Technische Universität Ilmenau
Fakultät für Informatik und Automatisierung
Institut für Praktische Informatik und Medieninformatik
Fachgebiet für Verteilte Systeme und Betriebssysteme

Masterarbeit

Formale Spezifikation politikbezogener Funktionen einer Trusted Computing Base

Vorgelegt von:
Felix Neumann

Betreuer:	Prof. Dr.-Ing. habil. Winfried E. Kühnhauser Dipl.-Inf. Anja Pölck
Studiengang:	Informatik 2006
Eingereicht:	Ilmenau, den 27. August 2012

Version zur Veröffentlichung

Inhaltsverzeichnis

Verzeichnis benannter Listen

Einige Auflistungen wurden zur besseren Referenzierbarkeit mit Buchstaben gekennzeichnet. Sie befinden sich an folgenden Orten der Arbeit.

Hinweise

Die *Übersetzungen* der Zitate aus englischsprachigen Arbeiten stammen vom Autor dieser Masterarbeit.

An einigen Stellen der Arbeit erscheinen *Vorwärtsreferenzen* – sie sind an Leser gerichtet, welche gezielte Informationen zu einem bestimmten Thema suchen. Zum Verständnis der jeweiligen Textstelle sind sie unnötig.

KAPITEL 1

Einleitung

1.1 Themengebiet und Motivation

Angriffe auf IT-Systeme bergen ein enormes Schadenspotential. So entstehen der deutschen Wirtschaft aufgrund von Computerkriminalität jährlich Verluste in zweistelliger Milliardenhöhe [KPMG10]; die Angriffe sind in zunehmendem Maße professionell und zielgerichtet [CSI10]. Dem gegenüber steht die Tatsache, dass heute verbreitete IT-Systeme nicht in der Lage sind, die von ihnen geforderten Sicherheitseigenschaften zu erfüllen, wie die steigende Bedrohung aufgrund von Sicherheitslücken belegt [BSI11]. [Pöl10] argumentiert, dass die Ursache hierfür im unkontrollierten Wachstum derjenigen Systemkomponenten liegt, welche die Sicherheitseigenschaften des jeweiligen Systems erfüllen – also seiner sogenannten *Trusted Computing Base* (TCB). Dies begünstigt große, komplexe und chaotische Sicherheitsarchitekturen, deren Implementierungen aufgrund ihrer hohen Komplexität eine hohe Fehlerwahrscheinlichkeit aufweisen.

Ein neuartiger Ansatz zur Konstruktion sicherer Systeme zielt auf die Steuerung der Größe und Komplexität der TCB ab, was die Voraussetzungen für eine kleine und wenig komplexe Sicherheitsarchitektur schafft und den Grundstein legt für eine verifizierbar korrekte Implementierung [Pöl10]. Dies geschieht aufgrund von kausalen Abhängigkeiten von den Anforderungen über die Trusted Computing Base bis hin zu deren Realisierung in Sicherheitsarchitektur und Implementierung. Die Anforderungen an die Sicherheitseigenschaften eines IT-Systems gehen dabei aus einer Sicherheitspolitik und deren Formalisierung, dem *Sicherheitsmodell*, hervor. Hieraus kann die Trusted Computing Base abgeleitet werden, und zwar derart, dass diese minimal bezüglich ihrer Größe und Komplexität ist. Dieser Vorgang heißt *TCB Engineering*, an seinem Ende steht eine Spezifikation der TCB.

Dabei besteht jedoch das Problem, eine adäquate Notation für die TCB-Funktionen zu finden. Diese muss einerseits nah am mathematischen Modell sein, damit die Spezifikation validierbar ist. Andererseits muss es möglich sein, anhand der Spezifikation eine verifizierbar korrekte Realisierung zu erstellen. Benötigt werden also mathematische Techniken zur Softwareentwicklung, welche Validierung und Verifikation aller Teile des Software-Lebenszyklus erlauben – dies sind *formale Methoden* [WLB+09].

Das Erstellen einer formalen Spezifikation ist gemäß [WLB+09] nicht trivial. Demnach sind entsprechende Verfahren noch Gegenstand aktueller Forschung, und trotz einer großen Zahl an Fallstudien und industriellen Anwendungen werden die Techniken noch nicht breit eingesetzt. Insbesondere die zur verlässlichen Validierung

und Verifikation benötigten werkzeuggestützten Methoden sind vergleichsweise neu; zuvor wurde die formale Spezifikation hauptsächlich zur Modellierung, Animierung und Inspektion genutzt. Daher, so [WLB+09] weiter, handelt es sich bei bekannten Fallstudien überwiegend um „Anwendungen der formalen Methoden durch technische Spezialisten“. Auch [Bos95], eine Fallstudie zur formalen Spezifikation von Sicherheitsmodellen, verwendet vergleichsweise primitive Werkzeuge und präsentiert keine nutzbare Spezifikationsmethode, sondern stellt lediglich einen beispielorientierten Erfahrungsbericht vor.

Soll also eine TCB so spezifiziert werden, dass Techniken zur Validierung und Verifikation genutzt werden können, bedarf es formaler Methoden; hier jedoch ist keine geeignete Vorgehensweise bekannt.

1.2 Zielstellung

Gegeben sei ein Sicherheitsmodell, das die Sicherheitspolitik eines zu konstruierenden Systems modelliert. Ziel dieser Masterarbeit ist die Gewinnung einer Methode, anhand dieses Modells die Funktionen der Trusted Computing Base des Systems formal zu spezifizieren. Dabei muss die entstandene Spezifikation validierbar sein, sodass deren Vollständigkeit und Konsistenz zum Sicherheitsmodell argumentierbar und deren innere Widerspruchsfreiheit nachweisbar sind. Die Methode soll dabei die Möglichkeit berücksichtigen, Sicherheitsmodelle zu Modellklassen zusammenzufassen: eine zu einem Sicherheitsmodell einer Modellklasse erstellte Spezifikation soll für andere Modelle derselben Klasse wiederverwendbar sein. Zudem soll eine mittels der Methode hergestellte Spezifikation dazu nutzbar sein, eine Realisierung der TCB-Funktionen auf Architektur- oder Implementierungsebene herzustellen und die Korrektheit dieser Realisierung zu verifizieren. Im Zusammenspiel mit Techniken wie dem *TCB Engineering* sowie geeigneten Realisierungen schafft die zu entwickelnde Methode die Voraussetzungen für Systeme, welche die von ihnen geforderten Sicherheitseigenschaften erfüllen können.

1.3 Aufbau der Arbeit

Im Anschluss an das vorliegende Einleitungskapitel folgt Kapitel 2, welches zum Verständnis der weiteren Arbeit wesentliche Grundlagen vermittelt. Zentrale Gegenstände sind dabei erstens der Prozess zum Software Engineering unter Sicherheitsanforderungen als Kontext der Arbeit, zweitens das *TCB Engineering* zur Herleitung einer Trusted Computing Base aus dem zugrundeliegenden Sicherheitsmodell, und drittens formale Methoden zur Softwarespezifikation. Im Zuge der Darstellungen werden auch Konzept und Aufbau von Sicherheitsmodellen erläutert und ein beispielhaftes Modell eingeführt.

Anschließend entwickelt Kapitel 3 die in der Arbeit geforderte Methode zur Spezifikation von TCB-Funktionen. Dabei werden zuerst die Rahmenbedingungen der Methode analysiert und Anforderungen an die Methode aufgestellt. Hieraus entwickelt das Kapitel dann Lösungsstrategien und schließlich die Methode selbst. Diese wird in Kapitel 4 exemplarisch am Beispielmodell angewendet.

Die beispielhafte Anwendung nutzt Kapitel 5 zur Evaluierung. Es argumentiert, inwiefern die entwickelte Methode die formulierten Ziele erfüllt, und zeigt mögliche Verbesserungen und Weiterentwicklungen auf. Die Arbeit schließt mit der Zusammenfassung sowie dem Ausblick in Kapitel 6. Im Anhang findet sich die vollständige Beispielspezifikation sowie ergänzendes Material zur Arbeit.

KAPITEL 2

Grundlagen

Das vorliegende Kapitel erläutert grundlegende Konzepte der Arbeit sowie wesentliche Details zu ihrem Kontext. Abschnitt 2.1 enthält einleitend einige Hinweise zu mathematischen Konventionen der Arbeit.

Die weitere Struktur des Kapitels orientiert sich dann am *Security-Engineering*-Prozess zur Softwareentwicklung unter Sicherheitsanforderungen. Abschnitt 2.2 stellt diesen Prozess als übergeordneten Kontext der Arbeit vor und definiert wesentliche Begriffe. Wichtige Teilaspekte werden dann durch die verbleibenden Abschnitte des Kapitels ausführlicher behandelt.

In Abschnitt 2.3 wird das *Sicherheitsmodell* beschrieben, welches Ergebnis der Anforderungsanalyse und zentraler Ausgangspunkt dieser Arbeit ist. Auch führt dieser Abschnitt nachfolgend verwendete Notationen zu Sicherheitsmodellen ein. Ein konkretes Sicherheitsmodell, das der Arbeit als fortlaufendes Beispiel dient, beschreibt Abschnitt 2.4.

Das TCB Engineering, welches auf Grundlage des Sicherheitsmodells den Umfang der sicherheitskritischen Systemfunktionen bestimmt, wird durch Abschnitt 2.5 beschrieben. Um die korrekte Realisierung der TCB-Funktionen zu unterstützen, können diesen *formal spezifiziert* werden: Abschnitt 2.6 stellt solche Spezifikationen vor und erläutert relevante Notationen, Techniken und Werkzeuge.

2.1 Mathematische Konventionen

Der vorliegende Abschnitt gibt einige klärende Hinweise zu in dieser Arbeit verwendeten mathematischen Konventionen.

Boolesche Algebra. In der Arbeit kommt an mehreren Stellen die zweielementige booleschen Algebra auf dem Wertebereich $\mathbb{B} = \{0, 1\}$ zum Einsatz. Je nach Kontext wird $0 = \text{false}, 1 = \text{true}$ äquivalent verwendet.

Relationen. Zu Relationen, welche nachfolgend als Mengen von Zweitupeln aufgefasst werden, verwendet die vorliegende Arbeit eine Reihe von Notationen, die im Umfeld des Autors unüblich, jedoch für die Arbeit hilfreich sind und dem mathematischen Modell von Event-B entstammen [Eve12]. Letzteres ist eine in dieser Arbeit verwendete Notation zur formalen Spezifikation (siehe Unterabschnitt 2.6.4).

Für zwei Mengen A und B seien $R \subseteq A \times B$ eine Relation sowie $A' \subseteq A$ und $B' \subseteq B$ Mengen. A heißt Definitionsbereich und B Wertebereich von R. Nachfolgend werden die Operatoren $\lhd$ und $\rhd$ definiert, welche aus einer Relation eine neue

Relation mit verändertem Definitions- bzw. Wertebereich herstellen; dabei wird der entsprechende Bereich auf eine Teilmenge eingeschränkt. Es ist

- $A' \lhd R := \{(a, b) \in R \mid a \in A'\}$ die Einschränkung des Definitionsbereichs von R auf A', und
- $R \rhd B' := \{(a, b) \in R \mid b \in B'\}$ die Einschränkung des Wertebereichs von R auf B'.

Zusätzlich sind die Operatoren $\lhd$ und $\rhd$ definiert; sie sind eine abkürzende Schreibweise für die Subtraktion einer Teilmenge vom Definitions- oder Wertebereich einer Relation. Es ist

- $A' \lhd R := (A \setminus A') \lhd R = \{(a, b) \in R \mid a \notin A'\}$ die Subtraktion von A' vom Definitionsbereich von R, und
- $R \rhd B' := R \rhd (B \setminus B') = \{(a, b) \in R \mid b \notin B'\}$ die Subtraktion von B' vom Wertebereich von R.

Weiterhin ist $R[A'] := \{b \in B \mid \exists\, a \in A' : (a, b) \in R\}$ das relationale Bild von A' unter R; dies ist diejenige Menge aller Elemente $b \in B$, welche in R einem $a \in A'$ zugeordnet sind.

Funktionen. Mathematische Funktionen werden als linksvollständige und rechtseindeutige Relationen interpretiert, sodass für Funktionen f, g etwa $f \cup g$ ebenfalls eine Funktion ist, sofern $\forall x \in \mathrm{dom}(f) \cap \mathrm{dom}(g) : f(x) = g(x)$. Sind nun $f : X \to Z$ und $g : Y \to Z$, so ist $f \oplus g : (X \cup Y) \to Z$ diejenige Funktion, die durch *Überschreiben* von f durch g entsteht. Es gilt: $f \oplus g := ((X \setminus Y) \lhd f) \cup g = (Y \lhd f) \cup g$.

Konkatenation, Sequenz. Für zwei Relationen $R \subseteq X \times Y$ und $S \subseteq Y \times Z$ bezeichnet $R \fatsemi S := \{(x, z) \in X \times Z \mid \exists\, y \in Y : (x, y) \in R \wedge (y, z) \in S\}$ diejenige Relation, die durch Konkatenation von R und S entsteht. Für eine Relation $R \subseteq X \times X$ ist $R^0 := \{(x, x) \in X \times X\}$ und $\forall\, i > 0 : R^i := R \fatsemi R^{i-1}$.

Für zwei Funktionen $f : X \to Y$ und $g : Y \to Z$ bezeichnet die Funktion $f \fatsemi g : X \to Z$ entsprechend die sequenzielle Anwendung beider Funktionen. Es ist dann also $f \fatsemi g(x) = g(f(x))$. Alternativ zum starken Semikolon „$\fatsemi$“ kann auch das normale Semikolon „;“ verwendet werden.

2.2 Security Engineering

Immer dann wenn Software zu entwickeln ist, die schützenswerte Daten und Informationen enthält, jedoch durch mögliche Angriffe bedroht wird, so bestehen *Sicherheitsanforderungen*. Im Rahmen des Entwicklungsprozesses müssen diese unbedingt so früh wie möglich berücksichtigt werden. Die Folge ist die Verwendung eines Security-Engineering-Prozesses, welcher sich in das allgemeine Vorgehen zu Softwareentwicklung eingliedert.

Da das Security Engineering den Rahmen für diese Masterarbeit bildet, gibt der vorliegende Abschnitt einen Überblick hierüber. Zuerst erfolgt eine kurze Vorstellung des verwendeten Softwareentwicklungsprozesses.

Den meisten Vorgehensmodellen zur Softwareentwicklung ist folgende Aufteilung gemein: in der einleitenden Analysephase werden Anforderungen an das System ermittelt und eine Spezifikation wird erstellt (vgl. Abschnitt 2.6). Es folgt die Realisierungsphase, in welcher zuerst eine grobe und später eine detaillierte Architektur entworfen und dann die Implementierung durchgeführt wird. Sowohl Architektur als auch Implementierung werden nachfolgend *Realisierung* genannt. Je nach dem schließt sich eine Validierungsphase an, möglicherweise weitere Iterationen oder die Produkteinführung und -wartung.

In Bezug auf Sicherheitsanforderungen betont [Gas88]: „Sicherheitsaspekte fügen dem Entwicklungsprozess keine Schritte hinzu, die nicht bereits Teil konventioneller Softwareentwicklung sind." Vielmehr würde das Security Engineering jeden einzelnen Schritt wesentlich beeinflussen. Dies wird im verbleibenden Teil des Abschnittes dargestellt, zuerst anhand der Analysephase (Unterabschnitt 2.2.1), danach anhand der Realisierungsphase (Unterabschnitt 2.2.2).

2.2.1 Anforderungsanalyse

Im Rahmen des Security Engineering erfolgt während der Analysephase das *Security Requirements Engineering*. Ziel ist es nach [Eck04], den Schutzbedarf (Soll-Zustand) sowie Bedrohungen und Risiken für das System (Ist-Zustand) zu ermitteln, um dann daraus Abwehrmaßnahmen zu entwickeln und in einer Sicherheitspolitik festzuhalten.

Besteht ein hoher Schutzbedarf, kann diese Sicherheitspolitik formalisiert werden. Dabei entsteht ein *Sicherheitsmodell* genanntes formales Modell des Systems, welches die genannte Sicherheitspolitik umsetzt. Dieses Modell beschreibt dann knapp und präzise die Anforderungen an die sicherheitskritischen Systemteile, und dient so als Grundlage für die Herleitung, Spezifikation und Realisierung sicherheitsrelevanter Systemfunktionen.

Diese Eigenschaften machen das Sicherheitsmodell zum zentralen Ausgangspunkt für diese Arbeit; Abschnitt 2.3 beschreibt detailliert seine Inhalte und Notation.

2.2.2 Realisierung

Wurde eine Sicherheitspolitik zur Herstellung bestimmter Sicherheitseigenschaften ausgearbeitet, so muss diese im Rahmen des Security-Engineering-Prozesses nun *wirksam* realisiert werden. Dies aber hat Auswirkungen auf alle Teile des Systems, inklusive des Betriebssystemkerns [And72; Gas88; Här02; KEH+09].

Der dieser Arbeit zugrunde liegende Ansatz besteht darin, jene Funktionen des Systems, welche die Grundlage für das Vertrauen in die Einhaltung der Sicherheitseigenschaften sind, nach ingenieurmäßigen Prinzipien herzuleiten und zu realisieren. Die Menge dieser Funktionen wird *Trusted Computing Base* (TCB) genannt, diese ist wie folgt definiert [Pöl10].

Definition 2.1 (Trusted Computing Base). *Die Trusted Computing Base eines Systems ist eine Teilmenge der Funktionen dieses Systems. Sie enthält genau diejenigen Funktionen, welche notwendig und hinreichend zur dauerhaften Erfüllung der Sicherheitseigenschaften des Systems sind.*

Dabei meint der Begriff der „Funktion“ die abstrakte Eigenschaft eines Systems, ein bestimmtes Ein-/Ausgabeverhalten aufzuweisen. Zur Erfüllung der Sicherheitseigenschaften sind also ausschließlich die Funktionen der TCB verantwortlich, wodurch deren Korrektheit zentrale Bedeutung erhält. Dabei spielen folgende Aspekte eine Rolle [Boe84; Nis99].

(T1) **Vollständigkeit.** Die TCB muss das durch das Sicherheitsmodell vorgegebene Verhalten vollständig beschreiben.

(T2) **Externe Konsistenz.** Die Funktionen der TCB müssen konsistent zu den Vorgaben des Sicherheitsmodells spezifiziert sein. Im Rahmen eines hohen Schutzbedarfs lassen sich dabei die TCB und ihre Funktionen formal spezifizieren. Ihre externe Konsistenz kann dann mit formalen, mathematischen Methoden sichergestellt werden.

(T3) **Interne Konsistenz.** Die Spezifikation der TCB-Funktionen muss widerspruchsfrei sein. Dies betrifft insbesondere das Aufrechterhalten von Invarianten bei Änderung des Systemzustands.

(T4) **Präzise, verständliche Formulierung.** Damit die TCB in nachfolgenden Entwicklungsphasen korrekt realisiert werden kann, muss sie verständlich und präzise spezifiziert sein. Dies beinhaltet eine klare, knappe Formulierung in einer bekannten Sprache; zusätzlich muss die Spezifikation präzise und frei von Mehrdeutigkeiten sein. Im Rahmen eines hohen Schutzbedarfs kann eine formale Spezifikation zu diesen Eigenschaften wesentlich beitragen.

Für die Erfüllung dieser Anforderungen, insbesondere jedoch bei der formalen Spezifikation der TCB, ist deren geringe Größe und Komplexität wesentlich [Gas88; Sch00]. Im Zusammenhang mit formalen Spezifikationen können Werkzeuge einen Teil der Anforderungen automatisiert überprüfen (siehe Unterabschnitt 2.6.2).

Im Gegensatz zu anderen Arbeiten [SPH+06; MPP+08; SEK+09] liegt dieser Arbeit der Ansatz zugrunde, die TCB nicht auf Implementierungsebene, sondern auf der Ebene abstrakter Systemfunktionen zu betrachten. Sie dient damit allen Realisierungen sicherheitskritischer Systemteile als Spezifikation. Dies betrifft nicht nur die Implementierung, sondern insbesondere auch die Architektur, die eine entscheidende Rolle bei der Gewährleistung sämtlicher nichtfunktionaler Aspekte von Software spielt.

Im Rahmen eines Entwicklungsprozesses, der Vertrauen in sicherheitsrelevante Systemfunktionen ermöglicht, ist es erforderlich, sicherheitsrelevante Teile der Architektur zu identifizieren; diese werden nachfolgend zur *Sicherheitsarchitektur* zusammengefasst [Pöl10].

Definition 2.2 (Sicherheitsarchitektur). *Die Sicherheitsarchitektur ist ein Teil der Systemarchitektur. Sie enthält genau diejenigen Komponenten, Interaktionen und Beschränkungen, welche die Trusted Computing Base des Systems realisieren.*

Die Anforderung an eine geeignete Umsetzung der Sicherheitsarchitektur sind bereits seit langer Zeit bekannt (vgl. etwa [And72]): es handelt sich um die sogenannten *Referenzmonitorprinzipien*.

(A1) **Vollständigkeit.** Alle Informationsflüsse müssen vollständig und daher unumgehbar kontrolliert werden.

(A2) **Manipulationssicherheit, Robustheit.** Weder die bewusste noch die unbewusste Manipulation eines der Bestandteile der Sicherheitsarchitektur darf möglich sein.

(A3) **Korrektheit.** Die Sicherheitsarchitektur muss die Spezifikation überprüfbar korrekt umsetzen. Im Rahmen eines hohen Schutzbedarfs bedeutet dies die Verifizierbarkeit der Umsetzung; um diese zu ermöglichen, sind wiederum geringe Größe und Komplexität erforderlich.

Die korrekte Umsetzung einer TCB in eine Sicherheitsarchitektur kann eine Vielzahl an Komponenten erfordern. Beispiele für solche Komponenten sind etwa Interzeptoren, eine vertrauenswürdige graphische Oberfläche, kryptographische Mechanismen oder ein Trusted Platform Module [HHF+05].

Zur Abgrenzung von den beiden oben eingeführten Begriffen erfolgt nun noch die Definition der Implementierung einer TCB.

Definition 2.3 (Implementierung der TCB). *Die Implementierung der TCB eines Systems enthält genau diejenigen Datentypen, Operationen und Algorithmen, welche zur Implementierung der Funktionen der TCB sowie der zugehörigen Sicherheitsarchitektur gehören.*

Die Implementierung geht direkt aus der Architektur und damit indirekt aus der TCB hervor und setzt diese um. Insgesamt besteht also von der Sicherheitspolitik hin zur TCB-Implementierung eine kausale Kette; sollen Eigenschaften der Implementierung gesteuert werden, so muss dies bereits auf Ebene der Spezifikation erfolgen, etwa durch Betrachtung der TCB. Das nutzt etwa das in Abschnitt 2.5 beschriebene *TCB Engineering*.

2.3 Sicherheitsmodelle

Ein mögliches Produkt des Security Engineerings (vgl. vorhergehender Abschnitt) sowie wesentlicher Ausgangspunkt dieser Arbeit ist das Sicherheitsmodell. Es handelt sich um ein formales, verhaltensorientiertes Modell sicherheitsrelevanter Teile des Systems [Lan81] und dient der Beschreibung, Analyse und Implementierung der Sicherheitseigenschaften [KP11]. Aufgrund seiner Formalität gestattet es dabei insbesondere das Führen mathematischer Beweise zu den Sicherheitseigenschaften und bietet – dies ist im Kontext der vorliegenden Masterarbeit entscheidend – eine formale Spezifikation sicherheitskritischer Funktionen.

Der vorliegende Abschnitt führt zu formal notierten Sicherheitsmodellen solche Darstellungsformen ein, die für diese Arbeit eine Rolle spielen. Dies ist einerseits

der *Modellkern* (Unterabschnitt 2.3.1), welcher allen hier betrachteten Modellen gemeinsam ist und ihre grundlegenden Eigenschaften beschreibt. Eine Notation zur Modellierung der Dynamik von Sicherheitsmodellen ist das *Autorisierungsschema*, vorgestellt in Unterabschnitt 2.3.2. Unterabschnitt 2.3.3 schließlich entwickelt eine Methode der Klassifizierung verschiedener Sicherheitsmodelle, das *Metamodell*. Dieses bildet die Grundlage zur Wiederverwendung von Modellteilen in dieser Arbeit.

2.3.1 Modellkern

Nach [KP11] weisen viele gebräuchliche Sicherheitsmodelle, welche die Dynamik von Sicherheitspolitiken modellieren, einen gemeinsamen Modellkern auf. Es handelt sich hierbei um einen deterministischen Zustandsautomaten, welcher je nach Anwendungsfall um dynamische und statische Modellkomponenten erweitert wird. So können dynamische Komponenten zur Laufzeit veränderliche Modellaspekte wie Zugriffsrechte oder Rollenzuordnungen modellieren; hinzu kommen Regeln, welche mögliche Änderungen beschreiben. Statische Modellkomponenten beschreiben dann zur Laufzeit unveränderliche Aspekte wie Rechtemengen, Attributdomänen oder Typisierungen. Die Autoren stellen heraus, dass der einheitliche Modellkern für Zugriffssteuerungsmodelle „eine allgemeine Semantik einführt und grundlegende Prinzipien zum Model Engineering bietet“. Diese Eigenschaften bringen im Rahmen der vorliegenden Masterarbeit großen Nutzen, weshalb das Konzept im folgenden verwendet wird.

Allgemein ist ein Sicherheitsmodell demnach ein Zustandsautomat[1] $M = (Q, \Sigma, \Gamma, \delta, \lambda, q_0)$, bestehend aus der Menge der Zustände Q, der Menge möglicher Eingaben Σ, der Menge möglicher Ausgaben Γ, der Zustandsüberführungsfunktion $\delta : Q \times \Sigma \to Q$, der Ausgabefunktion $\lambda : Q \times \Sigma \to \Gamma$, sowie dem Anfangszustand $q_0 \in Q$.

- **Zustandsraum Q.** Über den Zustandsraum können dynamische Komponenten des Modells definiert werden, die Menge Q ist dabei die Menge aller möglichen Zustände des Modells und wird üblicherweise als Kreuzprodukt aus den möglichen Werten der einzelnen Komponenten notiert.

- **Eingaberaum Σ.** Als Eingaben dienen dem Modell Informationen, welche den Zustand sicherheitskritischer Systemteile beeinflussen oder Ausgaben provozieren. Von den tatsächlich vom System verarbeiteten Informationen wird dabei abstrahiert. Eine übliche Semantik für die Eingaben sind Kommandos mit zugehörigen Parametern, siehe Unterabschnitt 2.3.2.

- **Ausgaberaum Γ.** Über Ausgaben können verschiedene Informationen über den Modellzustand aus dem Modell geleitet werden. Im Rahmen dieser Arbeit sind Ausgaben Wahrheitswerte über $\mathbb{B}$ und bilden Zugriffsentscheidungen im Modell ab.

[1] Abweichend zur Darstellung in [KP11] verwendet diese Arbeit Sicherheitsmodelle, welche Ausgaben explizit mittels einer Ausgabefunktion sowie der zugehörigen Menge möglicher Ausgaben modellieren.

- **Zustandsüberführungsfunktion δ.** Die Semantik der über den Eingaberaum festgelegten Operationen wird mittels der Zustandsüberführungsfunktion spezifiziert, hieraus ergibt sich das Autorisierungsschema des Modells. Zumeist erfolgt dies durch partielle Definition: zu jeder Operation werden ihre Parameter, Bedingungen zur Ausführung sowie die Auswirkungen der Ausführung der Operation notiert; letzteres erfolgt angelehnt an übliche imperative Programmierparadigmen. Diese Schreibweise ist leicht verständlich und trägt somit auch zur Korrektheit bei. Details sind Unterabschnitt 2.3.2 zu entnehmen.

- **Ausgabefunktion λ.** Die Ausgabefunktion beschreibt, welche Ausgabe das System in Abhängigkeit von Eingabe und Zustand erzeugt.

- **Initialzustand q_0.** Dem initialen Systemzustand kommt insbesondere in Simulationen sowie in Beweisen zu Sicherheitseigenschaften Bedeutung zu. Letztere schließen dabei häufig von q_0 aus über die möglichen Zustandsübergänge in δ induktiv auf alle erreichbaren Zustände des Modells.

Die Konstruktion anwendungsspezifischer Modelle geschieht also einerseits durch Spezialisierung der Komponenten des Kerns, wodurch dynamische Aspekte hinzugefügt werden können, und andererseits durch Erweiterung des Modelltupels um statische Komponenten. Die dynamischen Komponenten eines vorliegenden Modells werden nachfolgend mit $D = (D_1, D_2, \ldots, D_{\mathrm{n}(D)})$ bezeichnet; jedes D_i steht dabei für die Menge der möglichen Werte, welche die Komponente annehmen kann. Es ist also $Q = D_1 \times D_2 \times \cdots D_{\mathrm{n}(D)}$. Auch andere Komponenten, wie die Zustandsüberführungsfunktion, sind entsprechend zu spezialisieren. Die statischen Komponenten werden mit derselben Semantik als $S = (S_1, S_2, \ldots, S_{\mathrm{n}(S)})$ bezeichnet, sodass der Kern eines erweiterten Modells in $\{(Q, \Sigma, \Gamma, \delta, \lambda)\} \times (S_1 \times S_2 \times \cdots \times S_{\mathrm{n}(S)}) \times \{(q_0)\}$ liegt.

Das Model Engineering wird typischerweise begleitet durch Sätze und Beweise, welche Zusammenhänge und Eigenschaften des Modells herausstellen. Ein praxisnahes und häufig anzutreffendes Beispiel für ein nach diesem Schema konstruiertes Sicherheitsmodell beruht auf einer Zugriffssteuerungsmatrix mit dynamischen Subjekt- und Objektmengen und wird von [HRU76] beschrieben. Ein im Rahmen dieser Arbeit als Beispiel genutztes Modell stellt Abschnitt 2.4 vor.

2.3.2 Autorisierungsschema

Nachfolgend wird die Beschreibung der Zustandsüberführungsfunktion eines auf Zustandsautomaten basierten Sicherheitsmodells mittels partieller Definition formalisiert; die Notation ist angelehnt an [HRU76]. Parallel dazu wird am Ende des Abschnittes eine Möglichkeit zur Definition der Ausgabefunktion vorgestellt.

Die Zustandsüberführungsfunktion δ beschreibt die Semantik der möglichen anwendungsspezifischen, zustandsverändernden *Kommandos*. Ein solches Kommando könnte beispielsweise den Versuch eine Subjektes darstellen, die Rechtesituation bezüglich eines Objektes zu verändern. Entsprechend benötigt ein Kommando eine Reihe von *Parametern*, die hierfür möglichen Wertebereiche ergeben sich

häufig aus dem Modellzusammenhang und seien nachfolgend $T_1, T_2, \ldots, T_n$ genannt[2]. Ein Kommando κ_i hat $0 \leq \mathrm{n}(\kappa_i)$ Argumente aus den Wertebereichen $T_1^{\kappa_i}, T_2^{\kappa_i}, \ldots, T_{\mathrm{n}(\kappa_i)}^{\kappa_i}$. Es ist dann im Modell

$$\Sigma = \bigcup_{\kappa_i} \{\kappa_i\} \times T_1^{\kappa_i} \times T_2^{\kappa_i} \times \ldots \times T_{\mathrm{n}(\kappa_i)}^{\kappa_i}.$$

Jedes anwendungsspezifische Kommando besteht aus einem Bedingungs- und einem Anwendungsteil. Der *Bedingungsteil* besteht aus einem Ausdruck der Prädikatenlogik erster Ordnung. Die Variablen in den Formeln ergeben sich dabei aus statischen und dynamischen Modellkomponenten. Es sind eine Reihe modellspezifischer Prädikate (in Abgrenzung zu später definierten, anwendungsspezifischen Prädikaten auch „Primitivprädikate“ genannt) definiert; ein Prädikat p_i hat dabei $1 \leq \mathrm{n}(p_i) + 1$ Stellen und den Typ

$$p_i : Q \times T_1^{p_i} \times T_2^{p_i} \times \cdots \times T_{\mathrm{n}(p_i)}^{p_i} \to \mathbb{B}.$$

Der *Anwendungsteil* besteht aus einer Reihe von modellspezifischen Operationen (in Abgrenzung zu anwendungsspezifischen Kommandos auch „Primitivoperationen“ genannt), welche den Zustandsübergang im Falle der Ausführung des Kommandos definieren. Eine Operation o_i hat $1 \leq \mathrm{n}(o_i) + 1$ Stellen und den Typ

$$o_i : Q \times T_1^{o_i} \times T_2^{o_i} \times \cdots \times T_{\mathrm{n}(o_i)}^{o_i} \to Q.$$

Insgesamt wird ein anwendungsspezifisches Kommando üblicherweise wie folgt notiert.

command $\kappa_i\ (x_1 \in T_1^{\kappa_i}, x_2 \in T_2^{\kappa_i}, \ldots, x_{\mathrm{n}(\kappa_i)} \in T_{\mathrm{n}(\kappa_i)}^{\kappa_i})$
  if $(p_1(\ldots) \vee p_2(\ldots) \vee \cdots) \wedge \cdots$ **then**
    $o_1\ (\ldots)$;
    $o_2\ (\ldots)$;
    $\vdots$
    $o_m\ (\ldots)$
end command

Hinter dem Schlüsselwort „**command**“ werden der Name des anwendungsspezifischen Kommandos sowie die notwendigen Parameter $x_1 \in T_1^{\kappa_i}, x_2 \in T_2^{\kappa_i}, \ldots, x_{\mathrm{n}(\kappa_i)} \in T_{\mathrm{n}(\kappa_i)}^{\kappa_i}$ vermerkt. Die nachfolgende Zeile enthält den optionalen Bedingungsteil, dessen Bedingung wie oben vermerkt in boolscher Algebra auf den modellspezifischen Primitivprädikaten formuliert wird. Der Anwendungsteil besteht aus einer beliebigen Zahl modellspezifischer Primitivoperationen. Der jeweils erste Parameter von Prädikaten und Primitiven, der Zustand, ist nicht Teil der Notation.

Zur Semantik: die Spezifikation von $\kappa_i(x_1, x_2, \ldots, x_{\mathrm{n}(\kappa_i)})$ definiert zu jedem $q \in Q$ die Stelle $\delta(q, (\kappa_i, x_1, x_2, \ldots, x_{\mathrm{n}(\kappa_i)}))$, und zwar als q, falls ein Bedingungsteil

[2]Notation: nachfolgend werden auch T_y^x verwendet. Jedes (x, y) lässt sich auf einen Index in $\{1, \ldots, n\}$ abbilden.

vorhanden ist und q diesen nicht erfüllt, andernfalls als das Ergebnis der verketteten Anwendung von $o_1, o_2, \ldots, o_m$ auf q.

In der Literatur finden sich einige Erweiterungen dieses Schemas wie etwa eine weitere, modellspezifische Typisierung der Parameter in [San92], welche jedoch keine größere Ausdruckskraft mit sich bringen.

Ausgabefunktion. Die Ausgabefunktion λ gibt in hier eingesetzten Modellen Wahrheitswerte über $\mathbb{B}$ oder das leere Wort aus. Je nach Erfordernissen des vorliegenden Modells können dabei einfach den vorliegenden anwendungsspezifischen Kommandos κ_i Ausgaben entsprechend ihres Bedingungsteiles zugeordnet werden. Alternativ ist es möglich, zusätzlich ein nicht zustandsveränderndes Gegenstück zu definieren, die *anwendungsspezifischen Prädikate*. Ein solches Prädikat π_i hat $0 \leq \mathrm{n}(\pi_i)$ Argumente aus den Wertebereichen $T_1^{\pi_i}, T_2^{\pi_i}, \ldots, T_{\mathrm{n}(\pi_i)}^{\pi_i}$. Die Definition des Eingabealphabets Σ muss entsprechend ergänzt werden:

$$\Sigma = \left(\bigcup_{\kappa_i} \{\kappa_i\} \times T_1^{\kappa_i} \times T_2^{\kappa_i} \times \ldots \times T_{\mathrm{n}(\kappa_i)}^{\kappa_i} \right) \cup \left(\bigcup_{\pi_i} \{\pi_i\} \times T_1^{\pi_i} \times T_2^{\pi_i} \times \ldots \times T_{\mathrm{n}(\pi_i)}^{\pi_i} \right).$$

An den Stellen π_i ist δ dann zustandserhaltend zu definieren.

2.3.3 Metamodelle

Ein in dieser Arbeit betrachtetes Sicherheitsmodell besteht, wie oben beschrieben, aus statischen Komponenten, dynamischen Komponenten (in der Definition des Zustandsraums Q), einem Autorisierungsschema (bestehend aus Eingaberaum Σ und Überführungsfunktion δ), der Ausgabe (bestehend aus Ausgaberaum Γ und Ausgabefunktion λ), sowie einem initialen Zustand (q_0). Die Darstellung solcher Modelle mit einem gemeinsamen Modellkern (vgl. Unterabschnitt 2.3.1) bietet bereits erhebliche Erleichterungen durch vereinheitlichte Semantik und Model-Engineering-Prinzipien.

Es haben sich jedoch darüber hinaus bestimmte *Modellklassen* herausgebildet, deren Mitglieder zumeist über identische

- statische und dynamische Komponenten,
- Ausgaberäume sowie
- Primitivoperationen und Primitivprädikate

verfügen und sich lediglich in

- Autorisierungsschema,
- Ausgabefunktion,
- den Werten statischer Komponenten sowie

- den initialen Werten dynamischer Komponenten

unterscheiden: letztere sind anwendungsspezifisch, erstere hingegen allen Modellen gemein. Daher bezeichnet die vorliegende Arbeit solche Modellklassen als *Metamodelle*. Konkrete Sicherheitsmodelle müssen dann lediglich als Repräsentant des zugehörigen Metamodells installiert werden. Diese Abstraktion bietet folgende Vorteile:

- **Wiederverwendbarkeit.** Sobald zu einem Metamodell eine Spezifikation vorliegt, lässt sich diese für eine Vielzahl verschiedener Sicherheitsmodelle wiederverwenden. Dies schließt auch hiermit einhergehende Methoden des Model Engineerings ein, wie etwa der Nachweis von Sicherheitseigenschaften, und vermindert damit den Aufwand zur Erstellung eines Sicherheitsmodells erheblich.
- **Verständlichkeit.** Ein Metamodell bringt eine bestimmte Anschauung, insbesondere in Form gemeinsamer Abstraktionen (Subjekte, Objekte, Rollen, Attribute usw.) mit sich. Der Aufwand, ein vorliegendes Sicherheitsmodell zu verstehen, ist bei Kenntnis des zugehörigen Metamodells erheblich geringer. Dies trägt auch zur Korrektheit abgeleiteter Realisierungen bei.

2.4 Beispiel eines Sicherheitsmodells

Der vorliegende Abschnitt stellt ein Sicherheitsmodell vor, das dieser Arbeit als Beispiel dienen soll und anhand dessen Kapitel 4 die in der Arbeit entwickelten Konzepte anwendet.

Es handelt sich um ein rollenbasiertes Modell. Unterabschnitt 2.4.1 führt zuerst das zugehörige Metamodell ein, anschließend beschreibt Unterabschnitt 2.4.2 das konkrete Sicherheitsmodell.

2.4.1 RBAC-Metamodell

Das verwendete Metamodell basiert prinzipiell auf dem RBAC_3-Modell in [SCF+96]; es enthält Rollenhierarchien aus RBAC_1 sowie, als Constraint im Sinne von RBAC_2, gegenseitigen Rollenausschluss. Das Modell wurde in die Kern-Notation aus Unterabschnitt 2.3.1 überführt, die Einteilung der Komponenten in statische und dynamische erfolgte passend zum konkreten Beispiel (siehe Unterabschnitt 2.4.2).

Statische Aspekte. Ein mit dem RBAC-Metamodell formuliertes Modell M ist ein 12-Tupel

$$M = (Q, \Sigma, \Gamma, \delta, \lambda, \overbrace{O, OP, R, m, RH, RE}^{\text{dem Metamodell eigen}}, q_0),$$

wobei die Komponenten $Q, \Sigma, \Gamma, \delta, \lambda$ und q_0 dem in Unterabschnitt 2.3.1 vorgestellten Modellkern entnommen und entsprechend definiert sind. Die Definition der übrigen, dem Metamodell eigenen Komponenten ist wie folgt.

- O, OP und R sind die endlichen Mengen aller Objekte, Operationen auf diesen Objekten und Rollen.

- $m : R \times O \to \mathcal{P}(OP)$ ist die Zugriffssteuerungsmatrix, die Rollen Rechte auf Objekte einräumt.

- $RH \subseteq R \times R$ ist die Rollenhierarchie, eine partielle Ordnung (reflexiv, transitiv, antisymmetrisch) auf der Menge aller Rollen. Die Notation für $(r_1, r_2) \in RH$ ist $r_1 \succeq r_2$, wobei r_2 die Rolle mit weniger Rechten ist, und es gilt $\forall r_1, r_2 \in R : r_1 \succeq r_2 \Rightarrow (\forall o \in O : m(r_2, o) \subseteq m(r_1, o)) \wedge$ (alle r_1 zugeordneten Nutzer sind auch r_2 zugeordnet).

- $RE \subseteq R \times R$ ist die Menge aller sich gegenseitig im Rahmen der Rollen-Nutzer-Zuordnung ausschließender Rollenpaare. Dies ist eine Möglichkeit, Verantwortlichkeitstrennung (*Separation of Duties*) zu implementieren. RE ist irreflexiv, nicht transitiv und symmetrisch. Die Notation für $(r_1, r_2) \in RH$ ist $r_1 \approx_{\text{ex}} r_2$.

Dynamische Aspekte. Der Zustandsraum Q eines Modells im RBAC-Metamodell ist definiert als

$$Q = \mathcal{P}(U) \times \mathcal{P}(S) \times \mathcal{P}(UA) \times USER \times ROLES,$$

wobei die Definitionen der einzelnen Komponenten wie folgt sind.

- U und S sind die jeweils potentiell unendlichen Mengen aller Nutzer bzw. Sitzungen.

- $UA \subseteq U \times R$ setzt solche Nutzer und Rollen miteinander in Relation, welche einander zugeordnet sind, sodass $(u, r) \in UA$ genau dann gilt, wenn der Nutzer u das Recht hat, die Rolle r anzunehmen.

- $USER = \{user \mid user : S \to U\}$ ist die Menge aller Abbildungen von Sitzungen auf Nutzer. In einem System im vorliegenden Metamodell kann ein Nutzer also zugleich in mehreren Sitzungen vertreten sein; dies könnten beispielsweise Programme sein, welche in seinem Namen ausgeführt werden.

- $ROLES = \{roles \mid roles : S \to \mathcal{P}(R)\}$ ist die Menge aller Abbildungen von Sitzungen auf Rollen. Dies sind die Rollen, welche für die jeweilige Sitzung aktiviert wurden. Üblicherweise können für die Sitzung eines Nutzers nur Rollen aktiviert werden, welche dem Nutzer auch in UA zugeordnet sind.

Zu einem Zustand $q \in Q$ sind die Objekte U_q, S_q, UA_q, $user_q$, $roles_q$ entsprechend definiert.

Primitivoperationen. Die Kommandos eines Modells im RBAC-Metamodell bauen auf folgenden Primitivoperationen auf; diese werden in nutzerbezogene, sitzungsbezogene sowie rollenbezogene Operationen eingeteilt.

Um das Modell so allgemein wie möglich zu halten und abgeleitete Systeme nicht unnötig zu beschränken, sind Operationen so definiert, dass sie keine Fehlerzustände erzeugen: soll etwa ein Nutzer $u \notin U_q$ entfernt werden, bleibt der Zustand unverändert.

Nutzerbezogene Primitive. Die nutzerbezogenen Primitivoperationen erlauben die Erzeugung und Löschung von Nutzern. Dabei wird im Falle der Erzeugung ein Nutzer aus dem Universum aller möglichen Nutzer U in die Menge der in Zustand q aktiven Nutzer U_q überführt. Entsprechendes gilt bei der Löschung.

(O1) $\text{addUsers} : Q \times \mathcal{P}(U) \to Q$, mit
$\text{addUsers}(q, u) = (U_q \cup u, S_q, UA_q, user_q, roles_q)$

(O2) $\text{deleteUsers} : Q \times \mathcal{P}(U) \to Q$, mit
$\text{deleteUsers}(q, u) = (U_q \setminus u, S_q, u \lhd UA_q, user_q \rhd u, roles_q)$

Sitzungsbezogene Primitive. Die zweite Kategorie von Primitivoperationen erlaubt das Erzeugen und Löschen von Sitzungen sowie die Verwaltung der Abbildung von Sitzungen zu Nutzern.

Sitzungen können zuerst erzeugt und gelöscht werden. Sitzungen entstammen wieder stets dem Universum aller möglicher Sitzungen S.

(O3) $\text{createSessions} : Q \times \mathcal{P}(S) \to Q$, mit
$\text{createSessions}(q, s) = (U_q, S_q \cup s, UA_q, user_q, (s \times \{\varnothing\}) \oplus roles_q)$

(O4) $\text{destroySessions} : Q \times \mathcal{P}(S) \to Q$, mit
$\text{destroySessions}(q, s) = (U_q, S_q \setminus s, UA_q, s \lhd user_q, s \lhd roles_q)$

Die folgenden Primitive erlauben es dann, die Zuordnung von Sitzungen auf die zugehörigen Nutzer zu modifizieren; dies betrifft die Sitzungs-Nutzerzuordnung *USER*.

(O5) $\text{mapUserSessions} : Q \times \mathcal{P}(S \times U) \to Q$, mit
$\text{mapUserSessions}(q, su) = (U_q, S_q, UA_q, user_q \oplus su, roles_q)$

(O6) $\text{unmapUserSessions} : Q \times \mathcal{P}(S \times U) \to Q$, mit
$\text{unmapUserSessions}(q, su) = (U_q, S_q, UA_q, user_q \setminus su, roles_q)$

Rollenbezogene Primitive. Schließlich gestatten es rollenbezogene Primitivoperationen einerseits, die Zuordnung von Rollen und Nutzern zu verwalten, sowie andererseits derartig zugeordnete Rollen in Sitzungen einzusetzen.

Mittels der folgenden beiden Operationen können Rollen zu Nutzern zugeordnet bzw. von Nutzern entzogen werden. Dies betrifft die Nutzer-Rollen-Zuordnung UA.

(O7) $\text{assignRolesToUsers} : Q \times \mathcal{P}(U \times R) \to Q$, mit
$\text{assignRolesToUsers}(q, ua) = (U_q, S_q, UA_q \cup ua, user_q, roles_q)$

(O8) $\text{revokeRolesFromUsers} : Q \times \mathcal{P}(U \times R) \to Q$, mit
$\text{revokeRolesFromUsers}(q, ua) = (U_q, S_q, UA_q \setminus ua, user_q, roles_q)$

Zu jeder Sitzung können Rollen aktiviert und deaktiviert werden. Die nachfolgend beschriebene Aktivierung einer Rolle im Rahmen einer Sitzung s ist üblicherweise nur dann zulässig, wenn diese Rolle auch dem zugehörigen Nutzer $user_q(s)$ zugewiesen wurde. Auch hier werden keine Fehlerzustände modelliert.

(O9) $\text{activateRoles} : Q \times \mathcal{P}(S \times R) \to Q$, mit
$\text{activateRoles}(q, sr) = (U_q, S_q, UA_q, user_q,$
$\{\ (s, r) \mid s \in S_q \wedge r = (roles_q(s) \cup sr[s])\ \})$

(O10) $\text{deactivateRoles} : Q \times \mathcal{P}(S \times R) \to Q$, mit
$\text{deactivateRoles}(q, sr) = (U_q, S_q, UA_q, user_q,$
$\{\ (s, r) \mid s \in S_q \wedge r = (roles_q(s) \setminus sr[s])\ \})$

Prädikate. Das Metamodell definiert fünf Prädikate, mit deren Hilfe Zugriffsentscheidungen abhängig vom Modellzustand spezifiziert werden können. Die ersten vier Prädikate $access_{SR}$, $access_{SM}$, $access_{UR}$ und $access_{UM}$ prüfen, wie bei RBAC üblich, das Vorhandensein einer Rolle bzw. eines Zugriffsrechts bei einem Nutzer bzw. einer Sitzung und berücksichtigen hierbei die RBAC_1-Rollenhierarchie. Das letzte Prädikat „*sod*“ erlaubt die Modellierung eines Constraints im Sinne von RBAC_2, der sich auf die Verantwortlichkeitstrennung bezieht.

(O11) $access_{SR} : Q \times S \times R \to \mathbb{B}$, mit
$access_{SR}(q, s, r) = \text{true} \Leftrightarrow \exists r' \in R : (r' \succeq r) \wedge (r' \in roles_q(s))$

(O12) $access_{SM} : Q \times S \times O \times OP \to \mathbb{B}$, mit
$access_{SM}(q, s, o, op) = \text{true}$
$\Leftrightarrow \exists r \in R : access_{SR}(q, s, r) \wedge ((o, op) \in m_q(r))$
$\Leftrightarrow \exists r, r' \in R : (r' \succeq r) \wedge (r' \in roles_q(s)) \wedge ((o, op) \in m_q(r))$

(O13) $access_{UR} : Q \times U \times R \to \mathbb{B}$, mit
$access_{UR}(q, u, r) = \text{true} \Leftrightarrow \exists r' \in R : (r' \succeq r) \wedge (r' \in UA_q(u))$

(O14) $access_{UM} : Q \times U \times O \times OP \to \mathbb{B}$, mit
$access_{UM}(q, u, o, op) = \text{true}$
$\Leftrightarrow \exists r \in R : access_{UR}(q, u, r) \wedge ((o, op) \in m_q(r))$
$\Leftrightarrow \exists r, r' \in R : (r' \succeq r) \wedge (r' \in UA_q(u)) \wedge ((o, op) \in m_q(r))$

Das folgende Prädikat prüft, ob die Zuordnung einer Rolle zu einem Nutzer gemäß der Relation RE der sich gegenseitig ausschließenden Rollen zulässig ist; dies ist Verantwortlichkeitstrennung auf Nutzerebene[3].

(O15) $sod : Q \times U \times R \to \mathbb{B}$, mit
$sod(q, u, r) = \text{true} \Leftrightarrow \forall r' \in UA_q(u) : \neg(r \approx_{\text{ex}} r')$

2.4.2 Modell eines Gesundheitsinformationssystems

Das konkrete Sicherheitsmodell geht aus einer in [SYR+07a; SYR+07b] beschriebenen (und auf Untersuchungen in [EB04] basierenden) Politik zu einer Betreuungseinrichtung für Senioren hervor. Die modellierten Regeln bilden lediglich einen kleinen Auszug aus den tatsächlich für den Betrieb der Einrichtung nötigen Regeln, doch muss das Modell für Demonstrationszwecke übersichtlich bleiben.

Zuerst erfolgt die Beschreibung statischer Modellkomponenten, dann die Spezifikation des Initialzustandes; zuletzt werden die anwendungsspezifischen Kommandos und Prädikate definiert.

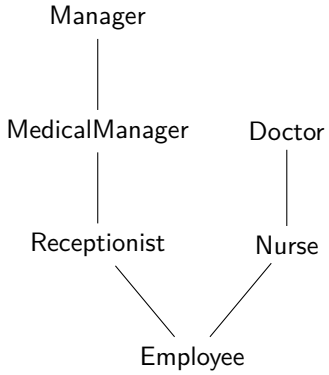

Abbildung 2.1: Hierarchie der Rollen. Rollen mit weniger Berechtigungen befinden sich weiter unten im Bild; so ist etwa Nurse $\succeq$ Employee.

Statische Komponenten. Nachfolgend werden die statischen Objekte, Operationen und Berechtigungen sowie die Rollen inklusive der ihnen zugewiesenen Berechtigungen, der Rollenhierarchie und der Ausschlussrelation beschrieben.

Das Modell beschreibt zehn verschiedene Nutzerrollen, diese sind:

$$R = \{Employee, Manager, Doctor, Nurse, Receptionist, Patient, MedicalManager, MedicalTeam, ReferredDoctor, UserAdmin\}$$

[3] Je nach Anwendungsfall wäre es etwa auch möglich, lediglich die in einer Sitzung aktivierbaren Rollen zu beschränken.

Besonderer Erklärung bedarf einerseits der *MedicalManager*, welcher Ärzte (Rolle *Doctor*) und Pflegekräfte (Rolle *Nurse*) zu medizinischen Teams (Rolle *MedicalTeam*) zusammenschließen kann, und andererseits der *ReferredDoctor*, zu welchem ein Patient durch einen anderen Arzt verwiesen wurde und der daher zeitweilig Zugriff auf die Patienteninformationen erhält.

Die Rollenhierarchie *RH* geht aus Abbildung 2.1 hervor. [SYR+07b] betont dabei, dass die Modellierung bestimmter Zusammenhänge in der Hierarchie unnötig sei: so werde etwa die Rolle *ReferredDoctor* ohnehin nur Personen zugewiesen, die bereits in der Rolle *Doctor* sind. Das Modell benennt weiterhin eine Reihe von Objekten, auf die mit folgenden Operationen zugegriffen werden kann:

$$OP = \{view, add, modify, access, enter, create, update, sign\}.$$

Während das *view*-Recht dabei die Möglichkeit zum direkten lesenden Zugriff auf das betreffende Objekt modelliert, stellt das *access*-Recht eine Einschränkung dar. [EB04] schlägt vor, solche Zugriffe etwa aufzuzeichnen und bei Häufungen das Management zu benachrichtigen.

Objekt	**Oper.**	**Rolle**	Manager	Doctor	Nurse	Receptionist	UserAdmin	Patient
OldMedicalRecords	view			×				×
OldMedicalRecords	enter		×					
OldMedicalRecords	access				×			
RecentMedicalRecords	view			×	×			×
RecentMedicalRecords	enter		×					
RecentMedicalRecords	add			×				
PrivateNotes	view			×				
PrivateNotes	add			×				
Prescriptions	view			×				×
Prescriptions	modify			×				
PatientPersonalInfo	access		×					
PatientFinancialInfo	access		×					
PatientMedicalInfo	access		×					
CarePlan	update		×					
CarePlan	view				×			
Appointment	create					×		
ProgressNotes	add				×			
LegalAgreement	sign							×
Bills	view							×
U_o	update						×	
UA_o	update						×	

Tabelle 2.1: Vollständige Übersicht aller Objekte, Operationen und den Rollen zugeordneten Berechtigungen im Beispielmodell.

Allgemein ist nicht jede Operation für jedes Objekt relevant. Die Objekte O sowie

die Zuordnung von den verwendeten Rollen zu Berechtigungen m sind Tabelle 2.1 zu entnehmen. Die Rollen *MedicalManager*, *PrimaryManager* und *ReferredDoctor* verfügen in diesem Beispielmodell über keine eigenen Berechtigungen und sind lediglich im Rahmen des Autorisierungsschemas entscheidend.

Beachtenswert sind einige Feinheiten in diesem Modell, durch welche Verantwortlichkeiten getrennt werden können. So ist eine Person in der *Manager*-Rolle in der Lage, medizinische Aufzeichnungen anzulegen; das Hinzufügen weiterer Einträge ist jedoch nur einem Arzt möglich.

Entgegen den Vorgaben von [SCF+96] sind die Berechtigung zum Ändern der Benutzermenge U sowie der Nutzer-Rollen-Zuordnung UA ebenfalls über die Objekte U_o und UA_o im Modell enthalten; ein vollständiges administratives Modell wäre im Rahmen dieses Beispieles unnötig.

Gemäß RE schließen sich genau die folgenden Rollen gegenseitig aus:

$$\begin{aligned} \mathit{Patient} &\approx_{\text{ex}} \mathit{Employee} \\ \mathit{Patient} &\approx_{\text{ex}} \mathit{ReferredDoctor} \\ \mathit{Patient} &\approx_{\text{ex}} \mathit{UserAdmin} \\ \mathit{Doctor} &\approx_{\text{ex}} \mathit{Manager} \\ \mathit{Doctor} &\approx_{\text{ex}} \mathit{Receptionist} \end{aligned}$$

Initialzustand. Der Initialzustand $q_0 = (U_0, S_0, UA_0, user_0, roles_0)$ sieht lediglich einen Administrator vor und ist wie folgt definiert:

$$\begin{aligned} U_0 &= \{u_1\} \\ S_0 &= \varnothing \\ UA_0 &= \{\, (u_1, \mathit{UserAdmin}) \,\} \\ user_0 &= \varnothing \\ roles_0 &= \varnothing \end{aligned}$$

Kommandos, Prädikate. Die Menge aller anwendungsspezifischen Prädikate ist $\{\pi_{op} \mid op \in OP\}$; es existiert also zu jeder Operation op ein Prädikat $\pi_{op} : Q \times S \times O$, die zugehörige Ausgabe wird definiert über $\pi_{op}(q, s, o) = access_{SM}(q, s, o, op)$.

Alle anwendungsspezifischen Kommandos sind nachfolgend definiert. Die Ausgabe zu jedem Kommando entspricht dem Wert ihres Bedingungsteils. Zu Beginn werden Mechanismen eingeführt, mit deren Hilfe die Nutzer verwaltet werden können. Hierfür genügt das *update*-Recht für U_o.

(C1)
command createUser ($s \in S, u \in U$)
  if $access_{SM}(s, U_o, update)$ **then**
    addUsers ($\{u\}$)
end command

(C2) **command** destroyUser $(s \in S, u \in U)$
if $access_{SM}(s, U_o, update)$ **then**
destroySessions $(user_q^{-1}(u))$;
deleteUsers $(\{u\})$
end command

Die folgenden Operationen zu unbeschränkter Zuweisung und Entzug von Rollen können nur durch den Administrator durchgeführt werden.

(C3) **command** assignRole $(s \in S, u \in U, r \in R)$
if $access_{SM}(s, UA_o, update) \wedge sod(u, r)$ **then**
assignRolesToUsers $(\{ (u, r) \})$
end command

(C4) **command** revokeRole $(s \in S, u \in U, r \in R)$
if $access_{SM}(s, UA_o, update)$ **then**
revokeRolesFromUsers $(\{ (u, r) \})$
end command

Die An- und Abmeldung am System muss allen Personen möglich sein, die sich als Benutzer authentifizieren können. Gleiches gilt für die Aktivierung und Deaktivierung von dem Benutzer zugeordneten Rollen. Es ist zu beachten, dass zum Login sowie zur erstmaligen Rollenaktivierung keine Sitzungsrollen vorausgesetzt werden können (diese sind schließlich nicht aktiviert).

(C5) **command** login $(u \in U, s \in S)$
createSessions $(\{s\})$;
mapUserSessions $(\{ (s, u) \})$
end command

(C6) **command** logout $(s \in S)$
unmapUserSessions $(\{ (s, user_q(s)) \})$;
destroySessions $(\{s\})$
end command

(C7) **command** activateRole $(s \in S, r \in R)$
if $access_{UR}(user_q(s), r)$ **then**
activateRoles $(\{ (s, \{r\}) \})$
end command

(C8) **command** deactivateRole $(s \in S, r \in R)$
deactivateRoles $(\{ (s, \{r\}) \})$
end command

Die folgenden Operationen erlauben es Rollen außerhalb des Administrators in eingeschränktem Maße Rollenzuordnungen vorzunehmen. Dabei wird jeweils im Rahmen einer Sitzung s einem Nutzer u eine Rolle zugeordnet oder entzogen.

In jedem Fall sind dabei der Rollenbesitz sowie der Rollenausschluss zu überprüfen.

(C9) **command** assignReferredDoctorRole ($s \in S, u \in U$)
if $access_{SR}(s, Doctor) \wedge access_{UR}(u, Doctor)$
$\wedge\ sod(u, ReferredDoctor)$
then
assignRolesToUsers ({ $(u, ReferredDoctor)$ })
end command

(C10) **command** revokeReferredDoctorRole ($s \in S, u \in U$)
if $access_{SR}(s, Doctor)$ **then**
revokeRolesFromUsers ({ $(u, ReferredDoctor)$ })
end command

(C11) **command** assignPatientRole ($s \in S, u \in U$)
if $access_{SR}(s, Receptionist) \wedge sod(u, Patient)$ **then**
assignRolesToUsers ({ $(u, Patient)$ })
end command

(C12) **command** revokePatientRole ($s \in S, u \in U$)
if $access_{SR}(s, Receptionist)$ **then**
revokeRolesFromUsers ({ $(u, Patient)$ })
end command

(C13) **command** assignMedicalTeamRole ($s \in S, u \in U$)
if $access_{SR}(s, MedicalManager)$
$\wedge\ (access_{UR}(u, Doctor) \vee access_{UR}(u, Nurse))$
$\wedge\ sod(u, MedicalTeam)$
then
assignRolesToUsers ({ $(u, MedicalTeam)$ })
end command

(C14) **command** revokeMedicalTeamRole ($s \in S, u \in U$)
if $access_{SR}(s, MedicalManager)$ **then**
revokeRolesFromUsers ({ $(u, MedicalTeam)$ })
end command

2.5 TCB Engineering

Soll also ein System konstruiert werden, das Sicherheitseigenschaften vertrauenswürdig gewährleistet, so muss dessen Implementierung die Vorgaben der Sicherheitsarchitektur sowie der TCB korrekt umsetzen. Es besteht also eine kausale Kette, die von den Anforderungen bis hin zur TCB-Implementierung reicht (vgl. Abbildung 2.2). Unerlässlich für das Erreichen der Korrektheitsanforderung sind wiederum die geringe Größe und Komplexität der TCB (siehe Abschnitt 2.2). Um diese herzustellen, müssen Umfang und Inhalt der TCB gezielt gesteuert werden.

Diese ingenieurmäßige Konstruktion der Trusted Computing Base wird als *TCB Engineering* bezeichnet.

Dieser Arbeit liegt dabei der Ansatz zu Grunde, die kausalen Abhängigkeiten zwischen den Systemrepräsentationen auszunutzen. So ist das Sicherheitsmodell ein Modell derjenigen Systemteile, welche die Sicherheitseigenschaften umsetzen (siehe Abschnitt 2.3), während die TCB genau die Funktionen derselben Systemteile (siehe Definition 2.1) enthält. Da das Sicherheitsmodell stets anwendungsspezifisch konstruiert wird, kann die TCB durch Ableitung ihrer Funktionen aus dem Sicherheitsmodell auf die konkrete Anwendung zugeschnitten werden.

Die Größe und Komplexität einer derartig *kausal abgeleiteten TCB* wird also allein durch die Sicherheitspolitik bestimmt; sie enthält keine überflüssigen Funktionen und keine funktionalen Redundanzen. Dieser Ansatz unterscheidet sich von heute verbreiteten TCBs, welche nicht explizit oder nur für ein möglichst allgemeines Anwendungsfeld konstruiert wurden und entsprechend zu groß und komplex sind, als dass sie Grundlage für ein verifizierbar korrektes System sein könnten [Pöl10].

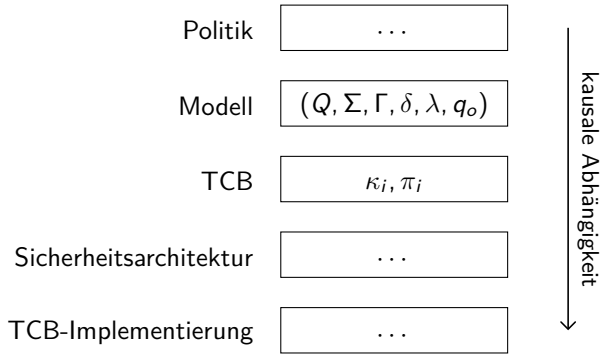

Abbildung 2.2: Übersicht verschiedener Systemrepräsentationen beim Security Engineering. Kausale Abhängigkeiten verlaufen von oben nach unten: aus der Politik ergibt sich das Modell, daraus die TCB usw.

Abbildung 2.2 veranschaulicht die kausalen Zusammenhänge zwischen Sicherheitsmodell und TCB und setzt diese in den Kontext der weiteren Repräsentationsformen sicherheitskritischer Systemteile. Dabei ist die TCB die Menge aller sicherheitsrelevanten Funktionen (siehe Abschnitt 2.2). Nun setzt die TCB das Sicherheitsmodell um; dieses jedoch spezifiziert das Ein-/Ausgabeverhalten durch δ und λ, welche nach der Methode in Unterabschnitt 2.3.2 partiell durch anwendungsspezifische Kommandos κ_i und Prädikate π_i definiert sind. Damit die TCB-Funktionen also hinreichend und notwendig zur Umsetzung des Sicherheitsmodells sind, muss die TCB alle κ_i und π_i umfassen, mit ihrer in δ und λ spezifizierten Semantik. Diese hängen dann wiederum von der Definition der anderen Strukturen im Modell ab.

Nach dem selben Prinzip kann die Sicherheitsarchitektur aus der TCB, und die Implementierung aus der Architektur hergeleitet werden. Die Politik steuert damit den Umfang des gesamten Systems und erlaubt es, die Anforderung der geringen Größe und Komplexität zu erreichen. Die vorliegende Arbeit realisiert dabei einen Teil der kausalen Kette. Sie entwickelt eine Überführungsmethode, welche das

Sicherheitsmodell in die Spezifikation der TCB-Funktionen überträgt. Diese ist dann der Ausgangspunkt zur Herstellung korrekter TCB-Implementierungen, welche die vom System geforderten Sicherheitseigenschaften verlässlich herstellen können.

2.6 Formale Spezifikation

Das Ergebnis der Analysephase des Software-Entwicklungsprozesses (vgl. Abschnitt 2.2) sowie Ausgangspunkt für die Realisierungsphase ist die *Spezifikation*. [Lam00] definiert die Spezifikation als „Ausdruck [...] einer Sammlung von Eigenschaften, die ein System erfüllen soll“. Sie hat den Charakter eines Vertrages zwischen Auftraggeber und Entwickler, dokumentiert also die (Nutzer-)Anforderungen an das System und ist somit Bestandteil der Systemdokumentation. [LS09] betont, dass die Spezifikation lediglich beschreibt, *was* das System leisten soll und dabei davon abstrahiert, *wie* dieses Problem zu lösen ist – es handelt sich also einerseits um eine Außensicht auf das System, andererseits jedoch auch um ein abstraktes Modell des Problems.

Spezifikationen sind in verschiedenen Formen denkbar. Die übliche *natürlichsprachliche Spezifikation* hat nach [LS09] das Problem, dass sie mehrdeutig und unpräzise ist, dabei jedoch „trügerischerweise verständlich und lesbar“. Nach [Nis99] sind derartige Mehrdeutigkeiten und Unsicherheiten bezüglich der Anforderungen eine Hauptquelle für Entwurfsfehler – diese zu beseitigen, ist demnach das Ziel der *formalen Spezifikation*, also der Softwarespezifikation unter Verwendung einer mathematischen Notation. Hierzu muss eine formale Spezifikation präzise, klar und knapp sein.

Weitere Anforderungen sind nach [Lam00] die interne Konsistenz der Spezifikation, ihre Vollständigkeit und externe Konsistenz in Bezug auf übergeordnete Spezifikationen, ihre Erfülltheit durch untergeordnete (kausal abgeleitete) Spezifikationen sowie ihre *Minimalität*. Die genannte Veröffentlichung definiert letztere als die Forderung, „keine Eigenschaften auszudrücken, die für das Problem irrelevant oder lediglich für eine Lösung des Problems relevant sind“.

Eine formale Spezifikation bringt nicht nur aufgrund der eindeutigen Formulierung Gewinn. Weitere wesentliche Vorteile sind nachfolgend aufgeführt.

- Sie ermöglicht die Überprüfung der Korrektheit nachfolgender Artefakte. Insbesondere in kritischen Sicherheitsanwendungen kann dabei der formale Korrektheitsbeweis (*Verifizierung*) eingesetzt werden. Wie [Bow96] betont, verbessert das Anlegen einer formalen Spezifikation jedoch auch ohne Verifizierung die Korrektheit des Systems.

- Auf Grundlage einer formalen Spezifikation können in der Realisierungsphase Entwicklungsverfahren eingesetzt werden, die auf die Korrektheit der entstehenden Implementierung abzielen („Korrektheit per Konstruktion“ [Nis99] bzw. „Top-Down-Refinement“ [LS09], siehe Unterabschnitt 2.6.1).

- Formale Spezifikationen können das Verständnis für das Systemverhalten fördern [Bow96]. Sie helfen ebenso beim Ordnen von Gedanken.

- Eine formale Spezifikation begünstigt die Erstellung von Testfällen [LS09].
- Durch die Technik der Animierung (siehe Unterabschnitt 2.6.2) kann anhand einer formalen Spezifikation bereits in frühen Entwicklungsphasen ein Prototyp erstellt werden [Nis99].
- Eng damit verbunden ist die Möglichkeit mittels formaler Systemmodelle Entwurfsentscheidungen zu erkunden [Bow96]. Dies beinhaltet das Identifizieren fehlender Bestandteile sowie das Abwägen von Alternativen.

Der verbleibende Abschnitt gibt einen Überblick über Vorgehensweisen zur formalen Spezifikation und ist wie folgt aufgebaut. Die ersten beiden Unterabschnitte beschreiben allgemeine Verfahren zur Herstellung der Korrektheitseigenschaft unter Einsatz formaler Methoden: Unterabschnitt 2.6.1 erläutert ein Verfahren zur Konstruktion korrekter Programme aus Spezifikationen; Unterabschnitt 2.6.2 stellt die Prozesse der Validierung und Verifizierung zur Überprüfung der Korrektheit vor. Die restlichen zwei Unterabschnitte gehen konkret auf in dieser Arbeit benötigte Methoden zur Spezifikation ein. Der Unterabschnitt 2.6.3 betrachtet die in dieser Arbeit vorliegenden zustandsbasierten Systeme und beschreibt auf abstrakte Weise die Inhalte der Spezifikationen solcher Systeme. Schließlich stellt Unterabschnitt 2.6.4 einige für diese Arbeit in Frage kommende Spezifikationssprachen vor und benennt die Vorteile der letztlich gewählten Sprache.

2.6.1 Korrektheit durch Konstruktion

Bei Vorliegen einer formalen Spezifikation liegt der Ansatz nahe, die Korrektheit der Implementierung konstruktiv herzustellen. Dabei werden aus abstrakten Artefakten schrittweise konkrete Artefakte gewonnen; jeder Teilschritt macht dabei hinreichend wenige Änderungen, um beherrschbar und im Nachhinein verifizierbar zu sein. Die Darstellungen folgen [LS09], welche dieses Vorgehen auch als „Top-Down-Refinement“ bezeichnen.

Die vorliegende Arbeit macht keinen expliziten Gebrauch von hier vorgestellten Techniken. Diese werden jedoch als Grundlage zum Verständnis der Spezifikation erachtet und sind im Rahmen einer Weiternutzung der Erkenntnisse dieser Arbeit wichtig.

Ausgangspunkt ist ein abstraktes Modell S des Problems, die formale Spezifikation. Inhalt der Methode ist nun die schrittweise Verfeinerung von S und damit das Ableiten weiterer, konkreterer Modelle. Es entsteht eine geordnete Folge $S = S_1, S_2, \ldots, S_f$ von Modellen, wobei S_f die letztliche Implementierung darstellt. Damit der Vorgang zielgerichtet, endlich und überprüfbar ist, unterliegt die Verfeinerung folgenden Regeln.

Definition 2.4 (Verfeinerung). *Seien S_i, S_{i+1} Modelle desselben Problems. S_{i+1} heißt* Verfeinerung *von S_i, wenn folgende drei Bedingungen erfüllt sind.*

- *S_{i+1} ist stärker als S_i. Das heißt erstens, dass jede zulässige* Eingabe *für S_i auch eine zulässige Eingabe für S_{i+1} sein muss, jedoch kann S_{i+1} insgesamt*

mehr Eingaben annehmen. Es bedeutet zweitens, dass jede zulässige Ausgabe *von S_{i+1} auch eine zulässige Ausgabe von S_i sein muss, jedoch kann die Definition von S_{i+1} insgesamt weniger zulässige Ausgaben erlauben.*

- *S_{i+1} ist konkreter (i.S.v. weniger abstrakt) als S_i.*
- *S_{i+1} ist auf Grundlage von S_i leicht verifizierbar. Dies kann einen formalen Beweis oder ein informales Argument bedeuten.*

S_{i+1} ist also im Allgemeinen detailierter und daher einerseits komplexer und schwerer verständlich als S_i, andererseits schwerer eigenständig zu beweisen. Jeder Verfeinerungsschritt umfasst die Verfeinerung von Operationen und/oder Daten. Stärker verfeinerte Modelle neigen dazu, stärker algorithmisch zu sein: die Beschreibung einer Operation durch mathematische Zusammenhänge wird ersetzt durch die Beschreibung als Algorithmus. Abstrakte Datentypen werden durch implementierungsnahe, konkrete Typen ausgetauscht.

In Zusammenhang mit der vorliegenden Arbeit ist zu beachten, dass zu dem Problem, ein System entsprechend bestimmter Sicherheitsanforderungen zu konstruieren, leicht ein abstraktes Modell gefunden werden kann: dies ist das Sicherheitsmodell (siehe Abschnitt 2.3). Eine Spezifikation auf beliebigem Abstraktionslevel kann dann durch schrittweise Verfeinerung entsprechend der beschriebenen Vorgehensweise erarbeitet werden. Da jeder Teilschritt (formal) verifizierbar ist, besteht immer auch eine Beweiskette zurück zum Sicherheitsmodell.

2.6.2 Validierung und Verifizierung

Der Einsatz einer formalen Spezifikation erfolgt zur Herstellung der Korrektheit der zu konstruierenden Software. Wie in Abschnitt 2.2 dargestellt, muss das Produkt entlag der kausalen Kette von den Anforderungen bis hin zur Implementierung nachweisbar korrekt sein. Dabei ist einerseits zu prüfen, ob die Spezifikation den tatsächlichen Anforderungen entspricht (*Validierung*) und andererseits, ob die Realisierung die Spezifikation umsetzt (*Verifizierung*, vgl. Abbildung 2.3). Beide Teilschritte werden im Rest des Unterabschnitts beschrieben.

Validierung. Der Prozess der Validierung sucht die folgenden Fragen zu klären:

- „Baue ich das richtige Produkt?“ [Boe84]
- „Löst die Software das richtige Problem?“ [IEEE1012]
- „Erfüllt die Software den vorgesehenen Zweck?“ [IEEE1012]

Er vollzieht sich anhand der Spezifikation, da diese als Vorschrift für die Realisierung dient. Dabei ist einerseits zu klären, inwiefern die formale Spezifikation den informalen Anforderungen entspricht (*externe Validierung*, [Nis99]). Im Rahmen der vorliegenden Arbeit muss dabei das formale Sicherheitsmodell (siehe Abschnitt 2.3) betrachtet werden. Andererseits muss die interne Konsistenz der Spezifikation

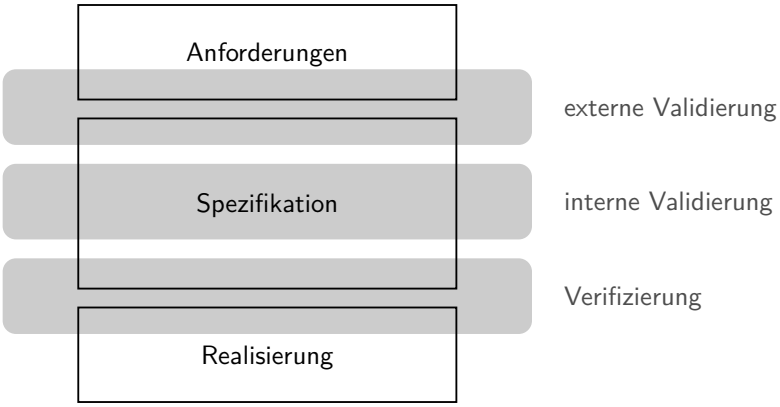

Abbildung 2.3: Validierung und Verifizierung zur Herstellung der Korrektheitsbeziehung zwischen Spezifikation und Anforderungen bzw. Realisierung.

überprüft werden (*interne Validierung*, [Nis99]); hierbei ist diese auf gewünschte und unerwünschte Eigenschaften hin zu analysieren. Damit sind die Forderungen aus Unterabschnitt 2.2.2 nach Vollständigkeit, interner und externer Konsistenz der TCB-Spezifikation behandelt.

Aufgrund deren Maschinenlesbarkeit kann die Diskussion formaler Spezifikationen werkzeuggestützt erfolgen, sodass viele Schritte der (insb. internen) Validierung automatisch vollziehbar sind. Dabei existieren folgende Arten von Werkzeugen:

- **Syntax-, Typprüfer.** Der Syntaxprüfer stellt die syntaktische Korrektheit eines Spezifikationsdokumentes sicher. Da formale Spezifikationen zumeist statisch typisiert werden, kann der Typprüfer zusätzlich Typfehler entdecken. Diese Werkzeuge liegen für einen Großteil der Sprachen zur formalen Spezifikation vor und arbeiten vollautomatisch; die erfolgreiche Prüfung ist oft Voraussetzung zur Anwendung weiterer Analyseschritte.

- **Erzeuger für Beweispflichten.** Solche Werkzeuge analysieren ein gegebenes Spezifikationsdokument und erstellen eine Liste an formalen Aussagen (sog. *Beweispflichten*, vgl. Unterabschnitt 2.6.3), deren Gültigkeit für die Korrektheit der Spezifikation notwendige Voraussetzung ist und die daher bewiesen werden müssen. Kann eine solche Aussage nicht bewiesen werden oder ist der Beweis äußerst schwierig, ist dies ein ernsthafter Hinweis auf Mängel an der Spezifikation. Die konkreten Beweispflichten hängen dabei von dem Modell ab, das der Spezifikationssprache zugrunde liegt; je konkreter und spezieller das Modell, desto detailiertere Beweispflichten kann ein solches Werkzeug erstellen.

- **Interaktive Theorembeweiser.** Der Beweis von Aussagen über Spezifikationen, wie die oben genannten Beweispflichten, kann sehr komplex und stark wiederholend sein, wodurch die Fehlerwahrscheinlichkeit bei manueller Ausführung steigt.

  Interaktive Theorembeweiser unterstützen daher den Entwickler beim Beweis solcher Aussagen: zu jedem Zeitpunkt präsentieren sie ihm neben der aktuell

zu zeigenden Aussage, dem Ziel oder Teilziel, eine Liste an Hypothesen, aus denen die Aussage folgen soll. Die Beweiser verfügen über eine Datenbank an gültigen Aussagen und Umformungsregeln und beherrschen Beweismethoden wie Fallunterscheidung, Widerspruchsbeweis usw., sodass sie einen vom Benutzer in der entsprechenden Notation eingegebenen Beweis überprüfen können. Zudem sind ihnen eine Reihe an Taktiken bekannt, über welche sie versuchen können, Ziele automatisiert zu zeigen. Dies kann den Gesamtaufwand je Beweis erheblich veringern[4].

Diese Art von Werkzeugen ist komplex, wird oft nur kommerziell im Rahmen intensiver industrieller Nutzung entwickelt und vertrieben und ist nicht für jede Spezifikationssprache verfügbar[5].

- **Animatoren.** Sowohl zur internen, als auch zur externen Validierung können Animatoren eingesetzt werden: dies sind Werkzeuge, welche im Rahmen der Spezifikation zustandsbasierter Systeme den spezifizierten Zustandsautomaten simulieren. So können zur internen Validierung für jeden erreichten Zustand Aussagen, wie etwa Invarianten oder Theoreme, überprüft werden (*Model Checking*). Zur externen Validierung kann dem Nutzer eine Schnittstelle präsentiert werden, über welche er das spezifizierte System praktisch erproben kann (*Rapid Prototyping*). Animatoren sind im Allgemeinen weniger komplex als interaktive Theorembeweiser und zu vielen Spezifikationssprachen erhältlich.

Verifizierung. Im Rahmen der Verifizierung sollen folgende Fragen beantwortet werden:

- „Baue ich das Produkt richtig?“ [Boe84]
- „Erfüllt die Software die spezifizierten Anforderungen?“ [IEEE1012]

Diese Fragen sind anhand der Realisierung zu klären, es ist deren Korrektheit gegenüber der Spezifikation zu diskutieren. Dabei ist das entscheidende Problem, dass oft eine große semantische Lücke zwischen dem Programm und der Spezifikation besteht [DS98]: Spezifikationssprachen operieren auf abstrakten mathematischen Objekten wie Mengen, Funktionen, Zuständen usw.; Implementierungssprachen hingegen kennen primitive und komplexe Datentypen wie Integer, Arrays oder Records. Dadurch wird die Verifizierung von Programmen, zu denen a priori keine Spezifikation vorlag, sehr aufwändig und komplex. Wird ein Programm jedoch auf Grundlage einer Spezifikation konstruiert (vgl. Unterabschnitt 2.6.1), liegt bereits eine (verifizierbar korrekte) Verfeinerung der Spezifikation in die semantische Domäne des Programmes vor, sodass die Verifizierung lediglich auf der korrekten Übersetzung des Programmes von der Spezifikations- in die Programmiersprache

[4] Da die Prädikatenlogik erster Ordnung im Allgemeinen nicht entscheidbar ist, sind vollautomatische Beweise nicht möglich.

[5] Allerdings existieren Beweissysteme, die nicht an eine konkrete Spezifikationsnotation gebunden sind und deren Eingaben sich prinzipiell aus allen mathematischen Aussagen erzeugen lassen.

beruht. Alternativ kann die Implementierung annotiert werden, um die semantische Lücke zu überbrücken.

Eine verifizierbar korrekte TCB-Implementierung erfüllt die Forderungen aus Unterabschnitt 2.2.2 nach Korrektheit der TCB-Realisierung.

2.6.3 Zustandsbasierte Spezifikation

Eine Teilmenge der Sprachen zur formalen Spezifikation dient der Beschreibung sequenzieller, zustandsbasierter Systeme. Dies sind genau solche Systeme, wie sie auch aus den Sicherheitsmodellen (siehe Abschnitt 2.3) hervorgehen. Im Allgemeinen verfügen solche Spezifikationssprachen über eine gemeinsame Semantik, welche im vorliegenden Unterabschnitt abstrakt, in mathematischer Notation, beschrieben wird. Spezifiziert wird jeweils eine Zustandsmaschine (ZSM), also ein System mit eigenem Zustand und darauf definierten Operationen. Die Maschine steht im Kontext von Typen, Konstanten und Axiomen.

Zur Schreibweise. Nachfolgend werden verschiedene Elemente einer Spezifikation benannt. Hierfür werden Symbole in serifenloser Schrift verwendet, um Verwechslungen mit Bezeichnern für Elemente in Sicherheitspolitiken auszuschließen. Abkürzend für $\mathsf{a}_1, \ldots, \mathsf{a}_n$ wird a_i geschrieben. Abkürzend für die geordnete Folge aller a_i wird a geschrieben.

Jedes a_i ist ein Symbol, dem gemäß der Spezifikation ein Typ und, je nach Art des Symbols, möglicherweise auch ein Wert zugeordnet ist. Für ein einstelliges Prädikat p meint der Ausdruck $\exists\, \mathsf{a} \cdot \mathsf{p}(\mathsf{a})$ dann: es existiert eine Funktion β, welche dem Symbol a einen Wert $\beta(\mathsf{a})$ so zuordnet, dass $\mathsf{p}(\beta(\mathsf{a}))$ wahr ist.

Kontext. In einer formalen Spezifikation definiert der Kontext zuerst eine Reihe von *Trägermengen* T_i. Jede Trägermenge ist dabei endlich oder abzählbar unendlich; ihre Elemente werden zumeist als atomar aufgefasst, sodass Trägermengen keine Mengen enthalten können. Alle Trägermengen sind paarweise disjunkt.

Zusammen mit einigen vordefinierten Mengen wie $\mathbb{B}$, $\mathbb{N}$ oder $\mathbb{Z}$ bilden die T_i die elementaren Typen der Spezifikation. Komplexe Typen werden dann aus elementaren Typen mittels Kreuzprodukt und Potenzmenge gebildet, oftmals werden auch explizit Relationen und Funktionen unterstützt.

Ebenfalls zum Kontext gehören eine Reihe von *Konstanten* c_i. Jeder Konstante ist ein Typ zugeordnet[6]. Auf den Konstanten können dann *Axiome* genannte, prädikatenlogische Aussagen $\mathsf{A}_i(c)$ definiert werden, die als wahr anzunehmen sind. Zusätzlich lassen sich *Theoreme* $\mathsf{B}_i(c)$ angeben, welche sich aus Axiomen herleiten lassen müssen und daher zu beweisen sind.

Zustandsmaschine. Eine Zustandsmaschine besteht zunächst aus einem Zustand, zusammengesetzt aus mehreren *Variablen* x_i, welche wiederum über einen festen Typ verfügen.

[6] Viele Spezifikationssprachen erlauben Nichtdeterminismus, sodass die Konstanten nicht über einen festen Wert verfügen. Ihre möglichen Werte können jedoch über Axiome eingeschränkt werden.

Auf dem Zustand werden *Invarianten* $\mathsf{I}_i(\mathsf{x}, \mathsf{c})$ spezifiziert; dies sind prädikatenlogische Aussagen, welche gültige Zustände einschränken. Ein korrektes System darf niemals in einen Zustand gelangen, der eine der Invarianten verletzt. Zusätzlich lassen sich aus den Invarianten beweisbare *Theoreme* $\mathsf{J}_i(\mathsf{x}, \mathsf{c})$ angeben.

Zu jeder Variablen x_i wird ein Wert $\mathsf{v}_i(c)$ angegeben; die einzelnen v_i bilden dann den *Initialzustand*.

Auf dem Zustand lassen sich nun eine Reihe von *Operationen* $\mathsf{o}_i : [\mathsf{p}, \mathsf{pre}, \mathsf{post}]$ definieren. Jede Operation verfügt über *Parameter* p_i, jedem ist ein fester Typ zugeordnet. Die *Vorbedingung* $\mathsf{pre}(\mathsf{p}, \mathsf{x}, \mathsf{c})$ einer Operation ist eine prädikatenlogische Aussage, welche die Bedingungen angibt, unter der eine Ausführung der Operation gestattet ist. Die Semantik der Operation wird über ihre *Nachbedingung* $\mathsf{post}(\mathsf{x}', \mathsf{p}, \mathsf{x}, \mathsf{c})$ spezifiziert, welche den Vorzustand x mit dem Nachzustand x' in Relation setzt.

Beweispflichten. Aussagen zur internen Konsistenz einer Spezifikation lassen sich über Beweispflichten (engl. *Proof Obligations*) treffen. Jede Beweispflicht ist dabei eine Aussage, die für eine konsistente Spezifikation gültig sein muss. Nachfolgend werden wesentliche Beweispflichten aufgeführt. Details zu den Beweispflichten konkreter Sprachen finden sich in [Abr96; Cle11; Eve12].

Im Rahmen einer allgemeinen Spezifikation ist dabei folgendes zu zeigen.

(P1) *Wohldefiniertheit* [Eve12]. Alle Ausdrücke müssen wohldefiniert sein, d.h. ihr Wert muss im entsprechenden Kontext definiert sein.

(P2) *Typkorrektheit.* Alle Operationen müssen entsprechend der Typen korrekt verwendet werden.

(P3) *Widerspruchsfreiheit von Axiomen.* Es ist zu zeigen, dass eine Belegung der Konstanten existiert, sodass die Axiome erfüllt sind, also $\exists\, \mathsf{c} \cdot \mathsf{A}(\mathsf{c})$.

(P4) *Gültigkeit von Theoremen.* Theoreme, also Aussagen, welche aus anderen Aussagen folgen sollen, sind zu beweisen.

Bei Zustandsmaschinen ist zusätzlich folgendes zu zeigen.

(P5) *Widerspruchsfreiheit der Invarianten* [Abr89]. Es ist zu zeigen, dass ein Zustand existiert, der die Invariante erfüllt, also $\exists\, \mathsf{x} \cdot \mathsf{I}(\mathsf{x}, \mathsf{c})$. Dies geschieht bereits durch Angabe eines Initialzustandes.

(P6) *Konsistenz des Initialzustandes* [Abr89]. Es ist zu zeigen, dass der Initialzustand die Invariante erfüllt, also $\mathsf{I}(\mathsf{v}(\mathsf{c}), \mathsf{c})$.

(P7) *Konsistenz der Operationen* [Abr89]. Zu jeder Operation ist zu zeigen, dass die Nachbedingung dieser Operation die Invariante herstellt, also $\forall\, \mathsf{x}', \mathsf{p}, \mathsf{x} \cdot \mathsf{I}(\mathsf{x}, \mathsf{c}) \land \mathsf{pre}(\mathsf{p}, \mathsf{x}, \mathsf{c}) \land \mathsf{post}(\mathsf{x}', \mathsf{p}, \mathsf{x}, \mathsf{c}) \Rightarrow \mathsf{I}(\mathsf{x}', \mathsf{c})$.

(P8) *Aktivierbarkeit der Operationen.* Zu jeder Operation o_i ist folgendes zu zeigen: es existiert eine Folge von Zuständen $\mathsf{x}^0, \mathsf{x}^1, \ldots, \mathsf{x}^n$ mit $n \geq 1$ und $\mathsf{v}(\mathsf{c}) = \mathsf{x}^0$ so, dass

(a) zu jedem Paar $\mathsf{x}^{\ell}, \mathsf{x}^{\ell+1}$ mit $0 \leq \ell < n$ eine Operation $\mathsf{o}_j : [\mathsf{p}, \mathsf{pre}, \mathsf{post}]$ und eine Belegung für p existieren, sodass $\mathsf{pre}(\mathsf{p}, \mathsf{x}^{\ell}, \mathsf{c}) \wedge \mathsf{post}(\mathsf{x}^{\ell+1}, \mathsf{p}, \mathsf{x}^{\ell}, \mathsf{c})$.

(b) zu dem Paar $\mathsf{x}^{n-1}, \mathsf{x}^{n}$ diese Operation o_j die betrachtete Operation o_i ist.

2.6.4 Sprachen zur zustandsbasierten Spezifikation

Es existieren mehrere Sprachen zur Notation zustandsbasierter Spezifikationen. Die vorliegende Masterarbeit beschreibt den Spezifikationsprozess, soweit möglich, anhand des abstrakten Schemas aus Unterabschnitt 2.6.3 sowie im konkreten Fall anhand einer ausgewählten Spezifikationssprache. Hierfür stellt dieser Unterabschnitt verschiedene Spezifikationssprachen vor, gibt einen knappen Überblick über Prinzipien und Unterschiede und zeigt auf, weshalb *Event-B*, die gewählte Sprache, am besten für diese Arbeit geeignet ist.

Dabei werden die folgenden Kriterien zur Bewertung verwendet [Lam00].

- **Ausdruckskraft.** Sind sie auf die domänenspezifische Semantik angepasst, können interessante Eigenschaften ohne großen Aufwand notiert werden.

- **Konstruierbarkeit, Änderbarkeit.** Sprachen sollten die inkrementelle Konstruktion einer Spezifikation durch geeignete Techniken zur Strukturierung und Modularisierung ermöglichen.

- **Benutzbarkeit, Kommunizierbarkeit.** Einerseits muss es für entsprechend geschultes Personal möglich sein, Spezifikationen von guter Qualität zu schreiben. Wichtig ist dabei sind eine einfache theoretische Grundlage und eine einfache Notation. Andererseits sollte es für entsprechend geschultem Personal möglich sein, qualitativ hochwertige Spezifikationen zu lesen und zu überprüfen.

- **Mächtige und effiziente Analysetechniken.** Die Prüfung der zu Beginn dieses Abschnittes genannten Anforderungen an eine Spezifikation sollte möglichst effizient und zu einem möglichst hohen Grad automatisiert geschehen. Werkzeuge wie die oben genannten können dies leisten.

Zusätzlich stellt diese Masterarbeit folgende Anforderungen.

- **Freie Verfügbarkeit.** Zum Einsatz in Forschung und Lehre sollten eingesetzte Techniken frei Verfügbar und möglichst selbst Ergebnis offener Forschungsarbeit sein.

- **Reife.** Die Tauglichkeit der Spezifikationssprachen sowie eingesetzter Werkzeuge sollte in industriellen Projekten erprobt sein.

- **Erweiterbarkeit.** Je stärker eine Spezifikationstechnik an die jeweilige Domäne angepasst ist, desto nützlicher ist sie [Lam00]. Sind Sprache oder Werkzeuge erweiterbar, kann diesem Umstand Rechnung getragen werden.

Da keine geeigneten Sprachen zur formalen Spezifikation von Sicherheitsmodellen bekannt sind, handelt es sich bei allen nachfolgend vorgestellten Sprachen um formale Sprachen zur Spezifikation sequenzieller, zustandsbasierter Systeme. Jede verfügt über eine definierte Syntax, Semantik und ein Typsystem.

Z. Die Sprache Z [Spi98; ISO13568; WD96; Nis99] ist eine formale Sprache zur Spezifikation allgemeiner Systeme. Sie enthält einerseits eine Sprache zur Notation mathematischer Aussagen, welche aus der Prädikatenlogik erster Ordnung sowie der Mengentheorie besteht und deren Notation eng den üblichen mathematischen Konventionen folgt. Andererseits bietet Z mit den sog. Schemas eine Möglichkeit zur Strukturierung der Spezifikation sowie des zu spezifizierenden Systems. Diese Notation ist allgemein gehalten, lässt sich jedoch insbesondere zur Spezifikation zustandsbasierter Systeme nutzen. Es existieren verschiedene Darstellungsformen für Z, so etwa zur Verwendung in E-Mails und in LaTeX.

Z kommt größtenteils ohne Schlüsselworte oder enge Anforderungen an die Spezifikationsdokumente aus; die formalen Teile können mit informellem, natürlichsprachlichem Text gemischt werden. Eine Z-Spezifikation liest sich dann exakt wie ein mathematisches Dokument, sodass die Hürde zum Verständnis verhältnismäßig klein ist. Dennoch können automatische Tools die Spezifikation verarbeiten und etwa zur Typprüfung nutzen.

Z ist nicht an eine spezielle Methode zur Spezifikation gebunden [HJN93], was es einerseits universell nutzbar macht. Andererseits wird die Nutzung zur formalen Spezifikation zustandsbasierter Softwaresysteme damit komplizierter: ohne explizite syntaktische Unterstützung wird der Spezifikationsprozess schwieriger und Fehler werden wahrscheinlicher. Zudem fehlt die Möglichkeit, Zusammenhänge auch für computergestützte Werkzeuge zu beschreiben.

Besonderheiten bei der Spezifikation von Zustandsmaschinen bestehen vor allem

1. in der relationalen Schreibweise der Zustandsüberführung. Dabei werden Variablen des Vor- und Nachzustandes miteinander ins Verhältnis gesetzt; wird eine Variable des Nachzustandes nicht explizit eingeschränkt, kann sie jeden beliebigen Wert annehmen (!).

2. in den nicht explizit dargestellten Vor- und Nachbedingungen. Diese werden zusammen, höchstens optisch durch einen Leerraum abgetrennt, als eine Menge an Prädikaten notiert.

Beide Besonderheiten stellen potentielle Fehlerquellen dar.

Zu Z existieren eine Reihe an Tools, darunter Typprüfer, Animatoren und Beweissysteme, welche jedoch größtenteils veraltet sind und nicht mehr gepflegt werden. Zu erwähnen sind hier insbesondere der Typprüfer *fuzz*[7] sowie der Animator *jaza*[8], welche beide die Spivey-Version der Sprache unterstützen, außerdem die veralteten Beweisumgebungen *CadiZ*[9], *ProofPower*[10] und *HOL/Z*[11]; zudem die

[7] Siehe `http://spivey.oriel.ox.ac.uk/mike/fuzz/`.
[8] Siehe `http://www.cs.waikato.ac.nz/~marku/jaza/`.
[9] Siehe `http://www.cs.york.ac.uk/hise/cadiz/home.html`.
[10] Siehe `http://www.lemma-one.com/ProofPower/index/index.html`.
[11] Siehe `http://www.brucker.ch/projects/hol-z/`

aktuell in der Entwicklung befindlichen *Community Z Tools*[12], welche insbesondere einen Typprüfer und einen Animator im Anfangsstadium umfassen. Z wurde im Rahmen vieler industrieller Projekte eingesetzt [WLB+09].

Z ist eine allgemein gehaltene Sprache ohne zugrunde liegende Spezifikationsmethode oder spezialisierte Sprachkonstrukte, was sie schlecht benutzbar macht. Die verfügbaren Werkzeuge sind veraltet und nicht mächtig genug.

VDM. Die *Vienna Development Method Specification Language* (VDM-SL) [Jon90; ISO13817] ist eine ausgereifte Spezifikationssprache, speziell auf die Spezifikation zustandsbasierter Systeme zugeschnitten und Teil der *Vienna Development Method*, einer speziellen Methode zur Softwareentwicklung. Die Erweiterungen VDM++ und VDM-RT der Sprache bieten verbesserte Modularisierung bzw. Sprachfeatures für Echtzeitsysteme.

Die VDM-SL wurde speziell zur Softwareentwicklung entwickelt und verfügt über entsprechende Sprachkonstrukte. Ähnlich wie in Z werden Zustände in VDM-SL relational überführt. Besonders ist der Umstand, dass (im Gegensatz zur Notation in Unterabschnitt 2.6.3) die Nachbedingungen einer Operation die Erhaltung der Invariante implizit einschließen, sodass entsprechende Bedingungen nicht explizit notiert werden müssen [BR93].

VDM-SL wurde in vielen industriellen Projekten eingesetzt [WLB+09]. Es existieren umfangreiche Entwicklungsumgebungen, darunter die kommerziellen *VDM Tools*[13] sowie die Eclipse-basierte OpenSource-Umgebung *Overture*[14]. Die Werkzeuge unterstützen Typprüfung, LaTeX-Export sowie die Erzeugung von Beweispflichten, enthalten jedoch keinen interaktiven Theorembeweiser. Ein frei verfügbarer, auf VDM zugeschnittener Beweiser ist dem Autor nicht bekannt.

Während VDM-SL besser benutzbar als Z und zudem sehr ausgereift ist, fehlen der Sprache mächtige, frei verfügbare Analysewerkzeuge.

B. Eng verwandt mit Z und VDM ist B [Abr96]. Neben der Spezifikation von Zustandsmaschinen verfügt die Sprache über explizite Mittel zu deren Verfeinerung, bis hin zu einer Abstraktionsebene, von der aus die Spezifikation direkt in eine Implementierung überführt werden kann.

Anders als in Z und VDM werden Zustandsübergänge in einer von [Dij76] inspirierten, *verallgemeinerte Ersetzungen* genannten Notation spezifiziert, welche sehr ausdrucksstark ist und die Konzepte von Vorbedingungen, Verzweigungen, nichtdeterministischer Auswahl und Variablenbindung verallgemeinert. Nach [BR93] erscheint diese Schreibweise algorithmischer und konkreter als die relationale Notation. Insbesondere bezüglich den Verfeinerungen weist die Sprache jedoch starke semantische Einschränkungen auf, welche darauf schließen lassen, dass B eher für technische, vorallem auf Ganzzahlenarithmetik basierende Systeme gedacht ist.

Auch B wurde umfangreich in industriellen Anwendungen eingesetzt [WLB+09]. Die vorrangig genutzte Entwicklungsumgebung ist das kommerzielle, jedoch frei

[12] Siehe `http://czt.sourceforge.net/`.
[13] Siehe `http://www.vdmtools.jp/en/`.
[14] Siehe `http://www.overturetool.org/`.

erhältliche *AtelierB*[15], das neben Typprüfer, automatischer Verfeinerung und Codeerzeugung einen guten interaktiven Beweiser enthält. Zudem existiert zu B der quelloffene Animator und Model Checker *ProB*[16].

B ist eine ausdrucksstarke Sprache mit mächtigen und reifen Analysewerkzeugen, die jedoch größtenteils dem kommerziellen Umfeld entspringen und nicht erweiterbar sind. Die Sprache ist im Rahmen der Spezifikation von Softwaresystemen gut benutzbar und kommunizierbar, jedoch entspricht die fachliche Domäne nicht jener dieser Masterarbeit.

Event-B. Die Sprache *Event-B* [Abr10; Eve12] ist eine Verallgemeinerung und Erweiterung von B.

Die Sprache gestattet eine höhere Modularität der Spezifikationen als B. Der Begriff der Zustandsmaschinen wurde verallgemeinert; statt Operationen sind nun atomare „Ereignisse“ für Zustandsübergänge zuständig. Event-B hebt viele semantische Einschränkungen von B auf; dies betrifft insbesondere die Verfeinerung von Spezifikationen.

Auch zu Event-B existieren eine Reihe an industriellen Fallstudien [Evea]. Die vorrangig genutzte Entwicklungsumgebung ist *Rodin*[17], eine Eclipse-basierte Open-Source-Werkzeugsammlung, die über Plugins erweiterbar ist. Die Erweiterbarkeit geht hier bis hin zur Ergänzung neuer Sprachfunktionen. In Rodin ist ein interaktiver Theorembeweiser enthalten. Während die Software mit einem quelloffenen Beweiser ausgeliefert wird, existieren frei erhältliche, aber proprietäre Plugins, um die Beweiser von AtelierB (siehe oben) in Rodin zu integrieren: diese sind hochwertiger und werden von den Rodin-Entwicklern empfohlen [Eve12]. Rodin wird im Rahmen von Forschungsprojekten aktiv weiterentwickelt. Der B-Animator ProB lässt sich auch für Event-B nutzen.

Event-B ist nochmals ausdrucksstärker als B, die höhere Modularität erlaubt leichte Konstruierbarkeit und Änderbarkeit der Spezifikationen. Die Sprache ist in ihrem Design ausgereift und daher gut benutzbar und kommunizierbar. Mächtige Analysewerkzeuge sind frei verfügbar, jedoch noch nicht ganz ausgereift. Das gesamte System ist auf gute Erweiterbarkeit ausgelegt.

Unter den vorgestellten Sprachen sind VDM sowie Event-B die geeignetsten Sprachen, jedoch bietet Event-B die beste Werkzeugunterstützung (nämlich bezüglich Erweiterbarkeit und Mächtigkeit) und das ausgereifteste Sprachdesign und wird daher in der vorliegenden Masterarbeit zum Einsatz kommen.

2.7 Zusammenfassung

Das vorliegende Kapitel erläuterte dieser Arbeit zugrunde liegende Konzepte und Zusammenhänge. Nachdem Abschnitt 2.1 wesentliche mathematische Definitionen einführte, stellte Abschnitt 2.2 den *Security-Engineering*-Prozess als thematischen

[15] Siehe `http://www.atelierb.eu/en/`.
[16] Siehe `http://www.stups.uni-duesseldorf.de/ProB/`.
[17] Siehe `http://www.event-b.org/`.

Rahmen dieser Arbeit vor: dabei geht aus der Anforderungsanalysephase des Softwareentwicklungsprozesses (vgl. Unterabschnitt 2.2.1) eine Sicherheitspolitik sowie deren formale Repräsentation, das Sicherheitsmodell, hervor. Letzteres beschrieb Abschnitt 2.3 ausführlich. In der Realisierungsphase (vgl. Unterabschnitt 2.2.2) muss dieses Modell dann wirksam realisiert werden, und zwar in einer Sicherheitsarchitektur sowie einer Implementierung. Die Anforderungen an solche Realisierungen beschrieb Unterabschnitt 2.2.2. Das Kapitel stellte zudem die *Trusted Computing Base* als die Menge derjenigen Systemfunktionen vor, welche von entscheidender Bedeutung für die vertrauenswürdige Umsetzung der Sicherheitseigenschaften sind. Deren korrekte Realisierung ist unerlässlich, wenn Vertrauen in die Sicherheitseigenschaften des Systems bestehen soll. Abschnitt 2.2 argumentierte, dass zur Erfüllung der Korrektheitsanforderung die geringe Größe und Komplexität der TCB und ihrer Realisierungen nötig ist.

Den Ansatz zur Lösung dieses Problems, welcher dieser Arbeit zugrunde liegt, beschrieb Abschnitt 2.5: das *TCB Engineering* nutzt den kausalen Zusammenhang zwischen den informalen Anforderungen, dem Sicherheitsmodell, der Sicherheitsarchitektur sowie der Implementierung, um Umfang und Komplexität der TCB und ihrer Realisierung gezielt zu steuern und so zu minimieren. Diese Arbeit realisiert dabei einen Teil der kausalen Kette, indem sie die formale Spezifikation der Funktionen der TCB leistet, aus welcher dann eine vertrauenswürdige Realisierung erstellt werden kann. Die nötigen Grundlagen zur formalen Spezifikation legte Abschnitt 2.6.

KAPITEL 3

Entwicklung der Methode

Das Ziel dieser Masterarbeit ist es, Funktionen einer Trusted Computing Base formal zu spezifizieren. Wie Kapitel 2 argumentiert, beschreiben solche Funktionen die sicherheitskritischen Teile des Systemverhaltens und werden somit in Sicherheitsmodellen formalisiert. Das vorliegende Kapitel erarbeitet nun die Methode, welche die Spezifikation der Funktionen auf Grundlage des Sicherheitsmodelles leistet. Die so entstandene Spezifikation ermöglicht dann eine korrekte Implementierung der Trusted Computing Base, auf welcher die Sicherheitseigenschaften des zu entwickelnden Softwaresystems beruhen.

Der Abschnitt 3.1 stellt die Rahmenbedingungen der Methode dar und beschreibt dabei die Struktur des Ausgangsdokumentes, des Zieldokumentes sowie die Anforderungen an die Überführung vom Ausgangs- zum Zieldokument. Anschließend benennt Abschnitt 3.2 solche Strategien zur Behandlung dieser Anforderungen, die auf den gesamten Spezifikationsprozess zutreffen. Einer dieser Strategien zufolge soll die Abstraktion zwischen Metamodellen und Sicherheitsmodellen zur Wiederverwendung genutzt werden: daher widmen sich die Abschnitte 3.3 und 3.4 sowie die Abschnitte 3.5 und 3.6 jeweils getrennt der Spezifikation von Metamodell und konkretem Sicherheitsmodell, und zwar einerseits abstrahiert von jeder konkreten Spezifikationssprache, andererseits anhand von Event-B.

3.1 Rahmenbedingungen

Die in diesem Kapitel zu spezifizierenden TCB-Funktionen beschreiben also den sicherheitskritischen Teil des Systemverhaltens und werden anhand von zustandsbasierten Sicherheitsmodellen (vgl. Abschnitt 2.3) formal beschrieben. Daher wird diese Arbeit das entsprechende Sicherheitsmodell als Grundlage zur formalen Spezifikation der TCB-Funktionen verwenden. Um die Komplexität der Spezifikation und somit die Fehlerwahrscheinlichkeit so gering wie möglich zu halten, muss die Überführung des Sicherheitsmodells in eine Spezifikationsnotation lediglich einen Wechsel der Darstellungsform bedeuten; nötig ist also eine zustandsbasierte Spezifikationssprache. Änderungen und Verfeinerungen des Inhaltes können dann anhand der formalen Spezifikation in einem geordneten Rahmen unter kontrollierten Bedingungen durchgeführt werden (vgl. Unterabschnitt 2.6.1).

Während dieses Kapitel eine Methode zu dieser Überführung entwickelt, beschreibt der vorliegende Abschnitt zuerst die Rahmenbedingungen dieser Überführung genauer, nämlich deren Eingabedokument (das Sicherheitsmodell), das

Ausgabedokument (die Spezifikation), sowie die Anforderungen, welche an die Überführung bestehen.

Das Eingabedokument für die zu erarbeitende Methode ist also ein zustandsbasiertes Sicherheitsmodell, wie in Abschnitt 2.3 beschrieben: es basiert auf dem einheitlichen Modellkern, erweitert bzw. spezialisiert diesen um statische und dynamische Komponenten und enthält ein anwendungsspezifisches Autorisierungsschema. Das Sicherheitsmodell ist zugleich Instanz eines Metamodells, mit welchem auch auf Spezifikationsebene Wiederverwendung praktiziert werden kann. Allen Instanzen eines Metamodells gemein sind Wertebereiche, Anzahl, Semantik und Typen statischer und dynamischer Komponenten, die Primitivoperationen sowie die Primitivprädikate.

Das Zieldokument soll eine formale Spezifikation sein, wie sie Abschnitt 2.6 beschreibt, formuliert in einer Notation für sequenzielle, zustandsbasierte Systeme. Eine solche Spezifikation enthält also die Beschreibung einer Zustandsmaschine (ZSM) mit Zustandsraum, Initialzustand, Invarianten und Operationen, gesetzt in einen mathematischen Kontext aus Trägermengen, Konstanten und Axiomen.

Modell	**Spezifikation**
Wertebereiche T statische Komponenten S dynamische Komponenten D Primitivoperationen o Primitivprädikate p	Trägermengen T Konstanten c Axiome A(c), Theoreme B(c)
Autorisierungsschema $\kappa, \pi, \Sigma, \delta$ Ausgabefunktion λ, -raum Γ Werte statischer Komponenten v^S Initialzustand v^D	Variablen x Invarianten I(x, c), Theoreme J(x, c) Initialzustand v(c) Operationen o : [p, pre, post]

Tabelle 3.1: Abstrakte Inhalte eines Sicherheitsmodells sowie seiner formalen Spezifikation.

Eine Gegenüberstellung der abstrakten Inhalte von Modell und Spezifikation enthält Tabelle 3.1. Die Spezifikation soll dabei sowohl den Anforderungen an eine TCB (vgl. Unterabschnitt 2.2.2), als auch an eine formale Spezifikation (vgl. Abschnitt 2.6) genügen, zusätzlich ist die Zielstellung der Masterarbeit (vgl. Abschnitt 1.2) zu berücksichtigen. Es ergeben sich die folgenden Anforderungen.

(S1) **Vollständigkeit.** Die Spezifikation soll das Sicherheitsmodell vollständig umsetzen. Das heißt, dass ein Homomorphismus zwischen beiden bestehen muss derart, dass sich jede Eingabe und jeder Zustand des Sicherheitsmodells auf eine Eingabe und einen Zustand des spezifizierten Systems abbilden lässt. Gemeinsam mit (S2) ergibt dies einen Isomorphismus zwischen Modell und Spezifikation.

(S2) **Externe Konsistenz.** Die Spezifikation soll konsistent zum Sicherheitsmodell sein. Ergänzend zu (S1) bedeutet dies einen Homomorphismus zwischen

beiden: hier aber müssen umgekehrt Eingaben und Zustände des Systems auf Eingaben und Zustände des Modells abbildbar sein. Dies vervollständigt den in (S1) erwähnten Isomorphismus zwischen Modell und Spezifikation.

(S3) **Interne Konsistenz.** Zur Spezifikation sollen alle Beweispflichten (P1) bis (P8) erfüllt sein.

(S4) **Präzision.** Wesentliche Konzepte sollen klar aus der Spezifikation hervorgehen, die Formulierungen sollen eindeutig sein.

(S5) **Verständlichkeit.** Die Spezifikation soll in Aufbau, Formatierung, Wahl der Bezeichner und Dokumentiertheit so beschaffen sein, dass sie durch einen Betrachter mit Domänenwissen leicht und möglichst intuitiv korrekt verstanden werden kann.

(S6) **Wiederverwendbarkeit.** Die Abstraktionsbeziehung zwischen konkretem Sicherheitsmodell und Metamodell soll zur Wiederverwendung genutzt werden, sodass aus einem einmal spezifizierten Metamodell leicht die Spezifikationen von Instanzen des Metamodells abgeleitet werden können.

(S7) **Geringe Größe und Komplexität.** Die Spezifikation soll, unter Bedingung der übrigen Anforderungen, insbesondere jedoch (S1) und (S5), möglichst klein sein und geringe Komplexität aufweisen. Ziel dieser Anforderung ist die Minimierung der Fehlerwahrscheinlichkeit in der Spezifikation und der abgeleiteten Realisierung.

[Bos95] bemerkt, dass Redundanzen in Spezifikationen zur Fehlerbeseitigung dienen können. Das steht in scheinbarem Widerspruch zu dieser Anforderung, vergrößern doch Redundanzen die Spezifikation quantitiv. Tatsächlich tragen sie bei entsprechender Strukturierung, Kennzeichnung und beweisbarer Äquivalenz nicht zur Komplexität bei und helfen Fehler zu vermeiden.

3.2 Strategien zur Anforderungsbehandlung

Zu den im vorherigen Abschnitt vorgestellten Anforderungen an den Spezifikationsvorgang erarbeitet der vorliegende Abschnitt Behandlungsstrategien, welche notwendiger Bestandteil und zugleich Grundlage für die in diesem Kapitel erarbeitete Vorgehensweise sind. Hierzu behandelt Unterabschnitt 3.2.1 jede der Anforderungen und erörtert mögliche Lösungsansätze. Unterabschnitt 3.2.2 beschäftigt sich vertiefend mit Konzepten zur Modularisierung, und Unterabschnitt 3.2.3 schließlich stellt bewährte Vorgehensweisen vor, die verschiedenen Anforderungen zuträglich sind.

3.2.1 Anforderungsspezifische Strategien

Den Anforderungen (S1) und (S2) der **Vollständigkeit und externen Konsistenz** begegnet die vorliegende Arbeit insbesondere methodisch. Der Abschnitt 2.3 stellte die hier genutzte Auffassung eines Sicherheitsmodelles vor und zählte alle darin

enthaltenen Elemente auf. Sicherheitsmodelle, welche von dieser Beschreibung abweichen, liegen außerhalb des Bereichs der Arbeit und müssen gesondert betrachtet werden. Anhand der Aufzählung aller Elemente eines Modells kann dieses Schritt für Schritt überführt werden, sodass die Vollständigkeit der Spezifikation erreicht wird. Die detailierte Unterscheidung verschiedener Fälle sowie die genaue Beschreibung der einzelnen Teilschritte senken die Komplexität der Überführung und ermöglichen so das Erstellen konsistenter Spezifikationen.

Während und nach dem Spezifikationsvorgang sollte die externe Konsistenz über einen Animator stichprobenartig geprüft werden. Dabei sind, ausgehend vom Initialzustand, Operationen an der spezifizierten Zustandsmaschine auszuführen und die Ergebnisse mit den aus dem Sicherheitsmodell erwarteten Effekten zu vergleichen. Testfälle sollten sich erstens an der Struktur des Modells, zweitens an Beispielszenarien und drittens an komplexen Bestandteilen der Spezifikation orientieren.

Darüber hinaus ist der Einsatz aufwändigerer Vorgehensweisen zur weiteren Steigerung des Vertrauens in Vollständigkeit und externe Konsistenz denkbar; so kann parallel und unabhängig zur Spezifikation eine prototypische Implementierung des Sicherheitsmodells erstellt und diese (halb-)automatisch mit der Spezifikation verglichen werden.

Zur Sicherung der **internen Konsistenz** (S3) kann auf fortgeschrittene Techniken zur Validierung zurückgegriffen werden (vgl. Unterabschnitt 2.6.2); insbesondere die von der Entwicklungsumgebung bereitgestellten Werkzeuge Syntax- und Typprüfer, Erzeuger für Beweispflichten, der interaktive Theorembeweiser sowie der Animator bieten verschiedenste Möglichkeiten, die interne Konsistenz mit mathematischen Methoden zu beweisen. Das Erzeugen und interaktive Beweisen der Beweispflichten hilft dabei, statische Aspekte der Spezifikation zu validieren, während ein Animator über die Technik des Model Checking den Zustandsraum der Spezifikation erkundet und dabei prüft, ob eine Belegung für die Variablen des Modells gibt und keiner der erkundeten Modellzustände den Invarianten widerspricht.

Für die Anforderungen (S4) und (S5), **Präzision und Verständlichkeit**, ist durch die Nutzung einer formalen, mathematischen Notation wie in Abschnitt 2.6 beschrieben bereits ein wichtiger Beitrag geleistet. Innerhalb der Spezifikation sollte die Wahl von Namen und Bezeichnern stets sorgfältig und so erfolgen, dass die potentiellen Leser der Spezifikation den Zusammenhang zwischen Bezeichner und Bezeichnungsgegenstand erkennen und ihn gegebenenfalls ihrem Domänenwissen zuordnen können. Die Spezifikation soll gut dokumentiert und kommentiert werden; im besten Fall ist sie selbstdokumentierend. Generell sollten Redundanzen vermieden oder gekennzeichnet bzw. vom übrigen Inhalt getrennt dargestellt werden.

Zur Verständlichkeit sowie zur **Wiederverwendbarkeit** (S6) trägt auch die Modularisierung der Spezifikation bei, Näheres erläutert Unterabschnitt 3.2.2. Wiederverwendung, die über die Metamodell-Instanz-Beziehung hinausgeht – etwa zwischen Metamodellen – ist nur begrenzt sinnvoll: sofern überhaupt Gemeinsamkeiten zwischen verschiedenen Metamodellen bestehen, können wenig komplexe Bestandteile leicht kopiert werden; bei komplexeren Bestandteilen hingegen ist die Spezifikation stark vom konkreten Metamodell abhängig und muss ohnehin für den Einzelfall zugeschnitten werden. Letztlich bieten die bekannten Spezifikationssprachen und

-paradigmen hier keine sinnvolle Unterstützung.

Die Grundlage zur Behandlung der Anforderung (S7), **geringe Größe und Komplexität**, bildet die Nutzung einer präzisen, mathematischen Notation in Verbindung mit der schrittweisen Überführung des mathematischen Modells in die Darstellungsform der Spezifikation (vgl. Abschnitt 3.1). So kann das Modell verständlich übertragen und Redundanzen können mittels der präzisen, formalen Notation gut kontrolliert werden. Nötige Redundanzen sind, wie oben erläutert, zu kennzeichnen und deren Äquivalenz anhand von Theoremen zu beweisen. Werden Metamodell und dessen Instanz separat spezifiziert (siehe unten in Unterabschnitt 3.2.2), so sollte Komplexität eher in die Spezifikation des Metamodells verlagert und dort gut gekennzeichnet werden: die häufiger anfallende Arbeit der Spezifikation eines konkreten Sicherheitsmodells ist so weniger komplex und somit weniger fehleranfällig.

3.2.2 Modularisierung

Wie in (S6) erläutert kann die Abstraktion zwischen Metamodell und konkretem Sicherheitsmodell dazu genutzt werden, die Spezifikation eines Metamodells und eines konkreten Sicherheitsmodells zu trennen; die Spezifikation wird dadurch modularisiert. Die Spezifikationsmethodik kann so in dieser Arbeit getrennt erarbeitet und später getrennt angewendet werden. Je nach verwendeter Notation können Spezifikationen zusätzlich in einzelne Dokumente zerlegt werden; die Modularisierung verbessert so auch die Verständlichkeit (S5).

Der Rest des Unterabschnittes beschreibt das zugehörige Gliederungskonzept getrennt für allgemeine Spezifikationen, sowie für Spezifikationen in Event-B.

Sprachunabhängige Spezifikation. Gegenstand einer Spezifikation, wie sie oben vorgestellt wurde, ist eine Zustandsmaschine; ein Sicherheitsmodell wird also in die Spezifikation einer Zustandsmaschine überführt. Ein Weg, die Abstraktion zwischen Metamodell und dessen Instanz in einer Spezifikation auszudrücken läge darin, mittels einer Vererbungsbeziehung im Sinne der Objektorientierung das Metamodell als abstrakte Zustandsmaschine zu modellieren und jedes Sicherheitsmodell davon abzuleiten. Dies ist jedoch aufgrund von Einschränkungen der bekannten Spezifikationssprachen nicht möglich. Es bleibt, das Metamodell außerhalb der Zustandsmaschine, also in einem Kontext (im Sinne von Unterabschnitt 2.6.3) zu spezifizieren: die zum Metamodell gehörigen Modellbestandteile müssen daher durch Trägermengen, Konstanten und Axiome ausgedrückt werden. Ein konkretes Sicherheitsmodell wird dann spezifiziert, indem unter Verwendung des Metamodell-Kontextes eine Zustandsmaschine nach einem festen Schema erarbeitet wird.

Spezifikation in Event-B. Event-B unterstützt die Trennung zwischen Spezifikation von Kontext und Zustandsmaschine, sodass dieses Konzept übernommen werden kann. Darüber hinaus ist es möglich, innerhalb eines Kontextes einen oder mehrere fremde Kontexte zu importieren (Event-B spricht von einer *Erweiterung* des Kontextes), sodass Kontexthierarchien aufgebaut werden können (siehe Abbildung 3.1), um Wiederverwendbarkeit und Verständlichkeit weiter Rechnung zu

tragen. Die Erweiterung eines Kontextes um einen oder mehrere fremde Kontexte geschieht durch Vereinigung seiner Inhalte mit den fremden Inhalten; Namenskonflikte resultieren in einem Fehler.

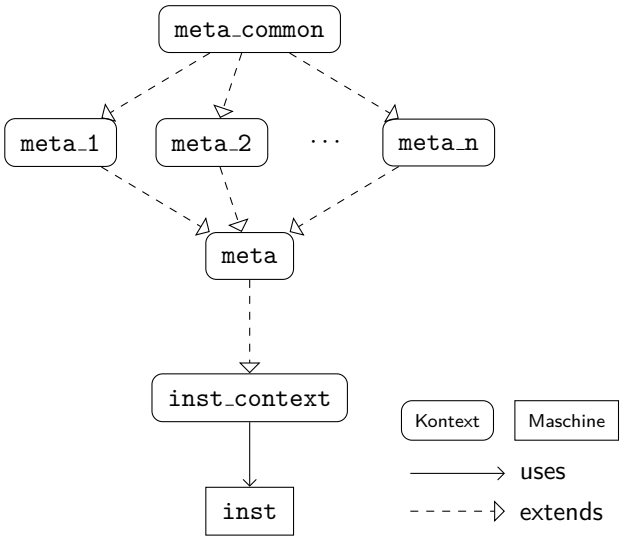

Abbildung 3.1: Beispielhafte Kontexthierarchie in der Event-B-Spezifikation eines Metamodells sowie einer Instanz dessen. Das Metamodell, dessen gesamte Spezifikation hier über den Kontext „`meta`“ zugreifbar ist, kann aus mehreren Kontext-Dokumenten zusammengesetzt werden (etwa „`meta_1`“, „`meta_2`“, . . . , „`meta_n`“), welche eine Reihe gemeinsamer Definitionen verwenden können (etwa „`meta_common`“). Die Spezifikation des konkreten Sicherheitsmodells, welche hier als Maschine „`inst`“ angelegt wird, kann modellspezifische Definitionen in einem eigenen Kontext platzieren.

Um ein Metamodell in der Spezifikation eines Sicherheitsmodells zu nutzen, soll der Spezifizierende nicht mehr als einen Kontext darin einbinden müssen. Das konkrete Sicherheitsmodell kann dann als Zustandsmaschine spezifiziert werden, welche auch über einen eigenen, den Metamodell-Kontext erweiternden Kontext verfügen kann.

3.2.3 Best Practices

Bei der Überführung eines Sicherheitsmodells in eine formale Spezifikation sind viele Entscheidungen zu treffen, und die Zahl der Möglichkeiten entsprechend groß. Entsprechend viele Quellen für Komplexität, Fehler, Unklarheiten und Inkonsistenzen bestehen – Probleme, die sich beim derzeitigen Stand der Kunst nicht systematisch, sondern nur anhand von Erfahrungswerten ausräumen lassen. Der verbleibende Teil des Unterabschnittes enthält eine Auflistung bewährter Vorgehensweisen, zum einen allgemein für mathematische Spezifikationen, zum zweiten für die abstrakte Spezifikation der Modelle als Zustandsmaschine, zum dritten für die Spezifikation in Event-B. Die Angaben entstammen einerseits den Empfehlungen aus [Bos95], andererseits den Erfahrungen des Autors.

Mathematische Spezifikation.

(B1) *Endlichkeit von Mengen.* Werden in Konstanten, Variablen oder Ausdrücken Mengen genutzt, sollte stets geklärt werden, ob diese unendlich, oder doch als endlich zu deklarieren sind. Viele Spezifikationssprachen unterstützen entsprechende Syntax, im Falle von Event-B ist dies etwa das Prädikat `finite()`. Dabei ist zu diskutieren, wann die Endlichkeits-Einschränkung einen Mehrwert bringt und wann nicht. Überall dort, wo mit der endlichen Menge operiert wird, ist nachzuweisen, dass sie endlich bleibt – dies bedeutet einen erheblichen Mehraufwand. Zudem sind Trägermengen (und somit auch deren Teilmengen) häufig ohnehin endlich.

(B2) *Quantifizierung über Teilmengen.* Erfolgt eine Quantifizierung über nur einige Elemente einer Menge, sollte stets geklärt sein, ob die ausgeschlossenen Elemente nicht ebenfalls (gesondert) betrachtet werden müssen.

(B3) *Definition von Relationen und Abbildungen.* Bei der Deklaration von Relationen und Abbildungen ist es wichtig, einerseits möglichst präzise, aber andererseits auch möglichst verständlich zu formulieren. Der Präzision halber müssen Definitions- und Wertebereiche letztlich so weit wie möglich eingeschränkt werden. Für Abbildungen ist in diesem Zusammenhang auch zu prüfen, ob es sich um eine partielle oder totale Abbildung handelt. Damit jedoch die Typen der Relationen und Abbildungen klar erkennbar sind, sollten sie stets zuerst anhand von Trägermengen deklariert und dann erst später weiter eingeschränkt werden.

Beispiel 3.1. Der Zustand in der Spezifikation des Beispielmodells (vgl. Unterabschnitt 4.1.2) spezifiziert etwa die Abbildung von Sitzungen auf Nutzer `user` wie folgt:

$$\begin{aligned} &\texttt{user} \in \texttt{SESSION} \pfun \texttt{USER} \land \\ &\quad \texttt{dom}(\texttt{user}) \subseteq \texttt{S} \land \\ &\quad \texttt{ran}(\texttt{user}) \subseteq \texttt{U} \end{aligned}$$

Dabei sind `SESSION` und `USER` Trägermengen und es gilt $\texttt{S} \subseteq \texttt{SESSION}$ sowie $\texttt{U} \subseteq \texttt{USER}$. △

(B4) *Partielle Abbildungen.* Wird eine Abbildung nur partiell definiert, ist sicherzustellen, dass sich Anwendungen nicht auf Obermengen des definierten Bereichs beziehen; Gleiches gilt für die Definition auf Teilmengen einer Obermenge. Wird eine Abbildung hingegen total definiert, ist sicherzustellen, dass sie tatsächlich an allen Stellen definiert ist.

(B5) *Eigenschaften von Abbildungen und Relationen.* Zu jeder deklarierten Abbildung und Relation sollte geprüft werden, ob nicht zusätzliche Eigenschaften als Axiome definiert werden müssen. Bei Relationen betrifft dies insbesondere (Irr-)Reflexivität, (Anti-)Symmetrie, Transitivität, (Links-, Rechts-)Vollständigkeit und (Links-, Rechts-)Eindeutigkeit; bei Funktionen Injektivität, Surjektivität und Bijektivität (vgl. beispielsweise [MM06; Ste01]).

(B6) *Abbildungen auf Potenzmengen.* Wird der Wertebereich einer Abbildung als Potenzmenge definiert, so ist zu klären, ob die Definition als Relation nicht besser verständlich und nutzbarer wäre – und umgekehrt.

Beispiel 3.2. Das RBAC-Metamodell (vgl. Unterabschnitt 2.4.1) enthält die dynamische Komponente $UA \subseteq U \times R$, die Nutzer und die ihnen zugeordneten Rollen ins Verhältnis miteinander setzt. Eine alternative, gleich mächtige Definition der Komponente wäre die der Abbildung $roles : U \to \mathcal{P}(R)$ (nicht zu verwechseln mit der tatsächlich genutzten $roles : S \to \mathcal{P}(R)$).

Die Unterschiede zwischen beiden Varianten sollen an folgenden Punkten exemplarisch erläutert werden.

1. Hat ein Nutzer u eine Rolle r inne, so gilt $(u, r) \in UA$ bzw. $r \in roles(u)$.
2. Alle Rollen eines Nutzers u sind $UA[u] = \{r \mid (u, r) \in UA\}$ bzw. $roles(u)$.
3. Soll ein Nutzer $u \notin \mathrm{dom}(UA) \wedge u \notin \mathrm{dom}(roles)$ zur Komponente hinzugefügt werden, ohne ihm Rollen zuzuordnen, ist für UA keine Aktion notwendig, für eine totale Abbildung $roles$ hingegen ist zu definieren, dass $roles := roles \oplus \{(u, \varnothing)\}$.
4. Muss dem Nutzer u eine Rolle r verliehen werden, so lautet die Aussage $UA := UA \cup \{(u, r)\}$ bzw. $roles := roles \oplus \{(u, roles(u) \cup \{r\})\}$.
5. Sollen alle Sitzungen mit den *potentiell aktivierbaren* Rollen in Relation gebracht werden, ist die Abbildung von Sitzungen auf die zugeordneten Nutzer $user : S \to U$ zu Rate zu ziehen. Wird UA genutzt, kann die gewünschte Relation schlicht durch Komposition erzeugt werden, denn $user \fatsemi UA = \{(s, r) \mid \exists u \cdot user(s) = u \wedge (u, r) \in UA\}$.

Während die Aussage $roles(u)$ intuitiv verständlich ist, lässt sich die Relation UA in einigen Situationen besser nutzen. Semantisch können, je nach konkretem Fall, beide Varianten naheliegend sein. △

Abstrakte ZSM-Spezifikation.

(B7) *Auswahl der Vorbedingungen von Operationen.* Sowohl interne, etwa als Konstanten c definierte Funktionen, als auch die Operationen o der Zustandsmaschine können mit Vorbedingungen pre versehen werden. Diese haben zur Folge, dass der Definitionsbereich der Funktion oder Operation eingeschränkt wird.

Mit Vorbedingungen kann unterschiedlich verfahren werden: einerseits können sie so stark wie möglich ausgewählt sein, sodass nur genau die Parameterbelegungen akzeptiert werden, für welche die jeweilige Funktion bzw. Operation ihren vorgesehenen Effekt bewirken kann. Andererseits lassen sich die Vorbedingungen sehr schwach wählen, sodass bei Parameterbelegungen, für welche die Funktion bzw. Operation nicht in Effekt treten kann, die Anwendung dennoch zulässig (aber eben effektlos) ist.

Bei der Entscheidung zwischen der Verwendung starker und schwacher Vorbedingungen sind folgende Aspekte zu berücksichtigen: starke Vorbedingungen dokumentieren zugleich die Menge derjenigen Parameterbelegungen, für die eine Anwendung der Funktion bzw. Operation effektvoll ist. Dies ist auch im transitiven Sinne gültig: überall dort, wo eine mit strengen Vorbedingungen versehene Funktion bzw. Operation eingesetzt wird, sind diese Bedingungen explizit zu erfüllen – wodurch es notwendig wird, dort entweder auch strenge Vorbedingungen einzusetzen, oder aber Fälle effektlosen Verhaltens explizit auszuschließen. Das befördert das Verständnis für das Verhalten der Spezifikation und hilft, Missverständnisse (wie unbeabsichtigt effektloses Verhalten) zu vermeiden. Zu diesem Effekt starker Vorbedingungen trägt bei, dass das Führen interaktiver Beweise oder die Animierung der Spezifikation Verletzungen der Vorbedingungen aufgedeckt.

Schwache Vorbedingungen hingegen bedeuten, dass das Verhalten im Fehlerfall nicht einfach ausgeschlossen wird, sondern explizit aus der Spezifikation hervorgeht. Dies kann einerseits bedeuten, dass eine Fehlermeldung erzeugt wird, dass das System in einen Fehlerzustand übergeht, oder dass der Fehler toleriert wird. Schwache Vorbedingungen erhöhen also die Robustheit der Operationen und senken das benötigte Vorwissen beim Aufrufer: dies ist gerade für Operationen an den Schnittstellen des Systems sinnvoll.

In einigen Fällen können zum Ausführen einer Operation notwendige Bedingungen entweder durch die Operation selbst hergestellt, oder durch ihre Vorbedingung vorausgesetzt werden. Dabei senkt zwar die implizite Herstellung solcher Bedingungen durch die Operation die Komplexität beim Aufrufer, vermindert jedoch zugleich die Transparenz des Verhaltens. Könnte das Herstellen solcher Bedingungen Sicherheitseigenschaften gefährden – beispielsweise dann, wenn Nutzern zusätzliche Rechte einzuräumen sind – ist die Bedingung auf jeden Fall als Vorbedingung zu formulieren.

Beispiel 3.3. Das *activateRole*-Kommando des Autorisierungsschemas kann zu einer Sitzung nur solche Rollen aktivieren, welche dem zugehörigen Nutzer zugewiesen sind. Diese Zuordnung muss offensichtlich Bedingung zur Ausführung des Kommandos sein, um keine Sicherheitseigenschaften zu verletzen. Das *revokeRole*-Kommando hingegen entzieht einem Nutzer eine Rolle; hierzu ist es erforderlich, dass keine Sitzung des Nutzers diese Rolle aktiviert hat. Das implizite Deaktivieren dieser Rolle in allen Sitzungen des Nutzers könnte dabei durchaus Teil der Semantik des Kommandos sein, ohne Sicherheitseigenschaften zu gefährden. △

Bei Betrachtung des Entwicklungsprozesses eines Systems ist zu berücksichtigen, dass Operationen mit starken Vorbedingungen zu solchen mit schwächeren Vorbedingungen verfeinert werden können (vgl. Unterabschnitt 2.6.1). Es ist also empfehlenswert, Operationen an der Systemgrenze zu Beginn und interne Funktionen stets mit strengen Vorbedingungen auszustatten, und die Operationen dann schrittweise zu möglichst allgemeinen Vorbedingungen zu verfeinern.

(B8) *Abhängigkeiten im Zustandsraum.* Teilweise sind die Definitionen dynamischer Komponenten von anderen dynamischen Komponenten abhängig, etwa bezüglich ihrer Definitions- oder Wertebereiche. Dies hat zur Folge, dass Änderungen an einer Komponente auch Änderungen an anderen Komponenten nach sich ziehen, was im Rest der Arbeit *Seiteneffekt* genannt wird. Bei Zustandsänderungen sind solche Abhängigkeiten zu berücksichtigen.

Beispiel 3.4. Im RBAC-Metamodell (vgl. Unterabschnitt 2.4.1) ist die Relation UA definiert als Teilmenge von $U \times R$; die Nutzermenge U ist dabei ebenso wie UA Teil des Zustandes. Wird nun mittels der *deleteUsers*-Primitivoperation eine Nutzermenge U' gelöscht, sind die entsprechenden Tupel mittels $UA := U' \lhd UA$ aus UA auszuschließen. △

(B9) *Komponentenübergreifende Eigenschaften.* Die Semantik einiger Komponenten, wie etwa die Rollenhierarchie oder der Rollenausschluss im Beispielmodell (vgl. Abschnitt 2.4) beeinflusst das Verhalten des Modells bezüglich anderer Komponenten. Die korrekte Spezifikation solcher Einflüsse sollte für alle statischen und dynamischen Modellkomponenten überprüft werden. Dies ist gerade dann wichtig, wenn solche Eigenschaften bei jedem Zugriff auf Komponenten berücksichtigt werden müssen.

Beispiel 3.5. Im Beispielmodell (vgl. Unterabschnitt 2.4.2) verfügt ein Nutzer u über die Rollen $UA[\{u\}]$. Die Rollenhierarchie jedoch bewirkt, dass er auch Berechtigungen derjenigen Rollen besitzt, die von seinen Rollen dominiert werden. Die Primitivprädikate implementieren diese Semantik und sollten daher, sofern sinnvoll, eingesetzt werden. △

Spezifikation in Event-B.

(B10) *Reihenfolge der Axiome.* Bei den in einem Kontext unter `AXIOMS` spezifizierten Axiomen und Theoremen sowie den in einem Ereignis unter `where` spezifizierten Guards[1] ist die Reihenfolge entscheidend. Der Beweis eines Axioms oder Guards erfolgt unter Annahme aller vorhergehenden Axiome bzw. Guards, sowie aller Axiome bzw. Guards aus erweiterten Kontexten bzw. Ereignissen.

Beispiel 3.6. Gehören zu einem Kontext etwa die Axiome $\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_n$, so gelten zum Beweis von Axiom $\mathtt{a}_i$ lediglich die Axiome $\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_{i-1}$ als bewiesen. △

(B11) *Theoreme zur Erleichterung von Beweisen.* Theoreme können zur Erleichterung von Beweisen genutzt werden. Muss als Teil vieler Beweise ein und derselbe Sachverhalt gezeigt werden, kann dieser schlicht als Theorem eingeführt und einmalig bewiesen werden.

[1] Guards sind die Vorbedingungen zum Ereignis.

3.3 Abstrakte Spezifikation eines Metamodells

Ein Metamodell (vgl. Unterabschnitt 2.3.3) ist eine Klasse von Sicherheitsmodellen mit gemeinsamen Wertebereichen T_i, statischen Komponenten S_i, dynamischen Komponenten D_i, gemeinsamem Ausgaberaum Γ sowie darauf definierten gemeinsamen Primitivoperationen o_i und -prädikaten p_i. Der vorliegende Abschnitt argumentiert zunächst auf abstrakter Ebene, wie diese Elemente in eine Spezifikation überführt werden können. Danach erläutert Abschnitt 3.4 spezielle Aspekte bei der Spezifikation in Event-B.

3.3.1 Wertebereiche

Die Wertebereiche T_i des Metamodells sind paarweise disjunkte Mengen, welche alle atomaren Objekte enthalten, mit denen das Modell operiert. Sie bestimmen zugleich die Typisierung der Objekte. Alle weiteren Datenstrukturen und Operationen des Modells sind auf diesen Wertebereichen definiert (vgl. Unterabschnitt 2.3.2). Sie sind anwendungsspezifisch und können etwa Prozesse, Benutzer, Dateien, Rollen etc. modellieren.

In Spezifikationen heißen die paarweise disjunkten Mengen, welche atomare Objekte enthalten, Trägermengen. Da der Übergang von Sicherheitsmodell auf Spezifikation lediglich ein Wechsel der Darstellungsform ist (vgl. Abschnitt 3.1), sind die Trägermengen der Spezifikation gleich den Wertebereichen des Modells.

Im Allgemeinen sind die einzelnen Elemente der Trägermengen nicht im Metamodell benannt, da sie abhängig vom konkreten Anwendungsszenario, und daher erst im konkreten Sicherheitsmodell bekannt sind.

3.3.2 Statische Komponenten

Die statischen Komponenten S_i des Metamodells sind beliebige mathematische Objekte (also insbesondere atomare Objekte, Tupel, Mengen, Relationen oder Funktionen), welche anwendungsspezifische, unveränderliche Aspekte zum Modell hinzufügen (vgl. Unterabschnitt 2.3.1). Diese Komponenten müssen von allen übrigen Teilen des Modells aus zugreifbar sein; sie bringen üblicherweise bestimmte Eigenschaften mit sich (etwa Transitivität oder Symmetrie bei Relationen); ebenso können verschiedene Komponenten durch prädikatenlogische Aussagen miteinander verknüpft sein (etwa im Falle des Verbotes, einem Nutzer sich gegenseitig ausschließende Rollen zuzuordnen). Die konkreten Werte der statischen Komponenten sind zumeist erst in einer Instanz des Metamodells bekannt (vgl. Unterabschnitt 3.5.2).

Spezifikationsmethode. Komponenten könnten prinzipiell als Konstanten oder Variablen in die Spezifikation eingebracht werden. Wie in Unterabschnitt 3.2.2 argumentiert, ist es zu Zwecken der Wiederverwendung sinnvoll, das Metamodell auf den Kontext der Spezifikation zu begrenzen. Da der Kontext die Konstanten, nicht jedoch die Variablen enthält, müssen statische Komponenten also in Konstante des Kontextes überführt werden. Eine weitere Kapselung, etwa in Tupel einer Menge, bringt keine Vorteile im Sinne der Anforderungen (da unverständlicher, unpräzise,

komplexer, nicht besser wiederverwendbar). Es sollte also jede Komponente S_i des Modells in genau eine der Konstanten c der Spezifikation überführt werden.

Eigenschaften. Die Eigenschaften der einzelnen statischen Modellkomponenten müssen vollständig in die Spezifikation eingehen, um später die Konsistenz des Modells mit diesen Eigenschaften zeigen zu können. Problematisch ist aber, dass diese in Modellen häufig (wenn überhaupt) nicht vollständig notiert sind und eher implizit angenommen werden. So könnte ein Modellkonstrukteur etwa für eine Ausschlussrelation implizit annehmen, dass diese symmetrisch und irreflexiv ist.

Im Rahmen des Specification Engineering sind solche Eigenschaften möglichst vollständig zu erfassen. Eine empfehlenswerte Vorgehensweise ist, alle naheliegenden Eigenschaften zu überprüfen, vgl. (B1) und (B5). Außerdem kann beispielhaft anhand der Semantik der Komponenten analysiert werden, ob bestimmte Belegungen auszuschließen sind. Zusätzlich sind Abhängigkeiten zwischen Komponenten zu analysieren.

Sind alle solche Eigenschaften bekannt, können sie als Axiome $\mathsf{A}(\mathsf{c})$ in die Spezifikation eingebracht werden. Es empfiehlt sich, bei komplexen Aussagen alternative Formen als Theoreme $\mathsf{B}(\mathsf{c})$ anzugeben (und die Zusammenhänge zu dokumentieren): dies erleichtert Beweise, verbessert die Verständlichkeit, und erhöht dabei (bei geeigneter Abgrenzung von den Axiomen) nicht die Komplexität der Spezifikation im Sinne von Anforderung (S7).

Im Übrigen muss die interne Konsistenz anhand Beweispflicht (P3) gezeigt werden; die Axiome sind im Sinne von Präzision und Verständlichkeit zu formulieren (vgl. Strategien in Abschnitt 3.2), was auch die Komplexität senkt.

3.3.3 Dynamische Komponenten

Die dynamischen Komponenten D_i des Metamodells sind ebenso wie statische Komponenten beliebige mathematische Objekte, anhand derer sich der Zustandsraum Q des Modells aufspannt. Sie müssen von Primitivoperationen und -prädikaten zugreifbar sein: das Autorisierungsschema wird später ausschließlich über solche Primitive auf den Zustand zugreifen (vgl. Unterabschnitt 2.3.2).

Spezifikationsmethode. Wie Unterabschnitt 3.2.2 belegt, ist das Metamodell ausschließlich im Kontext zu spezifizieren, sodass sich zwei Möglichkeiten der Definition von dynamischen Komponenten sowie Primitivoperationen und -prädikaten ergeben:

1. Der gesamte Zustandsraum Q wird als einzelne Konstante in die Spezifikation eingebracht; diese Konstante hat dann den selben Typ wie Q (also $D_1 \times D_2 \times \cdots \times D_n$). Sei diese Konstante STATE, so werden Primitivoperationen und -prädikate dann auf dieser Konstante definiert. Ist beispielsweise prim_i die Spezifikation der Primitivoperation o_i, so wäre sie vom Typ

$$\mathsf{prim}_i : \mathsf{STATE} \times \overbrace{\cdots}^{\text{weitere Argumente}} \to \mathsf{STATE}.$$

Ein konkretes Sicherheitsmodell nutzt dann lediglich eine einzige Zustandsvariable, etwa state, welche stets Element des Zustandsraumes ist, also $\mathsf{state} \in \mathsf{STATE}$.

2. Zu jeder dynamischen Komponente D_i wird die Menge aller möglichen Werte, welche diese Komponente annehmen kann, als Konstante in die Spezifikation eingebracht, also etwa DYN_i. Primitivoperationen und -prädikate werden dann jeweils auf diesen Wertebereichen definiert. Ist etwa wieder prim_j die Konstante, welche o_j spezifiziert, so wäre sie vom Typ

$$\begin{aligned} \mathsf{prim}_j : \mathsf{DYN}_{i(1)} &\times \mathsf{DYN}_{i(2)} \times \cdots \times \mathsf{DYN}_{i(k)} \times \overbrace{\cdots}^{\text{weitere Argumente}} \\ \rightarrow \mathsf{DYN}_{i(k+1)} &\times \mathsf{DYN}_{i(k+2)} \times \cdots \times \mathsf{DYN}_{i(k+\ell)}. \end{aligned}$$

Ein konkretes Sicherheitsmodell definiert dann pro dynamischer Komponente D_i eine Zustandsvariable dyn_i als Element des zugehörigen, als Konstante definierten Wertebereichs: $\forall\, i \cdot \mathsf{dyn}_i \in \mathsf{DYN}_i$.

Alternativ könnte das Metamodell lediglich informal verlangen, dass die Sicherheitsmodell-Spezifikation bestimmte Zustandsvariablen mit bestimmten Typen verwendet. Dies scheidet aus, ist der Ansatz doch höchst fehleranfällig, da nicht automatisch zu prüfen, zumal Primitivoperationen und -prädikate so nur schwer zu definieren sind und deren korrekte Anwendung nicht sicherzustellen ist. Zuletzt müssten Beweise für jedes Sicherheitsmodell neu erbracht werden. Der Strategie, Komplexität möglichst ins Metamodell auszulagern, widerspricht dies völlig (vgl. Abschnitt 3.2).

Im Vergleich fällt auf, dass Variante 1 näher am Sicherheitsmodell ist, welches ja auch auf einem einzigen Zustandsraum Q operiert und seine Operationen ebenso definiert. Variante 2 hingegen stellt bereits einen Schritt hin zu einer anwendungsnäheren Ebene dar, in welcher Zustandsvariablen eventuell über komplexe, anwendungs- oder programmiersprachenspezifische Datenstrukturen definiert sind. So wäre es auch denkbar, die Primitivoperationen als Operationen auf jeweils genau einer Datenstruktur zu definieren (etwa das Einfügen eines Elementes in einer Menge, das Manipulieren einer Abbildung, etc.). Eine solche Semantik bietet sich dann als Verfeinerung einer über Variante 1 erstellte Spezifikation an.

Variante 1 ist insbesondere in der Spezifikation des konkreten Sicherheitsmodells kompakter als Variante 2: sie bedarf nur einer Variable, und pro Primitivoperation nur eines zusätzlichen Parameters. Dies entspricht der oben notierten Strategie, Komplexität möglichst in die Metamodell-Spezifikation zu verlagern (vgl. Abschnitt 3.2).

Gegen beide Ansätze spricht ihre schlechte Animierbarkeit: ein Animator, welcher keine symbolische Auswertung beherrscht, wird immer versuchen, den vollständigen Wertebereich auszuwerten, was aufgrund der potentiell unendlichen Größe zum Scheitern verurteilt ist. Mit dem passenden Werkzeug – einem Animator nämlich, welcher die symbolische Auswertung entsprechend beherrscht, lässt sich das Problem jedoch lösen.

Letztlich entspricht also Variante 1 stärker den Anforderungen, sodass dynamische Komponenten letztlich als Zustandsraum zu definieren und erst in Verfeinerungen auf anwendungsnähere Datenstrukturen zu bringen sind.

Konsistenzeigenschaften. Oft gelten für den Zustandsraum zusätzliche Konsistenzbedingungen die zulässige Werte einschränken, insbesondere um Verbindungen zwischen verschiedenen dynamischen Komponenten herzustellen (vgl. (B8)), jedoch auch für andere Eigenschaften (vgl. etwa (B1)). Teilweise können Verletzungen von Konsistenzeigenschaften in Zwischenzuständen (etwa nach der Ausführung von Primitivoperationen) erlaubt sein.

Konsistenzeigenschaften, welche auch für Zwischenzustände gelten sollen, können direkt für den Zustandsraum (d.h. nach obigen Erläuterungen für die entsprechende Konstante STATE der Spezifikation) geltend gemacht werden. Der Raum konsistenter Zustände könnte dann vom normalen Zustandsraum (welcher Zwischenzustände enthält) entweder durch ein Konsistenzprädikat $\chi : \mathsf{STATE} \to \mathbb{B}$, oder durch Definition eines konsistenten Unterraums des Zustandsraums $\mathsf{CONSISTENT\_STATE} \subset \mathsf{STATE}$ abgetrennt werden.

An dieser Stelle wird die Nutzung des konsistenten Unterraums gegenüber der Nutzung von Konsistenzprädikaten empfohlen. Denn erstens ist das Handling aus Sicht des konkreten Sicherheitsmodells in diesem Fall einfacher, und zweitens können auf dem Unterraum auch Funktionen und Operationen definiert werden. Die Spezifikation des konkreten Sicherheitsmodells muss dann seine Zustandsvariable state so definieren, dass sie in CONSISTENT_STATE liegt. Einzelne Primitvoperationen müssen zumeist auf STATE definiert werden, sodass für das konkrete Sicherheitsmodell zu jedem Kommando des Autorisierungsschemas zu beweisen ist, dass das Ergebnis der Konkatenation der Primitivoperationen des Anweisungsteils stets in CONSISTENT_STATE liegt.

Zugriff auf Zustandskomponenten. Es empfiehlt sich, konstante Funktionen zu definieren, welche die einzelnen Komponenten aus dem Zustandsraum herausprojizieren. Sei der Zustandsraum beispielsweise wieder $\mathsf{STATE} = \mathsf{DYN}_1 \times \mathsf{DYN}_2 \times \cdots \times \mathsf{DYN}_n$, so wären dies für jedes $1 \leq i \leq n$ Funktionen $\mathsf{dyn}_i : \mathsf{STATE} \to \mathsf{DYN}_i$. Außerdem können zur Erleichterung von Beweisen solche Aussagen als Theoreme $\mathsf{B(c)}$ eingeführt werden, welche aus Konsistenzeigenschaften über den Zustandsraum folgen.

Zusammengefasst erscheint die Spezifikation dynamischer Komponenten im Metamodell äußerst komplex. Jedoch ist zu bedenken, dass dies die Komplexität der Spezifikationen konkreter Sicherheitsmodelle mindert (vgl. Abschnitt 3.2).

3.3.4 Primitivoperationen

Bei Primitivoperationen o_i handelt es sich (vgl. Unterabschnitt 2.3.2) um Abbildungen der Form

$$o_i : Q \times T_1^{o_i} \times T_2^{o_i} \times \cdots \times T_{\mathrm{n}(o_i)}^{o_i} \to Q.$$

welche im Autorisierungsschema dazu eingesetzt werden, die Änderungen des Zustandes durch anwendungsspezifische Operationen zu modellieren. In einer Spezifikation müssen sie also durch die Spezifikation der anwendungsspezifischen Kommandos

zugreifbar sein. Ebenso wie dynamische Komponenten sind sie im Kontext der Spezifikation, und somit als Konstanten, zu deklarieren.

Wie oben in Unterabschnitt 3.3.3 beschrieben, wird der gesamte Zustandsraum als einzelne Konstante STATE modelliert. Entsprechend sollte zu jeder Primitivoperation o_i eine Konstante prim_i in der Spezifikation erstellt werden mit Typ

$$\mathsf{prim}_i : \mathsf{STATE} \times \mathsf{T}_1^{o_i} \times \mathsf{T}_2^{o_i} \times \cdots \times \mathsf{T}_{\mathrm{n}(o_i)}^{o_i} \to \mathsf{STATE},$$

wobei die $\mathsf{T}_j^{o_i}$ den $T_j^{o_i}$ entsprechen. Der Verständlichkeit (S5) halber sollte jede Primitivoperation denselben Namen aufweisen, wie dies im Sicherheitsmodell der Fall ist.

Verallgemeinerte Signatur. Ein mögliches Problem ergibt sich aus der Beschaffenheit der meisten Spezifikationssprachen: sie folgen rein funktionalen Programmierparadigmen, die frei von impliziten Seiteneffekten sind. Muss ein anwendungsspezifisches Kommando nun eine a priori unbekannte Anzahl von Argumenten verarbeiten (etwa eine Menge von Nutzern, Rollen etc.), so kann sie dies daher nicht ohne weiteres durch wiederholtes Aufrufen einer Primitivoperation erzielen, da Schleifen in derartigen funktionalen Sprachen zumeist nicht unterstützt werden. Primitivoperationen sind deshalb so allgemein wie möglich zu definieren: statt einem Argument, das lediglich ein atomares Objekt verlangt, sollten sie eine Menge von Objekten akzeptieren. Abhängige Argumente (etwa Sitzungen, die zu Nutzern erstellt werden) können über Tupel realisiert werden.

Beispiel 3.7. Sei in einer Spezifikation mit Zustandsraum STATE die Menge der Nutzer USER und die Menge der Sitzungen SESSION. Wird die Primitivoperation *mapUserSessions* (vgl. Seite 16) als

$$\mathsf{mapUserSessions} : \mathsf{STATE} \times \mathsf{USER} \times \mathsf{SESSION} \to \mathsf{STATE}$$

spezifiziert, so wären Sicherheitsmodelle nicht in der Lage, innnerhalb eines Kommandos eine a priori unbekannte Zahl von Sitzungen und Nutzern zuzuordnen. Erst eine Definition wie

$$\mathsf{mapUserSessions} : \mathsf{STATE} \times \mathcal{P}(\mathsf{USER} \times \mathsf{SESSION}) \to \mathsf{STATE}$$

ermöglicht dies. △

Best Practices. Ansonsten sind bei der Spezifikation der Primitivoperationen nahezu alle Empfehlungen aus Unterabschnitt 3.2.3 relevant. An dieser Stelle sei insbesondere auf (B7) verwiesen, wonach zu jeder Primitivoperation möglichst strenge Vorbedingungen zu spezifizieren sind, sowie auf (B8), wonach mögliche Seiteneffekte, die sich aus Abhängigkeiten zwischen den dynamischen Modellkomponenten ergeben, zu berücksichtigen sind.

Eigenschaften. Zu einigen Primitivoperationen lassen sich Eigenschaften identifizieren, welche dann in der Spezifikation als Theoreme $\mathsf{B}(\mathsf{c})$ notiert werden müssen.

Das trägt zur Korrektheit bei, da Verletzungen dieser Eigenschaften dazu führen, dass die entsprechenden Theoreme unbeweisbar werden. Folgende Aspekte sollten geprüft werden, um Eigenschaften zu entdecken.

- *Idempotenz.* Ändern mehrere sukzessive Anwendungen einer Operation prim_i den Zustand gegenüber ihrer einmaligen Anwendung nicht mehr, sollte dies via $\mathsf{prim}_i(\mathsf{state}) = \mathsf{prim}_i(\mathsf{prim}_i(\mathsf{state}))$ vermerkt werden[2].
- *Reversibilität.* Kann der Effekt einer Operation prim_i mittels einer Operation prim_j umgekehrt werden, ist dies via $\mathsf{prim}_i(\mathsf{prim}_j(\mathsf{state})) = \mathsf{state}$ zu notieren.
- *Kommutativität.* Möglicherweise kann die Reihenfolge der Ausführung zweier Operationen prim_i und prim_j vertauscht werden, ohne dass sich der Effekt ändert. Dies ist via $\mathsf{prim}_i(\mathsf{prim}_j(\mathsf{state})) = \mathsf{prim}_j(\mathsf{prim}_i(\mathsf{state}))$ zu notieren.

Dabei ist zu beachten, dass das Notieren von Eigenschaften den Aufwand zur Spezifikation sowie die Komplexität der Spezifikation erhöht. Der Nutzen für die Korrektheit ist dabei bei der Kommutativität oftmals eher gering, da die Effekte des Vertauschens einer Operation leicht zu argumentieren sind – das ist gerade dann der Fall, wenn es um das Vertauschen ein und derselben Operationen mit lediglich unterschiedlichen Parametern geht. Umso größer ist der Nutzen etwa bei der Reversibilität; hier wird sichergestellt, dass zwei zumeist unterschiedliche Operationen gemeinsam wieder in exakt denselben Ausgangszustand führen. Unbeachtete Seiteneffekte würden sofort zur Unbeweisbarkeit des Theorems führen. Die Idempotenz ist eine spezielle Eigenschaft, welche unter Einsatz starker Vorbedingungen (vgl. Vorgehensweise (B7)) nur auf wenige Operationen zutrifft und dann auch zur Dokumentation besonders nützlich ist.

3.3.5 Primitivprädikate

Primitivprädikate p_i sind ähnlich zu Primitivoperationen; es handelt sich um Abbildungen der Form

$$p_i : Q \times T_1^{p_i} \times T_2^{p_i} \times \cdots \times T_{\mathrm{n}(p_i)}^{p_i} \to \mathbb{B}.$$

Im Autorisierungsschema eines anwendungsspezifischen Kommandos werden sie eingesetzt, um Bedingungen zu formulieren (vgl. Unterabschnitt 2.3.2). Der Wert eines Prädikates kann dabei von den Werten der statischen und dynamischen Komponenten des Modells abhängen. In einer Spezifikation müssen die Primitivprädikate durch die Spezifikation der anwendungsspezifischen Kommandos zugreifbar sein.

Im Übrigen gelten die Argumentationen für Primitivoperationen: ein Primitivprädikat p_i sollte also als Konstante pred_i in der Spezifikation deklariert werden mit Typ

$$\mathsf{pred}_i : \mathsf{STATE} \times \mathsf{T}_1^{p_i} \times \mathsf{T}_2^{p_i} \times \cdots \times \mathsf{T}_{\mathrm{n}(p_i)}^{p_i} \to \mathsf{BOOL},$$

[2] Auf die Darstellung weiterer Argumente der prim_i wurde zur Übersichtlichkeit verzichtet.

wobei BOOL die Trägermenge der Spezifikationssprache für Wahrheitswerte ist und die Trägermengen der Argumente $\mathsf{T}_j^{p_i}$ wieder den Wertebereichen $T_j^{p_i}$ entsprechen. Dabei sollten die Argumente eines Primitivprädikates wieder so allgemein wie möglich definiert werden.

3.4 Spezifikation eines Metamodells in Event-B

Abschnitt 3.3 stellte eine Vorgehensweise zur Überführung eines Metamodells in eine Spezifikation vor und abstrahierte dabei von konkreten Spezifikationssprachen. Der vorliegende Abschnitt erarbeitet hieraus Anweisungen zum Erstellen einer Metamodell-Spezifikation in Event-B. Er macht dabei nacheinander Angaben zur Modularisierung einer solchen Spezifikation (Unterabschnitt 3.4.1) sowie zur Spezifikation statischer Aspekte, dynamischer Aspekte und der Primitivoperationen / -prädikate (Unterabschnitte 3.4.2 bis 3.4.6).

3.4.1 Modularisierung

Wie Unterabschnitt 3.2.2 beschrieb, kann eine Metamodell-Spezifikationen in mehrere, durch Erweiterung verknüpfte Kontexte aufgeteilt werden. Diese Kontexte sollten einerseits gemäß der Struktur des Metamodells gegliedert sein, andererseits nach der Komplexität des Spezifikationsvorgangs bzw. der entstandenen Spezifikation. Aufgrund der in dieser Arbeit vorgestellten Methode können bestimmte Größenverhältnisse erwartet werden, sodass folgende Strukturierung vorgeschlagen wird (vgl. Abbildung 3.2).

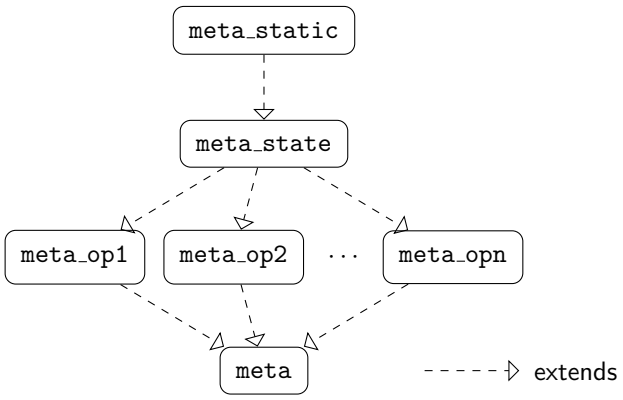

Abbildung 3.2: Kontexthierarchie in der Event-B-Spezifikation eines Metamodells. Der Name des Metamodells wird hier abstrakt mit `meta` bezeichnet.

Generell sollten alle Kontexte eines Metamodells einem gemeinsamen Namensschema folgen, erkennbar an einem Namenspräfix (etwa `rbac`, `tam`, `hru`, etc.). Nachfolgend wird dieser Präfix abstrakt `meta` genannt.

Sämtliche Teile der Spezifikation sind abhängig von statischen Deklarationen, nämlich den Trägermengen sowie den statischen Komponenten. Diese sind oft wenig komplex und sollen daher in einen Kontext `meta_static` zusammengefasst werden.

Die Spezifikation dynamischer Komponenten, also des Zustandsraumes, ist komplexer, jedoch nur schwer aufzugliedern. Sie ist in einem Kontext `meta_state` zu verfassen, welcher `meta_static` erweitert.

Zu einem Metamodell kann es eine große Zahl an Primitivoperationen geben, welche jeweils mit mehreren Axiomen und Theoremen zu notieren sind. Es empfiehlt sich daher eine Gruppierung der Operationen in verschiedene Kontexte, wie etwa `meta_userhandling`, `meta_righthandling` etc. Jeder dieser Kontexte sollte `meta_state` erweitern, um Zugriff auf statische und dynamische Komponenten des Metamodells zu haben. Abhängigkeiten zwischen den Primitivoperationen sind nicht zu erwarten. Mit Primitivprädikaten ist ebenso zu verfahren.

Alle Teilkontexte für Primitivoperationen und -prädikate sollte ein gemeinsamer Kontext erweitern, der ausschließlich den Bezeichner für das Metamodell trägt, also `meta`. Von ihm aus sind dann sämtliche Teile der Spezifikation eines Metamodells zugreifbar, sodass er von Spezifikationen konkreter Sicherheitsmodelle verwendet werden kann.

3.4.2 Trägermengen

Wie oben erläutert (vgl. Unterabschnitt 3.3.1), muss das Metamodell eine Reihe von Trägermengen mit modellspezifischen Objekttypen (Rechte, Rollen, Benutzer, etc.) definieren. Diese können in Event-B in der `SETS`-Klausel eines Kontextes untergebracht werden. Seien nun die Trägermengen $\mathsf{SET}_1, \mathsf{SET}_2, \ldots, \mathsf{SET}_n$, so wären sie im Kontext `meta_static` wie folgt einzubringen.

CONTEXT meta_static

SETS

SET_1

SET_2

$\vdots$

SET_n

END

Dabei werden Trägermengen in Event-B üblicherweise in Großbuchstaben notiert. Konkrete Elemente der Trägermengen, also atomare Objekte, sind in einem Metamodell zwar üblicherweise unbekannt. Allerdings können solche Elemente als Konstanten eingeführt werden, siehe dazu Unterabschnitt 3.6.2.

3.4.3 Statische Komponenten

Gemäß den obigen Erläuterungen (vgl. Unterabschnitt 3.3.2) sind statische Komponenten als Konstanten, sowie deren Eigenschaften als Axiome in einem Kontext festzuhalten. Die zugehörigen Klauseln einer Event-B-Spezifikation sind `CONSTANTS` und `AXIOMS`. Seien die statischen Komponenten stat_i, jede vom Typ STAT_i, so muss der Kontext `meta_static` gegenüber obiger Definition wie folgt erweitert werden.

```
CONTEXT meta_static
SETS
   ⋮
CONSTANTS
   stat_1      > Einführen der Namen
   stat_2
   ⋮
   stat_n
AXIOMS
   st1_t: stat_1 ∈ STAT_1      > Typisierung
   st2_t: stat_2 ∈ STAT_2
   ⋮
   stn_t: stat_n ∈ STAT_n
END
```

Eigenschaften müssen dann jeweils als zusätzliche Axiome möglichst dicht zu den Typisierungsaxiomen definiert werden.

In Event-B trägt jedes Axiom einen Namen, das sogenannte *Label*. An dieser Stelle wird vorgeschlagen, stets Labels nach dem Muster `xxx_y` zu nutzen, wobei `xxx` einen Bezug zum spezifizierten Element herstellt (etwa der Name der statischen Komponente), jedoch nicht länger als drei Buchstaben ist[3]. Das optionale Suffix `_y` gibt einen Hinweis auf die Rolle des jeweiligen Axioms (etwa `_t` zur Typisierung, oder Ziffern zur fortlaufenden Nummerierung von Eigenschaften).

3.4.4 Dynamische Komponenten

Nach der abstrakten Beschreibung (vgl. Unterabschnitt 3.3.3) werden die dynamischen Komponenten dyn_i vom Typ DYN_i über den Zustandsraum STATE eingeführt, welcher als Konstante in der Spezifikation vermerkt wird. Nach obigen Erläuterungen sollte dies im Kontext `meta_state` erfolgen. Zusätzlich wird zu jeder Komponente eine Projektionsfunktion benötigt, welche den Wert der Komponente aus dem Zustandsraum projiziert.

Die entsprechende Spezifikation erfolgt in Event-B nach folgendem Schema.

```
CONTEXT meta_state
EXTENDS
   meta_static
CONSTANTS
   STATE
   dyn_1_q
```

[3] Die Lesbarkeit der Spezifikation leidet, wenn Labels mit stark unterschiedlicher Länge eingesetzt werden. Die hier vorgeschlagene Länge hat sich als praktikabel herausgestellt.

$\mathtt{dyn_2\_q}$

$\vdots$

$\mathtt{dyn_n\_q}$

AXIOMS

$\mathtt{st}$: $\mathtt{STATE} = \{\mathtt{dyn_1} \mapsto \mathtt{dyn_2} \mapsto \cdots \mapsto \mathtt{dyn_n} \mid \mathtt{dyn_1} \in \mathtt{DYN_1} \wedge \mathtt{dyn_2} \in \mathtt{DYN_2} \wedge \cdots \wedge \mathtt{dyn_n} \in \mathtt{DYN_n}\}$

$\mathtt{dyn_1\_t}$: $\mathtt{dyn_1\_q} \in \mathtt{STATE} \rightarrow \mathtt{DYN_1}$

$\mathtt{dyn_2\_t}$: $\mathtt{dyn_2\_q} \in \mathtt{STATE} \rightarrow \mathtt{DYN_2}$

$\mathtt{dyn_3\_t}$: $\mathtt{dyn_3\_q} \in \mathtt{STATE} \rightarrow \mathtt{DYN_3}$

$\mathtt{dyn_{n-1}\_t}$: $\mathtt{dyn_{n-1}\_q} \in \mathtt{STATE} \rightarrow \mathtt{DYN_{n-1}}$

$\mathtt{dyn_n\_t}$: $\mathtt{dyn_n\_q} \in \mathtt{STATE} \rightarrow \mathtt{DYN_n}$

$\mathtt{prj\_d}$: $\forall \mathtt{q} \cdot \mathtt{q} \in \mathtt{STATE} \Rightarrow$
$\mathtt{dyn_1\_q} = \mathtt{prj1}(\mathtt{prj1}(\mathtt{prj1}(\ldots(\mathtt{prj1}(\mathtt{q}))))) \wedge$
$\mathtt{dyn_2\_q} = \mathtt{prj2}(\mathtt{prj1}(\mathtt{prj1}(\ldots(\mathtt{prj1}(\mathtt{q}))))) \wedge$
$\mathtt{dyn_3\_q} = \mathtt{prj2}(\mathtt{prj1}(\ldots(\mathtt{prj1}(\mathtt{q})))) \wedge$
$\vdots$
$\mathtt{dyn_{n-1}\_q} = \mathtt{prj2}(\mathtt{prj1}(\mathtt{q})) \wedge$
$\mathtt{dyn_n\_q} = \mathtt{prj2}(\mathtt{q})$

END

Die Projektionsfunktionen, welche hier abstrakt $\mathtt{dyn_1\_\_q}, \mathtt{dyn_2\_\_q}, \ldots, \mathtt{dyn_n\_\_q}$ genannt wurden, benutzen dabei die Event-B-Operatoren `prj1` und `prj2`, welche in der Lage sind, aus einem Zweitupel das erste bzw. zweite Element herauszuprojizieren. Komplexe Tupel der Form $\mathtt{x_1} \mapsto \mathtt{x_2} \mapsto \cdots \mapsto \mathtt{x_n}$ werden von rechts nach links aufgelöst, d.h. das Ergebnis von `prj1` auf dem genannten Tupel ist $\mathtt{x_1} \mapsto \mathtt{x_2} \mapsto \cdots \mapsto \mathtt{x_{n-1}}$.

Als Bezeichner für die Projektionsfunktionen ist der Name der dynamischen Komponente mit dem Zusatz `_q` zu verwenden. Dies behält den Komponentennamen zur Verwendung für Variablen vor und verweist auf die Verbindung zum Zustand. Als Label für die Axiome zur Typisierung wird wieder das Suffix `_t` genutzt; die Definition der Projektionsfunktionen erfolgt zusammengefasst im Axiom `prj_d`.

Konsistenzbedingungen. Etwaig vorhandene Konsistenzbedingungen müssen zusätzlich im Zustandsraum vermerkt werden. Seien die Bedingungen etwa $\chi_1, \chi_2, \ldots, \chi_m$ (jede sollte dabei von wenigstens einer der Variablen $\mathtt{dyn_1}, \mathtt{dyn_2}, \ldots, \mathtt{dyn_n}$ abhängig sein), so muss die obige Spezifikation nach folgendem Muster erweitert werden.

CONTEXT meta_state

$\vdots$

AXIOMS

$$
\begin{array}{l}
\quad \texttt{st: STATE} = \{\texttt{dyn}_1 \mapsto \texttt{dyn}_2 \mapsto \cdots \mapsto \texttt{dyn}_\texttt{n} \mid \\
\qquad\quad \texttt{dyn}_1 \in \texttt{DYN}_1 \land \texttt{dyn}_2 \in \texttt{DYN}_2 \land \cdots \land \texttt{dyn}_\texttt{n} \in \texttt{DYN}_\texttt{n} \land \\
\qquad\quad \chi_1 \land \chi_2 \land \cdots \land \chi_\texttt{m} \\
\qquad \} \\
\quad \vdots \\
\textbf{END}
\end{array}
$$

Wie oben vermerkt (vgl. Unterabschnitt 3.3.3), können strengere Konsistenzbedingungen gelten, welche Zwischenzustände ausschließen. Diese werden in einem zusätzlichen Raum CONSISTENT_STATE $\subset$ STATE vermerkt. Seien diese Bedingungen $\chi'_1, \chi'_2, \ldots, \chi'_k$, so wird die Spezifikation dann wie folgt erweitert.

$$
\begin{array}{l}
\textbf{CONTEXT}\ \texttt{meta\_state} \\
\vdots \\
\textbf{AXIOMS} \\
\quad \vdots \\
\quad \texttt{cst: CONSISTENT\_STATE} = \{\texttt{dyn}_1 \mapsto \texttt{dyn}_2 \mapsto \cdots \mapsto \texttt{dyn}_\texttt{n} \mid \\
\qquad\quad \texttt{dyn}_1 \mapsto \texttt{dyn}_2 \mapsto \cdots \mapsto \texttt{dyn}_\texttt{n} \in \texttt{STATE} \land \\
\qquad\quad \chi'_1 \land \chi'_2 \land \cdots \land \chi'_\texttt{k} \\
\qquad \} \\
\quad \texttt{cst\_1: CONSISTENT\_STATE} \subseteq \texttt{STATE} \quad \texttt{theorem} \\
\textbf{END}
\end{array}
$$

Durch die Angabe von $\texttt{dyn}_1 \mapsto \texttt{dyn}_2 \mapsto \cdots \mapsto \texttt{dyn}_\texttt{n} \in \texttt{STATE}$ gelten die $\chi_1, \chi_2, \ldots, \chi_m$ ebenfalls. Die Eigenschaft `cst_1`, dass die Menge konsistenter Zustände eine Teilmenge des gesamten Zustandsraumes ist, sollte als Theorem und damit als beweispflichtig markiert werden.

Variablen zum Zustandsraum. Zum einfacheren Handling ist es gerade in Beweisen sinnvoll, auf die Elemente des Zustandsraumes nicht über die Projektionsfunktionen, sondern über einzelne Variablen zuzugreifen. Diese Möglichkeit beschreibt das folgende Axiom `st_v`; es kann im interaktiven Theorembeweiser leicht eingesetzt werden, um die Variablen zu erzeugen. Das ebenfalls angegebene Axiom `st_c` überträgt die oben vermerkten Konsistenzbedingungen $\chi_1, \ldots, \chi_\texttt{m}$ auf die Variablen.

$$
\begin{array}{l}
\textbf{CONTEXT}\ \texttt{meta\_state} \\
\vdots \\
\textbf{AXIOMS} \\
\quad \vdots \\
\quad \texttt{st\_v}: \ \forall\, \texttt{q} \cdot \texttt{q} \in \texttt{STATE} \Rightarrow (\\
\qquad\quad \exists\, \texttt{dyn}_1, \texttt{dyn}_2, \ldots, \texttt{dyn}_\texttt{n} \cdot \\
\qquad\qquad \texttt{q} = \texttt{dyn}_1 \mapsto \texttt{dyn}_2 \mapsto \cdots \mapsto \texttt{dyn}_\texttt{n} \land
\end{array}
$$

$$
\begin{array}{l}
\qquad\qquad \mathtt{dyn_1\_q(q)} = \mathtt{dyn_1} \wedge \\
\qquad\qquad \mathtt{dyn_2\_q(q)} = \mathtt{dyn_2} \wedge \\
\qquad\qquad \vdots \\
\qquad\qquad \mathtt{dyn_n\_q(q)} = \mathtt{dyn_n} \\
\qquad) \quad \textbf{theorem} \\
\mathtt{st\_c}\text{: } \forall\, \mathtt{dyn_1}, \mathtt{dyn_2}, \ldots, \mathtt{dyn_n} \cdot (\\
\qquad\qquad \exists\, \mathtt{q} \cdot \mathtt{q} \in \mathtt{STATE} \wedge \mathtt{q} = \mathtt{dyn_1} \mapsto \mathtt{dyn_2} \mapsto \cdots \mapsto \mathtt{dyn_n} \\
\qquad) \Rightarrow \chi_1 \wedge \chi_2 \wedge \cdots \wedge \chi_m \qquad \textbf{theorem} \\
\textbf{END}
\end{array}
$$

Beide Axiome sollten als Theorem und damit als beweispflichtig markiert werden. Bei Einsatz des Zustandsraumes unter strengeren Konsistenzbedingungen in Axiom `cst` muss zudem ein zu `st_c` äquivalentes Axiom `cst_c` definiert werden, das die strengen Bedingungen $\chi'_1, \chi'_2, \ldots, \chi'_k$ auf Variablen überträgt.

3.4.5 Primitivoperationen

Der abstrakten Beschreibung folgend (vgl. Unterabschnitt 3.3.4), muss jede Primitivoperation als Konstante in die Spezifikation einfließen; und entsprechend den obigen Überlegungen zur Modularisierung (vgl. Unterabschnitt 3.4.1) sind hierfür ein oder mehrere eigene Kontexte, nachfolgend `meta_op` genannt, zu verwenden.

Jede Operation ist also eine Abbildung von Zustandsraum auf Zustandsraum unter weiteren Argumenten, seien dies n Stück:

$$\mathsf{prim}_i : \mathsf{STATE} \times \mathsf{T}_1^{o_i} \times \mathsf{T}_2^{o_i} \times \cdots \times \mathsf{T}_n^{o_i} \to \mathsf{STATE}.$$

Zuerst werden pro Operation Name, Typ und Semantik definiert.

```
CONTEXT meta_op
EXTENDS
   meta_state
CONSTANTS
   prim_i
AXIOMS
   prim_i_t: prim_i ∈ STATE × T_1^o_i × T_2^o_i × ··· × T_n^o_i → STATE
   prim_i_d: ∀ q, a_1, a_2, ..., a_n · (
             q ∈ STATE ∧
             a_1 ∈ T_1^o_i ∧ a_2 ∈ T_2^o_i ∧ ··· ∧ a_n ∈ T_n^o_i
          ) ⇒ prim_i(q ↦ a_1 ↦ a_2 ↦ ··· ↦ a_n) = q'
END
```

Dabei ist der Ausdruck $\mathtt{q}'$ durch eine Beschreibung des Zielzustandes zu ersetzen. Diese entsteht typischerweise, indem der Zustand in Tupelschreibweise notiert wird, wobei die einzelnen Werte des Ausgangszustandes über die Projektionsfunktionen $\mathtt{dyn_j\_q(q)}$ zu bezeichnen sind. Veränderungen können dann anhand mathematischer Operationen angegeben werden.

Beispiel 3.8. Die *addUsers*-Operation (vgl. Seite 16), welche eine via Argument übergebene Nutzermenge zur im Zustand enthaltenen Menge der Nutzer hinzufügen muss, wäre also wie folgt zu spezifizieren:

$$\begin{aligned}\texttt{addUsers}(\texttt{q} \mapsto \texttt{u}) = (\texttt{U\_q}(\texttt{q}) \cup \texttt{u}) \mapsto \texttt{S\_q}(\texttt{q}) \mapsto \\ \texttt{UA\_q}(\texttt{q}) \mapsto \texttt{user\_q}(\texttt{q}) \mapsto \\ \texttt{roles\_q}(\texttt{q})\end{aligned}$$ △

Vorbedingungen. Wie in Vorgehensweise (B7) argumentiert, sind die Vorbedingungen der Kommandos im Autorisierungsschema so weit wie möglich einzuschränken. So soll etwa eine Sitzung, die bereits existiert, nicht noch einmal erstellt werden können oder ein vorhandener Nutzer nicht nochmals zum System hinzufügbar sein. Nun kombinieren anwendungsspezifische Kommandos oft mehrere Primitivoperationen, um ihre Semantik zu beschreiben – sie sind also im Vergleich zu Primitivoperationen komplexer und weniger überschaubar. Da nun die Beweise für die Korrektheit der Kommandos auf den Definitionen der Primitivoperationen beruht ist es hilfreich, auch die Vorbedingungen für Primitivoperationen so weit wie möglich einzuschränken: zu allgemeine Vorbedingungen der Kommandos fallen dann dadurch auf, dass nicht alle Voraussetzungen für einen Beweis vorhanden sind.

Beispiel 3.9. Erneut wird die `addUsers`-Operation betrachtet. Um ihre Vorbedingungen so weit wie möglich einzuschränken wird sie so definiert, dass die hinzuzufügenden Nutzer noch nicht im Zustand vorhanden sein dürfen.

$$\begin{aligned}&\forall\, \texttt{q}, \texttt{u} \cdot (\\ &\quad \texttt{q} \in \texttt{STATE} \land \texttt{u} \in \mathbb{P}(\texttt{USER}) \land \\ &\quad \texttt{u} \cap \texttt{U\_q}(\texttt{q}) = \varnothing \\ &) \Rightarrow \texttt{addUsers}(\texttt{q} \mapsto \texttt{u}) = \ldots\end{aligned}$$ △

Beweispflichten. Da Primitivoperationen letztlich eingesetzt werden, um den Nachzustand der Operation einer Zustandsmaschine zu definieren, bedarf an dieser Stelle die Beweispflicht (P7) besondere Aufmerksamkeit. Bildet eine Primitivoperation auf einen Wert ab, der nicht Element des Zustandsraumes ist – etwa, weil er Konsistenzbedingungen verletzt –, könnte diese Beweispflicht nicht mehr gezeigt werden. Es handelt sich hier aber um ein Problem, das bei der Spezifikation des Metamodells entsteht und erst bei der Spezifikation eines Sicherheitsmodells bemerkt wird. Daher ist es sinnvoll, bereits während der Metamodell-Spezifikation zu prüfen, ob die Primitivoperationen tatsächlich in den Zustandsraum abbilden.

Dazu muss zu jeder Primitivoperation ein zusätzliches Theorem $\texttt{prim}_\texttt{i}\texttt{\_v}$ definiert werden, welches zeigt, dass der Zielzustand wieder im Zustandsraum liegt und somit dessen Konsistenzbedingungen nicht verletzt. Es sind dabei die selben Bedingungen wie im Axiom $\texttt{prim}_\texttt{i}\texttt{\_d}$ anzugeben.

CONTEXT meta_op

⋮

AXIOMS

⋮

$\mathtt{prim_i\_d}$: $\forall\, \mathtt{q}, \mathtt{a_1}, \mathtt{a_2}, \ldots, \mathtt{a_n} \cdot ($
$\mathtt{q} \in \mathtt{STATE} \wedge$
$\mathtt{a_1} \in \mathtt{T_1^{o_i}} \wedge \mathtt{a_2} \in \mathtt{T_2^{o_i}} \wedge \cdots \wedge \mathtt{a_n} \in \mathtt{T_n^{o_i}}$
$) \Rightarrow \mathtt{prim_i}(\mathtt{q} \mapsto \mathtt{a_1} \mapsto \mathtt{a_2} \mapsto \cdots \mapsto \mathtt{a_n}) = \mathtt{q}'$ > Def. wie oben

$\mathtt{prim_i\_v}$: $\forall\, \mathtt{q}, \mathtt{a_1}, \mathtt{a_2}, \ldots, \mathtt{a_n} \cdot ($
$\mathtt{q} \in \mathtt{STATE} \wedge$
$\mathtt{a_1} \in \mathtt{T_1^{o_i}} \wedge \mathtt{a_2} \in \mathtt{T_2^{o_i}} \wedge \cdots \wedge \mathtt{a_n} \in \mathtt{T_n^{o_i}}$
$) \Rightarrow \mathtt{q}' \in \mathtt{STATE}$

END

Der Ausdruck $\mathtt{q}'$ steht dabei wie oben für die Spezifikation der Semantik der Operation. Die entstehende Codeduplikation ist mit derzeitigem Entwicklungsstand der Werkzeuge nicht zu vermeiden. Größte Vorsicht ist geboten, um die beiden $\mathtt{q}'$-Ausdrücke stets identisch zu halten.

Eigenschaften. Aus der abstrakten Beschreibung (vgl. Unterabschnitt 3.3.4) geht hervor, dass zu den Primitivoperationen bestimmte Eigenschaften, wie etwa Idempotenz oder Reversibilität als Theoreme notiert werden können. Solche Theoreme führen Redundanz in die Spezifikation ein, wodurch ihr Beweis die Sicherheit bezüglich der Korrektheit der gesamten Spezifikation erhöht. Zur Benennung von Theoremen, welche sich nur auf eine Operation beziehen, wird dabei die Variante der schlichten Nummerierung empfohlen; bei Theoremen, welche Eigenschaften für eine Kombination von Operationen angeben, kann ein Kurzname für die Eigenschaft genutzt werden, wie etwa `idem` für die Idempotenz.

Ebenfalls als Theoreme sollten einige Abkürzungen für Beweise formuliert werden. Dabei kann es insbesondere wieder darum gehen, den Umgang mit Projektionsfunktionen zu vereinfachen, indem der Effekt einer Operation auf einzelne Zustandskomponenten (statt auf den gesamten Zustand) beschrieben wird.

Beispiel 3.10. Die obige `addUsers`-Operation kann in Bezug auf die `U_q`-Projektion definiert werden:

$$
\begin{aligned}
&\forall\, \mathtt{q}, \mathtt{u} \cdot (\\
&\quad \mathtt{q} \in \mathtt{STATE} \wedge \mathtt{u} \in \mathbb{P}(\mathtt{USER}) \wedge \\
&\quad \mathtt{u} \cap \mathtt{U\_q}(\mathtt{q}) = \varnothing \\
&) \Rightarrow \mathtt{U\_q}(\mathtt{addUsers}(\mathtt{q} \mapsto \mathtt{u})) = \mathtt{U\_q}(\mathtt{q}) \cup \mathtt{u}
\end{aligned}
$$

△

3.4.6 Primitivprädikate

Die Primitivprädikate (abstrakte Beschreibung siehe Unterabschnitt 3.3.5) verhalten sich ähnlich zu den Primitivoperationen. Da sie jedoch nicht in den Zustandsraum, sondern lediglich auf die Menge der Wahrheitswerte abbilden, ist ihre Spezifikation einfacher. Sie sollte gemäß den Überlegungen zur Modularisierung (siehe Unterabschnitt 3.4.1) wieder in einem der `meta_op`-Kontexte erfolgen, und zwar am günstigsten in einem eigenen Kontext, wie etwa `meta_predicate`.

Ein Primitivprädikat pred_i ist ein Konstantensymbol mit Typ

$$\mathsf{pred}_i : \mathsf{STATE} \times \mathsf{T}_1^{p_i} \times \mathsf{T}_2^{p_i} \times \cdots \times \mathsf{T}_{\mathrm{n}(p_i)}^{p_i} \to \mathsf{BOOL}.$$

Zur Spezifikation in Event-B wird die `bool(f)`-Funktion genutzt, welche den Wahrheitswert einer prädikatenlogischen Formel `f` repräsentiert.

Jede der Primitivoperationen pred_i wird nach folgendem Muster spezifiziert.

CONTEXT meta_predicate

EXTENDS

meta_state

CONSTANTS

$\mathtt{pred_i}$

AXIOMS

$\mathtt{pred_i\_t}$: $\mathtt{pred_i} \in \mathtt{STATE} \times \mathtt{T_1^{p_i}} \times \mathtt{T_2^{p_i}} \times \cdots \times \mathtt{T_n^{p_i}} \to \mathtt{STATE}$

$\mathtt{pred_i\_d}$: $\forall \mathtt{q}, \mathtt{a_1}, \mathtt{a_2}, \ldots, \mathtt{a_n} \cdot ($
$\mathtt{q} \in \mathtt{STATE} \land$
$\mathtt{a_1} \in \mathtt{T_1^{p_i}} \land \mathtt{a_2} \in \mathtt{T_2^{p_i}} \land \cdots \land \mathtt{a_n} \in \mathtt{T_n^{p_i}}$
$) \Rightarrow \mathtt{pred_i}(\mathtt{q} \mapsto \mathtt{a_1} \mapsto \mathtt{a_2} \mapsto \cdots \mapsto \mathtt{a_n}) = \mathtt{bool}(\mathtt{f}(\mathtt{q}, \mathtt{a_1}, \ldots, \mathtt{a_n}))$

END

Dabei ist $\mathtt{f}(\mathtt{q}, \mathtt{a_1}, \ldots, \mathtt{a_n})$ durch die prädikatenlogische Formel zu ersetzen, welche den Wahrheitswert des Prädikates beschreibt. Die Angabe weiterer Eigenschaften oder das Beachten von Beweispflichten ist bei Primitivprädikaten im Allgemeinen nicht nötig.

3.4.7 Gesamtkontext

Wie aus den Überlegungen zur Modularisierung (vgl. Unterabschnitt 3.4.1) hervorgeht, fasst ein Kontext, welcher nur den Namen des Metamodells – also hier `meta` – trägt, alle vorangegangenen Kontexte zusammen, indem er die Kontexte der Operationen erweitert. Seien dies $\mathtt{meta\_op_1}, \mathtt{meta\_op_2}, \ldots, \mathtt{meta\_op_n}$ sowie `meta_predicate`, so erfolgt seine Spezifikation nach dem folgenden Schema.

CONTEXT meta

EXTENDS

$\mathtt{meta\_op_1}$

```
    meta_op_2
    ⋮
    meta_op_n
    meta_predicate
END
```

Der Kontext `meta` enthält damit die in allen Kontexten des Metamodells spezifizierten Trägermengen, Konstanten, Axiome und Theoreme.

3.5 Abstrakte Spezifikation eines Sicherheitsmodelles

Nachdem die Abschnitte 3.3 und 3.4 die Spezifikation eines Metamodelles beschrieben haben, sollen nun die davon abgeleiteten anwendungsspezifischen Sicherheitsmodelle formal erfasst werden. Den Vorgang der Spezifikation dieser Modelle beschreibt zuerst dieser Abschnitt abstrahiert von jeder Spezifikationssprache; hierbei geht er nacheinander auf all die Aspekte ein, welche sich im Allgemeinen nicht über das Metamodell zusammenfassen lassen. Dies sind die Deklarationen atomarer Objekte als Elemente der Wertebereiche, die Initialisierung statischer Komponenten, das Einbinden des Zustandsraumes und somit der dynamischen Komponenten, die Initialisierung derselben im Initialzustand sowie die Spezifikation des Autorisierungsschemas (Unterabschnitte 3.5.1 bis 3.5.5). Anschließend beschäftigt sich der Abschnitt 3.6 mit der Spezifikation in Event-B.

3.5.1 Elemente der Wertebereiche

Ein Sicherheitsmodell benötigt zur Spezifikation der Werte statischer Komponenten sowie des initialen Zustandes dynamischer Komponenten konkrete Elemente aus seinen Wertebereichen (etwa benannte Benutzer, Rollen, Rechte etc.), die im Metamodell zumeist nicht bekannt sind (vgl. Unterabschnitt 3.3.1). Solche für das Sicherheitsmodell atomaren Objekte sind in jedem Fall Element eines der Wertebereiche T_i des Sicherheitsmodells, und daher als Element einer der Trägermengen T_i zu spezifizieren. Es ist also im Modellkontext zu jedem atomaren Objekt $e_j \in T_i$ eine Konstante e_j einzuführen, welche vom Typ T_i ist. Des weiteren sind Axiome zu formulieren, welche die paarweise Verschiedenheit der e_j ausdrücken. Werden die Elemente einer Trägermenge *vollständig* benannt, so muss ein weiteres Axiom dies ausdrücken, also etwa

$$\mathsf{T}_i = \{\mathsf{e}_{j(1)}, \mathsf{e}_{j(2)}, \ldots, \mathsf{e}_{j(n)}\}.$$

3.5.2 Initialisierung statischer Komponenten

Den im Metamodell definierten statischen Komponenten (vgl. Unterabschnitt 3.3.2) sind spätestens im Sicherheitsmodell ihre konkreten Werte v^S zuzuweisen, die

anhand der oben in Unterabschnitt 3.5.1 eingeführten atomaren Objekten definiert werden. Diese Wertzuweisungen sind in den Axiomen des Modellkontextes möglich, und müssen konsistent zu den als Axiome spezifzierten Eigenschaften der statischen Komponenten (vgl. Unterabschnitt 3.3.2) sein.

Sei also S_i eine statische Komponente und stat_i die zugehörige Konstante der Spezifikation; ihr im Sicherheitsmodell vorgegebener Wert sei v_i^S. Es ist dann ein Axiom $\mathsf{stat}_i = v_i^S$ zu formulieren.

3.5.3 Einbinden der dynamischen Komponenten

Der Zustandsraum des Sicherheitsmodells wird durch das Metamodell festgelegt: es handelt sich um die Konstanten STATE sowie eventuell CONSISTENT_STATE (vgl. Unterabschnitt 3.3.3). In der zustandsbasierte Spezifikation wird der Zustand anhand der Variablen x und eventuell der Invarianten $\mathsf{I}(\mathsf{x}, \mathsf{c})$ definiert. Es ist entsprechend eine Zustandsvariable state einzuführen, welche Element von STATE und, falls vorhanden, CONSISTENT_STATE ist. Zusätzliche Zustandsvariablen oder Invarianten sind im Allgemeinen nicht nötig.

3.5.4 Initialzustand

Zu jedem Sicherheitsmodell existiert ein Initialzustand, der den dynamischen Komponenten $D_1, \ldots, D_n$ ihre Werte $(v_1^D, v_2^D, \ldots, v_n^D) = q_0$ zuordnet. Die Entsprechung in der Zustandsmaschine ist deren Initialzustand $\mathsf{v}(\mathsf{c})$, der konsistent zur Definition der Zustandsvariablen und Invarianten sein muss. Die Definition des Initialzustandes erfolgt (analog zur Initialisierung statischer Komponenten) anhand der definierten Elemente der Trägermengen.

Auswahl des Initialzustandes. Der Initialzustand ist oft ein Zustand des Modells mit minimaler Zahl verwendeter Mengenelemente, Relationstupel etc. unter der Bedingung, dass alle erforderlichen Zustände von ihm aus erreichbar sind. Um diese Anforderungen zu erfüllen, bedarf es in Sicherheitsmodellen beispielsweise der Definition eines Nutzers mit hinreichend vielen Rechten, welcher dann die Operationen des Autorisierungsschemas nutzt, um etwa weitere Nutzer einzutragen oder Rechte zu vergeben. Alternativ kann der Initialzustand etwa den Zustand des Systems bei Auslieferung widerspiegeln. In jedem Fall sind verwendete Elemente zu den Trägermengen anzugeben (siehe Unterabschnitt 3.5.1).

Beispiel 3.11. Das RBAC-Beispielmodell (vgl. Unterabschnitt 2.4.2) verfügt über einen Initialzustand, in welchem ein Administrator definiert ist. Dieser kann sich dann durch das *login*-Kommando eine Sitzung erstellen, mittels *createUser* weitere Nutzer erstellen und ihnen durch *assignRole* Rollen zuordnen.

Wäre der Administrator nicht vordefiniert, bestünde keine Möglichkeit, dem System durch Operationen des Autorisierungsschemas Nutzer hinzuzufügen; somit wären keine weiteren Zustände erreichbar. Entdecken lassen sich solche Entwurfsfehler etwa mit einem Animator (vgl. Unterabschnitt 2.6.2). △

3.5.5 Autorisierungsschema

Das Autorisierungsschema eines Sicherheitsmodells besteht aus den Kommandos κ_i sowie den anwendungsspezifischen Prädikaten π_i, welche über das Eingabealphabet Σ, die Zustandsüberführungsfunktion δ und die Ausgabefunktion λ zusammen mit Ausgaberaum Γ definiert sind. Es beschreibt die Zustandsübergänge und somit das Verhalten des zustandsbasierten Sicherheitsmodells. Da das Spezifikationsvorgehen lediglich ein Wechsel der Darstellungsform ist (vgl. Abschnitt 3.1), ist das Autorisierungsschema somit in die Operationen $\mathsf{o}_i : [\mathsf{p}_i, \mathsf{pre}_i, \mathsf{post}_i]$ der zu spezifizierenden Zustandsmaschine zu überführen.

Es soll nun für ein j das Kommando κ_j in Operation o_j überführt werden. Gemäß der vorliegenden Notation eines Autorisierungsschemas (vgl. Unterabschnitt 2.3.2) besteht die Definition des Kommandos neben seinem Namen aus einer Liste von Parametern x_i, einem auf Primitivprädikaten p_i aufbauenden Bedingungsteil, sowie einem über Primitivoperationen o_i definierten Anweisungsteil. Die Parameter sind über den Trägermengen T_i des Modells definiert, welche auch in der Spezifikation vorhanden sind, sodass sie als Parameter $\mathsf{p}_{j,i}$ in die Spezifikation der Operation übernommen werden können. Der Bedingungsteil kann als Vorbedingung pre_j der Operation ausgedrückt werden, die Primitivprädikate stehen als pred_i zur Verfügung (vgl. Unterabschnitt 3.3.5). Entsprechend wird der Anweisungsteil in die Nachbedingungen post_j der Operation überführt, die Primitivoperationen sind als prim_i spezifiziert (vgl. Unterabschnitt 3.3.4).

Der Anweisungsteil einer Operation $\mathsf{o}_i : [\mathsf{p}_i, \mathsf{pre}_i, \mathsf{post}_i]$ besteht dabei aus Zuweisungen der Form $\mathsf{x}_k := \mathsf{a}(\mathsf{p}_i, \mathsf{x}, \mathsf{c})$, wobei x_k eine Zustandsvariable und a ein Wert ist, dessen Berechnung von den sichtbaren Parametern, Zustandsvariablen (des Vorzustandes) und Konstanten abhängen kann. Sofern die Spezifikation des Sicherheitsmodells lediglich die Zustandsvariable state aufweist, besteht der Anweisungsteil (abgesehen von möglichen Rückgabewerten) aus genau einer Zuweisung, dessen Wert sich aus einer verketteten Anwendung der Primitivoperationen prim_i ergibt. Sollte die Spezifikation mehrere Zustandsvariablen aufweisen, so muss dafür Sorge getragen werden, dass nicht zu verändernde Variablen tatsächlich als unverändert spezifiziert werden, etwa durch $\mathsf{x}_k := \mathsf{x}_k$ (Event-B-Spezifikationen erfordern diese Angabe nicht).

Beispiel 3.12. Das Kommando *login* des Autorisierungsschemas im RBAC-Beispielmodell (vgl. Gleichung (C5), Unterabschnitt 2.4.2) ist definiert als

command login $(u \in U, s \in S)$
createSessions $(\{s\})$;
mapUserSessions $(\{\,(s, u)\,\})$
end command.

Die zugehörige Operation der Zustandsmaschine müsste die Zuweisung im Anweisungsteil wie folgt spezifizieren:

$$\mathsf{state} := \mathsf{mapUserSessions}(\,\mathsf{createSessions}(\mathsf{state}, \{\mathsf{s}\}),\ \{(\mathsf{s}, \mathsf{u})\}\,),$$

wobei $\mathsf{mapUserSessions}, \mathsf{createSessions}, \mathsf{s}$ und u die Entsprechungen der gleichnamigen Objekte des Modells in der Spezifikation sind. △

Die Operationen der Zustandsmaschine können, entsprechend der Ausgabefunktion λ, über Rückgabewerte verfügen. In der vorliegenden Notation des Autorisierungsschemas trifft dies auf die boolschen Rückgaben der anwendungsspezifischen Prädikate π_i zu. Ein entsprechender Ausgabeparameter oder Rückgabewert ist zu spezifizieren.

3.6 Spezifikation eines Sicherheitsmodelles in Event-B

Die Vorgehensweise zur Spezifikation eines Sicherheitsmodells in Event-B zu einem gegebenen Metamodell (vgl. Abschnitte 3.3 und 3.4) beschreibt der vorliegende Abschnitt, und implementiert hierzu die abstrakte Vorgehensweise aus Abschnitt 3.5. Dabei beschäftigt er sich zuerst nochmals mit der Modularisierung der Spezifikation (Unterabschnitt 3.6.1) und anschließend mit den einzelnen zu spezifizierenden Komponenten, nämlich mit atomaren Objekten als Elemente der Trägermengen, weiterhin mit der Initialisierung statischer Komponenten, mit Zustandsvariablen und Invarianten, dem Initialzustand sowie dem Autorisierungsschema (Unterabschnitte 3.6.2 bis 3.6.6).

3.6.1 Modularisierung

Gemäß Unterabschnitt 3.2.2 erfolgt die Verbindung zwischen Meta- und konkretem Sicherheitsmodell durch Einbindung des Metamodell-Kontextes. Da auch ein konkretes Sicherheitsmodell verschiedene Konstanten und Axiome mitbringt und somit über einen eigenen Kontext verfügen muss, dieser jedoch gewöhnlich keine sonderliche Komplexität aufweist, wird hier vorgeschlagen, zu jedem konkreten Sicherheitsmodell `inst` einen Kontext `inst_context` zu verwenden, welcher den Metamodell-Kontext erweitert.

3.6.2 Elemente der Trägermengen

Zu einigen der Trägermengen muss das Sicherheitsmodell atomare Objekte als Elemente benennen (vgl. Unterabschnitt 3.5.1). Dies ist in Event-B über die Deklaration der Elemente als Konstanten, sowie der Zuordnung via Axiome möglich. Zu einer Trägermenge SET_i können Elemente $\mathsf{e}_1, \mathsf{e}_2, \ldots, \mathsf{e}_n$ dabei wie folgt eingeführt werden:

CONTEXT inst_context

EXTENDS

meta

CONSTANTS

e_1

e_2

$\vdots$

e_n

AXIOMS

SET_i_p: partition$(SET_i, \{e_1\}, \{e_2\}, \ldots, \{e_n\})$

END

Der Ausdruck partition$(M, P_1, P_2, \ldots)$ gibt dabei an, dass $P = \{P_1, P_2, \ldots\}$ eine Partition von M ist. Indem sich jedes e_j in einer anderen Teilmenge der Partition befindet wird die paarweise Verschiedenheit aller e_j ausgedrückt. partition leistet auch die Typisierung der e_j als Element von SET_i.

3.6.3 Initialisierung statischer Komponenten

Ebenfalls im Kontext inst_context des Sicherheitsmodells sind konkrete Werte für statische Komponenten zu spezifizieren (vgl. Unterabschnitt 3.5.2). Das Event-B-Axiom zur Angabe des Initialwertes v_i^S für Komponente stat_i lautet dann

$stat_i$: $stat_i = v_i^S$.

3.6.4 Zustandsvariablen und Invarianten

Der Zustand des Sicherheitsmodells wird bereits im Metamodell bestimmt (vgl. Unterabschnitte 3.3.3 und 3.4.4) und muss nun in die Modellspezifikation eingebunden werden. Entsprechend der abstrakten Beschreibung in Unterabschnitt 3.5.3 geschieht dies über das Einführen der Variable state als Element von STATE bzw. CONSISTENT_STATE. In Event-B sind diese Angaben Teil der Zustandsmaschine selbst; die Abschnitte VARIABLES und INVARIANTS (analog zu Konstanten und Axiomen des Kontextes) sind hierfür zu nutzen.

Die Spezifikation einer Zustandsmaschine inst unter Einsatz des Zustandes aus dem Metamodell sieht wie folgt aus:

MACHINE inst

SEES inst_context

VARIABLES

state

INVARIANTS

inv1: state $\in$ CONSISTENT_STATE

inv2: state $\in$ STATE theorem

END

Die Variable state wird nach Axiom inv1 als Element von CONSISTENT_STATE typisiert. Zur Erleichterung von Beweisen führt inv2 als Theorem die Aussage state $\in$ STATE ein, die wahr ist, da CONSISTENT_STATE $\subseteq$ STATE (vgl. Unterabschnitt 3.5.3).

3.6.5 Initialzustand

Der Initialzustand $(v_1^D, v_2^D, \ldots, v_n^D) = q_0$ des Modells, welcher in den Initialzustand der spezifizierten Zustandsmaschine überführt wurde (vgl. Unterabschnitt 3.5.4), kann in Event-B über eine spezielle Operation (in Event-B: ein spezielles „Ereignis“, siehe dazu Unterabschnitt 3.6.6) mit dem Namen `INITIALISATION` formuliert werden. Darin muss der Zustandsvariable state ein Wert zugeordnet werden. Da der Zustandsraum als Kreuzprodukt definiert ist (vgl. Unterabschnitt 3.3.3), erfolgt die Angabe des Initialzustandes in Tupelschreibweise.

Das Schema zur Formulierung des Initialzustandes lautet wie folgt.

MACHINE inst

⋮

EVENTS

INITIALISATION $\widehat{=}$

begin

act1: state $:= \mathtt{v}_1^{\mathtt{D}} \mapsto \mathtt{v}_2^{\mathtt{D}} \mapsto \cdots \mapsto \mathtt{v}_{\mathtt{n}}^{\mathtt{D}}$

end

END

3.6.6 Autorisierungsschema

Wie Unterabschnitt 3.5.5 argumentiert, werden anwendungsspezifische Kommandos und Prädikate in der Zielspezifikation als Operationen $\mathtt{o}_i : [\mathsf{p}_i, \mathsf{pre}_i, \mathsf{post}_i]$ ausgedrückt. Die Notation ihrer Semantik erfolgt anhand der Primitivoperationen und -prädikate des Metamodells.

In Event-B heißen Operationen *Ereignisse*, die Semantik ist gleich: ein Ereignis ist für eine Zustandsmaschine atomar; es verfügt über einen Namen sowie über Parameter. Die Vorbedingungen heißen *Guards*: ein Ereignis kann nur eintreten, wenn alle Guards (das sind prädikatenlogische Aussagen) erfüllt sind. Die Nachbedingungen werden über deterministische[4] Zuweisungsoperationen der Form $\mathtt{x} := \mathtt{a}$ modelliert.

Die Notation erfolgt nach folgendem Schema:

MACHINE ...

⋮

EVENTS

$\mathtt{event}_\mathtt{i}$ $\widehat{=}$

any

$\mathtt{p}_1$ > Parameter

$\mathtt{p}_2$

⋮

[4] Auch nichtdeterministische Zuweisungen sind möglich, werden hier jedoch nicht betrachtet.

```
        p_n
    where
        grd1: G_1      > Guards
        grd2: G_2
        ⋮
        grdm: G_m
    then
        act1: x_1 := a_1      > Aktionen
        act2: x_2 := a_2
        ⋮
        actl: x_ℓ := a_ℓ
    end
END
```

Zu jedem Kommando κ_i ist nun ein entsprechendes Ereignis mit gleichem Namen zu erstellen. Die *Parameter* des Kommandos sind auch die Ereignisparameter, ihre Typisierung erfolgt über Guards. Der *Bedingungsteil* der Kommandos gilt als Vorbedingung pre_i und ist somit ebenfalls in Guards zu überführen. Dabei empfiehlt es sich aufgrund der Semantik der Guards (um die Vorbedingungen des Ereignisses zu erfüllen, müssen *alle* Guards erfüllt sein), die Bedingung des Kommandos in konjunktive Normalform zu überführen und dann jeden Disjunktionsterm als einen Guard auszudrücken. Im einfachsten Fall enthält jeder Guard ein Primitivprädikat. Gemäß der Argumentation zu Vorgehensweise (B7) sollten die Ereignisse dabei über möglichst starke Vorbedingungen verfügen. In späteren Verfeinerungen können bei Bedarf auch Ereignisse mit abgeschwächten Vorbedingungen eingeführt werden: sie spezifizieren dann das Verhalten bei Nichtzutreffen der starken Vorbedingungen. Der *Anwendungsteil* schließlich besteht, wie erläutert, aus deterministischen Zuweisungen. Dabei ist im Allgemeinen nur der Zustandsvariable `state` ein Wert zuzuweisen, welcher aus der Anwendung von Primitivoperationen auf `state` entsteht (wie in Unterabschnitt 3.5.5 beschrieben).

Da unterschiedliche Guards mit verschiedenen Aufgaben eingesetzt werden, empfiehlt sich auch hier eine passende Benennung durch die Labels. So können etwa Parameter typisierende Guards mit Labels `typ1`, `typ2` etc. versehen werden, während Guards, die aus dem Bedingungsteil eines Kommandos hervorgehen, `cnd1`, `cnd2` etc. heißen können.

Beweispflichten. Von besonderer Bedeutung ist die Beweispflicht (P7) der Invariantenerhaltung, in Event-B mit `INV` abgekürzt. Im Kontext des Autorisierungsschemas ist bei dieser Beweispflicht zu zeigen, dass das Ereignis die Maschine stets wieder in einen Zustand im Zustandsraum überführt, und zwar unter Beachtung aller Konsistenzbedingungen (vgl. Unterabschnitt 3.3.3). Dabei gilt als Hypothese die Gültigkeit der Invarianten für den Vorzustand $\mathsf{I}(\mathsf{x}, \mathsf{c})$, sodass angenommen wird, dass dieser alle Konsistenzbedingungen erfüllt. Anhand der Definition der angewendeten Primitivoperationen sowie der notierten Guards ist sodann zu zeigen, dass auch der Nachzustand x' die Konsistenzbedingungen einhält. Wurden die Primitivoperationen unter starken Vorbedingungen spezifiziert (vgl. Unterabschnitt 3.3.4), kann

im Rahmen dieses Beweises festgestellt werden, wenn bestimmte Voraussetzungen zu deren Anwendung nicht erfüllt wurden.

Der erfolgreiche Beweis der Beweispflicht stellt sicher: ausgehend von einem konsistenten Anfangszustand können mit den spezifizierten Ereignissen, unter Beachtung aller Guards, nur konsistente Zustände erreicht werden. Dies ist entscheidend für die Herstellung der Sicherheitseigenschaften des Systems.

Wiederverwendung. Unter Umständen müssen mehrere Ereignisse mit ähnlichen Parametern, Vorbedinungen und Aktionen spezifiziert werden (vgl. etwa die *assign* ... *Role*- und *revoke* ... *Role*-Operationen in Unterabschnitt 2.4.2). Dies bedeutet, dass immer gleiche Ereignisse spezifiziert und eine Vielzahl gleicher Beweise geführt werden müssen, was die Größe der Spezifikation und somit auch die Wahrscheinlichkeit für Fehler erhöht.

Die Lösung des Problems besteht in der von Event-B gebotenen Möglichkeit, in Verfeinerungen von Zustandsmaschinen auch deren Ereignisse zu verfeinern. Die Nomenklatur ist dabei: in einer *konkreten* Maschine, welche eine *abstrakte* Maschine verfeinert, können ein oder mehrere *konkrete* Ereignisse je ein *abstraktes* Ereignis der abstrakten Maschine verfeinern. Die abstrakten Ereignisse sind dabei nicht mehr Bestandteil der konkreten Zustandsmaschine. Konkrete Ereignisse können gegenüber dem abstrakten Ereignis:

- Parameter hinzufügen oder entfernen. Wird ein Parameter entfernt, muss die Konsistenz des konkreten Ereignisses zum abstrakten Ereignis sichergestellt sein. Dazu gibt im konkreten Ereignis ein sogenannter *Zeuge* (engl. Witness) einen Wert an, welchen der entfernte Parameter annimmt. Dieser Wert kann abhängig von Konstanten, Variablen oder anderen Parametern sein und dient zum Vergleich der Guards und Aktionen des konkreten Ereignisses mit dem abstrakten Ereignis.

- die Guards stärken. Die Guards eines konkreten Ereignisses dürfen dabei nur dann gültig werden, wenn die Guards des abstrakten Ereignisses gültig sind.

- die Aktionen verändern. Dabei müssen die Zuweisungen im konkreten Ereignis konsistent zu den Zuweisungen im abstrakten Ereignis sein, d.h. es muss Gleichheit zwischen denjenigen Variablen bestehen, welche sowohl Teil der abstrakten, als auch der konkreten Maschine sind. Das bedeutet insbesondere auch: ein konkretes Ereignis, das kein Ereignis der abstrakten Maschine verfeinert, darf keine Zustandsvariable der abstrakten Maschine verändern.

Bezogen auf das Autorisierungsschema lassen sich also in konkreten Ereignissen die Vorbedingungen stärken sowie einzelne Parameter festschreiben. Dazu muss in der Event-B-Spezifikation der konkreten Maschine die abstrakte Maschine in der `REFINES`-Sektion vermerkt werden. Auch zum konkreten Ereignis werden zusätzliche Angaben gemacht: die dortige `refines`-Sektion benennt das abstrakte Ereignis, die `with`-Sektion weist weggefallenen Parametern Werte zu (der Name des Labels muss dort dem Namen des Parameters entsprechen). Die Konsistenz zwischen abstrakten und konkreten Ereignissen wird dann über Beweispflichten sichergestellt;

für abstrakte Ereignisse gezeigte Aussagen gelten auch für konkrete Ereignisse. Dies sorgt für geringe Komplexität und Fehleranfälligkeit.

Bezüglich des Vorgehens bei der Sicherheitsmodell-Spezifikation bedeutet dies: eine erste Maschine, genannt `inst_generic`, spezifiziert alle Ereignisse, außer jene, welche zu verallgemeinern sind – hier spezifiziert sie nur die verallgemeinerten Versionen. Die Verfeinerung `inst` dieser Maschine übernimmt dann alle Ereignisse außer den verallgemeinerten, und führt jene Ereignisse ein, welche von den verallgemeinerten Ereignissen abgeleitet werden können.

Beispiel 3.13. Im Beispielmodell in Unterabschnitt 2.4.2 werden die Kommandos *assignReferredDoctorRole* und *assignMedicalTeamRole* betrachtet. Beide weisen unter Erfüllung bestimmter Vorbedingungen einem Nutzer eine Rolle zu, was sich zu *assignGenericRole* verallgemeinern lässt. Eine Maschine `healthcare_generic` soll diese verallgemeinerten Ereignisse spezifizieren, während die dazu verfeinerte `healthcare`-Maschine die Spezialisierungen enthält. Alle anderen Ereignisse werden in `healthcare_generic` spezifiziert und in `healthcare` unverändert übernommen.

MACHINE healthcare_generic

SEES healthcare_context

VARIABLES

state

INVARIANTS

⋮

EVENTS

INITIALISATION $\widehat{=}$

begin

act1: state $:=$ $\{\mathtt{u1}\} \mapsto \varnothing \mapsto \{\mathtt{u1} \mapsto \mathtt{UserAdmin}\} \mapsto \varnothing \mapsto \varnothing$

end

createUser $\widehat{=}$

⋮

assignGenericRole $\widehat{=}$

any

s

u

r

where

typ1: $\mathtt{s} \in \mathtt{S\_q}(\mathtt{state})$

typ2: $\mathtt{u} \in \mathtt{U\_q}(\mathtt{state})$

typ3: $\mathtt{r} \in \mathtt{ROLE}$

cnd1: $\mathtt{sod}(\mathtt{state} \mapsto \mathtt{u} \mapsto \mathtt{r}) = \mathtt{TRUE}$

then

act1: $\mathtt{state} := \mathtt{assignUsersToRoles}(\mathtt{state} \mapsto \{\mathtt{u} \mapsto \mathtt{r}\})$

end

```
END
```

```
MACHINE healthcare
REFINES healthcare_generic
SEES healthcare_context
VARIABLES
   state
EVENTS
   INITIALISATION ≙
      extends
         INITIALISATION
      begin
         act1: state := {u1} ↦ ∅ ↦ {u1 ↦ UserAdmin} ↦ ∅ ↦ ∅
      end
   createUser ≙
      extends
         createUser
      any
         ⋮
      end
   ⋮
   assignReferredDoctorRole ≙
      refines
         assignGenericRole
      any
         s
         u
      where
         typ1: s ∈ S_q(state)
         typ2: u ∈ U_q(state)
         cnd1: sod(state ↦ u ↦ ReferredDoctor) = TRUE
         cnd2: cond_SR(state ↦ s ↦ Doctor) = TRUE
         cnd3: cond_UR(state ↦ u ↦ Doctor) = TRUE
      with
         r: r = ReferredDoctor
      then
         act1: state := assignUsersToRoles(state ↦
                  {u ↦ ReferredDoctor}
               )
      end
   assignMedicalTeamRole ≙
      refines
         assignGenericRole
      any
```

```
        ⋮
    where
        ⋮
        cnd1: sod(state ↦ u ↦ MedicalTeam) = TRUE
        cnd2: cond_SR(state ↦ s ↦ MedicalManager) = TRUE
        cnd3: cond_UR(state ↦ u ↦ Doctor) = TRUE ∨
              cond_UR(state ↦ u ↦ Nurse) = TRUE
    with
        ⋮
    then
        ⋮
    end
END
```

Dabei sind die Guards `typ1`, `typ2` und `cnd1` (oder stärkere Guards) Teil aller von `assignGenericRole` abgeleiteten Ereignisse, ebenso wie modifizierte Varianten von `act1`. Konkrete und abstrakte Maschinen teilen sich dieselbe Variable `state`, sodass die dafür gültigen Invarianten in der konkreten Maschine nicht nochmal notiert werden müssen. △

3.7 Zusammenfassung

Ziel des vorliegenden Kapitels war es, eine Methode zur formalen Spezifikation politikbezogener TCB-Funktionen zu erarbeiten. Abschnitt 3.1 erläuterte hierzu die Rahmenbedingungen: Ausgangspunkt ist ein zustandsbasiertes Sicherheitsmodell, welches sich in ein Metamodell eingliedern lässt. Die Spezifikation erfolgt in einer Notation zur Spezifikation von sequenziellen, zustandsbasierten Systemen; sie sollte lediglich einen Wechsel der Darstellungsform des Sicherheitsmodells bedeuten.

Im selben Abschnitt wurden Anforderungen an die Übertragung des Sicherheitsmodells in die formale Spezifikationsnotation gestellt, zu welchen dann Abschnitt 3.2 Lösungsstrategien erarbeitete. Vollständigkeit und externe Konsistenz werden dabei methodisch angegangen, während das Specification Engineering für die interne Konsistenz auf werkzeuggestützte Validierungsverfahren zurückgreifen kann. Mittels nötiger Sorgfalt bei der Spezifikation, guter Dokumentation sowie allgemein durch den Einsatz einer formalen Notation werden Präzisison und Verständlichkeit behandelt. Diese, und insbesondere die formale Notation, helfen auch bei der Anforderung der geringen Größe und Komplexität. Gute Wiederverwendbarkeit wird durch Modularisierung sowie der Möglichkeit der getrennten Spezifikation von Metamodellen erreicht.

Die verbleibenden Abschnitte beschrieben dann die entwickelte Überführungsmethode. Die Darstellungen orientierten sich dabei an der Struktur des Sicherheitsmodells: die Spezifikation von Metamodell und konkretem Sicherheitsmodell wurden getrennt beschrieben, dabei wurden die einzelnen Bestandteile des Sicherheitsmodells als Grundlage für die Untergliederung der Arbeitsschritte genutzt. Alle Überlegungen wurden zuerst anhand des abstrakten Modells von ZSM-Spezifikatio-

nen aus Unterabschnitt 2.6.3 erläutert; hier wurden Alternativen abgewogen und Entscheidungen getroffen. Schließlich wurden diese abstrakten Überlegungen auf die Spezifikationsnotation Event-B angewandt; dabei wurden Beispiele und Schemata angegeben. Besondere Sprachfeatures von Event-B wurden für Einzellösungen genutzt.

Das Ergebnis ist eine Methode zur Spezifikation allgemeiner Sicherheitsmodelle, die zur Anwendung in Event-B ausgelegt ist, sich jedoch leicht auf andere Notationen zur Spezifikation zustandsbasierter, sequenzieller Systeme übertragen lässt. Die Methode wird praktisch angewandt in Kapitel 4; eine Evaluierung erfolgt in Kapitel 5.

KAPITEL 4

Beispielhafte Anwendung

Das vorhergehende Kapitel erarbeitete eine Methode zur formalen Spezifikation politikbezogener Funktionen einer Trusted Computing Base. Zu Zwecken der Dokumentation, zur Beförderung des Verständnisses sowie zur späteren Evaluation in Kapitel 5 wendet das vorliegende Kapitel diese Spezifikationsmethode auf das in Abschnitt 2.4 vorgestellte Beispielmodell an. Hierzu wird Abschnitt 4.1 zuerst die Spezifikation des dort angegebenen RBAC-Metamodells erarbeiten. Auf dieser Grundlage spezifiziert Abschnitt 4.2 dann das ebenfalls präsentierte Sicherheitsmodell eines Gesundheitsinformationssystems. Während die komplette Spezifikation in Anhang A dokumentiert ist, konzentrieren sich die folgenden Erläuterungen einerseits darauf, ein umfassenes Bild von der Vorgehensweise zu bieten und vollziehen somit die Abläufe im Großen nach, unter Verzicht auf Details. Andererseits soll zur Dokumentation auf verhältnismäßig komplexe und somit potentiell schwerer verständliche Stellen eingegangen werden.

Zur Sicherung der internen Konsistenz ist es entscheidend, wesentliche Beweispflichten zu zeigen. Da die Nutzung der Beweiswerkzeuge keine Neuerung darstellt, geht diese Masterarbeit nicht detailiert auf sie ein, doch Anhang C dokumentiert zum besseren Verständnis den Beweis für Axiom `rev2` in `rbac_rolehandling`.

4.1 RBAC-Metamodell

Der vorliegende Abschnitt spezifiziert das $RBAC_3$-Metamodell aus Unterabschnitt 2.4.1 anhand der in Abschnitt 3.4 erarbeiteten Vorgehensweise. Das Metamodell kann dann als Ausgangspunkt zur Spezifikation konkreter $RBAC_3$-Sicherheitsmodelle genutzt werden.

Grundlage zur Spezifikation sind dabei die mathematisch notierten Modellkomponenten des Metamodells, nämlich Wertebereiche, statische Komponenten, dynamische Komponenten, Primitivoperationen sowie Primitivprädikate. Ziel ist eine Event-B-Spezifikation. Diese wird gemäß Unterabschnitt 3.4.1 auf verschiedene Spezifikationsdokumente aufgeteilt, von welchen jedes einen Kontext (im Sinne von Unterabschnitt 2.6.3) enthält. Für statische Komponenten wird, der Vorgehensweise folgend, der Kontext `rbac_static` erstellt, für dynamische Komponenten der Kontext `rbac_state`. Letzterer erweitert `rbac_static` und importiert somit die Spezifikation der statischen Komponenten. Die Primitivoperationen sollen in verschiedene Kontexte verteilt werden: Unterabschnitt 2.4.1 kategorisierte die Operationen des Metamodells in drei Kategorien, nämlich nutzer-, sitzungs- und rollenzentrierte Primitive. Die zu erstellende Spezifikation wird in

die entsprechenden Kontexte `rbac_userhandling`, `rbac_sessionhandling` sowie `rbac_rolehandling` gegliedert; jeder erweitert `rbac_state` und importiert somit die Spezifikation statischer und dynamischer Komponenten. Für Prädikate ist ein eigener Kontext `rbac_predicate` vorgesehen, welcher ebenso `rbac_state` erweitert. Der das Metamodell zusammenfassende Kontext heißt dann `rbac`; dieser erweitert `rbac_userhandling`, `rbac_sessionhandling`, `rbac_rolehandling` sowie `rbac_predicate` und importiert damit sämtliche Bestandteile der Metamodell-Spezifikation.

Der Rest des Abschnittes dokumentiert die Erstellung der einzelnen Dokumente.

4.1.1 Kontext `rbac_static`

Die statischen Aspekte des Metamodells umfassen die Trägermengen sowie statische Modellkomponenten. Beide Punkte behandelt dieser Unterabschnitt separat.

Trägermengen. Die Trägermengen sind disjunkte Mengen. Alle atomaren Objekte, die in einem aus dem Metamodell hervorgehenden Sicherheitsmodell verwendet werden, müssen sich einer Trägermenge zuordnen lassen. Um die Trägermengen zu bestimmen, wird das gegebene Metamodell analysiert.

Dessen statische Komponenten führen die paarweise disjunkten Mengen O, OP und R für Objekte, Operationen und Rollen auf, die dann die Grundlage für die Definition der übrigen statischen und dynamischen Komponenten bilden. In den dynamischen Komponenten finden sich die paarweise disjunkten Mengen der Nutzer und Sitzungen U und S, welche ebenfalls Grundlage zur Definition der übrigen dynamischen Komponenten sind. Die Spezifikation des Metamodells umfasst keine weiteren Arten von Objekten, sodass sich in der Event-B-Spezifikation (vgl. Unterabschnitt 3.4.2) die Trägermengen `USER`, `ROLE`, `SESSION`, `OBJECT` sowie `OPERATION` ergeben.

Statische Komponenten. Abgesehen von den Trägermengen verfügt das verwendete RBAC_3-Metamodell über drei weitere statische Komponenten, nämlich die Zugriffssteuerungsmatrix, die Rollenhierarchie sowie die Rollenausschlussrelation. Jede wird, entsprechend der vorgestellten Spezifikationsmethodik (vgl. Unterabschnitt 3.4.3), in eine eigene Konstante überführt; entsprechend der Nomenklatur seien diese `M`, `RH` sowie `RE`.

Während die konkreten Werte aller statischen Komponenten erst in konkreten Sicherheitsmodellen bekannt werden, müssen im Metamodell Typisierungsinformationen sowie Eigenschaften der Konstanten vermerkt werden. Die Zugriffssteuerungsmatrix ist im mathematischen Modell notiert als $m : R \times O \to \mathcal{P}(OP)$. Gemäß Vorgehensweise (B3) wird die Abbildung an dieser Stelle als total definiert. Im Rahmen von Beweisen hat sich gezeigt, dass die Matrix stets nur auf endliche Mengen abbilden darf. Es ergeben sich die folgenden Axiome.

`M_t`: $\mathtt{M} \in \mathtt{ROLE} \times \mathtt{OBJECT} \to \mathbb{P}(\mathtt{OPERATION})$

`M1`: $\forall \mathtt{r}, \mathtt{o} \cdot \mathtt{r} \in \mathtt{ROLE} \land \mathtt{o} \in \mathtt{OBJECT} \land (\mathtt{r} \mapsto \mathtt{o} \in \mathtt{dom}(\mathtt{M})) \Rightarrow \mathtt{finite}(\mathtt{M}(\mathtt{r} \mapsto \mathtt{o}))$

Die Rollenhierarchie ist eine Relation $RH \subseteq R \times R$, welche das mathematische Modell als partielle Ordnung definiert und die eine spezielle Semantik für Nutzer- und Rechtezugehörigkeit der Rollen mit sich bringt. Die Eigenschaft der partiellen Ordnung lässt sich bereits an diesem Punkt deklarieren, während die Semantik erst in späteren Modellteilen implementiert werden kann.

$\texttt{RH\_t}\colon\ \texttt{RH} \in \texttt{ROLE} \leftrightarrow \texttt{ROLE}$

$\texttt{RH1}\colon\ \texttt{RH} = \texttt{RH} \fatsemi \texttt{RH}$ > Transitivität, Reflexivität

$\texttt{RH2}\colon\ \forall \texttt{r1}, \texttt{r2} \cdot \texttt{r1} \in \texttt{ROLE} \land \texttt{r2} \in \texttt{ROLE} \land$
$(\texttt{r1} \mapsto \texttt{r2}) \in \texttt{RH} \land (\texttt{r2} \mapsto \texttt{r1}) \in \texttt{RH}$
$\Rightarrow \texttt{r1} = \texttt{r2}$ > Antisymmetrie

Die Rollenausschlussrelation $RE \subseteq R \times R$ ist im mathematischen Modell als irreflexiv und symmetrisch vermerkt. Die entsprechende Deklaration lautet wie folgt.

$\texttt{RE\_t}\colon\ \texttt{RE} \in \texttt{ROLE} \leftrightarrow \texttt{ROLE}$

$\texttt{RE1}\colon\ \texttt{RE} = \texttt{RE}^{-1}$ > Symmetrie

$\texttt{RE2}\colon\ \forall \texttt{r1}, \texttt{r2} \cdot \texttt{r1} \in \texttt{ROLE} \land \texttt{r2} \in \texttt{ROLE} \land (\texttt{r1} \mapsto \texttt{r2}) \in \texttt{RE}$
$\Rightarrow \texttt{r1} \neq \texttt{r2}$ > Irreflexivität

Die gesamte Spezifikation ist Anhang A.1 zu entnehmen.

4.1.2 Kontext `rbac_state`

Die hier relevanten dynamischen Komponenten des Sicherheitsmodells sind (vgl. Unterabschnitt 2.4.2) die Nutzer- bzw. Sitzungsmenge U bzw. S, die Nutzer-Rollen-Relation UA und die Abbildungen von Sitzungen auf Nutzer *user* sowie von Sitzungen auf Rollen *roles*. Der vorliegende Unterabschnitt erarbeitet nacheinander die Konsistenzeigenschaften dieser Komponenten, bildet daraus die Spezifikation des Zustandsraumes und beschreibt schließlich Axiome, die gemäß der Methode in Unterabschnitt 3.4.4 ergänzend zu spezifizieren sind.

Zu jeder Komponente müssen nun deren Eigenschaften erarbeitet werden. Die Nutzer- bzw. Sitzungsmengen sind Teil des Systemzustandes und Teilmengen der Trägermengen aller Nutzer bzw. Sitzungen. Für sie können keine speziellen Eigenschaften notiert werden. Die Nutzer-Rollen-Relation ist eine Relation zwischen Nutzern und Rollen, wobei stets nur im System bekannte Nutzer in Relation stehen sollten, sodass $dom(UA) \subseteq U$ ist.

Die Abbildung *user*, welche jeder Sitzung einen Nutzer zuordnet, bezieht sich stets nur auf im System bekannte Sitzungen und Nutzer und ist weder in-, noch surjektiv. Gleiches gilt für die Abbildung *roles*, welche jeder Sitzung die Menge der aktiven Rollen zuordnet. Dabei wird *user* als partielle Abbildung definiert: der Grund ist, dass Primitivoperationen wie *createSessions* Zwischenzustände erzeugen, in welchen einer Sitzung kein Nutzer zugeordnet ist. Erst im Rahmen des Zustandsraumes unter strengen Konsistenzbedingungen `CONSISTENT_STATE` wird *user* total bezüglich S.

roles ist stets total: zu jeder Sitzung lässt sich eine Menge aktivierter Rollen angeben, anfangs ist dies die leere Menge. Exemplarisch (vgl. Diskussion in Vorgehensweise (B1)) wurde für jedes $s \in S$ die Menge $roles(s)$ als endlich definiert. Zwar ist der praktische Sinn dieser Einschränkung fraglich, da viele Sicherheitsmodelle nur über eine endliche Zahl an Rollen verfügen. Dennoch können anhand der *roles*-Zustandskomponente die Folgen der Endlichkeitseigenschaft gut beobachtet werden.

Ein wichtiger Zusammenhang besteht zwischen den Zustandskomponenten: eine jede Sitzung s, welcher ein Nutzer $user(s)$ zugeordnet ist, kann nur solche Rollen aktivieren, welche der Nutzer gemäß UA besitzt. Dabei ist die Einschränkung auf Sitzungen, denen ein Nutzer zugeordnet ist, unkritisch, da im Zustandsraum unter strengen Konsistenzbedingungen allen Sitzungen ein Nutzer zugeordnet sein muss (siehe oben). Aussagen wie diese erhöhen nicht nur das Vertrauen in die Konsistenz und Korrektheit der Spezifikation, sondern ermöglichen erst den Beweis bestimmter Aussagen (wie etwa `rev2` in `rbac_rolehandling`, siehe Anhang C).

Insgesamt ergibt sich der folgende Zustandsraum.

$$
\begin{aligned}
&\texttt{st: STATE} = \{ \\
&\qquad \texttt{U} \mapsto \texttt{S} \mapsto \texttt{UA} \mapsto \texttt{user} \mapsto \texttt{roles} \mid \\
&\qquad \texttt{U} \in \mathbb{P}(\texttt{USER}) \land \texttt{S} \in \mathbb{P}(\texttt{SESSION}) \land \\
&\qquad \texttt{UA} \in \texttt{USER} \leftrightarrow \texttt{ROLE} \land \texttt{dom}(\texttt{UA}) \subseteq \texttt{U} \land \\
&\qquad \texttt{user} \in \texttt{SESSION} \nrightarrow \texttt{USER} \land \texttt{dom}(\texttt{user}) \subseteq \texttt{S} \land \texttt{ran}(\texttt{user}) \subseteq \texttt{U} \land \\
&\qquad \texttt{roles} \in \texttt{SESSION} \nrightarrow \mathbb{P}(\texttt{ROLE}) \land \texttt{dom}(\texttt{roles}) = \texttt{S} \land \\
&\qquad (\, \forall \texttt{s} \cdot \texttt{s} \in \texttt{S} \Rightarrow \texttt{finite}(\texttt{roles}(\texttt{s}))\,) \\
&\qquad (\, \forall \texttt{s} \cdot \texttt{s} \in \texttt{dom}(\texttt{user}) \Rightarrow \texttt{roles}(\texttt{s}) \subseteq \texttt{UA}[\,\{\texttt{user}(\texttt{s})\}\,]\,) \\
&\quad \} \\
&\texttt{cst: CONSISTENT\_STATE} = \{ \\
&\qquad \texttt{U} \mapsto \texttt{S} \mapsto \texttt{UA} \mapsto \texttt{user} \mapsto \texttt{roles} \mid \\
&\qquad \texttt{U} \mapsto \texttt{S} \mapsto \texttt{UA} \mapsto \texttt{user} \mapsto \texttt{roles} \in \texttt{STATE} \land \\
&\qquad \texttt{dom}(\texttt{user}) = \texttt{S} \\
&\quad \}
\end{aligned}
$$

Zu jeder Zustandskomponente wird nun noch die Projektionsfunktion definiert (vgl. wieder Unterabschnitt 3.4.4), beispielsweise für die *user*-Abbildung:

$$
\begin{aligned}
&\texttt{stu-t: user\_q} \in \texttt{STATE} \rightarrow (\texttt{SESSION} \nrightarrow \texttt{USER}) \\
&\texttt{stu-d: } \forall \texttt{q} \cdot \texttt{q} \in \texttt{STATE} \Rightarrow \texttt{user\_q}(\texttt{q}) = \texttt{prj2}(\texttt{prj1}(\texttt{q}))
\end{aligned}
$$

Dabei ist $\texttt{prj1}(\texttt{q}) = (\,(\texttt{U} \mapsto \texttt{S}) \mapsto \texttt{UA}) \mapsto \texttt{user}$ und $\texttt{prj2}(\texttt{prj1}(\texttt{q}))$ selektiert dann aus dem äußersten Tupel das zweite Element. Schließlich wird über drei weitere Theoreme die Möglichkeit gegeben, die Komponenten des Zustandsraums mit Variablen zu verbinden (vgl. wieder Unterabschnitt 3.4.4). Dies ist zum Beispiel das Theorem `st3`, das besagt, dass solche Variablen existieren:

$$
\begin{aligned}
&\texttt{st3: } \forall \texttt{q} \cdot \texttt{q} \in \texttt{STATE} \Rightarrow (\\
&\qquad \exists \texttt{U}, \texttt{S}, \texttt{UA}, \texttt{user}, \texttt{roles} \cdot \\
&\qquad \texttt{U} \mapsto \texttt{S} \mapsto \texttt{UA} \mapsto \texttt{user} \mapsto \texttt{roles} = \texttt{q} \land
\end{aligned}
$$

```
        U_q(q) = U ∧
        S_q(q) = S ∧
        UA_q(q) = UA ∧
        user_q(q) = user ∧
        roles_q(q) = roles
    )    theorem
```

Die zwei anderen Theoreme `st-i` und `cst-i` projizieren auf solche Variablen die Konsistenzbedingungen (Invarianten) der Zustandsräume. Alle als Theoreme gekennzeichneten Aussagen wurden bewiesen. Die vollständige Spezifikation ist Anhang A.2 zu entnehmen.

4.1.3 Kontext `rbac_userhandling`

Die in diesem Kontext behandelten nutzerzentrischen Primitivoperationen sind *addUsers* (O1) sowie *deleteUsers* (O2). Gemäß der Vorgehensweise in Unterabschnitt 3.3.4 erörtert der vorliegende Unterabschnitt zuerst deren Vorbedingungen, danach die Eigenschaften der Operationen, schließlich eventuelle Seiteneffekte. Im Anschluss werden Auszüge aus der Spezifikation präsentiert.

Wie in der Vorgehensweise erörtert, sind die Vorbedingungen der Primitivoperationen möglichst stark zu formulieren: Nutzer, die hinzugefügt werden, müssen also dem System bis dahin unbekannt sein; zu löschende Nutzer müssen Teil des Zustandes sein. An derselben Stelle wird empfohlen, die Eigenschaften der Operationen zu analysieren: *addUsers* lässt sich mittels *deleteUsers* rückgängig machen und umgekehrt; zudem ist die Anwendung zweier *addUsers*- oder zweier *deleteUsers*-Operationen kommutativ. Aufgrund der starken Vorbedingungen liegt keine Idempotenz vor (d.h. es gilt nicht $\forall\, u \subseteq U \cdot addUsers(addUsers(q, u), u) = addUsers(q, u)$, denn die Vorbedingungen von *addUsers* verlangen, dass $u \cap U = \varnothing$), jedoch bestünde ohne starke Vorbedingungen keine Reversibilität (dann nämlich nicht, wenn eben $u \subseteq U$). Im Übrigen sind die Operationen im Sicherheitsmodell schon so vermerkt, dass Seiteneffekte (vgl. Vorgehensweise (B8)) auf andere Zustandskomponenten berücksichtigt wurden: so bereinigt *deleteUsers* auch *UA* sowie *user* von den gelöschten Nutzern.

Zu jeder Primitivoperation müssen nun verschiedene Axiome eingeführt werden, zunächst zur Deklaration des Typs der Operation. Für die *addUsers*-Operation leistet dies das Axiom `add_t`. Die Deklaration erfolgt dabei gemäß Vorgehensweise (B3); die Operation wird als total definiert[1], um dort, wo die Operation eingesetzt wird, das Entstehen unnötiger Beweispflichten zur Wohldefiniertheit zu verhindern[2].

`add_t: addUsers` $\in \mathtt{STATE} \times \mathbb{P}(\mathtt{USER}) \to \mathtt{STATE}$

Die Definition der Semantik leistet dann Axiom `add_d`:

[1] Operator $\to$, im Gegensatz zum Operator $\nrightarrow$ für partielle Funktionen

[2] In der vorgestellten Methode werden die Primitivoperationen stets anhand ihrer semantischen Definition durch die `add_d`-Axiome genutzt; diese schränken die Anwendbarkeit der Operationen dann effektiv ein und machen die Beweispflichten zur Wohldefiniertheit unnötig.

$$\begin{aligned}
&\texttt{add_d}\colon\ \forall\, \texttt{u}, \texttt{q} \cdot \\
&\qquad \texttt{q} \in \texttt{STATE} \wedge \\
&\qquad \texttt{u} \in \mathbb{P}(\texttt{USER}) \wedge \texttt{u} \cap \texttt{U_q}(\texttt{q}) = \varnothing \\
&\quad \Rightarrow \texttt{addUsers}(\texttt{q} \mapsto \texttt{u}) = (\texttt{U_q}(\texttt{q}) \cup \texttt{u}) \mapsto \texttt{S_q}(\texttt{q}) \mapsto \texttt{UA_q}(\texttt{q}) \mapsto \\
&\qquad \texttt{user_q}(\texttt{q}) \mapsto \texttt{roles_q}(\texttt{q})
\end{aligned}$$

Die starken Vorbedingungen sind erkennbar, insbesondere der Umstand, dass hinzuzufügende Nutzer dem System unbekannt sein müssen. Ein zusätzliches Theorem `add_v` stellt schließlich sicher, dass die Konsistenzeigenschaften des Zustands erfüllt werden (vgl. Unterabschnitt 3.4.5).

Die Eigenschaft der Kommutativität der *addUsers*-Operation wird in Theorem `com1` festgehalten:

$$\begin{aligned}
&\texttt{com1}\colon\ \forall\, \texttt{q}, \texttt{u1}, \texttt{u2} \cdot \\
&\qquad \texttt{q} \in \texttt{STATE} \wedge \\
&\qquad \texttt{u1} \in \mathbb{P}(\texttt{USER}) \wedge \texttt{u1} \cap \texttt{U_q}(\texttt{q}) = \varnothing \wedge \\
&\qquad \texttt{u2} \in \mathbb{P}(\texttt{USER}) \wedge \texttt{u2} \cap \texttt{U_q}(\texttt{q}) = \varnothing \wedge \\
&\qquad \texttt{u1} \cap \texttt{u2} = \varnothing \\
&\quad \Rightarrow \texttt{addUsers}(\texttt{addUsers}(\texttt{q} \mapsto \texttt{u1}) \mapsto \texttt{u2}) = \\
&\qquad \texttt{addUsers}(\texttt{addUsers}(\texttt{q} \mapsto \texttt{u2}) \mapsto \texttt{u1}) \qquad \textbf{theorem}
\end{aligned}$$

Die Spezifikation der *deleteUsers*-Operation erfolgt dann entsprechend, es entstehen die Axiome `del_t`, `del_d`, `del_v` und `com2`. Die Eigenschaft der Reversibilität verknüpft dann beide Operationen dies beschreibt das Theorem `rev`:

$$\begin{aligned}
&\texttt{rev}\colon\ \forall\, \texttt{u}, \texttt{q} \cdot \\
&\qquad \texttt{q} \in \texttt{STATE} \wedge \\
&\qquad \texttt{u} \in \mathbb{P}(\texttt{USER}) \wedge \texttt{u} \cap \texttt{U_q}(\texttt{q}) = \varnothing \\
&\quad \Rightarrow \texttt{deleteUsers}(\texttt{addUsers}(\texttt{q} \mapsto \texttt{u}) \mapsto \texttt{u}) = \texttt{q} \qquad \textbf{theorem}
\end{aligned}$$

Dieses Theorem muss alle Vorbedingungen der zuerst ausgeführten Operation `addUsers` erfüllen; die Vorbedingungen der zuletzt ausgeführten Operation `deleteUsers` müssen dann durch `addUsers` erfüllt sein.

Im Übrigen könnten noch Theoreme formuliert werden, welche die Effekte der beiden Primitivoperationen mithilfe der Projektionsfunktionen ausdrücken. Dies bringt im vorliegenden Fall keinen übermäßigen Nutzen und vergrößert die Spezifikation unnötig, kann jedoch Beweise erheblich vereinfachen. Das entsprechende Theorem zu *addUsers*, obschon nicht in der Spezifikation enthalten, wäre:

$$\begin{aligned}
&\texttt{add1}\colon\ \forall\, \texttt{q}, \texttt{u} \cdot \\
&\qquad \texttt{q} \in \texttt{STATE} \wedge \texttt{u} \subseteq \texttt{USER} \wedge \texttt{u} \cap \texttt{U_q}(\texttt{q}) = \varnothing \\
&\quad \Rightarrow \texttt{U_q}(\texttt{q}) \cup \texttt{u} = \texttt{U_q}(\texttt{addUsers}(\texttt{q} \mapsto \texttt{u})) \wedge \\
&\qquad \texttt{S_q}(\texttt{q}) = \texttt{S_q}(\texttt{addUsers}(\texttt{q} \mapsto \texttt{u})) \wedge \\
&\qquad \texttt{UA_q}(\texttt{q}) = \texttt{UA_q}(\texttt{addUsers}(\texttt{q} \mapsto \texttt{u})) \wedge \\
&\qquad \texttt{user_q}(\texttt{q}) = \texttt{user_q}(\texttt{addUsers}(\texttt{q} \mapsto \texttt{u})) \wedge \\
&\qquad \texttt{roles_q}(\texttt{q}) = \texttt{roles_q}(\texttt{addUsers}(\texttt{q} \mapsto \texttt{u})) \qquad \textbf{theorem}
\end{aligned}$$

Die vollständige Spezifikation kann Anhang A.3 entnommen werden.

4.1.4 Kontext `rbac_sessionhandling`

Primitivoperationen zur Sitzungsverwaltung sind einerseits die Operationen *createSessions* und *destroySessions* zum Anlegen und Löschen einer Sitzung, andererseits die Operationen *mapUserSessions* und *unmapUserSessions* zur Zuordnung von Nutzern zu Sitzungen (siehe (O3) bis (O6)). Der Vorgehensweise in Unterabschnitt 3.3.4 folgend geht der vorliegende Unterabschnitt dabei nacheinander auf Vorbedingungen, Seiteneffekte und Eigenschaften ein und präsentiert dabei Auszüge aus der sich ergebenden Spezifikation.

Wie oben sind die Vorbedingungen möglichst stark zu formulieren. Dabei erfordert die Endlichkeitseinschränkung für die *roles*-Komponente, dass einige Parameter ebenfalls als endlich definiert werden müssen. Zudem muss bei der Zuordnung von Sitzungen auf Nutzer berücksichtigt werden, dass zu jeder Sitzung die Menge der aktivierten Rollen eine Teilmenge derjenigen Rollen sein muss, die dem Nutzer zugeordnet sind. Eine entsprechende Vorbedingung stellt dies sicher. Das Aufheben der Sitzungs-Nutzer-Zuordnung durch *unmapUserSessions* erfordert diesbezüglich keine besonderen Vorbedingungen, da die Einschränkung der aktivierten Rollen nur für solche Sitzungen gilt, denen ein Nutzer zugeordnet ist (vgl. Unterabschnitt 4.1.2).

Das gegebene Sicherheitsmodell notiert die Operationen bereits so, dass Seiteneffekte (vgl. Vorgehensweise (B8)) auf andere Zustandskomponenten berücksichtigt werden. Dies betrifft insbesondere *user*, dessen Definitionsbereich höchstens die aktiven Sitzungen umfassen darf, sowie *roles*, dessen Definitionsbereich genau die aktiven Sitzungen umfassen muss.

Konkret muss die Spezifikation von *createSessions* etwa berücksichtigen, dass die erzeugte Sitzung mit einer leeren Menge aktiver Rollen erstellt werden muss. Zudem wirkt sich die Endlichkeitseinschränkung für *roles* auf die Vorbedingungen aus. Die Spezifikation der Operation lautet entsprechend wie folgt.

$$
\begin{aligned}
&\texttt{crt\_d}\colon \forall\, \texttt{q}, \texttt{s} \cdot \\
&\qquad \texttt{q} \in \texttt{STATE} \land \\
&\qquad \texttt{s} \in \mathbb{P}(\texttt{SESSION}) \land \texttt{finite}(\texttt{s}) \land \texttt{s} \cap \texttt{S\_q}(\texttt{q}) = \varnothing \\
&\quad \Rightarrow \texttt{createSessions}(\texttt{q} \mapsto \texttt{s}) = \texttt{U\_q}(\texttt{q}) \mapsto (\texttt{S\_q}(\texttt{q}) \cup \texttt{s}) \mapsto \texttt{UA\_q}(\texttt{q}) \mapsto \\
&\qquad \texttt{user\_q}(\texttt{q}) \mapsto (\texttt{roles\_q}(\texttt{q}) \cup (\texttt{s} \times \{\varnothing\}))
\end{aligned}
$$

Die Einschränkung der aktivierbaren Rollen einer Sitzung auf diejenigen Rollen, die dem Nutzer tatsächlich zugeordnet sind, hat auch Auwirkungen auf *mapUserSessions*. Nach Ausführen der Primitivoperation darf jede der Sitzungen, denen ein Nutzer zugeordnet wurde, also nur solche Rollen aktiviert haben, die der zugeordnete Nutzer inne hat. Dies bedeutet entweder, dass *mapUserSessions* aktivierte Rollen deaktivieren muss, oder, dass die Vorbedingung der Primitivoperation die vorherige Deaktivierung aller solcher Rollen verlangen muss. Da implizites Verhalten zu Missverständnissen und Fehlern führen kann, verwendet die vorliegende Spezifikation die zweite Variante; sie lautet:

$$
\begin{aligned}
\texttt{mp\_d}\colon\ & \forall\, \texttt{q}, \texttt{su} \cdot \\
& \quad \texttt{q} \in \texttt{STATE} \wedge \\
& \quad \texttt{su} \in \texttt{S\_q}(\texttt{q}) \setminus \texttt{dom}(\texttt{user\_q}(\texttt{q})) \nrightarrow \texttt{U\_q}(\texttt{q}) \\
& \quad (\forall \texttt{s} \cdot \texttt{s} \in \texttt{dom}(\texttt{su}) \Rightarrow \texttt{roles\_q}(\texttt{q})(\texttt{s}) \subseteq \texttt{UA\_q}(\texttt{q})[\,\{\texttt{su}(\texttt{s})\}\,]) \\
& \Rightarrow \texttt{mapUserSessions}(\texttt{q} \mapsto \texttt{su}) = \texttt{U\_q}(\texttt{q}) \mapsto \texttt{S\_q}(\texttt{q}) \mapsto \texttt{UA\_q}(\texttt{q}) \mapsto \\
& \quad (\texttt{user\_q}(\texttt{q}) \cup \texttt{su}) \mapsto \texttt{roles\_q}(\texttt{q})
\end{aligned}
$$

Bezüglich der Eigenschaften lässt sich feststellen, dass der Effekt von *createSessions* mittels *destroySessions* umgekehrt werden kann, ebenso der Effekt von *mapUserSessions* mit *unmapUserSessions*. Für disjunkte Sitzungsmengen gilt auch die Kommutativität der Operationen; deren Spezifikation bringt an dieser Stelle jedoch wenig Gewinn (vgl. Diskussion in Unterabschnitt 3.3.4).

Die entstandene Spezifikation ist vergleichbar zu `rbac_userhandling` (siehe oben) und findet sich in Anhang A.4.

4.1.5 Kontext `rbac_rolehandling`

Der vorliegende Unterabschnitt erarbeitet die Spezifikation zu rollenzentrischen Primitivoperationen; dies sind einerseits *assignRolesToUsers* und *revokeRolesFromUsers*, sowie andererseits *activateRoles* und *deactiateRoles* (siehe (O7) bis (O10)). Die Darstellungen folgen dabei wieder der Vorgehensweise in Unterabschnitt 3.3.4, betrachtet werden nacheinander Vorbedingungen, Seiteneffekte und Eigenschaften der Primitivoperationen, schließlich werden Auszüge aus der Spezifikation präsentiert.

Wie oben argumentiert, sind die Vorbedingungen der Primitivoperationen möglichst stark zu formulieren. Dies bedeutet einerseits, dass die Operation einen Effekt haben muss: die Rollen für *assignRolesToUsers* (*revokeRolesFromUsers*) dürfen noch nicht zugeordnet worden sein (müssen schon zugeordnet sein); weiterhin dürfen für *activateRoles* (*deactivateRoles*) die zu aktivierenden Rollen nicht in den entsprechenden Sitzungen aktiviert sein (müssen in den entsprechenden Sitzungen schon aktiviert sein). Andererseits ist die Konsistenz des Zustandes zu gewährleisten: so darf *revokeRolesFromUsers* keine Rolle entziehen, die im Vorzustand in einer Sitzung aktiviert ist und *activateRoles* darf nur solche Rollen aktivieren, welche dem Nutzer der Sitzung zugeordnet wurden. Dies zwingt die Entwickler der Spezifikation eines Sicherheitsmodells, dafür Sorge zu tragen, dass die Rollenzuordnung korrekt verwaltet wird. Zudem wird etwa das Deaktivieren von Rollen in der Spezifikation des Autorisierungsschemas explizit sichtbar.

Die hier betrachteten Primitivoperationen weisen keine Seiteneffekte (vgl. Vorgehensweise (B8)) auf. Zu den Eigenschaften kann einerseits die Kommutativität der verschiedenen Primitivoperationen beobachtet werden, welche hier jedoch nicht spezifiziert wird (vgl. Diskussion in Unterabschnitt 3.3.4). Andererseits sind *assignRolesToUsers* und *revokeRolesFromUsers* sowie *activateRoles* und *deactivateRoles* jeweils reversibel.

Die so entstandene Spezifikation ist insbesondere aufgrund der Teils komplexen Vorbedingungen interessant. So ist *revokeRolesFromUsers* mit der Vorbedingung, dass keine der zu entziehenden Rollen aktiviert sein darf, wie folgt notiert.

$$
\begin{aligned}
\texttt{rur\_d}\colon\ & \forall\, \texttt{q}, \texttt{ua} \cdot \\
& \quad \texttt{q} \in \texttt{STATE} \wedge \\
& \quad \texttt{ua} \subseteq \texttt{UA\_q}(\texttt{q}) \wedge \\
& \quad (\forall\, \texttt{u} \cdot \texttt{u} \in \texttt{dom}(\texttt{ua}) \Rightarrow \texttt{ua}[\,\{\texttt{u}\}\,] \cap \\
& \qquad (\bigcup \texttt{r} \cdot \texttt{r} \in \texttt{roles\_q}(\texttt{q})[\texttt{user\_q}(\texttt{q})^{-1}[\,\{\texttt{u}\}\,]] \mid \texttt{r}\,) = \varnothing \\
& \quad) \\
& \Rightarrow \texttt{revokeRolesFromUsers}(\texttt{q} \mapsto \texttt{ua}) = \texttt{U\_q}(\texttt{q}) \mapsto \texttt{S\_q}(\texttt{q}) \mapsto \\
& \quad (\texttt{UA\_q}(\texttt{q}) \setminus \texttt{ua}) \mapsto \texttt{user\_q}(\texttt{q}) \mapsto \texttt{roles\_q}(\texttt{q})
\end{aligned}
$$

Dabei ergibt sich $\texttt{user\_q}(\texttt{q})^{-1}[\,\{\texttt{u}\}\,]$ zu der Menge aller Sitzungen, welche u zugeordnet sind; jedes r ist dann die Menge aller Rollenmengen dieser einzelnen Sitzungen. $(\bigcup \texttt{r} \cdot \texttt{r} \in \texttt{roles\_q}(\texttt{q})[\texttt{user\_q}(\texttt{q})^{-1}[\,\{\texttt{u}\}\,]] \mid \texttt{r}\,)$ vereinigt die einzelnen r und ist somit die Menge aller Rollen, welche in einer u zugeordneten Sitzung aktiviert sind. Diese Vereinigung muss disjunkt von $\texttt{ua}[\,\{\texttt{u}\}\,]$ sein, also von denjenigen Rollen, welche u entzogen werden sollen – dann ist die Vorbedingung erfüllt, und es wird keine aktive Rolle entzogen.

Bei *activateRoles* hingegen muss vorausgesetzt werden, dass keine der Rollen bereits aktiviert ist; zudem verlangt die Endlichkeitseinschränkung für die *roles*-Komponente auch die entsprechende Einschränkung des Parameters. Es ergibt sich die folgende Spezifikation:

$$
\begin{aligned}
\texttt{acr\_d}\colon\ & \forall\, \texttt{q}, \texttt{sr} \cdot \\
& \quad \texttt{q} \in \texttt{STATE} \wedge \\
& \quad \texttt{sr} \subseteq \texttt{user\_q}(\texttt{q}); \texttt{UA\_q}(\texttt{q}) \wedge \texttt{finite}(\texttt{sr}) \wedge \\
& \quad (\forall\, \texttt{s} \cdot \texttt{s} \in \texttt{dom}(\texttt{user\_q}(\texttt{q})) \Rightarrow \texttt{roles\_q}(\texttt{q})(\texttt{s}) \cap \texttt{sr}[\{\texttt{s}\}] = \varnothing) \\
& \Rightarrow \texttt{activateRoles}(\texttt{q} \mapsto \texttt{sr}) = \texttt{U\_q}(\texttt{q}) \mapsto \texttt{S\_q}(\texttt{q}) \mapsto \texttt{UA\_q}(\texttt{q}) \mapsto \\
& \quad \texttt{user\_q}(\texttt{q}) \mapsto \\
& \quad \{\texttt{s} \mapsto \texttt{r} \mid \\
& \qquad \texttt{s} \in \texttt{dom}(\texttt{roles\_q}(\texttt{q})) \wedge \\
& \qquad \texttt{r} = (\texttt{roles\_q}(\texttt{q})(\texttt{s}) \cup \texttt{sr}[\,\{\texttt{s}\} \cap \texttt{dom}(\texttt{user\_q}(\texttt{q}))\,]) \\
& \quad \}
\end{aligned}
$$

Die letzte Zustandskomponente wird dabei so verändert, dass zu jeder Sitzung s die Menge der in dieser Sitzung aktivierten Rollen $\texttt{roles\_q}(\texttt{q})(\texttt{s})$ um $\texttt{sr}[\,\{\texttt{s}\} \cap \texttt{dom}(\texttt{user\_q}(\texttt{q}))\,]$ ergänzt wird. Dies ist für Sitzungen $\texttt{s} \notin \texttt{dom}(\texttt{sr})$ die leere Menge und sonst genau $\texttt{sr}[\{\texttt{s}\}]$.

Die vollständige Spezifikation ist in Anhang A.5 zu finden.

4.1.6 Kontext `rbac_predicate`

Die fünf in Unterabschnitt 2.4.2 eingeführten Primitivprädikate $access_{SR}$, $access_{SM}$, $access_{UR}$, $access_{UM}$ und *sod* (siehe (O11) bis (O15)) werden gemäß der Vorgehensweise in Unterabschnitt 3.4.6 in einem Kontext zusammengefasst. Es handelt sich um boolsche Funktionen, deren Ergebnis abhängig von Parametern und Modellzustand ist. Die Vorbedingungen von Primitivprädikaten fallen im Vergleich zu Primitivoperationen äußerst einfach aus, da sie meist auf jedem Zustand ausgewertet werden können und somit lediglich typisierende Vorbedingungen mit sich bringen.

Besondere Beweispflichten sind nicht zu beachten. Ein Primitivprädikat kann in Abhängigkeit von einem anderen formuliert werden, sodass die Definition von den aufeinander aufbauenden $access_{UR}$ und $access_{UM}$ beispielsweise wie folgt aussehen:

$$
\begin{aligned}
&\texttt{aUR\_d}\colon\ \forall\, \texttt{q}, \texttt{u}, \texttt{r} \cdot \\
&\qquad \texttt{q} \in \texttt{STATE} \land \\
&\qquad \texttt{u} \in \texttt{USER} \land \\
&\qquad \texttt{r} \in \texttt{ROLE} \\
&\quad \Rightarrow \texttt{access\_UR}(\texttt{q} \mapsto \texttt{u} \mapsto \texttt{r}) = \texttt{bool}(\\
&\qquad (\exists\, \texttt{r1} \cdot \texttt{r1} \in \texttt{ROLE} \land \\
&\qquad\quad (\texttt{r1} \mapsto \texttt{r}) \in \texttt{RH} \land \\
&\qquad\quad (\texttt{u} \mapsto \texttt{r1}) \in \texttt{UA\_q}(\texttt{q}) \\
&\qquad) \\
&\quad)
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{aUM\_d}\colon\ \forall\, \texttt{q}, \texttt{u}, \texttt{o}, \texttt{op} \cdot \\
&\qquad \texttt{q} \in \texttt{STATE} \land \\
&\qquad \texttt{u} \in \texttt{USER} \land \\
&\qquad \texttt{o} \in \texttt{OBJECT} \land \texttt{op} \in \texttt{OPERATION} \\
&\quad \Rightarrow \texttt{access\_UM}(\texttt{q} \mapsto \texttt{u} \mapsto \texttt{o} \mapsto \texttt{op}) = \texttt{bool}(\\
&\qquad (\exists\, \texttt{r} \cdot \texttt{r} \in \texttt{ROLE} \land (\texttt{r} \mapsto \texttt{o}) \in \texttt{dom}(\texttt{M}) \land \\
&\qquad\quad \texttt{op} \in \texttt{M}(\texttt{r} \mapsto \texttt{o}) \land \\
&\qquad\quad \texttt{access\_UR}(\texttt{q} \mapsto \texttt{u} \mapsto \texttt{r}) = \texttt{TRUE} \\
&\qquad) \\
&\quad)
\end{aligned}
$$

$\texttt{access\_UR}(\texttt{q} \mapsto \texttt{u} \mapsto \texttt{r})$ ist dabei genau dann wahr, wenn der Nutzer `u` in Zustand `q` eine `r` dominierende Rolle `r1` besitzt: `u` hat dann auch die Rechte aus `r`. Das Prädikat `access_UM` baut darauf auf. $\texttt{access\_UM}(\texttt{q} \mapsto \texttt{u} \mapsto \texttt{o} \mapsto \texttt{op})$ ist genau dann wahr, wenn im gegebenen Zustand eine Rolle `r` mit der gewünschten Berechtigung so existiert, dass $\texttt{access\_UR}(\texttt{q} \mapsto \texttt{u} \mapsto \texttt{r})$ wahr ist.

Da die Zusammenhänge zwischen solchen Prädikaten sehr einfach sind, bringt ein zusammenführendes Theorem wenig Nutzen. Die vollständige Spezifikation findet sich in Anhang A.6.

4.1.7 Kontext `rbac`

Der `rbac`-Kontext fasst alle Kontexte des Metamodells zusammen. Hierzu erweitert er, wie in der Vorgehensweise in Unterabschnitt 3.4.7 beschrieben, alle `rbac_state` erweiternden Kontexte. Die entsprechende Spezifikation findet sich in Anhang A.7.

4.2 Sicherheitsmodell

Der Unterabschnitt 2.4.2 stellte das Sicherheitsmodell für ein Gesundheitsinformationssystem im $RBAC_3$-Metamodell vor. Mittels der in Abschnitt 3.6 erarbeiteten Vorgehensweise soll der vorliegende Abschnitt dieses Modell formal spezifizieren.

Ausgangspunkt sind die mathematisch notierten Eigenschaften des Sicherheitsmodells, nämlich die Werte der statischen Komponenten, der Initialzustand sowie die Definition des Autorisierungsschemas. Es soll eine Event-B-Spezifikation entstehen, die gemäß Unterabschnitt 3.6.1 zumindest in den Kontext `healthcare_context` sowie die Zustandsmaschine `healthcare` zu gliedern ist. Der Argumentation aus Unterabschnitt 3.6.6 folgend, werden im Rahmen der Spezifikation des Autorisierungsschemas Methoden zur Wiederverwendung eingesetzt, welche eine zweite Maschine `healthcare_generic` erforderlich macht. Im Folgenden widmet sich jeder Unterabschnitt der Erstellung je eines Spezifikationsdokumentes.

4.2.1 Kontext `healthcare_context`

Gemäß der Vorgehensweise in Abschnitt 3.6 sind in diesem Kontext die bekannten Elemente der Trägermengen aufzuschlüsseln; außerdem ist den statischen Komponenten ihr Initialwert zuzuweisen. Die Elemente sowie die Initialisierung lassen sich der Beschreibung des Beispielmodells in Unterabschnitt 2.4.2 entnehmen. So ergeben sich beispielsweise bezüglich der Rollen die folgenden Spezifikationen:

$$
\begin{aligned}
&\texttt{axm1: partition}(\texttt{ROLE},\\
&\qquad \{\texttt{Employee}\}, \{\texttt{Manager}\}, \{\texttt{Doctor}\}, \{\texttt{Nurse}\}, \{\texttt{Receptionist}\},\\
&\qquad \{\texttt{Patient}\}, \{\texttt{MedicalManager}\}, \{\texttt{MedicalTeam}\},\\
&\qquad \{\texttt{ReferredDoctor}\}, \{\texttt{UserAdmin}\}\\
&\quad)\\
&\vdots\\
&\texttt{axm5: RH\_base} = \{\\
&\qquad \texttt{Doctor} \mapsto \texttt{Nurse},\\
&\qquad \texttt{Nurse} \mapsto \texttt{Employee},\\
&\qquad \texttt{Manager} \mapsto \texttt{MedicalManager},\\
&\qquad \texttt{MedicalManager} \mapsto \texttt{Receptionist},\\
&\qquad \texttt{Receptionist} \mapsto \texttt{Employee}\\
&\quad \} \cup \texttt{id}\\
&\texttt{axm6: RH} = \texttt{RH\_base} \fatsemi \texttt{RH\_base} \fatsemi \texttt{RH\_base}
\end{aligned}
$$

Dabei wird der Inhalt der eigentlichen Rollenhierarchie `RH` nicht direkt angegeben, denn diese Relation ist transitiv und eine Aufzählung aller ihrer Elemente ist unnötig groß, komplex, fehleranfällig und schwer zu ändern. Stattdessen wird `RH` aus der Menge `RH_base`, mit $\texttt{RH\_base}^0 \in \texttt{RH\_base}$, durch Bildung des reflexiv-transitiven Abschlusses $\texttt{RH\_base}^* = \texttt{RH\_base}^0 \cup \texttt{RH\_base}^1 \cup \texttt{RH\_base}^2 \cup \texttt{RH\_base}^3 \cup \cdots = \texttt{RH\_base}^3 = \texttt{RH\_base} \fatsemi \texttt{RH\_base} \fatsemi \texttt{RH\_base}$ erzeugt.

Vergleichbar wird mit der Relation zum Rollenausschluss `RE` verfahren. Hier wird die Symmetrie durch Vereinigung mit dem Inversen erzielt:

$$\texttt{axm9: RE} = \texttt{RE\_base} \cup \texttt{RE\_base}^{-1}$$

Die Definition der Zugriffssteuerungsmatrix `M` entsteht aus der Tabelle 2.1 und ist verhältnismäßig groß. Dies aber lässt sich mit den Mitteln aktueller Werkzeuge

der Event-B-Entwicklungsumgebung nicht vermeiden; eine mögliche Lösung wäre die Entwicklung spezieller Werkzeuge für den vorliegenden Anwendungsfall (etwa mit einer spezialisierten Spezifikationsnotation, siehe Unterabschnitt 5.2.2).

Die vollständige Spezifikation ist in Anhang A.8 verfügbar.

4.2.2 Maschine `healthcare_generic`

Der Initialzustand und das Autorisierungsschema des Beispielmodells (siehe Unterabschnitt 2.4.2) werden in einer Zustandsmaschine spezifiziert (vgl. Vorgehensweise in Abschnitt 3.6). Wie oben erwähnt, wird dabei auf eine Reihe ähnlicher Kommandos eine Technik der Wiederverwendung angewendet (vgl. „Wiederverwendung“ in Unterabschnitt 3.6.6), bei der die Kommandos zuerst verallgemeinert spezifiziert und anschließend in einem Verfeinerungsschritt konkretisiert werden. Die nun zu erstellende Maschine, welche die verallgemeinerte Spezifikation enthält, heißt daher `healthcare_generic`. Der Rest des Unterabschnitt beschreibt zuerst die Deklaration des Zustandes, nachfolgend die Spezifikation des Initialzustandes, und zuletzt argumentiert er, wie das Autorisierungsschema zu übertragen ist.

Die Maschine muss den `healthcare_context` importieren und deren Zustandsraum als Zustandsvariable übernehmen.

MACHINE healthcare_generic

SEES healthcare_context

VARIABLES

state

INVARIANTS

inv1: state $\in$ CONSISTENT_STATE

inv2: state $\in$ STATE theorem

$\vdots$

Das Theorem `inv2` ist dabei nur zu Anschauungszwecken notiert, es folgt direkt aus $\texttt{STATE} \subseteq \texttt{CONSISTENT\_STATE}$. Nun muss der Initialzustand angegeben werden. Die Vorgehensweise in Unterabschnitt 3.5.4 erläutert, dass ein Initialzustand einen funktionsfähigen Zustand minimaler Größe darstellt. Dieser besteht im vorliegenden Falle aus einem Nutzer mit Administratorrechten, welcher im Modell in der Lage ist, sich anzumelden, weitere Nutzer anzulegen und dann mit Rollen zu versehen. Die entsprechende Spezifikation lautet wie folgt.

$\vdots$

EVENTS

INITIALISATION $\widehat{=}$

begin

act1: state $:= \{\texttt{u1}\} \mapsto \varnothing \mapsto \{\texttt{u1} \mapsto \texttt{UserAdmin}\} \mapsto \varnothing \mapsto \varnothing$

end

$\vdots$

END

Bei der Spezifikation des Autorisierungsschemas (vgl. Vorgehen in Unterabschnitt 3.6.6) ist pro Kommando ein Event-B-Ereignis zu erstellen; dessen Typen und Bedingungen werden dabei als Guards (also Vorbedingungen) formuliert. Die Guards müssen dabei die Vorbedingungen der Primitivoperationen erfüllen und sollten gemäß der Argumentation in Vorgehensweise (B7) zuerst stark formuliert, und dann erst in Verfeinerungen abgeschwächt werden.

Besondere Aufmerksamkeit benötigt das Kommando *activateRole*, welches in einer Sitzung eine Rolle aktivieren kann, sofern diese dem Sitzungsnutzer zugeordnet wurde. Nun kann es zur Durchsetzung des *Principle of Least Privilege* [SS75] sinnvoll sein, nicht die Rolle mit der größten Rechtemenge, sondern lediglich eine von dieser dominierte Rolle zu aktivieren. Im Beispielmodell könnte etwa ein Arzt je nach benötigten Rechten anstelle der Rolle *Doctor* lediglich die Rollen *Nurse* oder *Employee* einsetzen. Dies ist in der vorliegenden Spezifikation nicht möglich; die Ursache hierfür ist die Spezifikation der Primitivoperation `activateRoles` (siehe Unterabschnitt 4.1.5). Um auch dominierte Rollen als Eingabe zu erlauben müsste sie die Rollenhierarchie einbeziehen. An dieser Stelle wird der Effekt starker Vorbedingungen sichtbar: sie schränken zwar einerseits die Anwendbarkeit der Operationen ein, zwingen aber dadurch den Spezifizierenden, sich intensiv der exakten Semantik jener Operationen auseinanderzusetzen.

Die Kommandos *assignRole*, *revokeRole*, *assignReferredDoctorRole*, *revokeReferredDoctorRole*, *assignPatientRole*, *revokePatientRole*, *assignMedicalTeamRole* und *revokeMedicalTeamRole* dienen alle dazu, einem Nutzer eine Rolle zuzuweisen oder zu entziehen; einzig die Rolle sowie der Bedingungsteil variieren. Um also die Zahl der zu zeigenden Beweispflichten zu veringern und die Größe und Komplexität der Spezifikation zu senken, ist es sinnvoll, alle konkreten Kommandos von verallgemeinerten Versionen abzuleiten. Nach den Empfehlungen zur Wiederverwendung in Unterabschnitt 3.6.6 kann dabei ein abstraktes Ereignis geschaffen werden, welches in Verfeinerungen der Maschine (den konkreten Maschinen) dann selbst verfeinert wird. Dabei können Parameter hinzugefügt oder entfernt werden, die Guards können gestärkt oder Aktionen modifiziert werden. Im vorliegenden Fall kann also ein Ereignis `assignGenericRole` spezifiziert werden, welches einem beliebigen Nutzer `u` ein beliebiges Recht `r` zuordnet. Der Rollenausschluss muss dabei stets beachtet werden, sodass das *sod*-Prädikat fest als Bedingung vorgesehen wird. Die Spezifikation sieht dann wie folgt aus.

```
assignGenericRole ≙
   any
      u     > Nutzer
      r     > dem Nutzer zuzuweisende Rolle
   where
      typ1: u ∈ U_q(state)
      typ2: r ∈ ROLE \ UA_q(state)[{u}]
      cnd1: sod(state ↦ u ↦ r) = TRUE
```

```
    then
       act1: state := assignRolesToUsers( state ↦ {u ↦ r} )
    end
```

Sämtliche oben genannten *assign*-Kommandos werden dann als solche Ereignisse spezifiziert, die `assignGenericRole` erweitern: ihre Guards werden durch zusätzliche Bedingungen zur Zugriffssteuerung gestärkt und der Parameter `r` wird (außer im Falle von *assignRole*) durch einen konkreten Wert ersetzt. Die konkrete Spezifikation dieser Kommandos ist im folgenden Unterabschnitt 4.2.3 dokumentiert. Das Vorgehen für die Kommandos zum Rollenentzug ist entsprechend.

Im Rahmen der Spezifikation von `revokeGenericRole` besteht allerdings das Problem, dass einem Nutzer prinzipiell nur solche Rollen entzogen werden können, welche in keiner seiner aktiven Sitzungen aktiviert sind. Dabei ist die Variante, die vorherige Deaktivierung aller zu entziehenden Rollen als Vorbedingung zu verlangen, sowohl unnötig als auch unerwünscht: unnötig, da durch einen automatischen Rollenentzug keine Sicherheitseigenschaften im Sinne von Vorgehensweise (B7) gefährdet sind, und unerwünscht, da Vorbedingungen an Schnittstellen wie dem Autorisierungsschema möglichst schwach zu formulieren sind. Das Kommando wird hier daher so spezifiziert, dass noch aktivierte Rollen automatisch deaktiviert werden.

```
revokeGenericRole ≙
   any
      u     > Nutzer
      r     > dem Nutzer zu entziehende Rolle
   where
      typ1: u ∈ U_q(state)
      typ2: r ∈ UA_q(state)[ {u} ]
   then
      act1: state := revokeRolesFromUsers(
              deactivateRoles(
                  state ↦
                  { s0 · s0 ∈ user_q(state)⁻¹[ {u} ] ∧
                      r ∈ roles_q(state)(s0) | s0 ↦ r}
              ) ↦
              {u ↦ r}
          )
   end
```

Die vollständige Spezifikation ist Anhang A.9 zu entnehmen.

4.2.3 Maschine `healthcare`

Das Dokument `healthcare_generic` (vgl. vorhergehender Unterabschnitt) spezifizierte lediglich eine verallgemeinerte Form der Kommandos *assignRole*, *revokeRole*, *assignReferredDoctorRole*, *revokeReferredDoctorRole*, *assignPatientRole*,

revokePatientRole, *assignMedicalTeamRole* sowie *revokeMedicalTeamRole*. Diese müssen nun in der `healthcare`-Maschine, einer Verfeinerung von `healthcare_generic`, notiert werden. Die Maschine wird dabei so spezifiziert, dass sie die Zustandsvariable und somit auch die Invarianten der abstrakten Maschine übernimmt:

MACHINE healthcare
REFINES healthcare_generic
SEES healthcare_context
VARIABLES
state
⋮

`inv1` und `inv2` behalten also Gültigkeit. Die Initialisierung kann unverändert übernommen werden, ebenso wie diejenigen Ereignisse, welche in dieser Verfeinerung der Maschine nicht anzupassen sind. Um ein Ereignis zu übernehmen, kann es in der konkreten Maschine mit dem selben Namen als Erweiterung des Ereignisses der abstrakten Maschine spezifiziert werden (vgl. „Wiederverwendung“ in Unterabschnitt 3.6.6). So ist die Notation für das Initialisierungsereignis dabei wie folgt[3].

INITIALISATION $\widehat{=}$
extends
INITIALISATION
begin
act1: state $:= \{\mathtt{u1}\} \mapsto \varnothing \mapsto \{\mathtt{u1} \mapsto \mathtt{UserAdmin}\} \mapsto \varnothing \mapsto \varnothing$
end

Schließlich müssen diejenigen Ereignisse behandelt werden, welche von einem verallgemeinerten Ereignis abzuleiten sind. Dabei ist in allen Fällen ein agierende Sitzung als Parameter zu ergänzen, welche dann im Rahmen des erweiterten Bedingungsteils verwendet wird. Bei allen Kommandos außer `assignRole` und `revokeRole` ist zudem der Parameter `r` festzuschreiben.

Für `assignRole` bleibt der Parameter und es ergibt sich die folgende, um den Parameter `s`, dessen Typisierung sowie das Prädikat zur Zugriffskontrolle ergänzte Spezifikation.

assignRole $\widehat{=}$
extends
assignGenericRole
any
s > handelnde Sitzung
u > Zielnutzer

[3] Im Rahmen einer Erweiterung werden alle Parameter, Guards und Aktionen des abstrakten Ereignisses unverändert in das konkrete Ereignis übernommen und grau dargestellt.

```
    r     >  zuzuweisende Rolle

  where
    typ1: s ∈ S_q(state)
    typ2: u ∈ U_q(state)
    typ3: r ∈ ROLE \ UA_q(state)[{u}]
    cnd1: sod(state ↦ u ↦ r) = TRUE
    cnd2: access_SM(state ↦ s ↦ UAo ↦ update) = TRUE

  then
    act1: state := assignRolesToUsers( state ↦ {u ↦ r} )

  end
```

Die übrigen Kommandos können nicht via Erweiterung spezifiziert werden, da `r`, wie erläutert, festzuschreiben und daher auch der Aktionsteil zu verändern ist (beispielsweise ist im Falle des `assignReferredDoctorRole`-Kommandos stets `r` = `ReferredDoctor`). Es ergibt sich nachstehende Spezifikation.

```
assignReferredDoctorRole ≙

  refines
    assignGenericRole

  any
    s     >  handelnde Sitzung
    u     >  Zielnutzer

  where
    typ1: s ∈ S_q(state)
    typ2: u ∈ U_q(state)
    cnd1: sod(state ↦ u ↦ ReferredDoctor) = TRUE
    cnd2: access_SR(state ↦ s ↦ Doctor) = TRUE
    cnd3: access_UR(state ↦ u ↦ Doctor) = TRUE
    cnd4: ReferredDoctor ∉ UA_q(state)[{u}]

  with
    r: r = ReferredDoctor

  then
    act1: state := assignRolesToUsers(
            state ↦ {u ↦ ReferredDoctor}
        )

  end
```

Zu beachten ist insbesondere die `with`-Klausel, deren sog. „Zeugen“ diejenigen Parameter festlegen, welche zum abstrakten Ereignis hin entfernt wurden. Die Behandlung der übrigen Kommandos erfolgt entsprechend; die gesamte Spezifikation ist Anhang A.10 zu entnehmen.

4.3 Zusammenfassung

Im Zuge der Evaluierung in Kapitel 5 sowie zur Dokumentation und Verständlichkeit beschrieb dieses Kapitel die beispielhafte Spezifikation eines Metamodells sowie eines dazugehörigen Sicherheitsmodells. Dabei wurde die in Kapitel 3 entwickelte Vorgehensweise angewendet. Entstanden ist die Spezifikation des Meta- sowie des Sicherheitsmodells, welche Anhang A vollständig dokumentiert. Sie besteht aus zehn Einzeldokumenten auf 22 Seiten und umfasst 833 Zeilen. Alle Beweispflichten wurden erfüllt, sodass die interne Konsistenz der Spezifikation entsprechend der Forderungen gegeben ist. Insgesamt lagen 105 Beweispflichten vor, von denen 56 automatisch bewiesen wurden (53,33 %). Auf der Grundlage der fertigen Spezifikation lässt sich die weitere Verfeinerung und Implementierung im Sinne von Unterabschnitt 2.6.1 vornehmen (vgl. Unterabschnitt 5.1.3). Eine Diskussion der Ergebnisse erfolgt in Kapitel 5.

KAPITEL 5

Evaluierung

Zur Konstruktion von IT-Systemen, welche die von ihnen geforderten Sicherheitseigenschaften erfüllen können, entwickelte diese Masterarbeit eine Methode zur formalen Spezifikation von TCB-Funktionen. An diese Methode wurde eine Reihe an Anforderungen gestellt (siehe Abschnitt 3.1); die Konstruktion der Methode beschrieb Kapitel 3. Die Ergebnisse wurden dann anhand eines Beispielmodells angewandt (siehe Kapitel 4). Im Rahmen der Evaluierung muss nun argumentiert werden, inwiefern die Vorgehensweise den Anforderungen gerecht wird und Spezifikationen hervorbringt, welche gemäß der Zielstellung dieser Arbeit Grundlage für eine verifizierbare Realisierung ist. Dabei sollen insbesondere Wege zur Verbesserung und Weiterentwicklung aufgezeigt werden.

Eine Evaluation kann sowohl qualitativ, als auch quantitativ erfolgen. Da jedoch kaum belastbare quantitative Aussagen getroffen werden können, beschränkt sich das vorliegende Kapitel größtenteils auf eine Diskussion qualitativer Aspekte. Abschnitt 5.1 wird dazu die Anforderungen analysieren und erörtern, inwiefern diese durch die entwickelte Vorgehensweise erfüllt wurden. Im Anschluss geht Abschnitt 5.2 auf einzelne Probleme detailiert ein. Schließlich stellt Abschnitt 5.3 die verfügbaren quantitativen Aussagen zur Evaluierung vor.

5.1 Erfüllung der Anforderungen

Die in dieser Arbeit entwickelte Methode zielt auf die Korrektheit der Spezifikation sowie davon abgeleiteter Dokumente ab, soll das Kriterium der Wiederverwendbarkeit erfüllen sowie zur korrekten Implementierung von Sicherheitspolitiken in TCB-Laufzeitsystemen anwendbar sein (vgl. Kapitel 1). Diese Kriterien werden nachfolgend in den Unterabschnitten 5.1.1 bis 5.1.3 getrennt evaluiert.

5.1.1 Korrektheit der Spezifikation

Bezüglich der Korrektheit muss die Vorgehensweise die Anforderungen aus Abschnitt 3.1 erfüllen. Die Anforderungen (S1) und (S2), **Vollständigkeit** und **externe Konsistenz**, wurden durch einen konstruktiven Ansatz behandelt: die vorgestellte Methode erstellt vollständige und zum zugrunde liegenden Sicherheitsmodell konsistente Spezifikationen, indem die Bestandteile solcher Modelle vollständig erfasst und dann systematisch behandelt werden. Notwendig zum Erfolg dieses Ansatzes ist dabei jedoch die Verständlichkeit der Spezifikation (Anforderung (S5)): nur so kann der Spezifizierende den nötigen Überblick wahren. Ein Problem stellt die

Maßgabe auf, die Spezifikation mittels eines Animators auf ihre Konsistenz zum Sicherheitsmodell zu überprüfen: derzeit ist eine Animierung der Spezifikation nur unter erhöhtem Aufwand möglich (vgl. Unterabschnitt 5.2.3). Um die Vollständigkeit und externe Konsistenz sicherzustellen, ist hier also noch ein erhöhtes Maß an Aufmerksamkeit durch den Spezifizierenden nötig.

Zur Sicherung der **internen Konsistenz** der Spezifikation (Anforderung (S3)) wird in den Behandlungsstrategien in Unterabschnitt 3.2.1 auf größtenteils werkzeuggestützte Validierungstechniken verwiesen. Hierbei ist es möglich, alle Beweispflichten zu erfüllen und somit ein hohes Maß an Vertrauen in die internen Konsistenz zu erreichen – dies zeigt die Beispielspezifikation. Allerdings konnte dort nur die Hälfte aller Beweispflichten automatisch erfüllt werden, was einen hohen Arbeitsaufwand für den Spezifizierenden bedeutet. Die Güte der Methode ist also stark von der Reife der Werkzeuge abhängig (siehe Diskussion in Unterabschnitt 5.2.3). Daneben spielen der Erfahrungsschatz (siehe Diskussion in Unterabschnitt 5.2.1) sowie die verwendete Spezifikationssprache eine Rolle. Letztere hat großen Einfluss darauf, inwiefern Inkonsistenzen für den Spezifizierenden sichtbar sind (siehe Diskussion in Unterabschnitt 5.2.2). Mit der vorgestellten Methode kann also ein hohes Maß an Vertrauen in die interne Konsistenz erzielt werden, der Aufwand hierzu kann jedoch weiter gesenkt werden.

Gleiches gilt für die **Präzision** (Anforderung (S4)). Letztere kann bezüglich sämtlicher Teile der Spezifikation gefordert werden, insbesondere bei Strukturen, Zusammenhängen, Abläufen / Operationen und Bedingungen. Dabei ist die Präzision überall dort, wo eine mathematische Notation zum Einsatz kommt, schon gegeben (also insbesondere in Ausdrücken, wie sie in Axiomen, Invarianten, Guards und Aktionen vorkommen). Aber auch die übrigen Zusammenhänge und Strukturen des Meta- und Sicherheitsmodells müssen in der Spezifikation präzise abgebildet werden. Dabei helfen starke Vorbedingungen, die Semantik der Operationen präziser abzubilden. Bezüglich der verwendeten Trennung von Meta- und Sicherheitsmodellen fällt jedoch auf, dass Event-B die Deklaration eines Zustandsraumes mit darauf definierten Funktionen nicht explizit unterstützt; die hier genutzte Notation ist unnötig komplex. Eine präzise Schreibweise könnte vorallem durch eine spezialisierte Spezifikationsnotation erreicht werden (siehe Diskussion in Unterabschnitt 5.2.2). Ungeachtet dessen wird mittels der präsentierten Methode ein hohes Maß an Präzision erreicht.

Da viele der hier angewandten Techniken und Methoden in einem hohen Maße vom Benutzer angewendet werden müssen, hängt die Korrektheit der Spezifikation und ihrer Realisierungen wesentlich von Anforderung (S5), ihrer **Verständlichkeit**, ab. Dabei fällt zuerst wieder die Wahl der Spezifikationssprache ins Gewicht, deren Design der Spezifikation ihre Form gibt. Nun ist Event-B als Spezifikationssprache für zustandsbasierte Softwaresysteme konzipiert. Daher sind die erzeugten Spezifikationen für Betrachter mit Wissen aus der Domäne der IT-Sicherheit nicht intuitiv verständlich; vielmehr ist Erfahrung mit der Spezifikationssprache selbst nötig. Auch hier wäre die Nutzung einer spezialisierten Spezifikationsnotation hilfreich (siehe Diskussion in Unterabschnitt 5.2.2). Abgesehen davon wurde die Spezifikation jedoch möglichst verständlich für Personen mit Domänenwissen in der IT-Sicherheit gehalten: so entstammen sämtliche Namen direkt aus dem Sicherheitsmodell, auch

dessen Struktur wurde für die Spezifikation wiederverwendet. Zudem hängt die Verständlichkeit der entstandenen Spezifikation sehr von deren Größe und Komplexität ab (siehe Anforderung (S7)).

Wie in Unterabschnitt 2.2.2 dargelegt, ist es zum Erreichen des Korrektheitsziels entscheidend, dass die entwickelte Spezifikation von **geringer Größe und Komplexität** ist (Anforderung (S7)). Bei Betrachten der Ergebnisse aus Kapitel 4 fällt auf, dass die komplexesten und umfänglichsten Teile der Spezifikation wie gewünscht (vgl. Unterabschnitt 3.2.2) im Metamodell liegen und sich mit dem Zustandsraum und den darauf definierten Primitivoperationen beschäftigen (Anhänge A.2 bis A.5). Ein Teil der Komplexität lässt sich dabei auf die Notwendigkeit zurückführen, einen Zustandsraum als Menge anzugeben, statt im Metamodell eine verallgemeinerte Zustandsmaschine zu konstruieren. Dies ließe sich über eine spezialisierte Spezifikationssprache beheben (siehe Diskussion in Unterabschnitt 5.2.2). Der andere Teil der Komplexität entsteht durch die Nutzung starker Vorbedingungen, wodurch insbesondere die Guards teilweise übermäßig komplex werden. Die Gefahr an dieser Stelle ist, dass der durch starke Vorbedingungen gewünschte Effekt der Dokumentation und verbesserten Verständlichkeit verloren geht. Andererseits entstehen die komplexeren Guards (wie sie etwa in `rbac_rolehandling` in Anhang A.5 zu finden sind) vorallem daraus, dass die Zustandskomponente `roles` als Abbildung auf eine Potenzmenge modelliert wurde – dies ist ungünstig, da mengentheoretisch schwer handhabbar (vgl. Diskussion in Vorgehensweise (B6)) Durch geeignete Umformulierung ließe sich hier also leicht Abhilfe schaffen. Zur Größe der Spezifikation versucht Abschnitt 5.3, quantitative Aussagen zu treffen. Von auffälliger Größe sind erstens die Initialisierung statischer Komponenten in `healthcare_context` (vgl. Unterabschnitt 4.2.1) und zweitens die Spezifikation der Primitivoperationen. Ursächlich für den Umfang der letzteren sind vorallem die mit „`_v`“ gekennzeichneten Validierungstheoreme (vgl. Unterabschnitt 3.6.6), welche im Wesentlichen eine Duplizierung vorhandener Aussagen darstellen. Eine Auslagerung dieser Theoreme in einen separaten Kontext wäre möglich, würde aber Inkonsistenzen zwischen Definitioenn und Validierungstheoremen provozieren. Hier müsste entweder über eine spezialisierte Spezifikationssprache oder über in die Entwicklungsumgebung integrierte Werkzeugunterstützung nachgedacht werden (siehe Diskussion in Unterabschnitt 5.2.2).

Zwar mangelt es an Metriken, die geringe Größe und Komplexität zu begründen. Traten in der Methode oder der Spezifikation jedoch Stellen auffälliger Größe oder Komplexität auf, so wurde deren Notwendigkeit stets argumentiert. Der Erfolg ist also von der geschickten Formulierung der Spezifikation abhängig und lässt sich durch eine spezialisierte Notation weiter verbessern.

5.1.2 Wiederverwendbarkeit

Der gewünschte Grad der Wiederverwendbarkeit (dies ist gleichzeitig Anforderung (S6)) wurde erzielt: die aufgezeigte Methode ermöglicht es hierzu, Metamodelle separat von konkreten Sicherheitsmodellen zu spezifizieren. Eine Metamodell-Spezifikation kann für zugehörige Sicherheitsmodelle verwendet werden.

Eine zusätzliche Verbesserungsmöglichkeit liegt darin, die Wiederverwendbarkeit

zwischen verschiedenen Metamodellen zu erhöhen: dazu könnten diese schrittweise durch statische bzw. dynamische Komponenten erweitert werden, um so eine Hierarchie aus Metamodellen zu konstruieren. Hierzu wäre jedoch eine Spezifikationsnotation mit umfangreichen Vererbungsbeziehungen, oder aber eine für den vorliegenden Anwendungsfall spezialisierte Spezifikationsnotation notwendig (siehe Diskussion in Unterabschnitt 5.2.2).

5.1.3 Praktische Anwendbarkeit

Zur Anwendbarkeit der vorgestellten Vorgehensweise ist es einerseits notwendig, dass diese selbst durchführbar ist, und andererseits, dass anhand der damit erstellten Spezifikation eine korrekte Implementierung herleitbar ist.

Die Anwendbarkeit der Vorgehensweise auf das gegebene Beispiel wurde in Kapitel 4 gezeigt. Zur Herleitung der Vorgehensweise war die Grundlage stets die allgemeine Struktur eines Sicherheitsmodells, wie in Abschnitt 2.3 beschrieben. Daher ist die Methode auch für allgemeine Sicherheitsmodelle anwendbar. Einzig der in den bewährten Vorgehensweisen in Unterabschnitt 3.2.3 gesammelte Erfahrungsschatz kann erst nach wiederholter Anwendung der Methode auf unterschiedlichste Modelle ein sinnvolles Maß an Vollständigkeit erreichen (siehe dazu auch die Diskussion in Unterabschnitt 5.2.1).

Die Möglichkeit, eine korrekte Implementierung aus den hier erzeugten Spezifikationen herzuleiten, ist selbst nicht Gegenstand dieser Arbeit, daher sollen hier Ansatzpunkte genannt werden. Erstens existiert eine Reihe industrieller Projekte mit Event-B [Evea]. Zweitens gibt es Reihe an Plugins, welche die Codeerzeugung aus Event-B-Spezifikationen auf unterschiedliche Art und Weise unterstützen können [Eveb]. Drittens besteht die in Unterabschnitt 2.6.1 abstrakt beschriebene Möglichkeit, die Spezifikation weiter zu verfeinern, um so die verwendeten Datenstrukturen und Algorithmen einem Implementierungsniveau anzunähern, wobei die Korrektheit eines jeden Teilschritts beweisbar ist. Ein Beispiel hierfür ist in Anhang B dokumentiert.

5.2 Probleme im Detail

Der vorliegende Abschnitt diskutiert vertiefend einzelne Fragestellungen zur Evaluierung, welche der vorhergehende Abschnitt aufgeworfen hat.

5.2.1 Vollständigkeit der Best Practices

Unterabschnitt 3.2.3 argumentiert, dass die Qualität einer Spezifikation bezüglich Konsistenz, Verständlichkeit und Komplexität stark von der Erfahrung des Spezifizierenden abhängt. Hierzu gab der Unterabschnitt eine Liste bewährter Vorgehensweisen an. Diese jedoch können nicht mehr als ein wachsender Erfahrungsschatz sein, der in der vorliegenden Arbeit zwar bereits Quellen wie [Bos95] zur Grundlage hat, jedoch zweifellos weiter ausgebaut werden muss. So konnte in dieser Arbeit lediglich *ein* Sicherheitsmodell zum Beispiel genommen werden; praktisch sind jedoch mit unterschiedlichen Sicherheitsmodellen auch neue Sonderfälle

zu erwarten, welche spezielle Lösungen erfordern. Eine konsolidierte, gesicherte Methode kann also erst mit einiger Erfahrung entstehen.

5.2.2 Eignung der Spezifikationssprache

Die genutzte Spezifikationssprache, Event-B (siehe Unterabschnitt 2.6.4), überzeugt als reife Sprache zur Spezifikation zustandsbasierter Systeme mit umfangreicher, erweiterbarer Werkzeugunterstützung. Sprachfeatures wie die Benennung von Axiomen, Guards etc. erhöhen die Lesbarkeit und Verständlichkeit enorm; die mathematische Notation sowie die allgemeine Struktur der Notation sind gut niederzuschreiben und gut zu lesen. Für den vorliegenden Einsatzzweck sind jedoch folgende Unzulänglichkeiten sichtbar geworden.

Die intuitive Verständlichkeit einer Spezifikation für einen Betrachter, der lediglich über Wissen aus der Domäne der IT-Sicherheit verfügt, ist nicht gegeben: Struktur und Notation sind zwar an allgemeingültige, mathematische Prinzipien angelehnt und daher gut erlernbar, jedoch immernoch sehr speziell (so etwa die Semantik von Spezifikationsbestandteilen wie Axiomen und Ereignissen, oder die Definition einiger mathematischer Operatoren). Zudem gibt es zwischen Zustandsmaschinen keine geeigneten Vererbungsbeziehungen, sodass die Abstraktion zwischen Metamodellen und konkreten Sicherheitsmodellen nur implementiert werden kann, indem im Metamodell ein Zustandsraum deklariert wird. Diese Notation jedoch ist unpräzise, komplex und daher schwer verständlich.

Zur Lösung dieser Probleme schlägt die Masterarbeit die Schaffung einer domänenspezifischen Sprache vor, entweder durch Erweiterung von Event-B, oder mittels einer eigenständigen Notation zur Spezifikation von Sicherheitsmodellen. [Lam00] argumentiert, dass formale Methoden besonders dann nützlich sind, wenn sie in einer begrenzten Domäne anwendbar sind: denn „spezielle Arten von Systemen benötigen spezielle Arten von Ausdrucks- und Analysetechniken“. Eine spezialisierte Sprache zur Spezifikation von Sicherheitsmodellen ist erstens verständlicher für Personen mit Domänenwissen, was die Korrektheit der Resultate erhöht und die Eignung der Spezifikation zu Dokumentationszwecken verbessert. Zweitens enthält sie anwendungspezifische Sprachkonstrukte, welche mittels syntaktischem Zucker die Komplexität bestimmter, häufig benötigter Ausdrücke (wie etwa in Unterabschnitt 4.2.1) senken können. Drittens können mit einer solchen domänenspezifischen Sprache die Vorteile mehrerer formaler Methoden genutzt werden, indem ein Compilerwerkzeug anhand einer Spezifikation verschiedene Arten von Dokumenten erzeugen kann: eine leicht animierbare B-Spezifikation, welche auf die Wiederverwendbarkeit zwischen Metamodell und Sicherheitsmodell verzichtet, kann ebenso automatisch ausgegeben werden wie die vollständige Event-B-Spezifikation, anhand derer dann die Beweispflichten geprüft werden können. Viertens könnte diese domänenspezifische Sprache möglicherweise als Ausgangspunkt zur halbautomatisierten Verfeinerung oder Realisierung genutzt werden.

Die Korrektheit der Spezifikation in der domänenspezifischen Notation basiert dann auf einem Isomorphismus zwischen domänenspezifischer Notation und Event-B, sowie auf dem Compiler, welcher diesen Isomorphismus korrekt abbilden muss.

5.2.3 Eignung der Werkzeuge

Zur Sicherung der internen Konsistenz der Spezifikationen verlässt sich die vorgestellte Vorgehensweise in hohem Maße auf die Unterstützung von Werkzeugen (vgl. Unterabschnitt 3.2.1). Damit jedoch ist die Korrektheit der entstandenen Spezifikation abhängig von der Korrektheit, dem Funktionsumfang, der Benutzbarkeit und insbesondere der Verständlichkeit der eingesetzten Werkzeuge. Die Vorgehensweise dieser Arbeit verlässt sich dabei primär auf die Rodin-Entwicklungsumgebung sowie den ProB-Animator (vgl. Unterabschnitt 2.6.4). Diese Werkzeuge wurden zwar bereits in industriellen Fallstudien erprobt [Evea; ProB], sind aber dennoch Gegenstand aktueller Forschungsprojekte, sodass sie einerseits zur Spezifikation gut einsetzbar sind, jedoch andererseits noch nicht über den Reifegrad der üblichen Softwareentwicklungswerkzeuge verfügen.

So zeigte sich, dass der interaktive Theorembeweiser in Rodin einen zu hohen Grad an Nutzerinteraktion erfordert: viele Aussagen, die umständlich vom Nutzer bewiesen werden müssen, könnten mit ausgereifterem Werkzeug auch automatisch gezeigt werden. Siehe hierzu auch die Ergebnisse in Abschnitt 5.3. Der Animator ist derzeit nicht in der Lage, Spezifikationen zu animieren, welche mit der vorliegenden Methode erstellt wurden. Die Ursache hierfür liegt darin, dass der Zustandsraum als sehr große Menge modelliert ist und die Primitivoperationen auf dieser Menge definiert sind – solche (potentiell unendlich großen) Modelle sind mit ProB momentan nicht animierbar[1]. Abhilfe schafft außer einer Verbesserung des Werkzeugs auch die Verfeinerung der Spezifikation hin zu einer Version mit vereinzelten Zustandsvariablen.

5.3 Quantitative Evaluierung

Im Kontext des vorliegenden Problems liegen kaum Metriken mit vergleichbaren Werten vor.

Aussagen zur Größe können allein anhand von [Bos95] getroffen werden: dort ist die Spezifikation einer Sicherheitspolitik in Z dokumentiert; diese Sicherheitspolitik, eine Kombination aus Bell & LaPadua sowie Clark & Wilson, ist jedoch kaum mit der des vorliegenden Beispielmodells (vgl. Abschnitt 2.4) vergleichbar. Der Autor gibt an, dass die meisten Z-Schemas der entstandenen Spezifikation zwischen 11 und 20 Zeilen umfassen, darunter mehrere („serveral“) über 20 Zeilen. Schemas in Z sind mit der Deklaration des Zustandes oder eines Ereignisses in Event-B vergleichbar. Da Z keine Schlüsselwörter benutzt, kann die vorliegende Beispielspezifikation in Anhang A grob verglichen werden, indem etwa für jedes Ereignis die Zeilenzahl des Guard- und Aktionsabschnittes summiert wird. Dabei ist festzustellen, dass die Spezifikation dieser Arbeit kein Ereignis mit mehr als 15 solcher Zeilen umfasst. Die Deklarationen der Primitivoperationen und des Zustandes liegen ebenfalls unterhalb dieser Schranke. Die einzelnen Elemente der Spezifikation dieser Arbeit sind demnach kleiner als jene in [Bos95].

[1] siehe hierzu `http://www.stups.uni-duesseldorf.de/ProB/index.php5/Tutorial_Modeling_Infinite_Datatypes`

	[Abr07]	[ITL+10]	[Wri08]	vorl.MA
Sprache	B	Event-B	Event-B	Event-B
Zeilen	183.000	n/a	n/a	833
Beweispflichten	43.610	191	5.456	105
automatisch bewiesen	97 %	83 %	53 %	53 %

Tabelle 5.1: Metriken aus Fallstudien zur Anwendung von B und Event-B.

Im Rahmen von [Bos95] wurden keine Beweiswerkzeuge eingesetzt; die Validierung erfolgte dort aufgrund informeller Argumentation, redundanter Spezifikation sowie der Anwendung von Best Practices. Im Rahmen der hier vorliegenden Masterarbeit wurde zur Validierung ein interaktiver Theorembeweiser eingesetzt, dabei sind von 105 Beweispflichten 53,3 % automatisch bewiesen worden. Tabelle 5.1 bietet einen Vergleich dieser Zahl mit drei Fallstudien zum Einsatz von B und Event-B. Die Auswahl bezieht Studien mit unterschiedlichen Problemdomänen, Größen und Werkzeugen ein. Es handelt sich erstens um eine industrielle Anwendung von B [Abr07]; Gegenstand der Spezifikation war hier ein System zur automatischen Steuerung von Zügen. Die Spezifikation ist äußerst umfangreich und wurde mit ausgereiften Werkzeugen erstellt. Die zweite Studie beschreibt die Event-B-Spezifikation eines Systems zur Positionskontrolle in Satelliten [ITL+10]. Diese Spezifikation ist sehr klein und beschränkt sich darauf, in einem aus mehreren unabhängigen Komponenten bestehenden System den globalen Systemzustand zu modellieren. Ziel ist dabei, das korrekte Zusammenspiel der Komponenten zu sichern. Die Größe dieser Spezifikation ist Vergleichbar zur Größe der Beispielspezifikation in dieser Masterarbeit; die Problemstellung ist mit jener der ersten Studie vergleichbar. Bei der dritten Fallstudie handelt es sich um eine Event-B-Spezifikation des Befehlssatzes einer virtuellen Maschine [Wri08]. Hier ist die Problemstellung mit jener der Beispielspezifikation vergleichbar, die Spezifikation ist jedoch etwa eine Größenordnung größer[2]. Die Quote automatisch gezeigter Beweispflichten ist im Falle der letzten Fallstudie fast identisch zur Quote des Beispielmodells. [ITL+10] weist bereits eine wesentlich bessere Quote auf, und bei [Abr07] konnten fast alle Beweispflichten automatisch gezeigt werden.

Folgende Unterschiede zwischen den Projekten können ursächlich für diese Zahlen sein. Zunächst unterscheiden sich die Projekte in ihren Spezifikationssprachen, die sich wiederum in Ausdruckskraft, Präzision und Verständlichkeit unterscheiden können. Weiterhin wurden unterschiedliche Werkzeuge eingesetzt. Deren Reifegrad ist maßgeblich dafür, welche Zusammenhänge durch die automatischen Theorembeweiser erschlossen werden können. Drittens unterscheiden sich die Probleme selbst. Dabei können bestimmte Probleme zugänglicher für die vorliegenden Beweiser sein als andere, beispielsweise aufgrund der Komplexität der gegebenen Zusammenhänge oder durch die unterschiedlich ausgeprägte Nutzung verschiedener Datentypen wie Zahlen, Mengen oder Wahrheitswerte. Schließlich hängt der Erfolg automatischer Beweisverfahren wesentlich von einer geeigneten Formulierung der Spezifikation ab.

[2]Diese Aussage basiert auf der Annahme, dass die Zahl der Zeilen mit der Zahl der Beweispflichten wächst.

Zu den vorliegenden Studien ist bekannt, dass [Abr07] über die reifsten Werkzeuge verfügte – die Entwicklung der neueren Entwicklungsumgebung zu Event-B hat noch nicht denselben Stand erreicht. Zudem ist davon auszugehen, dass die Problemstellungen in [Abr07] und [ITL+10] zugänglicher für die Beweiswerkzeuge sind: deren Komplexität entsteht insbesondere im Rahmen der Dekomposition während der Konstruktion der Spezifikation, während die Spezifikationen in [Wri08] sowie der vorliegenden Masterarbeit bereits durch die inneren Zusammenhänge komplex sind. Auswirkungen durch Unterschiede zwischen den Sprachen sind nicht zu erwarten; über die Eignung der gewählten Formulierungen kann keine Aussage getroffen werden.

Der niedrige Automatisierungsgrad der Beweise kann also auf die Komplexität des Problems sowie fehlende Reife der Werkzeuge zurückgeführt werden. Rückschlüsse auf die Eignung der Formulierung sind nicht möglich.

5.4 Zusammenfassung

Ziel des Evaluierungskapitels war es, die Ergebnisse der Arbeit hinsichtlich der an sie gestellten Anforderungen zu evaluieren. Die Anforderungen gliedern sich dabei in Korrektheit, Wiederverwendbarkeit und die praktische Umsetzbarkeit. Neben qualitativen Argumenten wurden, soweit möglich, auch quantitative Aussagen betrachtet.

Von der Methode erstellte Spezifikationen erfüllen unter Einsatz der nötigen Sorgfalt alle an sie gestellten Korrektheitsanforderungen: sie sind vollständig und konsistent bezüglich des zugrunde liegenden Sicherheitsmodells, sie sind widerspruchsfrei, und darüber hinaus präzise und verständlich formuliert sowie von geringer Größe und Komplexität. Dabei besteht Raum für Verbesserungen: so können die Vollständigkeit und Konsistenz wesentlich besser abgesichert werden, wenn die Spezifikationen animierbar sind. Der Aufwand zum Nachweis der Widerspruchsfreiheit ist derzeit akzeptabel, kann aber durch Verbesserung der Werkzeuge weiter gesenkt werden. Vorallem aber brächte eine Spezialisierung der Sprache große Gewinne für alle Korrektheitsaspekte, insbesondere hinsichtlich der Verständlichkeit, der Größe und Komplexität, der Wiederverwendbarkeit sowie der Analysierbarkeit der Spezifikationen.

Zur Wiederverwendbarkeit wurden die gestellten Anforderungen erfüllt. Eine Sprachspezialisierung könnte zusätzliche Möglichkeiten der Wiederverwendung erschließen.

Die praktische Umsetzbarkeit der Methode wurde anhand eines Beispielmodells demonstriert. Die Implementierbarkeit der erstellten Spezifikationen liegt außerhalb der Aufgabenstellung dieser Masterarbeit, hier zeigte das Kapitel jedoch Ansatzpunkte auf.

KAPITEL 6

Zusammenfassung

Zur Konstruktion sicherer IT-Systeme erarbeitete die vorliegende Masterarbeit eine Methode, um die Funktionen der Trusted Computing Base des Systems formal zu spezifizieren. Ansatzpunkt der Methode ist dabei das Sicherheitsmodell des gegebenen Systems in einer zustandsbasierten Notation.

Die Methode wurde anforderungsorientiert entwickelt. Die Anforderungen ergaben sich hierbei aus der Zielstellung der Masterarbeit sowie den speziellen Bedürfnissen im Rahmen der formalen Spezifikation einer Trusted Computing Base. Zur besseren Wiederverwendbarkeit wurde die Spezifikationsmethode für Sicherheitsmodelle und Metamodelle (dies sind Klassen von Sicherheitsmodellen) getrennt erarbeitet. Zudem wurde die Methode zuerst losgelöst von einer konkreten Spezifikationsnotation konstruiert, sodass sie leicht auf beliebige Notationen zur formalen, zustandsbasierten Spezifikation übertragbar ist. Anschließend wurde die Methode für eine konkrete Notation beschrieben, nämlich für Event-B. Es handelt sich hierbei um eine Spezifikationssprache, die aus jüngeren Forschungsprojekten hervorgeht und aufgrund ihrer Modularität, Verständlichkeit, Benutzbarkeit, Erweiterbarkeit, freien Verfügbarkeit sowie der guten Werkzeugunterstützung gewählt wurde.

Eine Spezifikation, welche gemäß der Methode erstellt wird, setzt das gegebene Sicherheitsmodell vollständig um und ist konsistent zu diesem; beide Eigenschaften werden derzeit konstruktiv erreicht. Die Widerspruchsfreiheit der Spezifikation ist mittels eines Generators für Beweispflichten sowie eines interaktiven Theorembeweisers nachweisbar. Die Verständlichkeit der Spezifikation ist für Personen, die mit der Spezifikationsnotation vertraut sind, uneingeschränkt gegeben; für Personen, die lediglich über Domänenwissen verfügen, ist die Spezifikation nur eingeschränkt intuitiv verständlich. Sie ist für Sicherheitsmodelle desselben Metamodells wiederverwendbar. Die geringe Größe und Komplexität der Spezifikation ist derzeit nicht quantitativ belegbar, jedoch argumentierbar. Wo möglich, wurden komplexe Aspekte der Spezifikation in das (wiederverwendete) Metamodell ausgelagert. Die Spezifikation ist schließlich für eine Realisierung nutzbar, hier ist jedoch weitere Forschungsarbeit nötig.

Die Anwendbarkeit der Spezifikationsmethode wurde an einem Sicherheitsmodell im RBAC-Metamodell demonstriert. Die zugrunde liegende Politik modelliert eine Einrichtung im Gesundheitsbereich und umfasst zehn Rollen sowie ein aus 14 Kommandos bestehendes Autorisierungsschema. Die entstandene Event-B-Spezifikation weist ingesamt 833 Zeilen auf, ihre Widerspruchsfreiheit ist bewiesen.

6.1 Ausblick

Die vorliegende Masterarbeit präsentierte eine Methode zur Konstruktion von Spezifikationen. Es bleibt zunächst zu erforschen, mittels welcher Vorgehensweise aus solchen Spezifikationen mit möglichst wenig Aufwand eine verifizierbar korrekte Realisierung abgeleitet werden kann. Dabei sollten vorhandene Fallstudien, die vorhandene Werkzeugunterstützung sowie bekannte Techniken zur Verfeinerung und automatischen Transformation (z.B. zu Diagrammen oder Code) analysiert werden.

Zudem zeigte die Evaluierung, dass es vorteilhaft wäre, die Spezifikationssprache auf die Domäne zu spezialisieren. Dies würde sich erstens positiv auf die Verständlichkeit der entstehenden Spezifikationen auswirken, zweitens auf deren Größe und Komplexität. Drittens könnten sie besser mit verschiedenen Werkzeugen zur Analyse und Codeerzeugung verwendet werden. Wie gezeigt, müsste keine vollkommen neue Sprache entwickelt werden: durch die Erweiterbarkeit der Event-B-Werkzeuge ist es denkbar, eine Sprache zur Spezifikation von Meta- und Sicherheitsmodellen zu schaffen, die auf Event-B aufsetzt, in die vorhandene Entwicklungsumgebung integriert ist und alle dortigen Werkzeuge (wie etwa den Theorembeweiser) nutzen kann. Ermöglicht wird das beispielsweise dadurch, dass alle Event-B-Spezifikationen im XML-Format gespeichert werden, welches erweiterbar ist.

Im Zuge der Sprachspezialisierung ist auch zu untersuchen, inwiefern die Wiederverwendung innerhalb der entstandenen Methode verbessert werden kann. Dabei wäre es erstens denkbar, Wiederverwendung zwischen Metamodellen zu ermöglichen. Zweitens sollte ein Weg gefunden werden, Modellkomponenten je nach Sicherheitsmodell flexibel als statisch oder dynamisch zu deklarieren, statt dies im Metamodell festzuschreiben.

Schließlich müssen die Spezifikationen zur besseren Validierung animierbar gemacht werden. Hierzu würde es genügen, eine Verfeinerung der Spezifikation zu erstellen, welche durch den Animator lesbar ist. Möglich wäre jedoch auch die automatische Umformung der Spezifikation, beispielsweise im Zuge der oben erörterten Sprachspezialisierung, oder aber die Verbesserung der vorhandenen Werkzeuge.

Anhang

A Vollständige Spezifikation des Beispielmodells

Der vorliegende Anhang enthält die vollständige Spezifikation des Beispielmodells aus Abschnitt 2.4. Der Spezifikationsprozess des Beispiels ist dokumentiert in Kapitel 4.

A.1 Kontext `rbac_static`

```
1   CONTEXT rbac_static
2   SETS
3      USER
4      ROLE
5      SESSION
6      OBJECT
7      OPERATION
8   CONSTANTS
9      M     > Rechtematrix
10     RH    > Rollenhierarchie
11     RE    > Rollenausschluss
12  AXIOMS
13     M_t: M ∈ ROLE × OBJECT ⇸ ℙ(OPERATION)
14     M1: ∀r, o·r ∈ ROLE ∧ o ∈ OBJECT ∧ (r ↦ o ∈ dom(M))
15         ⇒ finite(M(r ↦ o))      > Endlichkeit

16     RH_t: RH ∈ ROLE ↔ ROLE
17     RH1: RH = RH; RH      > Transitivität, Reflexivität
18     RH2: ∀r1, r2·
19              r1 ∈ ROLE ∧ r2 ∈ ROLE ∧
20              (r1 ↦ r2) ∈ RH ∧ (r2 ↦ r1) ∈ RH
21         ⇒ r1 = r2      > Antisymmetrie

22     RE_t: RE ∈ ROLE ↔ ROLE
23     RE1: RE = RE^−1      > Symmetrie
```

```
24      RE2: ∀r1, r2·
25              r1 ∈ ROLE ∧ r2 ∈ ROLE ∧
26              (r1 ↦ r2) ∈ RE
27          ⇒ r1 ≠ r2      > Irreflexivität
28  END
```

A.2 Kontext `rbac_state`

```
1   CONTEXT rbac_state
2   EXTENDS
3      rbac_static
4   CONSTANTS
5      STATE     > Zustandsraum
6      CONSISTENT_STATE
7      U_q     > Projektionsfunktionen
8      S_q
9      UA_q
10     user_q
11     roles_q
12  AXIOMS
13     st: STATE = {
14             U ↦ S ↦ UA ↦ user ↦ roles |
15             U ∈ ℙ(USER) ∧ S ∈ ℙ(SESSION) ∧
16             UA ∈ USER ↔ ROLE ∧ dom(UA) ⊆ U ∧
17             user ∈ SESSION ⇸ USER ∧ dom(user) ⊆ S ∧ ran(user) ⊆ U ∧
18             roles ∈ SESSION ⇸ ℙ(ROLE) ∧ dom(roles) = S ∧
19             ( ∀ s · s ∈ S ⇒ finite(roles(s)) ) ∧
20             ( ∀ s · s ∈ dom(user) ⇒ roles(s) ⊆ UA[ {user(s)} ] )
21         }
22     cst: CONSISTENT_STATE = {
23             U ↦ S ↦ UA ↦ user ↦ roles |
24             U ↦ S ↦ UA ↦ user ↦ roles ∈ STATE ∧
25             dom(user) = S
26         }
27     cst1: CONSISTENT_STATE ⊆ STATE     theorem

28     st-i: ∀U, S, UA, user, roles · (
29             ∃ q · q ∈ STATE ∧ q = U ↦ S ↦ UA ↦ user ↦ roles
30         ) ⇒
31             dom(UA) ⊆ U ∧
32             dom(user) ⊆ S ∧ ran(user) ⊆ U ∧
33             dom(roles) = S ∧ ( ∀ s · s ∈ S ⇒ finite(roles(s)) ) ∧
34             ( ∀ s · s ∈ dom(user) ⇒ roles(s) ⊆ UA[ {user(s)} ] )     theorem
35         > Übertragen der Invarianten auf Variablen
```

```
36      cst-i: ∀U, S, UA, user, roles · (
37              ∃q · q ∈ CONSISTENT_STATE ∧
38              q = U ↦ S ↦ UA ↦ user ↦ roles
39          ) ⇒
40              dom(UA) ⊆ U ∧
41              dom(user) = S ∧ ran(user) ⊆ U ∧
42              dom(roles) = S ∧ ( ∀s · s ∈ S ⇒ finite(roles(s)) ) ∧
43              ( ∀s · s ∈ dom(user) ⇒ roles(s) ⊆ UA[ {user(s)} ] )     theorem

44      st1: ∀q · q ∈ STATE ⇒
45              prj2(prj1(q)) ∈ SESSION ⇸ USER ∧
46              prj1(prj1(q))  ∈  ℙ(USER) × ℙ(SESSION) × (USER ↔ ROLE)
47          theorem     > erleichtert Beweise
48      st2: ∀q · q ∈ STATE ⇒
49              prj2(q) ∈ SESSION ⇸ ℙ(ROLE) ∧
50              prj1(q) ∈ ℙ(USER) × ℙ(SESSION) × (USER ↔ ROLE) ×
51                  (SESSION ↔ USER)     theorem

52      stU-t: U_q ∈ STATE → ℙ(USER)
53      stS-t: S_q ∈ STATE → ℙ(SESSION)
54      stA-t: UA_q ∈ STATE → (USER ↔ ROLE)
55      stu-t: user_q ∈ STATE → (SESSION ⇸ USER)
56      str-t: roles_q ∈ STATE → (SESSION ⇸ ℙ(ROLE))
57      st-d: ∀q · q ∈ STATE ⇒ U_q(q) = prj1(prj1(prj1(prj1(q)))) ∧
58          ∀q · q ∈ STATE ⇒ S_q(q) = prj2(prj1(prj1(prj1(q)))) ∧
59          ∀q · q ∈ STATE ⇒ UA_q(q) = prj2(prj1(prj1(q))) ∧
60          ∀q · q ∈ STATE ⇒ user_q(q) = prj2(prj1(q)) ∧
61          ∀q · q ∈ STATE ⇒ roles_q(q) = prj2(q)

62      st3: ∀q · q ∈ STATE ⇒ (
63              ∃U, S, UA, user, roles ·
64              U ↦ S ↦ UA ↦ user ↦ roles = q ∧
65              U_q(q) = U ∧
66              S_q(q) = S ∧
67              UA_q(q) = UA ∧
68              user_q(q) = user ∧
69              roles_q(q) = roles
70          )    theorem     > Überführen der Zustandskomponenten in Variablen
71  END
```

A.3 Kontext `rbac_userhandling`

```
1   CONTEXT rbac_userhandling
2   EXTENDS
3       rbac_state
```

```
4   CONSTANTS
5      addUsers
6      deleteUsers
7   AXIOMS
8      add_t: addUsers ∈ STATE × ℙ(USER) → STATE
9      add_d: ∀ u, q ·
10            q ∈ STATE ∧
11            u ∈ ℙ(USER) ∧ u ∩ U_q(q) = ∅
12        ⇒ addUsers(q ↦ u) = (U_q(q) ∪ u) ↦ S_q(q) ↦ UA_q(q) ↦
13            user_q(q) ↦ roles_q(q)
14     add_v: ∀ u, q ·
15            q ∈ STATE ∧
16            u ∈ ℙ(USER) ∧ u ∩ U_q(q) = ∅
17        ⇒ (U_q(q) ∪ u) ↦ S_q(q) ↦ UA_q(q) ↦ user_q(q) ↦
18            roles_q(q) ∈ STATE     theorem

19     del_t: deleteUsers ∈ STATE × ℙ(USER) → STATE
20     del_d: ∀ u, q ·
21            q ∈ STATE ∧
22            u ∈ ℙ(USER) ∧ u ⊆ U_q(q)
23        ⇒ deleteUsers(q ↦ u) = (U_q(q) \ u) ↦ S_q(q) ↦ (u ⩤ UA_q(q)) ↦
24            (user_q(q) ⩥ u) ↦ roles_q(q)
25     del_v: ∀ u, q ·
26            q ∈ STATE ∧ u ∈ ℙ(USER) ∧ u ⊆ U_q(q)
27        ⇒ (U_q(q) \ u) ↦ S_q(q) ↦ (u ⩤ UA_q(q)) ↦ (user_q(q) ⩥ u) ↦
28            roles_q(q) ∈ STATE     theorem

29     rev: ∀ u, q ·
30            q ∈ STATE ∧
31            u ∈ ℙ(USER) ∧ u ∩ U_q(q) = ∅
32        ⇒ deleteUsers(addUsers(q ↦ u) ↦ u) = q     theorem     > Re-
33        versibilität
34     com1: ∀ q, u1, u2 ·
35            q ∈ STATE ∧
36            u1 ∈ ℙ(USER) ∧ u1 ∩ U_q(q) = ∅ ∧
37            u2 ∈ ℙ(USER) ∧ u2 ∩ U_q(q) = ∅ ∧
38            u1 ∩ u2 = ∅
39        ⇒ addUsers(addUsers(q ↦ u1) ↦ u2) =
40            addUsers(addUsers(q ↦ u2) ↦ u1)     theorem     > Kommu-
41        tativität
42     com2: ∀ q, u1, u2 ·
43            q ∈ STATE ∧
44            u1 ∈ ℙ(USER) ∧ u1 ⊆ U_q(q) ∧
45            u2 ∈ ℙ(USER) ∧ u2 ⊆ U_q(q) ∧
46            u1 ∩ u2 = ∅
```

$\Rightarrow$ deleteUsers(deleteUsers(q $\mapsto$ u1) $\mapsto$ u2) =
deleteUsers(deleteUsers(q $\mapsto$ u2) $\mapsto$ u1) theorem
END

A.4 Kontext rbac_sessionhandling

CONTEXT rbac_sessionhandling
EXTENDS
rbac_state
CONSTANTS
createSessions
destroySessions
mapUserSessions
unmapUserSessions
AXIOMS
crt_t: createSessions $\in$ STATE $\times$ $\mathbb{P}$(SESSION) $\rightarrow$ STATE
crt_d: $\forall$ q, s $\cdot$
q $\in$ STATE $\wedge$
s $\in$ $\mathbb{P}$(SESSION) $\wedge$ finite(s) $\wedge$ s $\cap$ S_q(q) = $\varnothing$
$\Rightarrow$ createSessions(q $\mapsto$ s) = U_q(q) $\mapsto$ (S_q(q) $\cup$ s) $\mapsto$ UA_q(q) $\mapsto$
user_q(q) $\mapsto$ (roles_q(q) $\cup$ (s $\times$ {$\varnothing$}))
crt_v: $\forall$ q, s $\cdot$
q $\in$ STATE $\wedge$
s $\in$ $\mathbb{P}$(SESSION) $\wedge$ finite(s) $\wedge$ s $\cap$ S_q(q) = $\varnothing$
$\Rightarrow$ U_q(q) $\mapsto$ (S_q(q) $\cup$ s) $\mapsto$ UA_q(q) $\mapsto$ user_q(q) $\mapsto$
(roles_q(q) $\cup$ (s $\times$ {$\varnothing$})) $\in$ STATE theorem
dty_t: destroySessions $\in$ STATE $\times$ $\mathbb{P}$(SESSION) $\rightarrow$ STATE
dty_d: $\forall$ q, s $\cdot$
q $\in$ STATE $\wedge$
s $\subseteq$ S_q(q)
$\Rightarrow$ destroySessions(q $\mapsto$ s) = U_q(q) $\mapsto$ (S_q(q) $\setminus$ s) $\mapsto$ UA_q(q) $\mapsto$
(s $\lhd$ user_q(q)) $\mapsto$ (s $\lhd$ roles_q(q))
dty_v: $\forall$ q, s $\cdot$
q $\in$ STATE $\wedge$
s $\subseteq$ S_q(q)
$\Rightarrow$ U_q(q) $\mapsto$ (S_q(q) $\setminus$ s) $\mapsto$ UA_q(q) $\mapsto$ (s $\lhd$ user_q(q)) $\mapsto$
(s $\lhd$ roles_q(q)) $\in$ STATE theorem

rev1: $\forall$ q, s $\cdot$
q $\in$ STATE $\wedge$
s $\in$ $\mathbb{P}$(SESSION) $\wedge$ finite(s) $\wedge$ s $\cap$ S_q(q) = $\varnothing$
$\Rightarrow$ destroySessions(createSessions(q $\mapsto$ s) $\mapsto$ s) = q theorem

mp_t: mapUserSessions $\in$ STATE $\times$ (SESSION $\nrightarrow$ USER) $\rightarrow$ STATE

```
38      mp_d: ∀ q, su ·
39              q ∈ STATE ∧
40              su ∈ S_q(q) \ dom(user_q(q)) ⇸ U_q(q)
41              (∀ s · s ∈ dom(su) ⇒ roles_q(q)(s) ⊆ UA_q(q)[ {su(s)} ])
42          ⇒ mapUserSessions(q ↦ su) = U_q(q) ↦ S_q(q) ↦ UA_q(q) ↦
43              (user_q(q) ∪ su) ↦ roles_q(q)
44      mp_v: ∀ q, su ·
45              q ∈ STATE ∧
46              su ∈ S_q(q) \ dom(user_q(q)) ⇸ U_q(q)
47              (∀ s · s ∈ dom(su) ⇒ roles_q(q)(s) ⊆ UA_q(q)[ {su(s)} ])
48          ⇒ U_q(q) ↦ S_q(q) ↦ UA_q(q) ↦ (user_q(q) ∪ su) ↦
49              roles_q(q) ∈ STATE     theorem
50      ump_t: unmapUserSessions ∈ STATE × (SESSION ⇸ USER) → STATE
51      ump_d: ∀ q, su ·
52              q ∈ STATE ∧
53              su ⊆ user_q(q)
54          ⇒ unmapUserSessions(q ↦ su) = U_q(q) ↦ S_q(q) ↦ UA_q(q) ↦
55              (user_q(q) \ su) ↦ roles_q(q)
56      ump_v: ∀ q, su ·
57              q ∈ STATE ∧
58              su ⊆ user_q(q)
59          ⇒ U_q(q) ↦ S_q(q) ↦ UA_q(q) ↦ (user_q(q) \ su) ↦ roles_q(q)
60              ∈ STATE     theorem

61      rev2: ∀ q, su ·
62              q ∈ STATE ∧
63              su ∈ S_q(q) \ dom(user_q(q)) ⇸ U_q(q)
64              (∀ s · s ∈ dom(su) ⇒ roles_q(q)(s) ⊆ UA_q(q)[ {su(s)} ])
65          ⇒ unmapUserSessions(mapUserSessions(q ↦ su) ↦ su) = q
66          theorem
67  END
```

A.5 Kontext rbac_rolehandling

```
1   CONTEXT rbac_rolehandling
2   EXTENDS
3      rbac_state
4   CONSTANTS
5      assignRolesToUsers
6      revokeRolesFromUsers
7      activateRoles
8      deactivateRoles
9   AXIOMS
10     aur_t: assignRolesToUsers ∈ STATE × ℙ(USER × ROLE) → STATE
```

```
11      aur_d: ∀ q, ua ·
12              q ∈ STATE ∧
13              ua ⊆ (U_q(q) × ROLE) \ UA_q(q)
14          ⇒ assignRolesToUsers(q ↦ ua) = U_q(q) ↦ S_q(q) ↦
15              (UA_q(q) ∪ ua) ↦ user_q(q) ↦ roles_q(q)
16      aur_v: ∀ q, ua ·
17              q ∈ STATE ∧
18              ua ⊆ (U_q(q) × ROLE) \ UA_q(q)
19          ⇒ U_q(q) ↦ S_q(q) ↦ (UA_q(q) ∪ ua) ↦ user_q(q) ↦ roles_q(q)
20              ∈ STATE     theorem
21      rur_t: revokeRolesFromUsers ∈ STATE × ℙ(USER × ROLE) → STATE
22      rur_d: ∀ q, ua ·
23              q ∈ STATE ∧
24              ua ⊆ UA_q(q) ∧
25              (∀ u · u ∈ dom(ua) ⇒ ua[ {u} ] ∩
26                  ( ⋃ r0 · r0 ∈ roles_q(q)[user_q(q)^-1[ {u} ]] | r0 ) = ∅
27              )
28          ⇒ revokeRolesFromUsers(q ↦ ua) = U_q(q) ↦ S_q(q) ↦
29              (UA_q(q) \ ua) ↦ user_q(q) ↦ roles_q(q)
30      rur_v: ∀ q, ua ·
31              q ∈ STATE ∧
32              ua ⊆ UA_q(q) ∧
33              (∀ u · u ∈ dom(ua) ⇒ ua[ {u} ] ∩
34                  ( ⋃ r0 · r0 ∈ roles_q(q)[user_q(q)^-1[ {u} ]] | r0 ) = ∅
35              )
36          ⇒ U_q(q) ↦ S_q(q) ↦ (UA_q(q) \ ua) ↦ user_q(q) ↦ roles_q(q)
37              ∈ STATE     theorem

38      rev1: ∀ q, ua ·
39              q ∈ STATE ∧
40              ua ⊆ (U_q(q) × ROLE) \ UA_q(q)
41          ⇒ revokeRolesFromUsers(assignRolesToUsers(q ↦ ua) ↦ ua) = q
42          theorem

43      acr_t: activateRoles ∈ STATE × ℙ(SESSION × ROLE) → STATE
44      acr_d: ∀ q, sr ·
45              q ∈ STATE ∧
46              sr ⊆ user_q(q);UA_q(q) ∧ finite(sr) ∧
47              (∀ s · s ∈ dom(user_q(q)) ⇒ roles_q(q)(s) ∩ sr[{s}] = ∅)
48          ⇒ activateRoles(q ↦ sr) = U_q(q) ↦ S_q(q) ↦ UA_q(q) ↦
49              user_q(q) ↦
50              {s ↦ r |
51                  s ∈ dom(roles_q(q)) ∧
52                  r = (roles_q(q)(s) ∪ sr[ {s} ∩ dom(user_q(q)) ])
53              }
54      acr_v: ∀ q, sr ·
```

```
55              q ∈ STATE ∧
56              sr ⊆ user_q(q); UA_q(q) ∧ finite(sr) ∧
57              (∀ s · s ∈ dom(user_q(q)) ⇒ roles_q(q)(s) ∩ sr[{s}] = ∅)
58          ⇒ U_q(q) ↦ S_q(q) ↦ UA_q(q) ↦ user_q(q) ↦
59              {s ↦ r |
60                  s ∈ dom(roles_q(q)) ∧
61                  r = (roles_q(q)(s) ∪ sr[ {s} ∩ dom(user_q(q)) ])
62              } ∈ STATE     theorem
63      dar_t: deactivateRoles ∈ STATE × ℙ(SESSION × ROLE) → STATE
64      dar_d: ∀ q, sr ·
65              q ∈ STATE ∧
66              sr ⊆ {s ↦ r | s ∈ dom(roles_q(q)) ∧ r ∈ roles_q(q)(s)}
67          ⇒ deactivateRoles(q ↦ sr) = U_q(q) ↦ S_q(q) ↦ UA_q(q) ↦
68              user_q(q) ↦
69              {s ↦ r |
70                  s ∈ dom(roles_q(q)) ∧
71                  r = (roles_q(q)(s) \ sr[ {s} ])
72              }
73      dar_v: ∀ q, sr ·
74              q ∈ STATE ∧
75              sr ⊆ {s ↦ r | s ∈ dom(roles_q(q)) ∧ r ∈ roles_q(q)(s)}
76          ⇒ U_q(q) ↦ S_q(q) ↦ UA_q(q) ↦ user_q(q) ↦
77              {s ↦ r |
78                  s ∈ dom(roles_q(q)) ∧
79                  r = (roles_q(q)(s) \ sr[ {s} ])
80              } ∈ STATE     theorem

81      rev2: ∀ q, sr ·
82              q ∈ STATE ∧
83              sr ⊆ user_q(q); UA_q(q) ∧ finite(sr) ∧
84              (∀ s · s ∈ dom(user_q(q)) ⇒ roles_q(q)(s) ∩ sr[{s}] = ∅)
85          ⇒ deactivateRoles(activateRoles(q ↦ sr) ↦ sr) = q
86          theorem
87  END
```

A.6 Kontext `rbac_predicate`

```
1   CONTEXT rbac_predicate
2   EXTENDS
3       rbac_state
4   CONSTANTS
5       access_UR     > Nutzer hat Rolle?
6       access_UM     > Nutzer hat Berechtigung?
7       access_SR     > Sitzung hat Rolle?
8       access_SM     > Sitzung hat Berechtigung?
```

```
9       sod     > Rollenausschluss?
10    AXIOMS
11      aUR_t: access_UR ∈ STATE × USER × ROLE → BOOL
12      aUR_d: ∀ q, u, r ·
13              q ∈ STATE ∧
14              u ∈ USER ∧
15              r ∈ ROLE
16          ⇒ access_UR(q ↦ u ↦ r) = bool(
17              (∃ r1 · r1 ∈ ROLE ∧
18                  (r1 ↦ r) ∈ RH ∧
19                  (u ↦ r1) ∈ UA_q(q)
20              )
21          )

22      aUM_t: access_UM ∈ STATE × USER × OBJECT × OPERATION → BOOL
23      aUM_d: ∀ q, u, o, op ·
24              q ∈ STATE ∧
25              u ∈ USER ∧
26              o ∈ OBJECT ∧ op ∈ OPERATION
27          ⇒ access_UM(q ↦ u ↦ o ↦ op) = bool(
28              (∃ r · r ∈ ROLE ∧ (r ↦ o) ∈ dom(M) ∧
29                  op ∈ M(r ↦ o) ∧
30                  access_UR(q ↦ u ↦ r) = TRUE
31              )
32          )

33      aSR_t: access_SR ∈ STATE × SESSION × ROLE → BOOL
34      aSR_d: ∀ q, s, r ·
35              q ∈ STATE ∧
36              s ∈ SESSION ∧
37              r ∈ ROLE
38          ⇒ access_SR(q ↦ s ↦ r) = bool(
39              s ∈ dom(roles_q(q)) ∧
40              (∃ r1 · r1 ∈ ROLE ∧
41                  (r1 ↦ r) ∈ RH ∧
42                  r1 ∈ roles_q(q)(s)
43              )
44          )

45      aSM_t: access_SM ∈ STATE × SESSION × OBJECT × OPERATION → BOOL
46      aSM_d: ∀ q, s, o, op ·
47              q ∈ STATE ∧
48              s ∈ SESSION ∧
49              o ∈ OBJECT ∧ op ∈ OPERATION
50          ⇒ access_SM(q ↦ s ↦ o ↦ op) = bool(
51              (∃ r · r ∈ ROLE ∧ (r ↦ o) ∈ dom(M) ∧
```

```
52                  op ∈ M(r ↦ o) ∧
53                  access_SR(q ↦ s ↦ r) = TRUE
54              )
55          )

56      sod_t: sod ∈ STATE × USER × ROLE → BOOL
57      sod_d: ∀ q, u, r ·
58              q ∈ STATE ∧
59              u ∈ USER ∧
60              r ∈ ROLE
61          ⇒ sod(q ↦ u ↦ r) = bool(
62              (∀ r1 · r1 ∈ ROLE ∧
63                  (u ↦ r1) ∈ UA_q(q) ⇒ ¬((r ↦ r1) ∈ RE)
64              )
65          )
66  END
```

A.7 Kontext `rbac`

```
1   CONTEXT rbac
2   EXTENDS
3      rbac_userhandling
4      rbac_sessionhandling
5      rbac_rolehandling
6      rbac_predicate
7   END
```

A.8 Kontext `healthcare_context`

```
1   CONTEXT healthcare_context
2   EXTENDS
3      rbac
4   CONSTANTS
5      Employee     > Rollen
6      Manager
7      Doctor
8      Nurse
9      Receptionist
10     Patient
11     MedicalManager
12     MedicalTeam
13     ReferredDoctor
14     UserAdmin
```

```
15      view     > Operationen
16      add
17      modify
18      access
19      enter
20      create
21      update
22      sign

23      OldMedicalRecords     > Objekte
24      RecentMedicalRecords
25      PrivateNotes
26      Prescriptions
27      PatientPersonalInfo
28      PatientFinancialInfo
29      PatientMedicalInfo
30      CarePlan
31      Appointment
32      ProgressNotes
33      LegalAgreement
34      Bills
35      Uo
36      UAo

37      u1     > Objekte des Initialzustands

38      RH_base     > Zur Konstruktion benötigt
39      RE_base
40  AXIOMS
41      axm1: partition(ROLE,
42              {Employee}, {Manager}, {Doctor}, {Nurse}, {Receptionist},
43              {Patient}, {MedicalManager}, {MedicalTeam},
44              {ReferredDoctor}, {UserAdmin}
45          )
46      axm2: partition(OBJECT,
47              {OldMedicalRecords}, {RecentMedicalRecords},
48              {PrivateNotes}, {Prescriptions}, {PatientPersonalInfo},
49              {PatientFinancialInfo}, {PatientMedicalInfo}, {CarePlan},
50              {Appointment}, {ProgressNotes}, {LegalAgreement}, {Bills},
51              {Uo}, {UAo}
52          )
```

```
53      axm3: partition(OPERATION,
54              {view}, {add}, {modify}, {access}, {enter}, {create},
55              {update}, {sign}
56          )
57      axm4: partition(USER, {u1})

58      axm5: RH_base = {
59              Doctor ↦ Nurse,
60              Nurse ↦ Employee,
61              Manager ↦ MedicalManager,
62              MedicalManager ↦ Receptionist,
63              Receptionist ↦ Employee
64          } ∪ id
65      axm6: RH = RH_base; RH_base; RH_base
66      axm7: Manager ↦ Employee ∈ RH ∧
67              Manager ↦ Manager ∈ RH     theorem

68      axm8: RE_base = {
69              Patient ↦ Employee,
70              Patient ↦ ReferredDoctor,
71              Patient ↦ UserAdmin,
72              Doctor ↦ Manager,
73              Doctor ↦ Receptionist
74          }
75      axm9: RE = RE_base ∪ RE_base^−1

76      axm10: M = {
77              Doctor ↦ OldMedicalRecords ↦ {view},
78              Patient ↦ OldMedicalRecords ↦ {view},
79              Manager ↦ OldMedicalRecords ↦ {enter},
80              Nurse ↦ OldMedicalRecords ↦ {access},
81              Doctor ↦ RecentMedicalRecords ↦ {view, add},
82              Nurse ↦ RecentMedicalRecords ↦ {view},
83              Patient ↦ RecentMedicalRecords ↦ {view},
84              Manager ↦ RecentMedicalRecords ↦ {enter},
85              Doctor ↦ PrivateNotes ↦ {view, add},
86              Doctor ↦ Prescriptions ↦ {view, modify},
87              Patient ↦ Prescriptions ↦ {view},
88              Manager ↦ PatientPersonalInfo ↦ {access},
89              Manager ↦ PatientFinancialInfo ↦ {access},
90              Manager ↦ PatientMedicalInfo ↦ {access},
91              Manager ↦ CarePlan ↦ {update},
92              Nurse ↦ CarePlan ↦ {view},
93              Receptionist ↦ Appointment ↦ {create},
94              Nurse ↦ ProgressNotes ↦ {add},
95              Patient ↦ LegalAgreement ↦ {sign},
```

```
96              Patient ↦ Bills ↦ {view},
97              UserAdmin ↦ Uo ↦ {update},
98              UserAdmin ↦ UAo ↦ {update}
99          }
100 END
```

A.9 Kontext `healthcare_generic`

```
1   MACHINE healthcare_generic
2   SEES healthcare_context
3   VARIABLES
4      state
5   INVARIANTS
6      inv1: state ∈ CONSISTENT_STATE
7      inv2: state ∈ STATE     theorem
8   EVENTS
9   INITIALISATION ≙
10     begin
11        act1: state := {u1} ↦ ∅ ↦ {u1 ↦ UserAdmin} ↦ ∅ ↦ ∅
12     end
13  createUser ≙
14     any
15        s     > handelnde Sitzung
16        u     > anzulegender Nutzer
17     where
18        typ1: s ∈ SESSION
19        typ2: u ∈ USER \ U_q(state)
20        cnd1: access_SM(state ↦ s ↦ Uo ↦ update) = TRUE
21     then
22        act1: state := addUsers(state ↦ {u})
23     end
24  destroyUser ≙
25     any
26        s     > handelnde Sitzung
27        u     > zu löschender Nutzer
28     where
29        typ1: s ∈ SESSION
30        typ2: u ∈ U_q(state)
31        cnd1: access_SM(state ↦ s ↦ Uo ↦ update) = TRUE
32     then
33        act1: state := deleteUsers(
34                 destroySessions(
35                     state ↦
36                     dom(user_q(state) ▷ {u})
```

```
                ) ↦
                {u}
            )
    end
login ≙
    any
        s     > zu erstellende Sitzung
        u     > zugeordneter Nutzer
    where
        typ1: s ∈ SESSION \ S_q(state)
        typ2: u ∈ U_q(state)
    then
        act1: state := mapUserSessions(
                createSessions(state ↦ {s}) ↦
                {s ↦ u}
            )
    end
logout ≙
    any
        s     > sich abmeldende Sitzung
    where
        typ1: s ∈ S_q(state)
    then
        act1: state := destroySessions(
                unmapUserSessions(
                    state ↦
                    {s ↦ user_q(state)(s)}
                ) ↦
                {s}
            )
    end
activateRole ≙
    any
        s     > handelnde Sitzung
        r     > zu aktivierende Rolle
    where
        typ1: s ∈ S_q(state)
        typ2: r ∈ ROLE \ roles_q(state)(s)
        cnd1: r ∈ (user_q(state);UA_q(state))[{s}]     > Nutzer darf r akti-
            vieren
    then
        act1: state := activateRoles( state ↦ {s ↦ r} )
    end
deactivateRole ≙
```

```
81        any
82           s     > handelnde Sitzung
83           r     > zu deaktivierende Rolle
84        where
85           typ1: s ∈ S_q(state)
86           typ2: r ∈ roles_q(state)(s)
87        then
88           act1: state := deactivateRoles( state ↦ {s ↦ r} )
89        end
90     assignGenericRole ≙
91        any
92           u     > Nutzer
93           r     > dem Nutzer zuzuweisende Rolle
94        where
95           typ1: u ∈ U_q(state)
96           typ2: r ∈ ROLE \ UA_q(state)[{u}]
97           cnd1: sod(state ↦ u ↦ r) = TRUE
98        then
99           act1: state := assignRolesToUsers( state ↦ {u ↦ r} )
100       end
101    revokeGenericRole ≙
102       any
103          u     > Nutzer
104          r     > dem Nutzer zu entziehende Rolle
105       where
106          typ1: u ∈ U_q(state)
107          typ2: r ∈ UA_q(state)[{u}]
108       then
109          act1: state := revokeRolesFromUsers(
110                  deactivateRoles(
111                      state ↦
112                      { s0 · s0 ∈ user_q(state)^-1[{u}] ∧
113                          r ∈ roles_q(state)(s0) | s0 ↦ r}
114                  ) ↦
115                  {u ↦ r}
116              )
117       end
118    END
```

A.10 Kontext `healthcare`

```
1     MACHINE healthcare
2     REFINES healthcare_generic
3     SEES healthcare_context
```

```
4    VARIABLES
5       state     >  es gelten die Invarianten aus healthcare_generic
6    EVENTS
7    INITIALISATION ≙
8       extends
9          INITIALISATION
10      begin
11         act1: state := {u1} ↦ ∅ ↦ {u1 ↦ UserAdmin} ↦ ∅ ↦ ∅
12      end
13   createUser ≙
14      extends
15         createUser
16      any
17         s
18         u
19      where
20         typ1: s ∈ SESSION
21         typ2: u ∈ USER \ U_q(state)
22         cnd1: access_SM(state ↦ s ↦ Uo ↦ update) = TRUE
23      then
24         act1: state := addUsers(state ↦ {u})
25      end
26   destroyUser ≙
27      extends
28         destroyUser
29      any
30         s
31         u
32      where
33         typ1: s ∈ SESSION
34         typ2: u ∈ U_q(state)
35         cnd1: access_SM(state ↦ s ↦ Uo ↦ update) = TRUE
36      then
37         act1: state := deleteUsers(
38                  destroySessions(
39                      state ↦ dom(user_q(state) ▷ {u})
40                  ) ↦ {u}
41               )
42      end

43   assignRole ≙
44      extends
45         assignGenericRole
46      any
```

```
47          s     >  handelnde Sitzung
48          u     >  Zielnutzer
49          r     >  zuzuweisende Rolle
50       where
51          typ1: s ∈ S_q(state)
52          typ2: u ∈ U_q(state)
53          typ3: r ∈ ROLE \ UA_q(state)[{u}]
54          cnd1: sod(state ↦ u ↦ r) = TRUE
55          cnd2: access_SM(state ↦ s ↦ UAo ↦ update) = TRUE
56       then
57          act1: state := assignRolesToUsers( state ↦ {u ↦ r} )
58       end
59    revokeRole ≙
60       extends
61          revokeGenericRole
62       any
63          s     >  handelnde Sitzung
64          u     >  Zielnutzer
65          r     >  zu entziehende Rolle
66       where
67          typ1: s ∈ S_q(state)
68          typ2: u ∈ U_q(state)
69          typ3: r ∈ UA_q(state)[{u}]
70          cnd1: access_SM(state ↦ s ↦ UAo ↦ update) = TRUE
71       then
72          act1: state := revokeRolesFromUsers(
73                  deactivateRoles(
74                      state ↦
75                      { s0 · s0 ∈ user_q(state)^-1[{u}] ∧
76                          r ∈ roles_q(state)(s0) | s0 ↦ r}
77                  ) ↦
78                  {u ↦ r}
79              )
80       end

81    login ≙
82       extends
83          login
84       any
85          s
86          u
87       where
88          typ1: s ∈ SESSION \ S_q(state)
89          typ2: u ∈ U_q(state)
90       then
```

```
91          act1: state := mapUserSessions(
92                  createSessions(state ↦ {s}) ↦
93                  {s ↦ u}
94              )
95      end
96  logout ≙
97      extends
98          logout
99      any
100         s
101     where
102         typ1: s ∈ S_q(state)
103     then
104         act1: state := destroySessions(
105                 unmapUserSessions(
106                     state ↦
107                     {s ↦ user_q(state)(s)}
108                 ) ↦ {s}
109             )
110     end
111 activateRole ≙
112     extends
113         activateRole
114     any
115         s
116         r
117     where
118         typ1: s ∈ S_q(state)
119         typ2: r ∈ ROLE \ roles_q(state)(s)
120         cnd1: r ∈ (user_q(state);UA_q(state))[{s}]
121     then
122         act1: state := activateRoles(state ↦ {s ↦ r})
123     end
124 deactivateRole ≙
125     extends
126         deactivateRole
127     any
128         s
129         r
130     where
131         typ1: s ∈ S_q(state)
132         typ2: r ∈ roles_q(state)(s)
133     then
```

```
134         act1: state := deactivateRoles(state ↦ {s ↦ r})
135      end

136   assignReferredDoctorRole ≙
137      refines
138         assignGenericRole
139      any
140         s    > handelnde Sitzung
141         u    > Zielnutzer
142      where
143         typ1: s ∈ S_q(state)
144         typ2: u ∈ U_q(state)
145         cnd1: sod(state ↦ u ↦ ReferredDoctor) = TRUE
146         cnd2: access_SR(state ↦ s ↦ Doctor) = TRUE
147         cnd3: access_UR(state ↦ u ↦ Doctor) = TRUE
148         cnd4: ReferredDoctor ∉ UA_q(state)[{u}]
149      with
150         r: r = ReferredDoctor
151      then
152         act1: state := assignRolesToUsers(
153                  state ↦ {u ↦ ReferredDoctor}
154               )
155      end
156   revokeReferredDoctorRole ≙
157      refines
158         revokeGenericRole
159      any
160         s    > handelnde Sitzung
161         u    > Zielnutzer
162      where
163         typ1: s ∈ S_q(state)
164         typ2: u ∈ U_q(state)
165         cnd1: access_SR(state ↦ s ↦ Doctor) = TRUE
166         cnd2: ReferredDoctor ∈ UA_q(state)[{u}]
167      with
168         r: r = ReferredDoctor
169      then
170         act1: state := revokeRolesFromUsers(
171                  deactivateRoles(
172                     state ↦
173                     {s0 · s0 ∈ user_q(state)^-1[{u}] ∧
174                        ReferredDoctor ∈ roles_q(state)(s0)
175                         | s0 ↦ ReferredDoctor}
176                  ) ↦
```

```
177                 {u ↦ ReferredDoctor}
178             )
179     end

180   assignPatientRole ≙
181     refines
182       assignGenericRole
183     any
184       s     >  handelnde Sitzung
185       u     >  Zielnutzer
186     where
187       typ1: s ∈ S_q(state)
188       typ2: u ∈ U_q(state)
189       cnd1: sod(state ↦ u ↦ Patient) = TRUE
190       cnd2: access_SR(state ↦ s ↦ Receptionist) = TRUE
191       cnd4: Patient ∉ UA_q(state)[{u}]
192     with
193       r: r = Patient
194     then
195       act1: state := assignRolesToUsers(
196               state ↦ {u ↦ Patient}
197             )
198     end
199   revokePatientRole ≙
200     refines
201       revokeGenericRole
202     any
203       s     >  handelnde Sitzung
204       u     >  Zielnutzer
205     where
206       typ1: s ∈ S_q(state)
207       typ2: u ∈ U_q(state)
208       cnd1: access_SR(state ↦ s ↦ Receptionist) = TRUE
209       cnd2: Patient ∈ UA_q(state)[{u}]
210     with
211       r: r = Patient
212     then
213       act1: state := revokeRolesFromUsers(
214               deactivateRoles(
215                   state ↦
216                   {s0 · s0 ∈ user_q(state)^-1[{u}] ∧
217                       Patient ∈ roles_q(state)(s0)
218                       | s0 ↦ Patient}
219               ) ↦
```

```
220                 {u ↦ Patient}
221             )
222     end

223   assignMedicalTeamRole ≙
224     refines
225       assignGenericRole
226     any
227       s    > handelnde Sitzung
228       u    > Zielnutzer
229     where
230       typ1: s ∈ S_q(state)
231       typ2: u ∈ U_q(state)
232       cnd1: sod(state ↦ u ↦ MedicalTeam) = TRUE
233       cnd2: access_SR(state ↦ s ↦ MedicalManager) = TRUE
234       cnd3: access_UR(state ↦ u ↦ Doctor) = TRUE ∨
235               access_UR(state ↦ u ↦ Nurse) = TRUE
236       cnd4: MedicalTeam ∉ UA_q(state)[{u}]
237     with
238       r: r = MedicalTeam
239     then
240       act1: state := assignRolesToUsers(
241               state ↦ {u ↦ MedicalTeam}
242           )
243     end
244   revokeMedicalTeamRole ≙
245     refines
246       revokeGenericRole
247     any
248       s    > handelnde Sitzung
249       u    > Zielnutzer
250     where
251       typ1: s ∈ S_q(state)
252       typ2: u ∈ U_q(state)
253       cnd1: access_SR(state ↦ s ↦ MedicalManager) = TRUE
254       cnd2: MedicalTeam ∈ UA_q(state)[{u}]
255     with
256       r: r = MedicalTeam
257     then
258       act1: state := revokeRolesFromUsers(
259               deactivateRoles(
260                   state ↦
261                   {s0 · s0 ∈ user_q(state)^-1[{u}] ∧
262                       MedicalTeam ∈ roles_q(state)(s0)
263                       | s0 ↦ MedicalTeam}
```

```
264                 ) ↦
265                 {u ↦ MedicalTeam}
266             )
267     end
268 END
```

B Verfeinerung der Datentypen in der Spezifikation des Beispielmodells

Ergänzend zu Anhang A stellt der vorliegende ne Spezifikation vor, welche beginnt, den kompakten Zustandsraum aufzulösen und beispielhaft für die Menge der Benutzer U eine eigene Zustandskomponente einführt. Aufgrund der Techniken der Verfeinerung (vgl. Unterabschnitt 2.6.2) ist die Korrektheit der so entstandenen Spezifikation gegenüber der Vorgabe durch das Metamodell beweisbar.

Nachfolgend wird dazu in Anhang B.1 der Kontext `rbac_userset` vorgestellt, welcher die Menge `USER_SET` aller möglichen Benutzermengen deklariert und darauf die Operationen `insert_U` und `delete_U` zum Einfügen und Löschen von Elementen aus einer Benutzermenge definiert.

`USER_SET` kann dann als Datentyp genutzt werden, wie Anhang B.2 anhand der Zustandsmaschine `healthcare_userset` demonstriert, welche `healthcare` verfeinert. Sie enthält die Zustandskomponente `U` $\in$ `USER_SET`, welche per Invariante stets äquivalent zur entsprechenden Komponente `U_q(state)` des Zustandsraumes ist.

Gleichermaßen kann in weiteren Verfeinerungen mit allen Zustandskomponenten so verfahren werden, sodass eine letzte Verfeinerung die Zustandskomponente `state` sowie die Primitivoperationen gänzlich entfernen kann. Es liegt dann eine Spezifikation vor, welche ausschließlich auf Datentypen und Operationen beruht, welche sich auch in typischen Programmiersprachen finden lassen. Dies erleichtert eine Überführung in Quelltext wesentlich und trägt so zu einer korrekten Implementierung bei. Ein Beispiel ist in Anhang B.3 dokumentiert, welches jedoch nur auf Verfeinerung der Nutzer basiert und die Primitivprädikate unbeachtet lässt.

B.1 Kontext `rbac_userset`

```
CONTEXT rbac_userset
EXTENDS
   rbac_static
CONSTANTS
   USER_SET
   insert_U
   delete_U
AXIOMS
```

axm1: $\mathtt{USER\_SET} = \{\, \mathtt{U} \mid \mathtt{U} \in \mathbb{P}(\mathtt{USER}) \,\}$

axm2: $\mathtt{insert\_U} \in \mathtt{USER\_SET} \times \mathtt{USER} \rightarrow \mathtt{USER\_SET}$

axm3: $\forall \mathtt{U}, \mathtt{u} \cdot \mathtt{U} \in \mathtt{USER\_SET} \land \mathtt{u} \in \mathtt{USER} \Rightarrow \mathtt{insert\_U}(\mathtt{U} \mapsto \mathtt{u}) = \mathtt{U} \cup \{\mathtt{u}\}$

axm4: $\mathtt{delete\_U} \in \mathtt{USER\_SET} \times \mathtt{USER} \rightarrow \mathtt{USER\_SET}$

axm5: $\forall \mathtt{U}, \mathtt{u} \cdot \mathtt{U} \in \mathtt{USER\_SET} \land \mathtt{u} \in \mathtt{USER} \Rightarrow \mathtt{delete\_U}(\mathtt{U} \mapsto \mathtt{u}) = \mathtt{U} \setminus \{\mathtt{u}\}$

END

B.2 Kontext `healthcare_userset`

MACHINE healthcare_userset

REFINES healthcare

SEES healthcare_context, rbac_userset

VARIABLES

state

U

INVARIANTS

inv1: $\mathtt{U} \in \mathtt{USER\_SET}$

inv2: $\mathtt{U} = \mathtt{U\_q}(\mathtt{state})$

EVENTS

INITIALISATION $\widehat{=}$

extends

INITIALISATION

begin

act1: $\mathtt{state} := \{\mathtt{u1}\} \mapsto \varnothing \mapsto \{\mathtt{u1} \mapsto \mathtt{UserAdmin}\} \mapsto \varnothing \mapsto \varnothing$

act2: $\mathtt{U} := \{\mathtt{u1}\}$

end

createUser $\widehat{=}$

extends

createUser

any

s > Session

u > User

where

typ1: $\mathtt{s} \in \mathtt{SESSION}$

typ2: $\mathtt{u} \in \mathtt{USER} \setminus \mathtt{U\_q}(\mathtt{state})$

cnd1: $\mathtt{access\_SM}(\mathtt{state} \mapsto \mathtt{s} \mapsto \mathtt{Uo} \mapsto \mathtt{update}) = \mathtt{TRUE}$

typ3: $\mathtt{u} \in \mathtt{USER} \setminus \mathtt{U}$

then

act1: $\mathtt{state} := \mathtt{addUsers}(\mathtt{state} \mapsto \{\mathtt{u}\})$

act2: $\mathtt{U} := \mathtt{insert\_U}(\mathtt{U} \mapsto \mathtt{u})$

end

```
destroyUser ≙
  extends
    destroyUser
  any
    s
    u
  where
    typ1: s ∈ SESSION
    typ2: u ∈ U_q(state)
    typ3: u ∈ U
    cnd1: access_SM(state ↦ s ↦ Uo ↦ update) = TRUE
  then
    act1: state := deleteUsers(destroySessions(state ↦
        dom(user_q(state) ▷ {u})) ↦ {u})
    act2: U := delete_U(U ↦ u)
  end
assignRole ≙
  extends
    assignRole
  any
    u
    r
    s
  where
    typ1: s ∈ S_q(state)
    typ2: u ∈ U_q(state)
    typ3: r ∈ ROLE \ UA_q(state)[{u}]
    typ4: u ∈ U
    cnd1: sod(state ↦ u ↦ r) = TRUE
    cnd2: access_SM(state ↦ s ↦ UAo ↦ update) = TRUE
  then
    act1: state := assignRolesToUsers(state ↦ {u ↦ r})
  end
revokeRole ≙
  extends
    revokeRole
  any
    u
    r
    s
  where
    typ1: s ∈ S_q(state)
    typ2: u ∈ U_q(state)
    typ3: r ∈ UA_q(state)[{u}]
```

```
      typ4: u ∈ U
      cnd1: access_SM(state ↦ s ↦ UAo ↦ update) = TRUE
   then
      act1: state := revokeRolesFromUsers(deactivateRoles(state ↦
          {s0 ↦ r0|s0 ∈ user_q(state)^-1[{u}] ∧ r0 ∈ roles_q(state)(s0) ∧
          r0 = r}) ↦ {u ↦ r})
   end
login ≙
   extends
      login
   any
      s
      u
   where
      typ1: s ∈ SESSION \ S_q(state)
      typ2: u ∈ U_q(state)
      typ3: u ∈ U
   then
      act1: state := mapUserSessions(createSessions(state ↦ {s}) ↦
          {s ↦ u})
   end
logout ≙
   extends
      logout
   any
      s
   where
      typ1: s ∈ S_q(state)
   then
      act1: state := destroySessions(unmapUserSessions(state ↦
          {s ↦ user_q(state)(s)}) ↦ {s})
   end
activateRole ≙
   extends
      activateRole
   any
      s
      r
   where
      typ1: s ∈ S_q(state)
      typ2: r ∈ ROLE \ roles_q(state)(s)
      cnd1: r ∈ (user_q(state); UA_q(state))[{s}]
   then
      act1: state := activateRoles(state ↦ {s ↦ r})
```

```
    end
  deactivateRole ≙
    extends
      deactivateRole
    any
      s
      r
    where
      typ1: s ∈ S_q(state)
      typ2: r ∈ roles_q(state)(s)
    then
      act1: state := deactivateRoles(state ↦ {s ↦ r})
    end
  assignReferredDoctorRole ≙
    extends
      assignReferredDoctorRole
    any
      s
      u
    where
      typ1: s ∈ S_q(state)
      typ2: u ∈ U_q(state)
      typ3: u ∈ U
      cnd1: sod(state ↦ u ↦ ReferredDoctor) = TRUE
      cnd2: access_SR(state ↦ s ↦ Doctor) = TRUE
      cnd3: access_UR(state ↦ u ↦ Doctor) = TRUE
      cnd4: ReferredDoctor ∉ UA_q(state)[{u}]
    then
      act1: state := assignRolesToUsers(state ↦ {u ↦
          ReferredDoctor})
    end
  revokeReferredDoctorRole ≙
    extends
      revokeReferredDoctorRole
    any
      s
      u
    where
      typ1: s ∈ S_q(state)
      typ2: u ∈ U_q(state)
      typ3: u ∈ U
      cnd1: access_SR(state ↦ s ↦ Doctor) = TRUE
      cnd2: ReferredDoctor ∈ UA_q(state)[{u}]
    then
```

```
      act1: state := revokeRolesFromUsers(deactivateRoles(state ↦
          {s0 · s0 ∈ user_q(state)^-1[{u}] ∧ ReferredDoctor ∈
          roles_q(state)(s0)|s0 ↦ ReferredDoctor}) ↦ {u ↦
          ReferredDoctor})
   end
assignPatientRole ≙
   extends
      assignPatientRole
   any
      s
      u
   where
      typ1: s ∈ S_q(state)
      typ2: u ∈ U_q(state)
      typ3: u ∈ U
      cnd1: sod(state ↦ u ↦ Patient) = TRUE
      cnd2: access_SR(state ↦ s ↦ Receptionist) = TRUE
      cnd4: Patient ∉ UA_q(state)[{u}]
   then
      act1: state := assignRolesToUsers(state ↦ {u ↦ Patient})
   end
revokePatientRole ≙
   extends
      revokePatientRole
   any
      s
      u
   where
      typ1: s ∈ S_q(state)
      typ2: u ∈ U_q(state)
      typ3: u ∈ U
      cnd1: access_SR(state ↦ s ↦ Receptionist) = TRUE
      cnd2: Patient ∈ UA_q(state)[{u}]
   then
      act1: state := revokeRolesFromUsers(deactivateRoles(state ↦
          {s0 · s0 ∈ user_q(state)^-1[{u}] ∧ Patient ∈
          roles_q(state)(s0)|s0 ↦ Patient}) ↦ {u ↦ Patient})
   end
assignMedicalTeamRole ≙
   extends
      assignMedicalTeamRole
   any
      s
      u
```

```
    where
      typ1: s ∈ S_q(state)
      typ2: u ∈ U_q(state)
      typ3: u ∈ U
      cnd1: sod(state ↦ u ↦ MedicalTeam) = TRUE
      cnd2: access_SR(state ↦ s ↦ MedicalManager) = TRUE
      cnd3: access_UR(state ↦ u ↦ Doctor) = TRUE∨access_UR(state ↦
          u ↦ Nurse) = TRUE
      cnd4: MedicalTeam ∉ UA_q(state)[{u}]
    then
      act1: state := assignRolesToUsers(state ↦ {u ↦ MedicalTeam})
    end
  revokeMedicalTeamRole ≙
    extends
      revokeMedicalTeamRole
    any
      s
      u
    where
      typ1: s ∈ S_q(state)
      typ2: u ∈ U_q(state)
      typ3: u ∈ U
      cnd1: access_SR(state ↦ s ↦ MedicalManager) = TRUE
      cnd2: MedicalTeam ∈ UA_q(state)[{u}]
    then
      act1: state := revokeRolesFromUsers(deactivateRoles(state ↦
          {s0 · s0 ∈ user_q(state)^−1[{u}] ∧ MedicalTeam ∈
          roles_q(state)(s0)|s0 ↦ MedicalTeam}) ↦ {u ↦
          MedicalTeam})
    end
  END
```

B.3 Kontext healthcare_highdata

```
MACHINE healthcare_highdata
REFINES healthcare_userset
SEES healthcare_context, rbac_userset
VARIABLES
  U
INVARIANTS
  inv1: U ∈ USER_SET
EVENTS
INITIALISATION ≙
  begin
```

```
      act2: U := {u1}
   end
createUser ≙
   refines
      createUser
   any
      s     > Session
      u     > User
   where
      grd1: u ∈ USER
      grd2: s ∈ SESSION
      grd5: u ∉ U
   then
      act2: U := insert_U(U ↦ u)
   end
destroyUser ≙
   refines
      destroyUser
   any
      s     > Session
      u     > User
   where
      grd1: s ∈ SESSION
      grd2: u ∈ USER
      grd5: u ∈ U
   then
      act2: U := delete_U(U ↦ u)
   end
END
```

C Der interaktive Theorembeweiser

Dieser Anhang gibt einen knappen Überblick über Beweise mit dem interaktiven Theorembeweiser der *Rodin*-Entwicklungsumgebung unter Einsatz der *AtelierB*-Beweiser, welche als Plugins erhältlich sind. Er dokumentiert zudem den Beweis des Theorems `rev2` aus `rbac_rolehandling` (vgl. Unterabschnitt 4.1.5).

C.1 Überblick über den interaktiven Theorembeweiser

Jeder Beweis hat ein *Ziel*: dies ist eine Aussage in der mathematischen Notation von Event-B, deren Gültigkeit auf Grundlage einer Menge an als gültig angenommenen Aussagen, sogenannten *Hypothesen*, zu zeigen ist. Dabei ist eine Reihe von Variablen

mit Namen und Typ bekannt. Zu zeigen ist also für Variablen $x_1, x_2, \ldots, x_m$ mit Hypothesen $H_1, H_2, \ldots, H_n$ das Ziel G:

$$\forall\, x_1, x_2, \ldots, x_m \cdot H_1, H_2, \ldots, H_n \Rightarrow G$$

Häufig ergeben sich im Laufe eines Beweises Teilziele, aus deren Gültigkeit dann die Gültigkeit des (Haupt-)ziels folgt. Zu den zur Verfügung stehenden Werkzeugen gehören:

- **An- und Abwählen von Hypothesen (`sl/ds`).** Hypothesen lassen sich auswählen. Von den meisten automatischen Werkzeugen werden stets nur ausgewählte Hypothesen betrachtet, was bei korrekter Auswahl die Automatismen teils erheblich beschleunigen kann.

- **Umformung des Ziels oder der Hypothesen.** Auf Nutzeranfrage können Ziel oder Hyopthesen gemäß bestimmter Regeln umgeformt werden. Lautet das Ziel etwa $A \Rightarrow B$, so kann a in die Hypothesen aufgenommen und das Ziel zu B umgeändert werden (*Modus Ponens*). Lautet das Ziel $\forall\, x \cdot A$, so kann x (möglicherweise unter anderem Namen) als Variable aufgenommen werden, das neue Ziel ist dann A. Lautet eine Hypothese $\exists\, x \cdot A$, so kann x ebenso als Variable aufgenommen und A als Hypothese eingeführt werden. Bei einer Hypothese $\forall\, x \cdot A$ hingegen kann in A eine Variable y für x eingesetzt und dieses umgeformte A dann als Hypothese hinzugefügt werden. Ebenso lassen sich Ersetzungen aufgrund von Hypothesen der Form $A = B$ durchführen und dergleichen mehr.

- **Einführen neuer Hypothesen (`ah`).** Diese werden zuerst bewiesen und werden anschließend in die Menge der Hypothesen aufgenommen.

- **Abstraktion (`ae`).** Ein Ausdruck kann durch eine Variable ersetzt werden; deren Gleichheit wird dann als Hypothese eingeführt. Dies hilft Menschen wie auch automatischen Beweiswerkzeugen oftmals, elementare Zusammenhänge anstelle von Details zu erkennen.

- **Beweis durch Fallunterscheidung (`dc`).** Um ein Ziel zu zeigen, wird anhand einer Aussage eine Fallunterscheidung vorgenommen. Das Ziel wird zuerst mit der Aussage als Hypothese, anschließend mit der negierten Aussage als Hypothese gezeigt.

- **Automatische Beweistechniken.** Die Entwicklungsumgebung kann bestimmte Beweisschritte vollautomatisiert durchführen. Die Beweiswerkzeuge `newPP`, `PP`, `P0`, `P1` und `ML` versuchen hierzu, das Ziel anhand der Hypothesen selbstständig zu zeigen (siehe [Eve12]). Nebst Lösungsalgorithmen enthalten diese Werkzeuge hierzu vorallem Umformungsregeln und als wahr bekannte Aussagen. Der Einsatz eines dieser Werkzeuge schließt fast immer den Beweis eines (Teil- oder Haupt-)ziels ab.

- **Automatische Umformungen.** Bestimmte, größtenteils regelbasierte Umformungen können automatisiert vorgenommen werden. Eine konfigurierbare

Menge an Umformungstechniken wendet Rodin automatisch nach jeder Nutzerinteraktion an (die sog. *Post-Tactics*), eine andere Menge an Umformungs- und Beweistechniken können auf Nutzeranfrage angewandt werden (die sog. *Auto-Tactics*).

C.2 Beweis zu `rbac_rolehandling/rev2/THM`

Zu zeigen ist also das Theorem `rev2` aus `rbac_rolehandling` (vgl. Unterabschnitt 4.1.5) Dabei wird hier nicht jeder Bedienschritt nachvollzogen – vielmehr ist es Ziel dieses Anhangs, einen Überblick über die Beweisführung zu geben und eine Person, welche den Theorembeweiser bedienen kann, in die Lage zu versetzen, ihn selbst durchzuführen.

Das Theorem sagt aus, dass die Anwendung von `activateRoles` durch die Anwendung von `deactivateRoles` innerhalb des Zustandsraumes aufgehoben wird. Zu zeigen ist also

1. Ziel:

$$\begin{aligned}&\forall \mathtt{q}, \mathtt{sr} \cdot \\ &\quad \mathtt{q} \in \mathtt{STATE} \land \\ &\quad \mathtt{sr} \subseteq \mathtt{user\_q(q)}; \mathtt{UA\_q(q)} \land \mathtt{finite(sr)} \land \\ &\quad (\forall \mathtt{s} \cdot \mathtt{s} \in \mathtt{dom(user\_q(q))} \Rightarrow \mathtt{roles\_q(q)(s)} \cap \mathtt{sr}[\{\mathtt{s}\}] = \varnothing) \\ &\Rightarrow \mathtt{deactivateRoles(activateRoles(q} \mapsto \mathtt{sr)} \mapsto \mathtt{sr)} = \mathtt{q}\end{aligned}$$

unter Annahme aller vorherigen Axiome und Theoreme, einschließlich jener aus via Erweiterung eingebundenen Kontexten. Erhalten durch Auflösung des Allquantors und der Implikation, zusammen mit den Variablen `q` und `sr`:

2. Ziel: $\mathtt{deactivateRoles(activateRoles(q} \mapsto \mathtt{sr)} \mapsto \mathtt{sr)} = \mathtt{q}$

3. Hyp: $\mathtt{q} \in \mathtt{STATE}$

4. Hyp: $\mathtt{sr} \subseteq \mathtt{user\_q(q)}; \mathtt{UA\_q(q)} \land \mathtt{finite(sr)}$

5. Hyp: $\forall \mathtt{s} \cdot \mathtt{s} \in \mathtt{dom(user\_q(q))} \Rightarrow \mathtt{roles\_q(q)(s)} \cap \mathtt{sr}[\{\mathtt{s}\}] = \varnothing$

Nun müssen zum besseren Handling die Projektionsfunktionen durch Variablen ersetzt und die Konsistenzbedingungen aus dem Zustandsraum importiert werden. Hierzu wird zunächst Axiom `rbac_state/st3` eingesetzt, woraus folgt, dass:

6. Hyp: $\mathtt{U} \mapsto \mathtt{S} \mapsto \mathtt{UA} \mapsto \mathtt{user} \mapsto \mathtt{roles} = \mathtt{q}$

7. Hyp: $\mathtt{U\_q(q)} = \mathtt{U}$

8. Hyp: $\mathtt{S\_q(q)} = \mathtt{S}$

9. Hyp: $\mathtt{UA\_q(q)} = \mathtt{UA}$

10. Hyp: $\mathtt{user\_q(q)} = \mathtt{user}$

11. Hyp: $\mathtt{roles\_q(q)} = \mathtt{roles}$

Nutzen nun Axiom `rbac_state/st-i` mit Variablen $\mathtt{U}, \mathtt{S}, \mathtt{UA}, \mathtt{user}$ und `roles` und erhalten:

12. Hyp: $\mathtt{dom}(\mathtt{UA}) \subseteq \mathtt{U}$

13. Hyp: $\mathtt{dom}(\mathtt{user}) \subseteq \mathtt{S} \land \mathtt{ran}(\mathtt{user}) \subseteq \mathtt{U}$

14. Hyp: $\mathtt{dom}(\mathtt{roles}) = \mathtt{S} \land (\forall\, \mathtt{s} \cdot \mathtt{s} \in \mathtt{S} \Rightarrow \mathtt{finite}(\mathtt{roles}(\mathtt{s})))$

15. Hyp: $(\forall\, \mathtt{s} \cdot \mathtt{s} \in \mathtt{dom}(\mathtt{user}) \Rightarrow \mathtt{roles}(\mathtt{s}) \subseteq \mathtt{UA}[\,\{\mathtt{user}(\mathtt{s})\}\,])$

Nun sind alle Voraussetzungen geschaffen, um das Ziel aufzulösen. Durch schrittweises Anwenden aller Definitionen lässt sich die Äquivalenz zu `q` beweisen. Das Vorgehen erfolgt dabei von innen nach außen, sodass zuerst die Definition von `activateRoles` angewendet wird. Grundlage hierzu sind `rbac_rolehandling/acr_d` und `rbac_rolehandling/acr_v`. Die entsprechenden Vorbedingungen sind trivial zu zeigen, und es folgt:

16. Hyp:
$$\begin{aligned}&\mathtt{activateRoles}(\mathtt{q} \mapsto \mathtt{sr}) = \mathtt{U\_q}(\mathtt{q}) \mapsto \mathtt{S\_q}(\mathtt{q}) \mapsto \mathtt{UA\_q}(\mathtt{q}) \mapsto \\ &\quad \mathtt{user\_q}(\mathtt{q}) \mapsto \\ &\quad \{\mathtt{s}, \mathtt{r} \cdot \\ &\qquad \mathtt{s} \in \mathtt{dom}(\mathtt{roles\_q}(\mathtt{q})) \land \\ &\qquad \mathtt{r} = (\mathtt{roles\_q}(\mathtt{q})(\mathtt{s}) \cup \mathtt{sr}[\,\{\mathtt{s}\} \cap \mathtt{dom}(\mathtt{user\_q}(\mathtt{q}))\,]) \\ &\qquad \mid \mathtt{s} \mapsto \mathtt{r} \\ &\quad \}\end{aligned}$$

17. Hyp:
$$\begin{aligned}&\mathtt{U\_q}(\mathtt{q}) \mapsto \mathtt{S\_q}(\mathtt{q}) \mapsto \mathtt{UA\_q}(\mathtt{q}) \mapsto \mathtt{user\_q}(\mathtt{q}) \mapsto \\ &\quad \{\mathtt{s}, \mathtt{r} \cdot \\ &\qquad \mathtt{s} \in \mathtt{dom}(\mathtt{roles\_q}(\mathtt{q})) \land \\ &\qquad \mathtt{r} = (\mathtt{roles\_q}(\mathtt{q})(\mathtt{s}) \cup \mathtt{sr}[\,\{\mathtt{s}\} \cap \mathtt{dom}(\mathtt{user\_q}(\mathtt{q}))\,]) \\ &\qquad \mid \mathtt{s} \mapsto \mathtt{r} \\ &\quad \} \in \mathtt{STATE}\end{aligned}$$

Die Projektionsfunktionen können durch die oben eingeführten Äquivalenzen durch Variablen ersetzt werden, anschließend wird die Definition von `activateRoles` ins Ziel eingesetzt. Erhalten:

18. Ziel:
$$\begin{aligned}&\mathtt{deactivateRoles}(\mathtt{U} \mapsto \mathtt{S} \mapsto \mathtt{UA} \mapsto \mathtt{user} \mapsto \\ &\quad \{\mathtt{s}, \mathtt{r} \cdot \\ &\qquad \mathtt{s} \in \mathtt{dom}(\mathtt{roles}) \land \\ &\qquad \mathtt{r} = (\mathtt{roles}(\mathtt{s}) \cup \mathtt{sr}[\,\{\mathtt{s}\} \cap \mathtt{dom}(\mathtt{user})\,]) \\ &\qquad \mid \mathtt{s} \mapsto \mathtt{r} \\ &\quad \} \mapsto \mathtt{sr}) = \mathtt{q}\end{aligned}$$

Dabei ist das Ergebnis von `activateRoles` lediglich ein Zwischenzustand, welcher der Anwendung von `deactivateRoles` als Ausgangszustand dient. Da `deactivateRoles` wiederum über Projektionsfunktionen definiert ist, müssen diese nun für den Zwischenzustand eingeführt werden. Über die Definition der Projektionsfunktionen in `rbac_state/st-d` und die Abstraktion des Zwischenzustands zur

Variable ae folgt:

19. Ziel:
$\texttt{deactivateRoles}(\texttt{ae} \mapsto \texttt{sr}) = \texttt{q}$

20. Hyp: $\texttt{U\_q}(\texttt{ae}) = \texttt{U}$

21. Hyp: $\texttt{S\_q}(\texttt{ae}) = \texttt{S}$

22. Hyp: $\texttt{UA\_q}(\texttt{ae}) = \texttt{UA}$

23. Hyp: $\texttt{roles\_q}(\texttt{ae}) = \{\texttt{s}, \texttt{r} \cdot$
$\texttt{s} \in \texttt{dom}(\texttt{roles}) \wedge$
$\texttt{r} = \texttt{roles}(\texttt{s}) \cup \texttt{sr}[\,\{\texttt{s}\} \cap \texttt{dom}(\texttt{user})\,])$
$\mid \texttt{s} \mapsto \texttt{r}$
$\}$

24. Hyp: $\texttt{user\_q}(\texttt{ae}) = \texttt{user}$

Nun muss deactivateRoles anhand seiner Definition aufgelöst werden, diese findet sich in rbac_rolehandling/dar_d und muss, gemeinsam mit rbac_rolehandling/dar_v, mit den Variablen ae, sr instantiiert werden. Es ergeben sich zunächst:

25. Hyp:
$\texttt{ae} \in \texttt{STATE} \wedge$
$\texttt{sr} \subseteq \{\texttt{s}, \texttt{r} \cdot \texttt{s} \in \texttt{dom}(\texttt{roles\_q}(\texttt{ae})) \wedge \texttt{r} \in \texttt{roles\_q}(\texttt{ae})(\texttt{s}) \mid \texttt{s} \mapsto \texttt{r}\}$
$\Rightarrow \texttt{deactivateRoles}(\texttt{ae} \mapsto \texttt{sr}) = \texttt{U\_q}(\texttt{ae}) \mapsto \texttt{S\_q}(\texttt{ae}) \mapsto \texttt{UA\_q}(\texttt{ae}) \mapsto$
$\texttt{user\_q}(\texttt{ae}) \mapsto$
$\{\texttt{s}, \texttt{r} \cdot \texttt{s} \in \texttt{dom}(\texttt{roles\_q}(\texttt{ae})) \wedge$
$\texttt{r} = (\texttt{roles\_q}(\texttt{ae})(\texttt{s}) \setminus \texttt{sr}[\,\{\texttt{s}\}\,]) \mid \texttt{s} \mapsto \texttt{r}$
$\}$

26. Hyp:
$\texttt{ae} \in \texttt{STATE} \wedge$
$\texttt{sr} \subseteq \{\texttt{s}, \texttt{r} \cdot \texttt{s} \in \texttt{dom}(\texttt{roles\_q}(\texttt{ae})) \wedge \texttt{r} \in \texttt{roles\_q}(\texttt{ae})(\texttt{s}) \mid \texttt{s} \mapsto \texttt{r}\}$
$\Rightarrow \texttt{U\_q}(\texttt{ae}) \mapsto \texttt{S\_q}(\texttt{ae}) \mapsto \texttt{UA\_q}(\texttt{ae}) \mapsto \texttt{user\_q}(\texttt{ae}) \mapsto$
$\{\texttt{s}, \texttt{r} \cdot \texttt{s} \in \texttt{dom}(\texttt{roles\_q}(\texttt{ae})) \wedge$
$\texttt{r} = (\texttt{roles\_q}(\texttt{ae})(\texttt{s}) \setminus \texttt{sr}[\,\{\texttt{s}\}\,]) \mid \texttt{s} \mapsto \texttt{r}$
$\} \in \texttt{STATE}$

Nachdem die Projektionsfunktionen wieder aufgelöst wurden, wird der Ausdruck komplizierter, da roles_q(ae) selbst komplex ist. Nun wird an mehreren Stellen der Wert von roles_q(ae) an einer bestimmten Stelle abgefragt, also für ein $\texttt{s} \in \texttt{S}$ der Wert von roles_q(ae)(s). Dies jedoch geht genau aus der Definition von roles_q(ae) hervor und kann somit verkürzend umgeschrieben werden. Die automatischen Beweiswerkzeuge scheinen hierzu nicht in der Lage zu sein. Behaupten daher:

27. Hyp:
$\forall\, \texttt{s} \cdot \texttt{s} \in \texttt{S} \Rightarrow$

$$\{\mathtt{s}, \mathtt{r} \cdot \mathtt{s} \in \mathtt{dom}(\mathtt{roles}) \land \\ \mathtt{r} = \mathtt{roles}(\mathtt{s}) \cup \mathtt{sr}[\{\mathtt{s}\} \cap \mathtt{dom}(\mathtt{user})] \mid \mathtt{s} \mapsto \mathtt{r} \\ \}(\mathtt{s}) = \mathtt{roles}(\mathtt{s}) \cup \mathtt{sr}[\{\mathtt{s}\} \cap \mathtt{dom}(\mathtt{user})]$$

Zunächst muss hierzu die Wohldefiniertheit gezeigt werden, dies ist einfach und erfolgt größtenteils mit den automatischen Beweisern. Anschließend ist das Theorem selbst zu beweisen. Stellen die Variable `s` frei und erhalten:

28. Ziel:
$$\{\mathtt{s}, \mathtt{r} \cdot \mathtt{s} \in \mathtt{dom}(\mathtt{roles}) \land \\ \mathtt{r} = \mathtt{roles}(\mathtt{s}) \cup \mathtt{sr}[\{\mathtt{s}\} \cap \mathtt{dom}(\mathtt{user})] \mid \mathtt{s} \mapsto \mathtt{r} \\ \}(\mathtt{s}) = \mathtt{roles}(\mathtt{s}) \cup \mathtt{sr}[\{\mathtt{s}\} \cap \mathtt{dom}(\mathtt{user})]$$

Stellen hierzu erstens fest, dass die Zuordnung $\mathtt{s} \mapsto \mathtt{roles}(\mathtt{s}) \cup \mathtt{sr}[\{\mathtt{s}\} \cap \mathtt{dom}(\mathtt{user})]$ tatsächlich Element jener Menge ist, und zweitens, dass die Menge eine Abbildung von `SESSION` auf $\mathbb{P}(\mathtt{ROLE})$ ist. Beides ist mittels automatischer Werkzeuge einfach zu zeigen. Abstrahieren schließlich von konkreten Werten durch Einführen der Variablen `ae0`, `ae1` mit:

29. Hyp: $\mathtt{ae0} = \{\mathtt{s}, \mathtt{r} \cdot \mathtt{s} \in \mathtt{dom}(\mathtt{roles}) \land \\ \mathtt{r} = \mathtt{roles}(\mathtt{s}) \cup \mathtt{sr}[\{\mathtt{s}\} \cap \mathtt{dom}(\mathtt{user})] \mid \mathtt{s} \mapsto \mathtt{r}\}$

30. Hyp: $\mathtt{ae1} = \mathtt{roles}(\mathtt{s}) \cup \mathtt{sr}[\{\mathtt{s}\} \cap \mathtt{dom}(\mathtt{user})]$

Erhalten dann insgesamt:

31. Ziel:
$\mathtt{ae0}(\mathtt{s}) = \mathtt{ae1}$

32. Hyp: $\mathtt{s} \mapsto \mathtt{ae1} \in \mathtt{ae0}$

33. Hyp: $\mathtt{ae0} \in \mathtt{SESSION} \nrightarrow \mathbb{P}(\mathtt{ROLE})$

Diese abstrakte Version der Behauptungen kann problemlos automatisch bewiesen werden, wodurch Hypothese 27 bewiesen ist. Hiermit ist es nun möglich, die Vorbedingungen für `deactivateRoles` zu zeigen. Während $\mathtt{ae} \in \mathtt{STATE}$ bereits aus `rbac_rolehandling/acr_v` hervorgeht, muss die zweite Vorbedingung explizit gezeigt werden. Da dies für `dar_d` und für `dar_v` nötigt ist, zeigen wir sie als Hypothese und erhalten das Teilziel:

34. Ziel:
$$\mathtt{sr} \subseteq \{\mathtt{s}, \mathtt{r} \cdot \\ (\exists \mathtt{r} \cdot \\ \mathtt{s} \in \mathtt{dom}(\mathtt{roles}) \land \\ \mathtt{r} = \mathtt{roles}(\mathtt{s}) \cup \mathtt{sr}[\{\mathtt{s}\} \cap \mathtt{dom}(\mathtt{user})]) \land \\ \mathtt{r} \in \{\mathtt{s}, \mathtt{r} \cdot \\ \mathtt{s} \in \mathtt{dom}(\mathtt{roles}) \land \\ \mathtt{r} = \mathtt{roles}(\mathtt{s}) \cup \mathtt{sr}[\{\mathtt{s}\} \cap \mathtt{dom}(\mathtt{user})] \\ \mid \mathtt{s} \mapsto \mathtt{r}\}(\mathtt{s})$$

$$\mid \mathtt{s} \mapsto \mathtt{r}$$
$$\}$$

Zuerst wird das Ziel umgeformt, indem $\subseteq$ durch die Definition aufgelöst wird:

35. Ziel:

$$\forall \mathtt{x}, \mathtt{x0} \cdot$$
$$\mathtt{x} \mapsto \mathtt{x0} \in \mathtt{sr}$$
$$\Rightarrow$$
$$(\exists \mathtt{r} \cdot$$
$$\mathtt{x} \in \mathtt{dom}(\mathtt{roles}) \wedge$$
$$\mathtt{r} = \mathtt{roles}(\mathtt{x}) \cup \mathtt{sr}[\{\mathtt{x}\} \cap \mathtt{dom}(\mathtt{user})]) \wedge$$
$$\mathtt{x0} \in \{\mathtt{s}, \mathtt{r} \cdot \mathtt{s} \in \mathtt{dom}(\mathtt{roles}) \wedge$$
$$\mathtt{r} = \mathtt{roles}(\mathtt{s}) \cup \mathtt{sr}[\{\mathtt{s}\} \cap \mathtt{dom}(\mathtt{user})] \mid \mathtt{s} \mapsto \mathtt{r}\}(\mathtt{x})$$

Nach weiterer Umformung des Ziels ergeben sich zwei Teilziele, nämlich erstens folgendes leicht zu zeigende:

36. Ziel:

$$\exists \mathtt{r} \cdot$$
$$\mathtt{x} \in \mathtt{dom}(\mathtt{roles}) \wedge$$
$$\mathtt{r} = \mathtt{roles}(\mathtt{x}) \cup \mathtt{sr}[\{\mathtt{x}\} \cap \mathtt{dom}(\mathtt{user})]$$

Sowie zweitens:

37. Ziel:

$$\mathtt{x0} \in \{\mathtt{s}, \mathtt{r} \cdot \mathtt{s} \in \mathtt{dom}(\mathtt{roles}) \wedge$$
$$\mathtt{r} = \mathtt{roles}(\mathtt{s}) \cup \mathtt{sr}[\{\mathtt{s}\} \cap \mathtt{dom}(\mathtt{user})] \mid \mathtt{s} \mapsto \mathtt{r}\}(\mathtt{x})$$

Hier kann nun die oben gezeigte Hypothese 27 angewandt werden, wobei für `s` unser `x` eingesetzt wird.

38. Ziel:

$$\mathtt{x0} \in \mathtt{roles}(\mathtt{x}) \cup \mathtt{sr}[\{\mathtt{x}\} \cap \mathtt{dom}(\mathtt{user})]$$

Da $\mathtt{x} \mapsto \mathtt{x0} \in \mathtt{sr}$ und somit $\mathtt{x} \in \mathtt{dom}(\mathtt{sr})$ gilt, sowie $\mathtt{sr} \subseteq \mathtt{user} \fatsemi \mathtt{UA}$, folgt $\mathtt{x} \in \mathtt{dom}(\mathtt{user})$, und damit

39. Ziel:

$$\mathtt{x0} \in \mathtt{roles}(\mathtt{x}) \cup \mathtt{sr}[\{\mathtt{x}\}]$$

Dies ist leicht zu zeigen. Damit ist Hypothese 34 und somit die Vorbedingung für `deactivateRoles` bewiesen und die Definition kann angewandt werden. Mit der Definition von `q` und der Vereinfachung aus Hypothese 27 erhalten wir:

40. Ziel:

$$\mathtt{U} \mapsto \mathtt{S} \mapsto \mathtt{UA} \mapsto \mathtt{user} \mapsto$$

$$
\begin{array}{l}
\quad \{\mathtt{s}, \mathtt{r} \cdot \\
\qquad (\exists\, \mathtt{r} \cdot \\
\qquad\quad \mathtt{s} \in \mathtt{dom}(\mathtt{roles}) \wedge \\
\qquad\quad \mathtt{r} = \mathtt{roles}(\mathtt{s}) \cup \mathtt{sr}[\{\mathtt{s}\} \cap \mathtt{dom}(\mathtt{user})]) \wedge \\
\qquad \mathtt{r} = \mathtt{roles}(\mathtt{s}) \cup \mathtt{sr}[\{\mathtt{s}\} \cap \mathtt{dom}(\mathtt{user})] \setminus \mathtt{sr}[\{\mathtt{s}\}] \\
\qquad \mid \mathtt{s} \mapsto \mathtt{r} \\
\quad \} \\
= \mathtt{U} \mapsto \mathtt{S} \mapsto \mathtt{UA} \mapsto \mathtt{user} \mapsto \mathtt{roles}
\end{array}
$$

und folglich:

41. Ziel:

$$
\begin{array}{l}
\{\mathtt{s}, \mathtt{r} \cdot \\
\qquad (\exists\, \mathtt{r} \cdot \\
\qquad\quad \mathtt{s} \in \mathtt{dom}(\mathtt{roles}) \wedge \\
\qquad\quad \mathtt{r} = \mathtt{roles}(\mathtt{s}) \cup \mathtt{sr}[\{\mathtt{s}\} \cap \mathtt{dom}(\mathtt{user})]) \wedge \\
\qquad \mathtt{r} = \mathtt{roles}(\mathtt{s}) \cup \mathtt{sr}[\{\mathtt{s}\} \cap \mathtt{dom}(\mathtt{user})] \setminus \mathtt{sr}[\{\mathtt{s}\}] \\
\qquad \mid \mathtt{s} \mapsto \mathtt{r} \\
\} = \mathtt{roles}
\end{array}
$$

Die Anwendung von `deactivateRoles` hebt also die Anwendung von `activateRoles` unter den gegebenen Bedingungen genau dann auf, wenn die Umformungen an `roles` wieder äquivalent zu `roles` sind. Um dies nachzuweisen, wenden wir auf das Ziel $A = B \Leftrightarrow A \subseteq B \wedge B \subseteq A$ an und erhalten zwei Teilziele, die gemeinsam hinreichend zum Beweis des gesamten Theorems sind.

Erstes Teilziel („⊆“). Zu zeigen ist folgendes.

42. Ziel:

$$
\begin{array}{l}
\{\mathtt{s}, \mathtt{r} \cdot \\
\qquad (\exists\, \mathtt{r} \cdot \\
\qquad\quad \mathtt{s} \in \mathtt{dom}(\mathtt{roles}) \wedge \\
\qquad\quad \mathtt{r} = \mathtt{roles}(\mathtt{s}) \cup \mathtt{sr}[\{\mathtt{s}\} \cap \mathtt{dom}(\mathtt{user})]) \wedge \\
\qquad \mathtt{r} = \mathtt{roles}(\mathtt{s}) \cup \mathtt{sr}[\{\mathtt{s}\} \cap \mathtt{dom}(\mathtt{user})] \setminus \mathtt{sr}[\{\mathtt{s}\}] \\
\qquad \mid \mathtt{s} \mapsto \mathtt{r} \\
\} \subseteq \mathtt{roles}
\end{array}
$$

Dies lässt sich umschreiben zu:

43. Ziel:

$$
\begin{array}{l}
\forall\, \mathtt{s}, \mathtt{r} \cdot \\
\qquad (\exists\, \mathtt{r} \cdot \\
\qquad\quad \mathtt{s} \in \mathtt{dom}(\mathtt{roles}) \wedge \\
\qquad\quad \mathtt{r} = \mathtt{roles}(\mathtt{s}) \cup \mathtt{sr}[\{\mathtt{s}\} \cap \mathtt{dom}(\mathtt{user})]) \wedge \\
\qquad \mathtt{r} = \mathtt{roles}(\mathtt{s}) \cup \mathtt{sr}[\{\mathtt{s}\} \cap \mathtt{dom}(\mathtt{user})] \setminus \mathtt{sr}[\{\mathtt{s}\}] \\
\Rightarrow \\
\qquad \mathtt{s} \mapsto \mathtt{r} \in \mathtt{roles}
\end{array}
$$

Durch Auflösen des Allquantors und der Implikation erhalten wir:

44. Ziel:
$\mathtt{s} \mapsto \mathtt{r} \in \mathtt{roles}$

45. Hyp: $\exists\, \mathtt{r} \cdot \mathtt{s} \in \mathtt{dom}(\mathtt{roles}) \land \mathtt{r} = \mathtt{roles}(\mathtt{s}) \cup \mathtt{sr}[\{\mathtt{s}\} \cap \mathtt{dom}(\mathtt{user})]$

46. Hyp: $\mathtt{r} = \mathtt{roles}(\mathtt{s}) \cup \mathtt{sr}[\{\mathtt{s}\} \cap \mathtt{dom}(\mathtt{user})] \setminus \mathtt{sr}[\{\mathtt{s}\}]$

Setzen Hypothese 46 ins Ziel ein:

47. Ziel:
$\mathtt{s} \mapsto \mathtt{roles}(\mathtt{s}) \cup \mathtt{sr}[\{\mathtt{s}\} \cap \mathtt{dom}(\mathtt{user})] \setminus \mathtt{sr}[\{\mathtt{s}\}] \in \mathtt{roles}$

Betrachten wir diesen Ausdruck näher. $\mathtt{roles}(\mathtt{s})$ sind die aktivierten Rollen im Ausgangszustand; die Vereinigung mit $\mathtt{sr}[\{\mathtt{s}\} \cap \mathtt{dom}(\mathtt{user})]$ erfolgte durch `activateRoles`, und das Entfernen von $\mathtt{sr}[\{\mathtt{s}\}]$ durch `deactivateRoles`. Mit $\mathtt{dom}(\mathtt{sr}) \subseteq \mathtt{dom}(\mathtt{user})$ ist $\mathtt{sr}[\{\mathtt{s}\} \cap \mathtt{dom}(\mathtt{user})] = \mathtt{sr}[\{\mathtt{s}\}]$ und es ergibt sich:

48. Ziel:
$\mathtt{s} \mapsto (\mathtt{roles}(\mathtt{s}) \cup \mathtt{sr}[\{\mathtt{s}\}]) \setminus \mathtt{sr}[\{\mathtt{s}\}] \in \mathtt{roles}$

Um das Ziel zeigen zu können, müssen $\mathtt{sr}[\{\mathtt{s}\}]$ und $\mathtt{roles}(\mathtt{s})$ disjunkt sein. Zeige dies per Fallunterscheidung. Sei $\mathtt{s} \in \mathtt{dom}(\mathtt{user})$. Dann folgt mit Hypothese 5 genau dies und wir erhalten das Ziel. Sei hingegen $\mathtt{s} \notin \mathtt{dom}(\mathtt{user})$. Dann ist $\mathtt{sr}[\{\mathtt{s}\}] = \varnothing$ und es folgt das Ziel.

Zweites Teilziel („⊇“). Es ist zu zeigen:

49. Ziel:
$$
\begin{array}{l}
\mathtt{roles} \subseteq \\
\{\mathtt{s}, \mathtt{r} \cdot \\
\quad (\exists\, \mathtt{r} \cdot \\
\qquad \mathtt{s} \in \mathtt{dom}(\mathtt{roles}) \land \\
\qquad \mathtt{r} = \mathtt{roles}(\mathtt{s}) \cup \mathtt{sr}[\{\mathtt{s}\} \cap \mathtt{dom}(\mathtt{user})]) \land \\
\quad \mathtt{r} = \mathtt{roles}(\mathtt{s}) \cup \mathtt{sr}[\{\mathtt{s}\} \cap \mathtt{dom}(\mathtt{user})] \setminus \mathtt{sr}[\{\mathtt{s}\}] \\
\quad \mid \mathtt{s} \mapsto \mathtt{r} \\
\}
\end{array}
$$

Entfernen $\subseteq$ durch Umschreibung aus dem Ziel und erhalten:

50. Ziel:
$$
\begin{array}{l}
\forall\, \mathtt{x}, \mathtt{x0} \cdot \\
\quad \mathtt{x} \mapsto \mathtt{x0} \in \mathtt{roles} \\
\Rightarrow \\
\quad \mathtt{x} \mapsto \mathtt{x0} \in \{\mathtt{s}, \mathtt{r} \cdot \\
\qquad (\exists\, \mathtt{r} \cdot \\
\qquad\quad \mathtt{x} \in \mathtt{dom}(\mathtt{roles}) \land
\end{array}
$$

$$
\begin{array}{l}
\quad\quad\quad\quad \mathtt{r} = \mathtt{roles}(\mathtt{x}) \cup \mathtt{sr}[\{\mathtt{x}\} \cap \mathtt{dom}(\mathtt{user})]) \wedge \\
\quad\quad\quad\quad \mathtt{x0} = \mathtt{roles}(\mathtt{x}) \cup \mathtt{sr}[\{\mathtt{x}\} \cap \mathtt{dom}(\mathtt{user})] \setminus \mathtt{sr}[\{\mathtt{x}\}] \\
\quad\quad\quad\quad \mid \mathtt{s} \mapsto \mathtt{r} \\
\quad\quad \}
\end{array}
$$

Nun können wiederum der Allquantor sowie die Implikation aufgelöst werden, es entstehen zwei Teilziele. Das erste, $\exists\, \mathtt{r} \cdot \mathtt{s} \in \mathtt{dom}(\mathtt{roles}) \wedge \mathtt{r} = \mathtt{roles}(\mathtt{s}) \cup \mathtt{sr}[\{\mathtt{s}\} \cap \mathtt{dom}(\mathtt{user})]$, kann leicht gezeigt werden und es verbleibt:

51. Ziel:
$\mathtt{x0} = \mathtt{roles}(\mathtt{x}) \cup \mathtt{sr}[\{\mathtt{x}\} \cap \mathtt{dom}(\mathtt{user})] \setminus \mathtt{sr}[\{\mathtt{x}\}]$

Dieser Beweis erfolgt äquivalent zum Beweis von Ziel 47. Es folgt wie gewünscht das Theorem.

Literatur

[Abr89] Jean-Raymond Abrial. „A Formal Approach to Large Software Construction“. In: *Mathematics of Program Construction*. Springer-Verlag, 1989.

[Abr96] Jean-Raymond Abrial. *The B Book*. Cambridge: Cambridge University Press, 1996. 779 S.

[Abr07] Jean-Raymond Abrial. „Formal Methods: Theory Becoming Practice“. In: *Journal of Universal Computer Science* 13.5 (2007), S. 619–628.

[Abr10] Jean-Raymond Abrial. *Modelling in Event-B*. Zürich: Cambridge University Press, 2010. 612 S.

[And72] James P. Anderson. *Computer Security Technology Planning Study Vol. 1*. Bedford, Massachusetts: Air Force Electronic Systems Division, 1972.

[BR93] Juan Bicarregui und Brian Ritchie. „Invariants, Frames and Postconditions: a Comparison of the VDM and B Notations“. In: *Industrial-Strength Formal Methods. First International Symposium on Formal Methods Europe. Proceedings*. Odense, Denmark, 1993, S. 162–182.

[Boe84] Barry W. Boehm. „Verifying and validating software requirements and design specifications“. In: *IEEE Software* 1.1 (1984), S. 75–88.

[Bos95] Anthony Boswell. „Specification and Validation of a Security Policy Model“. In: *IEEE Transactions on Software Engineering* 21.2 (1995), S. 63–68.

[Bow96] Jonathan Bowen. *Formal Specification & Documentation Using Z : A Case Study Approach*. London; Boston: International Thomson Publishing, 1996. 302 S.

[BSI11] *Die Lage der IT-Sicherheit in Deutschland 2011*. Bonn: Bundesamt für Sicherheit in der Informationstechnik (BSI), 2011.

[Cle11] *Atelier B - Proof Obligations Reference Manual*. Version 3.7. Clearsy. Aix-en-Provence, France, 2011. 59 S. URL: `http://www.tools.clearsy.com/index.php5?title=Documents` (besucht am 03. 05. 2012).

[CSI10] *Computer Crime and Security Survey 2010/2011*. New York: Computer Security Institute (CSI), 2010. 44 S. URL: `http://gocsi.com/survey/` (besucht am 12. 08. 2012).

[DS98] Henning Dierks und Michael Schenke. „A Unifying Framework for Correct Program Construction“. In: *Mathematics of Program Construction. 4th International Conference. Proceedings*. Springer-Verlag, 1998, S. 122–150.

[Dij76] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[Eck04] Claudia Eckert. *IT-Sicherheit: Konzepte - Verfahren - Protokolle*. 3. Auflage. München: Oldenbourg Verlag, 2004.

[Eve12] *Rodin User's Handbook*. Version 2.4. 2012. URL: `http://handbook.event-b.org/release-2012-04-04/html/` (besucht am 04. 07. 2012).

[Evea] *Event-B – Industrial Projects*. URL: `http://wiki.event-b.org/index.php/Industrial_Projects` (besucht am 09. 08. 2012).

[Eveb] *Event-B – Rodin Plug-ins*. URL: `http://wiki.event-b.org/index.php/Rodin_Plug-ins` (besucht am 09. 08. 2012).

[EB04] Mark Evered und Serge Bögeholz. „A Case Study in Access Control Requirements for a Health Information System“. In: *Proceedings of the Second Workshop on Australasian Information Security, Data Mining and Web Intelligence, and Software Internationalisation*. Australian Computer Society, Inc., 2004, S. 53–61.

[Gas88] Morrie Gasser. *Building a Secure Computer System*. Van Nostrand Reinhold, 1988.

[HRU76] Michael A. Harrison, Walter L. Ruzzo und Jeffrey D. Ullman. „Protection in Operating Systems“. In: *Communications of the ACM* 19.8 (Aug. 1976), S. 461–471.

[Här02] Hermann Härtig. „Security Architectures Revisited“. In: *Proceedings of the 10th ACM SIGOPS European Workshop*. New York, NY, USA: ACM Press, 2002.

[HHF+05] Hermann Härtig, M. Hohmuth, N. Feske u. a. „The Nizza Secure-System Architecture“. In: *First International Conference on Collaborative Computing*. IEEE, 2005, S. 1–10.

[HJN93] I.J. Hayes, Cliff B. Jones und J.E. Nicholls. *Understanding the differences between VDM and Z*. Techn. Ber. UMCS-93-8-1. Department of Computer Science, University of Manchester, 1993. 25 S.

[ITL+10] Alexei Iliasov, Elena Troubitsyna, Linas Laibinis u. a. „Developing Mode-Rich Satellite Software by Refinement in Event-B“. In: *15th International Workshop on Formal Methods for Industrial Critical Systems* (2010). URL: `http://deploy-eprints.ecs.soton.ac.uk/240/` (besucht am 20. 08. 2012).

[IEEE1012] *Software Verification and Validation*. Standard 1012. IEEE Computer Society, 2005. 120 S.

[ISO13568] *Z formal specification notation*. Int. Standard 13568:2002. ISO/IEC, März 2002.

[ISO13817] *Vienna Development Method - Specification Language - Base Language*. Int. Standard 13817-1:1996. ISO/IEC, 1996. 399 S.

[Jon90] Cliff B. Jones. *Systematic Software Development using VDM*. 2. Aufl. Manchester: Prentice Hall, 1990. 347 S.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser u. a. „seL4: Formal Verification of an OS Kernel“. In: *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles*. 2009, S. 207–220.

[KPMG10] *KPMG-Studie: Milliardenschäden durch Computerkriminalität*. 2010. URL: `http://www.kpmg.de/Presse/21498.htm` (besucht am 11. 08. 2012).

[KP11] Winfried E. Kühnhauser und Anja Pölck. „Towards Access Control Model Engineering“. In: *Proceedings of the 7th International Conference on Information Systems Security*. 2011, S. 379–382.

[Lam00] Axel van Lamsweerde. „Formal Specification: a Roadmap“. In: *Proceedings of the conference on The Future of Software Engineering* (2000).

[Lan81] Carl E. Landwehr. „Formal Models for Computer Security“. In: *ACM Computing Surveys* 13.3 (1981), S. 247–278.

[LS09] Janusz Laski und William Stanley. *Software Verification and Analysis*. London: Springer-Verlag, 2009. 224 S.

[MPP+08] Jonathan M. McCune, Bryan Parno, Adrian Perrig u. a. „Flicker: An execution infrastructure for TCB minimization“. In: *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems*. 2008, S. 315–328.

[MM06] Christoph Meinel und Martin Mundhenk. *Mathematische Grundlagen der Informatik*. 3. Auflage. Wiesbaden: B.G. Teubner Verlag, 2006. 333 S.

[Nis99] Nimal Nissanke. *Formal Specification: Techniques and Applications*. Springer-Verlag, 1999. 295 S.

[Pöl10] Anja Pölck. „Causal Trusted Computing Bases“. In: *55. Internationales Wissenschaftliches Kolloquium*. Technische Universität Ilmenau, 2010, S. 748–753.

[ProB] *The ProB Animator and Model Checker*. URL: `http://www.stups.uni-duesseldorf.de/ProB/` (besucht am 09. 08. 2012).

[SS75] Jerome H. Saltzer und Michael D. Schroeder. „The Protection of Information in Computer Systems“. In: *Proceedings of the IEEE* 63.9 (1975), S. 1278–1308.

[San92] Ravinderpal Singh Sandhu. „The Typed Access Matrix Model“. In: *Proceedings of the 1992 IEEE Symposium on Security and Privacy*. Washington DC, USA: IEEE Computer Society, 1992.

[SCF+96] Ravinderpal Singh Sandhu, Edward J. Coyne, Hal L. Feinstein u. a. „Role-based Access Control Models“. In: *IEEE Computer* 29.2 (1996), S. 38–47.

[Sch00] Bruce Schneier. *Crypto-Gram Newsletter: Software Complexity and Security*. 2000. URL: `http://www.schneier.com/crypto-gram-0003.html` (besucht am 11. 08. 2012).

[SEK+09] Takahiro Shinagawa, Hideki Eiraku, Tanimoto Kouichi u. a. „BitVisor: A Thin Hypervisor for Enforcing I/O Device Security“. In: *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 2009, S. 121–130.

[SPH+06] L Singaravelu, C Pu, Hermann Härtig u. a. „Reducing TCB Complexity for Security-Sensitive Applications: Three Case Studies“. In: *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems*. 2006.

[Spi98] J. Michael Spivey. *The Z Notation: A Reference Manual*. 2. Aufl. Oxford: Prentice Hall International (UK) Ltd, 1998.

[Ste01] Angelika Steger. *Diskrete Strukturen. Band 1: Kombinatorik - Graphentheorie - Algebra*. 2. Auflage. Springer-Verlag, 2001. 270 S.

[SYR+07a] Scott D. Stoller, Ping Yang, C.R. Ramakrishnan u. a. „Efficient Policy Analysis for Administrative Role Based Access Control“. In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. 2007, S. 445–455.

[SYR+07b] Scott D. Stoller, Ping Yang, C.R. Ramakrishnan u. a. *RBAC and ARBAC Policies for a Small Health Care Facility*. 2007. URL: `http://www.cs.sunysb.edu/~stoller/ccs2007/healthcare.txt` (besucht am 28. 02. 2012).

[WD96] Jim Woodcock und Jim Davies. *Using Z*. Prentice Hall, 1996.

[WLB+09] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui u. a. „Formal Methods: Practice and Experience“. In: *ACM Computing Surveys* 41.4 (Okt. 2009), S. 1–36.

[Wri08] Stephen Wright. „Using EventB to Create a Virtual Machine Instruction Set Architecture“. In: *Abstract State Machines, B and Z. First International Conference Proceedings*. 2008, S. 265–279.

Selbständigkeitserklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Ilmenau, den 27. August 2012

Felix Neumann