e2* (siehe Abb. 5.12). Damit muss es eine Zuordnung geben, deren Startkante vom Knoten *iaNode(i)=x* ausgeht. Umgekehrt existiert eine Zuordnung mit einer Startkante ausgehend vom Knoten *iaNode(i)=x*. Damit muss die Interaktion *i* der Auslöser des Übergangs *e2* sein.

Im Falle des Starts einer Funktion des Systems gilt Folgendes: Die Startkante einer Zuordnung geht von einem Knoten vom Typ y_{init} genau dann aus, wenn eine neue

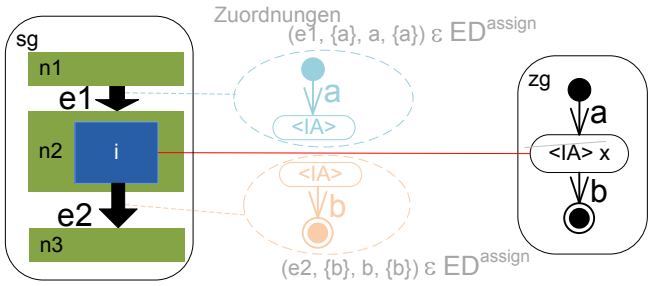


Abbildung 5.12: Startkanten einer Zuordnung

Systeminteraktion im folgenden Szenarienschritt aktiv ist und keine Systeminteraktion der Auslöser des Übergangs ist (siehe Definition 38).

Definition 38.

$$\begin{aligned}&\forall(Nd^{SG}, Ed^{SG}, r, L) \in SG, esg = (s,t) \in Ed^{SG}: \\ &\quad \nexists(A,E) \in I^{Sys}_{ACT}: esg \in E \wedge (\exists(A,E) \in I^{Sys}_{ACT}: (s \notin A \wedge t \in A)) \\ &\quad \Leftrightarrow \exists(ed, E, ((i,y),t'), F) \in ED^{assign}: ed = esg \wedge y = y_{init}\end{aligned}$$

Im Szenario *sg* ist beispielsweise die Interaktion *i* im Knoten *n2* erstmals aktiv (siehe Abb. 5.12) und keine Interaktion ist Auslöser des Übergangs zum Szenarienschritt *n2*. Damit existiert für *e1* eine Zuordnung mit einer Startkante vom Typ y_{init}.

Einfache Fortsetzung

Der Kontrollfluss durchläuft im Rahmen eines Szenarios das zustandsbasierten Modell nach genau festgelegten Regeln ausgehend von der Startkante einer Zuordnung. Der Zusammenhang des szenarienbasierten und des zustandsbasierten Modells wird so definiert, dass die vom Kontrollfluss durchlaufenen Kanten ebenfalls zugeordnet werden. Wenn sich eine Kante zu den Knotentypen $y_{init}, y_{merge}, y_{fork}$ oder y_{sa} in der Zuordnung befindet, müssen alle weiterführenden Kanten auch zugeordnet sein (siehe Definition 39).

Definition 39.

$$\begin{aligned}&\forall(ed, E, b, F) \in ED^{assign}, (s, t = (i,y)) \in E, (Nd^{ZG}, Ed^{ZG}) \in ZG, \\ &(s',t') \in Ed^{ZG} \cup ED^{call} \cup ED^{return}: \\ &\quad y \in \{y_{init}, y_{merge}, y_{fork}, y_{sa}\} \wedge t = s' \Rightarrow (s',t') \in E\end{aligned}$$

Befindet sich eine Kante zu einem Knoten vom Typ y_{dec} in einer Zuordnung, muss mindestens eine weiterführende Kante zugeordnet sein (siehe Definition 40).

Definition 40.

$$\forall(ed,E,b,F) \in ED^{assign}, (s,t=(i,y)) \in E : y = y_{dec}$$
$$\Rightarrow \exists(s',t') \in E : s' = t$$

Die Zuordnungen zum Szenario *sg* zeigen die Zuordnungsvorschriften am Beispiel (siehe Abb.5.13). Da die Kante *a* des zustandsbasierten Modells *zg* der Kante *e1* von *sg* zugeordnet ist, folgt die Zuordnung der Kante *b* oder *c*. Bei Zuordnung der Kante *b* folgt die Zuordnung der Kante *d* und *e*. Da die Kante *f* des zustandsbasierten Modells

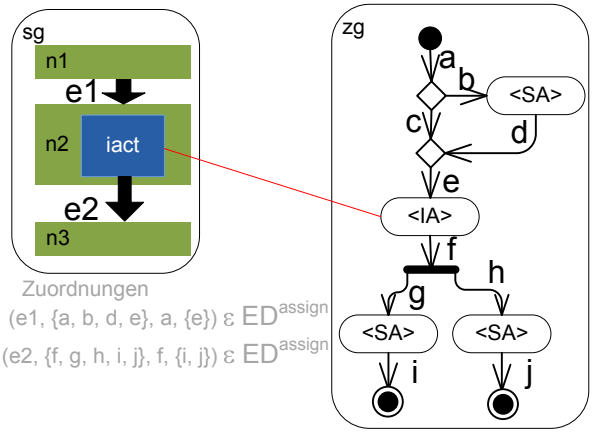


Abbildung 5.13: Weiterführende Zuordnung der Kanten

zg der Kante *e2* von *sg* zugeordnet ist, folgt die Zuordnung der Kanten *g* und *h* sowie *i* und *j*.

Endkanten

Wird im Rahmen eines Szenarios vom Kontrollfluss ein Knoten vom Typ y_{ia} erreicht, muss die über die Abbildung *iaNode* verknüpfte Interaktion Teil des nächsten Szenarienschritts sein. Der Kontrollfluss kann erst im Übergang zu einem folgenden Szenarienschritt fortgesetzt werden. Dieser Zusammenhang des szenarienbasierten und zustandsbasierten Modells wird durch eine Zuordnungsvorschrift definiert. Innerhalb einer Zuordnung darf mit Ausnahme der Startkante keine Kante aus einem Interaktionsknoten führen (siehe Definition 41).

Definition 41.

$$\forall(ed,E,b,F) \in ED^{assign}, ((i,y),t) \in E \setminus \{b\} : y \neq y_{ia}$$

In Abbildung 5.13 ist beispielsweise die Kante *f* nicht *e1* zugeordnet. Dies würde der beschriebenen Bedingung widersprechen. Der Kontrollfluss kann innerhalb einer Zuordnung bei mehreren Knotentypen wie beispielsweise Knoten vom Typ y_{final} enden.

Die Endkanten einer Zuordnung sind Kanten, die zu einem Knoten vom Typ y_{ia} führen und Kanten, denen keine weiterführenden Kanten zugeordnet sind (siehe Definition 42).

Definition 42.

$$\forall(ed,E,b,F) \in ED^{assign} : F = \{ed = (s,t = (i,y)) \in E \mid \\ y = y_{ia} \vee \nexists ed' = (s',t') \in E : ed \neq ed' \wedge s' = t\}$$

Fortsetzung in einer anderen Zuordnung

Jede Endkante einer Zuordnung steht für einen aktiven Kontrollfluss im Szenario. Ein Kontrollfluss startet beispielsweise eine Interaktion, wenn die Endkante zu einem Knoten vom Typ y_{ia} führt. Jeder Kontrollfluss kann nur genau einmal fortgesetzt werden. Für den Zusammenhang des szenarienbasierten und zustandsbasierten Modells bedeutet dies, dass maximal eine Zuordnung existieren darf innerhalb derer eine Endkante einer Zuordnung fortgesetzt wird.

Die Menge aller möglicher konkreter Fortsetzungsbeziehungen wird durch die Menge ED^{con} beschrieben (siehe Definition 43).

Definition 43.

$$ED^{con} =_{DEF} \begin{cases} (sA,sE,tA,tE) \mid \\ sA = (ed,E,b,F) \in ED^{assign}, \\ sE \in F, \\ tA = (ed',E',b',F') \in ED^{assign}, \\ tE \in E' \end{cases}$$

hierbei ist

- *sA* die Zuordnung der fortgesetzten Endkante,
- *sE* die fortgesetzte Endkante,
- *tA* die Zuordnung der fortsetzenden Kante und
- *tE* die fortsetzende Kante.

Der Kante *e1* des Szenarios *sg* sind beispielsweise die Kanten *a*, *b* und *c* über *x* zugeordnet und *e2* sind die Kanten *d* und *e* über y zugeordnet (siehe Abb. 5.14). Innerhalb der Zuordnung *x* sind *b* und *c* Endkanten. Diese Endkanten werden innerhalb von *y* fortgesetzt. Die Fortsetzungsbeziehungen werden durch die Relationen (x, b, y, d) und (x, c, y, e) beschrieben. Die Kanten *b* und *c* aus *x* werden hierbei durch die kanten *d* und *e* aus *y* fortgesetzt (siehe Abb. 5.14).

Jede Endkante kann nur genau einmal fortgesetzt werden (siehe Definition 44).

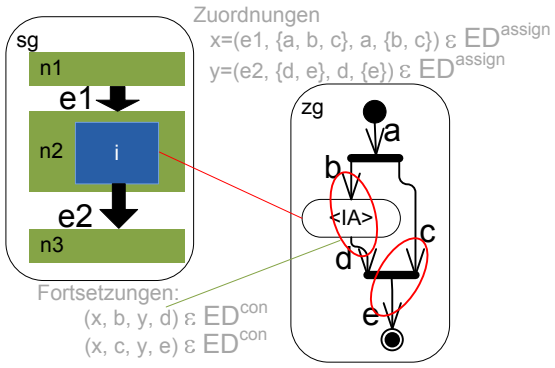


Abbildung 5.14: Fortsetzungsbeziehung zwischen Kanten

Definition 44.

$$\forall a = (sA, sE, tA, tE), b = (sA', sE', tA', tE') \in ED^{con} :$$
$$sA = sA' \wedge sE = sE' \Rightarrow a = b$$

Es darf beispielsweise keine weitere Fortsetzungsbeziehung zu der Kante b der Zuordnung x geben neben der Fortsetzungsbeziehung (x, b, y, d) (siehe Abb. 5.14). Zudem darf eine Kante innerhalb einer Zuordnung nur einmal eine Fortsetzung sein (siehe Definition 45).

Definition 45.

$$\forall a = (sA, sE, tA, tE), b = (sA', sE', tA', tE') \in ED^{con} :$$
$$tA = tA' \wedge tE = tE' \Rightarrow a = b$$

Der Kontrollfluss kann nur innerhalb einer anderen Zuordnung fortgesetzt werden, wenn dieser an einem Knoten vom Typ y_{ia} oder y_{join} gehalten hat (siehe Definition 46).

Definition 46.

$$\forall (sA, ((i, y), t), tA, tE) \in ED^{con} : y \in \{y_{ia}, y_{join}\}$$

Der Kontrollfluss muss am selben Knoten fortgesetzt werden (siehe Definition 47).

Definition 47.

$$\forall (sA, (s, t), tA, (s', t')) \in ED^{con} : s' = t$$

Innerhalb der Zuordnung y wird beispielsweise die Kanten b durch d fortgesetzt (siehe Abb. 5.14). Die Kante d hat als Quelle den Zielknoten von b.

Um einen Kontrollfluss fortsetzen zu können, muss dieser zuvor im Szenario gestartet wurden sein und halten. Die über eine Fortsetzungsbeziehung verbundenen Kanten

des zustandsbasierten Modells müssen daher Kanten des szenarienbasierten Modells zugeordnet sein, die über einen Pfad miteinander verbunden sind (siehe Definition 48).

Definition 48.

$$\forall(((s,t),E,b,F),sE,((s',t'),E',b',F'),tE) \in ED^{con} :$$
$$\exists(Nd^{SG},Ed^{SG},r,L) \in SG : ed \in Ed^{SG} \wedge$$
$$\exists p \in pathsFrom((s),Ed^{SG}) : get(p,len(p)) = t'$$

Die Kante b der Zuordnung x wird beispielweise durch die Kante d der Zuordnung y fortgesetzt (siehe Abb. 5.14). Die Kanten *e1* und *e2* der Zuordnungen x und y müssen daher über einen Pfad verbunden sein.

Eine Interaktion muss durch einen Kontrollfluss innerhalb einer vorangegangenen Zuordnung gestartet werden. Wenn eine Zuordnung von einem Knoten vom Typ y_{ia} startet, muss hierfür eine Fortsetzungsbeziehung bestehen (siehe Definition 49).

Definition 49.

$$\forall eda = (ed,E,x = ((i,y),t),F) \in ED^{assign} : y = y_{ia} \Rightarrow$$
$$\exists(sA,sE,tA,tE) \in ED^{con} : tA = eda \wedge x = tE$$

Der Kontrollfluss wird an einem Knoten vom Typ y_{join} erst fortgesetzt, wenn an allen eingehenden Kanten ein Kontrollfluss hält. Folglich wird eine ausgehende Kante aus einem Knoten vom Typ y_{join} nur zugeordnet, wenn jede eingehende Kante entweder zugeordnet ist oder es eine Fortsetzungbeziehung zu einer anderen Zuordnung existiert (siehe Definition 50).

Definition 50.

$$\forall eda = (ed,E,b,F) \in ED^{assign}, e = (u = (i,y),v) \in E : y = y_{join} \Rightarrow$$
$$\forall(Nd^{ZG},ED^{ZG}) \in ZG, e' = (u' = (i',y'),v') \in ED^{ZG} : v' = u \Rightarrow$$
$$e' \in E \vee \exists(sA,sE,tA,tE) \in ED^{con} : sE = e' \wedge tA = eda \wedge tE = e$$

Die Kante e ist beispielsweise *e1* über y zugeordnet (siehe Abb. 5.14). Die Kante e geht von einem Knoten vom Typ y_{join} aus. In diesen Knoten gehen die Kanten d und c ein. Die Kante d befindet sich in der selben Zuordnung. Die Kante c der Zuordnung x hat eine Fortsetzungsbeziehung zu der Kante e der Zuordnung y. Für jede eingehende Kante ist damit die oben genannte Bedingung erfüllt.

5.4.3 Vorschriften für Hierarchien

Im vorangegangenen Abschnitt wurde der Zusammenhang des szenarienbasierten und des zustandsbasierten Modells mit Hilfe von Zuordnungen und Zuordnungsvorschriften definiert. Diese legen eindeutige Regeln für die Zuordnung von Kanten des zustandsbasierten Modells zu den Kanten des szenarienbasierten Modells fest. Beispielsweise muss sich jede aus einem Parallelisierungsknoten ausgehende Kante in einer Zuordnung befinden, wenn die Zuordnung die Kante zum Parallelisierungsknoten bein-

haltet. In diesem Abschnitt werden die Zuordnungsvorschriften für Hierarchien mit Aufrufknoten und finalen Knoten definiert.

Einfacher Aufruf

Erreicht der Kontrollfluss im Szenario einen Aufrufknoten, wird der Kontrollfluss am initialen Knoten des aufgerufenen zustandsbasierten Modells fortgesetzt. Dieser Zusammenhang des szenarienbasierten und zustandsbasierten Modells wird ebenfalls durch eine Zuordnungsvorschrift definiert. Wenn sich in einer Zuordnung eine Kante zu einem Aufrufknoten befindet, muss die Aufrufkante des Aufrufknotens zugeordnet werden (siehe Definition 51).

Definition 51.

$$\forall (ed, E, b, F) \in ED^{assign}, e = (s, t = (i, y)) \in E,$$
$$(s', t') \in ED^{call} : y = y_{cba} \wedge e \notin ED^{return} \wedge t = s' \Rightarrow (s', t') \in E$$

Beim Szenario *sg* befindet sich beispielsweise die Kante *b* zum Aufrufknoten *1* in der Zuordnung *y* (siehe Abb. 5.15). Folglich muss auch die Aufrufkante *c* zugeordnet wer-

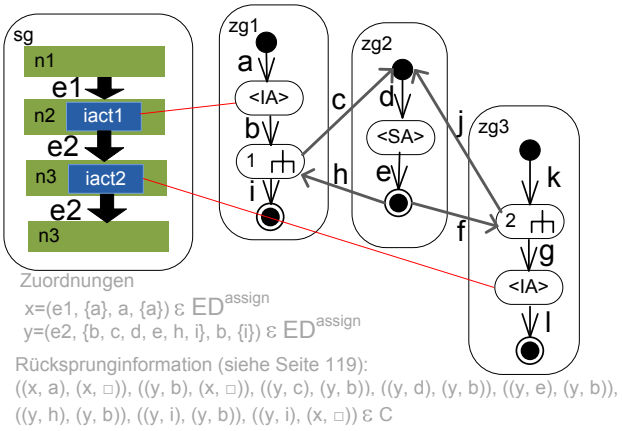


Abbildung 5.15: Zuordnung mit Hierarchien

den, die vom Knoten *1* ausgeht. Entsprechend der einfachen Zuordnungsvorschriften ohne Hierarchien muss daraufhin auch die Kante *d* zugeordnet werden.

Korrekte Rückkehr in der Aufrufhierarchie

Erreicht der Kontrollfluss im Rahmen eines Szenarios einen finalen Knoten, wird entweder der Kontrollfluss beendet oder an einem Aufrufknoten fortgesetzt. Der Kontrollfluss muss bei dem Aufrufknoten fortgesetzt werden, der das beendete zustandsbasierte Modell aufgerufen hat. Im Szenario *sg* wird beispielsweise das zustandsbasierte Modell *z2* vom Aufrufknoten *1* aufgerufen (siehe Abb. 5.15). Der Kontrollfluss

muss folglich nach Abschluss von *z2* am Aufrufknoten *1* fortgesetzt werden. Zuordnungen sind an dieser Stelle wie folgt definiert: Wenn sich eine Kante zu einem finalen Knoten in einer Zuordnung befindet, muss diese eine Endkante der Zuordnung sein oder es muss sich die Rückkehrkante zum korrekten Aufrufknoten in der Zuordnung befinden. Die Zuordnung *y* beinhaltet beispielsweise die Kante *e* (siehe Abb. 5.15). Eine Zuordnungsvorschrift muss nun definieren, ob eine Rückkehrkante und welche Rückkehrkante zugeordnet sein muss. Diese Information kann nicht aus dem Szenario abgeleitet werden, da dieses inkonsistent zum zustandsbasierten Modell sein kann. Die Interaktionsknoten, die die Interaktionen *iact1* und *iact2* realisieren, können beispielsweise im zustandsbasierten Modell bei Einhaltung der korrekten Aufrufhierarchie nicht nacheinander vom Kontrollfluss erreicht werden. Eine Zuordnungsvorschrift muss die Aufrufhierarchie mit der korrekten Anzahl und Reihenfolge der Aufrufe berücksichtigen. *A* und *B* stellen beispielsweise zustandsbasierte Modelle dar, die sich gegenseitig und verschränkt aufrufen können (siehe Abb. 5.16). Eine mögliche Auf-

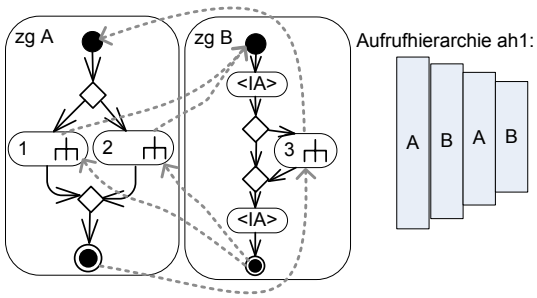


Abbildung 5.16: Aufrufhierarhie mit Rekursion

rufhierarchie ist beispielsweise die Hierarchie *ah1* (siehe Abb. 5.16). Die Aktivität *B* kann sowohl vom Aufrufknoten *1* als auch vom Aufrufknoten *2* aufgerufen wurden sein. Wie zuvor beschrieben, ist es entscheidend, bei der Zuordnung der Rückkehrkante die Aufrufhierarchie einzuhalten. Eine Zuordnungsvorschrift muss daher festlegen, dass die korrekte Anzahl und die Reihenfolge der Aufrufe beim zuordnen der Rückkehrkante eingehalten wird. Auch bei der Einschränkung von Schleifen innerhalb von Zuordnungen sind Schleifen über mehrere Zuordnungen hinweg weiterhin zugelassen (siehe Abb. 5.4 links). Dies gilt auch für Rekursionsschleifen. Mögliche mehrfache und verschränkt rekursive Aufrufe wie beispielsweise in den zustandsbasierten Modellen *A* und *B* mit den möglichen Aufrufknoten *1* und *2* müssen bei der Definition der Zuordnungsvorschrift für Rückkehrkanten berücksichtigt werden.

Für die Definition von Zuordnungen unter Verwendung von Hierarchien werden im Folgenden Rückkehrinformationen definiert, die den Kanten innerhalb einer Zuordnung hinzugefügt werden. Diese Rückkehrinformation wird ausgehend von der Aufrufkante an die nachfolgenden Kanten weitergereicht. Mit Hilfe dieser Information kann eine allgemeine Zuordnungsvorschrift definiert werden, die festlegt ob und welche Rückkehrkante zugeordnet sein muss. Beispielsweise geht die Aufrufkante *c* innerhalb der Zuordnung *y* vom Aufrufknoten *1* aus (siehe Abb. 5.15). Diese Information

wird an die Kanten d und e weitergereicht. Mit Hilfe der Zuordnungsvorschrift wird definiert, dass die Kante h die Rückkehrkante zum korrekten Aufrufknoten ist. Eine Rückkehrinformation ist eine Kante zu einem Aufrufknoten innerhalb einer Zuordnung. Die Menge aller möglichen Selektionen einer Kante aus einer Zuordnung wird durch die Menge ED^{sel} beschrieben (siehe Definition 52).

Definition 52.

$$ED^{sel} =_{DEF} \begin{cases} (sA \in ED^{assign}, sE \in ED^{ZG} \cup ED^{call} \cup ED^{return} \cup \square) \mid \\ sA = (ed, E, b, F) \wedge (sE \in E \vee sE = \square) \end{cases}$$

hierbei ist

- sA eine Zuordnung und
- sE die selektierte Kante aus der Zuordnung.

Die Menge aller möglicher Rückkehrinformationen wird durch die Menge C beschrieben (siehe Definition 53).

Definition 53.

$$C =_{DEF} \begin{cases} (eds, c) \mid \\ eds \in ED^{sel}, c \in ED^{sel} \end{cases}$$

hierbei ist

- eds die selektierte Kante und
- c die Rückkehrinformation der Kante.

Wenn eine Aufrufkante in einer Zuordnung enthalten ist, hat diese die Kante zum Aufrufknoten als Rückkehrinformation (siehe Definition 54).

Definition 54.

$$\begin{aligned} &\forall eda = (ed, E, b, F) \in ED^{assign}, (s,t) \in E \cap ED^{call} : \\ &\quad \exists (eds, (cA, (s', t'))) \in C : cA = eda \wedge t' = s \wedge \\ &\quad eds = (eda, (s,t)) \end{aligned}$$

Die Aufrufkante c ist beispielsweise in der Zuordnung y enthalten (siehe Abb. 5.15). Damit befindet sich *((y, c), (y, b))* in C.

Wenn die Startkante einer Zuordnung von einem initialen Knoten ausgeht, erhält diese ebenfalls eine Rückkehrinformation. Die Rückkehrinformation ist hierbei die Selektion des leeren Elements aus der aktuellen Zuordnung (siehe Definition 55).

Definition 55.

$$\begin{aligned} &\forall eda = (ed, E, b = ((i,y),t), F) \in ED^{assign} : \\ &\quad y = y_{init} \Rightarrow ((eda, b), (eda, \square)) \in C \end{aligned}$$

Die Startkante a der Zuordnung x geht beispielsweise von einem initialen Knoten aus. Damit ist die Rückkehrinformation $((x,a),(x,\square))$ in C enthalten.

Die Rückkehrinformation wird jeweils zu der nächsten Kante weitergereicht. Wenn eine Kante zu einem Knoten vom Typ y_{init}, y_{fork}, y_{join}, y_{merge}, y_{dec}, y_{final} oder y_{sa} führt, hat die nachfolgende Kante die selbe Rückkehrinformation (siehe Definition 56).

Definition 56.

$$\begin{aligned}&\forall((eda=(ed,E,b,F),(s,t=(i,y))),c)\in C\\&\quad(s',t')\in E : y\in\{y_{init},y_{fork},y_{join},y_{merge},y_{dec},y_{ia},y_{final},y_{sa}\}\\&\quad\wedge s'=t\Rightarrow((eda,(s',t')),c)\in C\end{aligned}$$

Die Rückkehrinformation *((y, c), (y, b))* ist beispielsweise in C enthalten und die Kante d befindet sich ebenfalls der Zuordnung y (siehe Abb. 5.15). Folglich muss auch *((y, d), (y, b))* in C enthalten sein.

Wenn eine Kante zu einem finalen Knoten zugeordnet ist, muss sich ebenfalls die Rückkehrkante zum korrekten Aufrufknoten in der Zuordnung befinden (siehe Definition 57).

Definition 57.

$$\begin{aligned}&\forall eda=(ed,E,b,F)\in ED^{assign},e=(s,t=(i,y))\in E,\\&\quad(eds,(cA,er=(sC,tC))\in C : y=y_{final}\wedge eds=(eda,e)\wedge er\neq\square\\&\quad\Rightarrow\exists(s',t')\in E : s'=t\wedge t'=tC\end{aligned}$$

Die Kante e befindet sich beispielsweise in der Zuordnung y und führt zu einem finalen Knoten (siehe Abb. 5.15). Zudem befindet sich *((y, e), (y, b))* in C. Damit muss sich in y auch die Kante h vom finalen Knoten zum Aufrufknoten der Kante b befinden.

Wenn eine Rückkehrkante zugeordnet ist, muss auch eine Kante aus dem Aufrufknoten zugeordnet sein (siehe Definition 58).

Definition 58.

$$\begin{aligned}&\forall eda=(ed,E,b,F)\in ED^{assign},(s,t)\in E\cap ED^{return}:\\&\quad\exists(s',t')\in E : s'=t\end{aligned}$$

Da sich beispielsweise die Kante h in der Zuordnung y befindet, ist auch die Kante i zugeordnet (siehe Abb. 5.15).

Die Rückkehrinformation wird auch weitergereicht, wenn sich Kanten in verschiedenen Zuordnungen befinden und über eine Fortsetzungsbeziehung miteinander verbunden sind (siehe Definition 59).

Definition 59.

$$\begin{aligned}&\forall((a,e),c)\in C,(sA,sE,tA,tE)\in ED^{con}:\\&sA=a\wedge sE=e\Rightarrow\exists((a',e'),c)\in C : tA=a'\wedge tE\in e'\end{aligned}$$

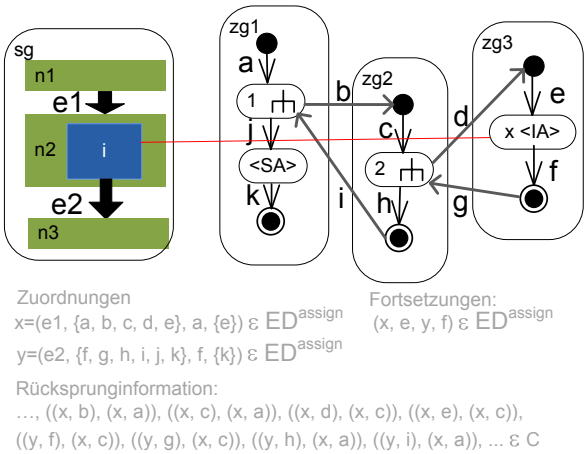


Abbildung 5.17: Zuordnung über Hierarchiegrenzen mit Rekursion

Die Kante e der Zuordnung x wird beispielsweise mit der Kante f der Zuordnung y fortgesetzt (siehe Abb. 5.17). Zudem befindet sich *((x, e), (x, a))* in C. Folglich muss auch *((y, f), (x, a))* in C enthalten sein. Die Rückkehrinformation ist immer die Kante zum Aufrufknoten. Diese Kante kann wiederum rekursiv mit einer übergeordneten Rückkehrinformation in Beziehung stehen. Dies ist der Fall, wenn der Kontrollfluss ebenfalls durch einen Aufruf erzeugt wurde. Der Kontrollfluss wird nach Beendigung eines Aufrufs am Aufrufknoten fortgesetzt. Wird hiernach ein finaler Knoten erreicht, muss der Kontrollfluss an der übergeordneten Rückkehradresse fortgesetzt werden. Die aus einem Aufrufknoten ausgehende Kante muss daher die übergeordnete Rückkehrinformation erhalten und an die nachfolgenden Kanten weiterreichen (siehe Definition 60).

Definition 60.

$$\begin{aligned}
&\forall eda = (ed, E, b, F) \in ED^{assign}, e = (s = (i, y), t) \in E \setminus ED^{call}, \\
&\quad (s', t') \in E \cap ED^{return}, (eds, c), (eds', c') \in C : \\
&\quad y = y_{cba} \wedge s = t' \wedge eds = (eda, (s', t')) \wedge eds' = c \\
&\quad \Rightarrow \exists (eds'', c'') \in C : eds'' = ((s, t), eda) \wedge c'' = c'
\end{aligned}$$

Die Kante h ist beispielsweise in der Zuordnung y enthalten und *((y, g), (x, c))* in C (siehe Abb. 5.17). Zudem existiert die übergeordnete Rückkehrinformation ((x, c), (x, a)) in C. Damit ist auch die Rückkehrinformation ((y, h), (x, a)) in C enthalten.

5.4.4 Korrektes Kontrollflussende

Wenn im Szenario ein finaler Knoten vom Kontrollfluss erreicht wird, wird die Aufrufinstanz des zustandsbasierten Modells und alle untergeordneten Aufrufinstanzen beendet. Kontrollflüsse dieser Aufrufinstanzen können folglich nicht weiter fortgesetzt

werden. Innerhalb der Zuordnung y wird durch die Kante f beispielsweise ein finaler Knoten erreicht (siehe Abb. 5.18). Der Kontrollfluss an der Kante e kann in den folgenden Szenarienschritten nicht weiter fortgesetzt werden. Dieser Zusammenhang

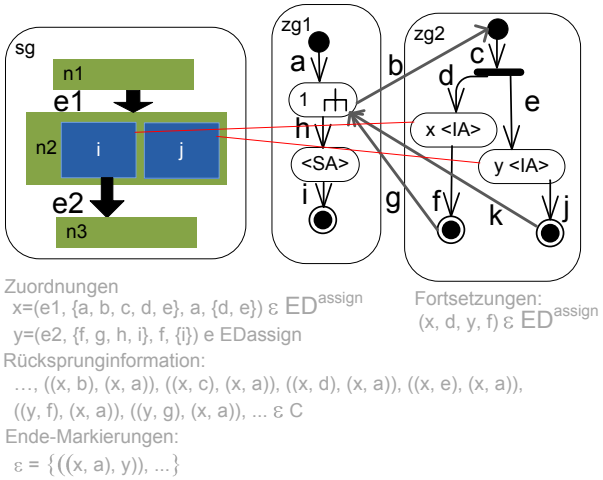


Abbildung 5.18: Hierarchien und Kontrollflussenden

des szenarienbasierten und zustandsbasierten Modells wird im Folgenden durch die Markierung von Rückkehradressen definiert. Innerhalb der Zuordnungen ist die Aufrufhierarchie über Rückkehradressen beschrieben. Das Beenden einer Aufrufinstanz ist daher durch die Markierung der Rückkehradresse innerhalb einer Zuordnung definiert. Diese Markierung wird durch die Relation ε beschrieben (siehe Definition 61).

Definition 61.

$$\varepsilon =_{DEF} \left\{ (c, fA) \mid c \in C, fA \in ED^{assign} \right.$$

Eine Rückkehradresse wird innerhalb einer Zuordnung genau dann als als beendet markiert, wenn sich in einer Zuordnung eine Kante zu einen finalen Knoten mit dieser Rückkehradresse befindet (siehe Definition 62).

Definition 62.

$$\forall eda = (ed, E, b, F) \in ED^{assign}, e = (s, t = (i, y)) \in E, ((a, e'), c) \in C :$$
$$(y = y_{final} \wedge e' = e \wedge a = eda) \Rightarrow$$
$$\exists (c', eda') \in \varepsilon : c' = c \wedge eda' = eda$$

Die Rückkehrkante g befindet sich beispielsweise in der Zuordnung y und hat die Rückkehrinformation *(x, a)* (siehe Abb. 5.15). Die Rückkehradresse wird folglich durch $\varepsilon = \{((x, a), y)\}$ als beendet markiert.

Wurde eine Rückkehradresse im Szenario in einem Szenarienschritt als beendet markiert, muss die Rückkehradresse auch in den darauf folgenden Szenarienschritten

als beendet markiert sein. Zu den folgenden Szenarienschritten existiert ein Pfad im Baum. Die Rückkehradresse wird damit für jede Zuordnung als beendet markiert, deren Szenarienkante auf einem Pfad ausgehend von der Kante der beendeten Zuordnung liegt (siehe Definition 63).

Definition 63.

$$\forall eda = ((s,t),E,b,F) \in ED^{assign}, (c,((s',t'),E',b',F')) \in \varepsilon,$$
$$(Nd^{SG},Ed^{SG},r,L) \in SG : (s',t') \in Ed^{SG} \wedge$$
$$\exists p \in pathsFrom((s',t'),Ed^{SG}) : get(p,len(p)) = t \Rightarrow \exists (c,eda) \in \varepsilon$$

Eine Rückkehradresse darf nur als beendet markiert sein, wenn diese im Szenario zuvor als beendet markiert wurde oder sich eine Kante zum finalen Knoten in der angegebenen Zuordnung befindet (siehe Definition 64).

Definition 64.

$$\forall (c,eda = (ed = (s,t),E,b,F)) \in \varepsilon :$$
$$\exists eda' = (ed' = (s',t'),E',b',F') \in ED^{assign}, (c',eda'') \in \varepsilon : eda'' = eda' \wedge t' = s$$
$$\vee \exists e = (s',(i,y)) \in F, ((a,e'),c'') \in C : a = eda \wedge e' = e \wedge y = y_{final} \wedge c'' = c$$

Endkanten dürfen nur in einer Zuordnung fortgesetzt werden, wenn die Rückkehradresse nicht als beendet markiert ist oder die Markierung in der selben Zuordnung erfolgt (siehe Definition 65).

Definition 65.

$$\forall (sA,sE,tA = (ed,E,b,F),tE) \in ED^{con}, (eds,c) \in C :$$
$$eds = (sA,sE) \wedge \exists (c',fA) \in \varepsilon : c = c' \Rightarrow$$
$$\exists (s,t = (i,y)) \in E : y = y_{final} \wedge \exists (eds',c'') \in C : eds` = (tA,e) \wedge c'' = c$$

Die Kante d der Zuordnung x wird beispielsweise durch die Kante f der Zuordnung y fortgesetzt (siehe Abb. 5.15). Durch die Kante f wird die Rückkehradresse *(x, a)* ab der Zuordnung y als beendet markiert. Da diese Markierung durch die Kante f in y erfolgt, ist die Fortsetzung (x,d,y,f) zulässig. Die Kante e der Zuordnung x kann hingegen im weiteren Verlauf des Szenarios nicht weiter fortgesetzt werden. Da untergeordnete Aufrufinstanzen rekursiv beendet werden, werden Rückkehradressen ebenfalls rekursiv als beendet markiert (siehe Definition 66).

Definition 66.

$$\forall (c,tA) \in \varepsilon, (eds',c'), (eds'',c'') \in C :$$
$$c' = c \wedge c'' = eds' \Rightarrow (c'',tA) \in \varepsilon$$

5.5 Syntaxeinschränkungen

Eine Konsistenzüberprüfung des szenarienbasierten Anforderungsmodells und des zustandsbasierten Architekturmodells ist im Allgemeinen nicht entscheidbar. Eine entscheidbare und effiziente Konsisistenzüberprüfung kann durch eine Einschränkung der Syntax der verwendeten Modelle erreicht werden. Für die praktische Einsetzbarkeit des modellbasierten Ansatzes sind jedoch Konstrukte wie Schleifen, Hierarchien und parallele Abläufe im zustandsbasierten Modell erforderlich. In den vorangehenden Kapiteln wurde ein szenarienbasiertes Anforderungsmodell und ein zustandsbasiertes Architekturmodell mit der erforderlichen Ausdrucksmächtigkeit für die Konsistenzüberprüfung definiert. Zudem wurde der Zusammenhang der Modelle durch die Definition von Zuordnungen festgelegt. Im Folgenden werden Syntaxeinschränkungen vorgestellt, die eine entscheidbare und effiziente Konsistenzüberprüfung ermöglichen. Die Ausdrucksmächtigkeit der Modelle wird hierbei nicht verringert. Vielmehr werden die Variationen zur Beschreibung eines Sachverhalts eingeschränkt.

5.5.1 Einschränkung von Schleifen

Die Ursache für die Nicht-Entscheidbarkeit der Konsistenzüberprüfung sind Schleifen. Dies gilt jedoch nicht für jede Schleifenart. Die zustandsbasierten Modelle *zg1* und *zg2* enthalten beispielsweise verschiedene Schleifenarten (siehe Abb. 5.19). Die Interaktio-

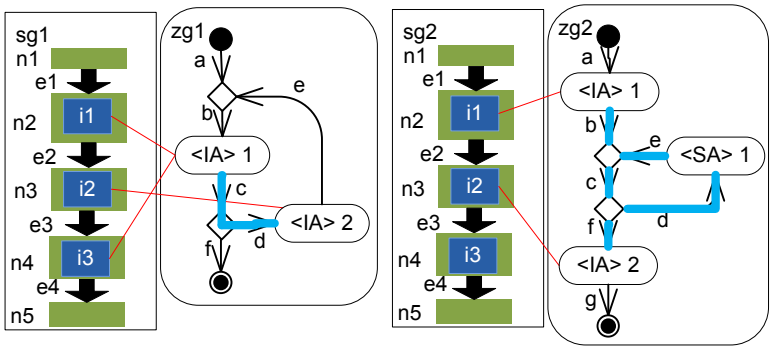


Abbildung 5.19: Schleifen und Zuordnungen

nen *i1* und *i2* der Szenarienschritte *n2* und *n3* des Szenarios *sg1* sind beispielsweise mit den Interaktionsknoten *1* und *2* verknüpft. Der Auslöser für den Übergang *e2* ist der Abschluss der Interaktion *i1*. Im Szenarienschritt *n3* ist die Interaktion *i2* aktiv. Wie in Abschnitt 5.4 beschrieben, können der Kante *e2* folglich die Kanten *c* und *d* des zustandsbasierten Modells zugeordnet werden (siehe Definitionen 39 und 40). Innerhalb dieser Zuordnung existiert keine Schleife. Dies gilt für alle möglichen Zuordnungen dieses Beispiels. Die Schleife ausgehend von der Kante *b* wird im Rahmen des Szenarios genau einmal durchlaufen. Das zustandsbasierte Modell *zg2* enthält ebenfalls eine Schleife (siehe Abb. 5.19). Die Interaktionen *i1* und *i2* der Szenarienschritte *n2* und *n3* des Szenarios *sg2* sind beispielsweise mit den Interaktionsknoten *1* und *2* verknüpft.

Wie in Abschnitt 5.4 beschrieben, können der Kante *e2* die Kanten *b*, *c*, *d*, *e* und *f* zugeordnet werden (siehe Definitionen 39 und 40). Innerhalb dieser Zuordnung existiert eine Schleife.

Eine Ursache der Nicht-Entscheidbarkeit der Konsistenzüberprüfung sind Schleifen innerhalb von Zuordnungen. Diese Schleifen sind jedoch zur Beschreibung der Architektur nicht relevant. Die Architektur muss beschreiben, wie Anforderungen umgesetzt werden. Anforderungen werden im szenarienbasierten Modell durch Interaktionen mit dem zu entwickelnden System beschrieben. Diese werden vom System mit den in Beziehung stehenden Interaktionsknoten realisiert. Die Interaktionen *i1* und *i2* des Szenarios *sg2* werden beispielsweise durch die Interaktionsknoten *1* und *2* realisiert (siehe Abb. 5.19). Entsprechend der Zuordnungsvorschriften darf es mit Ausnahme der Startkante keine ausgehende Kante aus einem Interaktionsknoten innerhalb einer Zuordnung geben (siehe Definition 41). Damit bilden die Kanten einer Zuordnung Pfade zwischen Interaktionsknoten. Die Kanten *b*, *c* und *f* des Graphs *zg2* bilden beispielsweise einen Pfad zwischen den Interaktionsknoten *1* und *2*. Andere Knoten auf dem Pfad realisieren keine Interaktion und stellen Verarbeitungsschritte des Systems dar. Die Anzahl und die Reihenfolge der Verarbeitungsschritte ist aus Sicht der Anforderungen irrelevant, sofern die Ein- und Ausgaben in Form der Interaktionen korrekt realisiert sind. Innerhalb der Zuordnungen können damit Pfade zu einem Knoten zusammengefasst werden. Dies gilt auch für Pfade mit Schleifen. Die Schleife im zustandsbasierten Modell *zg2* kann beispielsweise zum Verarbeitungsknoten *sa1* zusammengefasst werden (siehe Abb. 5.20). Durch Syntaxeinschränkungen können

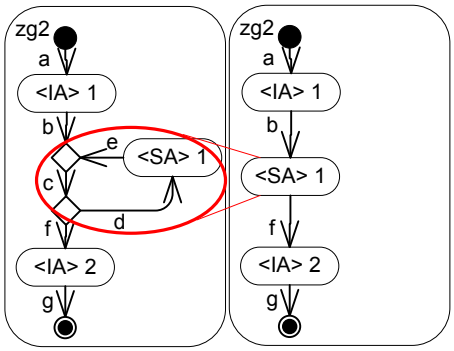


Abbildung 5.20: Zusammenfassung einer Schleife

Schleifen innerhalb von Zuordnungen ausgeschlossen werden. Da andere Schleifentypen durch diese differenzierende Syntaxeinschränkung weiterhin zugelassen sind, bleibt die Ausdrucksmächtigkeit der Modelle erhalten. Beispielsweise ist die Modellierung eines Loginvorgangs mit Schleifen weiterhin möglich.

Ein Ausschluss von Schleifen innerhalb von Zuordnungen kann durch die folgende Syntaxeinschränkung erreicht werden: Eine Schleife im zustandsbasierten Modell ist nur zulässig, wenn sich auf diesem ein Interaktionsknoten befindet (siehe Definition 67).

Definition 67.

$$\forall n \in ND^{ZG}, p \in pathsFrom(n, ED^{ZG}) :$$
$$\exists (s,t) \in ED^{ZG} : s = get(p, len(p)) \wedge t = get(p,1)$$
$$\Rightarrow \exists x \in \{1,..,len(p)\} : get(p,x) = (i,y) \wedge y = y_{ia}$$

Durch die Syntaxeinschränkung werden die Variationen zur Beschreibung eines Sachverhalts eingeschränkt. Dies hat Auswirkung auf die Art der Modellierung und potenziell auf die praktische Einsetzbarkeit des modellbasierten Ansatzes. Durch den Ausschluss von Schleifen innerhalb von Zuordnungen kann nicht weiter beschrieben werden, dass innerhalb eines Szenarienschritts beliebig viele Prozesse in einer Schleife gestartet werden können. Beispielsweise kann der parallele Start von beliebig vielen Interaktionen (z.B. Multicasting) innerhalb eines Szenarienschritts durch eine Schleife nicht weiter ausgedrückt werden. Das zustandsbasierte Modell *zg3* zeigt eine Schleife, die beliebig oft durchlaufen werden kann (siehe Abb. 5.21). Hierdurch kann mit dieser Architektur eine Interaktion durch den Interaktionsknoten *2* beliebig oft gestartet werden. Durch das szenarienbasierte Modell *sg3* wird beispielsweise ein dreimaliger Start gefordert. Durch den Auschluss von Schleifen innerhalb einer Zuordnung

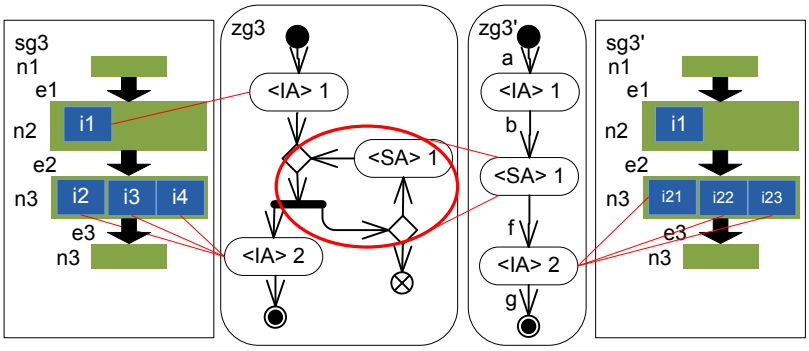


Abbildung 5.21: Auswirkungen Multicast

sind derartige Schleifen nicht zulässig und müssen beispielsweise zum Verarbeitungsknoten *1* des zustandsbasierten Modells *zg3'* zusammengefasst werden. Trotz dieser Einschränkung ist die Eignung des modellbasierten Ansatzes für den Einsatz in der Praxis weiterhin gegeben. Die parallele Durchführung von beliebig vielen Interaktionen kann schließlich als eine Interaktion aus beliebig vielen Teilen angesehen werden. In der frühen Phase des Architekturentwurfs kann folglich die parallele Durchführung von beliebig vielen Interaktionen durch einen Interaktionsknoten ausgedrückt werden, der diese Interaktionen realisiert. Die Interaktionen *i2*, *i3* und *i4* des Szenarios werden somit beispielsweise durch die Teilinteraktionen *i21*, i22 und i23 beschrieben (siehe Abb. 5.21). Die Gesamtinteraktion ist erst abgeschlossen, wenn alle Teilinteraktionen abgeschlossen sind. Teilinteraktionen können nicht einzeln abgeschlossen und fortgesetzt werden. Um die Weiterverarbeitung der Ergebnisse einzelner Teilinteraktionen zu

beschreiben, können Variablen zur Beschreibung des Systemzustands verwendet werden (siehe Kapitel 3). Ein weiterer Anwendungsfall ist die Verwendung von Schleifen zum Start von beliebig vielen Prozessen innerhalb von Zuordnungen, die keine Interaktion starten. Diese sind jedoch aus Sicht der Anforderungen irrelevant und können zu Verarbeitungsknoten zusammengefasst werden. Eine detaillierte Beschreibung dieser Verarbeitungsknoten kann in späteren Entwicklungsstufen erfolgen.

Durch die Einschränkung der Schleifen innerhalb von Zuordnungen werden die Varianten zur Beschreibung eines Sachverhalts eingeschränkt. Für diese Varianten existiert für die Architektur jedoch stets eine alternative Beschreibungstechnik. Zudem sind Schleifen im Allgemeinen weiterhin zugelassen. Letztendlich ermöglichen die Einschränkung eine entscheidbare Konsistenzüberprüfung von Anforderungs- und Architekturmodellen. Die Konsistenzüberprüfung kann auf das Erreichbarkeitsproblem eines Zustands im zustandsbasierten Modell reduziert werden (siehe Kapitel 4). Dieses Problem ist nicht entscheidbar für Turing-vollständige Modelle. Durch die Syntaxeinschränkung sind innerhalb von Zuordnungen Schleifen durch Rekursion genauso wie einfache Schleifen ausgeschlossen. Durch den Ausschluss von Schleifen wird die Komplexität des Erreichbarkeitsproblems erheblich reduziert. Im einfachen Petrinetz ohne Schleifen wurde beispielsweise die NP-Vollständigkeit des Erreichbarkeitsproblems nachgewiesen [26]. Schleifen durch Rekursion sind ebenfalls ausgeschlossen. Daher ist die Ausdrucksmächtigkeit des zustandsbasierten Modells wie beispielsweise eines rekursiven Petrinetzes oder eines Aktivitätsdiagramms innerhalb der Zuordnung nicht größer als die eines einfachen Petrinetzes (siehe Kapitel 2.2.4). Folglich ist auch das Erreichbarkeitsproblem für diese Modelle innerhalb von Zuordnungen NP-vollständig. Die Konsistenzüberprüfung des szenarienbasierten Anforderungsmodells und des zustandsbasierten Architekturmodells von CREATE innerhalb einzelner Zuordnungen ist damit nach der Abbildung auf die abstrakte Syntax in Kapitel 5.3 und der Einschränkung von Schleifen trotz der Turing-vollständigkeit des AVD entscheidbar.

5.5.2 Zuordnungsvarianten

Die Konsistenzüberprüfung über mehrere Zuordnungen hinweg hat ohne weitere Einschränkungen eine exponentielle Komplexität. Innerhalb einer Zuordnung können mehrere Entscheidungsmöglichkeiten existieren, die bei der Konsistenzüberprüfung einer hierauf folgenden Zuordnung berücksichtigt werden müssen. Hierdurch kann die Anzahl der zu prüfenden Zuordnungsvarianten mit der Anzahl der Szenarienschritte exponentiell wachsen. Zu der Kante *e1* existieren beispielsweise zwei Zuordnungsmöglichkeiten (siehe Abbildung 5.22). Bei einer Variante werden neben der Kante *e* die Kanten *b* und *d* zugeordnet, bei der anderen die Kante *c*. In beiden Fällen ist die Kante *e* die Endkante. Die Kante *e* führt zu dem Interaktionsknoten *2*, die die Interaktion *i1* realisiert. Beide Zuordnungen entsprechen damit dem Szenario. Für die Zuordnung zu *e2* existeren ebenfalls zwei Zuordnungsmöglichkeiten. Bei einer Variante werden die Kanten *f* und nach Synchronisierung die Kante *i* zugeordnet. Die Kante *i* führt zu dem Interaktionsknoten *5*, der die Interaktion *i2* realisiert. Diese Zuordnung würde dem Szenario entsprechen. Bei der zweiten Variante wird die Kante *h* zugeordnet. Der Interaktionsknoten *5* wird nicht erreicht. Diese Zuordnung entspricht damit nicht dem Szenario. Für die erste Variante muss die Kante *d* des zustandsbasierten Modells

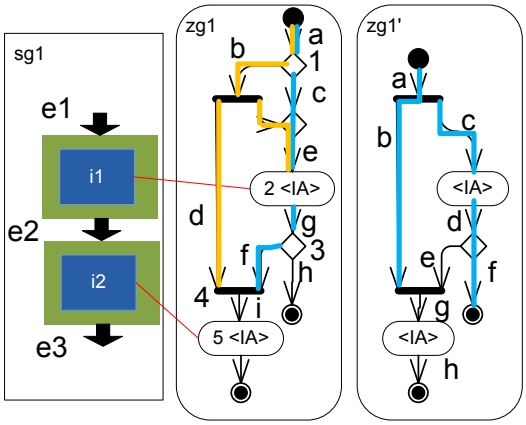


Abbildung 5.22: Komplexität Zuordnungsvarianten

der Kante *e1* des szenarienbasierten Modells zugeordnet werden (siehe Definition 50). Dies ist nur in einer der beiden Zuordnungsvarianten zu *e1* der Fall. Die Varianten der vorangehenden Zuordnungen müssen demnach bei der Konsistenzüberprüfung berücksichtigt werden. Die Komplexität der Konsistenzüberprüfung steigt damit exponentiell mit der Anzahl der Szenarienschritte.

Mehrere mögliche Zuordnungsvarianten, die dem Szenario entsprechen, führen zu einer exponentiell wachsenden Komplexität der Konsistenzüberprüfung. Für Architektur wird die Beschreibung mehrerer Zuordnungsvarianten jedoch nicht benötigt. Endkanten einer Zuordnung führen immer zu einem Interaktionsknoten oder zu einem Vereinigungsknoten. Varianten, deren Endkanten zu den selben Interaktionsknoten und Vereinigungsknoten führen, sind redundant und können weggelassen werden. Varianten zu anderen Interaktionsknoten entsprechen nicht dem Szenario. Interessant sind daher nur die Varianten, bei denen die Endkanten zu Vereinigungsknoten variieren. In diesem Fall kann die Ablaufbeschreibung jedoch so abgeändert werden, dass der Kontrollfluss immer die selben Vereinigungsknoten erreicht. Beispielsweise kann das zustandsbasierte Modell *zg1* zu *zg1'* abgeändert werden (siehe Abbildung 5.22). In diesem Modell existiert keine Alternative zur Zuordnung der Kante *b* zur Kante *e1* des szenarienbasierten Modells. Bei der Prüfung der Zuordnung zur Kante *e2* ist daher eine Berücksichtigung mehrerer Varianten der Zuordnung *e1* nicht erforderlich. Bei der Prüfung muss nur noch die zusätzliche Endkante in der Zuordnung zu *e1* berücksichtigt werden. Durch einen Ausschluss mehrerer zulässiger Zuordnungsvarianten, wächst die Anzahl der zu prüfenden Varianten im Allgemeinen nicht weiter exponentiell mit der Anzahl der Szenarienschritte. Dieser Ausschluss kann durch Syntaxeinschränkungen erreicht werden. Ausgehend von einem Entscheidungsknoten darf es keine zwei unterschiedlichen Pfade zu einem Interaktionsknoten geben, ohne einen Interaktionsknoten auf dem Pfad (siehe Definition 68).

Definition 68.

$$\forall n = (i,y) \in ND^{ZG} : y = y_{dec} \Rightarrow$$
$$\forall p, p' \in pathsFrom(n, ED^{ZG}) : p = p' \vee get(p, len(p)) \neq get(p', len(p))$$
$$\vee (get(p, len(p)) = (i', y') \wedge y' \neq y_{ia}) \vee$$
$$\exists x \in \{1, .., len(p)\} : get(p,x) = (j,y) \wedge y = y_{ia} \wedge x \neq len(p)$$

Zudem darf es ausgehend von einem Entscheidungsknoten keine zwei unterschiedlichen Pfade zu zwei unterschiedlichen Vereinigungsknoten geben, ohne einen Interaktionsknoten auf dem Pfad (siehe Definition 69).

Definition 69.

$$\forall n = (i,y) \in ND^{ZG} : y = y_{dec} \Rightarrow$$
$$(\forall p, p' \in pathsFrom(n, ED^{ZG}) : p = p' \vee get(p, len(p)) = get(p', len(p))$$
$$\vee (get(p, len(p)) = (i', y') \wedge y' \neq y_{join}) \vee$$
$$(get(p', len(p')) = (i'', y'') \wedge y'' \neq y_{join})) \vee$$
$$\exists x \in \{1, .., len(p)\} : get(p,x) = (j,y) \wedge y = y_{ia}$$

Durch diese Einschränkungen kann nur eine der Zuordnungsvarianten dem Szenario entsprechen. Damit ist eine Prüfung der Varianten anderer Zuordnungen nicht weiter erforderlich. Der Entscheidungsknoten im zustandsbasierten Modell *zg1* ist beispielsweise nicht weiter zulässig und die Beschreibung kann zu *zg1'* abgeändert werden. Damit existiert nur eine Zuordnungsvariante zur Kante *e1* des szenarienbasierten Modells.

Durch die Einschränkung der Beschreibung von Zuordnungsvarianten werden die Möglichkeiten zur Beschreibung eines Sachverhalts eingeschränkt. Im Allgemeinen legt die Wahl der Entscheidungen die Interaktionknoten zusammen mit den erreichten allgemeinen Knoten innerhalb der Zuordnung fest. In bestimmten Fällen befinden sich damit innerhalb einer Zuordnung zusätzliche Kanten zu Vereinigungsknoten wie die Kante *b* im szenarienbasierten Modell *sg1* und dem zustandsbasierten Modell *zg1'* (siehe Abb. 5.22). Diese können jedoch bei einer automatisierten Konsistenzüberprüfung berücksichtigt werden. Zudem gelten diese Einschränkungen nur, wenn sich die Kanten der Pfade in ein und der selben Zuordnung befinden. Letztendlich verringern die Einschränkungen der Zuordnungsvarianten die Komplexität der Konsistenzüberprüfung. Durch die Syntaxeinschränkungen ist eine Berücksichtigung mehrerer Zuordnungsvarianten anderer Zuordnungen nicht erforderlich. Die Prüfung ist damit lokal auf die Varianten einer Zuordnung beschränkt und die Anzahl der zu prüfenden Zuordnungsvarianten steigt nicht weiter exponentiell mit der Anzahl der Szenarienschritte.

5.5.3 Mischung paralleler Abläufe

Durch den Ausschluss von Schleifen haben zustandsbasierte Modelle wie Aktivitätsdiagramme und das AVD nach der Abbildung auf die Graphensyntax innerhalb von

Zuordnungen die Ausdrucksmächtigkeit eines Petrinetzes ohne Schleifen. Das Problem der Erreichbarkeit eines Zustands im Petrinetz ohne Schleifen ist NP-vollständig [26] und damit auch das Problem der Konsistenzüberprüfung. Ein Problem kann als effizient lösbar angesehen werden, wenn dieses P-vollständig ist [18][98]. Die Ursache für die NP-Vollständigkeit der Konsistenzüberprüfung trotz des Ausschlusses von Schleifen ist die Mischung paralleler Abläufe. Das zustandsbasierte Modell *zg1* zeigt ein Beispiel einer Mischung paralleler Abläufe (siehe Abb. 5.23). Durch den Paralle-

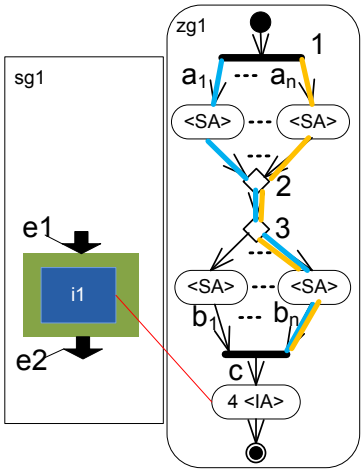


Abbildung 5.23: Mischung paralleler Abläufe

lisierungsknoten *1* werden *n* parallele Abläufe gestartet und durch den Mischknoten gemischt. Durch diese Mischung steigt die Anzahl der Entscheidungsmöglichkeiten am Entscheidungsknoten *3* mit *n* stark an.

Die Mischung paralleler Abläufe innerhalb von Zuordnungen ist die Ursache für die NP-Vollständigkeit der Konsistenzüberprüfung. Sie ist jedoch für die Beschreibung der Architektur nicht relevant. Interaktionsknoten und Verarbeitungsknoten können, wie im Abschnitt 5.5.1 beschrieben, parallele Abläufe zusammenfassen. Das mehrfache Durchlaufen ein und der selbe Kante innerhalb einer Zuordnung ist demnach nicht zwingend erforderlich. Die folgende Syntaxeinschränkung schließt den Fall der Mischung paralleler Abläufe aus: Es darf keine zwei Pfade ohne Interaktionsknoten von einem Parallelisierungsknoten aus zu dem selben Mischknoten geben (siehe Definition 70).

Definition 70.

$$\forall n = (i,y) \in ND^{ZG} : y = y_{fork} \Rightarrow$$
$$\forall p, p' \in pathsFrom(n, ED^{ZG}) : p = p' \vee get(p, len(p)) \neq get(p', len(p'))$$
$$\vee (get(p, len(p)) = (i', y') \wedge y' \neq y_{merge})$$
$$\exists x \in \{1, .., len(p)\} : get(p,x) = (j,y) \wedge y = y_{ia}$$

Das zustandsbasierte Modell *zg1* ist damit nicht weiter zulässig, da ausgehend vom Parallelisierungsknoten mehr als einen Pfad zu dem Mischknoten ohne Interaktionsknoten existieren (siehe Abb. 5.23). Die parallelen Abläufe können in der frühen Phase des Architekturentwurfs alternativ hierzu durch einen Verarbeitungsknoten beschrieben werden, der für diese Abläufe steht.

Durch die Einschränkung der Mischung paralleler Abläufe innerhalb von Zuordnungen werden die Variationen zur Beschreibung eines Sachverhalts eingeschränkt. Für die eingeschränkten Varianten zur Beschreibung eines Sachverhalts existiert in der Architektur jedoch stets eine alternative Beschreibungstechnik. Beispielsweise können parallel durchzuführende Serviceaktionen zu einer Serviceaktion zusammengefasst werden. Mit Variablen besteht die Möglichkeit die Anzahl der parallelen Abläufe zu beschreiben. Zudem sind Mischungen paralleler Abläufe im Allgemeinen weiterhin zugelassen, wenn sich Interaktionsknoten auf den Pfaden befinden. Letztendlich ermöglichen die Einschränkungen eine effiziente Konsistenzüberprüfung von Anforderungs- und Architekturmodellen. Das Problem der Erreichbarkeit einer Stelle im Petrinetz ist P-vollständig, wenn das Netz *1-safe* ist [26]. Ein Petrinetz ist *1-safe*, wenn sich in jedem Fall immer maximal eine Marke in einer Stelle befindet. Durch mehrere initiale Knoten und Schleifen können mehrere Marken an einer Stelle anliegen. Bei einem Ausschluss von Schleifen und mehreren initialen Knoten muss folgender Fall eintreten, damit sich mehrere Marken in einer Stelle befinden können: Eine Transition muss mehrere Marken erzeugen. Innerhalb der darauf folgenden Schaltsequenz müssen zwei Transitionen schalten, die ein und die selbe Stelle markieren. Der parallele Ablauf wird damit gemischt. Dieser Fall ist durch die Einschränkung der Mischung paralleler Abläufe innerhalb von Zuordnungen ebenfalls ausgeschlossen. Die Mächtigkeit des zustandsbasierten Modells wie beispielsweise eines Aktivitätsdiagramms und des AVDs innerhalb einer Zuordnung entspricht damit der eines schleifenfreien *1-safe* Petrinetzes (siehe 2.2.4). Damit ist das Problem der Konsistenzüberprüfung P-vollständig und kann effizient durchgeführt werden.

5.6 Konsistenzbedingungen

Auf Basis des eingeführten szenarienbasierten Anforderungsmodells und des zustandsbasierten Architekturmodells sowie derer Zusammenhänge können Konsistenzbedingungen für eine abschließende Konsistenzüberprüfung definiert werden. Durch die Syntaxeinschränkungen hat das zustandsbasierte Modell wie beispielsweise das AVD innerhalb einer Zuordnung die Mächtigkeit eines *1-safe* Petrinetzes (siehe Kapitel 5.5.3). Damit kann innerhalb einer Zuordnung kein Knoten mehrfach vom Kontrollfluss erreicht werden. Dies gilt auch für Entscheidungsknoten. Damit kann der Kontrollfluss innerhalb einer Zuordnung nur eine aus einem Entscheidungsknoten ausgehende Kante durchlaufen. In einer Zuordnung muss folglich auch für jede Kante zu einem Entscheidungsknoten genau eine ausgehende Kante zugeordnet sein (siehe Definition 71). Durch die gegebenen Eigenschaften bilden die Kanten und die verbundenen Knoten einer Zuordnung einen Baum.

Definition 71.

$$\forall(ed,E,b,F) \in ED^{assign}, (s,t=(i,y)) \in E : y = y_{dec} \\ \Rightarrow \exists!(s',t') \in E : s' = t$$

Das Element b des Tupels $(ed,E,b,F) \in ED^{assign}$ legt hierbei die Wurzel des Baumes fest. Die Menge F beschreibt die Blätter des Zuordnungsbaums. Die Kante a der Zuordnung x legt beispielsweise den initialen Knoten als Wurzel des Baums fest (siehe Abb. 5.24). Die Endkanten b und c legen die beiden Interaktionsknoten des zustandsbasierten Modells zg als Blätter des Zuordnungsbaums fest. Da das zustandsbasierte

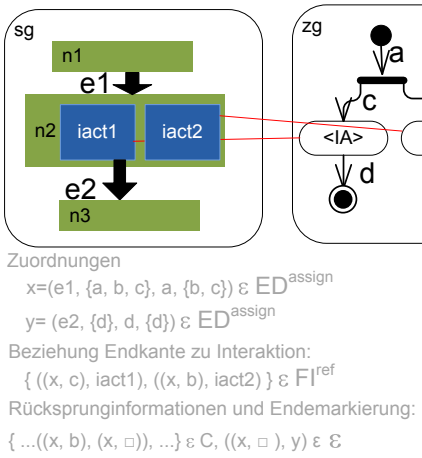


Abbildung 5.24: Konsistenzbedingungen am Beispiel

Modell innerhalb einer Zuordnung durch die Syntaxeinschränkungen die Mächtigkeit eines *1-safe* Petrinetzes hat (siehe Abschnitt 5.5), wird jede Endkante vom Kontrollfluss genau einmal erreicht. Durch diese Eigenschaft der Zuordnungen kann eine Konsistenzüberprüfung des szenarienbasierten Modells und des zustandsbasierten Modells auf Konsistenzbedingungen zwischen Endkanten der Zuordnung und den Interaktionen der Szenarienschritte beruhen.

Für die Definition der Konsistenzbedingungen werden die Endkante und die Interaktion über die Relation FI^{ref} miteinander in Beziehung gesetzt (siehe Definition 72).

Definition 72.

$$FI^{ref} =_{DEF} \left\{ (f \in ED^{sel}, iact \in I_{ACT}^{SYS}) \mid \right.$$

Jede mit einer Interaktion über FI^{ref} in Beziehung stehende Kante muss eine Endkante einer Zuordnung sein und zu einen Interaktionsknoten führen (siehe Definition 73).

Definition 73.

$$\forall(((ed,E,b,F), f = (a,(i,y))), iact) \in FI^{ref} : f \in F \wedge y = y_{ia}$$

Die Kanten b und c der Zuordnung x sind beispielsweise mit den Interaktionen *iact1* resp. *iact2* über FI^{ref} in Beziehung gesetzt (siehe Abb. 5.24). Da beide Kanten Endkanten der Zuordnung x sind, ist FI^{ref} zulässig. Jede Endkante kann nur mit genau einer Interaktion in Beziehung stehen und umgekehrt (siehe Definition 74).

Definition 74.

$$\forall (f, iact), (f', iact') \in FI^{ref} : iact = iact' \Leftrightarrow f = f'$$

Die Endkante b darf beispielsweise nicht mit der Interaktion *iact2* und gleichzeitig mit der Interaktion *iact2* in Beziehung gesetzt werden (siehe Abb. 5.24). Umgekehrt darf die Interaktion *iact1* nicht gleichzeitig mit den Endkanten c und d in Beziehung gesetzt sein. Genau dann wenn eine Systeminteraktion in einem Szenarienschritt neu aktiv ist, muss sie mit einer Endkante über FI^{ref} in Beziehung stehen (siehe Definition 75).

Definition 75.

$$\forall iact = (A, E) \in I_{ACT}^{SYS}, (Nd^{SG}, Ed^{SG}, r, L) \in SG :$$
$$(\exists (s,t) \in Ed^{SG} : s \notin A \wedge t \in A) \Leftrightarrow (\exists (f, iact') \in FI^{ref} : iact = iact')$$

Die Interaktion *iact1* ist beispielsweise neu aktiv im Szenarienschritt *n2* (siehe Abb. 5.24). Damit muss die Interaktion über FI^{ref} mit einer Endkante in Beziehung stehen.

Zu jeder aktiven Interaktion muss im zustandsbasierten Modell ein Kontrollfluss aktiv sein. Innerhalb von Zuordnungen entspricht ein aktiver Kontrollfluss einer Endkante. Genau dann wenn eine Endkante einer Zuordnung zu einem Interaktionsknoten führt und die Rückkehradresse nicht über ε als beendet markiert ist, muss diese mit einer Interaktion über FI^{ref} in Beziehung stehen (siehe Definition 76).

Definition 76.

$$\forall eda = (ed, E, s, F) \in ED^{assign}, e = (a, (i, y)) \in F :$$
$$y = y_{ia} \wedge \nexists ((eda', e'), c) \in C, (c', tA) \in \varepsilon : eda' = eda \wedge e' = e \wedge c = c'$$
$$\wedge tA = eda \Leftrightarrow \exists ((a, e'), iact) \in FI^{ref} : a = eda \wedge e' = e$$

Die Endkante b führt beispielsweise zu einem Interaktionsknoten (siehe Abb. 5.24). Die Rückkehradresse dieser Kante innerhalb der Zuordnung x ist nicht über ε als beendet Markiert. Damit muss die Kante b über FI^{ref} mit einer Interaktion in Beziehung gesetzt werden.

Für jede Endkante-Interaktionsbeziehung muss folgendes gelten: Die Endkante ist einer Kante im szenarienbasierten Modell zugeordnet, die einen Übergang zu einem Szenarienschritt darstellt. In diesem Szenarienschritt ist die in Beziehung stehende Interaktion neu aktiv. Die Endkante führt in einen Interaktionsknoten der die Interaktion realisiert (siehe Definition 77).

Definition 77.

$$\forall ((((s,t), E, b, F), f = (a, b)), iact = (A, E)) \in FI^{ref} :$$
$$s \notin A \wedge t \in A \wedge iaNode(iact) = b$$

Die Endkante *b* befindet sich beispielsweise in der Zuordnung zur Kante *e1* (siehe Abb. 5.24). Die Kante *e1* führt in den Knoten *n2*. Die Endkante *b* ist mit der Interaktion *iact2* über FI^{ref} in Beziehung gesetzt. Die Interaktion *iact2* ist im Szenarienschritt *n2* neu aktiv. Zudem führt die Endkante *b* in den Interaktionsknoten, der *iact2* realisiert. Die Bedingung ist damit erfüllt.

Durch diese Bedingungen kann festgestellt werden, ob die erreichten Interaktionsknoten einer Zuordnung genau zu den neuen Interaktionen des nächsten Szenarienschritts passen. Hierbei wird auch die Beendigung von Interaktionen durch Beendigung von Aufrufinstanzen berücksichtigt. Inkonsistenzen durch neu aktivierte Interaktionen sind somit ausgeschlossen. Ein weiterer Fall sind Inkonsistenzen, die durch das Fehlen zuvor aktivierter Interaktionen entstehen. Das Fehlen von aktiven Interaktionen kann durch Prüfung der deaktivierten Interaktionen aufgedeckt werden. Eine Interaktion wird entweder direkt beendet oder dadurch, dass die Aufrufinstanz eines Kontrollflusses über einen finalen Knoten beendet wurde. Eine Konsistenzbedingung kann wie folgt definiert werden: Jede Interaktion, die in einem Übergang deaktiviert wird, ist der Auslöser des Übergangs oder die Rückkehradresse der in Beziehung stehenden Endkante muss über ε als beendet markiert sein (siehe Definition 78).

Definition 78.

$$\begin{aligned}
&\forall iact = (A,E) \in I_{ACT}^{SYS}, (Nd^{SG}, Ed^{SG}, r, L) \in SG, (s,t) \in Ed^{SG} : \\
&s \in A \wedge t \notin A \Leftrightarrow (s,t) \in E \vee \exists((eda,f), iact') \in FI^{ref}, \\
&((eda',e'),c) \in C, (c',tA) \in \varepsilon : iact = iact' \wedge eda' = eda \wedge \\
&e' = f \wedge c = c' \wedge tA = (ed,E',b,F) \wedge ed = (s,t)
\end{aligned}$$

Im Szenarienschritt *n3* sind beispielsweise die Interaktionen *iact1* und *iact2* deaktiviert (siehe Abb. 5.24). Die Interaktion *iact1* ist der Auslöser des Übergangs. Die Deaktivierung ist damit zulässig. Die Interaktion *iact2* ist mit der Endkante *b* in Beziehung gesetzt. Die Rückkehradresse $(x,\square)$ dieser Endkante ist als beendet markiert, da sich die Endkante *c* zum finalen Knoten des zustandsbasierten Modells in der Zuordnung *y* befindet. Die Deaktivierung der Interaktion *iact2* ist damit auch zulässig.

Durch die eingeführten Bedinungen wird eindeutig die Konsistenz zwischen szenarienbasierten und zustandsbasierten Modellen definiert. Es werden die Bedingungen definiert zwischen den neuen Interaktionen eines Szenarienschritts im szenarienbasierten Modell und den vom Kontrollfluss erreichten Interaktionsknoten im zustandsbasierten Modell. Zudem werden auch Bedingungen definiert zwischen den deaktivierten Interaktionen eines Szenarienschritts im szenarienbasierten Modell und den beendeten Kontrollflüssen im zustandsbasierten Modell.

Ein Verfahren, welches diese Konsistenzbedingungen prüft, muss die durch Entscheidungsknoten entstehenden Zuordnungsvarianten hinsichtlich der definierten Konsistenzbedingungen prüfen. Wenn eine Zuordnung gefunden ist, bei der die Konsistenzbedinungen erfüllt sind, sind die Modelle konsistent. Eine Zuordnung beschreibt nach den Zuordnungsvorschriften im Abschnitt 5.4 einen möglichen Kontrollfluss. Die Suche nach einer Zuordnung entspricht damit der Prüfung, ob ein bestimmter Zustand im zustandsbasierten Modell erreicht werden kann. Durch die Definition der Syntaxeinschränkungen in Kapitel 5.5 ist das Erreichbarkeitsproblem für zustandsbasierte

Modelle wie Petrinetze, Aktivitätsdiagramme und AVDs P-vollständig (siehe Kapitel 5.5.3). Damit kann ein Verfahren definiert werden, mit welchem die Konsistenzüberprüfung effizient durchgeführt werden kann.

Kapitel 6

Werkzeugunterstützung

In dieser Arbeit wurde ein Ansatz zur Konsistenzsicherung von Anforderungen und Architekturen entwickelt. Ein modellbasierter Ansatz zur formalen und eindeutigen Beschreibung von Anforderungen und Architekturen wurde in Kapitel 3 vorgestellt. Die Konsistenz der Strukturmodelle kann zwischen Typen definiert werden. Dieser Zusammenhang erlaubt die Definition einfacher Konsistenzbedingungen, die eine automatisierte Konsistenzsicherung erlauben. Eine besondere Herausforderung ist die Konsistenzsicherung der Verhaltensmodelle, welche Hauptgegenstand des im Kapitel 5 vorgestellten Lösungsansatzes ist. Im Folgenden wird eine mögliche Implementierung des Konsistenzüberprüfungsverfahrens durch ein *Ecore*-Metamodell [84] und Syntaxeinschränkungen in der *Epsilon Validation Language* [48] vorgestellt. Zunächst werden die Anforderungen beschrieben, die von der Werkzeugunterstützung erfüllt werden. Anschließend wird die Architektur und die Implementierung der Werkzeugunterstützung vorgestellt.

6.1 Anforderungen

6.1.1 Domänenstruktur

Die Werkzeugunterstützung des in Kapitel 5 vorgestellten Lösungsansatzes soll für die Domäne einer Modellierungsumgebung entwickelt werden. Die Modellierungsumgebung besteht aus einem Benutzer, einem Modellierungswerkzeug, einem Modelltransformator und einem Konsistenzprüfer sowie aus einer Menge von Dokumentationen, Anforderungen und Architekturen (siehe Abb. 6.1). Der Benutzer soll mit Hilfe des Modellierungswerkzeugs ein Modell der Dokumentation für Anforderungen und die Architektur erstellen können. Zudem soll der Benutzer die Möglichkeit haben mit Hilfe des Konsistenzprüfers, die Konsistenz zwischen den Anforderungs- und Architekturmodellen zu prüfen. Der Konsistenzprüfer prüft hierbei die Konsistenz zwischen dem in Kapitel 5 vorgestellten szenarienbasierten Anforderungsmodell und dem zustandsbasierten Architekturmodell. Die vom Modellierungswerkzeug als Datei im Format *Extensible Markup Language* (XMI) [13] gespeicherten Modelle werden hierfür von dem

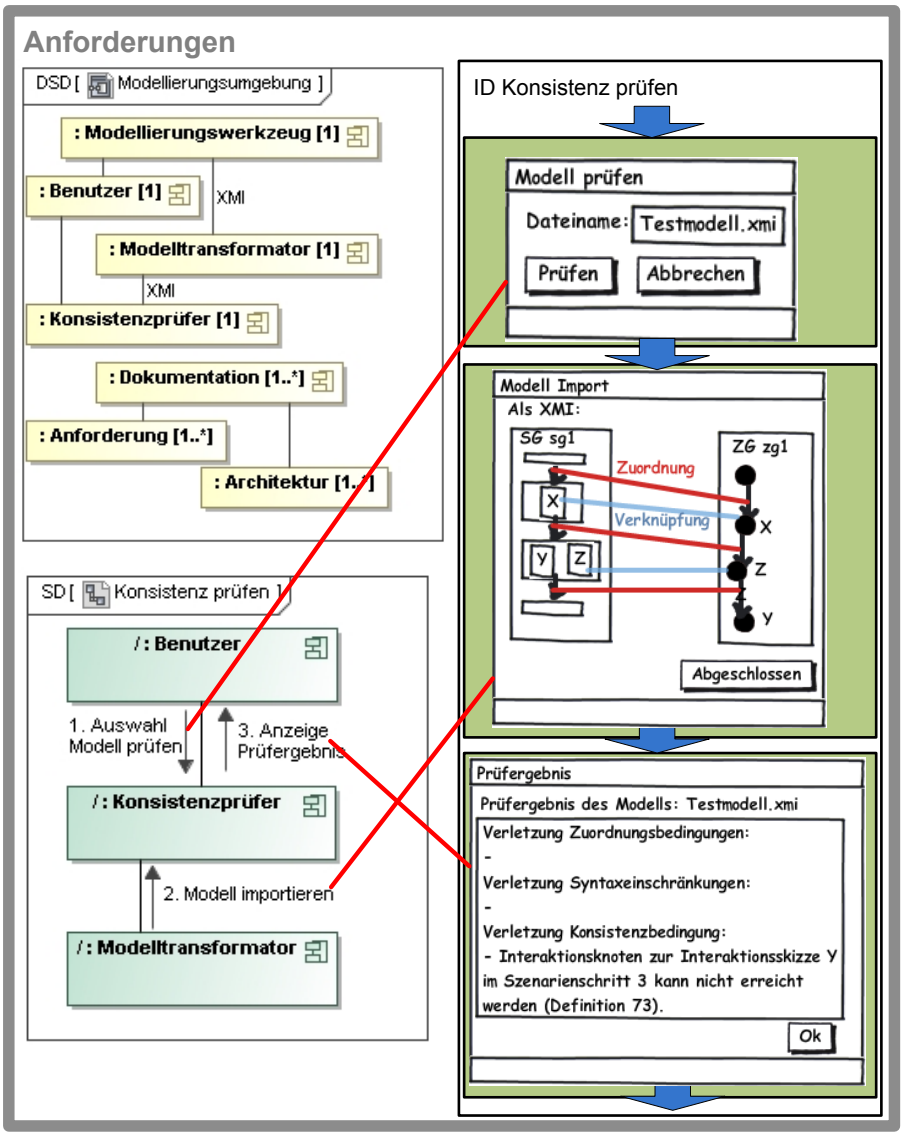


Abbildung 6.1: Anforderungen an den Konsistenzprüfer

Modelltransformator zu Modellen mit der in Kapitel 5 eingeführten abstrakten Syntax transformiert. Der Konsistenzprüfer liest diese Modelle für die Prüfung ein (siehe Abb. 6.1).

6.1.2 Szenario Konsistenz prüfen

Ein typischer Prozess in der Domäne, um die Konsistenz eines szenarienbasierten Anforderungsmodells und eines zustandsbasierten Architekturmodells zu prüfen, wird durch das SD *Konsistenz prüfen* und dem zugehörigen ID beschrieben (siehe Abb. 6.1). Das Szenario besteht aus den folgenden drei Schritten:

1. Im ersten Szenarienschritt wählt der Benutzer für den Konsistenzprüfer die XMI-Datei Testmodell.xmi aus. In der Datei ist das zu prüfende szenarienbasierte Anforderungsmodell und das zustandsbasierte Architekturmodell gespeichert. Der Benutzer startet den Prüfvorgang durch die Auswahl des Befehls *Prüfen*.

2. Im zweiten Szenarienschritt importiert der Konsistenzprüfer das szenarienbasierte Anforderungsmodell und das zustandsbasierte Architekturmodell aus der Datei Testmodell.xmi. Die abstrakte Syntax der Modelle ist in Kapitel 5 beschrieben. In der Datei des Szenarios befindet sich das szenarienbasierte Modell *sg1* und das zustandsbasierte Modell *zg1* (siehe Abb. 6.1). Die Zuordnungen wurden vom Modelltransformator hierbei entsprechend der Zuordnungsvorschriften aus Kapitel 5 und der gesetzten Verknüpfungen vorgenommen.

3. Im dritten Szenarienschritt gibt der Konsistenzprüfer die Inkonsistenzen aus, die nach den in Kapitel 5 definierten Bedingungen zwischen den Modellen bestehen. In den Modell des Szenarios ist beispielsweise die Konsistenzbedingung nach Definition 73 verletzt.

6.2 Architektur

6.2.1 Systemübersicht

Das im Rahmen der Werkzeugunterstützung entwickelte System ist der Konsistenzprüfer der Modellierungsumgebung. Dieses stellt daher die Systemgrenze der Architektur dar (siehe Abb. 6.2).

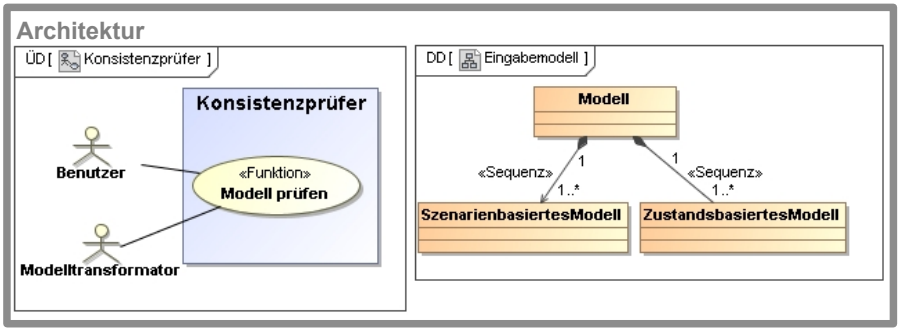


Abbildung 6.2: Überblick und Datenobjekte des Konsistenzprüfers

Der Konsistenzprüfer interagiert mit dem Benutzer und dem Modelltransformator. Diese Teile der Domänenstruktur sind damit relevante Akteure. Eine Übersicht über die Berücksichtigung der Teile der Domäne durch die Architektur ist in Tabelle 6.1 gegeben. Die durch das Szenario *Konsistenz prüfen* spezifizierten Anforderungen werden

Teil der Domäne	**Realisierung in der Architektur**
Benutzer	Akteur Benutzer
Konsistenzprüfer	Systemgrenze Konsistenzprüfer
Modelltransformator	Akteur Modelltransformator
Dokumentation	Datenobjekt Modell
Anforderung	Datenobjekt SzenarienModell
Architektur	Datenobjekt ZustandsbasiertesModell

Tabelle 6.1: Abbildung der Domänenteile auf die Architekturrealisierung

durch die Funktion *Modell prüfen* des Systems realisiert (siehe Abb. 6.2). Die Realisierung von Nachrichten wird im Abschnitt 6.2.3 beschrieben.

6.2.2 Datenobjekte

Das Datenmodell DD *Eingabemodell* beschreibt die vom System zu verarbeitenden Datenobjekte (siehe Abb. 6.2). Die vom Konsistenzprüfer zu verarbeitenden Datenobjekte sind Objekte vom Typ *Modell*, *SzenarienbasiertesModell* und *ZustandsbasiertesModell*. Ein Objekt vom Typ *Modell* besteht hierbei aus einer Menge von Objekten vom Typ *SzenarienbasiertesModell* und *ZustandsbasiertesModell*. Die Datenobjekte des Datenmodells realisieren die Entitäten *Dokumentation*, *Anforderung* und *Architektur* der Domäne. Vom Konsistenzprüfer wird das Modell zur Dokumentation der Anforderungen und Architekturen geprüft. Da vom Konsistenzprüfer die Konsistenz zwischen den Verhaltensmodellen geprüft werden soll, werden die Anforderungen und Architekturen durch die in Kapitel 5 eingeführten szenarienbasierten resp. zustandsbasierten Modelle in der Architektur berücksichtigt. Eine Übersicht über die Berücksichtigung der Teile der Domäne durch die Architektur ist in Tabelle 6.1 gegeben.

6.2.3 Verhaltensbeschreibung des Systems

Der Konsistenzprüfer stellt die Funktion *Modell prüfen* zur Verfügung. Der Ablauf dieser Funktion wird durch das AVD *Modell prüfen* beschrieben (siehe Abb. 6.3). Das AVD beschreibt den folgenden Ablauf:

1. Der Dateiname wird über die Aktion *Modell Auswahl* eingelesen und in der Variablen *dateiName* vom Typ einer Zeichenkette abgelegt.

2. Das Modell wird in der Aktion *Modell importieren* aus der in der Variablen *dateiName* gespeicherten Datei importiert und in der Variablen *modell* vom Typ *Modell* abgelegt.

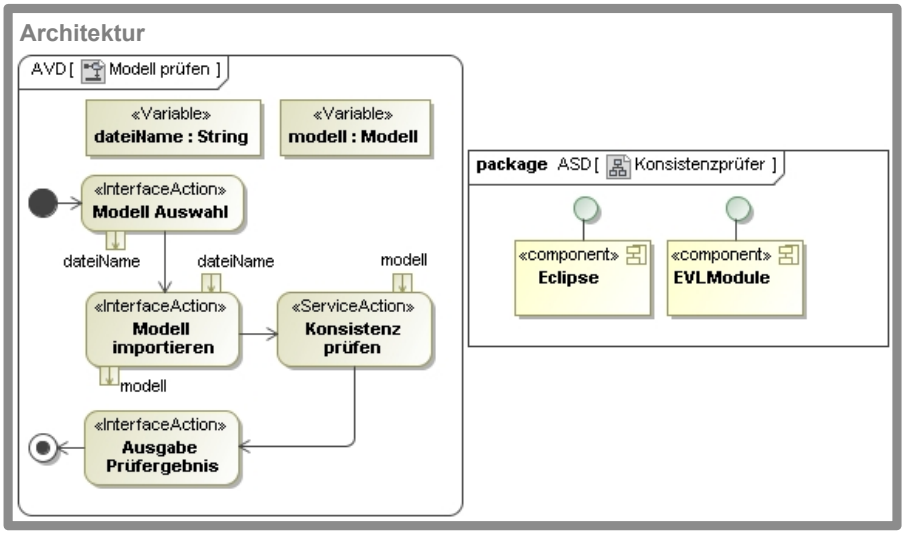


Abbildung 6.3: Ablauf der Funktion *Modell prüfen* und abgeleitete Struktur

3. Das Modell wird in der Aktion *Konsistenz prüfen* hinsichtlich seiner Konsistenz geprüft. Gegenstand der Prüfung sind die in Kapitel 5 eingeführten Syntaxeinschränkungen sowie die Zuordnungs- und Konsistenzbedingungen.

4. Das Ergebnis der Prüfung wird im Rahmen der Aktion *Ausgabe Prüfergebnis* dem Benutzer angezeigt.

Eine Übersicht über die Realisierung der Nachrichten des Szenarios *Konsistenz prüfen* durch die Aktionen des AVD *Modell prüfen* ist in Tabelle 6.2 gegeben.

Interaktionsskizze	**Realisierende Schnittstellenaktion**
Modell prüfen	Modell Auswahl
Modell Import	Modell importieren
Prüfergebnis	Ausgabe Prüfergebnis

Tabelle 6.2: Abbildung der Systeminteraktionen auf die realisierende Aktion

6.2.4 Struktur des Systems

Für die Realisierung des im AVD *Modell prüfen* beschriebenen Ablaufs werden bestehende Softwarekomponenten verwendet. Die Benutzeroberfläche des Konsistenzprüfers wird durch *Eclipse* [38] und die Konsistenzüberprüfung durch das *EVLModule* realisiert (siehe Abb. 6.3). *Eclipse* ist eine integrierte Entwicklungsumgebung urpsprünglich für die Programmierung in Java [95]. Die Entwicklungsumgebung wurde unter anderem um Werkzeuge zur Entwicklung von *Ecore*-Modellen im Rahmen des *Eclipse Modeling Framework* (EMF) [84] erweitert. *EVLModule* ist die *Eclipse*-Integration

der Funktionseinheit für die Überprüfung von Bedingungen, die in der Sprache *EVL* [48] geschrieben wurden. *EVL* ist eine Sprache zur Beschreibung von Einschränkungen und Bedingungen über Modelle vergleichbar mit der *Object Constraint Language* (OCL) [69]. Die Funktionseinheit *EVLModule* liest Modelle basierend auf einem *Ecore*-Metamodell. Das Modell wird hinsichtlich der Erfüllung der in *EVL* beschriebenen Bedingungen geprüft. Das Ergebnis der Prüfung wird über die Eclipse-Oberfläche ausgegeben. Eine Übersicht über die Realisierung der Aktionen des AVD *Modell prüfen* durch die Komponenten *Eclipse* und *EVLModule* ist in Tabelle 6.3 gegeben. Die Ge-

Aktion	**Komponente**	**Beschreibung**
Modell Auswahl	Eclipse	Auswahl über die *Eclipse*-Oberfläche
Modell importieren	EVLModule	Import des Modells als *EMF*
Konsistenz prüfen	EVLModule	Bedingungen in *EVL*
Ausgabe Prüfergebnis	Eclipse	Ausgabe über die *Eclipse*-Oberfläche

Tabelle 6.3: Abbildung der Aktionen auf die realisierende Komponenten

genstände der im Folgenden beschriebenen Implementierung ist das EMF-Metamodell der in Kapitel 5 vorgestellten szenarienbasierten und zustandsbasierten Modelle sowie die Syntaxeinschränkungen, Zuordnungs- und Konsistenzbedingungen in der Sprache *EVL*.

6.3 Implementierung

Die Konsistenzüberprüfung wird von der Komponente *EVLModule* durchgeführt. Im Folgenden wird die Implementierung des EMF-Metamodells der zu prüfenden Modelle und die Implementierung der Syntaxeinschränkungen, Zuordnungs- und Konsistenzbedingungen in der Sprache *EVL* vorgestellt. Hierbei wird jeweils beispielhaft ein Teil des *EMF*-Metamodells gezeigt. Über diesen Teil des Metamodells werden repräsentative Bedingungen als *EVL*-Implementierung dargestellt.

6.3.1 Abstrakte Syntax

Die Komponente *EVLModule* importiert die zu prüfenden szenarienbasierten und zustandsbasierten Modelle im XMI-Format. Die abstrakte Syntax der Modelle ist als *Ecore*-Metamodell in *EMF* implementiert. Die *EClasses ScenarioTree* und *StateModelGraph* implementieren hierbei die Datenobjekte *SzenarienbasiertesModell* und *ZustandsbasiertesModell* der Architektur und haben eine Kompositionsbeziehung mit der *EClass Model* (siehe Abb. 6.4). *Model* implementiert das Datenobjekt *Modell* der Architektur. Die Datenobjekte *SzenarienbasiertesModell* und *ZustandsbasiertesModell* der Architektur haben die in Kapitel 5 eingeführte abstrakte Syntax der szenarien- und zustandsbasierten Modelle. Die *EClasses* beschreiben die detaillierte Implementierung der abstrakten Syntax mit *EMF* (siehe Abb. 6.4). Eine in der abstrakten Syntax definierte mathematische Menge wird hierbei als *EClass* implementiert. Die Tupel-Komponenten der abstrakten Syntax werden durch Assoziationen realisiert. Beispielsweise implementieren die *EClasses ScenarioTree* und *StateModelGraph* die Mengen

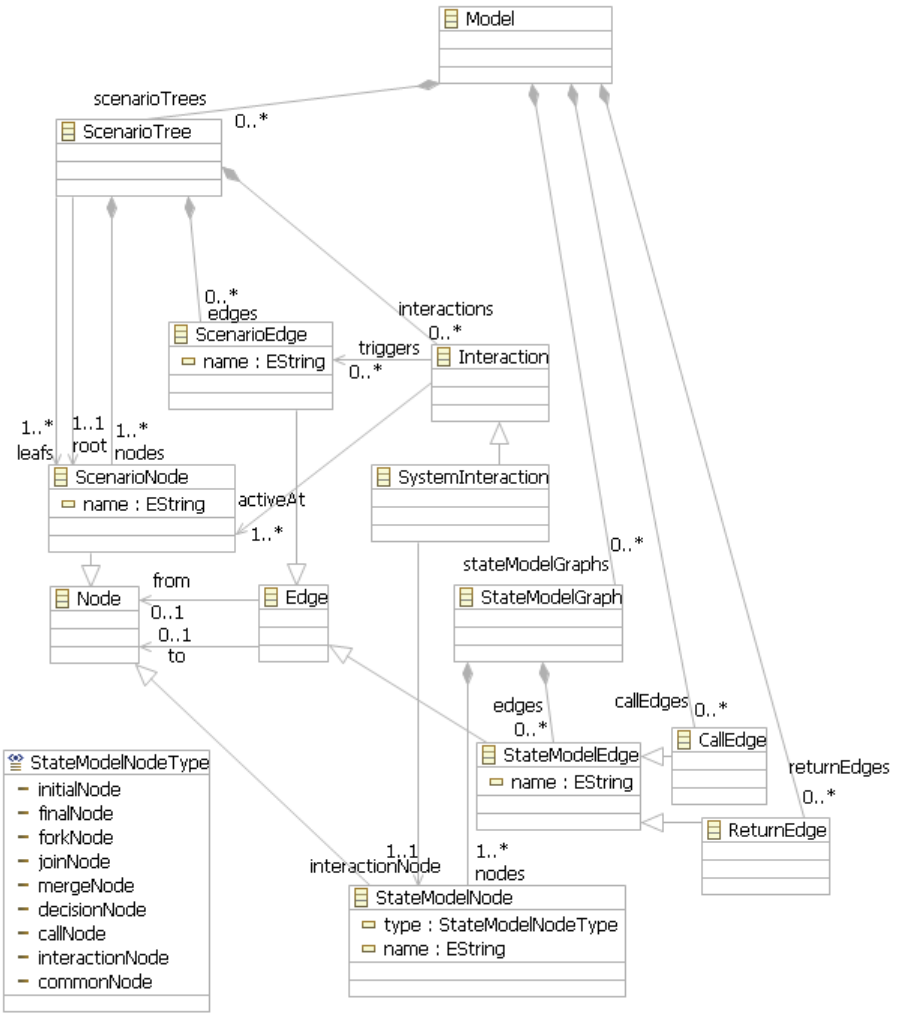


Abbildung 6.4: Graphstruktur-Sicht auf das Metamodell

SG und *ZG* (siehe Definitionen 9 und 16). Die Tupel-Komponente Nd^{SG} der Elemente aus *SG* wird beispielsweise durch die Kompositionsbeziehung mit der *EClass ScenarioNode* realisiert.

Zur vollständigen Definition der abstrakten Syntax in Kapitel 5 werden über die Mengen, Tupel und Tupel-Komponenten zusätzliche Einschränkungen beschrieben. Beispielsweise ist ein szenarienbasiertes Modell ein Baum (siehe Definition 9). Ein Baum hat keine Zyklen. Dies wird beispielsweise durch die Einschränkung in Definition 11 festgelegt. Die Implementierung dieser Einschränkung in *EVL* zeigt der Quellcodeabschnitt 6.1. Die Einschränkung in *EVL* wird direkt für den Kontext *EClass ScenarioTree* definiert. Eine Einschränkung besteht in EVL unter anderem aus den Ab-

```
context ScenarioTree{
 constraint NoCycles {
  check: self.nodes->forAll(n |
   pathsFrom(Sequence{n}, self.edges)->forAll(p |
    not self.edges->exists(e |
     e.from = p->at(p->size()-1) and e.to = p->at(0))))
  message: 'Schleife in Szenario ' + self
 }
}
```

Listing 6.1: Implementierung der Definition 11

schnitten *check* und *message*. Innerhalb des Abschnitts *check* wird die Einschränkung formuliert. Aus der Überprüfung der Einschränkung resultiert ein Wahrheitswert. Ist der Wahrheitswert *falsch*, ist die Einschränkung verletzt. In diesem Falle wird als Ergebnis der Prüfung die Zeichenkette im Abschnitt *message* ausgegeben. Innerhalb der *EVL*-Einschränkung im Quellcodeabschnitt 6.1 wird die Operation *pathsFrom* aus dem Quellcodeabschnitt 6.2 verwendet. Diese Operation ermittelt alle Pfade ausgehend von

```
operation pathsFrom(nodeSequence : Sequence(Node),
   edges : Set(Edge)) : Set(Sequence(Node)) {
 if(isPath(nodeSequence, edges)) {
  var newSet = Set{nodeSequence};
 for(e in edges) {
  newSet =  newSet->includingAll(
   pathsFrom(nodeSequence->including(e.to), edges));
 }
 return newSet;
 } else {
 return Set{};
 }
}
```

Listing 6.2: Implementierung der Definition 6

einem Knoten und implementiert damit die Funktion nach Definition 6. Die Operation *pathsFrom* verwendet unter anderem die Operation *isPath*, welche die Funktion in Definition 5 implementiert.

6.3.2 Zuordnungsbedingungen

Für die Konsistenzüberprüfung werden vom Modelltransformator den Kanten des Szenarios Kanten des zustandsbasierten Modells nach den Zuordnungsregeln aus Kapitel 5.4 zugeordnet. Die Definition der Zuordnungen erfolgt in Kapitel 5.4 ebenfalls durch Mengen, Tupel und Tupel-Komponenten. Diese werden als *EClass* resp. Assoziationen implementiert. Hierfür wird das Metamodell der Graphensyntax durch die implementierenden *EClasses* erweitert. Beispielsweise wird das Tupel in Definition 34 durch die *EClass Assignment* implementiert (siehe Abb. 6.5). Ein Objekt vom Typ *Model* kann hierbei eine Menge von Objekten vom Typ *Assignment* enthalten. Die Tupel-

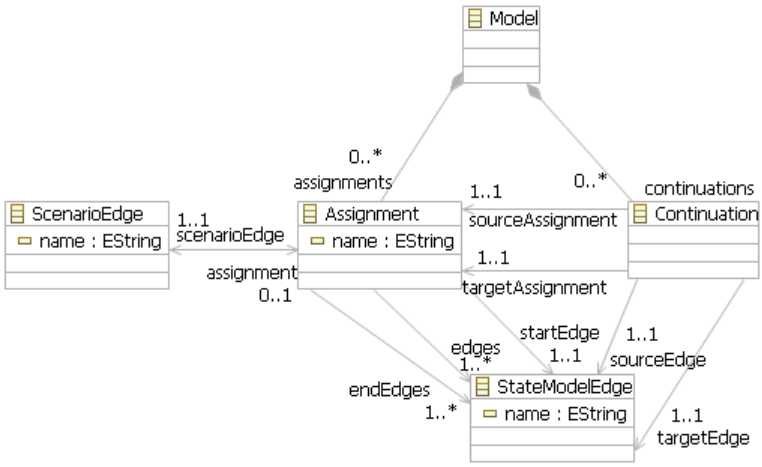


Abbildung 6.5: Zuordnungs-Sicht auf das Metamodell

Komponente *ed* wird durch die Assoziation *scenarioEdge* mit der *EClass ScenarioEdge* und die Tupel-Komponente *E* wird durch die Assoziation *edges* mit der *EClass StateModelEdge* realisiert. Die Assoziation scenarioEdge wurde für eine einfachere Realisierung von Zuordnungsvorschriften in *EVL* bidirektional implementiert. Ein Objekt vom Typ *ScenarioEdge* kann demnach mit einem Objekt vom Typ *Assignment* assoziiert sein.

Vom Konsistenzüberprüfer *EVLModule* wird die Einhaltung der Zuordnungsvorschriften überprüft. Die Implementierung der Zuordnungsvorschrift in Definition 38 zeigt beispielsweise der Quellcodeabschnitt 6.3. Die Einschränkung in *EVL* wird direkt

```
context ScenarioTree {
 constraint AssignmentAtNewInteraction{
 check: self.edges->forAll(se | (not self.interactions->exists(i |
    i.isTypeOf(SystemInteraction) and i.triggers->includes(se))
  and
  self.interactions->exists(i | (i.isTypeOf(SystemInteraction) and
    i.activeAt->excludes(se.from) and i.activeAt->includes(se.to))))
  = (se.assignment.isDefined() and se.assignment.startEdge.from.type =
      StateModelNodeType#initialNode))
  message: 'Verletzung der Definition ...'
 }
}
```

Listing 6.3: Implementierung der Definition 38

für den Kontext *EClass ScenarioTree* definiert. Im Gegensatz zu der Zuordnungsvorschrift in Definition 38 wird nicht über alle Objekte vom Typ *Assignment* iteriert. Durch die bidirektionale Assoziation zwischen *ScenarioEdge* und *Assignment* wird beispielsweise direkt auf die Startkante der Zuordnung für die Prüfung zugegriffen.

6.3.3 Syntaxeinschränkungen

Eine Konsistenzüberpfüfung des szenarienbasierten Anforderungsmodells und des zustandsbasierten Architekturmodells ist im Allgemeinen nicht entscheidbar. Eine entscheidbare und effiziente Konsisistenzüberprüfung kann durch eine Einschränkung der Syntax der verwendeten Modelle erreicht werden. Für die praktische Einsetzbarkeit des Modellierungsansatzes sind jedoch Konstrukte wie Schleifen, Hierarchien und parallele Abläufe im zustandsbasierten Modell erforderlich. Im Kapitel 5.5 wurden Syntaxeinschränkungen definiert, die eine entscheidbare und effiziente Konsistenzüberprüfung ermöglichen. Die Ausdrucksmächtigkeit der Modelle wird hierbei nicht verringert. Vielmehr werden die Varianten zur Beschreibung eines Sachverhalts eingeschränkt. Die Implementierung der Syntaxeinschränkung in Definition 67 zeigt beispielsweise der Quellcodeabschnitt 6.4. Die Einschränkung in *EVL* wird direkt für den Kontext

```
context StateModelGraph {
 constraint NoCyclesWithoutInteractionNode {
  check: self.nodes->forAll(n |
  pathsFrom(Sequence{n}, self.edges)->forAll(p |
  self.edges->exists(e | e.from=p->at(p->size()-1)
  and e.to=p->at(0))
  implies
  Set{0..p->size()-1}->exists(x |
   p->at(x).type=StateModelNodeType#interactionNode)))
  message: 'Verletzung der Definition ...'
 }
}
```

Listing 6.4: Implementierung der Definition 67

EClass StateModelGraph definiert. Innerhalb der Einschränkung wird die Operation *pathsFrom* verwendet, deren Implementierung im Quellcodeabschnitt 6.2 gezeigt wird.

6.3.4 Konsistenzbedingungen

Auf Basis des in Kapitel 5.2 eingeführten szenarienbasierte Anforderungsmodells und des zustandsbasierten Architekturmodells sowie der in Kapitel 5.4 definierten Zusammenänge wurden in Kapitel 5.6 Konsistenzbedingungen für eine abschließende Konsistenzüberprüfung definiert. Durch die in Kapitel 5.5 beschriebenen Syntaxeinschränkungen besteht die Möglichkeit die Konsistenzbedingungen so zu formulieren, dass die Konsistenzüberprüfung effizient durchgeführt werden kann. Für die Implementierung der Konsistenzbedingungen in *EVL* wird für jede Menge aus Kapitel 5.6 eine *EClass* im *EMF*-Metamodell eingeführt. Das Tupel *FI* wird beispielsweise durch die *EClass EndEdge2InteractionMapping* implementiert (siehe Abb. 6.6). Die Implementierung der Selektion von Kanten durch ED^{sel} (siehe Definition 52) erfolgt durch die *EClass EdgeSelection*. Objekte vom Typ *Model* werden aus Objekten vom Typ *EndEdge2InteractionMapping* und *EdgeSelection* aggregiert. Die Tupel-Komponenten werden durch Assoziationen zwischen *EClasses* realisiert. Die Tupel-Komponente *iact* aus *FI* wird beispielsweise durch die Assoziation *interaction* realisiert.

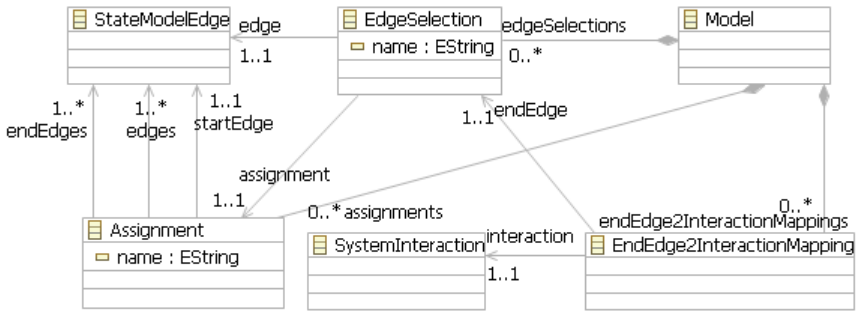


Abbildung 6.6: Sicht auf die Endkante-Interaktions-Beziehung im Metamodell

```
context EndEdge2InteractionMapping {
 constraint MappingToCorrectEndEdge {
 check: not self.interaction.activeAt->includes(
     self.endEdge.assignment.scenarioEdge.from)
  and self.interaction.activeAt->includes(
     self.endEdge.assignment.scenarioEdge.to) and
  self.interaction.interactionNode = self.endEdge.edge.to
 message: 'Verletzung der Definition ...'
 }
}
```

Listing 6.5: Implementierung der Definition 77

Die Interaktionen werdem vom Modelltransformator mit den Endkanten der Zuordnungen in Beziehung gesetzt. Vom Konsistenzüberprüfer *EVLModule* wird die Einhaltung der Konsistenzbedingungen überprüft. Die Implementierung der Konsistenzbedingung in Definition 77 zeigt beispielsweise der Quellcodeabschnitt 6.5. Genau dann wenn eine Systeminteraktion in einem Szenarienschritt neu aktiv ist, muss sie mit einer Endkante über *EndEdge2InteractionMapping* in Beziehung stehen. Durch diese Konsistenzbedingung wird die Übereinstimmung der neu aktivierten Interaktionen und der neu aktivierten Kontrollflüsse sichergestellt. Analog hierzu werden die übrigen Konsistenzbedingungen implementiert.

Kapitel 7

Evaluation

In dieser Arbeit wurde ein Ansatz zur Konsistenzsicherung von Anforderungen und Architekturen entwickelt. Ein modellbasierter Ansatz zur formalen und eindeutigen Beschreibung von Anforderungen und Architekturen wurde in Kapitel 3 vorgestellt. Die Konsistenz der Strukturmodelle kann zwischen Typen definiert werden. Dieser Zusammenhang erlaubt die Definition einfacher Konsistenzbedingungen, die eine automatisierte Konsistenzsicherung erlauben. Eine besondere Herausforderung ist die Konsistenzsicherung der Verhaltensmodelle, welche Hauptgegenstand des im Kapitel 5 vorgestellten Lösungsansatzes ist. In diesem Kapitel wird der entwickelte Ansatz zur Konsistenzsicherung von Anforderungen und Architekturen evaluiert. Das Ziel der Evaluation ist die Prüfung der Einsetzbarkeit des Ansatzes zur Erkennung und Behebung von Inkonsistenzen bei einer iterativen und evolutionären Entwicklung von Anforderungen und Architekturen. Der Ansatz wird anhand von Anwendungsszenarien in der frühen Phase der Entwicklung von Anforderungen und Architekturen evaluiert. Hierbei werden sowohl Änderungen an den Anforderungen als auch Änderungen an der Architektur vorgenommen. Bei den Änderungen entstehen Inkonsistenzen, die erkannt und behoben werden müssen. Die Anwendungsszenarien bauen hierbei auf den in Kapitel 4 eingeführten Fallbeispiel auf, bei dem ein Bibliothekssystem entwickelt wird. Im Fokus der Evaluation steht die Konsistenzsicherung der Verhaltensmodelle. Der Ansatz muss bei der Beschreibung gängiger Konstrukte wie Schleifen, Variablen, parallele Abläufe und Hierarchien einsetzbar sein. In den Anwendungsszenarien werden daher Änderungen an Modellen vorgenommen, in denen diese Konstrukte verwendet werden. Abschließend wird der Test der Werkzeugunterstützung beschrieben und die Ergebnisse der Evaluation zusammenfassend erklärt.

7.1 Anwendungsszenario - Anforderungsänderung

7.1.1 Änderung der Anforderungen an das Bibliothekssystem

Im ersten Anwendungsszenario erfolgt eine Änderung der Anforderungen an das Bibliothekssystem. Die zugrunde liegenden Modelle wurden in Kapitel 4 eingeführt.

Im Rahmen der Anforderungserhebung wurde festgestellt, dass der zu verwendende Emailserver bereits vorhanden ist und nicht Teil des zu entwickelnden Systems ist. Die Versendung von Emails kann damit über eine Interaktion des Systems mit dem Emailserver erfolgen. Dies hat Auswirkungen auf das Szenario *Bücher ausleihen* (vgl. Abb. 4.6 und Abb. 7.1). Die Interaktionsskizzen *Bestätigung* und *Email Bestätigung* des ID

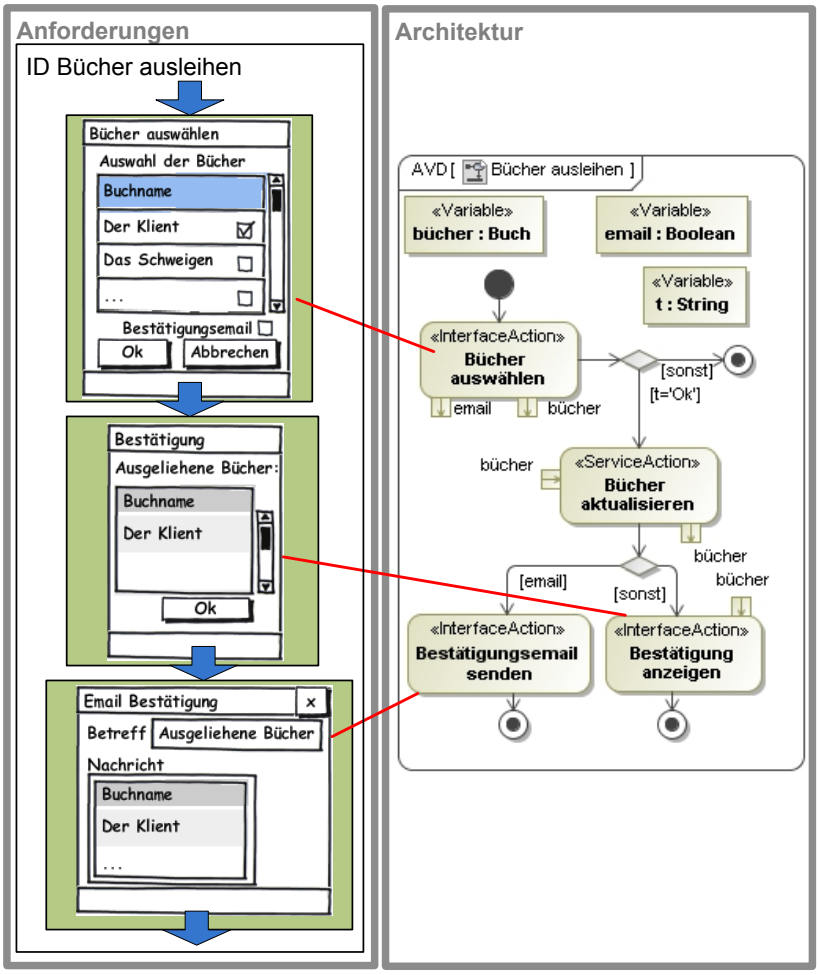
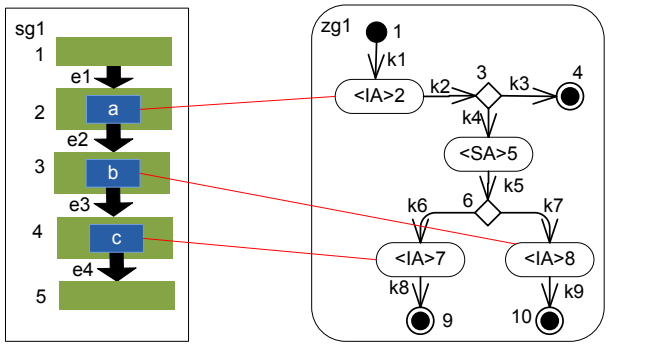


Abbildung 7.1: Änderungsszenario 1

Bücher ausleihen werden nacheinander angezeigt. Die Versendung der Email wird direkt nach der Dialogbestätigung durchgeführt. Diese Interaktionsskizzen werden von den Schnittstellenaktionen *Bestätigung anzeigen* resp. *Bestätigungsemail senden* realisiert.

Für die Konsistenzüberprüfung wird das Modell ID *Bücher ausleihen* und das Modell AVD *Bücher ausleihen* auf den Baum *sg1* resp. den Graph *zg1* nach dem in Kapitel

5 vorgestellten Verfahren abgebildet (siehe. Abb. 7.2). Die Interaktion *a* ist im Szena-



Formalisierung:
SG = {sg1, ...}
sg1=({1, 2, 3, 4, 5},
{e1, e2, e3, e4},
1, {5})
I_{ACT}=(a, b, c)
a= ({2}, {e2})
b= ({3}, {e3})
c= ({4}, {e4})

ZG = {zg1, ...}
zg1=({(1, y_{init}), (2, y_{ia}), (3, y_{dec}), (4, y_{final}),
(5, y_{sa}), (6, y_{dec}), (7, y_{ia}), (8, y_{ia}),
(9, y_{final}), (10, y_{final})},
{k1=(1, 2), k2, k3, k4, k4, k6, k7, k8, k9)

iaNode = {(a, (2, y_{ia})), (b, (8, y_{ia})),
(c, (7, y_{ia})),}

Abbildung 7.2: Abstrakte Syntax im Änderungsszenario 1

rienschritt *2* neu aktiv. Nach Definition 38 muss der Kante *e1* des szenarienbasierten Modells eine Kante ausgehend von einem initialen Knoten des zustandsbasierten Modells zugeordnet werden. Eine mögliche Zuordnung der Kante *e1* ist damit x=(e1, { k1 }, k1, { k1 }). Nach Definition 75 muss die Interaktion *a* über FI^{ref} mit einer Endkante einer Zuordnung in Beziehung gesetzt werden. Definition 77 legt zudem fest, dass die Interaktion über FI^{ref} mit der Kante in Beziehung gesetzt werden muss, die in den Interaktionsknoten *2* führt und *e1* zugeordnet ist. Wenn die Endkante *k1* über ((x, k1), a) $\in FI^{ref}$ mit der Interaktion *a* in Beziehung gesetzt wird, sind die Bedingungen erfüllt. Die Definition 78 ist in diesem Fall ebenfalls erfüllt, da weder eine Interaktion im Übergang zum Szenarienschritt *2* deaktiviert noch die mit *a* in Beziehung gesetzte Endkante über ε als beendet markiert wird. Die Konsistenzbedingungen sind damit an dieser Stelle erfüllt.

Das Szenario muss nach Definition 47 am Interaktionsknoten *2* fortgesetzt werden. Für die Kante *e2* existieren damit die folgenden drei Zuordnungsvarianten:

$$y1 = (e2, \{k2, k3\}, k2, \{k3\})$$
$$y2 = (e2, \{k2, k4, k5, k6\}, k2, \{k6\})$$
$$y3 = (e2, \{k2, k4, k5, k7\}, k2, \{k7\})$$

Die Interaktion *b* ist im Szenarienschritt *3* neu aktiv und ist über *iaNode* mit den Interaktionsknoten *8* verknüpft. Nach den Definitionen 75 und 77 muss sie damit über FI^{ref} mit einer Endkante der Zuordnung in Beziehung gesetzt werden, die in den Interaktionsknoten *8* führt. Da nur die Kante *k7* der Zuordnung *y3* in diesen Interaktions-

knoten führt, kann diese Bedingungen nur mit der Zuordnung *y3* erfüllt werden. Das Szenario muss dann nach Definition 47 am Interaktionsknoten *8* fortgesetzt werden. Für die Kante *e3* gibt es ausgehend von dieser Situation keine Zuordnung mit einer Endkante, die über FI^{ref} mit der neu aktivierten Interaktion *c* in Beziehung gesetzt werden könnte und die Definitionen 75 und 77 erfüllt. Das szenarienbasierte Modell ist damit Inkonsistent zum zustandsbasierten Modell. Übertragen auf das ID und AVD bedeutet dies, dass im Übergang vom Szenarienschritt *3* zum Szenarienschritt *4* die neu aktivierte Interaktionsskizze *Email Bestätigung* im AVD nicht erreicht werden kann.

7.1.2 Inkonsistenzbehebung

Die Inkonsistenz kann nun behoben werden, indem entweder das szenarienbasierte Anforderungsmodell oder das zustandsbasierte Architekturmodell geändert wird. In diesem Szenario einigen sich die Entwickler und Interessenvertreter darauf das Architekturmodell zu ändern. Die Aktion *Bestätigung anzeigen* des AVD *Bücher ausleihen* wird direkt nach der Aktion *Bücher auswählen* durchgeführt (vgl. Abb. 4.6 und 7.3). Die Schnittstellenaktion *Bestätigungsemail senden* wird anschließend optional durch-

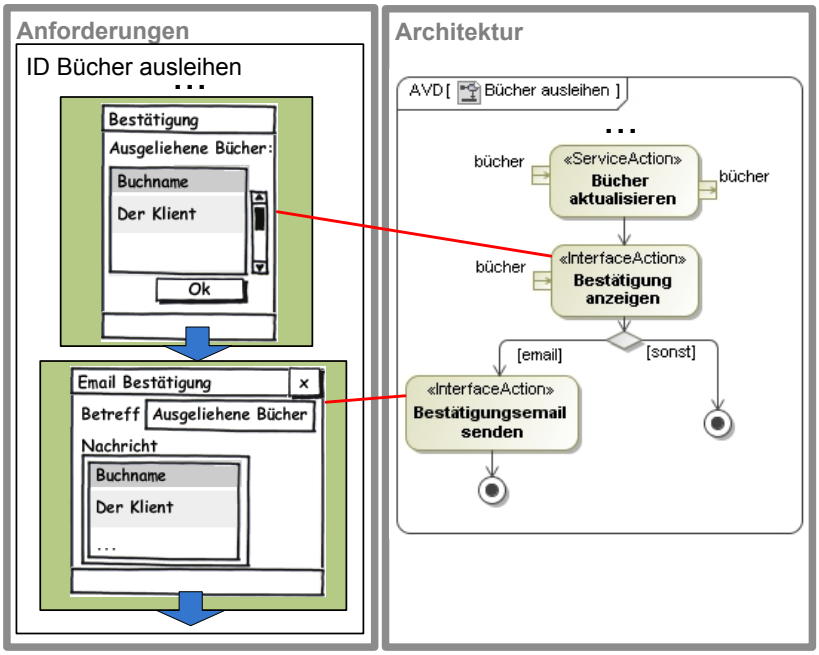


Abbildung 7.3: Behebung der Inkonsistenz im Anwendungsszenario 1

geführt. Für die Konsistenzüberprüfung kann die abstrakte Syntax des geänderten Modells auf den Baum *sg1'* und den Graph *zg1'* abgebildet werden (siehe Abb. 7.4). Ein Teil des zustandsbasierten Modells ist geändert. Der Knoten *6* ist nun beispielsweise vom Typ y_{ia}. Die einzige mögliche Zuordnung zur Kante *e1* ist weiterhin x=(e1, { k1 },

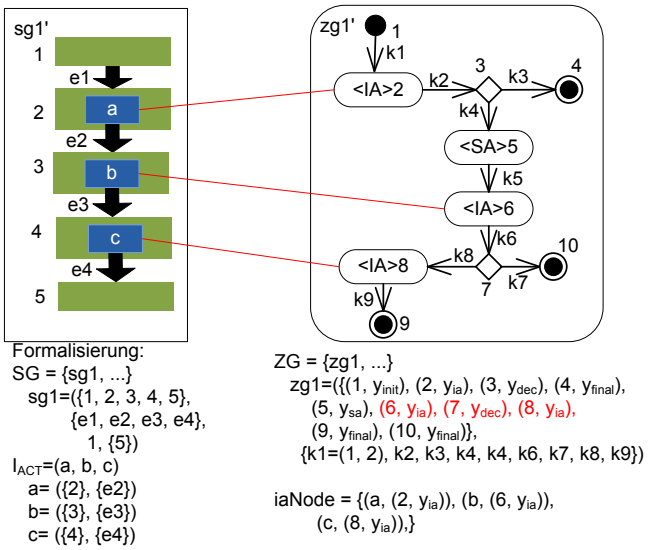


Abbildung 7.4: Abstrakte Syntax nach Inkonsistenzbehebung Änderungsszenario 1

k1, { k1 }). Das Szenario muss nach Definition 47 am Interaktionsknoten *2* fortgesetzt werden. Durch die Änderung des Architekturmodells existiert für die Kante *e2* nur die Zuordnung y=(e2, { k2, k4, k5 }, k2, { k5 }), die die Bedingungen der Definitionen 75 und 77 erfüllt. Die Interaktionen *b* muss dann über ((y, k5), b) $\in FI^{ref}$ mit der Endkante *k5* der Zuordnung *y* in Beziehung gesetzt sein. Nach Definition 47 muss das Szenario am Interaktionsknoten *6* fortgesetzt werden. Durch die Zuordnungen z=(e3, { k6, k8 }, k6, { k8 }) und z2=(e4, { k9 }, k9, { k9 }) werden die Konsistenzbedingungen erfüllt. Die Interaktion *c* muss dann über ((z, k8), c) $\in FI^{ref}$ in Beziehung gesetzt sein. Der Kontrollfluss der Kante *k8* wird über die Kante *k9* fortgesetzt und endet am finalen Knoten *9*.

7.2 Anwendungsszenario - Änderung der Architektur

7.2.1 Änderung der Architektur des Bibliothekssystems

Im zweiten Anwendungszenario erfolgt eine Änderung des Architekturmodells zum Bibliothekssystem. Die zugrunde liegenden Modelle wurden in Kapitel 4 eingeführt. Im Rahmen des Architekturentwurfs wurde festgestellt, dass eine Softwarekomponente existiert, die die gestellten Anforderungen nahezu erfüllt. Die Verwendung dieser Komponente würde die Entwicklungskosten erheblich senken. Leider hat die Verwendung dieser Komponente eine weitere Änderung des Ablaufs des AVD *Bücher ausleihen* zur Folge. Der Beginn des AVD bis zur Aktion *Bücher aktualisieren* bleibt unverändert. Die Durchführung der Schnittstellenaktionen *Bestätigungsemail senden* und *Bestätung anzeigen* erfolgt jedoch parallel zueinander (siehe Abb. 7.5). Es wird durch

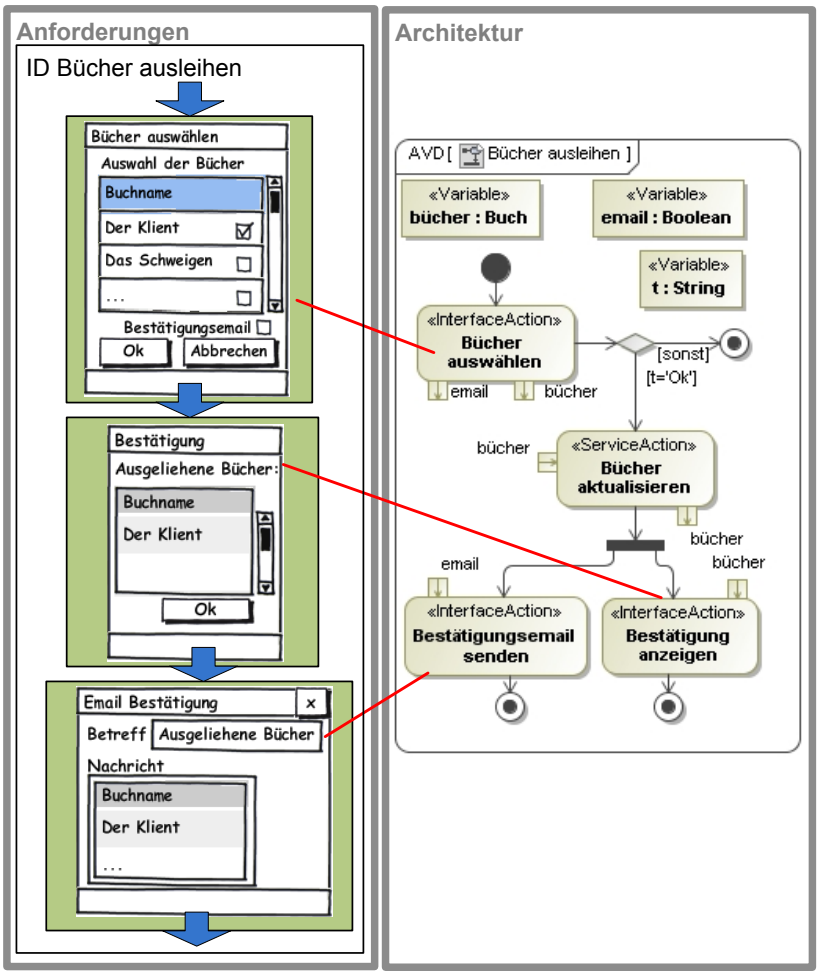


Abbildung 7.5: Änderungsszenario 2

die Architekten beschlossen, den Ablauf versuchsweise in einem Entwurf zu Ändern. Mit Hilfe der automatisierten Konsistenzüberprüfung werden die erforderlichen Anpassungen des Anforderungsmodells identifiziert. Abschließend werden diese mit den Interessenvertretern abgestimmt. Die Interessenvertreter sollen hierbei entscheiden, ob die Änderungen durch die Kostenersparnis wegen der Verwendung einer bestehenden Komponente vertretbar sind.

Für die Konsistenzüberprüfung des Modells erfolgt die Abbildung des szenarienbasierten und zustandsbasierten Modells auf die abtrakte Syntax nach den im Kapitel 5 vorgestellten Verfahren. Die abstrakte Syntax des szenarienbasierten Modells ID *Bücher ausleihen* wird durch den Baum *sg1* und die abstrakte Syntax des zustandsbasierten Modells AVD *Bücher ausleihen* wird durch den Graph *zg1* beschrieben (siehe.

Abb. 7.6). Die Interaktion *a* ist im Szenarienschritt *2* neu aktiv. Nach Definition 75

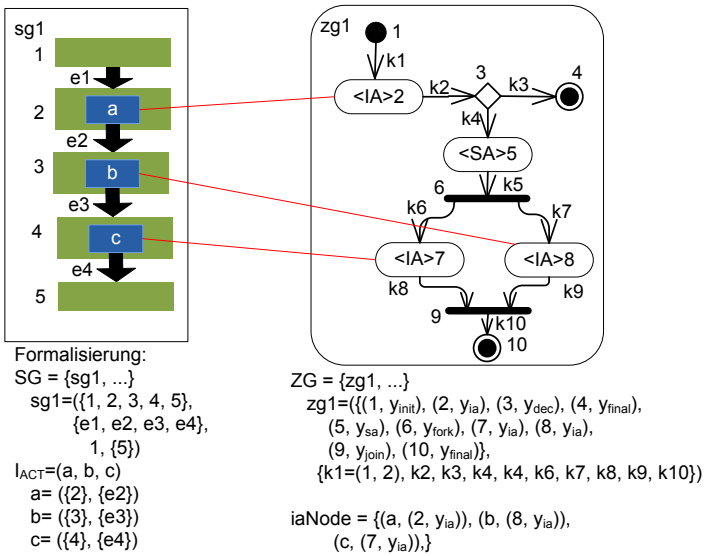


Abbildung 7.6: Abstrakte Syntax im Änderungsszenario 2

muss diese Interaktion über FI^{ref} mit einer Endkante einer Zuordnung in Beziehung gesetzt werden. Nach Definition 38 muss der Kante *e1* des szenarienbasierten Modells eine Kante ausgehend von einem initialen Knoten des zustandsbasierten Modells zugeordnet werden. Definition 77 legt fest, dass die Interaktion über FI^{ref} mit einer Kante in Beziehung gesetzt werden muss, die in den Interaktionsknoten *2* führt und *e1* zugeordnet ist. Die einzige zulässige Zuordnung der Kante *e1* ist damit x=(e1, { k1 }, k1, { k1 }). Die Endkante *k1* muss dann über ((x, k1), a) $\in FI^{ref}$ mit der Interaktion *a* in Beziehung gesetzt sein. Die Definition 78 ist in diesem Fall ebenfalls erfüllt, da weder eine Interaktion im Übergang zum Szenarienschritt *2* deaktiviert noch die mit *a* in Beziehung gesetzte Endkante über ε als beendet markiert wird. Die Konsistenzbedingungen sind damit an dieser Stelle erfüllt.

Das Szenario muss nach Definition 47 am Interaktionsknoten *2* fortgesetzt werden. Für die Kante *e2* existieren damit die folgenden zwei Zuordnungsvarianten:

$$y1 = (e2, \{k2, k3\}, k2, \{k3\})$$
$$y2 = (e2, \{k2, k4, k5, k6, k7\}, k2, \{k6, k7\})$$

Die Interaktion *b* ist im Szenarienschritt *3* neu aktiv und ist über *iaNode* mit den Interaktionsknoten *8* verknüpft. Nach den Definitionen 75 und 77 muss sie damit über ((y3, k7), b) $\in FI^{ref}$ mit der Endkante *k7* der Zuordnung *y2* in Beziehung gesetzt sein. Nach Definition 76 und 47 muss dann auch die Endkante *k6* der Zuordnung *y2* über FI^{ref} mit einer Interaktion in Beziehung gesetzt werden, die im Szenarienschritt *3* neu aktiv ist und über *iaNode* auf den Interaktionsknoten *7* verweist. Eine derartige Interaktion existiert nicht im szenarienbasierten Modell. Die Modelle sind damit Inkonsistent.

Übertragen auf das ID und AVD bedeutet dies, dass im Szenarienschritt *3* zur erreichten Schnittstellenaktion *Bestätigungsemail senden* keine passende Interaktionsskizze existiert.

7.2.2 Inkonsistenzbehebung

Mit Hilfe der Informationen der Konsistenzüberprüfung leiten die Architekten eine mögliche Anpassung des Anforderungsmodells ab, die die Inkonsistenz behebt. Das angepasste Anforderungsmodell wird anschließend mit den Interessenvertretern abgestimmt. Die Architekten beheben die Inkonsistenz, indem die Interaktionsskizze *Bestätigungsemail senden* im Szenario bereits parallel zur Interaktionsskizze *Bestätigung* aktiviert ist (siehe Abb. 7.7).

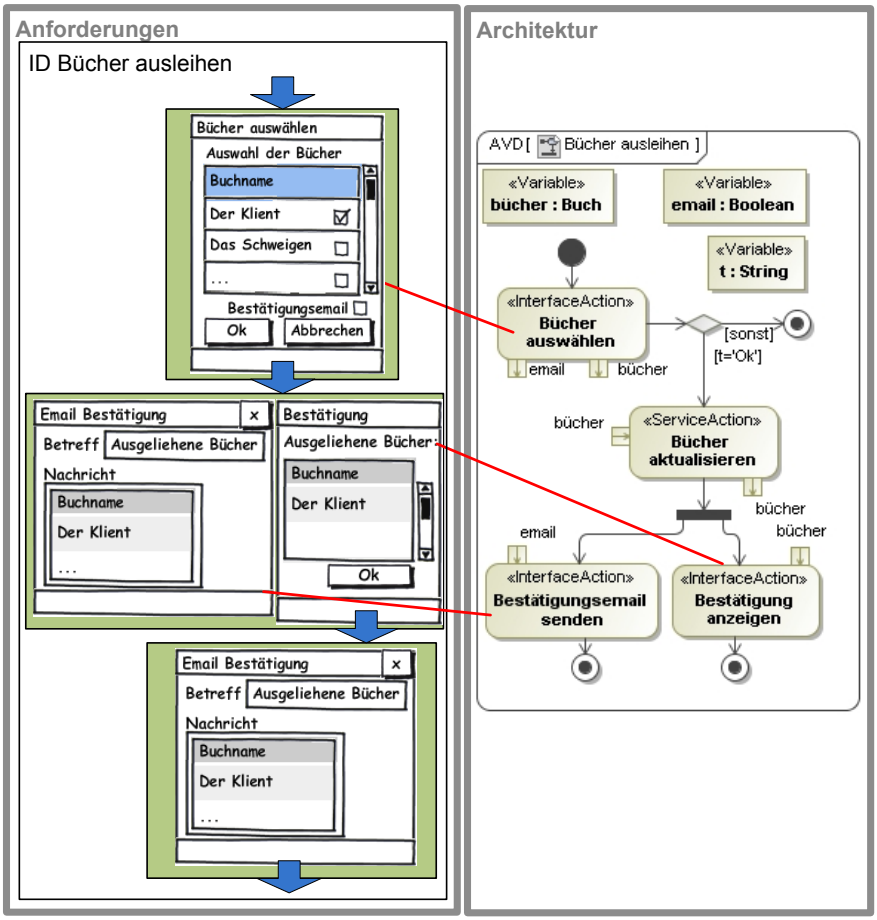


Abbildung 7.7: Behebung der Inkonsistenz im Anwendungsszenario 2

Nach Abschluss der Interaktion *Bestätigung* bleibt die Interaktion *Bestätigungsemail senden* aktiv. Die Interaktion *Bestätigungsemail senden* wird im folgenden Szenarienschritt beendet. Für die erneute Konsistenzüberprüfung kann die abstrakte Syntax des geänderten Modells auf den Baum *sg1'* und den Graph *zg1'* abgebildet werden (siehe Abb. 7.8). Ein Teil des szenarienbasierten Modells ist geändert. Der Knoten *3*

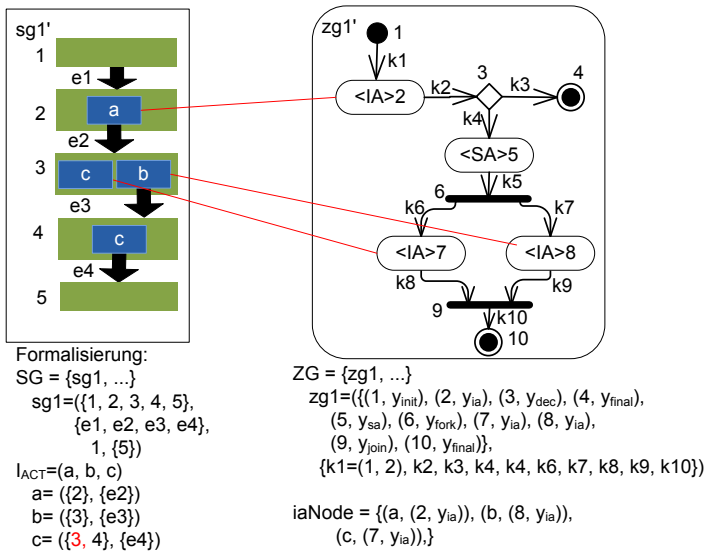


Abbildung 7.8: Abstrakte Syntax nach Inkonsistenzbehebung Änderungsszenario 2

befindet sich nun neben Knoten *4* ebenfalls in der Menge der Knoten, an dem die Interaktion *c* aktiv ist. Die einzige mögliche Zuordnung zur Kante *e1* ist weiterhin x=(e1, { k1 }, k1, { k1 }). Das Szenario muss nach Definition 47 weiterhin am Interaktionsknoten *2* fortgesetzt werden. Für die Kante *e2* existieren damit die folgenden zwei Zuordnungsvarianten:

$$y1 = (e2, \{k2, k3\}, k2, \{k3\})$$
$$y2 = (e2, \{k2, k4, k5, k6, k7\}, k2, \{k6, k7\})$$

Die Interaktion *b* ist im Szenarienschritt *3* neu aktiv und ist über *iaNode* mit den Interaktionsknoten *8* verknüpft. Nach den Definitionen 75 und 77 muss sie damit über ((y3, k7), b) $\in FI^{ref}$ mit der Endkante *k7* der Zuordnung *y2* in Beziehung gesetzt sein. Nach Definition 76 und 47 muss dann auch die Endkante *k6* der Zuordnung *y2* über FI^{ref} mit einer Interaktion in Beziehung gesetzt werden, die im Szenarienschritt *3* neu aktiv ist und über *iaNode* auf den Interaktionsknoten *7* verweist. Durch die Änderung im szenarienbasierten Modell kann nun die Endkante *k6* über ((y2, k6), c) $\in FI^{ref}$ mit der Interaktion *c* in Beziehung gesetzt werden, da *c* im Szenarienschritt *3* neu aktiv ist. Die Konsistenzbedingungen sind an dieser Stelle damit erfüllt.

Das Szenario muss nach den Definitionen 37 und 47 am Interaktionsknoten *8* fortgesetzt werden. Für die Kante *e3* ist die Zuordnung (e3, { k9 }, k9, { k9 }) zulässig.

Das Szenario muss nach den Definitionen 37 und 47 am Interaktionsknoten *7* fortgesetzt werden. Für die Kante *e4* ist die Zuordnung (e4, { k8, k10 }, k8, { k8 }) zulässig. Die Modelle sind konsistent. Das geänderte Anforderungsmodell kann mit den Interessenvertretern abgestimmt werden.

7.3 Anwendungsszenario - Anforderungserweiterung

7.3.1 Erweiterung der Anforderungen an das Bibliothekssystem

Im dritten Anwendungszenario erfolgt eine Erweiterung der Anforderungen an das Bibliothekssystem. Die zugrunde liegenden Modelle wurden im Kapitel 4 eingeführt. Die Interessenvertreter wollen sicher stellen, dass ein neu hinzugefügtes Buch auch in der Liste aller Bücher aufgenommen wird. Hierfür wird das Szenario ID *Buch hinzufügen* erweitert. Nach Bestätigung der Buchdaten wird erneut der Verwaltungsdialog mit der Übersicht über die Bücher angezeigt (siehe Abb. 7.9).

Für die Konsistenzüberprüfung des Modells erfolgt die Abbildung des szenarienbasierten und zustandsbasierten Modells auf die abtrakte Syntax nach den im Kapitel 5 vorgestellten Verfahren. Die abstrakte Syntax des szenarienbasierten Modells ID *Buch hinzufügen* wird durch den Baum *sg1* und die abstrakte Syntax der zustandsbasierten Modelle AVD *Bücher verwalten* und *Buch hinzufügen* wird durch die Graphen *zg1* und *zg2* beschrieben (siehe. Abb. 7.10). Die Interaktion *a* ist im Szenarienschritt *2* neu aktiv. Nach Definition 75 muss diese Interaktion über FI^{ref} mit einer Endkante einer Zuordnung in Beziehung gesetzt werden. Nach Definition 38 muss der Kante *e1* des szenarienbasierten Modells eine Kante ausgehend von einem initialen Knoten des zustandsbasierten Modells zugeordnet werden. Definition 77 legt fest, dass die Interaktion *a* über FI^{ref} mit einer Kante in Beziehung gesetzt werden muss, die in den Interaktionsknoten *3* führt und *e1* zugeordnet ist. Die einzige zulässige Zuordnung der Kante *e1* ist damit x=(e1, { k1, k2 }, k1, { k2 }). Die Endkante *k2* muss dann über ((x, k2), a) $\in FI^{ref}$ mit der Interaktion *a* in Beziehung gesetzt sein. Die Definition 78 ist in diesem Fall ebenfalls erfüllt, da weder eine Interaktion im Übergang zum Szenarienschritt *2* deaktiviert noch die mit *a* in Beziehung gesetzte Endkante über ε als beendet markiert wird. Die Konsistenzbedingungen sind damit an dieser Stelle erfüllt.

Das Szenario muss nach Definition 47 am Interaktionsknoten *3* fortgesetzt werden. Für die Kante *e2* existieren damit die folgenden zwei Zuordnungsvarianten:

$$y1 = (e2, \{k3, k4\}, k3, \{k4\})$$
$$y2 = (e2, \{k3, k5, c1, k7, k8, k9\}, k3, \{k9\})$$

Die Aufrufkante sind hierbei entsprechend der Zuordnungsbedingung in Definition 51 zugeordnet. Die Interaktion *b* ist im Szenarienschritt *3* neu aktiv und ist über *iaNode* mit den Interaktionsknoten *11* verknüpft. Nach den Definitionen 75 und 77 muss sie damit über ((y2, k9), b) $\in FI^{ref}$ mit der Endkante *k9* der Zuordnung *y2* in Beziehung gesetzt sein. Das Szenario muss nach Definition 47 am Interaktionsknoten *11* fortgesetzt

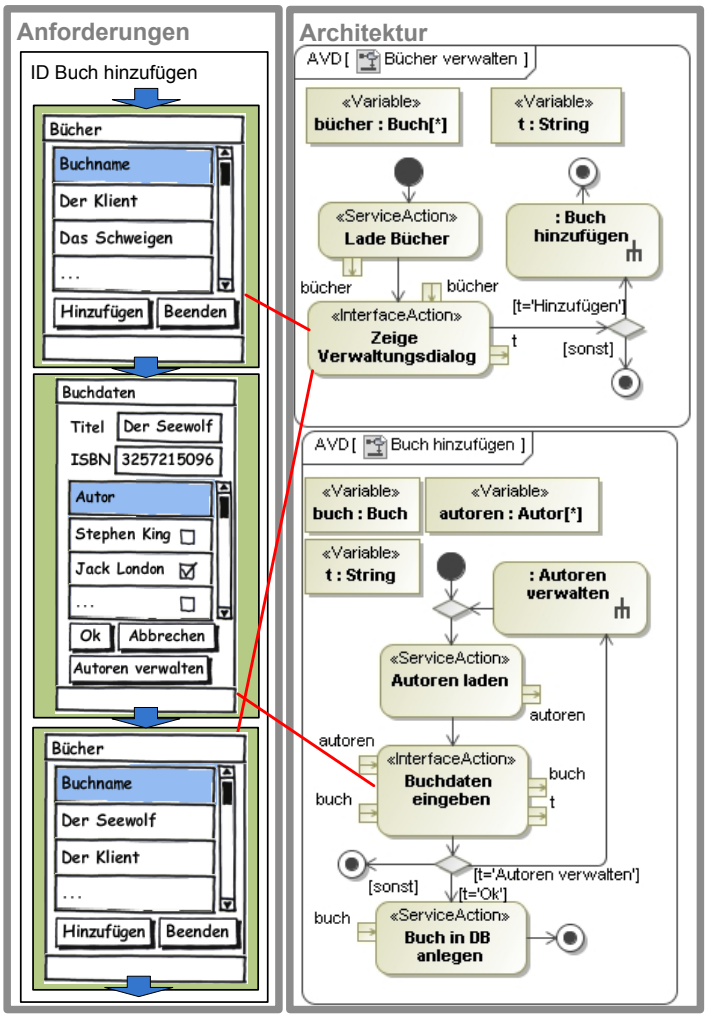


Abbildung 7.9: Änderungsszenario 3

werden. Für die Kante *e3* existieren damit die folgenden drei Zuordnungsvarianten:

$$z1 = (e3, \{k10, k11, r1, k6\}, k10, \{k6\})$$
$$z2 = (e3, \{k10, k12, k15, k8, k9\}, k10, \{k9\})$$
$$z3 = (e3, \{k10, k13, k14, r2, k6\}, k10, \{k6\})$$

Die Rückkerkanten sind hierbei entsprechend der Zuordnungsbedingung in Definition 57 zugeordnet. Die Interaktion c ist im Szenarienschritt *4* neu aktiv. Nach den Definitionen 75 und 77 muss diese Interaktion über FI^{ref} mit einer Endkante einer Zuordnung in Beziehung gesetzt werden, die in den Interaktionsknoten *3* führt. Keine der

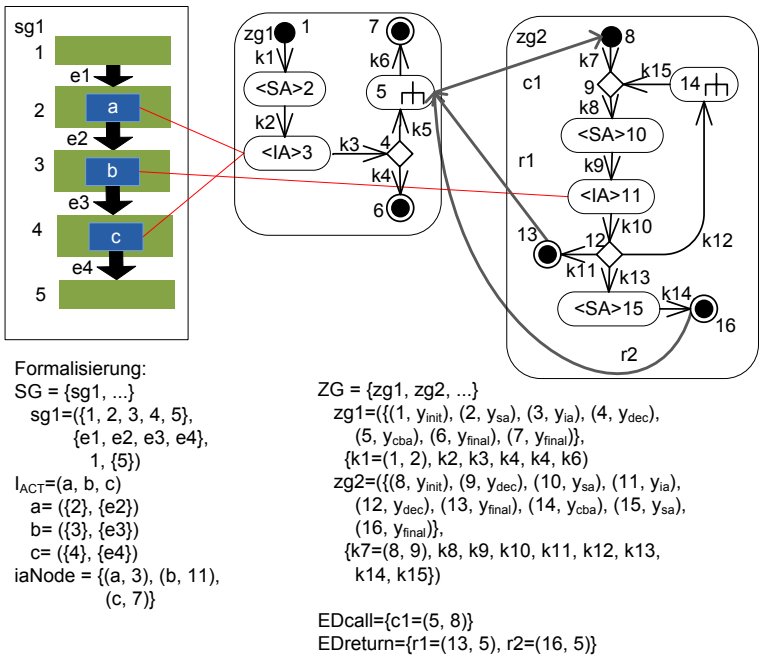


Abbildung 7.10: Abstrakte Syntax im Änderungsszenario 3

möglichen Zuordnungsvarianten hat eine Endkante, die in diesen Interaktionsknoten führt. Die Modelle sind damit Inkonsistent. Übertragen auf das ID und AVD bedeutet dies, dass die im Szenarienschritt *4* aktivierte Interaktionsskizze *Bücher* nicht erreicht werden kann.

7.3.2 Inkonsistenzbehebung

Die Inkonsistenz kann nun behoben werden indem entweder das szenarienbasierte Anforderungsmodell oder das zustandsbasierte Architekturmodell geändert wird. Das Szenario könnte beispielsweise um einen Szenarienschritt erweitert werden, bei dem die Funktion *Bücher verwalten* neu gestartet wird. In diesem Szenario einigen sich die Entwickler und Interessenvertreter darauf das Architekturmodell zu ändern. Die Architekten beheben die Inkonsistenz, indem eine Schleife im AVD *Bücher verwalten* eingefügt wird. Die Aktionen *Lade Bücher* und *Zeige Verwaltungsdialog* erfolgen erneut nach der Aufrufaktion *:Buch hinzufügen* (siehe Abb. 7.11). Die Syntaxeinschränkung in Definition 68 legt fest, dass keine zwei unterschiedlichen Pfade von einem Entscheidungsknoten zu einem Interaktionsknoten führen darf. Dies wird von den Architekten ebenfalls berücksichtigt. Daher erfolgt die Enscheidung im AVD *Buch hinzufügen*, ob das Ablegen des Buches in der Datenbank mit *Ok* bestätigt wurde, im Rahmen der Aktion *Buch in DB ablegen* (siehe Abb. 7.11).

Für die erneute Konsistenzüberprüfung kann die abstrakte Syntax des geänderten

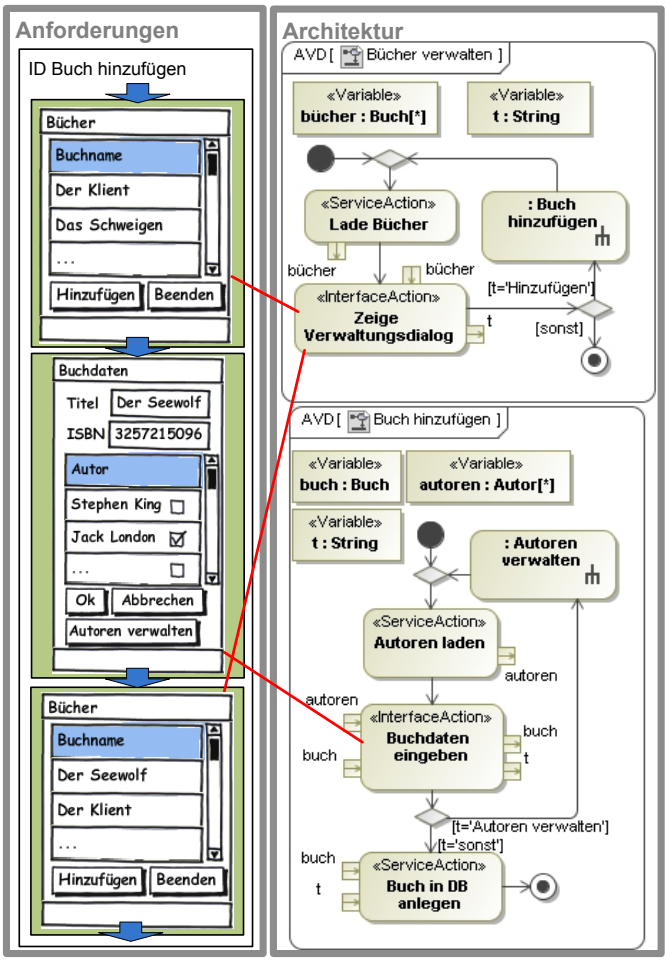


Abbildung 7.11: Behebung der Inkonsistenz im Anwendungsszenario 3

Modells auf den Baum *sg1'* und die Graphen *zg1'* und *zg2'* abgebildet werden (siehe Abb. 7.12). Ein Teil des zustandsbasierten Modells ist geändert. Der finale Knoten *7* ist jetzt ein Mischknoten in der Menge der Knoten des Graphs *zg1'*. Die Kante *k1* führt von dem initialen Knoten *1* zu diesem Mischknoten und die Kante *k17* vom Mischknoten zum allgemeinen Knoten *2*. Zudem ist der Knoten *13* genau so wie die Kante *k11* nicht weiter Teil des Graphs *zg2'*. Hierdurch ändern sich die erforderliche Zuordnung zur Kante *e1* geringfügig zu x=(e1, { k1, k17, k2 }, k1, { k2 }). Die erforderliche Zuordnung y2=(e2, { k3, k5 c1, k7, k8, k9 }, k3, { k9 }) zur Kante *e2* bleibt bestehen. Die möglichen Zuordnungen zu der Kante *e3* ändern sich wie folgt zu:

$$z1 = (e3, \{k10, k12, k15, k8, k9\}, k10, \{k9\})$$
$$z2 = (e3, \{k10, k13, k14, r2, k6, k17, k2\}, k10, \{k2\})$$

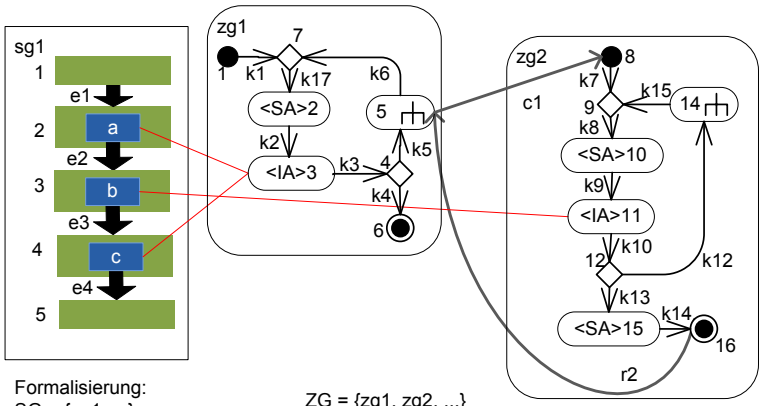


Abbildung 7.12: Abstrakte Syntax nach Inkonsistenzbehebung Änderungsszenario 3

Die Interaktion *c* ist im Szenarienschritt *4* neu aktiv. Nach den Definitionen 75 und 77 muss diese Interaktion über FI^{ref} mit einer Endkante einer Zuordnung in Beziehung gesetzt werden, die in den Interaktionsknoten *3* führt. Mit der Zuordnung *z2* können diese Bedingungen erfüllt werden. Die Interaktion *c* muss dann über ((z2, k2), c) $\in$ FI^{ref} mit der Kante *k2* der Zuordnung *z2* in Beziehung gesetzt sein. Das Szenario muss nach den Definitionen 37 und 47 am Interaktionsknoten *3* fortgesetzt werden. Mit der Zuordnung (e4, { k3, k4 }, k3, { k4 }) werden alle Konsistenzbedingungen erfüllt.

7.4 Test der Werkzeugunterstützung

Im Kapitel 6 werden die Anforderungen, die Architektur und die Implementierung der im Rahmen dieser Arbeit entwickelten Werkzeugunterstützung der automatisierten Konsistenzsicherung beschrieben. Die Entwicklungsumgebung *Eclipse* wird hierbei als Benutzeroberfläche für beispielsweise die Auswahl des zu prüfenden Modells verwendet. Das *EVL*-Modul führt die Prüfung des Modells durch. Die Implementierung der automatisierten Konsistenzsicherung erfolgt durch ein *EMF*-Metamodell für szenarienbasierte und zustandsbasierte Modelle sowie durch in *EVL* beschriebenen Syntaxeinschränkungen und Konsistenzbedingungen. In diesem Abschnitt wird die Werkzeugunterstützung anhand des in Kapitel 6 vorgestellten Szenarios getestet.

Die Modellierung des im Szenario aus Kapitel Kapitel 6 skizzenhaft beschriebenen Modells nach dem EMF-Metamodell wird in Abb. 7.13 links dargestellt (vgl. Abb. 6.1). Das szenarienbasierte Modell und das zustandsbasierte Modell sind hierbei je-

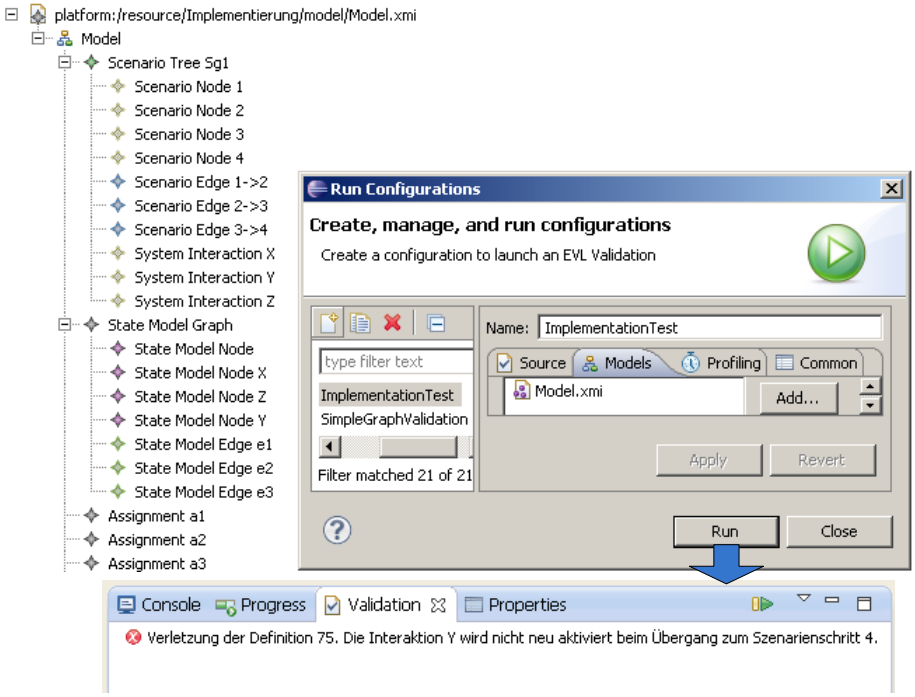


Abbildung 7.13: Eingabemodell für den Test

weils durch Objekte vom Typ *ScenarioTree* resp. *StateModelGraph* modelliert. Die Zuordnungen werden durch Objekte vom Typ *Assignments* beschrieben. Die Abb. 7.13 zeigt die Durchführung der Konsistenzüberprüfung. Der obere Bildschirmausschnitt zeigt die Auswahl des Modells in der Datei *Modell.xmi* zur Prüfung und die Auswahl der Funktion *Run*. Im unteren Bildschrimausschnitt wird das Ergebnis der Prüfung angezeigt. Das Ergebnis ist, dass die Definition 75 im Modell verletzt ist. Da der Interaktionsknoten Y vom Kontrollfluss im dritten Szenarienschritt nicht erreicht werden kann, ist dies das erwartete Ergebnis.

7.5 Ergebnis der Evaluation

Das Ziel der Evaluation ist die Prüfung der Einsetzbarkeit des Ansatzes zur Erkennung und Behebung von Inkonsistenzen. Die Konsistenzsicherung ist insbesondere in den frühen Phasen entscheidend, da eine Korrektur hier entstandener Fehler in späten Phasen mit hohen Kosten verbunden sind. Bei Einsatz des Ansatzes zur Konsistenzsicherung der Verhaltensmodelle in frühen Phasen bei den drei vorgestellten Anwend-

ngsszenarien konnte folgendes festgestellt werden:

- Inkonsistenzen, die durch Änderungen in den Anwendungsszenarien entstanden sind, konnten mit Hilfe des Ansatzes aufgedeckt und behoben werden. Der Ansatz zur Konsistenzsicherung erlaubt sowohl Änderungen an Anforderungen als auch an Architekturen vorzunehmen. Hierdurch wird eine iterative und evolutionäre Entwicklung von Anforderungen und Architekturen unterstützt, die in der Praxis häufig der Fall ist. In den Anwendungsszenarien 1 und 3 wurden Änderungen an den Anforderungen vorgenommen. Im Anwendungsszenario 2 wurde die Architektur geändert.

- Der Ansatz konnte hierbei unter anderem auch bei der Beschreibung paralleler Abläufe, Hierarchien, Variablen und Schleifen eingesetzt werden. Variablen wurden in allen Anwendungsszenarien verwendet. Die Beschreibung eines parallelen Ablaufs erfolgte im Anwendungsszenario 2. Schleifen und Hierarchien wurden im Anwendungsszenario 3 verwendet.

- Das Ergebnis der Behebung der entdeckten Inkonsistenzen ist die vollständige Konsistenz der szenarienbasierten Anforderungsmodelle und zustandsbasierten Architekturmodelle in der frühen Phase.

Durch eine Automatisierung der Konsistenzüberprüfung nach den vorgestellten Ansatz kann damit das Risiko inkonsistenter Anforderungen und Architekturen gesenkt werden. Beim Test der Werkzeugunterstützung durch Prüfung inkonsistenter szenarienbasierter und zustandsbasierter Modelle wurden die Inkonsistenzen automatisiert aufgedeckt. Damit ist auch die Machbarkeit einer Implementierung des Konsistenzsicherungsverfahrens gezeigt.

Kapitel 8

Zusammenfassung und Ausblick

8.1 Zusammenfassung

In dieser Arbeit wurde ein Ansatz zur Automatisierung der Konsistenzsicherung von Anforderungen und Architekturen entwickelt. Die Anforderungserhebung und der Architekturentwurf eines Softwareentwicklungsprojekts sind für die erfolgreiche Entwicklung hochqualitativer Softwaresysteme von besonderer Wichtigkeit. Das Ziel des Entwurfs ist die Entwicklung einer Architektur, die die gestellten Anforderungen an das Softwaresystem erfüllt. Anforderungen und Architekturen werden in der Realität zumeist iterativ und evolutionär entwickelt [56]. Ein fundamentales Problem hierbei ist die Entstehung von Inkonsistenzen, die zu einer fehlerhaften Berücksichtigung von Anforderungen und folglich zu unerfüllten Anforderungen führen. Aktuelle modellbasierte Ansätze erlauben eine eindeutige und formale Beschreibung von Anforderungen und Architekturen. Eine Automatisierung der Konsistenzsicherung dieser Modelle kann das Problem der Entstehung von Inkonsistenzen lösen. In dieser Arbeit wurde der modellbasierte Ansatz CREATE [40] im Detail eingeführt, der eine eindeutige und formale Beschreibung von Anforderungen und Architekturen erlaubt. Eine automatische Konsistenzsicherung der Strukturmodelle kann auf eine einfache Weise durch ein Metamodell mit zusätzlichen formalen Konsistenzbedingungen erfolgen, da die Strukturmodelle auf Typebene miteinander in Beziehung stehen [74]. Eine Automatisierung der Konsistenzsicherung der Verhaltensmodelle, insbesondere bei einer vollständigen Berücksichtigung ihrer Ausführungssemantik, ist eine Herausforderung.

In dieser Arbeit wurde ein Ansatz zur Automatisierung der Konsistenzsicherung der Verhaltensmodelle mit vollständiger Berücksichtigung der Ausführungssemantik vorgestellt. Für die Beschreibung von Anforderungen sind szenarienbasierte Verhaltensbeschreibungen wie beispielsweise Sequenzdiagramme geeignet. Zustandsbasierte Verhaltensbeschreibungen wie beispielsweise höhere Petrinetze eignen sich für den Architekturentwurf. Eine Automatisierung der Konsistenzsicherung von Anforderungen und Architekturen kann folglich durch eine Automatisierung der Konsistenzsicherung von szenarien- und zustandsbasierten Modellen erreicht werden [74]. Eine Konsistenzsicherung derartiger Modelle ist problematisch. Zwischen den Verhaltensbeschreibun-

gen besteht eine Verfeinerungsbeziehung. Zudem ist häufig eine Turing-vollständige Sprache für die Beschreibung der Architektur erforderlich. In dieser Arbeit wurde eine Lösung vorgestellt, mit der eine entscheidbare und effiziente Konsistenzüberprüfung durch Syntaxeinschränkungen der Verhaltensmodelle ermöglicht wird. Durch die Einschränkung der Syntax wird die Ausdrucksmächtigkeit der Modelle nicht verringert. Es werden lediglich die Variationen zur Beschreibung eines Sachverhalts eingeschränkt und Regeln für den Zusammenhang der szenarienbasierten und zustandsbasierten Modelle festgelegt. In dieser Arbeit wurden folgende Ergebnisse erzielt:

- Im Rahmen der Arbeit wurde ein konkretes und ein allgemeines szenarienbasiertes Verhaltensmodell für die Beschreibung von Anforderungen und ein zustandsbasiertes Verhaltensmodell für die Beschreibung von Architekturen vorgestellt (siehe Kapitel 3 und 5.2). Die allgemeinen Verhaltensmodelle wurden als Bäume resp. Graphen beschrieben (siehe Kapitel 5.2). Diese Modelle haben die erforderliche Ausdrucksmächtigkeit für die Beschreibung von Anforderungen und Architekturen sowie der Automatisierung der Konsistenzsicherung im Detail. Zudem wurden Regeln für den Zusammenhang dieser Modelle in Form von Zuordnungen von Kanten des zustandsbasierten Modells zu Kanten des szenarienbasierten Modells definiert (siehe Kapitel 5.4).
- Syntaxeinschränkungen wurden definiert, die eine entscheidbare und effiziente Konsistenzüberprüfung ermöglichen (siehe Kapitel 5.5). Hierbei wurde die Ausdrucksmächtigkeit der Modelle nicht verringert. Vielmehr wurden die Variationen zur Beschreibung eines Sachverhalts eingeschränkt.
- Konsistenzbedingungen zwischen dem szenarienbasierten Anforderungsmodell und dem zustandsbasierten Architekturmodell wurden definiert, die die Ausführungssemantik dieser Modelle vollständig berücksichtigen und effizient geprüft werden können (siehe Kapitel 5.6).

Bei einer abschließenden Betrachtung wurde gezeigt, dass dieser Ansatz zur Automatisierung der Konsistenzsicherung von Anforderungen und Architekturen geeignet ist. Zudem wurde eine mögliche Implementierung der werkzeuggestützten Konsistenzsicherung vorgestellt.

8.2 Ausblick

Der in dieser Arbeit im Detail vorgestellte modellbasierte Ansatz CREATE für die Beschreibung von Anforderungen und Architekturen ist domänenspezifisch für interaktive Informationssysteme wie webbasierte Systeme und moderne Kommunikationssysteme. Für eine Beschreibung von Anforderungen und Architekturen anderer Systemtypen ist zu analysieren, welche Anpassungen an CREATE erforderlich sind. Beispielsweise wäre für eingebette Systeme wie die Steuereinheit eines Roboters zu analysieren in welcher Form Nachrichten bzw. Interaktionen für die Modellierung des Verhaltens geeignet sind. Die Bedeutung von Nachrichten zur Beschreibung von beispielsweise der Bewegung eines Roboterarms müsste hierbei geklärt werden. Des Weiteren berücksichtigt CREATE im Wesentlichen funktionale Anforderungen. Querschnittliche

nicht-funktionale Anforderungen werden nicht im Detail betrachtet und können aktuell nicht formal beschrieben werden. Damit kann eine zukünftige Arbeit die automatisierte Konsistenzsicherung nicht-funktionaler Anforderungen im Detail sein. Ein weiteres Thema für zukünftige Arbeiten ist die Konsistenzsicherung der formalen und textbasierten Anforderungen. Textbasierte Anforderungen werden in CREATE durch die HRL beschrieben. Zwischen den Substantiven und den Typen der Domänenstruktur des DSD sind aktuell nur einfache Querbezüge definiert. Eine zukünftige Arbeit könnte daher eine weitergehende Automatisierung der Konsistenzsicherung zwischen den textbasierten Anforderungen der HRL und den szenarienbasierten Anforderungen des formalen DSD und des SD sowie des ID sein. Des Weiteren handelt es sich bei der in CREATE beschriebenen Architektur um eine Grobarchitektur. Eine Feinarchitektur, die die Klassen und Methoden des Softwaresystems definieren wird aktuell nicht betrachtet. Raum für zukünftige Arbeiten besteht daher bei der Automatisierung der Konsistenzsicherung zwischen Anforderungen und Architekturen mit Berücksichtigung der Feinarchitektur.

Der in dieser Arbeit vorgestellte Ansatz zur automatisierten Konsistenzüberprüfung von szenarienbasierten Anforderungsmodellen und zustandsbasierten Architekturmodellen unterstützt Turing-vollständige architektonische Verhaltensmodelle. Konstrukte wie Variablen, Schleifen und parallele Abläufe können verwendet werden genau so wie Hierarchien. Weitere Konstrukte wie beispielsweise unterbrechbare Bereiche oder Signale wurden jedoch im Rahmen dieser Arbeit nicht betrachtet. Diese Sprachkonstrukte können die Modellierung einiger Sachverhalte erleichtern. Zudem kann hierdurch die architektonische Verhaltensbeschreibung an die Beschreibung der Implementierung angenähert werden. In einer zukünftigen Arbeit könnte daher untersucht werden, ob und wie diese Konstrukte bei Beibehaltung der Berechenbarkeit oder gar der Effizienz der Konsistenzüberprüfung unterstützt werden können. Ein weiteres Thema für zukünftige Arbeiten kann die Untersuchung einer weitergehenden Automatisierung der Konsistenzsicherung sein. In dieser Arbeit wurde die Konsistenzüberprüfung als grundlegendes Verfahren zur automatischen Konsistenzsicherung gewählt, da unter anderem durch das Turing-vollständige architektonische Verhaltensmodell die Anzahl der manuellen Entscheidungen stark mit der Modellgröße ansteigt. In einer zukünftigen Arbeit könnte untersucht werden, ob und wie weit die Anzahl der manuellen Entscheidungen eingeschränkt werden kann. Ließe sich die Anzahl begrenzen, könnte durch eine bidirektionale Transformation auch die Anpassung der Modelle zur Herstellung der Konsistenz automatisiert werden.

Zusammenfassend wurde in dieser Arbeit ein wichtiger Aspekt der Konsistenzsicherung von Anforderungen und Architekturen behandelt. Für eine vollumfängliche Automatisierung der Konsistenzsicherung von Anforderungen und Architekturen im Bereich des *Software Engineering* sind in Zukunft jedoch weitere Untersuchungen und Entwicklungen durchzuführen.

Literaturverzeichnis

[1] A. Agrawal, G. Karsai, and F. Shi. Graph transformations on domain-specific models. Technical report, Institute for Software Integrated Systems, Vanderbilt University, ISIS-03-403, 2003.

[2] D. Akehurst and S. Kent. A relational approach to defining transformations in a metamodel. In *Proceedings of the International Conference on The Unified Modeling Language*, pages 243–258. Springer, 2002.

[3] Y. Aoki, H. Okuda, S. Matsuura, and S. Ogata. Data lifecycle verification method for requirements specifications using a model checking technique. In *Proceedings of the International Conference on Software Engineering Advances*, pages 194–200. IARIA, 2013.

[4] A. Appel, S. Herold, H. Klus, and A. Rausch. Modelling the cocome with disccomp. In *CoCoME*, volume 5153 of *Lecture Notes in Computer Science*, pages 267–296. Springer, 2007.

[5] C. Atkinson, J. Bayer, and D. Muthig. Component-based product line development: The kobra approach. In *Proceedings of the Software Product Line Conference*, pages 289–309. Kluwer Academic Publishers, 2000.

[6] J. Barnat, L. Brim, I. Cerna, P. Moravec, P. Rockai, and P. Simecek. Divine – a tool for distributed verification. In *Computer Aided Verification*, volume 4144 of *Lecture Notes in Computer Science*, pages 278–281. Springer Berlin Heidelberg, 2006.

[7] S. Becker, S. Herold, S. Lohmann, and B. Westfechtel. A graph-based algorithm for consistency maintenance in incremental and interactive integration tools. In *Proceedings of the Software & Systems Modeling*, pages 287–315. Springer, 2007.

[8] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. Uppaal – a tool suite for automatic verification of real-time systems. In *Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science*, pages 232–243. Springer Berlin Heidelberg, 1996.

[9] T. L. Bennett and P. Wennberg. Eliminating embedded software defects prior to integration test. *Crosstalk, Journal of Defence Software Engineering*, pages 13–18, 2005.

[10] B. Böhm. A spiral model of software development and enhancement. *Computer*, 21(5):61–72, 1988.

[11] Y. Bontemps and P. Heymans. As fast as sound lightweight formal scenario synthesis and verification. In *Proceedings of the International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools*, pages 27–34. IET, 2004.

[12] F. Bordeleau, J. P. Corriveau, and B. Selic. A scenario-based approach to hierarchical state machine design. In *Proceedings of the IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2000*, pages 78–85. IEEE, 2000.

[13] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (xml) 1.0 (fifth edition). World Wide Web Consortium, Recommendation REC-xml-20081126, November 2008.

[14] R. Chitchyan, M. Pinto, A. Rashid, and L. Fuentes. Compass: Composition-centric mapping of aspectual requirements to architecture. In *Transactions on Aspect-Oriented Software Development IV*, volume 4640 of *Lecture Notes in Computer Science*, pages 3–53. Springer, 2007.

[15] Y. Choi and C. Bunse. Behavioral consistency checking for component-based software development using the kobra approach. In *Proceedings of the Workshop on Consistency Problems in UML-based Software Development: Understanding and Usage of Dependency*, pages 77–90. Citeseer, 2004.

[16] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT press, 1999.

[17] P. C. Clements. A survey of architecture description languages. In *Proceedings of the International Workshop on Software Specification and Design*, pages 16–25. IEEE Computer Society, 1996.

[18] A. Cobham. The intrinsic computational difficulty of functions. In *Proceedings of the International Congress on Logic, Methodology and Philosophy of Science*. North-Holland, 1965.

[19] Compuware Corporation and Sun Microsystems. *MOF Query/Views/Transformation, Revised Submission, OMG Document ad/03-08-07*, 2003.

[20] I. Crnkovic. Component-based software engineering - new challenges in software development. *Software Focus*, 2(4):127–133, 2001.

[21] K. Czarnecki and S. Helsen. Classification of model transformation approaches. In *Proceedings of the Workshop on Generative Techniques in the Context of Model-Driven Architecture*, pages 1–17, 2003.

[22] Z. Diskin, Y. Xiong, and K. Czarnecki. From state- to delta-based bidirectional model transformations: the asymmetric case. *Journal of Object Technology*, 10:6:1–25, 2011.

[23] Z. Diskin, Y. Xiong, K. Czarnecki, H. Ehrig, F. Hermann, and F. Orejas. From state- to delta-based bidirectional model transformations: the symmetric case. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*, Model Driven Engineering Languages and Systems, pages 304–318. Springer, 2011.

[24] M. Elkoutbi. Automated prototyping of user interfaces based on uml scenarios. *Journal of Automated Software Engineering*, 13(1):5–40, 2006.

[25] M. Elkoutbi and R. K. Keller. User interface prototyping based on uml scenarios and high-level petri nets. In *Proceedings of the International Conference Application and Theory of Petri Nets*, pages 166–186. Springer-Verlag, 2000.

[26] J. Esparza and M. Nielsen. Decidability issues for petri nets. *Petri nets newsletter*, 94:5–23, 1994.

[27] J. Fernandes, S. Tjell, J. Jorgensen, and O. Ribeiro. Designing tool support for translating use cases and uml 2.0 sequence diagrams into a coloured petri net. In *Proceedings of the International Workshop on Scenarios and State Machines*, pages 2–12, 2007.

[28] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems*, page 17, 2007.

[29] V. Garousi, L. Briand, and Y. Labiche. Control flow analysis of uml 2.0 sequence diagrams. In *Proceedings of the Conference on Model Driven Architecture Foundations and Applications*, volume 3748 of *Lecture Notes in Computer Science*, pages 160–174. Springer Berlin Heidelberg, 2005.

[30] H. Genrich and K. Lautenbach. System modelling with high-level petri nets. *Theoretical Computer Science*, 13(1):109–135, 1981.

[31] P. Grünbacher, A. Egyed, E. Egyed, and N. Medvidovic. Reconciling software requirements and architectures with intermediate models. In *Proceedings of the Conference Software and Systems Modeling*, pages 202–211. Springer, 2003.

[32] S. Haddad and D. Poitrenaud. Theoretical aspects of recursive petri nets. In *Proceedings of the International Conference Application and Theory of Petri Nets*, pages 228–247, 1999.

[33] D. Harel. Statecharts: A visual formalism for complex systems. *Journal Science of Computer Programming*, 8(3):231–274, 1987.

[34] D. Harel, H. Kugler, and A. Pnueli. Synthesis revisited: Generating statechart models from scenario-based requirements. In *Proceedings of the Conference Formal Methods in Software and Systems Modeling*, Lecture Notes in Computer Science, pages 309–324. Springer Berlin Heidelberg, 2005.

[35] R. Harper. Introduction to standard ml. Technical report, Laboratoty for the Foundations of Computer Science, Edinburgh University, Scotland, 1986.

[36] T. Hettel, M. Lawley, and K. Raymond. Model synchronisation: Definitions for round-trip engineering. In *Proceedings of the International Conference on Theory and Practice of Model Transformations*, Lecture Notes in Computer Science, pages 31–45. Springer Berlin Heidelberg, 2008.

[37] G. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[38] S. Holzner. *Eclipse*. O'Reilly, 2004.

[39] P. Huber, K. Jensen, and R. Shapiro. Hierarchies in coloured petri nets. In *Advances in Petri Nets*, volume 483 of *Lecture Notes in Computer Science*, pages 313–341. Springer Berlin / Heidelberg, 1991.

[40] M. Ibe, M. Vogel, B. Schindler, and A. Rausch. Create: A co-modeling approach for scenario-based requierements and component-based architectures. In *Proceedings of the International Conference on Software Engineering Advances*, pages 220–227. IARIA, 2013.

[41] ITU. *Message Sequence Chart (MSC), Recommendation Z.120*, 1999.

[42] K. Jensen. Coloured petri nets: a high level language for system design and analysis. In *Proceedings on Advances in Petri nets*, volume 483 of *Lecture Notes in Computer Science*, pages 342–416. Springer-Verlag New York, Inc., 1991.

[43] K. Jensen and L. M. Kristensen. *Coloured petri nets*. Springer, 2009.

[44] R. Kazman, G. Abowd, L. Bass, and P. Clements. Scenario-based analysis of software architecture. *Software, IEEE*, 13(6):47–55, 1996.

[45] I. Khriss, M. Elkoutbi, and R. K. Keller. Automatic synthesis of behavioral object specifications from scenarios. *Journal of Integrated Design & Process Science*, 5(3):53–77, 2001.

[46] A. G. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[47] C. Knieke and U. Goltz. An executable semantics for uml 2 activity diagrams. In *Proceedings of the International Workshop on Formalization of Modeling Languages*, pages 11–15. ACM Press, 2010.

[48] D. Kolovos, L. Rose, A. Garcia-Dominguez, and R. Paige. *The Epsilon Book*. Addison-Wesley, 2013.

[49] I. Krüger, R. Grosu, P. Scholz, and M. Broy. From mscs to statecharts. In *Proceedings of the International Workshop on Distributed and parallel embedded systems*, pages 61–71. Kluwer Academic Publishers, 1999.

[50] C. Lakos. The object orientation of object petri nets. In *Proceedings of Workshop on Object Oriented Programming and Models of Concurrency*, pages 2–7. Springer, 1995.

[51] H. Liang, J. Dingel, and Z. Diskin. A comparative survey of scenario-based to state-based model synthesis approaches. In *Proceedings of the International Workshop on Scenarios and state machines: models, algorithms, and tools*, pages 5–12. ACM, 2006.

[52] T. Maier and A. Zündorf. The fujaba statechart synthesis approach. In *Proceedings of the International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools*. Wiley-VCH, 2003.

[53] D. Michel and A. Reniers. Message sequence chart: Syntax and semantics. Technical report, Faculty of Mathematics and Computing, 1998.

[54] C. Nebut, F. Fleurey, Y. Le-Traon, and J.-M. Jezequel. Automatic test generation: a use case driven approach. *IEEE Transactions on Software Engineering*, 32(3):140–155, 2006.

[55] D. Niebuhr and A. Rausch. Guaranteeing correctness of component bindings in dynamic adaptive systems. In *Proceedings of the Conference on Software Engineering and Advanced Applications*, pages 454–457. ACM, 2009.

[56] B. Nuseibeh. Weaving together requirements and architectures. *Computer*, 34(3):115–117, 2001.

[57] B. Nuseibeh and S. Easterbrook. Requirements engineering: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 35–46. ACM Press, 2000.

[58] Object Management Group. *OMG Unified Modeling Language (OMG UML) Superstructure, version 2.2, number formal/2009-02-02*, 2009.

[59] Object Management Group. *OMG Systems Modeling Language (OMG SysML), version 1.3, number formal/2012-06-01*, 2010.

[60] Object Management Group. *MOF 2.0 Query/View/Transformation Specification, version 1.1, number formal/2011-01-01*, 2011.

[61] Object Management Group. *OMG Meta Object Facility (MOF) Core Specification, version 2.4.1, number formal/2013-06-01*, 2013.

[62] B.-U. Pagel and H.-W. Six. *Software Engineering, Band 1: Die Phasen der Softwareentwicklung*. Addison-Wesley, 1994.

[63] J. L. Peterson. Petri nets. *ACM Computing Surveys*, 9(3):223–252, 1977.

[64] J. L. Peterson. A note on colored petri nets. *Information Processing Letters*, 11(1):40–43, 1980.

[65] R. Petrasch and O. Meimberg. *Model Driven Architecture Eine praxisorientierte Einführung in die MDA*. dpunkt.verlag, 2006.

[66] C. A. Petri. *Kommunikation mit Automaten*. Universität Hamburg, 1962.

[67] A. Rausch and D. Niebuhr. Erfolgreiche it-projekte mit dem v-modell xt. *OBJEKTspektrum*, 3, 2005.

[68] A. Rausch, R. Reussner, R. Mirandola, and F. Plasil. *The Common Component Modeling Example: Comparing Software Component Models*. Lecture Notes in Computer Science. Springer, 2008.

[69] M. Richters and M. Gogolla. On formalizing the uml object constraint language ocl. In *Proceedings of the International Conference on Conceptual Modeling*, pages 449–464. Springer, 1998.

[70] M. Riebisch, I. Philippow, and M. Götze. Uml-based statistical test case generation. In *Proceedings of the International Conference on Objects, Components, Architectures, Services, and Applications for a Networked World*, Lecture Notes in Computer Science, pages 394–411. Springer Berlin Heidelberg, 2003.

[71] W. W. Royce. Managing the development of large software systems: concepts and techniques. In *Proceedings of the 9th international conference on Software Engineering*, pages 1–9. IEEE Computer Society Press, 1970.

[72] A.-E. Rugina, K. Kanoun, and M. Kaâniche. A system dependability modeling framework using aadl and gspns. In *Architecting Dependable Systems IV*, pages 14–38. Springer, 2007.

[73] C. Rupp, S. Queins, and B. Zengler. *UML 2 glasklar: Praxiswissen für die UML-Modellierung*. Hanser, 2007.

[74] B. Schindler and A. Rausch. Automatic consistence maintenance of requirements and architectures. In *IASTED International Conference on Software Engineering*, pages 15–22. ACTA Press, 2014.

[75] K. Schneider. Generating fast feedback in requirements elicitation. In *Proceedings of the 13th international working conference on Requirements engineering: foundation for software quality*, pages 160–174. Springer-Verlag, 2007.

[76] A. E. F. Seghrouchni, S. Haddad, A. El, and F. Seghrouchni. A recursive model for distributed planning. In *Proceedings of the International Conference on Multi-Agent Systems*, pages 307–314. AAAI Press, 1996.

[77] S. Sendall and W. Kozaczynski. Model transformation: the heart and soul of model-driven software development. *Software, IEEE*, 20(5):42–45, 2003.

[78] M. Sgroi, A. Kondratyev, Y. Watanabe, L. Lavagno, and A. Sangiovanni-Vincentelli. Synthesis of petri nets from message sequence charts specifications for protocol design. In *Proceedings of the Conference Design, Analysis and Simulation of Distributed Systems Symposium*, pages 193–199. ACM, 2004.

[79] S. Somé, R. Dssouli, and J. Vaucher. From scenarios to timed automata: Building specifications from users requirements. In *Proceedings of the Asia Pacific Software Engineering Conference*, pages 48–57. IEEE, 1995.

[80] I. Sommerville. *Software Engineering*. Pearson Studium, 2007.

[81] T. Stahl, M. Völter, S. Efftinge, and A. Haase. *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. dpunkt, 2007.

[82] A. S. Staines. A triple graph grammar mapping of uml 2 activities into petri nets. *International Journal of Computers*, 4(1):27–35, 2010.

[83] A. Steger. *Kombinatorik, Graphentheorie, Algebra*. Springer Berlin Heidelberg, 2007.

[84] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, 2009.

[85] P. Stevens. A landscape of bidirectional model transformations. In *Generative and Transformational Techniques in Software Engineering II*, Lecture Notes in Computer Science, pages 408–424. Springer Berlin Heidelberg, 2008.

[86] P. Stevens. Bidirectional model transformations in qvt: semantic issues and open questions. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems*, pages 1–15. Springer Berlin / Heidelberg, 2010.

[87] H. Storrle. Semantics of control-flow in uml 2.0 activities. In *Proceedings of the IEEE Symposium on Visual Languages and Human Centric Computing*, pages 235–242. IEEE, 2004.

[88] H. Störrle. Semantics and verification of data flow in uml 2.0 activities. *Electronic Notes in Theoretical Computer Science*, pages 35–52, 2005.

[89] H. Störrle and J. H. Hausmann. Towards a formal semantics of uml 2.0 activities. In *Proceedings of the German Software Engineering Conference*, pages 117–128. Springer-Verlag, 2005.

[90] A. Sutcliffe, N. A. M. Maiden, S. Minocha, and D. Manuel. Supporting scenario-based requirements engineering. *IEEE Transactions on Software Engineering*, 24(12):1072–1088, 1998.

[91] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[92] A. Terry, F. Hayes-Roth, L. Erman, N. Coleman, M. Devito, G. Papanagopoulos, and B. Hayes-Roth. Overview of teknowledge's domain-specific software architecture program. *SIGSOFT Software Engineering Notes*, 19(4):68–76, 1994.

[93] W. Thomas. The reachability problem over infinite graphs. In *Proceedings of the International Computer Science Symposium in Russia on Computer Science - Theory and Applications*, pages 12–18. Springer-Verlag, 2009.

[94] S. Uchitel and J. Kramer. A workbench for synthesising behaviour models from scenarios. In *Proceedings of the International Conference on Software Engineering*, pages 188 – 197. IEEE, 2001.

[95] C. Ullenboom. *Java ist auch eine Insel*, volume 1475. Galileo Press, 2003.

[96] H. Vogl, K. Lehner, P. Grünbacher, and A. Egyed. Reconciling requirements and architectures with the cbsp approach in an iphone app project. In *Proceedings of the IEEE International Requirements Engineering Conference*, pages 273–278, 2011.

[97] J. Whittle. Generating statechart designs from scenarios. In *Proceedings of the International Conference on Software engineering*, pages 314–323. ACM Press, 2000.

[98] A. Wigderson and S. Smale. P, np and mathematics - a computational complexity perspective. In *Proceedings of the International Congress of Mathematicians*, pages 665–712, 2006.

[99] J. Zhang, C. K. Chang, and J.-Y. Chung. Mockup-driven fast-prototyping methodology for web requirements engineering. In *Proceedings of the Annual International Conference on Computer Software and Applications*, pages 263–268. IEEE Computer Society, 2003.

[100] J. Zhang, Y. Liu, J. Sun, J. S. Dong, and J. Sun. Model checking software architecture design. In *Proceedings of the International Symposium on High-Assurance Systems Engineering (HASE), 2012 IEEE 14th*, pages 193–200. IEEE, 2012.

[101] B. Zimmerova, P. Vareková, N. Benes, I. Cerná, L. Brim, and J. Sochor. Component-interaction automata approach (coin). In *CoCoME*, pages 146–176. Springer, 2007.

Abbildungsverzeichnis

Definitionsverzeichnis

Curriculum vitae - Lebenslauf

Persönliche Angaben

Name:	Björn Schindler
Anschrift:	Tannenhöhe 7, 38678 Clausthal-Zellerfeld
Geburtsdatum:	01.07.1980
Familienstand:	verheiratet
Staatsangehörigkeit:	deutsch

Schulausbildung

1987-1991	Grundschule Bad Sachsa
1991-2000	Internatsgymnasium Pädagogium Bad Sachsa

Hochschulausbildung

01.10.2000-02.02.2007	TU Clausthal, Wirtschaftsinformatik (Diplom)

Berufstätigkeit

01.01.2007-31.12.2007	Wirtschaftsinformatiker bei der E.ON Avacon AG
seit 01.02.2008	Wissenschaftlicher Mitarbeiter an der TU-Clausthal, Institut für Informatik

Fremdsprachen

Englisch	fließend in Wort und Schrift
Latein	Großes Latinum

Sonstige Kenntnisse

Führerschein	Personenkraftfahrzeug
Anwendungsprogramme	Microsoft Word, Excel, Powerpoint
Programmiersprachen	Java, C/C++, HTML, PHP
Betriebssysteme	Windows XP/7/8, Linux