Fachbereich 3 – Mathematik und Informatik

Validierung eines RBAC-Ecore-OCL-Modells mittels des USE-Tools

(Validation of a RBAC Ecore/OCL model using the USE tool)

Masterarbeit

Master of Science (M.Sc.) im Studiengang Informatik

Autor: **Daniel Tietjen**

E-Mail: dtietjen@informatik.uni-bremen.de
Matrikelnr.: 2385631

Erstgutachter: Dr. Karsten Sohr
Zweitgutachter: Prof. Dr. Martin Gogolla

18.09.2017

Inhaltsverzeichnis

1 Einleitung

Informations- und Kommunikationstechnologie sind die Grundlage für viele Abläufe in Unternehmen geworden, da sich dadurch unter anderem der Zugang zu Daten erleichtern lässt sowie Geschäftsprozesse beschleunigt werden können. Dies führt zu gesteigerter Produktivität und damit besserer Wettbewerbsfähigkeit [34, S. 4]. So nutzen im Jahr 2016 91% der deutschen Unternehmen Computer, 89% besaßen einen Internetzugang und 77% der Unternehmen mit 250 und mehr Beschäftigten beschäftigten eigene IT-Fachkräfte [34, S. 11 f.].

Allerdings steigen mit der Verbreitung von Informations- und Kommunikationstechnologie sowie der voranschreitenden globalen Vernetzung auch die Bedrohungen, die auf der Informations- und Kommunikationstechnologie basieren. Entsprechend muss sich auch die Informationssicherheit (engl. information security) immer weiterentwickeln, um auf neue Technologien und neue Bedrohungen vorbereitet zu sein. Somit ist auch für Unternehmen, die schützenswerte Daten besitzen, die Informationssicherheit ein kontinuierlicher Prozess. Informationssicherheit muss dabei großteils vorsorglich betrieben werden, um auf absichtliche und unabsichtliche Angriffe von innen und außen vorbereitet zu sein – erst bei der ersten Verletzung der Informationssicherheit über Informationssicherheit nachzudenken ist zu spät [20, S. V, 1][16, S. V, 1].

Eine Sicherheitsrichtlinie (engl. security policy) definiert die Sicherheitsziele und das Informationssicherheitskonzept. Dazu kann beispielsweise zählen, wie die Informationssicherheit für bestimmte Organisationseinheiten und Geschäftsprozesse eines Unternehmens umgesetzt werden soll [16, S. 119 ff.].

Ziel dieser Arbeit ist es ein bestehendes, allgemein gehaltenes Modell und eine darin eingebettete Sicherheitsrichtlinie zu validieren. Das Modell beschreibt den strukturellen Aufbau eines rollenbasierten Zugriffskontrollsystems für Geschäftsprozesse. Es geht darum, festzustellen, ob durch regulär ausgeführte Geschäftsprozessoperationen Systemzustände erreicht werden können, die die Sicherheitsrichtlinie verletzen – womit das Modell eine Sicherheitslücke enthalten würde.

Rollenbasierte Zugriffskontrolle (RBAC: Role-Based Access Control, [30]) zieht eine Abstraktionsebene in Systeme ein, in denen Benutzern bestimmte Berechtigungen zugewiesen werden müssen. Anstatt einzelne Berechtigungen direkt Benutzern zuzuordnen, werden die Berechtigungen in logische Rollen gruppiert und nur noch diese Rollen den Benutzern zugeordnet.

Das spezielle RBAC-Modell, welches Ausgangspunkt für diese Arbeit ist, wurde in der Arbeitsgruppe „Softwaretechnik“ der Universität Bremen mit dem Eclipse Modeling Framework (EMF, [35]) erstellt und liegt als Ecore-Modell vor. Somit kann das Modell als eine Repräsentationsform eines UML-Klassendiagramms [26] ange-

sehen werden, welches den strukturellen Aufbau der Klassen (Subject, Permission, Role, Process, ...) und deren Beziehungen zueinander (Role $\xrightarrow{\text{assigned Permissions [0..*]}}$ Permission, ...) beschreibt, ohne dabei konkrete Instanzen der Klassen zu beinhalten. Mit Hilfe von OCL (Object Constraint Language, [23]) werden dem Modell Constraints hinzugefügt, die mit UML allein nicht realisierbar wären und hier dazu dienen, die Modellierung der Zugriffskontrolle und Geschäftsprozessabläufe zu spezifizieren. So kann beispielsweise modelliert werden, dass eine bestimmte Aktion nur durchgeführt werden darf, wenn der Benutzer auch die Berechtigung dafür besitzt.

Das allgemeine, domänenunabhängige Modell wird in einem konkreten Anwendungsfall gezeigt, um besser nachvollziehen zu können, welches gültige Systemzustände – konkrete Instanziierung der Klassen (Objektdiagramm) – sind und welche nicht. Als Fallstudie wurden Geschäftsprozesse eines Krankenhauses gewählt.

Ein wichtiges Konzept, mit dem sich rollenbasierte Zugriffskontrollsysteme in ihrem Funktionsumfang zusätzlich erweitern lassen, ist die Delegation (Weitergabe von Berechtigungen). Auch im Krankenhaus-Anwendungsfall spielt Delegation eine wichtige Rolle, weshalb das Modell im Rahmen dieser Arbeit um ein Delegationskonzept erweitert wurde, welches ebenfalls Teil der Validierung ist.

Das Modell wird mit dem USE-Tool [13] der Arbeitsgruppe „Datenbanksysteme“ der Universität Bremen untersucht. Das USE-Tool verarbeitet als Input ein UML-Klassendiagramm und OCL-Constraints und erlaubt es Systemzustände zu erzeugen und zu überprüfen, ob die OCL-Constraints eingehalten werden. Sowohl Ecore als auch USE unterstützen jeweils eine unterschiedliche Teilmenge vom offiziellen UML-Standard und haben ihre Besonderheiten bei bestimmten Modellierungsaspekten. Im Rahmen dieser Arbeit wird eine selbstentwickelte Software präsentiert, die eine automatische Ecore-zu-USE-Modelltransformation erlaubt.

Ist das Modell transformiert, wird analysiert, ob die Prinzipien der rollenbasierten Zugriffskontrolle korrekt modelliert wurden und nicht Systemzustände erzeugt werden können, die zwar laut OCL-Constraints gültig sind, aber laut Sicherheitskonzept nicht gültig sein dürften.

Für USE gibt es den sogenannten Model Validator als Plugin [14]. Dieser erlaubt es, nach der Konfiguration der Rahmenbedingungen (unter anderem: minimale und maximale Anzahl von Instanzen einer Klasse und Links einer Assoziation), automatisch Systemzustände (Objektdiagramme) zu erzeugen, die diese Rahmenbedingungen erfüllen und gleichzeitig die OCL-Constraints nicht verletzen.

Der Model Validator bietet verschiedene Herangehensweisen zur Validierung des Modells, die unter Nutzung von Whitebox-Wissen über das konkrete Modell spezifiziert und kombiniert angewendet werden müssen. Dazu zählen unter anderem:

- Ergänzung von [negierten] OCL-Constraints ohne Modellmodifizierung
- [Schrittweise] Erstellung aller Systemzustände innerhalb von Rahmenbedingungen
- Vervollständigung manuell erzeugter Systemzustände

Dabei wird in dieser Arbeit auch untersucht, bis zu welcher Systemzustandsgröße (Rahmenbedingungen des Model Validators) sich automatische Validierungen in realistisch nutzbaren Zeiten durchführen lassen.

1.1 Kapitelübersicht

Zunächst werden in Kapitel 2 die Grundlagen, die für diese Arbeit relevant sind, kurz vorgestellt. Dazu gehören Informationssicherheit, rollenbasierte Zugriffskontrolle sowie für die Modellierung UML und OCL. Anschließend wird in Kapitel 3 das USE-Tool und das Model-Validator-Plugin eingeführt. Dann folgt in Kapitel 4 zunächst die Vorstellung des Eclipse Modeling Frameworks bevor anschließend das in diesem Format modellierte, gegebene RBAC-Modell beschrieben wird; gefolgt vom modellierten Delegationskonzept. Nachdem mit dem EMF das Quellformat, sowie mit USE das Zielformat für das Modell präsentiert wurden, folgt in Kapitel 5 die entwickelte Software zur Modelltransformation vom Quell- ins Zielformat. In Kapitel 6 wird auf Basis des RBAC-Modells eine Krankenhausfallstudie entwickelt, die den Einsatz des RBAC-Modells an konkreten Szenarios verdeutlicht. Darauf folgen in Kapitel 7 zunächst die theoretischen Grundlagen zur Modellvalidierung sowie die entwickelten Softwaretools zur weiteren Automatisierung der Modellvalidierung, bevor dann die eigentliche Modellvalidierung unter Verwendung der vorab vorgestellten Software durchgeführt und die Ergebnisse präsentiert werden. Abschließend rundet in Kapitel 8 ein Fazit und Ausblick die Arbeit ab.

In Anhang A ist das in Kapitel 4 vorgestellte Delegationskonzept komplett abgebildet. Weitere Dokumente befinden sich auf der CD in Anhang B. Dazu gehören beispielsweise das Modell im EMF- und USE-Format, sowie der Quellcode von der im Rahmen der Arbeit entwickelten Software.

2 Grundlagen

Dieses Kapitel liefert die Grundlagen für die darauffolgenden Kapitel, indem es in die Informationssicherheit im Allgemeinen und die rollenbasierte Zugriffskontrolle im Speziellen sowie mit UML und OCL in die Modellierungsaspekte einführt. Da die Themengebiete einzeln betrachtet bereits sehr umfangreich sind, ist eine vollumfängliche Darstellung jedes Themas im Rahmen dieser Arbeit allerdings nicht möglich – für tiefer gehendes Fachwissen sei an die dort genannte Literatur verwiesen.

2.1 Informationssicherheit

2.1.1 Informationsbegriff und Schutzziele

Informationen sind für viele Organisationen (und Privatpersonen) wichtige Werte, die es zu schützen gilt. Für Unternehmen beispielsweise können Informationen (wie z.B. Betriebsgeheimnisse und Kundendaten) die Grundlage für den wirtschaftlichen Erfolg sein. Zum abstrakten Begriff der „Information“ gehört auch eine konkrete Repräsentation inklusive Interpretationsvorschrift: Dies können beispielsweise Daten in Form einer Datei sein, die die Informationen im Klartext nach UTF-8-Kodierung enthält. Es kann aber auch ein aktives Objekt, wie ein informationsverarbeitender Prozess sein. Nicht digitale Repräsentationsformen, wie etwa Informationen auf Papier (Ausdrucke), werden hier nicht behandelt.

Informationssicherheit [11, S. 1 ff.][6, S. 1 ff.][16, S. 63 ff.] definiert sich über die Schutzziele, die es versucht aufrecht zu erhalten:

- **Vertraulichkeit:** Informationen können nur von autorisierten Personen oder Prozessen eingesehen werden; eine Eigenschaft, der beispielsweise bei Staatsgeheimnissen eine besondere Bedeutung zukommt.
- **Integrität:** Informationen können nicht auf unautorisierte oder unbemerkte Weise manipuliert (geändert, gelöscht) werden.
- **Verfügbarkeit:** Autorisierten Personen oder Prozessen stehen die Informationen nach einer akzeptablen Dauer zur Verfügung. Ein Verlust der Verfügbarkeit kann dabei durch Verlust der Daten bzw. Datenträger an sich geschehen oder, wenn die Daten eigentlich vorhanden sind, durch den Verlust der Zugriffsmöglichkeiten: Ein Denial-Of-Service-Angriff, um ein System zu überlasten, ist somit beispielsweise auch ein Angriff auf die Verfügbarkeit.

Der Verlust dieser Schutzziele ist dabei unterschiedlich früh/einfach festzustellen und die obige Aufzählung gibt dies in absteigender Reihenfolge wieder: Dass eine unautorisierte Person vertrauliche Information gewonnen hat, kann eventuell gar nicht festgestellt werden; wohingegen der Verlust der Verfügbarkeit beim Zugriff auf die Informationen direkt deutlich wird.

Neben den drei genannten Hauptschutzzielen können der Informationssicherheit unter anderem die weiteren folgenden Schutzziele zugeordnet werden:

- **Authentizität:** Die Eigenschaft einer Entität echt, glaubwürdig und damit das, was sie vorgibt zu sein, zu sein. Dafür muss die Entität eine eindeutige Identität (Charakteristik) aufweisen, die überprüft werden kann. Die eigentliche Überprüfung wird als Authentisierung bezeichnet. Beispielsweise könnte die Authentizität bei einer Benutzerkontoverwaltung durch die korrekte Eingabe eines zu einem eindeutigen Benutzerkonto gehörenden Passworts überprüft werden.

- **Zurechenbarkeit/Verbindlichkeit:** Aktionen können einer Entität fest zugeordnet werden und sind später nicht abstreitbar. Dies erfordert eine Art der Protokollierung der Aktionen. Alle Kommandos, die ein bestimmter Benutzer an eine Datenbank gesendet hat, könnten beispielsweise für eine Zurechenbarkeit protokolliert werden. Besondere Relevanz hat die Verbindlichkeit dort, wo sie benötigt wird, um Rechtsgültigkeit nachzuweisen, wie etwa ein beim Online-Handel abgeschlossener Kaufvertrag.

- **Verlässlichkeit:** Ein System verhält sich dauerhaft bestimmungsgemäß und liefert konsistente Ergebnisse.

Hat ein System eine Schwachstelle, die sich ausnutzen lässt, um die Schutzziele zu umgehen, spricht man von einer Verwundbarkeit.

2.1.2 Designprinzipien sicherer Systeme

Einige grundlegende Designprinzipien für die Entwicklung sicherer Systeme [28][27, S. 11-5 ff.][31][11, S. 188 ff.] haben sich über mehrere Jahrzehnte hinweg etabliert und sind dabei – trotz der rasanten Entwicklung in der IT-Branche – größtenteils unverändert geblieben: 1975 haben Saltzer und Schroeder acht Designprinzipien in einem Grundlagenpapier [28] aufgestellt und, wie es Saltzer und Kasshoek in der Veröffentlichung [27] von 2009 zeigen, wurden diese beibehalten oder nur geringfügig angepasst.
Die Relevanz solcher Designprinzipien entstammt der Tatsache, dass es extrem schwierig ist, zu beweisen, dass ein System sicher ist. Denn dafür muss bewiesen werden, dass *alle* Möglichkeiten unautorisierten Zugriff zu erlangen bzw. Tätigkeiten durchzuführen, die dem Sicherheitskonzept widersprechen, verhindert werden. Im Vergleich dazu, muss nur *eine einzige* Sicherheitslücke gefunden werden, um zu beweisen, dass ein System unsicher ist. Die Designprinzipien schaffen eine Grundlage, die möglichst bereits im Entwicklungsprozess grundlegende Sicherheitsprobleme verhindern soll.
Die acht Designprinzipien sind:

1. **Economy of mechanism:** Das Design soll einfach gehalten werden, da dies weniger Möglichkeiten für Fehler bietet und das Testen vereinfacht wird.

2. **Fail-safe defaults:** Standardmäßig sollte ein Benutzer keinen Zugriff auf ein System haben und diesen erst benutzerspezifisch erteilt bekommen, wenn er den Zugang wirklich benötigt.

3. **Complete mediation:** Die Berechtigung für ein Objekt sollte bei jedem Zugriff auf das Objekt überprüft werden und, soweit wie möglich, sollte sich nicht darauf verlassen werden, dass eine Überprüfung aus der Vergangenheit immer noch gültig ist.

4. **Open design:** Die Sicherheit sollte nicht darauf aufbauen, dass das *Design* an sich geheim bleibt (Security by obscurity).

5. **Separation of privilege:** Zur Durchführung einer kritischen Aufgabe müssen mehrere Subjekte mitwirken, sodass der komplette Tätigkeitsablauf (z.B. Abschuss einer Rakete) nicht von einer Person allein durchgeführt werden kann.

6. **Least privilige:** Jedes Subjekt sollte nur genau die Zugriffsrechte besitzen (bzw. aktiviert haben), die für die aktuelle Tätigkeit notwendig sind.

7. **Least common mechanism:** Ressourcen und Kommunikationspfade, die von mehreren oder allen Subjekten verwendet werden, sollten von der Anzahl her so gering wie möglich gehalten werden. Beispielsweise sind Angriffe über das Internet möglich, weil Angreifer und Angegriffener den gemeinsamen Kommunikationspfad Internet nutzen.

8. **Psychological acceptability / Least astonishment:** Die Benutzerschnittstelle zum System muss einfach zu bedienen sein, sodass die Sicherheitsmaßnahmen routiniert angewendet werden. Des Weiteren steht das Prinzip auch dafür, dass die Standard-Sicherheitseinstellungen einer Software so sind, dass sie genügend Sicherheit bieten, da die meisten Benutzer die Einstellungen nicht ändern werden.

2.2 Zugriffskontrollmodelle / RBAC (Role-Based Access Control)

Viele der genannten Schutzziele setzen voraus, dass die autorisierten Personen oder Prozesse bereits festgelegt wurden. Dafür muss definiert sein, welche Subjekte in welcher Art und Weise auf welche Objekte zugreifen dürfen, also welche Berechtigungen ein Subjekt hat. Um diese Berechtigungen aufzustellen, existieren verschiedene grundlegende Zugriffskontrollmodelle [11, S. 259 ff.][25, S. ES-1 ff., 2-1 ff.][16, S. 63 ff.].

Bei der benutzerbestimmbaren Zugriffskontrolle (Discretionary Access Control) legt jeweils der Eigentümer eines Objektes fest, welches Subjekt welche Zugriffsrechte erhält. Ein Beispiel für eine Realisierung dieses Modells sind Access Control Lists (ACL), wo die Berechtigungen beim Objekt gespeichert werden, wodurch leicht erkennbar ist, welche Subjekte welche Berechtigungen bezüglich eines bestimmten Objektes besitzen. Die umgekehrte Realisierung sind Capability Lists, wo dieselben

Berechtigungen nicht beim Objekt, sondern beim Subjekt vermerkt sind, mit dem Vorteil leicht zu sehen, welche Berechtigungen ein bestimmtes Subjekt insgesamt hat.

Die systembestimmte Zugriffskontrolle (Mandatory Access Control) verfolgt den Ansatz, dass Berechtigungen von einer zentralen Stelle vergeben werden, was vor allem in Organisationen relevant ist, in denen die Vertraulichkeit eine große Rolle spielt (z.B. Geheimdienst). Die Berechtigungen entstehen dabei oft durch die geordnete Klassifizierung der Objekte (z.B. offen, vertraulich, geheim, streng geheim) und einer Ermächtigung der Subjekte bis zu einer bestimmten Klassifizierungsstufe.

Das Zugriffskontrollmodell RBAC (Role-Based Access Control) [30][25, S. ES-1 ff., 2-7 ff.], bei dem Subjekten nicht direkt einzelne Berechtigungen, sondern Rollen zugewiesen werden, ist vor allem dort erfolgreich, wo ein gutes Zusammenspiel von Zugriffsmodell und Geschäftsprozessen wichtig ist. Bei der rollenbasierten Zugriffskontrolle werden Berechtigungen nicht einem Subjekt, sondern Rollen zugewiesen. Subjekten werden dann wiederum diese Rollen zugeordnet. Rollen können dabei flexibel, den Geschäftsprozessen einer Organisation angepasst, definiert werden. So können Rollen beispielsweise entsprechend der Abteilungen oder Tätigkeit erstellt werden. Über hierarchische rollenbasierte Zugriffskontrolle lassen sich Vererbungsstrukturen abbilden, sodass z.B. ein Facharzt von der Assistenzarztrolle erbt und damit alle Berechtigungen besitzt, die auch ein Assistenzarzt hat.
RBAC erlaubt es, einige der genannten Designprinzipien für sichere Systeme zu realisieren: Wenn die Rollen fein genug für die jeweiligen Tätigkeiten definiert sind, hat ein Subjekt auch nur die Berechtigungen, die benötigt werden (least privilige). Und ohne zugewiesene Rollen besitzt ein Subjekt auch keine Berechtigungen (fail-safe defaults). Ebenfalls kann man „separation of privilege“ (separation of duty) umsetzen, wenn man definiert, welche Rollen sich gegenseitig ausschließen, sodass ein Subjekt nicht beide Rollen besitzen kann. Des Weiteren sorgt die Vereinfachung der Rechtevergabe durch die Rollen dafür, dass weniger Fehler passieren im Vergleich zur Vergabe der Rechte direkt an Subjekte – dies fördert „economy of mechanism“.

2.3 Modellierung

2.3.1 UML (Unified Modeling Language)

Die Unified Modeling Language (UML) [26, S. 1 ff.][24][29, S. 20] ist eine visuelle Modellierungssprache, die vor allem im Rahmen der objektorientierten Softwareentwicklung (OOP) zum Einsatz kommt, um ein Softwaresystem, bzw. Teilaspekte davon, zu spezifizieren und zu dokumentieren. UML ist allerdings nicht darauf beschränkt, sondern ist eine plattform-, programmiersprachen-, domänen- und vorgehensmodell*un*abhängige Form, um Strukturen, Beziehungen und Verhalten zu modellieren. Somit kann sie beispielsweise auch verwendet werden, um Geschäftsprozesse abzubilden.

Die ursprüngliche Entwicklung von UML begann in den 1990er Jahren und führte 1997 zur Spezifikation von UML 1.1 durch die Object Management Group. Seitdem wurde die UML-Spezifikation stetig weiterentwickelt (2015: UML 2.5) und hat sich weltweit zu einem Standard etabliert.
Verschiedene Arten von UML-Diagrammen erlauben es, verschiedene statische und dynamische Aspekte eines Systems zu modellieren. UML definiert – einer Sprache entsprechend –, welche grafischen Elemente in den jeweiligen Diagrammen zur Verfügung stehen und wie sich diese auf wohlgeformte Weise kombinieren lassen (Syntax) sowie was bestimmte Kombinationen von syntaktischen Elementen bedeuten (Semantik). Daraus resultieren eindeutige Diagramme, sodass zwei Personen ein Diagramm – bei korrektem Verständnis von UML – nicht unterschiedlich interpretieren können. Durch die Eindeutigkeit können aus UML-Diagrammen auch direkt Quellcode-Grundgerüste erstellt werden, worauf in Kapitel 5 *Softwaregestützte Modelltransformation* näher eingegangen wird.

Im Kontext dieser Arbeit werden nur das UML-Klassendiagramm und das UML-Objektdiagramm verwendet, weswegen auch nur diese beiden Diagrammtypen nachfolgend vorgestellt werden. Das Klassendiagramm [26, S. 11 ff.][29, S. 35 ff.] zeigt den strukturellen Aufbau des Systems über Klassen und deren Beziehungen zueinander. Der UML-Klassen-Begriff ist eng verbunden mit Klassen in objektorientierten Programmiersprachen; entsprechend können UML-Klassen, neben ihrem Namen, Attribute und Methoden sowie Beziehungen (Generalisierung, Assoziation, Aggregation, Komposition) besitzen. Das Objektdiagramm ist ein eigenständiger UML-Diagrammtyp, aber trotzdem eng verbunden mit dem Klassendiagramm, da ein Objektdiagramm eine Art Momentaufnahme eines Klassendiagramms ist und konkret alle Instanzen der Klassen (Objekte) und deren Beziehungen zueinander aufzeigt: Jede einzelne Instanz einer Klasse trägt einen eigenständigen Namen, die Attribute besitzen konkrete Werte und, da die Beziehungen zwischen konkreten Objekten bestehen (Links), ist ihre Anzahl eindeutig.

2.3.2 OCL (Object Constraint Language)

Die Object Constraint Language [26, S. 192 ff.][8][23, S. 5 ff.] ist eine formale Sprache, die zum UML-Standard gehört und von der Object Management Group festgelegt wird. Sie erlaubt es, textuelle Constraints (Bedingungen, Zusicherungen) zum UML-Modell hinzuzufügen, die sich über die rein grafische Repräsentation in Diagrammen nicht spezifizieren lassen. Dazu zählen vor allem Invarianten bezüglich Klassen (Bedingungen, die für alle Instanzen einer Klasse zu jeder Zeit gültig sein müssen) sowie Vor- und Nachbedingungen von Methoden. Des Weiteren lässt sich mit OCL auch das Verhalten von Operationen (Methoden) sowie deren Übergabeparameter und Rückgabewert beschreiben.
Der Vorteil dieser formalen Sprache – gegenüber Anmerkungen im Diagramm in natürlicher Sprache – ist die Möglichkeit der automatischen Auswertung der Constraints sowie deren Eindeutigkeit, da natürliche Sprache von verschiedenen

Personen unterschiedlich gedeutet werden kann. OCL ist keine Programmiersprache im klassischen Sinne, basiert aber auf einer Syntax, die Keywords und eingebaute Methoden definiert sowie eingebettete Datentypen und Typsicherheit besitzt.
Die Ausführung von OCL-Constraints ist immer atomar und hat keine Seiteneffekte, sodass das eigentliche Modell durch OCL nicht modifiziert werden kann (deklarative Spezifikationssprache).

Ein OCL-Ausdruck ist für einen spezifizierten Datentyp (Kontext) definiert (`context <DATA TYPE>`), wobei `self` in einem OCL-Ausdruck für eine Instanz des im Kontext genannten Datentyps steht. Da beim EMF und USE der Kontext implizit dadurch bekannt ist, dass der OCL-Ausdruck einem Datentyp bereits strukturell durch den Aufbau des Modelldatenformats zugeordnet ist, entfällt die explizite Nennung des Kontextes vor jedem OCL-Ausdruck.

Die Datentypen, die unter OCL verwendet werden können [23, S. 156 ff.], sind primitive Datentypen (Boolean, Integer, Real, String), im Modell definierte Klassen und Collections, welche mehrere Elemente von primitiven oder im Modell definierten Datentypen halten können. Boolean mit den Werten `true` oder `false` ist dabei für OCL-Constraints der Datentyp der Wahl für den Rückgabewert: Liefert das OCL-Constraint `true` zurück, gilt das OCL-Constraint als erfüllt; `false` bedeutet das OCL-Constraint wurde nicht eingehalten. OCL bietet verschiedene Collection-Datentypen, die alle vom abstrakten Datentypen `Collection` abgeleitet sind und sich darin unterscheiden, ob dasselbe Objekt mehrfach in einer Collection auftreten kann oder nicht (unique), und ob die Objekte geordnet sind (ordered) oder nicht. Für die unique-Eigenschaft gilt, dass zwei Objekte, die die gleichen Attributwerte und Links besitzen, nicht als dasselbe Objekt angesehen werden und somit parallel in einer Unique-Collection enthalten sein können. Die Tabelle 1 listet alle OCL-Collection-Datentypen und ihre Eigenschaften auf.

Tabelle 1: Collection-Datentypen

Collection-Typ	**Unique**	**Ordered**
Bag	☐	☐
Set	☒	☐
Sequence/OrderedBag	☐	☒
OrderedSet	☒	☒

Neben Operationen von eigenen Datentypen, bringt OCL eine große Anzahl grundlegender Operationen mit [23], wovon eine kleine Auswahl der wichtigsten und im Rahmen dieser Arbeit am meisten verwendeten nachfolgend vorgestellt werden[1]:

- `let <VARIABLE NAME> : <VARIABLE DATA TYPE> = <VARIABLE VALUE> in` erstellt eine Variable, die nachfolgend im selben OCL-Ausdruck verwendet

[1] Viele OCL-Operationen bieten mehrere Möglichkeiten, wie sie anhand von Übergabeparametern aufgerufen werden können. Hier wird jeweils nur eine Variante vorgestellt.

werden kann und einen festgelegten Datentyp besitzt sowie einen Wert, der zur Laufzeit bestimmt werden kann.

- `forAll(<BOOLEAN EXPRESSION>)` wird auf einer Collection aufgerufen und evaluiert, ob der übergebene Ausdruck für jedes Element `true` ist. Ist dies der Fall, liefert auch `forAll` ein `true` zurück, ansonsten `false`.
- `select(<BOOLEAN EXPRESSION>)` filtert eine Teilmenge, basierend auf einem boolschen Ausdruck, aus einer Collection heraus und liefert diese Teilmenge als Rückgabewert.
- `includes(<OBJECT>)` gibt `true` zurück, wenn das übergebene Objekt in der Collection, auf der die Operation aufgerufen wurde, enthalten ist.
- `collect(<EXPRESSION>)` liefert eine Collection, basierend auf der Collection, auf der die Operation aufgerufen wurde, und dem angegebenen Ausdruck. Der Ausdruck kann beispielsweise der Name einer Assoziation sein, sodass das Resultat eine Collection ist, die alle Instanzen, auf die die Links der Elemente in der Collection, auf der die Operation aufgerufen wurde, enthält. Die Datentypen in der Quell- und Resultat-Collection können sich also unterscheiden.
- `closure(<EXPRESSION>)` kann als *rekursive* Variante von `collect` betrachtet werden und ist besonders nützlich, wenn Datentypen Assoziationen zu sich selbst besitzen, um eine Hierarchie aufzubauen. Dann können mittels `closure` alle Instanzen des sich aufgespannten Baumes (Instanzen als Knoten, Links als Kanten) mit nur einem `closure`-Aufruf ermittelt werden.
- `oclIsTypeOf(<DATA TYPE>)` überprüft, ob das Objekt, auf dem die Operation aufgerufen wurde, mit dem angegebenen Datentyp übereinstimmt.
- `oclAsType(<DATA TYPE>)` konvertiert das Objekt (oder die Collection), auf dem die Operation aufgerufen wurde, in den angegebenen Datentyp (type cast).
- `<DATA TYPE>.allInstances()` liefert eine Collection aller im Objektdiagramm befindenden Instanzen des angegebenen Datentyps.

Abhängig davon, ob die Quelle für einen OCL-Ausdruck ein einzelnes Objekt oder eine Collection ist, wird über einen anderen Operator auf die verfügbaren Attribute, Assoziationen, Methoden oder OCL-Operationen zugegriffen. Bei einem einzelnen Objekt wird der Punkt „`.`“ (Dot-Notation) und bei Collection der Pfeil „`->`“ (Arrow-Notation) verwendet. Beispiele: `objectA.name` und `collectionB->size()`. Dabei ist zu beachten, dass die Dot-Notation angewendet auf eine Collection (z.B. `collectionC.name`) für ein implizite `collect`-Operation steht.

3 USE-Tool (UML-based Specification Environment)

Die Software, die für die eigentliche Modellvalidierung verwendet wird, wird in diesem Kapitel vorgestellt: USE und das dazugehörige Plugin „Model Validator“.

3.1 Hauptprogramm

Das USE-Tool (UML-based Specification Environment) [13][9][14] kann eine Teilmenge des UML-Standards sowie den OCL-Standard interpretieren und ermöglicht es somit, mit OCL-Constraints versehene UML-Modelle zusammen mit dem Fachwissen des Modellentwicklers zu analysieren und zu validieren. Das UML-Modell kann aus Klassen mit Attributen und Operationen, Assoziationen zwischen Klassen, OCL-Constraints in Form von Invarianten (Aussagen, die immer gültig sein müssen) und Pre-/Postconditions für Operationen sowie Enumerations bestehen. Abstrakte Klassen und (Mehrfach-)Vererbung zwischen Klassen ist ebenfalls möglich. Basierend auf dem Modell unterstützt USE Klassen-, Objekt-, Zustands-, Sequenz- und Kommunikationsdiagramme, wobei für diese Arbeit nur die ersten beiden genutzt werden. USE wird an der Universität Bremen entwickelt. Die erste USE-Version war im Jahr 1998 verfügbar; die aktuelle Version ist 4.2.0.

Das UML-Modell an sich ist eine Datei mit der Dateiendung `*.use`, die die Modellbeschreibung in USE-eigener Klartext-Syntax enthält. USE verfügt über eine kommandozeilenbasierte und eine grafische Benutzerschnittstelle (GUI). Die meisten Funktionen sind über beide Schnittstellen verfügbar, wobei erstere den Vorteil automatisierter Batchverarbeitung und letztere den Vorteil der grafischen Darstellung der Diagramme bietet. Beim Öffnen eines Modells wird automatisch die Modell-Syntax – nicht die Semantik – überprüft, wobei auch auf die syntaktisch korrekte Nutzung von OCL geachtet wird. Wird die Syntax nicht eingehalten, wird die betroffene Stelle in der USE-Modell-Datei als Fehlermeldung ausgegeben. Das Modell ist anschließend nicht geladen. Somit ist sichergestellt, dass nur syntaxkonforme und wohlgeformte Modelle die Grundlage für nachfolgende Schritte sind, da sich das Modell nach dem Laden nicht mehr modifizieren lässt; die Bearbeitung des Modells ist nur über einen Texteditor möglich.

Die eigentliche Modellvalidierung findet auf Basis von Systemzuständen (Snapshots, Modellinstanzen) in Form von Objektdiagrammen statt, deren Objekte und Links instanziierte Klassen und Assoziationen des geladenen Modells sind. Über die grafische Benutzeroberfläche lassen sich Objekte unter anderem per Drag-And-Drop einer Modellklasse erzeugen und über das „Object properties“-Fenster die Attribute setzen. Links lassen sich über das Kontextmenü nach dem Markieren von zwei Objekten erstellen. Auf der Kommandozeile dienen für die genannten Schritte folgende Befehle:

- Objekt erzeugen: `!create <OBJECT NAME> : <CLASS NAME>` oder äquivalent `!new <CLASS NAME>('<OBJECT NAME>')`

- Attribut setzen: `!set <OBJECT NAME>.<ATTRIBUTE NAME> := <VALUE>`
- Link zwischen zwei Objekten erstellen: `!insert (<OBJECT NAME 1>, <OBJECT NAME 2>) into <ASSOCIATION NAME>`

USE lässt es nur zu, Objekte und Links von Klassen und Assoziationen zu erstellen, die auch im geladenen Modell existieren. Die Objekte und Links sind also stets passend zum Modell. Allerdings verblocken im Modell befindliche OCL-Constraints und Multiplizitäten weder das Erzeugen von Objekten, das Setzen von Attributen noch das Erstellen von Links, die gegen diese OCL-Constraints oder Multiplizitäten verstoßen. Das erlaubt es Soll-Objektdiagramme zu erstellen und anschließend mittels USE zu evaluieren, ob es laut Modell ein gültiger Systemzustand ist. Ebenfalls können Negativfälle von Objektdiagrammen angefertigt und dann überprüft werden, ob das Modell dies korrekterweise als ungültige Systemzustände identifiziert. Die grafische Benutzeroberfläche bietet zur Überprüfung der Einhaltung der Modellstruktur (Multiplizitäten) dafür die Menüfunktion `State` → `Check structure now`, dessen Ergebnis eine textuelle Ausgabe aller Modellstrukturverstöße im aktuellen Objektdiagramm ist. Welche OCL-Constraint-Invarianten das aktuelle Objektdiagramm erfüllt, lässt sich über das Fenster `View` → `Create View` → `Class invariants` einsehen. Für jede Invariante kann der „Evaluation browser“ geöffnet werden, der detailliert wiedergibt, wie das OCL-Constraint ausgewertet wurde und welche Objekte ggf. das OCL-Constraint verletzen. Der `check`-Befehl der Kommandozeile liefert ein kombiniertes, textuelles Ergebnis der genannten Modellstruktur- und Invariantenüberprüfung. Neben den im Model befindlichen OCL-Constraints können auch jederzeit, basierend auf dem aktuellen Objektdiagramm, beliebige OCL-Ausdrücke evaluiert werden (GUI-Fenster: `State` → `Evaluate OCL expression...`, Kommandozeile: `?? <OCL EXPRESSION>`).

Die Kommandozeilenbefehle für das Erstellen von Objekten und Links sind Teil der Programmiersprache SOIL (Simple OCL-like Imperative Language) [7], die von der Kommandozeile von USE unterstützt wird. SOIL erweitert OCL, welche selbst keine Seiteneffekte ermöglicht, um die Möglichkeit Systemzustandsmodifikationen vorzunehmen und bietet klassische imperative Konstrukte wie Variablenzuweisung und Flusskontrolle (z.B. `if` und `for`). Gespeichert werden die SOIL-Skripte im Klartext in Dateien mit der Endung `*.soil`. Ausführen lassen sich diese SOIL-Skripte durch das Öffnen und Lesen der Datei über die Kommandozeile mit dem Befehl `open <SOIL FILE>`. Objektdiagramme lassen sich somit als Sammlung von `!create`-/`!new`-, `!set`- und `!insert`-Befehlen in einem SOIL-Skript speichern und später erneut öffnen. Der Kommandozeilenbefehl `write <SOIL FILE>` schreibt das aktuelle Objektdiagramm als SOIL-Befehle in ein SOIL-Skript.

3.2 Model Validator Plugin

USE besitzt eine Softwarearchitektur, die es leicht ermöglicht den Funktionsumfang über Plugins nachträglich zu erweitern. Ein solches Plugin für USE ist der Model

Validator [14]. Der Model Validator versucht automatisch, basierend auf einem geöffneten UML-Modell und einer Konfigurationsdatei, die unter anderem die minimale und maximale Anzahl an Objekten pro Klasse und Links pro Assoziation aus dem Modell spezifiziert, *gültige* Systemzustände (Objektdiagramme) zu erzeugen, die alle Vorgaben vom Modell, wie Multiplizitäten und OCL-Constraints, erfüllen. Die Konfigurationsdatei, die genau zu den im Modell befindlichen Klassen und Assoziationen abgestimmt sein muss, ist eine Datei mit der Endung `*.properties`, die sich über die grafische Benutzeroberfläche von USE (`Plugins` → `Model Validator` → `Configuration`) generieren und editieren lässt. Da die Konfigurationsdatei eine einfache Textdatei ist, lässt sie sich aber auch mit einem einfachen Texteditor bearbeiten.

Nach dem Erstellen der Konfigurationsdatei, lässt sich der Model Validator (`mv`) über die USE-Kommandozeilenschnittstelle ausführen und steuern, wobei nur die im Rahmen dieser Arbeit benötigten Befehle nachfolgend ausgelistet sind:

- `mv -validate <CONFIGURATION FILE>` validiert, ob es möglich ist, basierend auf dem aktuellem Modell und der gegebenen Konfigurationsdatei, *einen* gültigen Systemzustand zu erzeugen. Wenn es möglich ist, ist das Ergebnis das entsprechende Objektdiagramm. Sollte es hingegen nicht möglich sein, wird ggf. ein Nichterfüllbarkeitsbeweis geliefert. Ein Beispiel für die Nichterfüllbarkeit ist, dass die maximale Anzahl Objekte einer Klasse in der Konfigurationsdatei die minimale Multiplizität einer mit der Klassen verbunden Assoziation unterschreitet. Ein weiteres, triviales Beispiel wäre ein OCL-Constraint, das immer `false` liefert.

- `mv -scrollingAll <CONFIGURATION FILE>` sucht alle gültigen Systemzustände im durch die Konfigurationsdatei aufgespannten Suchraum; im Gegensatz zu `mv -validate` wird also die Suche nicht abgebrochen, sobald der erste gültige Systemzustand gefunden wurde. Am Ende liefert der Befehl die Anzahl n gefundener, gültiger Systemzustände: Bei Nichterfüllbarkeit ist $n = 0$, andernfalls ist $n > 0$. Bei $n > 0$ kann mittels `mv -scrollingAll show(<INDEX>)` mit `<INDEX>` $\in 1, \ldots, n$ über die Lösungsmenge iteriert und das jeweilige Objektdiagramm generiert werden.

- `mv -config objExtraction:=on` weist sowohl `mv -validate` als auch den `mv -scrollingAll` an, die Systemzustände nicht komplett selbst zu generieren, sondern auf dem vor der Validierung existierenden Objektdiagramm aufzubauen. Somit können manuell Teilobjektdiagramme erzeugt werden, die dann von `mv -validate` bzw. `mv -scrollingAll` versucht werden zu komplettieren. Entsprechend muss das vorab erstellte Teilobjektdiagramm und die Konfigurationsdatei aufeinander abgestimmt sein: Ein Trivialbeispiel für Nichterfüllbarkeit ist, wenn das Teilobjektdiagramm bereits über mehr Objekte einer bestimmte Klasse verfügt als in der Konfigurationsdatei maximal zugelassen sind.

- `constraints -flags <INVARIANT> +d` deaktiviert die gegebene Invariante, so-

dass die von `mv -validate` und `mv -scrollingAll` nicht mehr berücksichtigt wird. Somit erhöht sich ggf. die Anzahl an Lösungen, die bei gleichbleibender Konfiguration gefunden werden, da nun in der Lösungsmenge auch Systemzustände enthalten sein können, die eigentlich gegen die deaktivierte Invariante verstoßen. Ein `-d` statt `+d` aktiviert die Invariante wieder.

Intern wandelt der Model Validator bei Verwendung von `mv -validate` und `mv -scrollingAll` die gegebenen Informationen in Relational Logic um und übergibt sie an den Constraint-Solver Kodkod [36][17]. Kodkod wiederum übersetzt die Relational-Logic in Boolean-Logic und überführt damit die Modellvalidierung in das boolsche Erfüllbarkeitsproblem (engl. Satisfiability Problem, kurz: SAT) [15, S. 438 ff.]. Bei SAT geht es darum, festzustellen, ob es eine Belegung der Variablen in einer boolschen Formel mit TRUE und FALSE gibt, sodass die gesamte Formel zu TRUE auswertet. Zur Lösung des Erfüllbarkeitsproblems kann Kodkod verschiedene SAT-Solver verwenden, die sich über `mv -config satsolver:=<SAT SOLVER NAME>` auswählen lassen.

Aus Sicht der Komplexitätstheorie ist SAT ein $\mathcal{NP}$-vollständiges Problem [15, S. 425 ff.]. Das bedeutet, dass es nach aktueller – allerdings nicht bewiesener – Annahme ($\mathcal{P} \neq \mathcal{NP}$) keinen Algorithmus gibt, der das SAT-Problem in einer Dauer berechnen kann, die polynomial abhängig von der Eingangsgröße ist. Stattdessen sind Probleme der Klasse $\mathcal{NP}$ abhängig von der Eingangsgröße nur in exponentieller oder noch mehr Zeit lösbar. Ein *$\mathcal{NP}$-vollständiges* Problem gehört dabei zu den schwierigsten Problemen in $\mathcal{NP}$. Entsprechend zeitintensiv können die Ausführungen des Model Validators je nach Modellgröße und Einstellungen der Konfigurationdatei werden. Drei verschiedene Übersetzungs- und Lösungszeiten werden vom Model Validator ausgegeben: initiale, einmalige Übersetzungsdauer vom USE-Modell zu Kodkod, sowie pro gefundenem, gültigen Systemzustand die Übersetzungsdauer von Kodkod zu SAT gefolgt von der Dauer, wie lange der SAT-Solver benötigte, um ein Ergebnis zu ermitteln.

4 RBAC-Ecore-OCL-Modell

Das RBAC-Ecore-OCL-Modell, welches die Grundlage für diese Arbeit ist, wird in diesem Kapitel vorgestellt. Dafür wird zunächst mit dem Eclipse Modeling Framework das Tool eingeführt, mit dem das Modell technisch realisiert wurde. Anschließend wird der Teil vom Modell vorgestellt, der nicht im Rahmen dieser Arbeit erstellt wurde, sondern vorab von der Arbeitsgruppe „Softwaretechnik“ der Universität Bremen. Die im Rahmen dieser Arbeit entworfene Modellerweiterung, die dem Modell das Konzept der Delegation hinzufügt, wird abschließend dargestellt. Delegation wird in Kapitel 6 *Fallstudie Krankenhaus* zum Einsatz kommen.

4.1 EMF (Eclipse Modeling Framework)

Das Eclipse Modeling Framework (EMF) ist ein Plugin für die Entwicklungsumgebung Eclipse, welches Modellierungsaspekte in Form von UML-Klassendiagrammen bietet. Es ist dabei aber nicht nur ein Werkzeug für Model-Driven Development (MDA) mittels Klassendiagrammen, sondern es versteht sich als Highlevel-Modell, welches die gemeinsame Basis für Java-Quellcode (Grundgerüst), XML-Schemata und UML-Klassendiagramme ist, da alle drei letztendlich die gleichen Informationen enthalten. Somit können Java-Quellcode, XML-Schemata und UML-Klassendiagramme Quelle für ein EMF-Modell sein, aber auch aus einem EMF-Modell automatisch generiert werden [35]. Dieser mächtige Funktionsumfang, den EMF bietet, wird für diese Arbeit allerdings nicht benötigt: Eine weitere Möglichkeit, EMF-Modelle zu erstellen, ist, diese direkt in Eclipse über eine Baumstruktur und Eigenschaftenliste zu definieren – so wurde es auch für den Delegationsanteil des RBAC-Ecore-OCL-Modell gemacht.

EMF-Modelle basieren auf dem Ecore-Meta-Modell, welches die grundlegenden Komponenten (wie z.B. EClass, EAttribute, EReference und EOperation) und Datentypen (beispielsweise EInt, EDouble, EString, EBoolean und EEnum) als Java-Interfaces und Java-Klassen definiert. Abbildung 4.1 gibt in einem UML-Klassendiagramm einen Überblick über die Vererbungshierarchie vom Ecore-Meta-Modell sowie der Assoziationen der Klassen zueinander.

Ein EMF-Modell lässt sich in einer einzigen Datei serialisieren, wofür das XMI-Format (XML Metadata Interchange) verwendet wird. Da EMF-Modelle auf dem Ecore-Meta-Modell basieren, sind sie selbst Ecore-Modelle und tragen entsprechend die Dateiendung `*.ecore`.

In UML gibt es neben dem Diagrammtyp „Klassendiagramm“ auch das „Objektdiagramm“, welches konkrete Instanzen, der im Klassendiagramm aufgeführten Klassen und deren Beziehungen zueinander, zeigt. Dasselbe Konzept existiert auch beim EMF: Von den Klassen im Ecore-Modell können konkrete Dynamic Instances erstellt werden, dessen Klassenattribute konkrete Werte haben und die entspre-

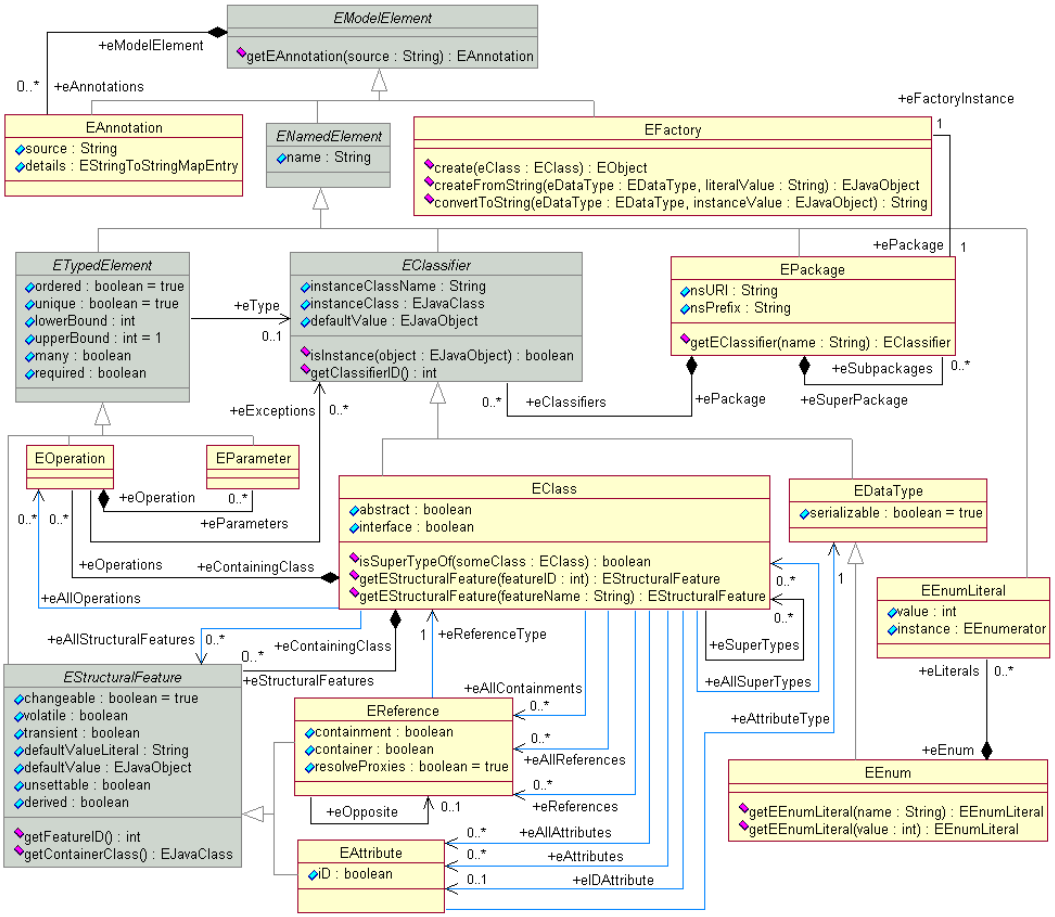


Abbildung 4.1: Ecore-Metamodell [2]

chend des Ecore-Modells Beziehungen zueinander haben können – nachfolgend Ecore-Instanz-Modell genannt. EMF überprüft dabei automatisch, ob das Ecore-Instanz-Modell die Bedingungen des Ecore-Modells erfüllt. Ein Ecore-Instanz-Modell wird in einer `*.xmi`-Datei gespeichert, die das Ecore-Modell, auf dem es basiert, referenziert und somit nur in Kombination mit der `*.ecore`-Datei verwendbar ist.

OCL-Constraints werden über `EAnnotation`, die beispielsweise einer `EClass` oder `EOperation` zugewiesen sind, ins Ecore-Modell eingefügt, indem das `source`-Attribut einer `EAnnotation` auf `http://www.eclipse.org/emf/2002/Ecore/OCL` gesetzt wird. Anschließend können unter dieser `EAnnotation` beliebig viele `Details Entries` erstellt werden, wobei jeder für ein eigenständiges OCL-Constraint steht. Der `Key` eines `Details Entrys` ist der frei wählbare Name des OCL-Constraints und der `Value` das eigentliche OCL-Contraints, das der offiziellen OCL-Syntax entsprechen muss.
Das Ecore-Modell kann zudem so konfiguriert werden, dass bei erstellten Ecore-Instanz-Modellen automatisch validiert wird, ob diese die OCL-Constraints einhalten. EMF selbst unterstützt diese Validierung nicht, erlaubt es aber, diese an OCL zu

delegieren, wofür spezielle `EAnnotation` verwendet werden:
Jedes `EPackage`, das die OCL-Validierung nutzen können soll, benötigt eine `http://www.eclipse.org/emf/2002/Ecore-EAnnotation` mit dem `Details Entry` Key-Value-Paar:
`validationDelegates -> http://www.eclipse.org/emf/2002/Ecore/OCL`.
Zusätzlich muss jedes Modellelement, welches OCL-Constraints besitzt, eine `http://www.eclipse.org/emf/2002/Ecore-EAnnotation` einem `Details Entry` Key-Value-Paar nach folgender Syntax aufweisen:
`constraints -> <OCL CONSTRAINT NAME> <OCL CONSTRAINT NAME> ...`
Nur die OCL-Constraints, deren Namen (`Key` des `Details Entry`) als `<OCL CONSTRAINT NAME>` gelistet sind, werden auch automatisch validiert.

4.2 Grundlagenmodell

4.2.1 Allgemein

Die Grundlage, auf der diese Arbeit basiert, ist das RBAC-Ecore-OCL-Modell, welches in der Arbeitsgruppe „Softwaretechnik“ der Universität Bremen modelliert wurde [5]. Ausgewählte Bestandteile und Eigenschaften dieses Grundlagenmodells, die für diese Masterarbeit relevant sind, werden nachfolgend vorgestellt. Änderungen und Erweiterungen, die im Kontext dieser Arbeit am Grundlagenmodell vorgenommen werden, beziehen sich alle auf das entwickelte Delegationskonzept, welches im nächsten Abschnitt vorgestellt wird.

Allgemein beschrieben, bietet das Grundlagenmodell den Bausatz, um mehrschrittige Geschäftsprozesse und Subjekte, die Berechtigungen über eine rollenbasierte Zugriffskontrolle erhalten, zu erstellen. Bei den Geschäftsprozessen ist zudem hinterlegt, welche Berechtigungen zur Ausführung dieser benötigt werden. Verbunden wird die Ebene der Geschäftsprozesse und die der Subjekte über die eigentliche Ausführung, die ausdrückt, welches Subjekt welche Geschäftsprozessoperation durchführt. Dabei wird überprüft, ob das Subjekt überhaupt zur Ausführung berechtigt ist.

4.2.2 `highlevel`-Paket

Das EMF-Modell ist gegliedert in vier Pakete; das Hauptpaket ist `highlevel`, welches Klassen und die drei weiteren Pakete als Unterpakete enthält. `highlevel` enthält die abstrakten Klassen `Subject`, `Operation` und `Constraint` (siehe Abbildung 4.2), wovon in den Unterpaketen fast alle Klassen erben. Das allgemeine Konzept, das darüber abgebildet wird, ist ein Subjekt (`Subject`), welches Tätigkeiten (`Operation`) ausführen kann, wenn es die Berechtigungen dafür besitzt bzw. es nicht durch eine Beschränkung blockiert wird (`Constraint`). Die eigentliche Ausführung eine Tätigkeit durch ein Subjekt, also die Kombination einer `Subject`- und `Operation`-Instanz, wird durch die Klasse `Execution` repräsentiert. Eine Abfolge von mehreren Ausführungen wird über die Klasse `Trace` abgebildet, die `Execution`-Instanzen in einer geordneten

Collection hält.

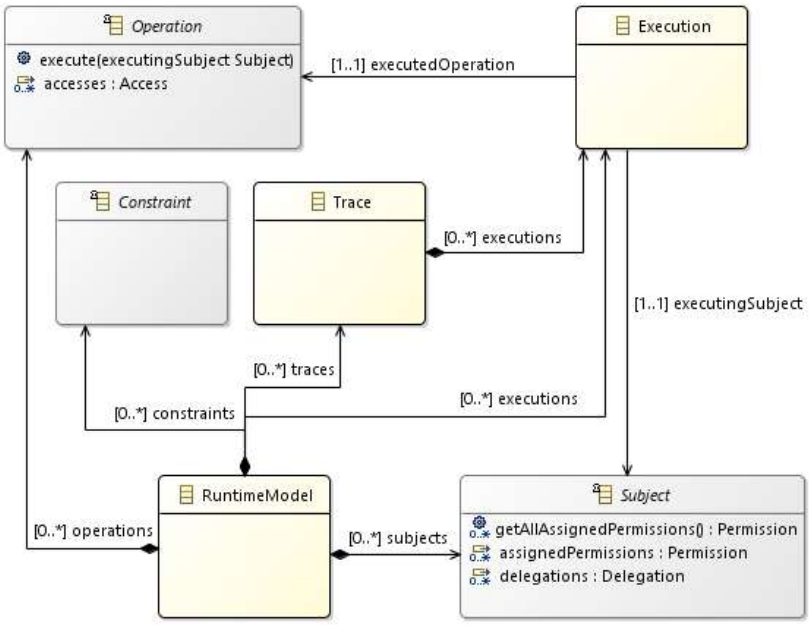


Abbildung 4.2: `highlevel`-Package Klassendiagramm

4.2.3 `permissions`-Paket

Das Unterpaket `permissions` (siehe Abbildung 4.3) modelliert ein allgemeines, von RBAC unabhängiges, Berechtigungskonzept. Für eine Berechtigung steht die Klasse `Permission`, die wiederum den Zugriff (`Access`) auf mehrere Daten (`DataAccess`) und Operationen (`OperationAccess`) ermöglicht. Für die Umsetzung des Datenzugriffs wird allerdings ein zusätzliches, auf den konkreten Anwendungsfall zugeschnittenes Datenmodell benötigt, auf dessen Nutzung im Rahmen dieser Arbeit verzichtet wurde. Stattdessen wurde neben der Zugriffskontrolle auf `Operationen` auch die Zugriffskontrolle auf Daten über `OperationAccess` abgebildet – vorstellbar als Kapselung der Daten über getter- und setter-Methoden. Des Weiteren enthält das `permissions`-Paket die von `Subject` abgeleiteten Klassen `Person` und `Login`, wobei die `Person`-Klasse auch eine `[0..*]`-Assoziation zur `Login`-Klasse besitzt. Somit lassen sich abhängig vom Einsatzumfeld und den zur Verfügung stehen Informationen unterschiedliche Subjekt-Konzepte realisieren: Personen werden direkt Berechtigungen zugewiesen, Personen besitzen Logins oder es werden direkt Logins vergeben.
Das grundlegendste OCL-Constraint des Berechtigungskonzepts ist in der `Execution`-Klasse enthalten: Das nachfolgende Listing überprüft, ob das Subjekt überhaupt die Berechtigung zur Ausführung der Tätigkeit besitzt, wobei zu beachten ist, dass das OCL-Constraint `subjectHasPermissions` im Laufe der Arbeit erweitert wird.

```
1 -- subjectHasPermissions, context: highlevel::Execution
2 executingSubject.assignedPermissions->collect(ap | ap.getAllPermissions())->
      includesAll( permissions::Permission.allInstances()->select(p | not p.
      accesses->selectByKind(permissions::OperationAccess)->select(acc | acc.
      execution = executedOperation)->isEmpty()))
```

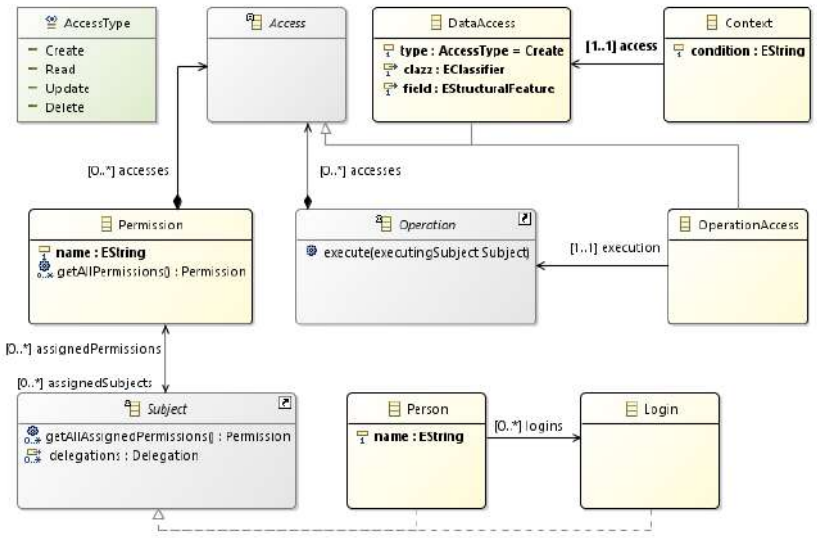


Abbildung 4.3: `permissions`-Package Klassendiagramm

4.2.4 `rbac`-Paket

Mit dem `rbac`-Unterpaket (siehe Abbildung 4.4) wird das Berechtigungskonzept um Rollen (`Role`) und rollenbasierte Konzepte zur Berechtigungsbeschränkung ergänzt. `Role` hat eine Assoziation zu `Permission`, womit die Berechtigungen, die einer Rolle zugewiesen sind, spezifiziert werden. Die Assoziationen `parents` und `children` der `Role-Klasse`, die auf die Klasse selbst verweisen, erlauben die Erstellung von Rollenhierarchien. „Separation of duty“ lässt sich über Rollenausschluss realisieren, wozu die Klassen `RoleExclusion` und `RoleExclusionCheck` dienen. `RoleExclusion` ermöglicht es allgemein, eine Menge von Rollen zu definieren, die sich gegenseitig ausschließen und `RoleExclusionCheck` kombiniert solch eine `RoleExclusion` mit einer `Person` und führt die eigentliche Überprüfung, dass die Person nicht mehr als eine Rolle aus der Rollenausschlussmenge besitzt, über ein OCL-Constraint aus:

```
1 -- check, context: rbac::RoleExclusionCheck
2 check.excludingRoles->selectByKind(Permission)->asSet->intersection(Set{person
      }->collect(logins)->collect(assignedPermissions)->asSet)->size() = 1
```

Eine Rollenvoraussetzung lässt sich über die Klasse `PrerequisiteRole` festlegen: Um die Rolle `role` zu besitzen, muss man auch die Rollen `requiredRoles` besitzen. Über `PrerequisiteRoleCheck` wird `PrerequisiteRole` mit einer Person (`Person`) verbunden und über das folgende OCL-Constraint sichergestellt, dass, wenn diese

Person die Rolle `PrerequisiteRole::role` inne hat, sie auch alle `PrerequisiteRole::requiredRoles` besitzt:

```
1 -- prerequisiteRole, context: rbac::PrerequisiteRoleCheck
2 let assignedPermissions : Set(highlevel::permissions::Permission) = person.
      getAllAssignedPermissions() in
3 if assignedPermissions->includes(check.role) then
4   assignedPermissions->includesAll(check.requiredRoles)
5 else
6   true
7 endif
```

Für jede `Person`-Instanz im Objektdiagramm, für die ein Rollenausschluss oder eine Rollenvoraussetzung gelten soll, muss also eine `RoleExlcusionCheck`- bzw. `PrerequisiteRole`-Instanz erstellt werden.
Des Weiteren ist es möglich, Rollenkardinalitäten über die Klasse `Cardinality` festzulegen. Dabei wird für eine Rolle ein Minimum und ein Maximum an Subjekten spezifiziert, die diese Rolle haben dürfen. Soll nur das Minimum oder das Maximum verwendet werden, kann der jeweils andere Wert über `-1` auf „unendlich“ gesetzt werden. Wird für eine Rolle keine Rollenkardinalität definiert, kann die Rolle beliebig häufig an Subjekte vergeben werden. Über OCL-Constraints wird überprüft, ob die Limits weder unter- noch überschritten werden. Damit diese Überprüfung logisch korrekt ist, muss sichergestellt werden, dass der Maximumwert mindestens so groß ist wie der Minimumwert.

```
1 -- minSmallerThanMax, context: rbac::Cardinality
2 if maxCount <> -1 then minCount <= maxCount else true endif
```

```
1 -- minCardinalityCheck, context: rbac::Cardinality
2 if minCount <> -1 then role.assignedSubjects->size() >= minCount else true
      endif
```

```
1 -- maxCardinalityCheck, context: rbac::Cardinality
2 if maxCount <> -1 then role.assignedSubjects->size() <= maxCount else true
      endif
```

4.2.5 `process`-Paket

Operationelle Abläufe, bestehend aus unterschiedlich granularen Schritten, werden im `process`-Paket modelliert (siehe Abbildung 4.5). Ein Geschäftsprozess (`Process`) besteht dabei aus mehreren Schritten (`Step`), die wiederum aus mehreren Eingabemasken (`Mask`) bestehen können. Die Übergänge und damit die Reihenfolge zwischen Schritten eines Geschäftsprozesses wird über `Transition`-Instanzen und einem initialen Schritt festgelegt. `Process`, `Step` und `Mask` implementieren das Interface `IStep`, welches von `Operation` erbt. Ein OCL-Constraint stellt sicher, dass modellierte Geschäftsprozesse prozesskonform bezüglich der Transitionen bei Verwendung von Eingabemasken sind: Es muss eine Transition von Step i zu Step j geben, wenn eine Eingabemaske dem Step i zugeordnet ist und auf eine nachfolgende Eingabemaske verweist, die zum Step j gehört.

4.2 Grundlagenmodell

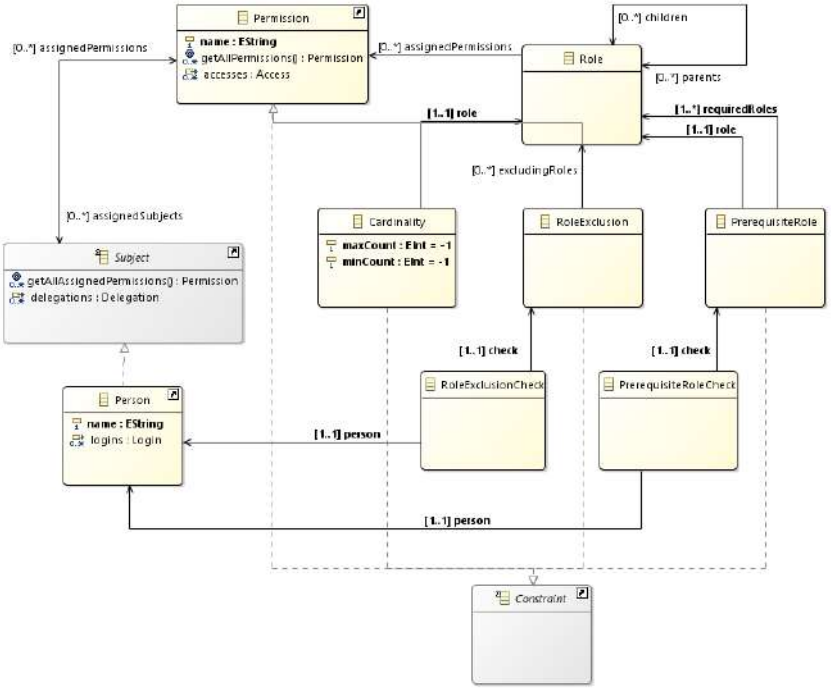


Abbildung 4.4: `rbac`-Package Klassendiagramm

```
1 -- processConformance, context: process::Process
2 steps->forAll(s|(s.masks->collect(m|m.getIncludedMasksIncludingSelf())->
      collect(m|m.successors)->collect(m|m.steps->selectByKind(IStep))->asSet()
      - transitions->select(t|t.source = s)->collect(t|t.target)->asSet())->
      isEmpty())
```

Da jeder `Process`, `Step` und `Mask` über Vererbung auch eine `Operation` ist, lassen sich über die oben vorgestellten `OperationAccess` den Geschätsprozesskomponenten Berechtigungen zuweisen, die angeben, welches Subjekt diese ausführen darf.

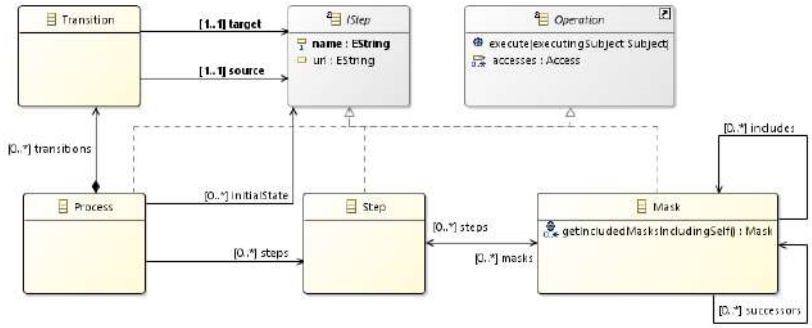


Abbildung 4.5: `process`-Package Klassendiagramm

4.3 Manuelle Modellmodifikation

Dieser Abschnitt greift eine Problematik auf, die erst in Kapitel 7 *Modellvalidierung* wirklich relevant wird, aber da das oben beschriebene Grundlagenmodell sowie auch das nachfolgende Delegationskonzept davon betroffen sind, wird die Thematik bereits an dieser Stelle behandelt: Das Grundlagenmodell kann nicht ohne Modifikation für die Validierung verwendet werden, da es OCL-Funktionalitäten nutzt, die vom USE Model Validator nicht unterstützt werden – USE an sich liefert beim Öffnen des Modells keine Fehler oder Warnungen. Genauer gesagt, kann das USE-Modell, dass in Kapitel 5 *Softwaregestützte Modelltransformation* aus dem Ecore-Modell generiert wird, nicht, wie in Abschnitt 3.2 *Model Validator Plugin* beschrieben, ohne Fehler und Warnungen in das Kodkod-Modell übersetzt werden.
Die Fehlermeldungen[2] werden durch *rekursive* Funktionsaufrufe, die nicht transformiert werden können, verursacht:

```
1 ERROR: Cannot transform invariant 'PrerequisiteRoleCheck::prerequisiterole'.
      Operation 'getAllAssignedPermissions' is recursive and thereby cannot be
      transformed.
2 ERROR: Cannot transform invariant 'Process::processconformance'. Operation '
      getIncludedMasksIncludingSelf' is recursive and thereby cannot be
      transformed.
```

`getAllAssignedPermissions()` und `getIncludedMasksIncludingSelf()` sind elementare Operationen, auf die für eine aussagekräftige Modellvalidierung nicht verzichtet werden kann. Somit müssen diese beiden Operationen in *iterative* Varianten überführt werden, die weiterhin bei gleicher Eingabe das gleiche Resultat liefern.
Bei `getAllAssignedPermissions()` entstand die Rekursion dadurch, dass `Person` und `Login` vom Typ `Subject` sind und eine Person mehrere Logins haben kann und für die Logins ebenfalls die `getAllAssignedPermissions()` aufgerufen wurde, um alle `Permissions` einer Person zusammenzusammeln. Aufgelöst wurde die Rekursion durch eine eigenständige Operation `getAllAssignedLoginPermissions()` für Logins:

```
1  -- ### VORHER (ohne Delegationskonzept) ###
2  -- getAllAssignedPermissions(), context: highlevel::Subject
3  if self.oclIsTypeOf(highlevel::permissions::Person) then
4    self.oclAsType(highlevel::permissions::Person).logins->collect(l|l.
         getAllAssignedPermissions())->union(assignedPermissions->collect(perm|
         perm.getAllPermissions()))->asSet()
5  else
6    assignedPermissions->collect(perm|perm.getAllPermissions())->asSet()
7  endif
8
9  -- ### NACHHER (inklusive Delegationskonzept) ###
10 -- getAllAssignedPermissions(), context: highlevel::Subject
11 if self.oclIsTypeOf(highlevel::permissions::Person) then
12   self.oclAsType(highlevel::permissions::Person).logins->collect(l|l.
         getAllAssignedLoginPermissions())->asSet()->union(assignedPermissions->
         collect(perm|perm.getAllPermissions())->asSet())->union(
         getAllActiveDelegationPermissions()->asSet())
```

[2] Anstatt des Stable-Release 4.2-r1 vom Model Validator wurde hier der Nightly-Build 5657 verwendet, da dieser bereits über ausgeprägtere USE-zu-Kodkod-Transformationsmeldungen verfügt.

```
13 else
14   getAllAssignedLoginPermissions()
15 endif
16 -- getAllAssignedLoginPermissions(), context: highlevel::Subject
17 assignedPermissions->collect(perm|perm.getAllPermissions())->asSet()->union(
     getAllActiveDelegationPermissions()->asSet())
```

Abbildung 4.6 zeigt einen umfangreichen Testfall in Form eines Objektdiagramms in USE, der verwendet wurde um zu evaluieren, dass die alte und neue Variante der `getAllAssignedPermissions()`-Operation das gleiche Ergebnis liefert. Die abweichenden Linknamen im Objektdiagramm resultieren aus der Ecore-zu-USE-Modelltransformation, welche in Kapitel 7 *Modellvalidierung* detailliert beschrieben wird. Das Rückgabewert ist bei beiden Operationsvarianten identisch: `Set{Permission1, Permission2, Permission3, Permission4, Permission5, Permission6, Role1, Role2, Role3, Role4} : Set(Permission)`.

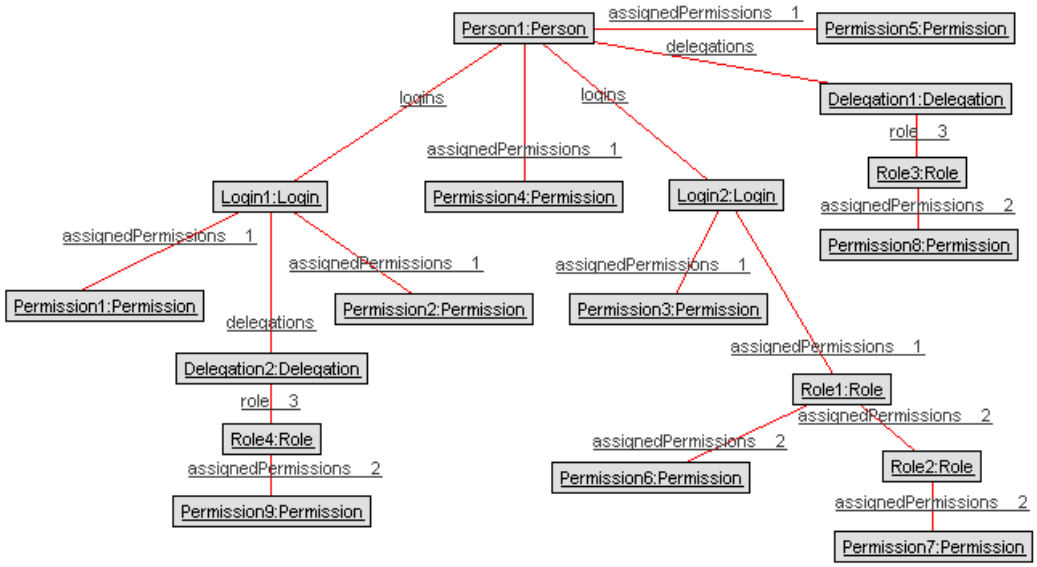


Abbildung 4.6: `getAllAssignedPermissions()`-Operation-Testfall

Der kleinste Geschäftsprozessschritt `Mask` erlaubt es wiederum Masken zu inkludieren, sodass unter einer Maske ein ganzer Baum aus Masken erstellt werden kann. `getIncludedMasksIncludingSelf()` betrachtet die Maske, auf der die Operation aufgerufen wird, als Wurzel des (Teil-)Baumes und liefert die dazugehörigen Masken inklusive der Wurzel/`self`. In der ursprünglichen Variante wird für die Traversierung durch den Baum `getIncludedMasksIncludingSelf()` rekursiv aufgerufen. Die neue Variante verwendet die OCL-Funktion `closure`, um das gleiche Ergebnis zu erziehen.

```
1 -- ### VORHER ###
2 -- getIncludedMasksIncludingSelf(), context: process::Mask
3 includes->collect(incl|incl.getIncludedMasksIncludingSelf() )->asSet()->union(
     Set{self})
4
5 -- ### NACHHER ###
6 -- getIncludedMasksIncludingSelf(), context: process::Mask
```

```
7 Set{self}->closure(children)->union(Set{self})
```

Die OCL-`closure`-Implementierung von Eclipse (Interactive OCL) und USE unterscheiden sich: Bei Eclipse ist bei `Setself->closure(children)` das Objekt `self` nicht in der Ergebnismenge enthalten, bei USE hingegen schon – nach OCL-Standard [23, S. 31] ist die USE-Implementierung die korrekte. Damit sich die `getIncludedMasksIncludingSelf`-Operation unter beiden Programmen gleich verhält, wurde das `union(Setself)` ergänzt. Dieses fügt `self` explizit zur Ergebnismenge hinzu. Da bei USE `self` bereits in der Ergebnismenge enthalten ist und ein `Set` keine Duplikate enthält, hat `union(Setself)` unter USE keine Veränderung der Ergebnismenge zur Folge.
Somit liefern Eclipse als auch USE – in der alten wie in der neuen Variante – für das in Abbildung 4.7 gezeigte Beispiel mit `self = Mask1` als Rückgabewert: `Set-Mask1,Mask2,Mask3,Mask4,Mask5 : Set(Mask)`.

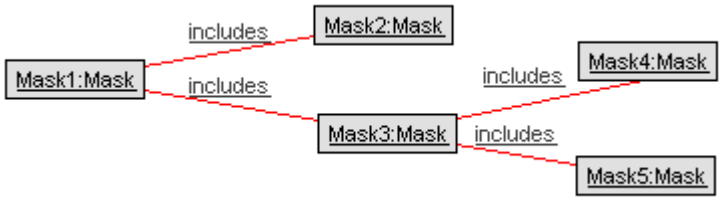


Abbildung 4.7: `getIncludedMasksIncludingSelf()`-Operation-Testfall

Alle weiteren Fehler- und Warnmeldung betreffen bestimmte OCL-Collection-Datentypen (siehe Tabelle 1) bzw. die Konvertierung von einem OCL-Collection-Datentyp in einen anderen. Der einzige von den vier OCL-Collection-Types, der unterstützt wird, ist `Set`, also eine Sammlung ungeordneter Elemente ohne Duplikate. Das Verwenden von Konvertierungsfunktionen, dessen Rückgabewert kein `Set` ist (z.B. `asBag()` und `asOrderedSet`), führt zu Fehlermeldungen der Form:

```
1 ERROR: Cannot transform invariant <INVARIANT NAME>. OCL operation asBag is not
      supported.
```

Die OCL-`collect`-Operation bzw. die implizite `collect`-Operation über die Dot-Notation, dessen Rückgabewert zwar eine OCL-Collection aber kein `Set` ist, führt zu einer Warnung und einer impliziten Typkonvertierung, z.B.:

```
1 WARN: Collect operation 'self.executingSubject.assignedPermissions->collect(ap
      : Permission | ap.getAllPermissions())' results in unsupported type 'Bag
    '. It will be interpreted as 'Set'.
```

Eine explizite Typkonvertierung in ein `Set` – `collect<EXPRESSION>->asSet()` – verhindert die Warnung nicht, weil der `collect`-Rückgabetyp weiterhin kein `Set` ist, sondern die Typkonvertierung (`asSet()`) erst im darauf folgenden Schritt durchgeführt wird.
Alle OCL-Constraints, die zu einer Warnung führen, wurden darauf hin untersucht,

ob die implizite Typkonvertierung das Soll-Verhalten ungewollt beeinflusst. Denn, wenn der Rückgabetyp kein `Set` ist, kann die Collection dasselbe Objekt mehrfach enthalten und durch die Typkonvertierung werden alle Duplikate entfernt. Je nachdem wie die Collection anschließend weiterverarbeitet werden soll, könnte dies zu einem Fehlverhalten führen, wenn z.B. anschließend die Anzahl Objekte in der Collection ermittelt wird. Allerdings wurden bei einem Review der OCL-Constraints keine Fälle gefunden, in denen sich durch die implizite Typkonvertierung das OCL-Constraint anders zu verhalten scheint, sodass wegen den Warnungen keine Änderungen am Modell vorgenommen werden mussten.

4.4 Delegation

Delegation bezeichnet die Weitergabe von Rechten von einer Entität zu einer anderen, um Zugriff auf Ressourcen zu erlangen und Aufgaben im Auftrag des Delegierenden durchzuführen zu können. Im Kontext dieser Arbeit ist mit Delegation immer Mensch-zu-Mensch-Delegation gemeint; es wird also nicht an eine Maschine (Mensch-zu-Maschine, wie beispielsweise bei der Verwendung eines Geldautomaten) oder zwischen Maschinen delegiert. [4][37]

4.4.1 Delegation in der Theorie

Nachfolgend werden verschiedene Charakteristiken des Delegationskonzepts nach [4] eingeführt, um einen Überblick zu geben. Dabei ist zu beachten, dass nicht jede Permutation der Charakteristiken ein sinnvolles Delegationskonzept ergibt:

- **Permanence:** Eine permanente Delegation führt dazu, dass der Delegierende die delegierten Rechte dauerhaft abgibt und sie anschließend auch selbst nicht mehr besitzt.
- **Monotonicity:** Dies beschreibt die Eigenschaft, ob der Delegierende die delegierten Rechte für die Dauer der Delegation auch weiterhin besitzt und sie somit für den Delegierenden gleichbleibend (monoton) sind oder ob er sie durch die Delegation temporär verliert.
- **Totality:** Bezogen auf RBAC meint es die Eigenschaft, ob jeweils nur ganze Rollen delegiert werden können oder ob auch einzelne Berechtigungen einer Rolle möglich sind.
- **Administration:** Wer kann die eigentliche Delegation der Rechte durchführen: Der Inhaber der Rechte, die delegiert werden sollen, und/oder eine dritte, administrative Stelle? Letzteres ist beispielsweise relevant, wenn der eigentliche Rechteinhaber verhindert ist.
- **Levels of delegation:** Beschreibt, ob ein Delegationsempfänger die erhaltenen Rechte ebenfalls weiter delegieren darf und wenn ja, ob die Anzahl der Ebenen begrenzt ist.

- **Multiple delegation:** Diese Eigenschaft beschreibt, an wie viele Personen der Delegierende gleichzeitig dieselben Rechte delegieren kann.
- **Agreements:** Gibt an, ob der Delegationsempfänger der Delegation zustimmen muss und somit eine Delegation auch ablehnen kann.
- **Revocation:** Widerruf der delegierten Rechte durch den Delegierenden. Je nach Delegationskonzept kann sich der Widerruf als komplexe Eigenschaft erweisen: Werden beispielsweise bei einer Multi-Level-Delegation der gesamten Delegationskette die Rechte entzogen oder muss dies von jedem Delegierenden für seine Ebene explizit durchgeführt werden?

4.4.2 Delegation im Modell

Die Klassen des Delegationskonzepts wurden im Ecore-Modell in dem eigenen, neuen Package `delegation` unterhalb von `highlevel` gekapselt. Abbildung 4.8 zeigt die dazugehörigen Klassen sowie die Annotationen, zu der auch die OCL-Constraints gehören. Die Sichtweise in Form eines Klassendiagramms auf das `delegation`-Package sowie dessen Beziehungen zu anderen Klassen des Modells gibt die Abbildung 4.9 wieder. Das gesamte Delegationsmodell ist zudem in der OCLinEcore-Darstellung in Anhang A zu finden.

`Delegation` ist die Hauptklasse, die eine Delegation *einer* Rolle von *einem* Subjekt zu *einem* anderen Subjekt beschreibt. Die weiteren Klassen des `delegation`-Packages dienen in Kombinationen mit OCL-Constraints dazu, die Mächtigkeit der Delegation einschränken zu können sowie die im vorherigen Abschnitt vorgestellten Delegationcharakteristika zu realisieren. Die Aspekte die es laut Abschnitt 4.3 *Manuelle Modellmodifikation* zu beachten gibt, damit die Modellvalidierung mittels des USE Model Validators korrekt funktioniert, sind auch in das Delegationsmodell mit eingeflossen.

Um die Komplexität nicht unnötig zu erhöhen, können nur Rollen delegiert werden und nicht die allgemeinen Berechtigungen (`Permission`), von denen Rollen (`Role`) eine Spezialisierung sind. Da Rollen hierarchisch strukturiert werden können, können die Rollen feingranular genug definiert werden, um nicht die übergeordnete Rolle und damit alle Berechtigungen auf einmal delegieren zu müssen.

Das Delegationskonzept wurde dem Ecore-Modell und nicht direkt dem USE-Modell hinzugefügt, da es über die Modelltransformation (siehe Kapitel 5 *Softwaregestützte Modelltransformation*) möglich ist, aus dem Ecore-Modell inklusive Delegationskonzept automatisch ein USE-Modell generieren zu lassen, umgekehrt allerdings nicht. Würde man das Delegationskonzept also direkt dem USE-Modell hinzufügen, würde es im Ecore-Modell fehlen, weil die Modelltransformation nur die Richtung Ecore-Modell-zu-USE-Modell unterstützt.

4.4 Delegation

Das modellierte Delegationskonzept sieht vor, dass der Delegierende nicht *dauerhaft* seine Berechtigungen über eine Delegation abtreten kann und diese dabei selbst verliert (Permanence). Grund dafür, dass diese Möglichkeit nicht modelliert ist, ist, dass sie als relativ selten vorkommend angesehen wird. Und wenn sie doch benötigt wird, kann diese Berechtigungszuweisung auch über die übergeordnete Stelle durchgeführt werden, die generell Berechtigungen (nicht Delegationen) Subjekten zuweist. So kann dem einen Subjekt einfach die Berechtigung entzogen und dem anderen Subjekt diese zugewiesen werden.
Der Delegierende besitzt nach einer Delegation auch selbst weiterhin die delegierte Rolle (Monotonicity) und der Delegationsempfänger kann eine Delegation nicht ablehnen (Agreements).

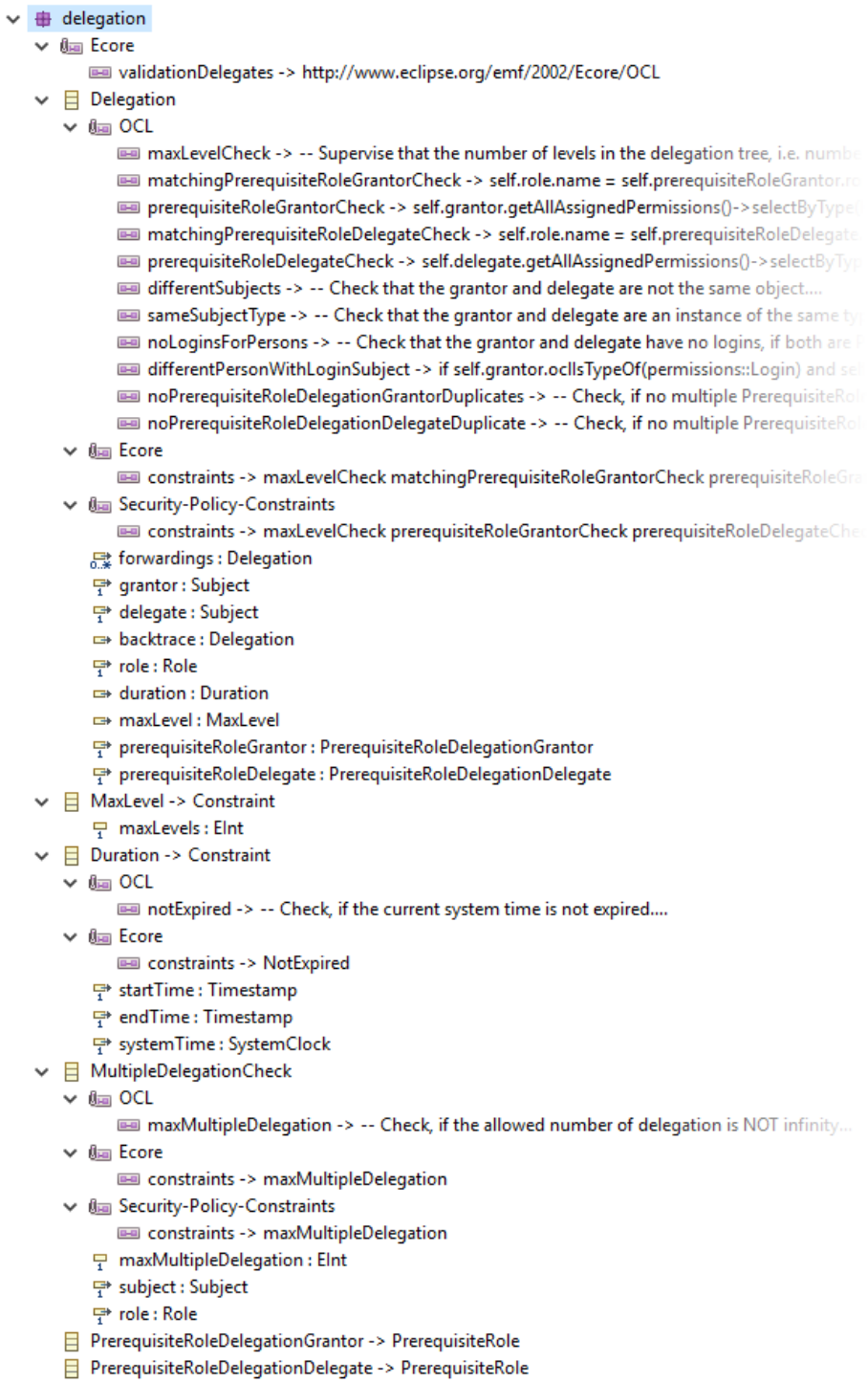


Abbildung 4.8: `delegation`-Package im Gesamtkontext (Ecore Model Editor)

4.4 Delegation

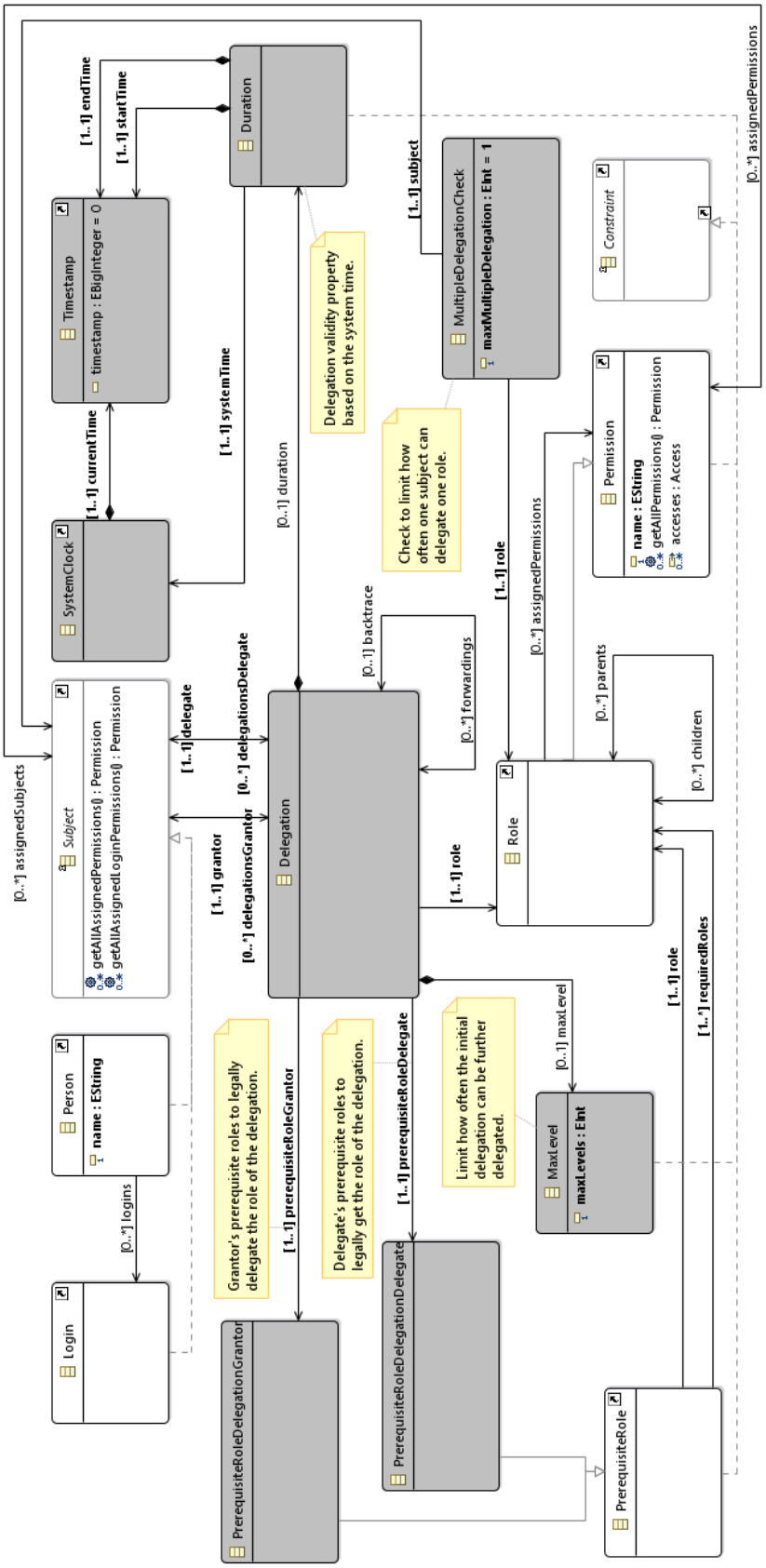


Abbildung 4.9: Delegation-Klassendiagramm. Alle grau-gefüllten Klassen wurden für das Delegationskonzept dem Modell hinzugefügt. Die weiß-gefüllten Klassen sind eine Auswahl aus dem Grundlagenmodell und wurden zum besseren Verständnis der Einbettung des Delegationskonzeptes in das Gesamtmodell mit abgebildet. Weisen Klassen einen Verknüpfungspfeil in ihrer oberen rechten Ecke auf, sind diese Klassen nicht Teil des `delegation`-Packages.

Delegierender und Delegationsempfänger: Im Grundlagenmodell ist das Subjekt über die abstrakte Klasse `Subject` modelliert, die in der konkreten Ausprägung `Person` oder `Login` sein kann, wobei eine Person mehrere Logins besitzen kann. Diese Flexibilität wurde auch für das Delegationskonzept berücksichtigt, um nicht durch eine dortige Einschränkung das Gesamtmodell einzuschränken. Entsprechend sind der Delegierende (`grantor`) und der Delegationsempfänger (`delegate`) vom Typ `Subject`, also vom konkreten Typ `Person` oder `Login`.
Allerdings muss diese Flexibilität künstlich durch OCL-Constraints eingeschränkt werden, da nicht jede Kombination von Typen zwischen Delegierendem und Delegationsempfänger einen semantisch korrekten Systemzustand ergeben würde bzw. zu Sicherheitslücken im Berechtigungssystem führen würde. Erlaubt sind daher nur die folgenden Kombinationen von Delegationen von Delegierendem zu Delegationsempfänger:

(a) Person zu Person, wenn beide Personen keine Beziehung zu Logins haben.

(b) Login zu Login, wenn keine Person eine Beziehung zu diesen Logins hat.

(c) Login zu Login, wenn jeweils eine unterschiedliche Person eine Beziehung zu beiden Logins hat.

Daraus folgt, dass es nicht erlaubt ist, dass eine Person (mit oder ohne Logins) versucht, Berechtigungen an einen Login zu delegieren, hinter dem keine Person steht bzw. umgekehrt. Denn dies würde dazu führen, dass Berechtigungen direkt Personen zugewiesen werden könnten, obwohl die Berechtigungsverwaltung über die Logins geschieht, und zudem ist dann nicht überprüfbar, ob der Delegierende und der Delegationsempfänger dieselbe Identität haben – eine Person mit mehreren Logins könnte sich damit im Extremfall, sollten keine weiteren der zur Verfügung stehenden Beschränkungen des Delegationskonzeptes genutzt worden sein, unerlaubterweise ein Login erzeugen, der die gebündelten Berechtigungen all seiner Logins besitzt.
Um die beschriebenen gewollten Einschränkungen zu gewährleisten, besitzt die `Delegation`-Klasse mehrere OCL-Constraints. Da es keinen Sinn ergibt, von einem Subjekt Berechtigungen zu ein und demselben Subjekt zu delegieren und um dies zu verhindern, wenn die delegierten Berechtigungen erst durch eine Delegation erhalten wurden, darf das Objekt (im UML-Sinn), das den Delegierenden repräsentiert, nicht dasselbe Objekt sein wie der Delegationsempfänger.

```
1 -- differentSubjects, context: delegation::Delegation
2 self.grantor<>self.delegate
```

Die Objektungleichheit ist dabei auch wahr, wenn zwei unterschiedliche Objekte identische Attributwerte und Beziehungen besitzen.
Aus der Auflistung (a-c) ist zu erkennen, dass beide Seiten der Delegation vom selben konkreten Typ sein müssen:

```
1 -- sameSubjectType, context: delegation::Delegation
2 (self.grantor.oclIsTypeOf(permissions::Person) and self.delegate.oclIsTypeOf(
      permissions::Person))
```

```
3 or
4 (self.grantor.oclIsTypeOf(permissions::Login) and self.delegate.oclIsTypeOf(
      permissions::Login))
```

Speziell für den Fall (a) muss sichergestellt werden, dass beide Personen keine Logins besitzen. Ist es nicht der Fall (a), liefert das OCL-Constraint stets `true`:

```
1 -- noLoginsForPersons, context: delegation::Delegation
2 if self.grantor.oclIsTypeOf(permissions::Person) and self.delegate.oclIsTypeOf
      (permissions::Person) then
3   self.grantor.oclAsType(permissions::Person).logins->isEmpty() and
4   self.delegate.oclAsType(permissions::Person).logins->isEmpty()
5 else
6   true
7 endif
```

Die beiden anderen Fälle erfordern, dass entweder keine Person auf beide Logins verweist (Fall (b)) oder hinter beiden Logins eine Person steht (Fall (c)). Die Schwierigkeit hierbei ist, dass zwischen Person und Login nur eine unidirektionale Beziehung besteht, die von der Person ausgeht – ein Login hat also keine Beziehung zu seiner Person (sollte denn eine dazugehörige Person existieren). Die Entscheidung, eine grundlegende Änderung am bestehenden Modell zu vermeiden und keine zusätzliche Beziehung von Login nach Person einzufügen, führt zu einem komplexeren OCL-Constraint: Zunächst werden unabhängig von der aktuellen Delegationsinstanz alle im aktuellen Systemzustand befindlichen `Person`-Instanzen ermittelt (`persons` im folgenden Listing) und anschließend festgestellt, welche Personen davon eine Beziehung zum Login des Delegierenden (`grantorPersons`) und des Delegationsempfänger (`delegatePersons`) haben. Dann wird anhand der Eigenschaft der leeren bzw. nicht-leeren Menge überprüft, ob Fall (b), Fall (c) oder ein ungültiger Fall vorliegt. Für Fall (c) muss zudem sichergestellt sein, dass der Delegierende und der Delegationsempfänger unterschiedlich Personen sind.

```
1 -- differentPersonWithLoginSubject, context: delegation::Delegation
2 if self.grantor.oclIsTypeOf(permissions::Login) and self.delegate.oclIsTypeOf(
      permissions::Login) then
3   let persons : Set(permissions::Person) = permissions::Person.allInstances()
        in
4   let grantorPersons : Set(permissions::Person) = persons->select(person |
        person.logins->includes(self.grantor.oclAsType(permissions::Login)))->
        asSet() in
5   let delegatePersons : Set(permissions::Person) = persons->select(person |
        person.logins->includes(self.delegate.oclAsType(permissions::Login)))->
        asSet() in
6   if grantorPersons->isEmpty() and delegatePersons->isEmpty() then
7     -- No Person object used on both sides (grantor and delegate) => valid
          system state
8     true
9   else
10    if grantorPersons->isEmpty() xor delegatePersons->isEmpty() then
11      -- Person objects used on one side => invalid system state
12      false
13    else
14      -- Both, grantorPersons and delegatePersons, are not empty
15      -- It is not allowed that one person delegates permission from one of
            its logins to one of its other logins (even if also other persons
            using the same login)
```

```
16        grantorPersons->intersection(delegatePersons)->isEmpty()
17      endif
18    endif
19  else
20    -- Constraint is only relevant if grantor and delegate are Login objects
21    true
22  endif
```

Delegationsberechtigungen: Das Delegationsmodell verfügt über ein eigenes System zur Überwachung, ob das delegierende Subjekt die angegebene Rolle überhaupt delegieren darf und ob das delegierte Subjekt die delegierte Rolle erhalten darf.

Der Grund dafür, dass nicht einfach festgelegt wurde, dass nur die Rollen delegiert werden dürfen, die das Subjekt auch selbst besitzt, ist, dass dadurch das Konzept der „Administration“ realisiert werden kann: Eine zentrale Stelle, die Delegationen vornehmen darf, kann grundsätzlich jede existierende Rolle delegieren, wenn diese Stelle die Berechtigung besitzt, diese Rolle zu delegieren. Die Administration ist also möglich, ohne die Rolle selbst zu besitzen, und zudem wird das Sicherheitsrisiko verhindert, dass eine Administration geschaffen werden muss, die alle Rolle besitzt, nur um diese delegieren zu können.

Um zu verhindern, dass sich die administrative Stelle einfach alle Rollen, die sie selbst nicht besitzt, aber delegieren darf, sich selbst delegiert und damit übermächtig wird, gibt es das gleiche Prinzip wie auf der delegierenden Seite auch beim Delegationsempfänger: Der Delegationsempfänger kann die delegierte Rolle nur erhalten, wenn dieser alle Rollen besitzt, die für den Erhalt der delegierten Rolle nötig sind. Dieses Vorgehen erhöht zudem ganz allgemein, unabhängig von einer zentralen Stelle, die Sicherheit, weil damit verhindert werden kann, dass, durch falsch adressierte Delegationen, Subjekte Rollen erhalten, die sie *niemals* hätten erhalten dürfen.

Realisiert wird das vorgestellte System über die Klassen `PrerequisiteRoleDelegationGrantor` und `PrerequisiteRoleDelegationDelegate`, die beide von `PrerequisiteRole` erben, um ein eigenständiger Datentyp zu sein (nachfolgend benötigt), aber keine Attribute oder Beziehungen hinzufügen. `PrerequisiteRoleDelegationGrantor.role` ist die Rolle die delegiert werden soll und `PrerequisiteRoleDelegationGrantor.requiredRoles` sind die Rollen, die nötig sind, um `PrerequisiteRoleDelegationGrantor.role` zu delegieren. Dem entsprechend ist `PrerequisiteRoleDelegationDelegate.role` die Rolle, die der Delegationsempfänger erhalten soll und `PrerequisiteRoleDelegationGrantor.requiredRoles` die Rollen, die für einen erfolgreichen Empfang vorhanden sein müssen.

Die `PrerequisiteRoleDelegationGrantor-` und `PrerequisiteRoleDelegationDelegate`-Instanzen dürfen für eine bestimmte Rolle nur einmal im Systemzustand

vorhanden sein, da es ansonsten zu Widersprüchen kommen kann, wenn für ein und dieselbe Rolle verschiedene Instanzen verschiedene Rollen als Bedingung voraussetzen. Per OCL-Constraint wird überprüft, ob es nicht mehrere `PrerequisiteRoleDelegationGrantor`-Instanzen gibt, deren `role` auf dieselbe `Role`-Instanz verweisen. Dafür werden zunächst alle `PrerequisiteRoleDelegationGrantor`-Instanzen im Systemzustand ermittelt und anschließend deren Anzahl mit der Anzahl verschiedener Rolle, auf die sie über die `role`-Assoziation referenzieren, verglichen. Dass bei den Rollen keine Duplikate enthalten sind, wird über die Konvertierung in ein `Set` sichergestellt. Unterscheiden sich die Kardinalitäten der verglichenen Mengen, müssen mindestens zwei `PrerequisiteRoleDelegationGrantor`-Instanzen auf dieselbe `Role`-Instanz verweisen, und damit ist der gesamte Systemzustand ungültig.

```
1 -- noPrerequisiteRoleDelegationGrantorDuplicates, context: delegation::
     Delegation
2 let allPrerequisiteRoleDelegationGrantors : Set(delegation::
     PrerequisiteRoleDelegationGrantor) = delegation::
     PrerequisiteRoleDelegationGrantor.allInstances()->asSet() in
3 allPrerequisiteRoleDelegationGrantors->size() =
     allPrerequisiteRoleDelegationGrantors.role->asSet()->size()
```

Das OCL-Constraint für die Rolleneindeutigkeit von `PrerequisiteRoleDelegationDelegate` ist analog dazu umgesetzt.

Jede `Delegation`-Instanz muss auf genau eine `PrerequisiteRoleDelegationGrantor`- und `PrerequisiteRoleDelegationDelegate`-Instanz verweisen. Da es, wie vorab beschrieben, keine Duplikate bei `PrerequisiteRoleDelegationGrantor`- und `PrerequisiteRoleDelegationDelegate`-Instanzen geben darf, verweisen n `Delegation`-Instanzen auf dieselbe `PrerequisiteRoleDelegationGrantor`-Instanz, wenn alle n Delegationen dieselbe Rolle delegieren. Gleiches gilt für `PrerequisiteRoleDelegationDelegate`. Ein OCL-Constraint stellt sicher, dass die von `Delegation`-Instanz assoziierte `PrerequisiteRoleDelegationGrantor`- und `PrerequisiteRoleDelegationDelegate`-Instanz auch passend zur delegierten Rolle ist:

```
1 -- matchingPrerequisiteRoleGrantorCheck, context: delegation::Delegation
2 self.role.name = self.prerequisiteRoleGrantor.role.name
```

Analog dazu wird für die Delegationsempfängerseite `self.role.name` mit `self.prerequisiteRoleDelegate.role.name` verglichen.

Werden die gerade beschriebenen Randbedingungen – bestätigt durch die OCL-Constraints – erfüllt, muss noch die eigentliche Überprüfung, ob der Delegierende und der Delegierte überhaupt die benötigten Rollen für die Delegation besitzen, positiv ausfallen. Für den Delegierenden muss dafür von der `Delegation`-Instanz ausgehend `prerequisiteRoleGrantor.requiredRoles` eine Teilmenge all seiner Rollen sein. Die Menge der Rollen des Delegierenden enthält auch rekursiv alle untergeordneten Rollen (`closure(children)`), sodass das OCL-Constraint auch erfüllt ist, wenn der Delegierende eine in der Rollenhierarchie mächtigere Rolle besitzt als für die

Delegation nötig ist.

```
1 -- prerequisiteRoleGrantorCheck, context: delegation::Delegation
2 self.grantor.getAllAssignedPermissions()->selectByType(highlevel::rbac::Role)
    ->union(self.grantor.getAllAssignedPermissions()->selectByType(highlevel::
    rbac::Role)->closure(children))->includesAll(self.prerequisiteRoleGrantor.
    requiredRoles)
```

Für den Delegierten und seine Rollen existiert ein analoges OCL-Constraint.

Mehrfachdelegation (multiple delegation): Mit einer Instanz der Klasse `MultipleDelegationCheck` kann eine Obergrenze festgelegt werden, wie häufig ein bestimmtes Subjekt eine bestimmte Rolle delegieren darf. Existiert keine solche Instanz für ein konkretes Subjekt-Rolle-Paar, so kann diese Rolle von diesem Subjekt beliebig häufig delegiert werden. Wird die Obergrenze überschritten, wird der gesamte Systemzustand als ungültig angesehen.

```
1 -- maxMultipleDelegation, context: delegation::MultipleDelegationCheck
2
3 -- Check, if the allowed number of delegation is NOT infinity
4 if self.maxMultipleDelegation <> -1 then
5   self.subject.delegationsGrantor->select(role.name=self.role.name)->size() <=
        self.maxMultipleDelegation
6 else
7   true
8 endif
```

Delegationstiefe (levels of delegation): Generell erlaubt es das Delegationsmodell, Delegationen beliebig häufig erneut zu delegieren. Dies ist nicht zu verwechseln mit der Mehrfachdelegation, bei der *ein* Subjekt eine bestimmte Rolle mehrfach an andere Subjekte delegiert. Um erneute Delegation handelt es sich, wenn Subjekt A eine Rolle an Subjekt B delegiert und Subjekt B wiederum an Subjekt C. Dies kann zusätzlich mit Mehrfachdelegation kombiniert werden, sodass Subjekt B nicht nur an Subjekt C, sondern gleichzeitig auch an Subjekt D delegieren kann. Durch mehrere Delegationsebenen und Mehrfachdelegation entsteht somit ein Delegationsbaum, dessen Wurzel die ursprüngliche Delegationsinstanz ist.
Optional kann eine Delegationsinstanz über eine `MaxLevel`-Instanz verfügen, worüber die mögliche Delegationstiefe von vornherein eingeschränkt bzw. die Weitergabe der Delegation komplett untersagt werden kann. Um die Delegationstiefe über ein OCL-Constraint überhaupt überwachen zu können, haben die Delegationsinstanzen Beziehungen zu ihrer direkten Eltern-Delegationsinstanz (`backtrace`) und ihren Kinder-Delegationsinstanzen (`forwardings`). Für die Beschränkung der maximalen Delegationstiefe ist nur die `MaxLevel`-Instanz der Wurzelinstanz relevant. Eine Delegationsinstanz, die keine Wurzel im Delegationsbaum ist, darf auch keine `MaxLevel`-Instanz besitzen, da mehrere Angaben zur maximalen Tiefe desselben Baumes widersprüchlich sein könnten.
Für die Sicherstellung, dass die Delegationstiefe nicht überschritten wird, wird für jede Delegationsinstanz wie folgt vorgegangen: Ausgangspunkt sind die Delegationsinstanzen, die ein Blatt in ihrem Delegationsbaum sind (`self.forwardings->isEmpty()`

im folgenden Listing). Von den Blättern aus kann über die `backtrace`-Beziehung eindeutig zur Wurzel navigiert werden, da nur eine Wurzel pro Delegationsbaum existiert und jeder Knoten – bis auf die Wurzel – genau einen Elternknoten besitzt. Das n-fache Nutzen der `backtrace`-Beziehung vom Blatt aus, bis die Wurzel erreicht ist, ist in einem einzigen `self->closure(backtrace)`-Aufruf gekapselt. Das besuchten Knoten befinden sich anschließend im `traceNodes`-Set. Anschließend kann die Wurzel im `traceNodes`-Set über die Bedingung `backtrace->isEmpty()` identifiziert werden. Die Anzahl der Elemente in `traceNodes` entspricht der Delegationstiefe. Dieser Wert muss gegen `maxLevel.maxLevels` der Wurzel-Delegationsinstanz verglichen werden. Besitzt die Wurzel-Delegationsinstanz keine `MaxLevel`-Instanz wird „unendlich“ als maximale Delegationstiefe angenommen und das OCL-Constraint ist `true`. Das gleiche Ergebnis liefert das OCL-Constraint, wenn die Wurzel-Delegationsinstanz hingegen genau eine `MaxLevel`-Instanz besitzt, die dort genannte Obergrenze aber nicht überschritten wird. In allen anderen Fällen liefert das OCL-Constraint `false`. Die Knoten innerhalb des Delegationsbaumes – also weder Blätter noch Wurzel – müssen nicht auf diese Weise überprüft werden, da deren Delegationstiefe immer kleiner ist, als die ihres Kindknotens und *ein* Blatt, das die Obergrenze für die Delegationstiefe überschreitet, bereits ausreicht, um den ganzen Systemzustand mit all seinen Instanzen als ungültig zu klassifizieren.

```
1 -- maxLevelCheck, context: delegation::Delegation
2 if self.forwardings->isEmpty() then
3   let traceNodes : Set(Delegation) = Set{self}->closure(backtrace)->union(Set{
        self})->selectByType(delegation::Delegation) in
4   let roots : Set(Delegation) = traceNodes->select(n | n.backtrace->isEmpty())
         in
5   if roots->size() = 1 then
6     let root : Delegation = roots->any(true) in
7     if root.maxLevel->size() = 1 then
8       traceNodes->size() <= root.maxLevel.maxLevels
9     else
10      if root.maxLevel->size() = 0 then
11        true
12      else
13        false
14      endif
15    endif
16  else
17    false
18  endif
19 else
20  true
21 endif
```

Zeitliche beschränkte Delegation: Eine Delegation kann optional mittels `Duration`-Objekt auf einen festgelegten Zeitraum eingeschränkt werden. Nur während dieses Zeitraumes sind dann die der Delegation hinterlegten Rollen auch für den Delegationsempfänger aktiv. Besitzt ein `Delegation`-Objekt kein `Duration`-Objekt, ist die Delegation solange gültig, bis die Delegation widerrufen wird.
Um den Start- und Endzeitpunkt für eine zeitliche beschränkte Delegation angeben zu

können, wurde die Klasse `Timestamp` eingeführt, die einen Zeitpunkt beispielsweise in Form eines Unix-Zeitstempels (Sekunden seit 1.1.1970) repräsentiert. Zusätzlich muss eine globale Systemzeit existieren, gegen die der Delegationszeitraum abgeglichen werden kann. Dafür wurde dem Modell die Klasse `SystemClock` hinzugefügt, die ein OCL-Constraint besitzt, welches sicherstellt, dass es nur *eine* Systemzeit gibt, da mehrere Systemzeiten widersprüchlich wären; und ein Systemzustand den Ist-Zustand zu einem ganz bestimmten Zeitpunkt repräsentiert ohne irgendeine zeitliche Dynamik:

```
1 -- isSingleton, context: higlevel::SystemClock
2 SystemClock.allInstances()->size()=1
```

`Duration` selbst besitzt kein OCL-Constraint, welches überprüft, ob die dazugehörige Delegation gerade aktiv ist oder nicht, da es ein korrekter Systemzustand ist, wenn eine zeitlich beschränkte Delegation existiert, die erst in der Zukunft aktiv wird (`SystemClock.currentTime < Delegation.duration.startTime`). Allerdings wird überprüft, ob temporäre Delegationen nicht bereits abgelaufen sind:

```
1 -- notExpired, context: higlevel::delegation::Duration
2 self.systemTime.currentTime.timestamp <= self.endTime.timestamp
```

Ob eine temporäre Delegation zu einem bestimmten Zeitpunkt aktiv ist, wird ermittelt, wenn die Berechtigungen eines Subjekts abgefragt werden. Dafür wurde in der `Subject`-Klasse die Operation `getAllActiveDelegationPermissions() : Permission` hinzugefügt, die ermittelt auf welchen Rollen aus empfangenen Delegationen, die eigentliche, bereits im Modell vorhandene Operation `getAllPermissions()` zur weiteren Berechtigungsermittlung aufgerufen werden darf:

```
1 -- getAllActiveDelegationPermissions(), context: higlevel::Subject
2 delegationsDelegate->select(d | d.duration->size() = 1)->select(d | d.duration
      .systemTime.currentTime.timestamp >= d.duration.startTime.timestamp).role.
      getAllPermissions()->union(delegationsDelegate->select(d | d.duration->
      size() = 0).role.getAllPermissions())->asSet()
```

Widerruf (revocation): Der Widerruf einer Delegation geschieht implizit: Soll eine erteilte Delegation wieder entzogen werden, so muss das `Delegation`-Objekt aus dem System entfernt werden. Es wird also nicht explizit ein Objekt erstellt, das einen Widerruf ausdrückt – der Systemzustand vor einer Delegation ist identisch mit dem Systemzustand nach dem Widerruf der Delegation. Dies gilt auch für eine zeitlich beschränkte Delegation mittels `Duration`: Ist die Delegation zeitlich abgelaufen (`SystemClock.currentTime > Delegation.duration.endTime`), aber das `Delegation`-Objekt trotzdem noch Teil des Systemzustandes, so gilt der Systemzustand als ungültig.
Durch den impliziten Widerruf ist das hier beschriebene Delegationsmodell ausdrücklich nicht darauf ausgelegt, eine protokollierende Funktion zu haben. Soll vermerkt werden, wer wann welche Berechtigungen besaß, muss dies von einer übergeordneten Instanz übernommen werden, da bereits widerrufene Delegationen nicht mehr im aktuellen Systemzustand erkennbar sind.

5 Softwaregestützte Modelltransformation

Das ursprüngliche RBAC-Modell liegt als EMF-Ecore-Modell vor und muss in ein USE-Modell transformiert werden, um es mit USE validieren zu können. Dieses Kapitel gibt zunächst eine Einführung in das Eclipse Modeling Framework (EMF) und dessen Ecore-Modelle und erklärt anschließend, wie die Transformation vom Quellmodell zum USE-Modell mittels der im Rahmen dieser Arbeit entwickelten Software Ecore2USE realisiert wurde. Die entwickelte Software unterstützt zudem die Transformation von konkreten Instanzen des Ecore-Modells in USE-Objektdiagramme, was in der nachfolgenden Beschreibung als „Instanztransformation“ bezeichnet wird. Abschließend wird das Transformationsergebnis am RBAC-Modell evaluiert.

5.1 Ziel und Motivation

Ziel der softwaregestützten Modelltransformation Ecore2USE (Ecore to USE) ist es, ein Ecore-Modell – in diesem Fall das vorgestellte RBAC-Ecore-Modell – automatisch in eine soweit wie möglich äquivalente Version in USE-Syntax [9] zu transformieren. Die `*.ecore`-Datei bleibt dabei unverändert; das Ergebnis wird in einer neuen `*.use`-Datei gespeichert.
Gleiches gilt für Ecore-Instanzen, bei denen `*.xmi`-Dateien in USE-verständliche `*.soil`-Batchdateien transformiert werden. SOIL steht für „Simple OCL-based Imperative Language“ [7] und ist eine eigenständige Programmiersprache, wobei für die Transformation allerdings nur USE-Basisfunktionen zum Erstellen von Objektdiagrammen verwendet werden.

Der Grund dafür, dass extra eine Software zur automatischen Modelltransformation im Rahmen dieser Arbeit entwickelt wurde und nicht einfach manuell die Ecore-Versionen in USE nachgebaut wurden, ist, dass so die Ecore-Modelle beliebig weiterentwickelt werden können, ohne anschließend doppelten Aufwand zu haben, um auch das USE-Modell entsprechend nachzupflegen – lediglich eine erneute, automatische Transformation ist nötig. Zudem ist es somit zukünftig auch möglich, beliebig andere Ecore-Modelle mit USE zu analysieren.

5.2 Ecore-Modelltransformation

5.2.1 User-Interface

Die Modelltransformation mittels Ecore2USE ist entweder über die in Abbildung 5.1 gezeigte grafische Benutzeroberfläche oder über die Kommandozeile, um beispielsweise Batchverarbeitung zu unterstützen, möglich.

5.2.2 Transformationsschritte

Ausgehend davon, dass die Pfade der Quell-Ecore-Datei (`*.ecore`) und der Ziel-USE-Datei (`*.use`) bekannt sind, werden folgenden Schritte durchgeführt, um das Modell

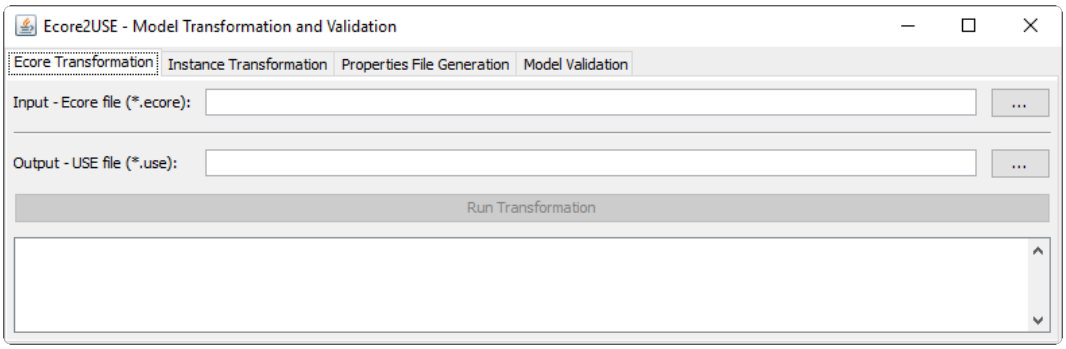


Abbildung 5.1: Ecore2USE-GUI für Ecore-Modelltransformation

zu transformieren:

1. Einlesen/Parsen des Ecore-Modells
2. Automatische Validierung des Ecore-Modells mittels EMF
3. Modellvorverarbeitung im klassischen Java-Umfeld
4. Modell-zu-Text-Transformation mittels „MOF Model To Text Transformation Language“

Im Rahmen der letzten beiden Transformationsschritte werden die nachfolgend beschriebenen Modellunterschiede zwischen Ecore- und USE-Modell gehandhabt.

5.2.3 Modellunterschiede

Das Ecore-Modell lässt sich nicht 1-zu-1 in das USE-Modell transformieren, da der Ecore- und USE-Funktionsumfang sowie die Ecore- und USE-Realisierung sich an einigen Stellen unterscheiden. Die Hauptunterschiede sind:

- In Ecore lassen sich Klassen hierarchisch in Packages gliedern; dies ist in USE nicht möglich. Die Entfernung dieser Hierarchie während der Transformation verlangt einige Anpassungen, damit das USE-Modell weiterhin wohlgeformt bleibt:
  - In unterschiedlichen Ecore-Packages können Klassen mit dem gleichen Namen abgelegt sind. Durch die Entfernung der Hierarchie kann es nun im einzigen, globalen Namensraum zu Namenskonflikten kommen, sodass Klassen umbenannt werden müssen.
  - OCL-Constraints können Pfade in der Form `package1::package2::class1` haben. Diese relativen bzw. absoluten Pfade zu Klassen müssen entfernt werden.
- Assoziationen, vor allem bi-direktional gerichtete, sind in Ecore und USE sehr unterschiedlich realisiert. In Ecore ist eine Assoziation eine sogenannte

`EReference` und einer `EClass` untergeordnet, womit die Quelle der Assoziation nicht mehr explizit definiert werden muss. Die wichtigsten Eigenschaften einer `EReference` sind somit `Name`, `EType` (Ziel der Assoziation), `Multiplicity` und, im Falle einer bi-direktional gerichteten Assoziation, der Verweis auf das jeweils andere `EReference`-Element als `EOpposite`. Bei USE hingegen sind Assoziationsdefinitionen eigenständig und somit keiner Klassendefinition untergeordnet. Ihr Aufbau ist in Listing 5.1 gezeigt.

  - Einen `associationname` gibt es unter Ecore in dieser Form nicht; zwar haben beide Richtungen einer bi-direktionalen Assoziation einen Namen, aber es existiert kein übergeordneter Name. Bei der Transformation wird als `associationname` der `EReference`-Name verwendet – bzw. bei einer bi-direktional gerichteten Assoziation, der `EReference`-Name jener `EReference`, die zuerst transformiert wurde. Sollte dieser Name im globalen Namensraum nicht eindeutig sein, bekommt der Name ein Suffix: `__i` (`i` $\in \mathbb{N}$).
  - Der `rolename` ist bei USE-Assoziationen optional; allerdings verwendet USE bei Nichtvergabe implizit den `classname` in Kleinbuchstaben als Rollenname. Daher wird der Rollenname bei der Transformation explizit angegeben, wofür der `EReference`-Name verwendet wird. Bei einfach gerichteten Assoziationen wird, wie es auch USE machen würde, als Rollenname der Quelle der Klassenname genutzt, allerdings zur eindeutigen Identifizierung mit einem Suffix: `__i` (`i` $\in \mathbb{N}$).

- Einige Klassennamen, die in Ecore zulässig sind, sind Syntax-Keywords in USE und somit als Klassennamen ungültig. Solche Klassen müssen umbenannt werden; sie bekommen einen `__`-Suffix.

Listing 5.1: USE Assoziationsdefintion [9]

```
1 (association | composition) ⟨associationname⟩ between
2     ⟨classname⟩ [ ⟨multiplicity⟩ ] [ role ⟨rolename⟩ ] [ ordered ]
3     ⟨classname⟩ [ ⟨multiplicity⟩ ] [ role ⟨rolename⟩ ] [ ordered ]
4 end
```

5.2.4 Modell-Parsing

Ecore ist in der Eclipse-Welt ein Standard, weshalb viele Softwarekomponenten Ecore bereits unterstützen. So reicht der Quellcode in Listing 5.2 aus, um die `*.ecore`-Datei (Ecore-Modell) zu parsen und somit das komplette Ecore-Modell in Form eines `EPackage` vorliegen zu haben.

Listing 5.2: Ecore-Modell parsen

```
1 ResourceSet resourceSetEcore = new ResourceSetImpl();
2 resourceSetEcore.getResourceFactoryRegistry().getExtensionToFactoryMap().put("
      ecore", new EcoreResourceFactoryImpl());
3 URI ecoreFileURI = URI.createURI("file:///" + ecoreModelPath);
```

```
4 Resource resourceEcore = resourceSetEcore.getResource(ecoreFileURI, true);
5 EObject rootElem = resourceEcore.getContents().get(0);
6 EPackage rootEPackage = (EPackage) rootElem;
```

5.2.5 Automatische Modellvalidierung

Bevor die eigentliche Transformation startet, wird zunächst die EMF-`Diagnostician`-Klasse verwendet, um automatisch zu validieren, ob das Modell laut Ecore-Syntax wohlgeformt ist. Diese Validierung entspricht dem im Eclipse-Kontextmenü befindlichen „Validate“ bei einer `*.ecore`-Datei. Ergibt die Validierung Fehler im Modell, wird die Transformation nicht gestartet, sondern dem Benutzer das Validierungsergebnis ausgegeben, da eine Transformation nur zu einem ebenfalls fehlerhaften USE-Modell führen würde; sowie die Sonderfallbehandlungen während der Transformation diese deutlich komplexer gestalten würde.

5.2.6 Modellvorverarbeitung

Um mit den vorab beschriebenen Modellunterschieden bei der Transformation umgehen zu können, werden einige der genannten Punkte bereits im Java-Umfeld am eingelesenen Ecore-Modell angepasst, bevor das Ecore-Modell an die `Ecore2UseMof2Text`-Klasse übergeben wird. Diese Vorverarbeitung ist nötig, da die Komplexität der Transformation durch die Modellunterschiede stellenweise relativ hoch ist und sich nur schwer bzw. gar nicht mit dem Mof2Text-Sprachumfang realisieren lässt.
Mof2Text iteriert über das Modell und wendet die Transformationstemplates auf die einzelnen Datentypen an. Um aber beispielsweise Konflikte im Namensraum zu finden, müssen Daten für Namensvergleiche zwischengespeichert werden, wofür Mof2Text nicht vorgesehen ist. Entsprechend werden die meisten Namenssuffixe bereits im klassischen Java-Umfeld ergänzt. Ein weiteres Beispiel sind bi-direktional gerichtete Assoziationen, wo bei der Transformation zwei `EReference`s zu einer USE-Assoziation zusammengefasst werden: Es darf also nicht aus jeder der beiden `EReference`s eine USE-Assoziation entstehen. Deswegen wird im klassischen Java-Umfeld der Name einer der beiden `EReference`s mit dem Suffix `ECORE2USE_IGNORE_ME_` versehen, auf den dann Mof2Text prüft.

5.2.7 Modell-zu-Text-Transformation

Ein USE-Modell wird im Klartext in einer Textdatei gespeichert und ist darauf ausgelegt, einfach per Editor editiert zu werden – das Ziel der Transformation ist also eine einfache Textdatei. Genau für eine solche Transformation wurde von OMG die „MOF (Meta Object Facility) Model To Text Transformation Language“ [22] (kurz: Mof2Text) definiert. Die entwickelte Software verwendet das Eclipse-Plugin Acceleo [1] für die Modell-zu-Text-Transformation, da es zum einen den offiziellen Mof2Text-Standard realisiert und zum anderen das Ecore-Meta-Modell unterstützt.

Als einführendes Beispiel von Mof2Text soll die Transformation von Enumerations (Enum) dienen. Die nachfolgende Grammatik zeigt die USE-Syntax-Definition für ein Enum [9]:

⟨*enumdefinition*⟩ ::= `enum` ⟨*enumname*⟩ `{` ⟨*elementname*⟩ { `,` ⟨*elementname*⟩ } `}`
⟨*enumname*⟩ ::= ⟨*name*⟩
⟨*elementname*⟩ ::= ⟨*name*⟩
⟨*name*⟩ ::= (⟨*letter*⟩ | `_`) { ⟨*letter*⟩ | ⟨*digit*⟩ | `_` }
⟨*letter*⟩ ::= `a` | `b` | ... | `z` | `A` | `B` | ... | `Z`
⟨*digit*⟩ ::= `0` | `1` | ... | `9`

Ein der gezeigten Grammatik entsprechendes Enum wäre:

```
enum AccessType {Create, Read, Update, Delete}
```

Listing 5.3 zeigt, wie eine Liste von Ecore-Repräsentationen eines Enums (`EEnum`) per Acceleo/Mof2Text in die USE-Syntax transformiert wird. Ein `template` kann dabei als eine Funktionsdefinition angesehen werden.
Mof2Text funktioniert nach dem WYSIWYG-Prinzip: Alles was nicht in `[...]` geschrieben ist (inkl. Whitespaces), wird direkt als Text gewertet, wie beispielsweise `enum`. Alle Elemente in `[...]`, die keine Keywords wie etwa `for` oder `if` sind, müssen von einem Datentyp sein, der eine Textrepräsentation erlaubt. So ist `eEnum.name` beispielweise vom Typ `EString`.

Listing 5.3: Acceleo-Mof2Text-EEnum-Beispiel

```
1 [template public generateEnums(eEnums : Sequence(EEnum))]
2 [for (eEnum : EEnum | eEnums)]
3 enum [eEnum.name /] {[for (eEnumLit : EEnumLiteral | eEnum.eLiterals)
      separator (', ')][eEnumLit /][/for]}
4 [/for]
5 [/template]
```

Abbildung 5.2 gibt einen Überblick über die Gesamtheit der Templates, um das Ecore-Modell in ein USE-Modell zu transformieren; `generateUSEModel` ist dabei der zentrale Einstiegspunkt, in dem auch auf Dateisystemebene eine neue Datei für das USE-Modell erstellt wird.

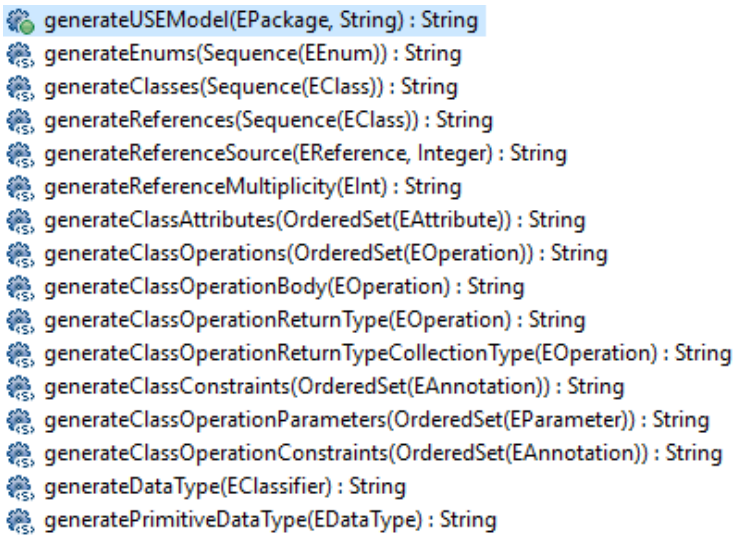


Abbildung 5.2: Übersicht aller Acceleo/Mof2Text Templates

Das Eclipse-Acceleo-Projekt generiert automatisch eine Java-Klasse, die als Wrapper für die Templates dient, sodass die Modell-zu-Text-Transformationen, wie in Listing 5.4 dargestellt, aus Java heraus gestartet werden kann. Übergabeparameter an die Transformation sind das Ecore-Modell in Form des Wurzelknotens, das Verzeichnis, in dem das Ergebnis gespeichert werden soll und im Rahmen einer `arguments`-Liste der Dateiname, der zu erstellenden `*.use`-Datei.

Listing 5.4: Start Acceleo-Modell-Transformation

```
1 List<String> arguments = new ArrayList<String>();
2 arguments.add(useModelFilename);
3 Ecore2UseMof2Text generatorEcore2Use;
4 generatorEcore2Use = new Ecore2UseMof2Text(rootEPackage, useModelDirectory,
      arguments);
5 generatorEcore2Use.doGenerate(new BasicMonitor());
```

5.3 Instanztransformation

5.3.1 User-Interface

Die Transformation einer Instanz geschieht, wie auch die Ecore-Modelltransformation, über Ecore2USE und ist entweder über den in Abbildung 5.3 gezeigten Tab der grafischen Benutzeroberfläche oder über die Kommandozeile möglich.

5.3.2 Transformationsschritte

Für die Instanztransformationen werden die folgenden drei Dateien benötigt: die Ecore-Datei (`*.ecore`), die aus der Ecore-Datei transformierte USE-Datei (`*.use`) und die Instanz-Datei (`*.xmi`). Folgende Schritte werden bei der Transformation durchlaufen:

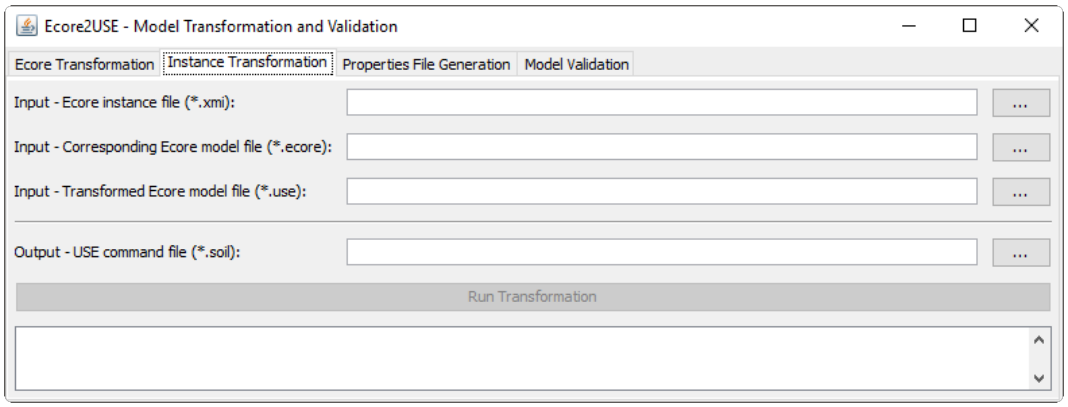


Abbildung 5.3: Ecore2USE-GUI für Instanztransformation

1. Einlesen/Parsen des Ecore-Modells und der Instanz-Datei sowie das Verknüpfen dieser beiden
2. Einlesen/Parsen des aus dem Ecore-Modell vorab transformierten USE-Modells
3. Erstellen von Befehlen zur Erstellung von USE-Objekten und USE-Assoziationen entsprechend dem Ecore/Instanz-Modell

Ergebnis dieser Transformation ist eine SOIL-Datei (`*.soil`), die `!create`-/`!set`-Befehle zum Erstellen von USE-Objekten enthält sowie `!insert`-Befehle, um eine Assoziation zwischen zwei vorab erstellen Objekten zu erzeugen. Diese Datei kann nach dem Öffnen des transformierten Ecore-Modells in USE über die USE-Kommandozeile mit einem Befehl komplett eingelesen werden, wodurch ein entsprechendes Objektdiagramm in USE generiert wird.
Im Rahmen der letzten beiden Schritte werden auch Unterschiede zwischen beiden Repräsentationen gehandhabt, die nachfolgend beschrieben werden.

5.3.3 Modellunterschiede

- In USE müssen Objekte einen Namen tragen – nicht zu verwechseln mit einem Klassenattribut `name`. Auf Ecore-Seite ist es hingegen nicht möglich, einen solchen Namen zu vergeben. Bei der Transformation muss also ein eindeutiger Name generiert werden. Damit der neue Name sprechend ist, wird der Name des Objekttyps verwendet und für die Eindeutigkeit dieser durchnummeriert.
- Bei der vorangegangenen Transformation des Ecore-Modells in ein USE-Modell kann es zu Umbenennungen gekommen sein, um Eindeutigkeit im nun einzigen, globalen Namensraum zu schaffen. Entsprechend müssen auch die neuen Namen in der SOIL-Datei verwendet werden, um mit dem USE-Modell kompatibel zu sein.
- Wie bei Programmiersprachen üblich – und somit auch bei SOIL – müssen Variablen erst deklariert werden, bevor sie verwendet werden können. Dies ist

relevant, wenn es um die Transformation von Assoziationen geht, da dort unter SOIL die Objektnamen verwendet werden, um die Assoziation zu befüllen. Entsprechend müssen zunächst die Objekte erzeugt werden. Bei der Ecore-Instanz hingegen ist innerhalb der Datei keine bestimmte Reihenfolge erforderlich.

Listing 5.5 zeigt ein SOIL-Beispiel: `Person1` ist eine Instanz vom Typ `Person`, deren Attribut `name` auf den Wert `Alice` gesetzt wird. Dann wird ein weiteres Objekt namens `Login1` erzeugt. Abschließend werden beide Objekte einer Assoziation hinzugefügt. Dafür musste der Transformationsalgorithmus ermitteln, dass die eigentlich `assignedPermissions` genannte Assoziation im Rahmen der Ecore-Transformationen umbenannt wurde und das Suffix `__1` erhalten hat.

Listing 5.5: USE-`*.soil`-Beispiel

```
!create Person1 : Person
!set Person1.name := 'Alice'
!create Login1 : Login
!insert (Person1,Role1) into assignedPermissions__1
```

5.3.4 Instanz-Modell-Parsing

Analog zum Parsen von Ecore-Modellen, werden auch die Instanz-Dateien (`*.xmi`) geparst, wie Listing 5.6 zeigt. Die in Listing 5.7 gezeigten `/i`-Referenzen werden dabei automatisch aufgelöst, wobei `i` für die `i`-te Instanz, vom `*.xmi`-Dateianfang aus gezählt, steht. Dabei zu beachten ist, dass `*.xmi`-Dateien nicht eigenständig sind, weshalb stets auch das dazugehörige Ecore-Modell benötigt wird. Die Verlinkung zwischen beiden Modellen geschieht über Namespace-URIs, die für `EPackage`s vergeben werden.

Listing 5.6: Ecore-Instanz-Modell parsen

```
resourceSetEcore.getResourceFactoryRegistry().getExtensionToFactoryMap().put("
    xmi", new XMIResourceFactoryImpl());
EPackage.Registry.INSTANCE.put(rootEPackage.getNsURI(), rootEPackage);
for (EPackage subEPackage : rootEPackage.getESubpackages()) {
    EPackage.Registry.INSTANCE.put(subEPackage.getNsURI(), subEPackage);
}
URI xmiFileURI = URI.createURI("file:///" + xmiModelPath);
Resource resourceXmi = resourceSetEcore.getResource(xmiFileURI, true);
for (EObject eObj : resourceXmi.getContents()) {
// Handle every Ecore instance defined in the *.xmi file
}
```

Listing 5.7: Ecore-Instanz-Format-Beispiel

```
<permissions:Person assignedPermissions="/0" name="Alice" logins="/4"/>
```

5.3.5 USE-Modell-Parsing

Die Bestandteile des USE-Modells, die für die Instanztransformation relevant sind, werden über reguläre Ausdrücke beim Lesen der `*.use`-Datei herausgefiltert und für die spätere Nutzung in Maps verwaltet.

5.3.6 SOIL-Befehle generieren

Aufgrund der Einfachheit des Aufbaus der wenigen benötigten USE-SOIL-Befehle wurde bei der Instanztransformation auf eine Nutzung von Mof2Text, wie bei der Ecore-Modelltransformation, verzichtet. Stattdessen wird eine Menge von eigens definierten `UseClassObj`-/`UseClassObjAttribute`-Objekten (für `!create`-/`!set`-Befehle) und `UseAssociationObj`-Objekten (für `!insert`-Befehle) beim Iterieren über `DynamicEObjectImpl` und Verknüpfungen mit vorab geparsten Daten erstellt, dessen `toString()`-Methode entsprechend der USE-SOIL-Befehlssyntax überladen wurde.

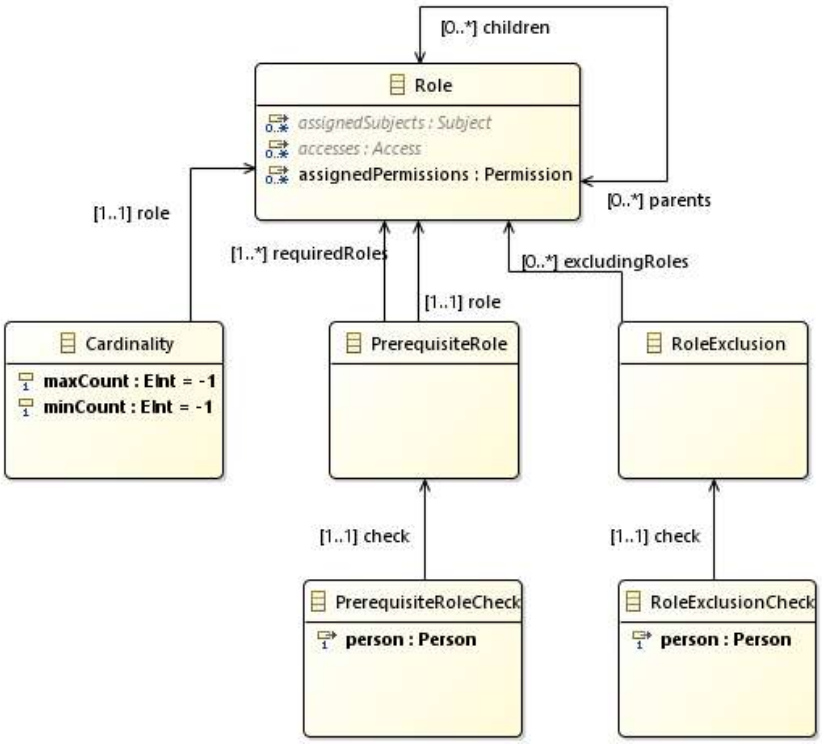
5.4 Transformationstests

Abgeschlossen wird dieses Kapitel mit der Evaluation der Transformation: Beschreibt das transformierte Modell (USE-Modell/SOIL) inhaltlich dasselbe wie das ursprüngliche Ecore-Modell/Instanz? Dies ist gleichzeitig als Entwicklertest anzusehen, da auf Grund der hohen Komplexität und des hohen Zeitaufwandes von automatisierten Tests, wie beispielsweise Unit-Tests, für Ecore2USE von diesen abgesehen wurde.

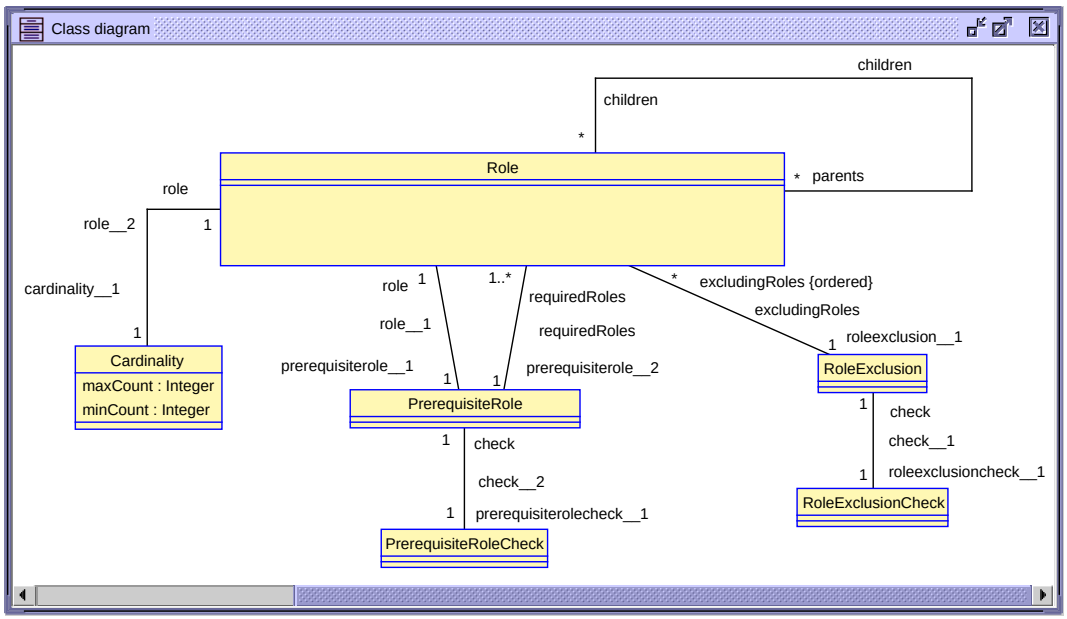
5.4.1 Ecore-Transformation

Um die Beschreibung übersichtlich zu halten, wird zur Evaluation der Ecore-Transformation nur das `rbac`-Unterpaket betrachtet. Abbildung 5.4 stellt das Ecore-Quellmodell und das daraus hervorgegangene USE-Modell gegenüber und zeigt, dass diese inhaltlich identisch sind und die Transformation damit erfolgreich war. Die rein syntaktischen Unterschiede sind die folgenden:

- Kardinalitätsnotation: `*` steht für `[0..*]` und `1` für `[1..1]`
- USE-Datentypen für `Cardinality`-Attribute (`Integer` statt `EInt`). Standardwerte für Attribute sind unter USE nicht möglich.
- Die exakten Ecore-EReference-Namen sind unter USE Rollennamen, da der Assoziationsname eindeutig sein muss. Daher hat beispielsweise der Assoziationsname `role`, da dieser doppelt im Ecore-Modell vorkommt, einen Suffix bekommen: `role__1` und `role__2`. Die Rollennamen mit einem `__i`-Suffix (`i` $\in \mathbb{N}$) wurden bei der Transformation neu erzeugt, da es diese so nicht im Ecore-Modell gibt.



(a) Original `rbac`-Unterpaket des Ecore-Modells



(b) Transformationsresult: `rbac`-Klassen des USE-Modell. Bei den Assoziationen sind die Rollennamen jeweils an den Enden und der Assoziationsname mittig eingeblendet.

Abbildung 5.4: Gegenüberstellung des Quellmodells und des transformierten Modells. Jegliche Beziehungen zu weiteren Klassen sind der besseren Übersicht halber ausgeblendet.

5.4.2 Instanztransformation

Das einfache Instanz-Modell für den Test in Listing 5.8 repräsentiert eine Person namens `Alice`, die einen Login besitzt. Das Ergebnis der Transformation (SOIL-Datei) zeigt Listing 5.9, wo zu erkennen ist, wie die identische Objektkonstellation für USE-Seite schrittweise generiert wird.

Listing 5.8: Ecore-Instanz-Modell-Transformation-Quelle

```
1 <permissions:Person name="Alice" logins="/1" />
2 <permissions:Login />
```

Listing 5.9: Ecore-Instanz-Modell-Transformation-Ziel

```
1 !create Person1 : Person
2 !set Person1.name := 'Alice'
3 !create Login1 : Login
4 !insert (Person1,Login1) into logins
```

6 Fallstudie Krankenhaus

Das in Kapitel 4 *RBAC-Ecore-OCL-Modell* vorgestellte Modell ist allgemein und flexibel gehalten, um es domänenunabhängig einsetzen zu können. Bei der Validierung des Modells mittels des USE-Tools müssen allerdings konkrete Instanzen der Modellklassen erzeugt werden. Anstatt eine Sammlung nichtssagender Instanzen zu validieren, wurde das Modell auf ein realitätsnahes Szenario angewendet: In der Fallstudie werden Geschäftsprozesse aus dem alltäglichen Krankenhausbetrieb abgebildet. Als Grundlage dafür dient die Diplomarbeit „Rollenbasiertes Sicherheitskonzept für Krankenhäuser unter Berücksichtigung der aktuellen Entwicklungen in der Gesundheitstelematik“ [12], die 2007 in der Arbeitsgruppe „Rechnernetze“ der Universität Bremen erstellt wurde. Die Diplomarbeit hat über Interviews mit Krankenhauspersonal im Raum Bremen den Ist-Zustand der Prozessabläufe ermittelt und anschließend Verbesserungsvorschläge vor allem im Hinblick auf die IT-Sicherheit und den damit verbundenen Datenschutz der Patientenakten erarbeitet. Neben den Prozessen werden dort auch die Subjekte, Berechtigungen und Delegationen in dem rollenbasierten Konzept erläutert. Basierend auf der Diplomarbeit und dem Modell wurden Objektdiagramme modelliert.

Über die Fallstudie soll in Kapitel 7 *Modellvalidierung* exemplarisch untersucht werden, ob das RBAC-Modell inklusive des Delegationskonzepts allgemein und flexibel genug ist, um einen Sachverhalt zu modellieren, für den es nicht direkt entwickelt wurde. Zugleich darf die IT-Sicherheit unter dem allgemeinen Modellansatz nicht leiden, indem Systemzustände vom Modell und deren OCL-Constraints als korrekt angesehen werden, obwohl es diese nicht sind.

Für die Fallstudie wird angenommen, dass die Verwaltung der Patientenakte komplett digital geschieht und Krankenhauspersonal, wenn es die nötigen Berechtigungen besitzt, auf die Patientenakte über eine Krankenhausinformationssystemsoftware (KIS) zugreifen kann. Um die Identität des KIS-Benutzers festzustellen, muss sich der Benutzer vorab authentisieren (beispielsweise über Benutzernamen und Passwort). Der Patient selbst wird in den Objektdiagrammen nicht dargestellt, da die Objektdiagramme die Geschäftsprozesse aus Sicht des Krankenhauses modellieren und der Patient dort keinerlei Berechtigungen besitzt. Der Patient muss in diesem Szenario nicht als Person, sondern als ein Datensatz (Patientenakte) angesehen werden, wobei das Berechtigungssystem dafür sorgen soll, dass das Krankenhauspersonal immer nur auf die Patientenaktenteile zugreifen darf, für die es eine Berechtigung besitzt.

Aufgeteilt auf die zwei Beispiele „Patientenaufnahme“ und „Konsil“ wird der Funktionsumfang des Modells in möglichst einfachen Objektdiagrammen gezeigt, um die Komplexität nicht unnötig zu erhöhen. Dem Objektdiagramm könnten viele weitere Personen, Logins, Rollen, Berechtigungen, usw. hinzugefügt werden, dies würde aber nicht dazu beitragen, das Modell bzw. die OCL-Constraints besser zu

validieren. Der Umfang des Objektdiagramms hat allerdings Auswirkungen auf die Dauer der softwaregestützten Modellvalidierung, worauf in Abschnitt 7.5 *Ergebnisse* eingegangen wird.

6.1 Patientenaufnahme

Der Geschäftsprozess „Patientenaufnahme“ ist in Abbildung 6.1 als Objektdiagramm basierend auf dem RBAC-Modell dargestellt und beschreibt den Vorgang, dass sich ein Patient beim Empfang meldet, der bereits früher einmal das Krankenhaus aufgesucht hat – somit muss keine neue Patientenakte angelegt werden. Die Empfangsmitarbeiterin Alice liest die Patientenakte und aktualisiert die *administrativen* Daten (z.B. Adresse des Patienten), hat dabei allerdings keinen Zugriff auf die medizinischen Daten. Alice verweist den Patienten auf Station 1. Auf Station 1 arbeitet Bob als Assistenzarzt, entsprechend hat er die gleichnamige Rolle, die es ihm erlaubt, auf Patientenakten von Patienten zuzugreifen, die sich auf Station 1 befinden.

Das Objektdiagramm repräsentiert einen *validen* Systemzustand: Laut USE verstößt die Struktur nicht gegen das darunterliegende Modell, wozu gehört, dass alle definierten Multiplizitäten eingehalten werden (z.B. ist bei einer `[1..*]`-Assoziation auch mindestens ein Link vorhanden). Zudem werden keine OCL-Constraints verletzt: USE Class Invariant View zeigt an, dass alle Invarianten erfüllt sind (satisfied = true).

Dieser Ablauf ist über `Step`-Instanzen realisiert, wobei die eigentliche Tätigkeit in den Eingabemasken (`Mask`) als unterste Geschäftsprozesseinheit durchgeführt wird. Die `Step`-Instanzen sind über `Transition`-Instanzen miteinander verbunden, woraus sich ergibt, welcher `Step` als Nächstes folgen kann. Der Geschäftsprozess besteht somit aus einer `Process`-Instanz, die Links zu `Step`- und `Transition`-Instanzen besitzt und per Link angibt, welche `Step`-Instanz der initiale Schritt ist.

Personen (`Person`-Instanzen) besitzen `Login`-Instanzen, die das eigentliche Benutzerprofil, mit dem man sich beim Krankenhausinformation anmeldet, repräsentieren. Diesen `Login`-Instanzen sind Rollen (`Role`-Instanzen) zugewiesen. Die `Role`-Instanzen wiederum verweisen auf die eigentlichen Berechtigungen (`Permission`-Instanzen), die ein Benutzer durch Zuweisung der Rolle erhält.

Um festzulegen, welche Berechtigungen nötig sind, um einen Geschäftsprozessschritt auszuführen, dienen `OperationAccess`-Instanzen zwischen `Step`- und `Permission`-Instanzen.

Die beiden Rollen „Empfangsmitarbeiter“ und „Assistenzarzt“ sind mit einem Rollenausschluss versehen: Die `RoleExclusion`-Instanz definiert, welche `Role`-Instanzen sich gegenseitig ausschließen und die `RoleExlusionCheck`-Instanzen übernehmen die konkrete Überprüfung auf Einhaltung des Rollenausschlusses bei einer `Person`-

Instanz. Ohne den modellierten Rollenausschluss wäre die Beschränkung, dass ein Arzt nur die Patientenakten seiner Stationen einsehen kann, umgehbar, wenn eine Person beide Rollen inne hätte, indem er selbst Patienten seinen Stationen zuweist. Des Weiteren wird die Rollenkardinalität (`Cardinality`-Instanz) verwendet, um die Anzahl Subjekte, die die jeweilige Rolle besitzen, einzuschränken. In diesem Beispiel wurde die untere und obere Grenze für die Rolle „Assistenzarzt“ auf 1 gesetzt, sodass es genau zur Anzahl Subjekte passt. Eine weiteres Subjekt, welches die Rolle „Assistenzarzt“ zugewiesen bekommen würde, würde einen detektierbaren Verstoß darstellen.

Alle bis jetzt vorgestellten Bereiche des Objektdiagramms beschreiben den quasi-permanenten Zustand des Systems: Die Personen und Berechtigungen werden sich nur ändern, wenn z.B. ein Arbeitnehmer die Abteilung wechselt oder das Krankenhaus verlässt. Geschäftsprozesse mit ihren Arbeitsschritten werden sich vermutlich noch seltener ändern, da dies bedeuten würde, dass sich z.B. die Abläufe in der Krankenhausinformationssystemsoftware geändert haben. Die Instanzen, die die alltägliche Nutzung des Systems darstellen und abhängig davon sind, welche Person welchen Geschäftsprozessschritt ausführt, sind die `Execution`-Instanzen, deren Abfolge durch eine `Trace`-Instanz zusammengefasst wird.

6.1 Patientenaufnahme

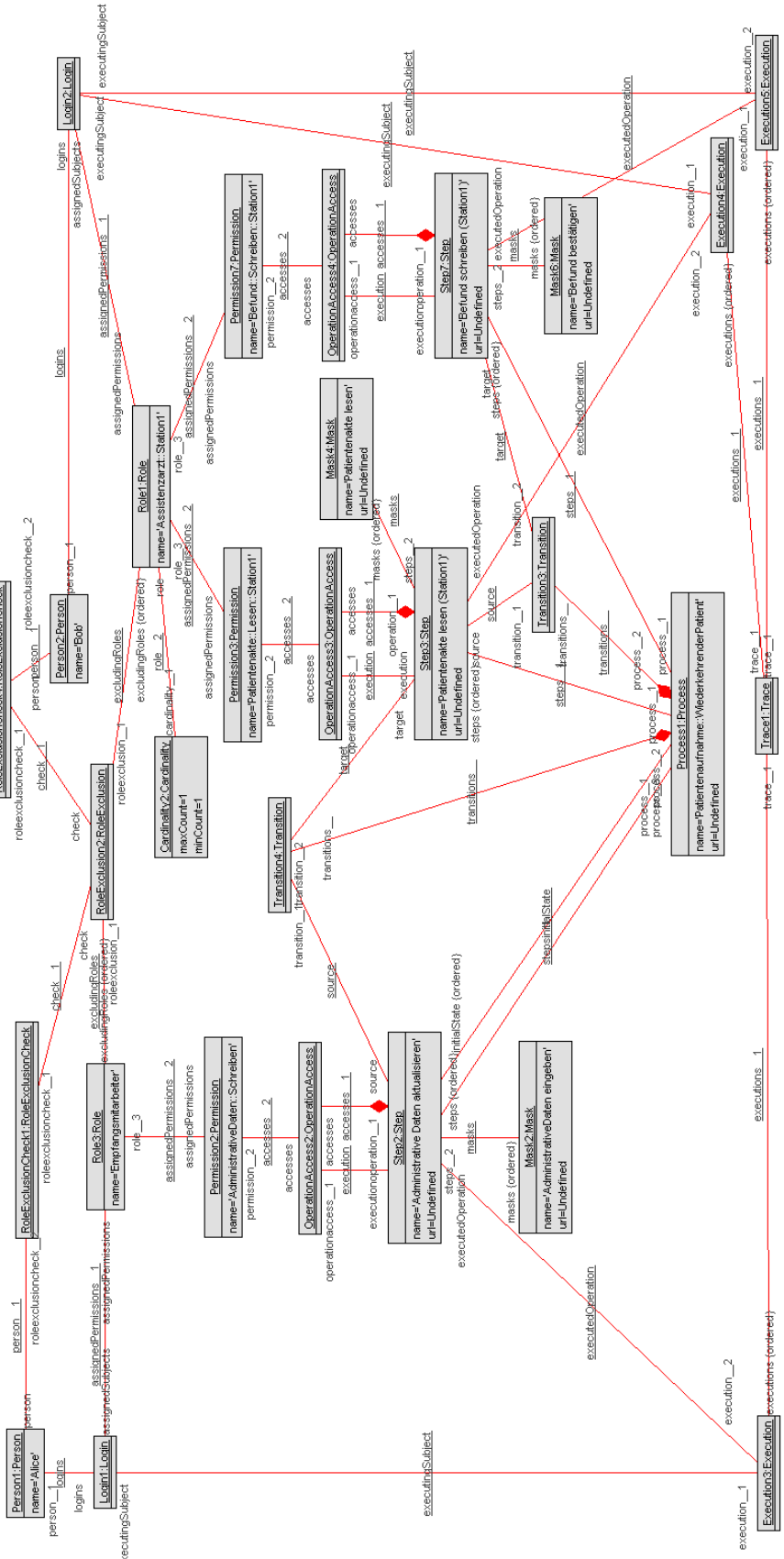


Abbildung 6.1: Krankenhaus-Patientenaufnahme: Objektdiagramm (USE-Darstellung) eines gültigen Systemzustands

6.2 Konsil

Im Krankenhaus kann es vorkommen, dass ein Arzt einer anderen Fachrichtung zur Beratung über einen Patienten hinzugezogen werden muss – Konsil genannt [12, S. 54, 100]. Ist der hinzugezogene Arzt einer anderen Station zugeordnet als der Patient, hat der Arzt in dieser Fallstudie standardmäßig auch eine keine Berechtigung die Patientenakte einzusehen oder einen Befund zu schreiben. Im Falle einer Konsiliaranforderung müssen im diese Berechtigungen aber *temporär* erteilt werden. Dies kann über eine Delegation geschehen, wie es das Objektdiagramm in Abbildung 6.2 zeigt, das einen gültigen Systemzustand repräsentiert: Bob ist Assistenzarzt auf Station 1 (`Role1`), auf der auch der Patient liegt. Außerdem besitzt Bob die Berechtigung die Assistenzarzt-Rolle zu delegieren (`Role2/PrerequisiteRoleDelegationGrantor1`). Übersichts- und einfachheitshalber wird direkt die ganze Rolle delegiert und es nicht weiter in Teilberechtigungen aufgeschlüsselt. Frank ist derjenige, der zur Beratung hinzugezogen werden soll. Er besitzt auch die benötigte Berechtigung, um überhaupt die delegierte Assistenzarzt-Rolle für Station 1 entgegennehmen zu können (`Role3/PrerequisiteRoleDelegationDelegate1`). Die Delegation (`Delegation1`) ist zeitlich begrenzt (`Duration1`), sodass sie automatisch unwirksam wird (ungültiger Systemzustand), sollte sie nicht vorab bereits widerrufen worden sein (z.B. automatisch durch das Krankenhausinformationssystem nach Abschluss des Konsils). Betrachtet man die Maßeinheit der `Timestamp`-Instanzen als Datum, wäre die Delegation am nächsten Tag (Systemzeit = 2) nicht mehr gültig. Frank kann die Delegation auch nicht weiterdelegieren, da die maximale Länge der Delegationskette bereits erreicht ist (`maxLevel=1`) – unabhängig davon, besitzt er auch nicht die nötige Rolle, um dies zu tun.
Durch die Delegation ist es Frank mit seinem Login nun möglich Patientenakten von Station 1 zu lesen (vereinfachte Geschäftsprozess-Darstellung): Um `Step1` auszuführen (`Execution1`) wird die Rolle `Role1` benötigt, die Frank über die Delegation erhalten hat.

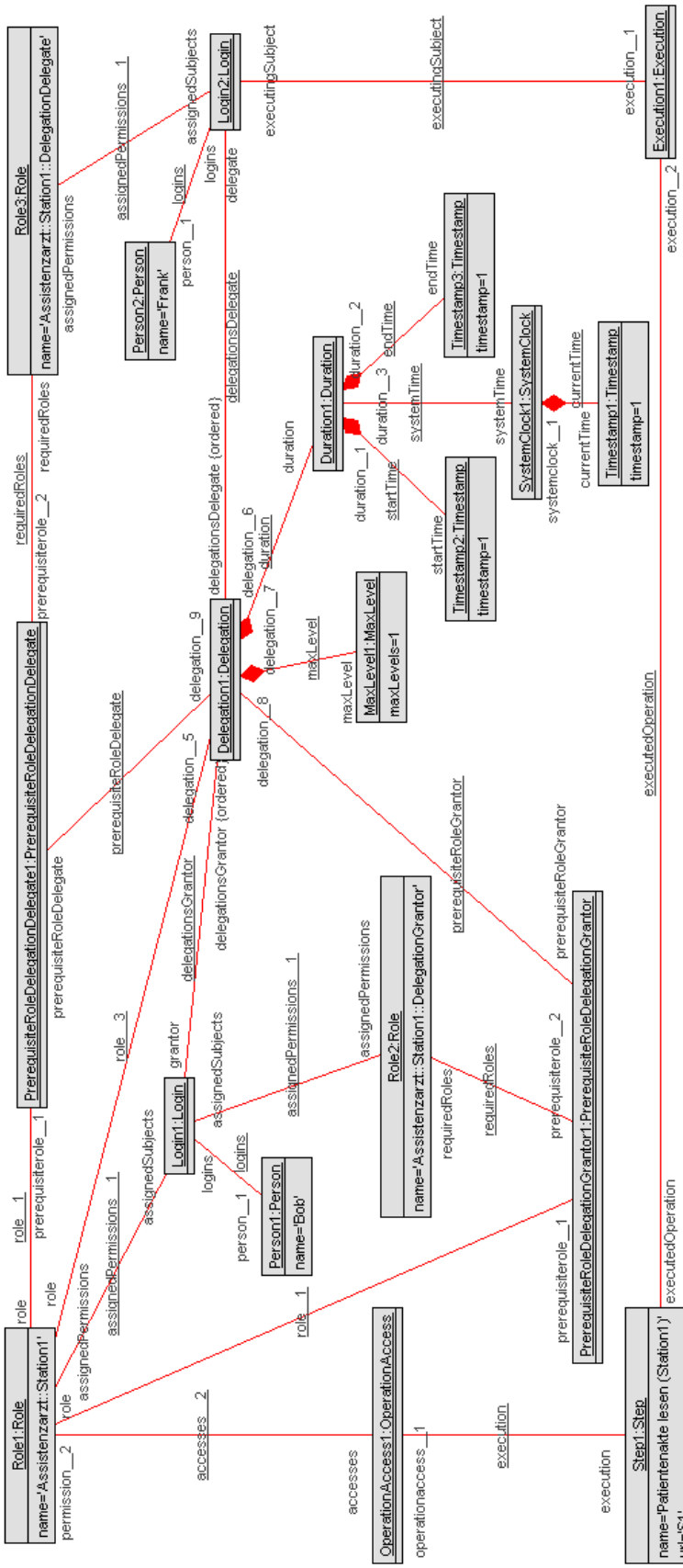


Abbildung 6.2: Krankenhaus-Konsil: Objektdiagramm (USE-Darstellung) eines gültigen Systemzustands

7 Modellvalidierung

In diesem Kapitel fließen das Wissen und die Ergebnisse aller vorherigen Kapitel zusammen und es wird beschrieben, wie die im vorherigen Kapitel eingeführte Fallstudie bzw. das Modell im Allgemeinen validiert wurde. Dafür werden Methoden und Notationen genutzt, wie sie auch beim klassischen Testen von Software angewendet werden. Dabei ist das Modell das eigentliche Testobjekt und die Fallstudie eine Sammlung von Testdaten.

7.1 Einleitung

Zunächst werden einige grundlegende Konzepte und Begriffe des Testens vorgestellt.

7.1.1 Grundannahme

Eingebettet in ein reales Softwaresystem, wäre das Modell wohl nur eine Komponente von vielen; und zahlreiche sicherheitskritische Aspekte würden vielleicht bereits auf übergeordneten Ebenen behandelt werden. So könnten, nachdem sich der Benutzer beispielsweise über die Eingabe von Benutzername und Passwort authentisiert hat, auf der grafischen Oberfläche nur die Optionen angezeigt werden, zu denen der Benutzer auch berechtigt ist. Dadurch würden erst gar keine unberechtigten Anfragen andere Systemschichten/Komponenten erreichen, in denen sie dann geblockt werden müssten. Dennoch sollte idealerweise jede Komponente für sich gesehen fehlerfrei und sicher sein und auch korrekt mit Testdaten umgehen können, die ansonsten von höheren Systemschichten bereits verhindert werden – entsprechend wird das Modell getestet. Es wird angenommen, dass das Modell eigenständig ist und allein jegliche Form von Testdaten korrekt handhaben muss.

Beim Testen muss stets bedacht werden, dass es ab einer gewissen Komplexität und einem bestimmten Umfang des Testobjekts quasi nicht mehr möglich ist, zu beweisen, dass das Testobjekt in allen Belangen korrekt funktioniert: „Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence [10]“. Entsprechend müssen die Testverfahren und Testfälle mit Bedacht gewählt werden, um möglichst viele reguläre Fälle, Sonderfälle und Negativfälle des Testobjekts in einem angemessenen Kostenrahmen zu testen.

7.1.2 Validierung versus Verifikation

Wie es bereits der Titel der Arbeit sowie der Titel dieses Kapitels andeuten, wird das Modell *validiert* – und nicht *verifiziert*. Die Begriffe „Verifikation“ und „Validierung“ sind im Kontext von Software nach [3, S. 2] und [32, S. 247 f.] wie folgt zu unterscheiden:

- **Verifikation:** Überprüfung, ob das Modell die vorgegebenen funktionalen und nicht-funktionalen Anforderungen (Requirements) erfüllt und somit *das Modell*

korrekt implementiert wurde.

- **Validierung:** Überprüfung, ob das Modell die Nutzerbedürfnisse (Systemanforderungen) im operationellen Betriebsumfeld erfüllt – also *das korrekte Modell erstellt* wurde.

Da es keine vollständige, niedergeschriebene Definition von Anforderungen zum Modell gibt, kann keine formale Verifikation durchgeführt werden. Nur zur rollenbasierten Zugriffskontrolle bestehen einige Anforderungen, die allerdings eher grundlegende Konzepte auf System- und nicht auf Modellebene beschreiben. Über die Fallstudie und Testfälle soll der operationelle Betrieb simuliert werden, was einer Validierung entspricht.

7.1.3 Tester und Testzeitpunkt

Der Tester dieses Modells ist nicht der Entwickler des Modells (ausgenommen des Delegationsanteils im Modell). Daraus resultieren für das Testen Vor- und Nachteile. Die Vorteile sind, dass der Tester dann unvoreingenommen an das Modell herangehen kann, da er aus psychologischer Sicht nicht Fehler im eigenen Modell finden muss. Zudem besteht die Gefahr, wenn der Entwickler und der Tester dieselbe Person sind, dass Design- und Verständnisfehler auch nicht beim Testen gefunden werden – wenn solche falschen Zustände beim Entwickeln als korrekt angesehen werden. Der Nachteil generell ist, dass sich der Tester dann in die Thematik einarbeiten muss. Da allerdings die Einarbeitung in das Modell für die Arbeit sowieso zwingend erforderlich war, kann es in diesem speziellen Fall nicht als Negativpunkt gewertet werden. [33, S. 34 ff.][21, S. 14 f.]

Der große Vorteil beim Validieren des Modells zu so einem frühen Zeitpunkt (noch bevor es in einem Softwareprojekt verwendet wird) ist, dass auch Fehler früh gefunden werden können. Und je früher Fehler im Entwicklungsprozess gefunden werden, desto geringer sind die Kosten, um diese zu beseitigen. Werden Fehler erst später entdeckt, sind unter Umständen bereits in weiteren Entwicklungsphasen Folgefehler hinzugekommen. Im schlimmsten Fall wird ein Fehler vorab gar nicht gefunden, tritt allerdings nach Auslieferung beim Kunden auf, was zu Fehlerkosten (z.B. Kosten wegen Systemausfalls, für die der Softwarehersteller haften muss), indirekten Fehlerkosten (z.B. Verlust von Folgeaufträgen) oder Material-/Personenschaden führen kann. [33, S. 184 ff.]

7.1.4 Verlässlichkeit

Die Verlässlichkeit (engl. dependabilitiy) [32, S. 336 ff.][19, S. 16, 60 ff.] eines Systems beruht auf den folgenden vier Eigenschaften:

- **Verfügbarkeit** (engl. availability): Wahrscheinlichkeit, dass das System lauffähig ist und die geforderten Dienste bereitstellt.

- **Zuverlässigkeit** (engl. reliability): Wahrscheinlichkeit, dass über einen definierten Zeitraum das System den Anforderungen entsprechend funktioniert.
- **Betriebssicherheit** (engl. safety): Wahrscheinlichkeit, dass über einen definierten Zeitraum kein *katastrophaler* Fehler auftritt, der z.B. Menschenleben oder die Umwelt gefährdet.
- **Informationssicherheit** (engl. security): Wahrscheinlichkeit, dass das System zufälligen und beabsichtigten Angriffen von intern und extern widersteht.

Informationssicherheit wurde bereits in Abschnitt 2.1 *Informationssicherheit* eingeführt, weil es einer der zu untersuchenden Hauptaspekte dieser Arbeit ist. Allerdings spielt auch die Zuverlässigkeit sowie besonders die Betriebssicherheit eine wichtige Rolle: Da das Modell allgemein und flexibel einsetzbar sein soll, muss davon ausgegangen werden, dass es auch für kritische Systeme eingesetzt wird, in denen ein Systemausfall zu Personenschäden, Umweltschäden oder massiven wirtschaftlichen Schäden führen kann. Die im vorherigen Kapitel vorgestellte Fallstudie „Krankenhaus“ verdeutlicht die Relevanz der Betriebssicherheit; denn ein Fehler im System kann im schlimmsten Fall katastrophale Folgen in Form des Verlusts von Menschenleben haben, wenn beispielsweise bei einem Notfall nicht auf benötigte Patientendaten zugegriffen werden kann. Zu beachten ist, dass nur, weil ein System zuverlässig ist, daraus nicht folgt, dass es auch betriebssicher ist (z.B. können alle Anforderungen vom System erfüllt werden, die Anforderungen sind aber fehlerhaft und führen somit zu einem katastrophalen Fehler). Und umgekehrt kann ein System betriebssicher sein, aber nicht zuverlässig (z.B. es treten häufig kleine, aber nicht katastrophale Fehler auf).

Rein basierend auf dem Modell bzw. davon abgeleiteten Instanzmodellen können die Zuverlässigkeit und die Betriebssicherheit allerdings nicht vollumfänglich nachgewiesen werden, da beides als Wahrscheinlichkeiten über einen definierten *Zeitraum* angegeben werden. In dieser Arbeit können aber nur einzelne Zeitpunkte in Form von Instanzmodellen validiert werden. Fehler wie ein stetiger, ungerechtfertigter, über die Zeit anwachsender Arbeitsspeicherbedarf, der irgendwann nicht mehr gedeckt werden kann, kann beispielsweise hier nicht getestet werden. Auch wenn bei der Validierung berücksichtigt wurde, dass das Modell für kritische Systeme eingesetzt werden könnte, wird an dieser Stelle ausdrücklich darauf hingewiesen, dass ein späteres Softwaresystem, das auf dem Modell basiert, anhand konkreter, genau für den Einsatzzweck ausgelegter Anforderungen verifiziert, validiert und ggf. zertifiziert (z.B. Luftfahrtbehörde) werden muss.

Die Verfügbarkeit kann an dieser Stelle, rein basierend auf dem Modell, nicht getestet werden, da das Konstrukt in dem Sinne nicht lauffähig ist, da nur einzelne Systemzustände, die einen festen Zeitpunkt repräsentieren, erstellt werden können. Auch aneinander gereihte Systemzustände, die aufeinander folgende Zeitpunkte darstellen, eignen sich nicht zur Analyse der Verfügbarkeit: Würde sich in der

Folge ein Fehler ergeben, betrifft dies eher die Zuverlässigkeit, wenn z.B. nach zwei erfolgreichen Zugriffen auf eine Patientenakte der dritte Zugriff trotz vorhandener Berechtigungen nicht mehr funktioniert. Ein Beispiel für mangelhafte Verfügbarkeit wäre, wenn ein Krankenhausinformationssystem durch wenige parallele Patientenaktenabfragen überlastet wäre. Verfügbarkeit ist neben der genannten allgemeinen Definition auch ein Unterbereich der Informationssicherheit, bei dem es allerdings spezifischer darum geht, inwieweit die Verfügbarkeit bei Angriffen gewährleistet werden kann.

7.1.5 Statische und dynamische Tests

Bei statischen Tests [33, S. 81 ff.] handelt es sich im klassischen Sinn um Tests, bei denen das Testobjekt nicht aktiv ausgeführt wird. Stattdessen findet ein Review des Testobjekts – was in diesem Fall das Modell ist – statt, wobei das Modell und die OCL-Constraints intensiv begutachtet werden und nach Fehlern gesucht wird.

Beim dynamischen Testen [33, S. 109 ff.] wird das Testobjekt aktiv mittels Testdaten zur Ausführung gebracht und anschließend kontrolliert, ob die tatsächlichen Testergebnisse mit den vorab definierten Soll-Ergebnissen übereinstimmen. Im Kontext dieser Arbeit bedeutet „Ausführung“, dass ein konkreter Systemzustand aus dem Modell erzeugt wird, wofür ggf. USE und der Model Validator zum Einsatz kommen. Dabei gibt es mehrere Verfahren, um dynamische Testfälle zu erstellen:

- **Blackbox:** Die Testfälle werden basierend auf den Anforderungen erstellt, ohne den inneren Aufbau des Testobjektes zu kennen bzw. zu berücksichtigen. Das Testobjekt ist wie eine schwarze Box, in die man nicht hineinschauen kann, nur die Schnittstellen nach Außen sind bekannt, um Eingabewerte und Soll-Ergebnisse für die Testfälle zu spezifizieren. Auf das Modell bezogen bedeutet dies, dass das UML-Klassendiagramm bekannt ist, allerdings nicht die OCL-Constraints.

- **Intuitiv:** Die intuitive Testfallerstellung kann zu den Blackbox-Verfahren gezählt werden, weil die Kenntnis über den inneren Aufbau des Testobjektes dafür nicht zwingend zwingend erforderlich ist, allerdings dienen dabei nicht die Anforderungen als Grundlage für die Testfallerstellung, und es wird auch kein systematischer Ansatz verfolgt. Vielmehr geht es bei diesem Testverfahren darum, dass der Testfallersteller seine Intuition und Erfahrung nutzt, um Testfälle zu erstellen.

- **Whitebox:** Im Gegensatz zum Blackbox-Verfahren ist hier beim Erstellen der Testfälle der innere Aufbau bekannt. Dies ermöglicht es, gezielt Testfälle zu erstellen, um alle Bestandteile des Testobjektes auszuführen. Als innerer Aufbau werden hier die OCL-Constraints angesehen.

7.2 Betriebs- und Informationssicherheitsanforderungen

Wie bereits im vorherigen Abschnitt beschrieben, existiert kein Anforderungskatalog, der verifiziert werden könnte. Nichtsdestotrotz muss definiert sein, was vom Modell erwartet wird – was korrekte und was fehlerhafte Systemzustände sind. Andernfalls kann nicht spezifiziert werden, was das erwartete Ergebnis eines Testfalls ist, und entsprechend kann auch nicht beurteilt werden, ob das Modell Fehler enthält.

Eine Möglichkeit wäre, den kompletten Anforderungskatalog im Rahmen dieser Arbeit nachträglich zu erstellen. Dies eigenständig durchzuführen, wäre zum einen zeitaufwendig, aber vor allem hochgradig fehleranfällig: Die Anforderungen müssten aus dem Modell gewonnen werden, und dabei ist nicht bekannt, ob das Modell in all seinen Details korrekt ist, sodass ein Fehler im schlimmsten Fall als korrekter Soll-Zustand aufgefasst und als Anforderung festgehalten wird. „In summary, difficulties with requirements is the key root cause of the safety-related software errors which have persisted until integration and system testing.“ [18, S. 129]

Stattdessen werden Gefahren aus den Bereichen Betriebssicherheit und Informationssicherheit auf Basis der Fallstudie identifiziert. Das Krankenhausinformationssystem aus der Fallstudie, wofür das Modell eine Grundlage sein soll, ist dabei als „sekundäre sicherheitskritische Software“ [32, S. 346 ff.] einzustufen: Ein Ausfall der Software wäre keine direkte Gefahr für die Gesundheit eines Menschen, wie es beispielsweise beim Ausfall von Geräten für lebenserhaltende Maßnahmen wäre, sondern die Gefahr ist indirekt, indem nicht auf wichtige Informationen zugegriffen werden kann (z.B. Informationen über Allergien eines Patienten). Aus ermittelten Gefahren können dann wenige sogenannte „Darf nicht“-Anforderungen abgeleitet werden, die auf einer höheren Ebene beschreiben, was nicht passieren darf. Diese Art der Anforderungen sind allgemeiner verfasst und damit einfacher nachvollziehbar und als korrekt zu bewerten. Zudem sind sie weniger vom Ist-Modell abhängig und damit weniger anfällig für eine versehentliche Übernahme eines Fehlers vom Modell in eine Anforderung. Allerdings ist zu beachten, dass ein richtiges Maß zwischen Schutzanforderungen zur Vermeidung von Gefahren und der eigentlichen Funktionalität gefunden werden muss – die Schutzanforderungen dürfen den regulären Betrieb nicht komplett blockieren und müssen gleichzeitig für Sicherheit sorgen: Ein Webserver wäre sicher gegen Angriffe von außen, wenn er mit keinem Netz verbunden ist, allerdings könnte er dann aber auch keine Dienste mehr nach außen anbieten. [32, S. 357 ff.]

Nachfolgend sind die grundlegendsten erkannten Gefahren bezogen auf die Krankenhausinformationssystem-Fallstudie aufgeführt. Dabei ist zu beachten, dass Gefahren im Bereich der Betriebs- und Informationssicherheit betrachtet werden, die sich ergeben, wenn das Ist-Verhalten des eigentlichen Systems vom Soll-Verhalten abweicht und nicht, wenn das System im Rahmen gültiger Spezifikationen falsch bedient oder konfiguriert wurde. So wird hier beispielsweise nicht die Gefahr berücksichtigt,

dass der Administrator des Krankenhausinformationssystems vergisst, einem neu eingestellten Arzt Berechtigungen zu erteilen, wodurch der Arzt keinen Zugriff auf Patientenakten hat. Da das Testobjekt das Modell ist, werden ebenfalls alle Gefahren, die aufgrund von Hardwarefehlern auftreten könnten (z.B. Ausfall des Krankenhausinformationssystems durch einen Stromausfall) an dieser Stelle nicht berücksichtigt.

- Trotz vorhandener Berechtigungen kann eine Operation nicht ausgeführt werden.
- Operationen können ohne Vorhandensein der benötigen Berechtigungen ausgeführt werden.
- Delegieren von Berechtigungen funktioniert nicht korrekt.

Aus den Gefahren ergeben sich folgende Betriebs- und Informationssicherheitsanforderungen im „Darf nicht“-Format, die den Gefahren entgegen wirken sollen:

- Es darf nicht passieren, dass trotz vorhandener Berechtigungen eine Operation nicht ausgeführt werden kann.
- Es darf nicht passieren, dass Operationen ohne Vorhandensein der benötigen Berechtigungen ausgeführt werden können.
- Es darf nicht passieren, dass das Delegieren von Berechtigungen nicht korrekt funktioniert.

7.3 OCL-Constraints: Policy versus Enforcement

Die im vorherigen Abschnitt beschriebenen Betriebs- und Informationssicherheitsanforderungen sind die allgemeinen, globalen Anforderungen, die immer eingehalten werden müssen, wenn es sich um ein sicheres Modell handeln soll. Für die Einhaltung dieser Anforderungen sollen auf technischer Ebene die OCL-Constraints im Modell sorgen: Wird ein Systemzustand (Objektdiagramm) erstellt, bewerten die OCL-Constraints, ob dieser Systemzustand gültig ist oder ob ein OCL-Constraint verletzt wird, was sich aus technischer Sicht darin äußert, dass das es `false` zurückliefert.
Die im Modell befindlichen OCL-Constraints lassen sich in vier Kategorien einteilen:

- **Security-Policy-Constraints**: Diese OCL-Constraints beziehen sich auf Modellelemente, die es erlauben, eine Sicherheitsrichtlinie (engl. policy) für ein System umzusetzen. Die Sicherheitsrichtlinie ist immer vom aktuellen Anwendungsfall abhängig, das Modell bietet nur die Möglichkeiten ihrer Realisierung im Objektdiagramm. So könnte beispielsweise in der Krankenhausfallstudie über eine Instanz der Klasse `PrerequisiteRoleCheck` definiert werden, dass ein Subjekt, das die Rolle „Chefarzt“ hat, auch die Rolle „Arzt“ besitzen muss.

Dabei ist zu beachten, dass nur, weil das Modell einige Möglichkeiten zur Umsetzung einer Sicherheitsrichtlinie bietet, diese nicht alle zwingend genutzt werden müssen. Soll laut Sicherheitsrichtlinie beispielsweise das maximale Vorkommen einer bestimmte Rolle (wie „Chefarzt“) *nicht* beschränkt werden, kann die Nutzung der dafür vorgesehene `Cardinality`-Klasse einfach entfallen. Dadurch fällt die Auswertung des dazugehörigen OCL-Constraints weg, womit es auch nicht `false` zurückliefern und damit den Systemzustand nicht als ungültig bewerten kann.

- **Security-Enforcement-Constraints**: Verglichen mit den Security-Policy-Constraints sind die Security-Enforcement-Constraints die auf der Implementierungsebene operierende Low-Level-Grundlage, um die Sicherheitsrichtlinie überhaupt durchsetzen (engl. enforce) zu können. Security-Enforcement-Constraints sind zwingend erforderlich, aber nicht direkt Teil der Sicherheitsrichtlinie. Beispielsweise gibt die Sicherheitsrichtlinie vor, welche Rollen welche Operationen ausführen dürfen. Die eigentliche Überprüfung direkt vor der Ausführung einer Operation, ob das Subjekt überhaupt über die notwendigen Berechtigungen verfügt, ist das Security-Enforcement-Constraint.

- **Non-Security-Policy-Constraints**: Wie auch die Security-Policy-Constraints beschreiben die Non-Security-Policy-Constraints Vorgaben, allerdings beziehen sich diese explizit nicht auf die Sicherheit, sondern auf den allgemeinen Workflow. Dies kann z.B. die Reihenfolge sein, in der bestimmte Operationen ausgeführt werden müssen.

- **Non-Security-Enforcement-Constraints**: Dies ist die Low-Level-Variante der Non-Security-Policy-Constraints, um elementare Grundlagen beim Workflow sicherzustellen.

Diese Kategorisierung der OCL-Constraints wird für die nachfolgende Modellvalidierung benötigt und wurde im Rahmen der Arbeit in das gesamte Modell (inkl. Delegationsanteil) in Form von selbst definierten Annotationen eingepflegt (siehe Tabelle 2). Der Vorteil von solchen Annotationen ist, dass sie das eigentliche Modellverhalten unverändert lassen, da sie nur für die Leser relevant sind, die sich gezielt dafür interessieren. Besitzt eine Klasse im Modell mindestens ein Security-Policy-Constraint, so enthält die Klasse eine Annotation namens `Security-Policy-Constraint` mit dem Key `constraints` und als Value die Leerzeichen-separierten Namen der entsprechenden OCL-Constraints. Gleiches gilt für Non-Security-Policy-Constraints mit dem Unterschied, dass die Annotation entsprechend `Non-Security-Policy-Constraints` heißt. Alle anderen OCL-Constraints im Modell, die nicht explizit als Security-Policy-Constraints oder Non-Security-Policy-Constraints vermerkt sind, gelten implizit als Enforcement-Constraints. Eine Unterscheidung zwischen Security-Enforcement-Constraint und Non-Security-Enforcement-Constraint wird für die Modellvalidierung nicht benötigt.

Tabelle 2: Kategorisierung der OCL-Constraints. OCL-Constraints im Rahmen von Operationen sind nicht aufgezählt, da sie in USE auch nicht (de-)aktiviert werden können.

OCL-Constraint	**Constraint-Kategorie**
Execution::subjectHasPermissions	Security-Enforcement
SystemClock::isSingleton	Non-Security-Enforcement
Role::childCannotBeDirectOwnParent	Security-Enforcement
Role::childCannotBeTransitiveOwnParent	Security-Enforcement
RoleExclusionCheck::justOnePerson	Security-Enforcement
RoleExclusionCheck::justOneExclusion	Security-Enforcement
RoleExclusionCheck::check	Security-Policy
PrerequisiteRole::roleCannotNotRequireItself	Security-Enforcement
PrerequisiteRoleCheck::prerequisiteRole	Security-Policy
Cardinality::minCardinalityCheck	Security-Policy
Cardinality::maxCardinalityCheck	Security-Policy
Cardinality::minSmallerThanMax	Security-Enforcement
Process::processConformance	Non-Security-Enforcement
Delegation::maxLevelCheck	Security-Policy
Delegation::matchingPrerequisiteRoleGrantorCheck	Security-Enforcement
Delegation::prerequisiteRoleGrantorCheck	Security-Policy
Delegation::matchingPrerequisiteRoleDelegateCheck	Security-Enforcement
Delegation::prerequisiteRoleDelegateCheck	Security-Policy
Delegation::differentSubjects	Security-Enforcement
Delegation::sameSubjectType	Security-Enforcement
Delegation::noLoginsForPersons	Security-Enforcement
Delegation::differentPersonWithLoginSubject	Security-Enforcement
Delegation::noPrerequisiteRoleDelegationGrantorDuplicates	Security-Enforcement
Delegation::noPrerequisiteRoleDelegationDelegateDuplicate	Security-Enforcement
MultipleDelegationCheck::maxMultipleDelegation	Security-Policy

7.4 Vorgehen bei der Modellvalidierung

In diesem Abschnitt wird das angewandte, technische Vorgehen zur Modellvalidierung beschrieben, bevor dann im darauf folgenden Abschnitt die konkreten Testfälle und Ergebnisse präsentiert werden.

7.4.1 Statische Modellvalidierung

Jegliche Art von Modellvalidierung/Testen, die nicht unter Verwendung des Model Validators geschieht, wird als statisch angesehen, wobei eine Mischung aus direktem Review des Modells und der OCL-Constraints und Blackbox-, Intuitiv- und Whitebox-Tests in Form von kleinen Objektdiagrammen zum Einsatz kommen, wobei je nach Test überprüft wird, ob die OCL-Constraints eingehalten werden oder ob dagegen verstoßen wird, wobei bei einem Negativtest auch eine Verletzung der OCL-Constraints das erwartete Ergebnis sein kann.

Die statische Modellvalidierung muss vor der dynamischen geschehen und ist auch nicht durch die dynamische abgedeckt oder ersetzbar, denn die Modellvalidierung mittels USE und dem Model Validator – was hier als dynamisches Testen angesehen wird – liegt die Annahme zu Grunde, dass das Modell und die OCL-Constraints an sich korrekt sind. Wenn der Model Validator keine Policy-Verletztungen findet, kann daraus ansonsten (ohne vorherige statische Modellvalidierung) nicht geschlussfolgert werden, dass das Modell korrekt ist, denn die Security-Policy-Constraints könnten selbst fehlerhaft sein. Beispielsweise könnten alle Security-Policy-Constraints ohne tatsächliche Überprüfung einfach `true` zurückliefern. Um so etwas auszuschließen, wurde das Modell und die OCL-Constraints detailliert begutachtet und getestet, ob sie auch tatsächlich die korrekte Funktionsweise realisieren. Auch die Integration des Delegationskonzeptes kann als ein statischer Test angesehen werden; denn dabei wurde sich ebenfalls intensiv mit dem bereits bestehenden Modell auseinander gesetzt.

7.4.2 Dynamische Modellvalidierung mittels USE Model Validator

Das Ziel ist es, mit USE und dem dazugehörigen Model Validator, dessen grundlegende Funktionsweisen in Kapitel 3 *USE-Tool (UML-based Specification Environment)* vorgestellt wurden, zu ermitteln, ob es ausgehend von einem korrekten Systemzustand möglich ist, durch die reguläre Nutzung einen Systemzustand zu erreichen, der die Sicherheitsrichtlinie (Security-Policy-Constraints) verletzt. Ist dies der Fall, enthält das Modell ungewollte Seiteneffekte, die zu Sicherheitslücken führen.
Das Szenario lässt sich am einfachsten verdeutlichen, wenn man sich das Modell als Grundlage für ein Softwareprogramm vorstellt: Bevor der eigentliche Endnutzer die Anwendung bedient, werden von administrativer Seite aus – entsprechend der Sicherheitsrichtlinie – die Berechtigungen, Rollen, Personen und Logins definiert worden sein. Zudem gibt die Anwendung vor, welche Operationen überhaupt möglich sind. Es wird vorausgesetzt, dass zu diesem Zeitpunkt kein OCL-Constraint des

Modells verletzt wird und somit von einem gültigen, initialen Systemzustand ausgegangen werden kann. Die administrativ- und softwarebedingten Vorgaben werden während der Validierung als konstant angenommen, denn der normale Endnutzer hat darauf keinen Einfluss – er kann beispielsweise nicht einfach neue Rollen anlegen oder Softwarefunktionen erweitern. Für die Validierung eines solchen Szenarios mit dem Model Validator bedeutet dies, dass auch der Model Validator auf einem grundlegenden Objektdiagramm aufbauen muss, welches die administrativ- und softwarebedingten Vorgaben darstellt. Damit der Model Validator diese Vorgabe als gegeben und nicht erweiterbar ansieht, muss die Anzahl der vorgegebenen Objekte und Links im Objektdiagramm pro Klasse und Assoziation als minimale und maximale Anzahl eingetragen werden. Mit steigender Anzahl verwendeter Klassen/Objekte und Assoziationen/Links im Objektdiagramm wird das Setzen der Minima und Maxima in der Konfigurationsdatei (`*.properties`) immer aufwendiger und fehleranfälliger.

Da ein Fehler in der Konfigurationsdatei das spätere Validierungsergebnis verfälschen könnte, wurde ein Tool entwickelt, welches bei der Erstellung der Konfigurationsdatei entsprechend dem Objektdiagramm unterstützt. Das Tool ist als Tab „Properties Files Generation“ in Ecore2USE eingebunden, wie Abbildung 7.1 zeigt. Eingabe ist das Objektdiagramm in Form einer USE-SOIL-Datei, die die relevanten USE-Befehle `!new`/`!create` und `!insert` zur Erzeugung der Objektdiagrammelemente enthält. Diese Datei wird vom Tool eingelesen, die USE-Befehle über reguläre Ausdrücke geparst und zwei Maps erstellt, deren Keys Klassennamen bzw. Assoziationsnamen sind und deren Values Zähler für die Anzahl der dazugehörigen konkreten Objekte bzw. Links sind. Während des Erstellens der Maps werden auch zwei Sets verwaltet, welche alle Objektnamen bzw. Links in der Form `<association name>:<link endpoint 1>|<link endpoint 2>` enthalten. Darüber lassen sich Duplikate ermitteln. Denn USE würde solche Duplikate bei der Abarbeitung der SOIL-Datei nicht anlegen, die Abarbeitung aber auch nicht unterbrechen – entsprechend darf das Tool auch den Zähler dafür nicht inkrementieren. Das Tool setzt allerdings voraus, dass das Objektdiagramm konform zum Modell/Klassendiagramm ist, da ein Abgleich diesbezüglich nicht weiter stattfindet. Die Properties-Datei, die das Tool als Ergebnis liefert, ist allerdings nicht vollständig: In der Properties-Datei müssen alle Klassen und Assoziationen vorhanden sein. Die generierte Properties-Datei enthält allerdings nur die, die auch im Objektdiagramm verwendet wurden. Die grafische Benutzeroberfläche „Model Validator Configuration“ des Model Validators für die Bearbeitung der Properties-Datei erlaubt es aber, solche unvollständigen Properties-Datei zu öffnen und alle fehlenden Klassen, Assoziationen und sonstigen Einstellungen automatisch mit Standardwerten zu *ergänzen*. Die Standardwerte müssen dann für die eigentliche Modellvalidierung, bei der die Properties-Datei die Eingabe ist, manuell auf sinnvolle Werte abgeändert werden.

Zur Modellvalidierung wird eine modell- und ergebnisabhängige Abfolge von USE- und Model-Validator-Kommandozeilenbefehle verwendet. Dabei wird vorausgesetzt,

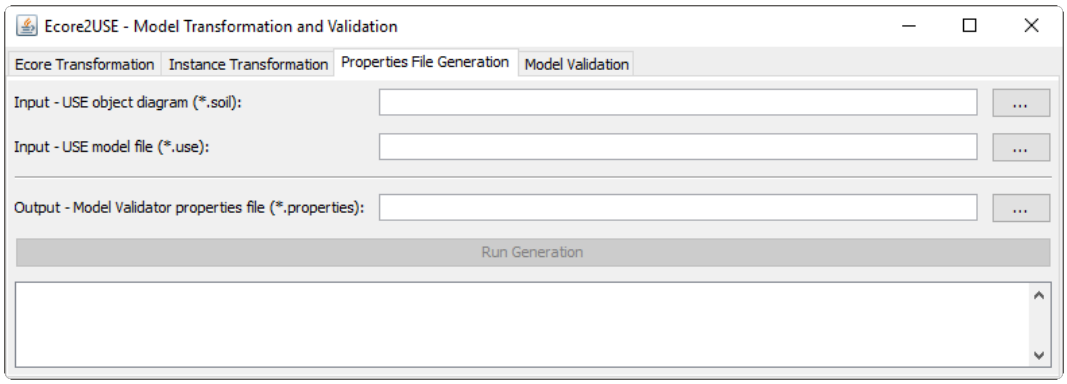


Abbildung 7.1: Ecore2USE-GUI zur Generierung einer Konfigurationsdatei (`*.properties`) des Model Validators

dass zum Zeitpunkt des Ausführens des ersten Befehls lediglich USE gestartet wurde, aber ansonsten keine Befehle oder Einstellungen vorgenommen wurden, die ggf. die Ergebnisse der nachfolgenden Befehle beeinflussen:

1. `open <USE-MODEL>` öffnet das USE-Modell (`*.use`), das validiert werden soll. Das USE-Modell wurde vorab mittels Ecore2USE aus dem Ecore-Modell gewonnen. Für die weiteren Schritte ist es notwendig, dass das USE-Modell wohlgeformt ist und USE beim Öffnen keine Fehler meldet.

2. `open <PARTIAL-OBJECTDIAGRAM>` öffnet optional eine `*.soil`-Datei, die Befehle enthält, um ein Teilobjektdiagramm basierend auf dem vorab geladenen USE-Modell zu erzeugen. Die erzeugten Objekte und Links sind dann die fest gesetzte Eingangsgröße für die weitere Modellvalidierung. Somit ist es möglich, eine Grundlage zu schaffen, auf die der Model Validator aufbaut.

3. `constraints -flags <CONSTRAINT-NAME> +d` deaktiviert das angegebene OCL-Constraint, sodass es in nachfolgenden Schritten nicht mehr berücksichtigt wird. Dieser Befehl wird für alle OCL-Constraints ausgeführt, die zu der Kategorie Security-Policy-Constraint oder Non-Security-Policy-Constraint gehören.

4. `mv -config objExtraction:=on` sorgt dafür, dass im nachfolgenden Schritt das in Schritt 2 geöffnete Teilobjektdiagramm die Grundlage bildet. Ohne diesen Befehl würde das vorhandene Teilobjektdiagramm vom Model Validator ignoriert und die Generierung von Systemzuständen stets von einem „leeren“ Objektdiagramm aus gestartet werden. Schritt 2 ist optional, sodass nicht zwingend ein Objektdiagramm als Basis geladen werden muss; in diesem Fall ist der Befehl wirkungslos.

5. `mv -scrollingAll <PROPERTIES-FILE>` unter Einsatz des Model Validators werden alle Systemzustände generiert, die die in der `*.properties`-Datei spezifizierten Rahmenbedingungen (u.a. min./max. Anzahl Objekte je Klasse

und min./max. Anzahl Links je Assoziation) einhalten. Für jedes Element der Lösungsmenge gilt, dass alle nicht deaktivierten OCL-Constraints eingehalten sind. Die Angaben in der `*.properties`-Datei müssen weise gewählt werden, da ansonsten der mögliche Lösungsraum so groß ist, dass die Bestimmung der Lösungsmenge nicht in einer realistischen Dauer möglich ist. Als Abschluss liefert der Befehl die Anzahl der Elemente in der Lösungsmenge. Wenn die Lösungsmenge leer ist, müssen die nachfolgenden Schritte nicht mehr ausgeführt werden.

6. `constraints -flags <POLICY-CONSTRAINT-NAME> -d` aktiviert das angegebene OCL-Constraint, sodass dieses für alle nachfolgenden Schritte berücksichtigt wird. Dieser Schritt wird für alle Constraints ausgeführt, die in Schritt 3 deaktiviert wurden. Abschließend sind somit wieder alle im Modell befindlichen OCL-Constraints aktiviert.

7. `mv -scrollingAll show(<i>)` veranlasst den Model Validator, das `i`-te Objektdiagramm der in Schritt 5 bestimmten Lösungsmenge in USE zu laden. Initial begonnen wird mit dem ersten Element der Lösungsmenge: `i` = 1.

8. `check` wertet alle aktivierten OCL-Constraints aus, was nach Schritt 6 alle im Modell befindlichen OCL-Constraints sind, für das im vorherigen Schritt erzeugte Objektdiagramm aus. Einzeln nach OCL-Constraint aufgeschlüsselt, ist das Ergebnis entweder `OK` oder `FAILED`.

9. Wenn im vorherigen Schritt mindestens ein OCL-Contraint verletzt wurde, wird das dazugehörige Objektdiagramm mittels `write <PATH>/invalid/<i>.soil` als Datei gespeichert. Damit auch die generierten Objektdiagramme begutachtet werden können, in denen keine OCL-Contraint-Verletzung detektiert wurde, werden diese ebenfalls gespeichert: `write <PATH>/valid/<i>.soil`. Durch das spätere manuelle Einsehen der gültigen Objektdiagramme kann weiter nach Fehlern im Modell gesucht werden.

10. Solange nicht alle Elemente der Lösungsmenge abgearbeitet worden sind, wird Schritt 7 mit dem nächsten Element in der Lösungsmenge wiederholt.

Das Ergebnis nach Ausführung der möglichen Schritte ist eines der folgenden:

- Mit den Spezifikationen in der Konfigurationsdatei (`*.properties`) konnten zusammen mit dem optionalen vorab geladenen Teilobjektdiagrammen keine Lösungen gefunden werden, sodass die Befehlsabfolge nach Schritt 5 beendet werden musste. Der Grund dafür ist entweder eine sich mit dem Teilobjektdiagramm widersprechende Konfigurationsdatei, eine Konfigurationsdatei, die dem Soll-Verhalten des Modells widerspricht, oder wenn dies nach manueller Überprüfung ausgeschlossen ist, ein grundlegender Designfehler im Modell.

- Alle in Schritt 5 gefundenen Lösungen halten auch die nachträglich wieder aktivierten Policy-Constraints ein, sodass das Modell für die durch die Kon-

figurationsdatei und das Teilobjektdiagramm erzeugten Systemzustände als korrekt und damit sicher angenommen werden kann.

- Mindestens bei einer der in Schritt 5 gefundenen Lösungen wurde mindestens eines der nachträglich wieder aktivierten Policy-Constraints verletzt. Dies kann an einem bereits fehlerhaften Teilobjektdiagramm liegen oder durch eine falsche Spezifikationen in der Konfigurationsdatei verursacht worden sein, die beispielsweise das automatische Generieren von Objekten von Klassen ermöglicht, die eigentlich fix sein sollten. Wenn diese beiden Gründe allerdings durch manuelle Kontrolle ausgeschlossen werden können, liegt ein Fehler, und aufgrund der in Schritt 3 gewählten Policy-Constraints, eine Verletzung der Sicherheitsrichtlinie vor – das Modell weist also eine Sicherheitslücke auf. Zur genauen Fehleranalyse müssen die in Schritt 9 abgespeicherten Objektdiagramme manuell weiter untersucht werden.

Da die exakten USE- und Model-Validator-Befehle vor jedem Schritt bestimmbar sind, kann die Modellvalidierung softwaregestützt weiter automatisiert werden. Dafür wurde die in Abbildung 7.2 gezeigte grafische Benutzeroberfläche entwickelt. Neben den bereits explizit in den Schritten 1, 2 und 5 genannten Dateien sind die weiteren Eingangsparameter der Pfad zur Batch-Datei, die das USE-Tool startet, eine Datei, die alle OCL-Contraints enthält, die in Schritt 3 deaktiviert und in Schritt 6 wieder aktiviert werden, sowie das Verzeichnis, welches in Schritt 9 verwendet wird. Die Datei mit den OCL-Constraints wird automatisch während der Ecore-zu-USE-Transformation in dem Verzeichnis, in dem auch das erzeugte USE-Modell abgelegt wird, gespeichert. Inhaltlich listet die Datei zeilenweise die OCL-Constraints, die über die Annotation `Security-Policy-Constraint` und `Non-Security-Policy-Constraints` im Modell markiert sind, in der Form `<CLASS-NAME>::<OCL-CONSTRAINT-NAME>` auf.
Wird nach Eingabe der Eingangsparameter die Modellvalidierung ausgeführt, wird im Hintergrund das USE-Tool gestartet und der Standardeingabe- (stdin) sowie der Standardausgabe- (stdout) und Standardfehlerausgabe-Datenstrom abgegriffen. Damit kann die USE-Kommandozeile über die Software so bedient werden, wie sie auch manuell bedient wird. Über die Standardeingabe werden die Befehle der Schritte 1-10 softwaregestützt an das USE-Tool übergeben und anschließend über die Standardausgabe/Standardfehlerausgabe die von USE gelieferten Ergebnisse zeilenweise mittels regulärer Ausdrücke geparst sowie in unveränderter Form dem Benutzer angezeigt. Über den regulären Ausdruck `Found (\d+) solutions` wird beispielsweise die Anzahl der in Schritt 5 gefunden Lösungen aus den von USE und dem Model Validator gelieferten Ausgaben ermittelt.
Der große Vorteil der Automatisierung ist die Zeitersparnis: Es muss nicht die ganze Befehlsabfolge manuell eingegeben werden, was vor allem bei einer großen Anzahl von zu (de-)aktivierenden OCL-Constraints oder gefundenen Lösungen aufwendig und zudem fehleranfällig ist.

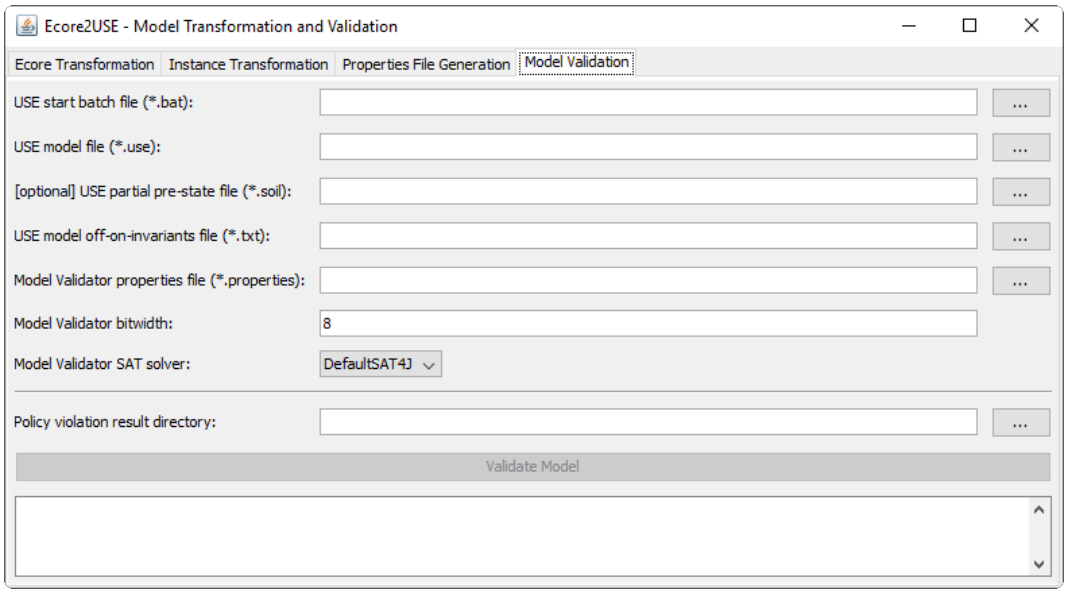


Abbildung 7.2: Ecore2USE-GUI für Modellvalidierung

7.5 Ergebnisse

Die Grundlage für die Ergebnisse sind das RBAC-Grundlagenmodell (siehe Abschnitt 4.2 *Grundlagenmodell*) inklusive der manuellen Modellmodifikationen, um ein lauffähiges Modell für die dynamischen Tests zu haben (Abschnitt 4.3 *Manuelle Modellmodifikation*), und inklusive dem Delegationskonzept (Abschnitt 4.4 *Delegation*).

7.5.1 Testausrüstung

Die Ergebnisse wurden unter Verwendung von USE 4.2.0 und dem Model Validator 4.2-r1 auf dem folgenden System erzielt:

- Betriebssystem: Windows 10, 64 Bit
- Java: SE 8u144, 64 Bit
- IDE: Eclipse Modeling Tools, Neon.1 Release RC1 (4.6.1RC2)
- Prozessor: Intel Core i7-6700 (4 Kerne), 4 GHz
- Arbeitsspeicher: G.Skill DDR4-3200, 16 GB
- Speichermedium: Samsung SSD 950 Pro, 512 GB

7.5.2 Ergebnisse statischer Tests

Die nachfolgend genannten Modelleigenschaften beschreiben zum einen Auffälligkeiten, die während der Arbeit mit dem Modell bzw. der statischen Validierung des Modells gefunden wurden, die aber nicht zwingend als Fehler angesehen werden

müssen, und zum anderen tatsächliche Fehler. Erstere resultieren vor allem aus der Domänenunabhängigkeit und Flexibilität des Modells; die Einschätzung der Kritikalität liegt dann zum Teil beim Modellierer, der das Modell einsetzt. Entsprechend werden nachfolgend auch Modelleigenschaften beschrieben, die einige Modellanwender vielleicht für ihr Anwendungsszenario als unproblematisches oder sogar gewolltes Verhalten ansehen.

Negativtests Das Ziel des Negativtests ist es, über ein Objektdiagramm zu zeigen, dass die im Grundlagenmodell bzw. Delegationskonzept vorhandenen OCL-Constraints grundlegend korrekt arbeiten und nicht lediglich z.B. ohne Überprüfung `true` zurückliefern. Das Gegenbeispiel/Positivbeispiel – dass nicht alle OCL-Constraints einfach `false` zurückliefern – wurde bereits implizit mit den Objektdiagrammen in Kapitel 6 *Fallstudie Krankenhaus* gegeben.

Zuerst wird der Negativtest des **Grundlagenmodells** besprochen: Im folgenden Listing 7.1 ist zu sehen, dass USE die Struktur (Multiplizitäten) für das Objektdiagramm in Abbildung 7.3 als korrekt bewertet und dies somit nicht der Grund für die Verletzung der OCL-Constraints (`FAILED`) ist.

Listing 7.1: USE: Überprüfung des Negativobjektdiagramms zum Grundlagenmodell

```
use> check
checking structure...
checked structure in 1ms.
checking invariants...
checking invariant (1) 'Cardinality::maxcardinalitycheck': FAILED.
-> false : Boolean
checking invariant (2) 'Cardinality::mincardinalitycheck': FAILED.
-> false : Boolean
checking invariant (3) 'Cardinality::minsmallerthanmax': FAILED.
-> false : Boolean
checking invariant (4) 'Delegation::differentpersonwithloginsubject': OK.
checking invariant (5) 'Delegation::differentsubjects': OK.
checking invariant (6) 'Delegation::matchingprerequisiteroledelegatecheck': OK
    .
checking invariant (7) 'Delegation::matchingprerequisiterolegrantorcheck': OK.
checking invariant (8) 'Delegation::maxlevelcheck': OK.
checking invariant (9) 'Delegation::nologinsforpersons': OK.
checking invariant (10) 'Delegation::
    noprerequisiteroledelegationdelegateduplicate': OK.
checking invariant (11) 'Delegation::
    noprerequisiteroledelegationgrantorduplicates': OK.
checking invariant (12) 'Delegation::prerequisiteroledelegatecheck': OK.
checking invariant (13) 'Delegation::prerequisiterolegrantorcheck': OK.
checking invariant (14) 'Delegation::samesubjecttype': OK.
checking invariant (15) 'Duration::notexpired': OK.
checking invariant (16) 'Execution::subjecthaspermissions': FAILED.
-> false : Boolean
checking invariant (17) 'MultipleDelegationCheck::maxmultipledelegation': OK.
checking invariant (18) 'PrerequisiteRole::rolecannotnotrequireitself': OK.
checking invariant (19) 'PrerequisiteRoleCheck::prerequisiterole': FAILED.
-> false : Boolean
checking invariant (20) 'Process::processconformance': FAILED.
-> false : Boolean
```

```
31 checking invariant (21) 'Role::childcannotbedirectownparent': FAILED.
32 -> false : Boolean
33 checking invariant (22) 'Role::childcannotbetransitiveownparent': FAILED.
34 -> false : Boolean
35 checking invariant (23) 'RoleExclusionCheck::check': FAILED.
36 -> false : Boolean
37 checking invariant (24) 'RoleExclusionCheck::justoneexclusion': OK.
38 checking invariant (25) 'RoleExclusionCheck::justoneperson': OK.
39 checking invariant (26) 'SystemClock::issingleton': OK.
40 checked 26 invariants in 0.029s, 9 failures.
```

Bezogen auf die Nummern in obigem Listing werden die OCL-Constraints aus folgenden Gründen verletzt:

(1)/(2) `Role2` verletzt sowohl die untere Grenze (0) als auch die obere Grenze (0) von `Cardinality2`, da diese Rolle einem Subjekt zugewiesen ist.

(3) Bei `Cardinality2` ist die obere Grenze kleiner als die untere Grenze ($2 > 1$).

(16) `Person1` versucht `Step1` auszuführen (`Execution1`), verfügt allerdings nicht über die Berechtigung `Permission1`, die dafür erforderlich ist.

(19) Wenn `Person1` `Role1` besitzt, muss sie laut `PrerequisiteRoleCheck1` auch `Role6` besitzen. Das ist aber nicht der Fall.

(20) Der Geschäftsprozessablauf ist für `Process1` nicht korrekt modelliert, da der dazugehörige `Step1` zwei Eingabemasken besitzt, die nacheinander abgearbeitet werden. Somit findet ein Übergang zwischen `Mask1` und `Mask2` statt, der über eine `Transition`-Instanz modelliert und `Process1` zugewiesen wird.

(21)/(22) `Role1` und `Role2` stehen im widersprüchlichen Verhältnis zueinander, da beide die Kind-Rolle von der jeweils anderen Rolle sind.

(23) `Login1` besitzt die Rollen `Role1` und `Role2`. Laut `RoleExclusion1` / `RoleExclusionCheck1` darf `Person1` aber nicht beide Rolle besitzen, da sich diese gegenseitig ausschließen.

Die OCL-Constraints (18), (24) und (25) beziehen sich auf die Modellstruktur und werden durch USE verhindert bzw. als Strukturfehler erkannt, sodass hierfür bei Verwendung mit USE kein OCL-Constraint nötig gewesen wäre.

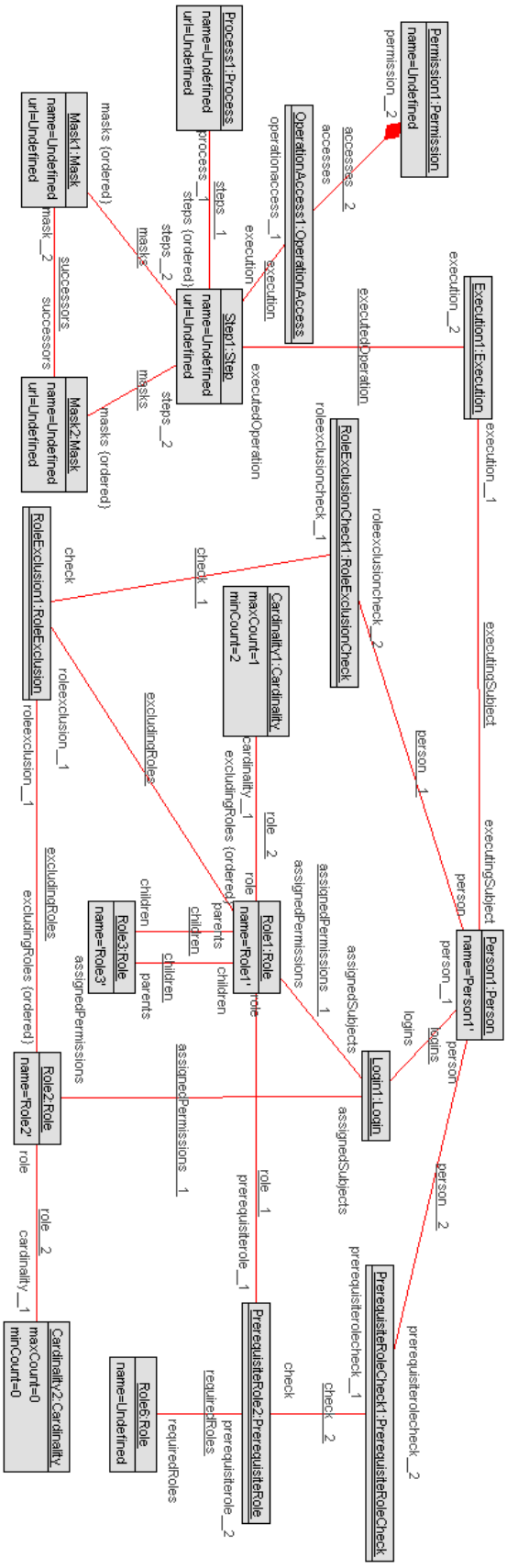


Abbildung 7.3: Negativbeispiel, das den Großteil der OCL-Constraints des Grundlagenmodells verletzt.

Das Objektdiagramm in Abbildung 7.3 ist der Negativtest für das **Delegationskonzept**. Wie das folgende Listing 7.2 zeigt, wurde auch hier von USE kein Fehler bezüglich der Modellstruktur gefunden, sodass für ein `FAILED` ausschließlich das OCL-Constraint selbst verantwortlich ist.

Listing 7.2: USE: Überprüfung des Negativobjektdiagramms zum Delegationskonzept

```
use> check
checking structure...
checked structure in 0ms.
checking invariants...
checking invariant (1) 'Cardinality::maxcardinalitycheck': OK.
checking invariant (2) 'Cardinality::mincardinalitycheck': OK.
checking invariant (3) 'Cardinality::minsmallerthanmax': OK.
checking invariant (4) 'Delegation::differentpersonwithloginsubject': FAILED.
-> false : Boolean
checking invariant (5) 'Delegation::differentsubjects': OK.
checking invariant (6) 'Delegation::matchingprerequisiteroledelegatecheck':
    FAILED.
-> false : Boolean
checking invariant (7) 'Delegation::matchingprerequisiterolegrantorcheck':
    FAILED.
-> false : Boolean
checking invariant (8) 'Delegation::maxlevelcheck': OK.
checking invariant (9) 'Delegation::nologinsforpersons': OK.
checking invariant (10) 'Delegation::
    noprerequisiteroledelegationdelegateduplicate': FAILED.
-> false : Boolean
checking invariant (11) 'Delegation::
    noprerequisiteroledelegationgrantorduplicates': FAILED.
-> false : Boolean
checking invariant (12) 'Delegation::prerequisiteroledelegatecheck': FAILED.
-> false : Boolean
checking invariant (13) 'Delegation::prerequisiterolegrantorcheck': FAILED.
-> false : Boolean
checking invariant (14) 'Delegation::samesubjecttype': OK.
checking invariant (15) 'Duration::notexpired': FAILED.
-> false : Boolean
checking invariant (16) 'Execution::subjecthaspermissions': OK.
checking invariant (17) 'MultipleDelegationCheck::maxmultipledelegation':
    FAILED.
-> false : Boolean
checking invariant (18) 'PrerequisiteRole::rolecannotnotrequireitself': OK.
checking invariant (19) 'PrerequisiteRoleCheck::prerequisiterole': OK.
checking invariant (20) 'Process::processconformance': OK.
checking invariant (21) 'Role::childcannotbedirectownparent': OK.
checking invariant (22) 'Role::childcannotbetransitiveownparent': OK.
checking invariant (23) 'RoleExclusionCheck::check': OK.
checking invariant (24) 'RoleExclusionCheck::justoneexclusion': OK.
checking invariant (25) 'RoleExclusionCheck::justoneperson': OK.
checking invariant (26) 'SystemClock::issingleton': FAILED.
-> false : Boolean
checked 26 invariants in 0.028s, 10 failures.
```

Aus den folgenden Gründen wurden die mit Nummern versehenen OCL-Constraints verletzt:

(4) Der Login des Delegierenden als auch der des Delegationsempfängers gehören derselben Person. Sich selbst Berechtigungen zu delegieren ist per OCL-

Constraint verboten.

(6)/(7) `PrerequisiteRoleDelegationDelegate1` und `PrerequisiteRoleDelegationGrantor1` sollen beschreiben welche Rollen der Delegationsempfänger bzw. der Delegierende besitzen muss, um die zu delegierende Rolle zu empfangen bzw. zu delegieren. Die OCL-Constraints haben detektiert, dass die Rolle, auf die sich `PrerequisiteRoleDelegationDelegate1` und `PrerequisiteRoleDelegationGrantor1` beziehen, nicht mit der zu delegierenden Rolle übereinstimmt (`Role7` statt `Role1`).

(10)/(11) Sowohl `PrerequisiteRoleDelegationGrantor1` und `PrerequisiteRoleDelegationGrantor2` bzw. `PrerequisiteRoleDelegationDelegate1` als auch `PrerequisiteRoleDelegationDelegate2` beschreiben, welche Rollen benötigt werden, um `Role7` delegieren bzw. empfangen zu können. Eine solche Form von Duplikaten ist nicht erlaubt, da sie sich widersprechen können, was hier auch der Fall ist: `PrerequisiteRoleDelegationGrantor1` fordert `Role3` für `Role7`; `PrerequisiteRoleDelegationGrantor2` fordert dagegen `Role5` für `Role7`. Analog dazu verhält es sich bei beiden `PrerequisiteRoleDelegationDelegate`-Instanzen.

(12)/(13) `Login1` besitzt nicht die Berechtigung `Role3`, die laut `PrerequisiteRoleDelegationGrantor1` allerdings für den Delegierenden erforderlich ist. Auch `Login2` als vermeintlicher Delegationsempfänger besitzt nicht die Berechtigung `Role4`, die `PrerequisiteRoleDelegationGrantor1` vorgibt. Dies ist unabhängig davon, dass bereits in (6)/(7) festgestellt wurde, dass sich `PrerequisiteRoleDelegationGrantor1` und `PrerequisiteRoleDelegationDelegate1` auf die falsche Rolle beziehen.

(15) Die Delegation ist über `Duration1` zeitlich beschränkt: Sie gilt nur für den Systemzeitraum 1-3. Die aktuelle Systemzeit (`SystemClock1`) ist allerdings bereits 5, womit die Delegation nicht mehr gültig ist.

(17) `MultipleDelegationCheck1` definiert, dass `Login1` `Role1` nicht (0-mal) delegieren darf, wogegen verstoßen wurde.

(26) Mit `SystemClock1` und `SystemClock2` existieren 2 Instanzen, die die globale Systemzeit repräsentieren. Wie `Timestamp1` und `Timestamp4` zeigen, kann dies zu Widersprüchen führen, weshalb es das OCL-Constraint verhindert.

(5), (9) und (14) beziehen sich auf die Handhabung und Kontrolle der weiteren möglichen Kombinationen aus `Person`- und `Login`-Instanzen als Delegierender und Delegationsempfänger, die an dieser Stelle nicht weiter betrachtet werden. Da das Negativbeispiel keine korrekte Delegation abbildet, kommt der optionale `maxlevelcheck`, der angibt, wie häufig eine Delegation weiter delegiert werden kann, nicht zum Einsatz.

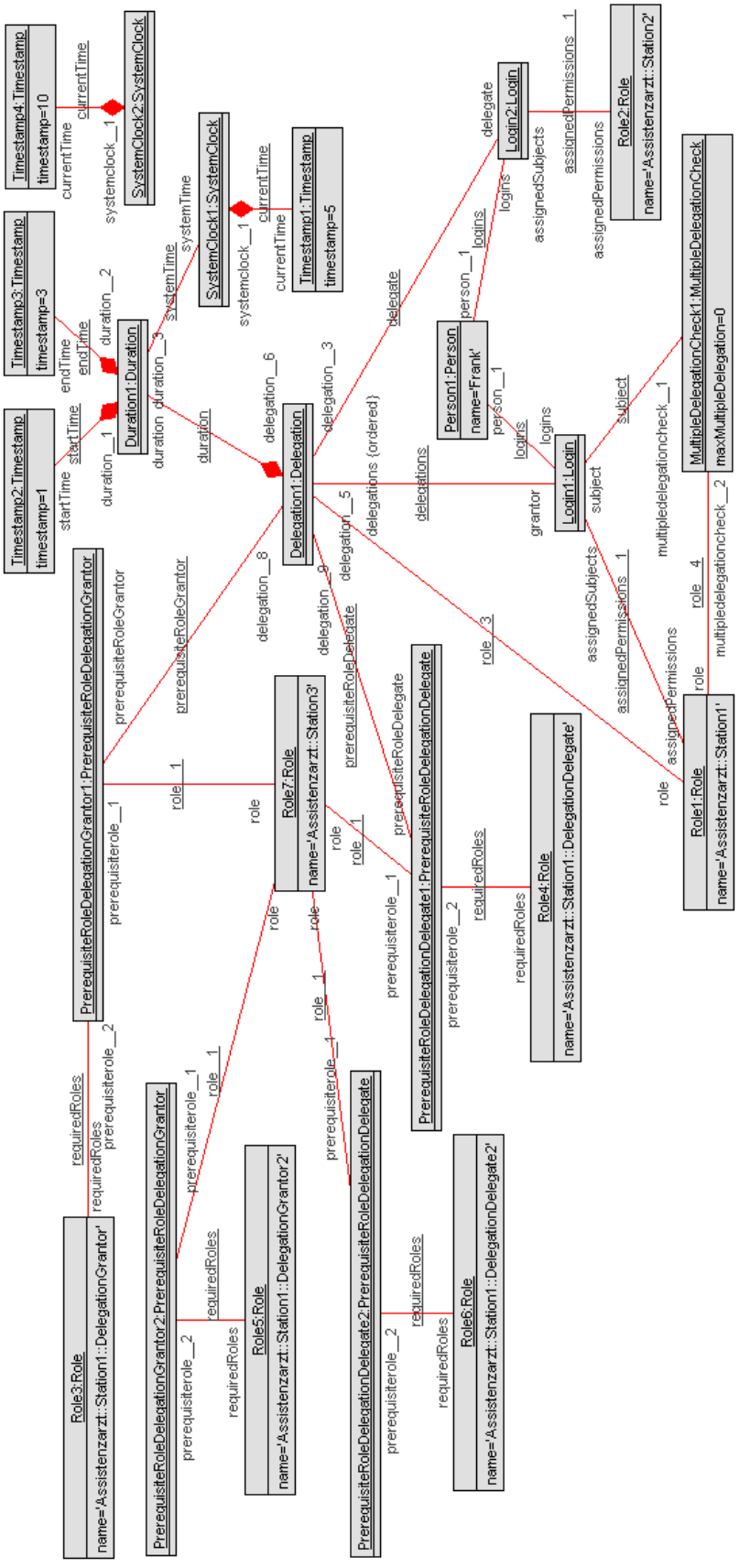


Abbildung 7.4: Negativbeispiel, das den Großteil der OCL-Constraints des Delegationskonzepts verletzt.

`OperationAccess`-Komposition: Im originalen Grundlagenmodell wurde die Beziehungen der `OperationAccess` so modelliert, dass es über Kompositionsbeziehungen sowohl Teil einer `Operation` als auch einer `Permission` sein kann (EMF „Containment"-Eigenschaft gesetzt), wobei `OperationAccess` auch genau als Bindeglied dazwischen gedacht ist, wie das Kapitel 6 *Fallstudie Krankenhaus* zeigt. Abbildung 7.5 zeigt dies beispielhaft, wobei `Step` als Spezialisierung von `Operation` ebenfalls eine Kompositionsbeziehung besitzt. Zu beachten ist nun, dass nicht beide Kompositionen vorhanden sein dürfen, da die `OperationAccess1` nicht gleichzeitig Bestandteil von `Permission1` und `Step1` sein kann. Der USE-`check`-Kommandozeilenbefehl liefert entsprechend folgende Fehlermeldung, falls es wie in Abbildung 7.5 doch geschieht:

```
1 Error: Object 'OperationAccess1' is shared by object 'Step1' and object '
    Permission1'.
```

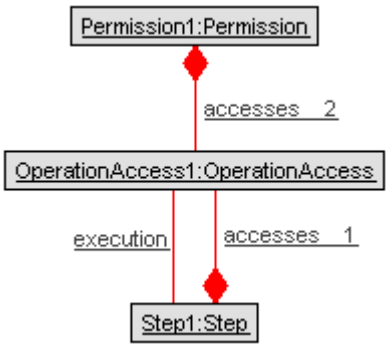


Abbildung 7.5: Invalider Systemzustand: Doppelte Komposition von `OperationAccess`.

Operationsausführung standardmäßig erlaubt: Standardmäßig müssen einer `Operation`-Instanz, wozu über Vererbung auch die Klassen `Process`, `Step` und `Mask` gehören, keine zur Ausführung (repräsentiert über `Execution`-Instanz) nötigen Berechtigungen zugewiesen werden – weder ein OCL-Constraint noch eine Multiplizität einer Assoziation fordert dies. Dies stellt eine Verletzung des in Unterabschnitt 2.1.2 *Designprinzipien sicherer Systeme* vorgestellten Designprinzips für sichere System „Fail-safe defaults" dar. Es besteht also das Risiko, die Berechtigungszuweisung zu vergessen, wodurch dann jedes Subjekt unabhängig davon, welche Berechtigungen es besitzt, diese Operation ausführen darf. Des Weiteren kann später durch Begutachtung des Objektdiagramms nicht festgestellt werden, ob das Nichtvorhandensein von Berechtigungen gewollt ist oder doch die Festlegung der erforderlichen Berechtigungen zur Ausführung vergessen wurde. In Abbildung 7.6 ist das beschriebene Szenario modelliert, welches von USE als gültiger Systemzustand bewertet wird. Im Vergleich dazu zeigt Abbildung 7.7 ein gültiges Beispiel mit Berechtigungen. Bei der zweiten Abbildung würde ein Weglassen des `assignedPermissions_1`-Links dazu führen, dass durch das OCL-Constraint `Execution::subjecthaspermissions` der Systemzustand von USE als ungültig bewertet worden wäre, da `Person1` nicht die

`Permission1` besitzt, die zur Ausführung (`Execution1`) von `Step1` erforderlich ist.

Eine mögliche Lösung, um das „Fail-safe defaults“-Designprinzip einzuhalten, wäre es, die Multiplizität einer Beziehung zwischen `Operation` und `Access`, wovon `OperationAccess` eine Spezialisierung ist, sowie zwischen `Access` und `Permission` so zu definieren, dass mindestens ein Link erstellt und damit eine Berechtigung der `Operation` zugewiesen werden muss (`[n..m]` mit $n \geq 1$ und $m \geq n$). Sollen wirklich alle Subjekte eine bestimmte `Operation` ausführen, können einfach alle Subjekte explizit die Berechtigung erhalten, diese auszuführen.

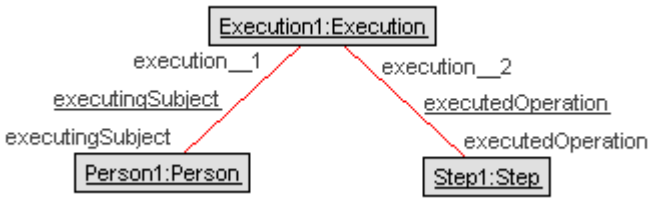


Abbildung 7.6: Valider Systemzustand: Ausführung eines `Step`s, der keine Berechtigungen erfordert.

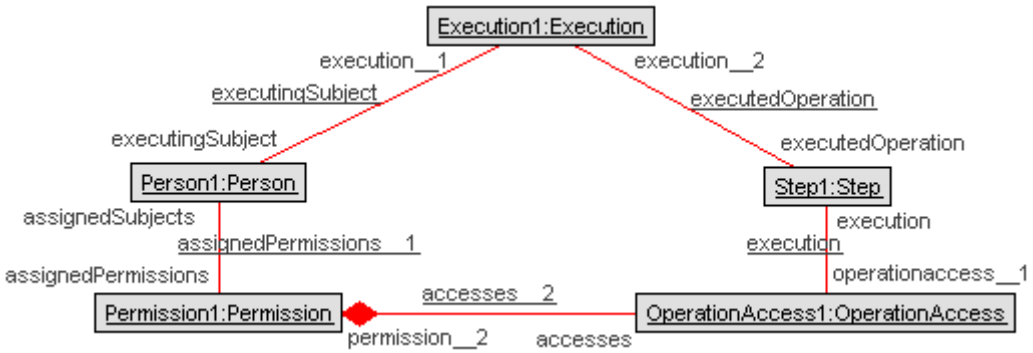


Abbildung 7.7: Valider Systemzustand: Ausführung eines `Step`s, für den Berechtigungen erforderlich sind.

Fehlerhafte Berechtigungsermittlung bei Rollenhierarchien: Bei der Bestimmung aller Berechtigungen eines Subjekts werden Rollenhierarchien (siehe Abschnitt 2.2 *Zugriffskontrollmodelle / RBAC (Role-Based Access Control)*) nicht korrekt berücksichtigt, sodass der Systemzustand in Abbildung 7.8 aufgrund des OCL-Constraints `Execution::subjecthaspermission` als ungültig gewertet wird. Die Verwendung von Rollen an sich funktioniert korrekt, wie der gültige Systemzustand in Abbildung 7.9 zeigt. Aus Sicht der IT-Sicherheit ist dieser Fehler allerdings nicht so kritisch, wie er klingen mag, denn bei Subjekten (`Person` und `Login`) wurde das „Fail-safe defaults“-Designprinzip eingehalten: Ein Subjekt hat standardmäßig keine Berechtigungen, sondern sie müssen explizit zugewiesen werden.

Aus Sicht der Betriebssicherheit ist es hingegen eine klare und kritische Verletzung der folgenden „Darf nicht“-Anforderung aus Abschnitt 7.2 *Betriebs- und Informationssicherheitsanforderungen*: „Es darf nicht passieren, dass trotz vorhandener Berechtigungen eine Operation nicht ausgeführt werden kann“.

Die Lösung ist die Anpassung des OCL-Ausdrucks, der die Funktion `Permission::getAllPermissions()` beschreibt, wie im folgenden Listing gezeigt: Statt nur die aktuelle `Role`-Instanz und deren `Permission`-Instanzen zurückzuliefern, werden zuerst alle hierarchisch niedrigeren Rollen (`childeren`-Assoziation) ermittelt und dann davon alle `Permission`-Instanzen.

```
1 -- ### VORHER ###
2 if self.oclIsTypeOf(Permission) then
3   Set{self}
4 else
5   if self.oclIsTypeOf(highlevel::rbac::Role) then
6     self.oclAsType(highlevel::rbac::Role).assignedPermissions->union(Set{self
        })
7   else
8     Set{}
9   endif
10 endif
11
12 -- ### NACHHER ###
13 if self.oclIsTypeOf(Permission) then
14   Set{self}
15 else
16   if self.oclIsTypeOf(highlevel::rbac::Role) then
17     let allSubRolesIncludingSelf : Set(highlevel::rbac::Role) = Set{self.
          oclAsType(highlevel::rbac::Role)}->closure(children)->union(Set{self.
          oclAsType(highlevel::rbac::Role)}) in
18     allSubRolesIncludingSelf->union(allSubRolesIncludingSelf.
          assignedPermissions)->asSet()
19   else
20     Set{}
21   endif
22 endif
```

Initialer Schritt eines Geschäftsprozesses nicht ausgewertet: Ein `Process` besitzt die drei `[0..*]`-Beziehungen `steps` (Geschäftsprozessteilschritte), `transitions` (Übergänge zwischen Geschäftsprozessschritten) und `initialState`, was nach der Namensgebung die möglichen Startpunkte in Form von Geschäftsprozesselementen (`IStep`) spezifiziert. Allerdings wird `initialState` im Modell nicht weiter verwendet und somit auch in keinem OCL-Constraint validiert. Eine mögliche Überprüfung wäre, dass, wenn der Geschäftsprozess Links in `steps` besitzt, mindestens ein – wenn es mehrere Startpunkte geben soll – Link davon auch `initialState` zugewiesen ist bzw. `initialState` auf eine `Mask`-Instanz verweist, die unter einem der `steps` hängt; denn `initialState` ist als Assoziation zum abstrakten `IStep` definiert.

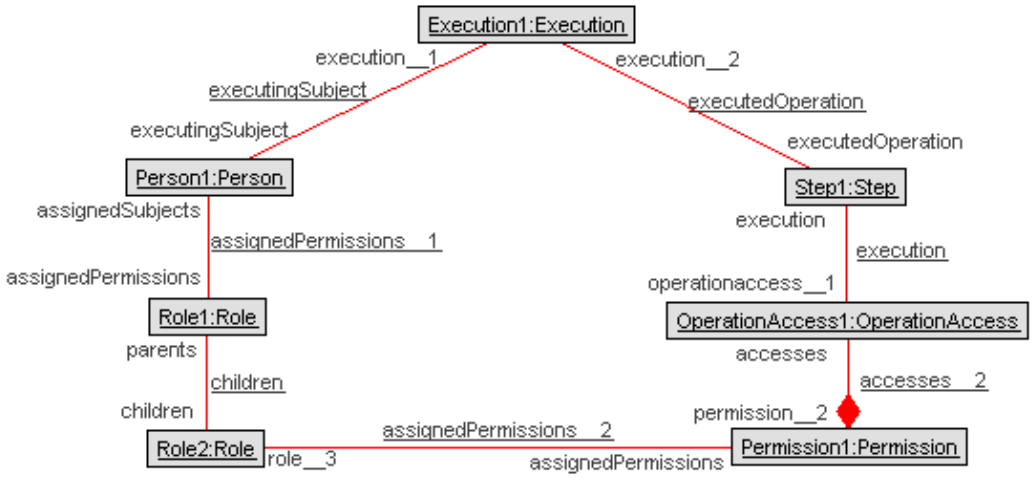


Abbildung 7.8: Fälschlicherweise invalider Systemzustand bei Rollenhierarchien.

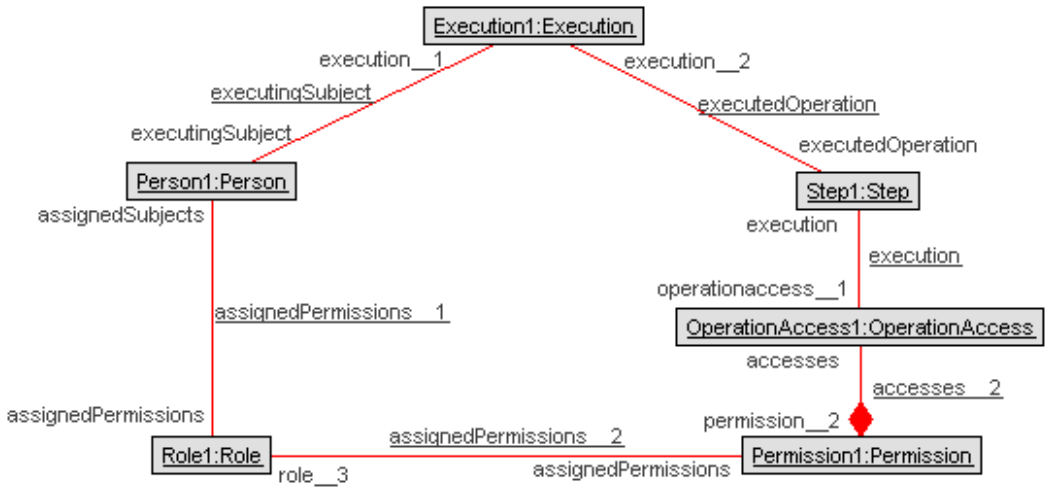


Abbildung 7.9: Valider Systemzustand unter Verwendung einer Rolle.

Ungebundene Ausführungsreihenfolge: Ein `Trace` fasst mehrere Ausführungsschritte (`Execution`) über die geordnete Containment-Beziehung `executions` zusammen. Es findet allerdings keine Überprüfung statt, ob die ausgeführten Schritte überhaupt über eine `Transition` miteinander verbunden sind, womit eine wahllose Kombination möglich ist, wie Abbildung 7.10 zeigt: Zuerst wird `Step3` ausgeführt und anschließend `Step1`, wobei diese nicht direkt über eine `Transition` verbunden sind. Basierend auf den OCL-Constraints im Modell bewertet USE den abgebildeten Systemzustand als einen validen. Die vorab beschriebene Problematik bezüglich der Nichtberücksichtigung des `initialStep` zeigt sich auch noch einmal in der genannten Abbildung.

Es gibt zwei Arten, wie mehrere `Mask`-Instanzen in Beziehung stehen können:

Eine Eingabemaske kann eine andere inkludieren (`includes`-Assoziation) oder eine Beziehung zu einer nachfolgenden Eingabemaske besitzen (`successors`-Assoziation). Die Handhabung dieser beiden Assoziationen funktioniert auch korrekt; so muss es beispielsweise bei Verwendung der `successors`-Assoziation zwischen zwei `Mask`-Instanzen, die derselben `Step`-Instanz zugewiesen sind, eine `Transition`-Instanz von der `Step`-Instanz auf sich selber geben. Wenn allerdings wie in Abbildung 7.10 die zwei `Mask`-Instanzen `Mask1` und `Mask2` ohne Beziehung zueinander stehen, wird dies als gültiger Systemzustand gewertet, obwohl nicht ersichtlich ist, wie die Ausführungsreihenfolge ist.

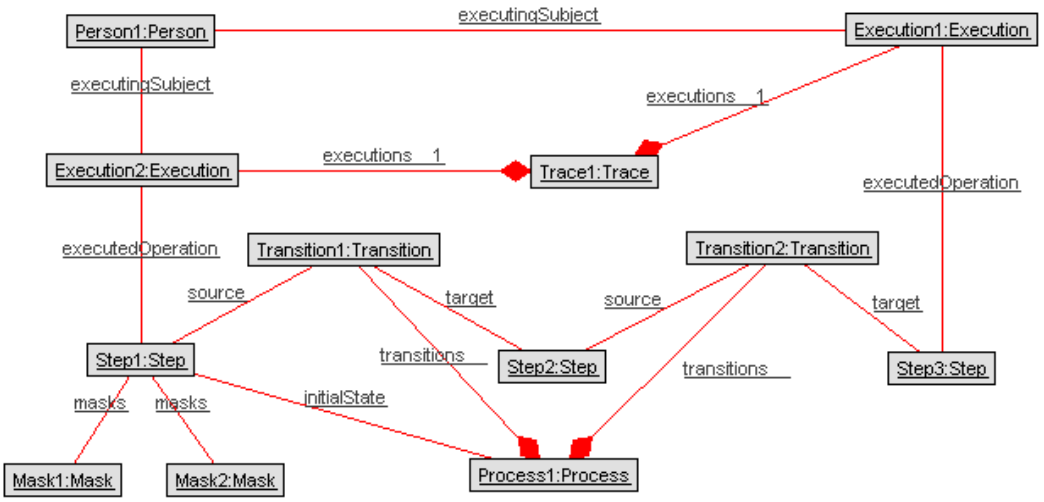


Abbildung 7.10: Fälschlicherweise valider Systemzustand bei der Ausführungsreihenfolge.

Ausführung beliebiger Geschäftsprozesselementtypen ohne tiefer gehende Berechtigungsprüfung: Das Modell erlaubt es, jede Art von `Operation` über eine `Execution`-Instanz auszuführen. Von `Operation` (abstrakt) erbt `IStep` (abstrakt) und davon wiederum `Process`, `Step` und `Mask`, womit die letzten drei ausführbar sind. Damit wird modelliert, dass ein Geschäftsprozess aus mehreren Arbeitsschritten bestehen kann, die wiederum über Eingabemasken realisiert sind. Bei der Überprüfung, ob das ausführende Subjekt überhaupt über die nötigen Berechtigungen verfügt, wird allerdings immer nur die direkte Ebene und nicht tiefer gehende Ebenen berücksichtigt. Abbildung 7.11 zeigt einen gültigen Systemzustand, wobei zu sehen ist, dass bei Ausführung von `Step1` über `Execution1` nicht erkannt wird, dass die zu `Step1` gehörende Eingabemaske `Mask1` die Berechtigung `Permission1` erfordert, die das Subjekt `Person1` allerdings nicht besitzt. Gleiches gilt für die Ausführung von `Process1` über `Execution2`.
Interpretiert man das Modell so, dass eine `Execution`-Instanz auch die Ausführung darunter liegender `Operation`-Instanzen meint, ist dies ein klarer Sicherheitsverstoß, weil Operationen ausführbar wären, für die das Subjekt keine Berechtigungen hat.

Würden hingegen darunter liegende `Operation`-Instanzen nicht mit ausgeführt werden, sondern das Modell so eingesetzt, dass auch diese `Operation`-Instanzen für die Ausführung mit einer `Execution`-Instanz versehen werden, würde die fehlende, rekursiv absteigende Berechtigungsprüfung immer noch ein unschönes Verhalten modellieren: Ein Subjekt kann beginnen, einen Geschäftsprozess auszuführen, muss dann aber mitten im Vorgang (`Trace`) feststellen, dass seine Berechtigungen für den n-ten Arbeitsschritt bzw. die m-te Eingabemaske des n-ten Arbeitsschrittes nicht ausreicht. Wünschenswert wäre es, wenn ein Geschäftsprozess nicht erfolgreich bis zum Ende durchgeführt werden kann, dieser erst gar nicht gestartet werden kann.

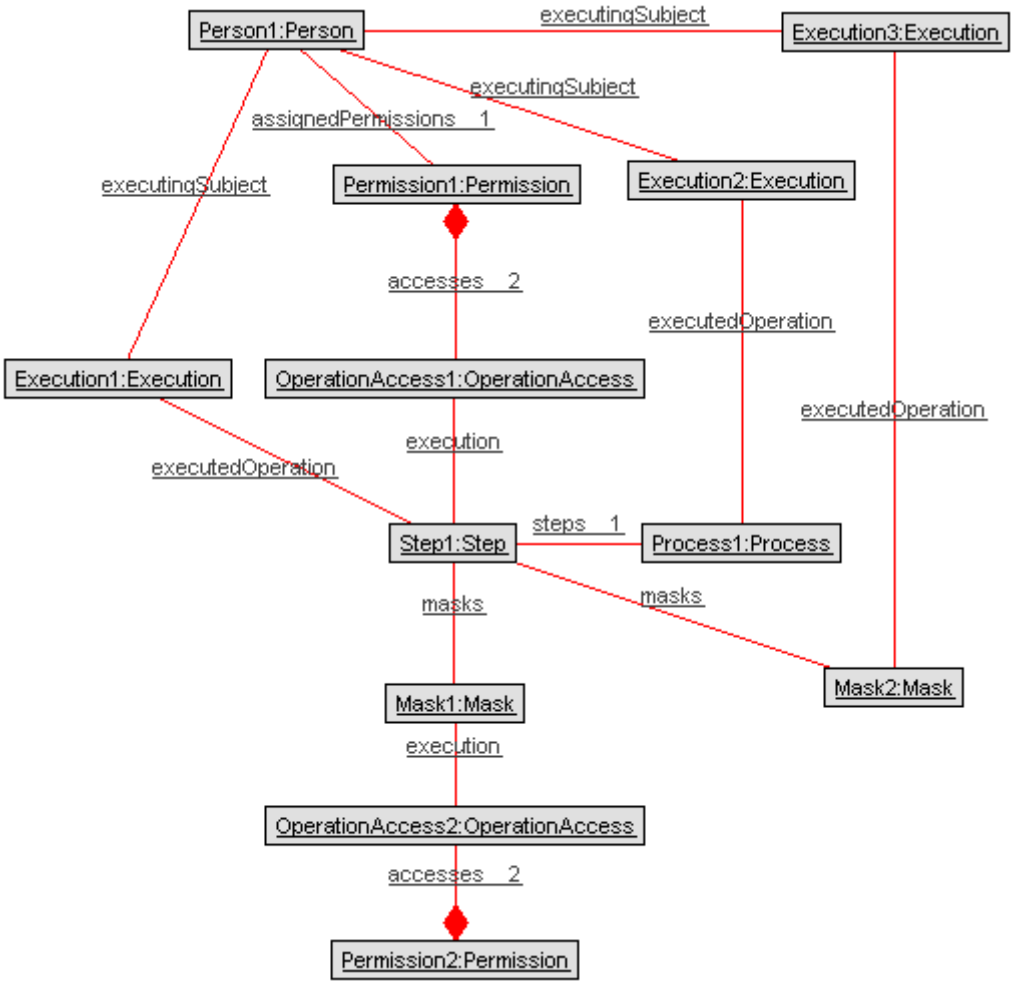


Abbildung 7.11: Fälschlicherweise valider Systemzustand bei der Ausführungsreihenfolge.

7.5.3 Ergebnisse dynamischer Tests (Model Validator)

Zuerst werden einige Punkte behandelt, die bei den dynamischen Tests beachtet werden müssen, bevor dann an einem Beispiel der Modellvalidierungsworkflow verdeut-

licht wird. Dann folgt ein Vergleich der im Model Validator verfügbaren SAT-Solver, und schließlich wird der Modellvalidierungsworkflow auf die Fallstudienbeispiele aus Kapitel 6 *Fallstudie Krankenhaus* angewendet.

Model Validator Containment-Beziehungen: Zu beachten ist, dass der Model Validator Containment-Links/`composition`, genauer gesagt deren Abwesenheit, anders bewertet als USE selbst: Durch den folgenden USE-Modell-Ausschnitt ist es dem Model Validator nicht möglich, eine Lösung zu finden (wie das darauf folgende Listing zeigt), wenn die `*.properties`-Datei zusätzlich zum Ausgangsobjektdiagramm ausschließlich eine `Execution`-Instanz, sowie jeweils einen Link ihrer beiden Assoziation fordert. Ein Ändern der Konfigurationsdatei-Flags `aggregationcyclefreeness` und `forbiddensharing` führt zu keinem anderen Ergebnis. Sobald eine der beiden Beziehungen allerdings auf `association` (statt `composition`) umgestellt wird, gilt die Problemstellung – bei korrekter Spezifizierung aller anderen Parameter – für den Model Validator nicht mehr als `TRIVIALLY_UNSATISFIABLE`. Dem Strukturcheck von USE selbst hält eine `Execution`-Instanz ohne `execution__1`- oder `execution__2`-Link hingegen stand – wobei die gleichzeitige Verwendung wieder Fehler bezüglich der Zugehörigkeit aufwirft.

```
composition executions__1 between
Trace [0..*] role trace__1
Execution [0..*] role executions ordered
end

composition executions__2 between
RuntimeModel [0..*] role runtimemodel__2
Execution [0..*] role executions
end
```

```
TRIVIALLY_UNSATISFIABLE
Unsatisfiable proof:
< node: (all self: one Execution | ((((true && !((executions__1 . self) =
    Undefined)) && ((executions__2 . self) = Undefined)) || ((true && ((
    executions__1 . self) = Undefined)) && !((executions__2 . self) =
    Undefined))) || ((true && ((executions__1 . self) = Undefined)) && ((
    executions__2 . self) = Undefined)))), literal: -2147483647, env: {}>
```

Als Lösung für das weitere Vorgehen wurde die `execution__2`-Beziehung von `composition` auf `association` umgestellt.

Verschachtelte `composition`-Beziehungen, wie sie bei der zeitlich beschränkten Delegation vorkommen (eine `Timestamp`-Instanz ist ein Teil von einer `Duration`-Instanz und diese wiederum ist Teil einer `Delegation`-Instanz), führt zu einem ähnlichen Verhalten wie in obigem Listing gezeigt. Deswegen wurden die Beziehungen `startTime` und `endTime` zwischen `Duration` und `Timestamp` im USE-Modell ebenfalls auf `association` umgestellt.

„Undefined“ Strings: Das Modell nutzt an vielen Stellen als Attribut-Datentyp `String`, z.B. beim Namen einer Rolle. Wird im Objektdiagramm einem String-

Attribut nicht explizit eine Zeichenkette zugewiesen, gilt das Attribut als `Undefined`, und bei der Generierung gültiger Systemzustände belegt der Model Validator nicht initialisierte String-Attribute mit konkreten Werten. In der Konfigurationsdatei können neben der Anzahl an Instanzen und Links nämlich unter anderem auch die minimale und maximale Anzahl erlaubter, *unterschiedlicher* Strings definiert werden. Der Model Validator würde dann, wenn das Maximum an verschiedenen Zeichenketten noch nicht durch das Teilobjektdiagramm erreicht ist, weitere Strings der Form `String1`, `String2`, ... dem Objektdiagramm hinzufügen und verschiedene Belegungen eines String-Attributs als mehrere Lösungen bewerten. Dieses Feature ist in diesem Fall allerdings nicht gewünscht, weshalb von der automatischen `*.properties`-Datei-Generierung keine minimale oder maximale Anzahl verschiedener Strings aktiviert wird. Somit stehen dem Model Validator nur die Strings zur Verfügung, die bereits im Teilobjektdiagramm verwendet werden. Da aber auch diese durch den Model Validator nicht vergeben werden sollen, darf das Teilobjektdiagramm für kein String-Attribut den Wert `Undefined` aufweisen. Ansonsten würde beispielsweise ein als `Undefined` ausgewiesenes `name`-Attribut einer `Step`-Instanz den Namen einer `Permission`-Instanz zugewiesen bekommen, was aus Sicht der Fallstudie zu einem falschen Ergebnis führen würde.
Dabei ist zu beachten, dass die Tatsache ausgenutzt wird, dass die vom Model Validator automatisch zu generierenden Instanzen beim zu validierenden Modell keine String-Attribute aufweisen. Andernfalls müsste die automatische Erzeugung von Strings durch den Model Validator aktiviert werden.

Beispielhafter Funktionsnachweis des dynamischen Modellvalidierungsworkflows: Bevor die komplexeren Fallbeispiele aus Kapitel 6 *Fallstudie Krankenhaus* evaluiert werden, wird zunächst an einem kleineren und übersichtlicheren Beispiel, das thematisch wie die Fallstudie auch beim Krankenhausinformationssystem angesiedelt ist, die allgemeine Funktionsweise des Ablaufs der dynamischen Modellvalidierung mit dem in Ecore2USE integrierten USE Model Validator nachgewiesen.

Abbildung 7.13 zeigt die Eingaben für die Modellvalidierung über Ecore2USE:

- Von USE mitgeliefertes Batchfile, um USE und letztlich auch den Model Validator skriptgesteuert starten zu können.
- Vorab mittels Ecore2USE von Ecore zu USE transformiertes Modell (Ecore2USE „Ecore Transformation“-Tab).
- Teilobjektdiagramm, welches ein Geschäftsprozess sowie die Berechtigungen für einen Login beschreibt und als Grundlage für die Arbeit des Model Validators dient. Abbildung 7.12 zeigt die grafische Repräsentation dieser SOIL-Datei.
- Liste aller Security-Policy-Constraints und Non-Security-Policy-Constraints im Modell (siehe folgendes Listing). Die Textdatei, die diese Liste enthält, wurde automatisch bei der Ecore-USE-Modelltransformation generiert (Ecore2USE

„Ecore Transformation“-Tab).

```
1  RoleExclusionCheck::check
2  PrerequisiteRoleCheck::prerequisiterole
3  Cardinality::mincardinalitycheck
4  Cardinality::maxcardinalitycheck
5  Delegation::maxlevelcheck
6  Delegation::prerequisiterolegrantorcheck
7  Delegation::prerequisiteroledelegatecheck
8  MultipleDelegationCheck::maxmultipledelegation
```

- Konfigurationsdatei, die als Eingabe für den intern gestarteten Model Validator dient. Sie wurde vorab mittels Ecore2USE („Properties File Generation“-Tab) unter Angabe des genannten Objektdiagramms und des Modells in USE-Form generiert, womit in der `*.properties`-Datei die Anzahl an Instanzen und Links genau auf die im Objektdiagramm verwendeten gesetzt wurde und für nicht verwendete Klassen und Beziehungen auf 0. Anschließend wurde diese Datei über die in USE integrierte grafische Benutzeroberfläche „Model Validator Configuration“ manuell um die Instanzen und Links ergänzt, die vom Model Validator während der Modellvalidierung automatisch erzeugt werden sollen. In diesem Fall wurde die Anzahl der `Execution`-Instanzen und der dazugehörigen Links fest auf 3 gesetzt, um auch eine Beziehung zu den Instanzen des Teilobjektdiagramms herstellen zu können. Zudem sollen eine `Trace`-Instanz erstellt werden und 3 `Execution`-Instanzen über die `executions__1`-Beziehung irgendeiner `Trace`-Instanz – die in diesem Fall durch ihre Einzigartigkeit bekannt ist – zugewiesen werden (siehe folgendes Listing).

```
1  # ------------------------------------------------------- Execution
2  Execution_min = 3
3  Execution_max = 3
4
5  # executingSubject (execution__1:Execution, executingSubject:Subject)
6  executingSubject_min = 3
7  executingSubject_max = 3
8
9  # executedOperation (execution__2:Execution, executedOperation:
       Operation)
10 executedOperation_min = 3
11 executedOperation_max = 3
12
13 # ---------------------------------------------------------- Trace
14 Trace_min = 1
15 Trace_max = 1
16
17 # executions__1 (trace__1:Trace, executions:Execution) - - - - - - - -
18 executions__1_min = 3
19 executions__1_max = 3
```

- Verzeichnis, in dem alle vom Model Validator generierten Systemzustände abgespeichert werden; entweder im Unterverzeichnis `invalid`, wenn sich her-

ausgestellt hat, dass der Systemzustand anschließend gegen Security-Policy-Constraints oder Non-Security-Policy-Constraints verstößt, oder im Unterverzeichnis `valid`, wenn keine Policy-Verletzung festgestellt werden konnte.

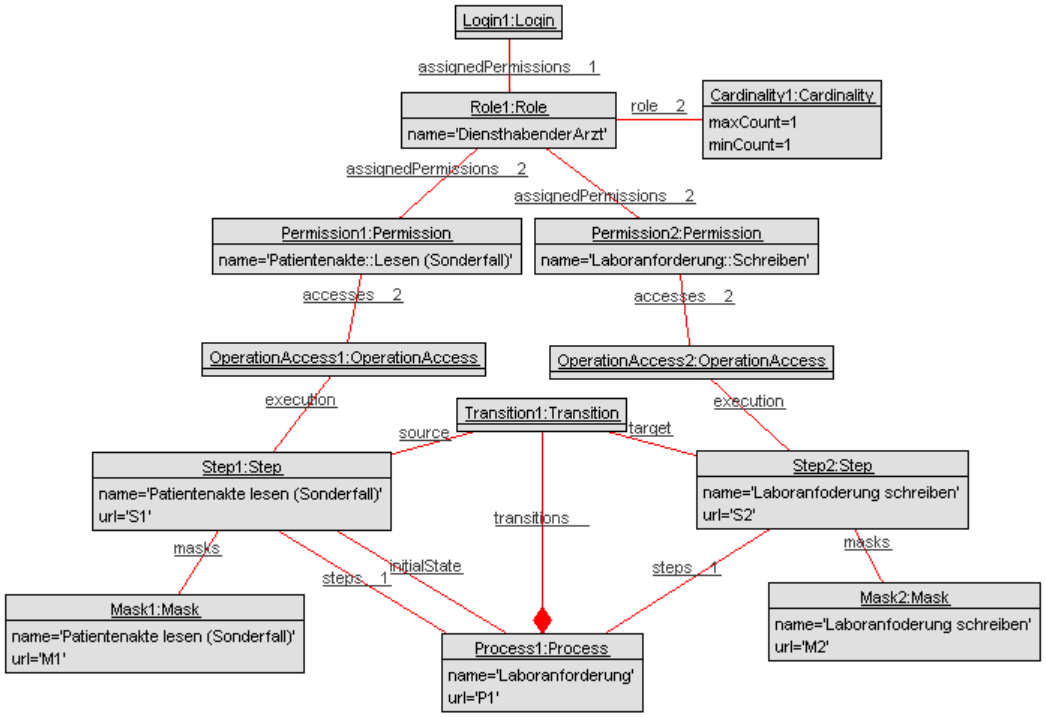


Abbildung 7.12: Objektdiagramm eines Arzt-Logins in USE.

Entsprechend dem in Unterabschnitt 7.4.2 *Dynamische Modellvalidierung mittels USE Model Validator* beschriebenen Verfahren konnte, wie Abbildung 7.13 zeigt, keine Policy-Verletzung bei einer Kardinalität der Lösungsmenge von 35 ermittelt werden. Dabei ist 35 auch rechnerisch die erwartete, vom Model Validator generierte Anzahl an Lösungen: Durch die vorgegebene Anzahl an `executingSubject`- und `executedOperation`-Links muss jede `Execution`-Instanzen mit einer `Subject`-Instanz verbunden werden, wofür hier nur `Login1` in Frage kommt, sowie einen Link zu einer der `Operationen`-Instanz (`Step1`, `Step2`, `Mask1`, `Mask2` oder `Process1`) besitzen. Über die Formel der Kombinatorik für k-Auswahlen aus einer n-Menge mit Wiederholung und ohne Berücksichtigung der Reihenfolge $\binom{n+k-1}{k}$ lässt sich die Anzahl berechnen, wobei n = Anzahl `Operation`-Instanzen und k = Anzahl `Execution`-Instanzen: $\binom{5+3-1}{3} = 35$.
Im angegebenen Ergebnisverzeichnis befinden sich somit 35 SOIL-Dateien, die die generierten Systemzustände repräsentieren, im `valid`-Unterverzeichnis und 0 im `invalid`-Unterverzeichnis. Abbildung 7.14 zeigt beispielhaft einen von ihnen.

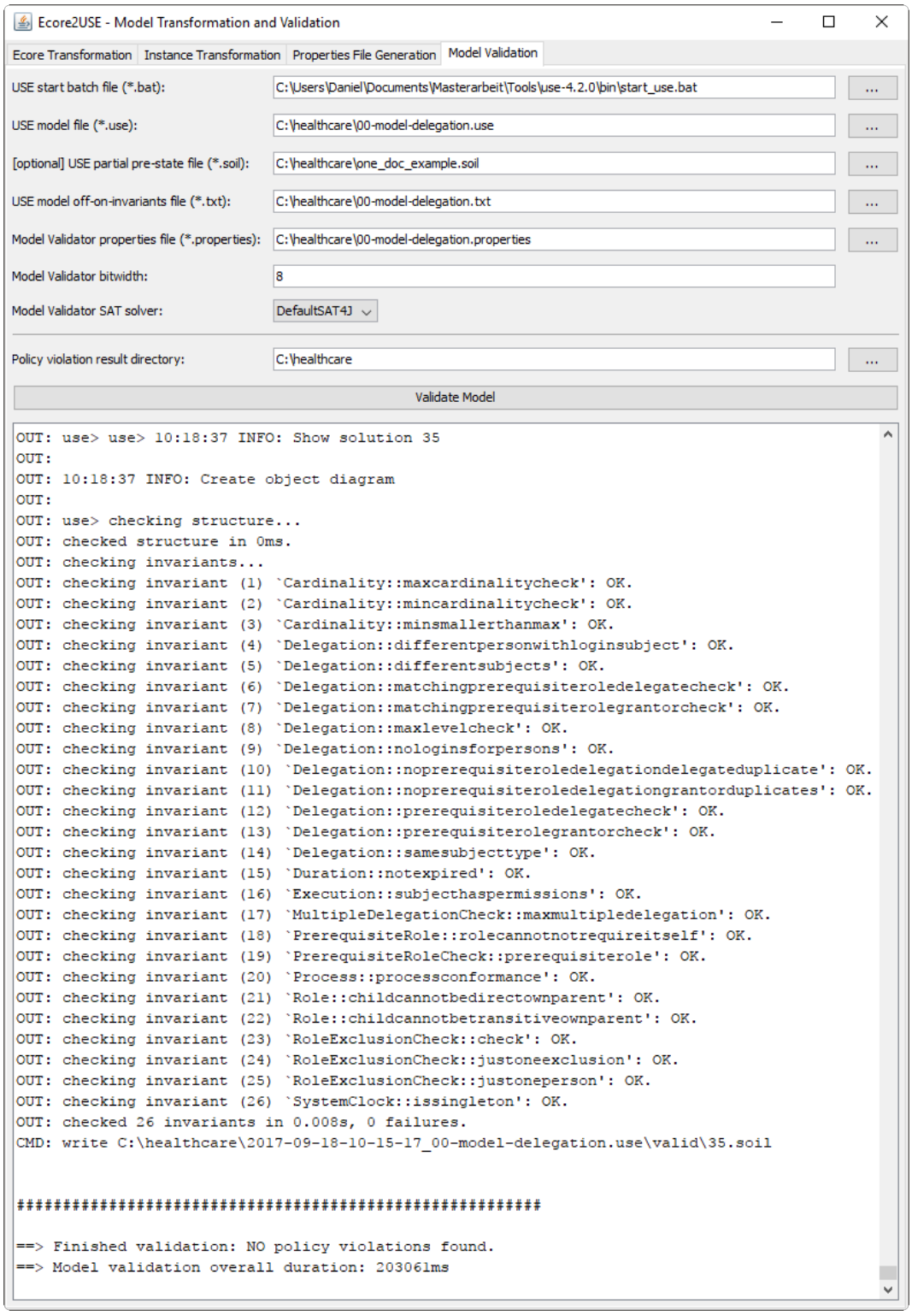


Abbildung 7.13: Model Validierung über Ecore2USE.

7.5 Ergebnisse

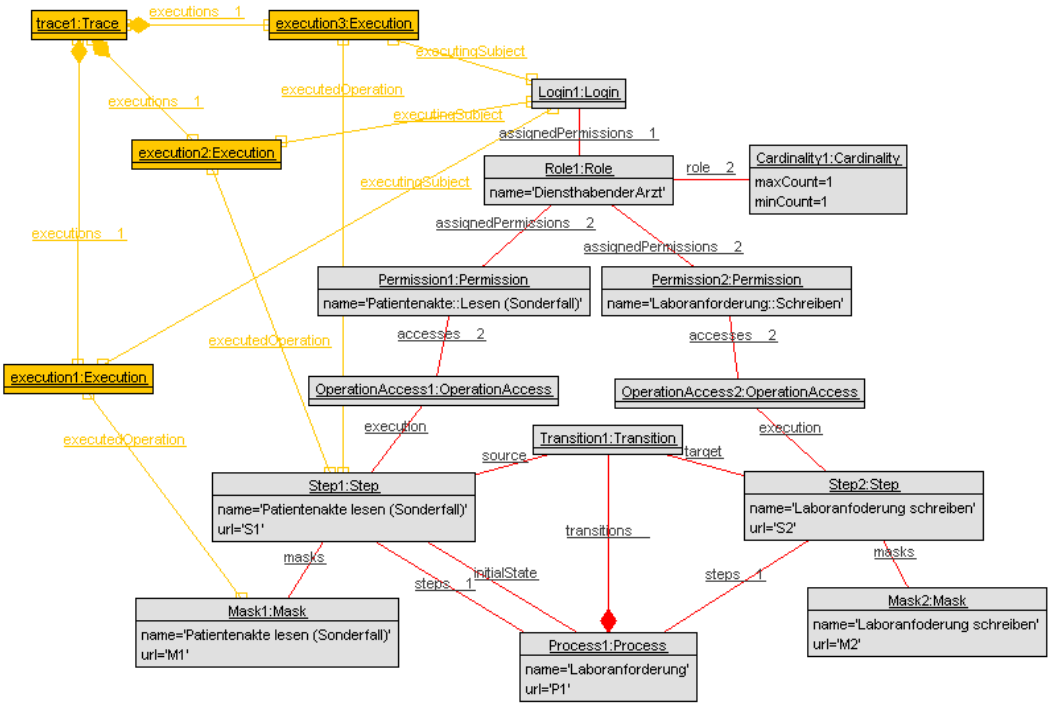


Abbildung 7.14: Valider, vom Model Validator generierter Systemzustand. Vom Model Validator erzeugte Instanzen und Links sind gelb hervorgehoben.

SAT-Solver-Auswahl: Der Model Validator stellt mehrere SAT-Solver zur Auswahl, die für die internen Berechnungen verwendet werden können. Unter Windows stehen die SAT-Solver DefaultSAT4J, LightSAT4J, MiniSat und MiniSatProver zur Verfügung. Unter Verwendung des vorab vorgestellten Einstiegsbeispiels – als Repräsentant für die vom Model Validator zu lösenden Aufgaben – wurden die SAT-Solver daraufhin verglichen, wie lange diese benötigen, um mittels Ecore2USE-„Model Validation“ die Lösungsmenge (bis zum Abschluss des `mv -scrollingAll`-Teils) zu bestimmen. Dabei wurde pro SAT-Solver die identische Lösungsmenge dreimal berechnet, um Schwankungen, z.B. durch die Ecore2USE-Hülle um den Model Validator oder Betriebssystem-Interrupts heraus mitteln zu können. Die detaillierten Ergebnisse zeigt Tabelle 3. Aus den Messwerten ergibt sich, dass der SAT-Solver MiniSat den besten/niedrigsten Mittelwert bei im Vergleich geringer Standardabweichung aufweist, weshalb dieser bei allen folgenden Einsätzen vom Model Validator genutzt wird.

Tabelle 3: Vergleich der Model Validator-SAT-Solver basierend auf jeweils 3 Durchläufen (x_i) zur Bestimmung der reinen Lösungsmenge für das Beispiel aus dem vorangegangenen Abschnitt. Model Validator-Bitwidth = 8. Alle Werte in [ms]. Arithmetisches Mittel: $\bar{x} = \frac{1}{3}\sum_{i=1}^{3} x_i$, Standardabweichung $s = \sqrt{\frac{1}{3-1}\sum_{i=1}^{3}(x_i - \bar{x})^2}$

SAT-Solver	x_1	x_2	x_3	Mittelwert $\bar{x}$	Standardabweichung s
DefaultSAT4J	97969	99335	98466	98590	691
LightSAT4J	99324	97845	95793	97654	1773
MiniSat	94503	96367	95349	95406	933
MiniSatProver	94083	102434	98676	98397	4182

Validierung Fallstudienbeispiele: Die Grundlage für die vorab beschriebene, dynamische Modellvalidierung (siehe Unterabschnitt 7.4.2 *Dynamische Modellvalidierung mittels USE Model Validator*) sind modifizierte Varianten der Fallstudienobjektdiagramme aus dem vorangegangenen Kapitel 6 *Fallstudie Krankenhaus*: Die Fallstudienobjektdiagramme, die beispielhaft zeigen sollten, wie das Modell eingesetzt werden kann, werden auf die Instanzen und Links reduziert, die als permanent angenommen werden. Permanent sind zum einen Geschäftsprozesse mit den nötigen Berechtigungen und zum anderen die Subjekte mit zugewiesenen Berechtigungen, andersherum ausgedrückt: Die `Execution`-Instanzen und die dazugehörigen Links, die die eigentliche Ausführung von Geschäftsprozessen durch Subjekte modellieren, werden entfernt. Diese sollen anschließend automatisch durch den Model Validator generiert werden.

Daraus ergibt sich als Grundlage für die Model Validierung mittels Ecore2USE das in Abbildung 7.15 gezeigte Objektdiagramm für den Anwendungsfall **„Patientenaufnahme“**. Dabei wurden alle „Undefined“-Strings des ursprünglichen Objektdiagramms zur Patientenaufnahme (Abbildung 6.1) durch konkrete Zeichen-

ketten ersetzt.

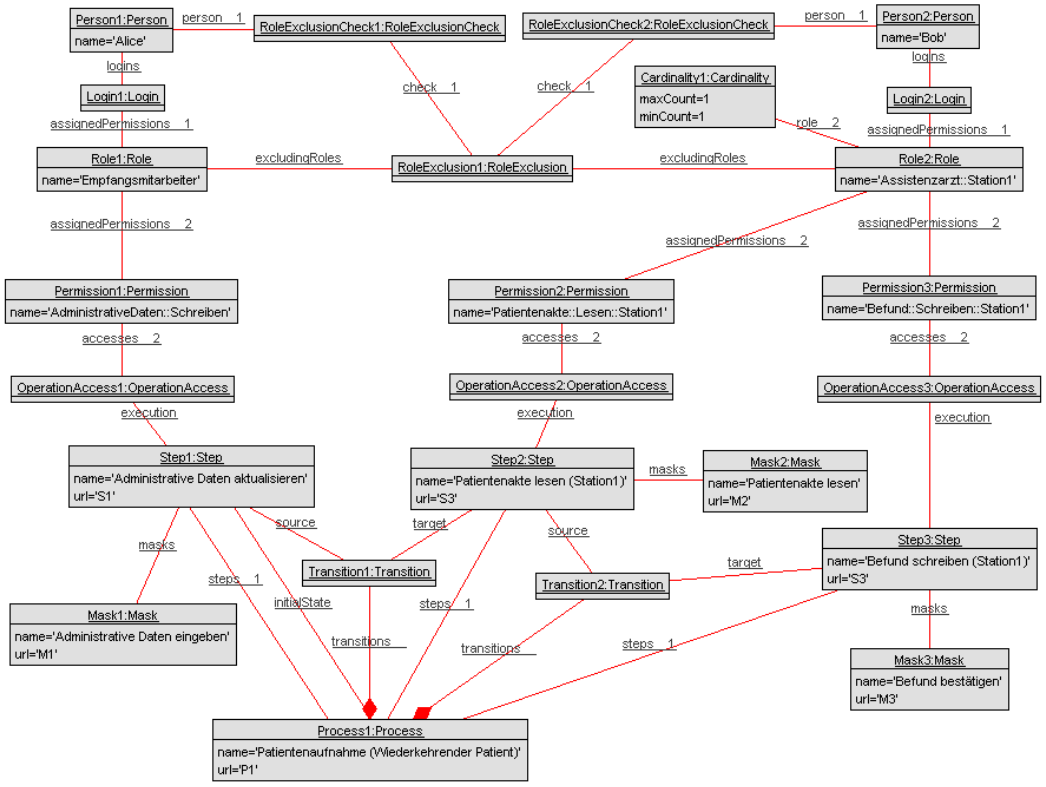


Abbildung 7.15: Grundlegendes Objektdiagramm der Patientenaufnahme zur Modellvalidierung.

Die `*.properties`-Datei ist so konfiguriert, dass genau eine `Execution`-Instanz inklusive einem Link zu einer `Subject`-Instanz und einem Link zu einer `Operation`-Instanz vom Model Validator zum gegebenen Objektdiagramm hinzugefügt werden soll.

```
# ------------------------------------------------------------ Execution
Execution_min = 1
Execution_max = 1

# executingSubject (execution__1:Execution, executingSubject:Subject)  - -
executingSubject_min = 1
executingSubject_max = 1

# executedOperation (execution__2:Execution, executedOperation:Operation)
executedOperation_min = 1
executedOperation_max = 1
```

Das gegebene Objektdiagramm, das als Grundlage für den Model Validator dient, scheint auf den ersten Blick erst einmal nicht viel komplexer zu sein als das vorangegangene Objektdiagramm (Abbildung 7.13), bedeutet intern aber einen enormen Anstieg des Berechnungsaufwands, wie das nachfolgende Listing zeigt.

```
18:03:14 INFO: Model transformation successful
18:03:14 INFO: Translation time: 126 ms
```

```
3
4 18:03:14 INFO: Searching solution with SatSolver 'MiniSat' and bitwidth 8...
5 18:03:14 INFO: SATISFIABLE
6 18:03:14 INFO: Translation time: 70 ms; Solving time: 1 ms
7 18:03:14 INFO: Searching solution with SatSolver 'MiniSat' and bitwidth 8...
8 19:47:28 INFO: SATISFIABLE
9 19:47:28 INFO: Translation time: 6253680 ms; Solving time: 1 ms
10 19:47:28 INFO: Searching solution with SatSolver 'MiniSat' and bitwidth 8...
11 23:26:42 INFO: SATISFIABLE
12 23:26:42 INFO: Translation time: 13154215 ms; Solving time: 1 ms
13 23:26:42 INFO: Searching solution with SatSolver 'MiniSat' and bitwidth 8...
14 04:48:33 INFO: SATISFIABLE
15 04:48:33 INFO: Translation time: 19310653 ms; Solving time: 1 ms
16 04:48:33 INFO: Searching solution with SatSolver 'MiniSat' and bitwidth 8...
17 12:11:50 INFO: SATISFIABLE
18 12:11:50 INFO: Translation time: 26597135 ms; Solving time: 1 ms
19 12:11:50 INFO: Searching solution with SatSolver 'MiniSat' and bitwidth 8...
```

Innerhalb von 24 Stunden konnten nur 5 der 24 möglichen Lösungen berechnet werden (bei einer `Execution`-Instanz: 4 `Subject`-Instanzen × 6 `Operation`-Instanzen = 24). Das, was dabei am meisten Zeit benötigt, ist die von Kodkod durchgeführte Übersetzung von Relational-Logic in Boolean-Logic (70 ms, 6253680 ms, 13154215 ms, 19310653 ms und 26597135 ms). Das anschließende Lösen durch den SAT-Solver MiniSat dauerte jeweils nur 1 ms. Auch das initiale und einmalige Übersetzen des Modells von USE nach Kodkod dauerte nur 126 ms. Die Zahlenfolge für die Dauern der Relational-Logic-Boolean-Logic-Übersetzung lässt zudem erkennen, dass der Zeitaufwand, um eine Lösung zu berechnen, monoton steigend ist. Da die Übersetzung nicht Multi-Threading nutzt, kann auch nur 1 CPU-Kern voll ausgenutzt werden. Der verwendete Arbeitsspeicher scheint nicht massiv während der Berechnungen anzusteigen ($\approx$ 400 MB).

Für das zweiten Anwendungsfall **„Konsil“** der Fallstudie können hingegen relativ schnell Ergebnisse mit dem in Ecore2USE integrierten Model Validator erzielt werden: Ausgangspunkt ist das Objektdiagramm aus Abbildung 6.2, allerdings ohne die `Execution`-Instanz. Die im Objektdiagramm modellierte Delegation gilt, wie die anderen Berechtigungen auch, als fix und soll nicht vom Model Validator erstellt werden. Vom Model Validator sollen 3 `Execution`-Instanzen mit jeweils den Links zum Subjekt und zur Operation sowie zur einzigen, zu erstellenden `Trace`-Instanz generiert werden. Unter Verwendung der identischen, vorab genannten Security-Policy-Constraints und Non-Security-Policy-Constraints konnte keine Policy-Verletzung gefunden werden. Der dynamische Modellvalidierungsvorgang dauert 105 s. Bei einer Erhöhung der Anzahl `Execution`-Instanzen und dazugehörigen Links auf 5, dauerte die dynamische Modellvalidierung bereits 24414 s, obwohl sich die Kardinalität der Lösungsmenge nur um 2 erhöht.
Somit konnte erfolgreich eine realistische Krankenhaussituation validiert werden, wie es Abbildung 7.16 zeigt: Dreimal wurde für die Behandlung eines Patienten von Station 1 auf dessen Patientenakte zugegriffen, wovon es zweimal Bob, der Assistenzarzt von Station 1 und einmal Frank mittels der über die Delegation

erhaltene `Assistenzarzt::Station1`-Rolle war.

Es konnte gezeigt werden, dass der Workflow der dynamischen Modellvalidierung grundsätzlich dafür geeignet ist, die Einhaltung einer Policy zu untersuchen. Bei der Anwendung am Fallbeispiel konnten keine Policy-Verletzungen festgestellt werden. Die „Darf nicht“-Anforderungen aus Abschnitt 7.2 *Betriebs- und Informationssicherheitsanforderungen* können, wie bereits oben beschrieben, aufgrund des Fehlverhaltens in Rollenhierarchien allerdings nicht als komplett erfüllt betrachtet werden. Bei der zukünftigen Arbeit mit dem Modell sind zudem auch die weiteren Auffälligkeiten aus Unterabschnitt 7.5.2 *Ergebnisse statischer Tests* zu beachten, wozu aus IT-Sicherheitssicht vor allem die Nichteinhaltung des Designprinzips „Fail-safe defaults“ zählt.
Die Anwendungsfälle der Fallstudie haben zudem verdeutlicht, welche Größenordnung das bei der dynamischen Modellvalidierung bereitzustellende Objektdiagramm bzw. das resultierende Objektdiagramm in Bezug zur zeitlichen Dauer der Modellvalidierung haben kann. Da solch eine dynamische Modellvalidierung im besten Fall aber nur einmal für ein Modell durchgeführt werden muss, können Berechnungszeiten von mehreren Tagen durchaus vertretbar sein. Soll die dynamische Modellvalidierung parallel zur Modellentwicklung als Hilfsmittel eingesetzt werden, was möglichst schnelle Resultate erfordert, kann dies über kleine, repräsentative Objektdiagramme erzielt werden.

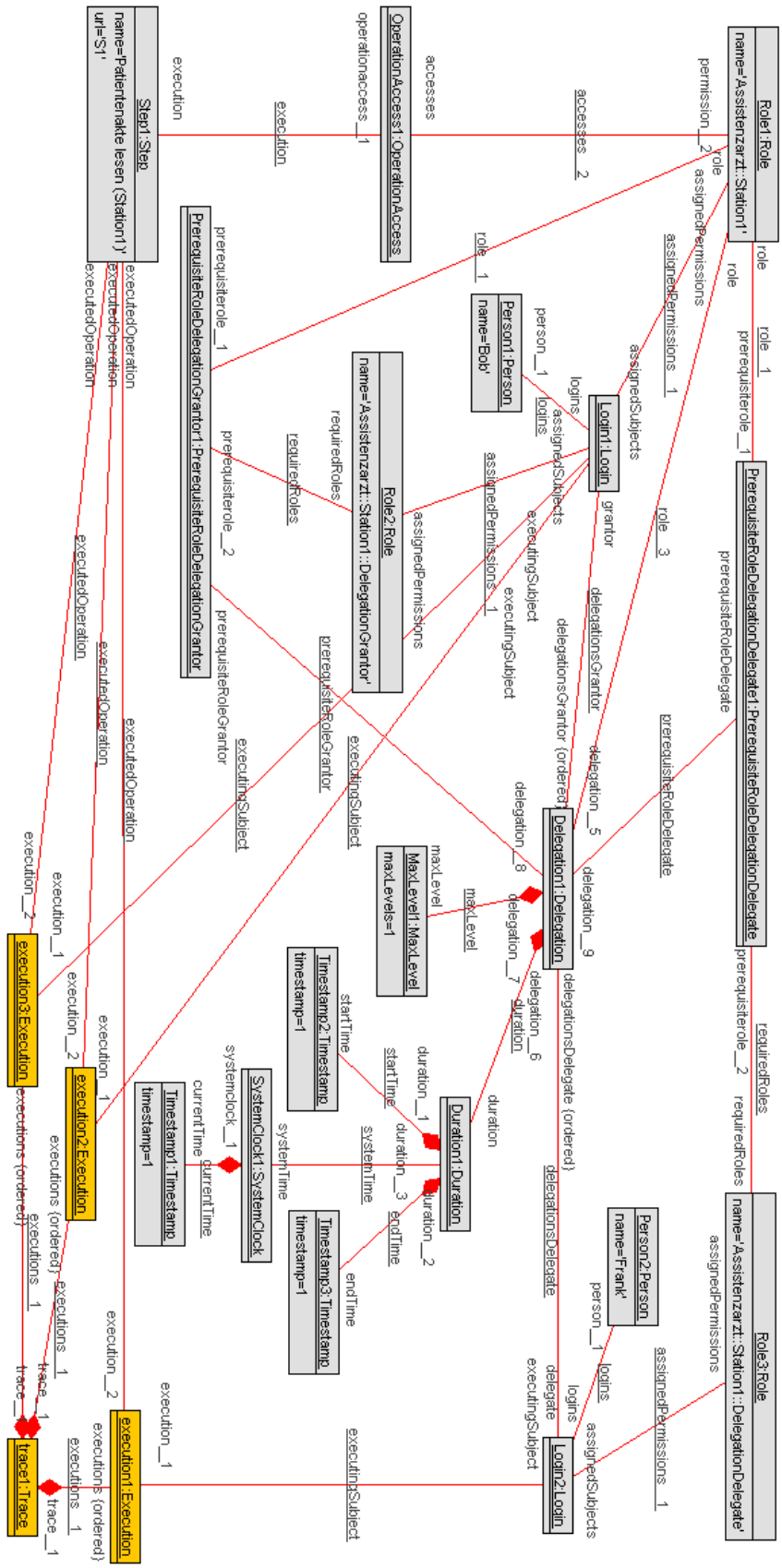


Abbildung 7.16: Generiertes Objektdiagramm zum Konsil-Anwendungsfall, das einen validen Systemzustand repräsentiert.

8 Abschluss

8.1 Zusammenfassung und Fazit

Ausgangspunkt der Arbeit war ein zur Verfügung gestelltes und mit dem Eclipse Modeling Framework erstelltes Modell, welches es ermöglicht, rollenbasierte Zugriffskontrollsysteme, Geschäftsprozesse sowie die Ausführung der Geschäftsprozesse abzubilden. Die dem Modell zugrunde liegende Modellierungssprache ist Ecore, welche eine Teilmenge des UML-Standards ist. Des Weiteren nutzt das Modell OCL-Ausdrücke, um Invarianten/Constraints sowie Operationen zu definieren. Ziel der OCL-Constraints ist es dabei, die Benutzung des Modells (Erstellung von Objektdiagrammen/Systemzuständen) eindeutig zu beschreiben – bei falscher Benutzung soll der Systemzustand von einem OCL-Interpreter aus als ungültig bewertet werden, da ein OCL-Constraint verletzt worden ist. Ob ungültige Systemzustände wirklich als ungültig erkannt werden, galt es im Rahmen der Arbeit unter Verwendung von USE zusammen mit dem Model-Validator-Plugin zu untersuchen. Da das Eclipse Modeling Framework und USE allerdings unterschiedliche Dateiformate verwenden, um ein Modell zu speichern, wurde eine Software entwickelt, die das Ecore-Modell in ein USE-Modell transformiert und dabei Unterschiede im Funktionsumfang beider Dateiformate soweit wie möglich automatisch auflöst, wozu beispielsweise gehört, die Verzeichnisstruktur zu entfernen und dadurch entstehende Namenskonflikte aufzulösen. Ergänzend zur Ecore-USE-Modell-Transformation wurde eine automatische Transformation von mit dem Eclipse Modeling Framework erstellten Objektdiagrammen zu USE-Objektdiagrammen entwickelt. Des Weiteren konnte ein Delegationskonzept entwickelt und in das bestehende Modell integriert werden. Die Anwendung des Modells inklusive Delegationskonzept wurde anschließend exemplarisch anhand der Krankenhausinformationssystem-Fallstudie gezeigt.

Es hat sich gezeigt, dass die Flexibilität und Domänenunabhängigkeit des Modells Stärke und Schwäche zugleich ist: Die Fallstudie ließ sich mit dem Modell realisieren, obwohl es nicht dafür entwickelt wurde. Und würde man über die `DataAccess`-Schnittstelle ein eigenes Datenmodell integrieren (nicht Bestandteil dieser Arbeit), könnten vermutlich Anwendungsfälle noch realistischer und detailgetreuer abgebildet werden. Auch hat die Integration des Delegationskonzepts gezeigt, dass das Modell erweiterbar ist. Allerdings scheint es an einigen Stellen im Modell schwierig zu sein, Flexibilität und restriktivere OCL-Constraints miteinander zu vereinbaren. Beispielsweise scheint es für den Anwendungsfall Krankenhausinformationssystem nicht realistisch, dass auch eine `Person`-Instanz, wenn es die nötigen Berechtigungen besitzen würde, Geschäftsprozesselemente ausführen kann, obwohl das Zugriffsverfahren auf `Login`-Instanzen basiert. Für diesen Anwendungsfall scheint es also sinnvoll zu sein, ein OCL-Constraint zu ergänzen, das vorschreibt, dass, sobald `Login`-Instanzen verwendet werden, `Person`-Instanzen nicht berechtigt sind, Geschäftsprozesselemente auszuführen (`Person`-Instanzen dienen ausschließlich zur

organisatorischen Verwaltung). Bei anderen Anwendungsfällen mag solch ein OCL-Constraint aber kontraproduktiv sein und verhindern, dass das Modell eingesetzt werden kann, weil die Realität damit nicht modelliert werden kann.

Die Arbeit hat nachgewiesen, dass der dynamische Modellvalidierungsworkflow es grundsätzlich ermöglicht, mit USE und dem Model Validator die Einhaltung der Policy eines Modells zu analysieren. Die mögliche Systemzustandsgröße ist dabei allerdings limitiert, wenn das Berechnungsergebnis in einer realistischen Zeit vorliegen soll. Somit müssen mit Expertenwissen gezielt einzelne Modellelemente validiert werden; ein realistischer Krankenhausbetrieb für die Fallstudie mit mehreren hundert Subjekten, vielen Berechtigungen und Geschäftsprozessen sowie einer noch größeren Anzahl an ausgeführten Geschäftsprozessschritten scheint sich in einer realistischen Zeit nicht automatisch validieren zu lassen.
Unabhängig von der Systemzustandsgröße kann der dynamische Modellvalidierungsvorgang nicht vollautomatisch durchgeführt werden, da modellspezifisches Expertenwissen benötigt wird: Es muss spezifiziert werden, welche OCL-Constraints zur Policy gehören, Objektdiagramme, die die Grundlage für die automatische Modellvalidierung sind, da sie die Instanzen und Links enthalten, die als fix angenommen werden sollen, müssen manuell erstellt werden, und die Anzahl an Instanzen und Links, die zusätzlich vom Model Validator erstellt werden sollen, müssen angegeben werden. Zusätzlich erfordern Einschränkungen des Model Validators, dass das Modell unter Umständen vorab einmalig modifiziert oder entsprechend der Warnmeldungen begutachtet werden muss. Außerdem bleibt es nicht aus, dass vom Model Validator generierte und somit laut OCL-Constraints gültige Systemzustände manuell darauf untersucht werden müssen, ob diese auch aus Sicht des Modellierers als gültige Systemzustände zu bewerten sind (statische Tests). Denn, wenn die OCL-Constraints an sich ein falsches Verhalten aufweisen, werden Systemzustände entstehen können, die aus Modellierersicht gültig sind, aber aus OCL-Constraint-Sicht ungültig und anders herum.

Für die Betriebssicherheit bei der Krankenhausfallstudie muss die fehlerhafte Berechtigungsermittlung bei Rollenhierarchien behoben werden. Der einzige kritische Punkt aus IT-Sicherheitssicht ist, dass das Designkonzept „Fail-safe defaults“ nicht eingehalten wurde. Auch wenn ansonsten keine IT-Sicherheitslücken bei den statischen oder dynamischen Tests gefunden wurden, kann daraus nicht geschlussfolgert werden, dass, wenn das Modell als Grundlage für eine Anwendungssoftware eingesetzt wird, diese frei von IT-Sicherheitslücken ist: Die einzelnen Geschäftsprozessschritte müssen für eine konkrete Anwendung mit Quellcode gefüllt werden, der wiederum IT-Sicherheitslücken enthalten kann. Und auch das Umfeld, in dem das Model letztendlich eingesetzt wird, kann die IT-Sicherheit gefährden, wenn beispielsweise die Authentifizierung der Subjekte fehlerhaft ist.

8.2 Ausblick

Basierend auf den Ergebnissen dieser Arbeit ist die Weiterarbeit daran in mehreren Themengebieten denkbar. Soll mit dem Modell weitergearbeitet werden, sollten zunächst die im Modell gefundenen Auffälligkeiten genau begutachtet und entsprechend im Modell korrigiert werden.

Zwar wurde gezeigt, dass die Ecore-zu-USE-Modelltransformation möglich ist und die dafür entwickelte Software dabei auch so allgemein wie möglich gehalten, sodass auch theoretisch andere Modelle damit transformiert werden können, allerdings wurde wenig mit anderen Modellen getestet oder automatisierte Unit-Tests geschrieben und somit vermutlich nicht alle möglichen Sonderfälle behandelt. Der Status der Transformationssoftware ist daher eher als Proof-of-Concept und nicht nicht als fertiges Produkt anzusehen. Ein konkreter Restpunkt ist dabei folgender: Müssen im Rahmen der Modelltransformation Umbenennungen von z.B. Beziehungen vorgenommen werden, weil die Namen nicht eindeutig sind, ist die Umbenennung noch nicht vollständig für OCL-Constraints umgesetzt. Der Grund dafür ist, dass dies nicht immer über reine Textersetzung (z.B. mit regulären Ausdrücken) möglich ist, sondern es erfordert teilweise das Parsen des OCL-Constraints, um zu ermitteln auf welchem Datentyp eine Beziehung aufgerufen wird. Das folgende Listing zeigt ein OCL-Constraint-Beispiel des RBAC-Modells, bei dem `m.steps` in `m.steps_2` fürs USE-Modell umbenannt werden muss – aktuell muss dies nachträglich manuell per Texteditor getan werden.

```
1 -- processConformance, context: process::Process
2 steps->forAll(s|(s.masks->collect(m|m.getIncludedMasksIncludingSelf())->
      collect(m|m.successors)->collect(m|m.steps->selectByKind(IStep))->asSet()
      - transitions->select(t|t.source = s)->collect(t|t.target)->asSet())->
      isEmpty())
```

Bezüglich der benötigten Zeit, die die dynamische Modellvalidierung (Model Validator) benötigt, um bei größeren Objektdiagrammen zu einem Ergebnis zu kommen, könnten weitere, detailliertere Untersuchungen unternommen werden; diese könnten Aufschluss darüber geben, wie genau die Berechnungskomplexität entsteht und ob sich mit dem gewonnen Wissen, das Modell so optimieren lässt, dass die Berechnungskomplexität bei gleichem Funktions- und IT-Sicherheitsumfang geringer wird.

Außerdem könnte ein nächster Schritt sein, das Modell und dessen Konzepte konsequent als Grundlage für eine neue Anwendungssoftware zu nutzen und dabei zu analysieren, ob die Domänenunabhängigkeit weiterhin standhält und wie das Umfeld, in dem das Modell eingesetzt wird, geschaffen sein muss, um aus Sicht der IT-Sicherheit auch sicher zu sein.

Literaturverzeichnis

[1] *Acceleo*. `http://wiki.eclipse.org/Acceleo`. Letzter Zugriff: 17.09.2017.

[2] *Ecore Javadoc: Package org.eclipse.emf.ecore*. `http://download.eclipse.org/modeling/emf/emf/javadoc/2.10.0/org/eclipse/emf/ecore/package-summary.html`. Letzter Zugriff: 17.09.2017.

[3] *IEEE Standard for System and Software Verification and Validation*. IEEE Std 1012-2012 (Revision of IEEE Std 1012-2004), S. 1–223, May 2012.

[4] Barka, E. und R. Sandhu: *Framework for role-based delegation models*. In: *Computer Security Applications, 2000. ACSAC '00. 16th Annual Conference*, S. 168–176, Dec 2000.

[5] Berger, B.: *Ergebnisbericht*. Techn. Ber., Universität Bremen, Version 0.9.

[6] Brenner, M.: *Praxisbuch ISO-IEC 27001: Management der Informationssicherheit und Vorbereitung auf die Zertifizierung*. Hanser, München, 2011.

[7] Büttner, F. und M. Gogolla: *Modular Embedding of the Object Constraint Language into a Programming Language*, S. 124–139. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[8] Cabot, J. und M. Gogolla: *Object Constraint Language (OCL): A Definitive Guide*. In: Bernardo, M., V. Cortellessa und A. Pierantonio (Hrsg.): *Formal Methods for Model-Driven Engineering: 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012, Bertinoro, Italy, June 18-23, 2012. Advanced Lectures*, S. 58–90. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[9] Database Systems Group Bremen University: *USE. A UML based Specification Environment*. `http://www.db.informatik.uni-bremen.de/projects/use/use-documentation.pdf`, 2007.

[10] Dijkstra, E. W.: *The Humble Programmer*. Commun. ACM, 15(10):859–866, 1972.

[11] Eckert, C.: *IT-Sicherheit: Konzepte - Verfahren - Protokolle*. Informatik 10-2012. Oldenbourg, München, 7. Aufl., 2012.

[12] Gerdes, S.: *Rollenbasiertes Sicherheitskonzept für Krankenhäuser unter Berücksichtigung der aktuellen Entwicklungen in der Gesundheitstelematik*. Diplomarbeit, Universität Bremen, 2007.

[13] Gogolla, M., F. Büttner und M. Richters: *USE: A UML-based specification environment for validating UML and OCL*. Science of Computer Programming, 69(1–3):27–34, 2007. Special issue on Experimental Software and Toolkits.

[14] Gogolla, M. und F. Hilken: *Model Validation and Verification Options in a Contemporary UML and OCL Analysis Tool*. In: Oberweis, A. und R. Reussner (Hrsg.): *Proc. Modellierung (MODELLIERUNG'2016)*, S. 203–218. GI, LNI 254, 2016.

[15] Hopcroft, J. E., R. Motwani und J. D. Ullman: *Introduction to Automata Theory, Languages, and Computation*. Pearson Addison-Wesley, 3. Aufl., 2006. Pearson International Edition.

[16] Kersten, H. und G. Klett: *Der IT Security Manager: aktuelles Praxiswissen für IT Security Manager und IT-Sicherheitsbeauftragte in Unternehmen und Behörden*. Edition kes. Springer Vieweg, Wiesbaden, 4. Aufl., 2015.

[17] Kuhlmann, M., L. Hamann und M. Gogolla: *Extensive Validation of OCL Models by Integrating SAT Solving into USE*. In: Bishop, J. und A. Vallecillo (Hrsg.): *Objects, Models, Components, Patterns: 49th International Conference, TOOLS 2011, Zurich, Switzerland, June 28-30, 2011. Proceedings*, S. 290–306. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[18] Lutz, R. R.: *Analyzing software requirements errors in safety-critical, embedded systems*. In: *[1993] Proceedings of the IEEE International Symposium on Requirements Engineering*, S. 126–133, Jan 1993.

[19] Mili, A. und F. Tchier: *Software Testing: Concepts and Operations*. Quantitative Software Engineering Series. Wiley, 2015.

[20] Müller, K.-R.: *IT-Sicherheit mit System: Integratives IT-Sicherheits-, Kontinuitäts- und Risikomanagement – Sichere Anwendungen – Standards und Practices*. Springer Vieweg, 5., neu bearbeitete und erweiterte Aufl., 2014.

[21] Myers, G. J., T. Badgett und C. Sandler: *The Art of Software Testing*. Wiley, 3. Aufl., 2012.

[22] Object Management Group: *MOF Model to Text Transformation Language, v1.0*. Techn. Ber., 2008.

[23] Object Management Group: *Object Constraint Language - Version 2.4*. Techn. Ber., 2014.

[24] Object Management Group: *OMG Unified Modeling Language (OMG UML) - Version 2.5*. Techn. Ber., 2015.

[25] O'Connor, A. C. und R. J. Loomis: *2010 Economic Analysis of Role-Based Access Control*. Techn. Ber., National Institute of Standards and Technology, 2010.

[26] Pilone, D. und N. Pitman: *UML 2.0 - in a nutshell: a desktop quick reference*. O'Reilly, 2005.

[27] Saltzer, J. H. und M. F. Kaashoek: *RES.6-004 Principles of Computer System Design: An Introduction*. Massachusetts Institute of Technology: MIT OpenCourseWare, https://ocw.mit.edu, 2009. `https://ocw.mit.edu/resources/res-6-004-principles-of-computer-system-design-an-introduction-spring-2009/online-textbook/`.

[28] Saltzer, J. H. und M. D. Schroeder: *The Protection of Information in Computer Systems*. Proceedings of the IEEE, 63(9):1278–1308, 1975.

[29] Salvanos, A.: *UML 2.5: Das umfassende Handbuch*. Galileo Computing, Bonn, 5., aktualisierte und erweiterte Aufl., 2015.

[30] Sandhu, R. S., E. J. Coyne, H. L. Feinstein und C. E. Youman: *Role-Based Access Control Models*. Computer, 29(2):38–47, Feb. 1996.

[31] Smith, R. E.: *A Contemporary Look at Saltzer and Schroeder's 1975 Design Principles*. IEEE Security & Privacy, 10(6):20–25, 2012.

[32] Sommerville, I.: *Software Engineering*. Pearson, München, 9. aktualisierte Aufl., 2012.

[33] Spillner, A. und T. Linz: *Basiswissen Softwaretest. Aus- und Weiterbildung zum Certified Tester; Foundation Level nach ISTQB-Standard*. Safari Tech Books Online. dpunkt Verl., Heidelberg, 5. überarb. und aktualisierte Aufl., 2012.

[34] Statistisches Bundesamt: *Unternehmen und Arbeitsstätten. Nutzung von Informations- und Kommunikationstechnologien in Unternehmen*, 2016. Artikelnummer: 5529102167004.

[35] Steinberg, D., F. Budinsky, M. Paternostro und E. Merks: *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, 2. Aufl., 2009.

[36] Torlak, E. und D. Jackson: *Kodkod: A Relational Model Finder*. In: Grumberg, O. und M. Huth (Hrsg.): *Tools and Algorithms for the Construction and Analysis of Systems: 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007. Proceedings*, S. 632–647. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[37] Wainer, J. und A. Kumar: *A Fine-grained, Controllable, User-to-user Delegation Method in RBAC*. In: *Proceedings of the Tenth ACM Symposium on Access Control Models and Technologies*, SACMAT '05, S. 59–66, New York, NY, USA, 2005. ACM.

A Delegationsmodell (OCLinEcore-Darstellung)

Listing A.1: Delegationmodell (OCLinEcore-Darstellung)

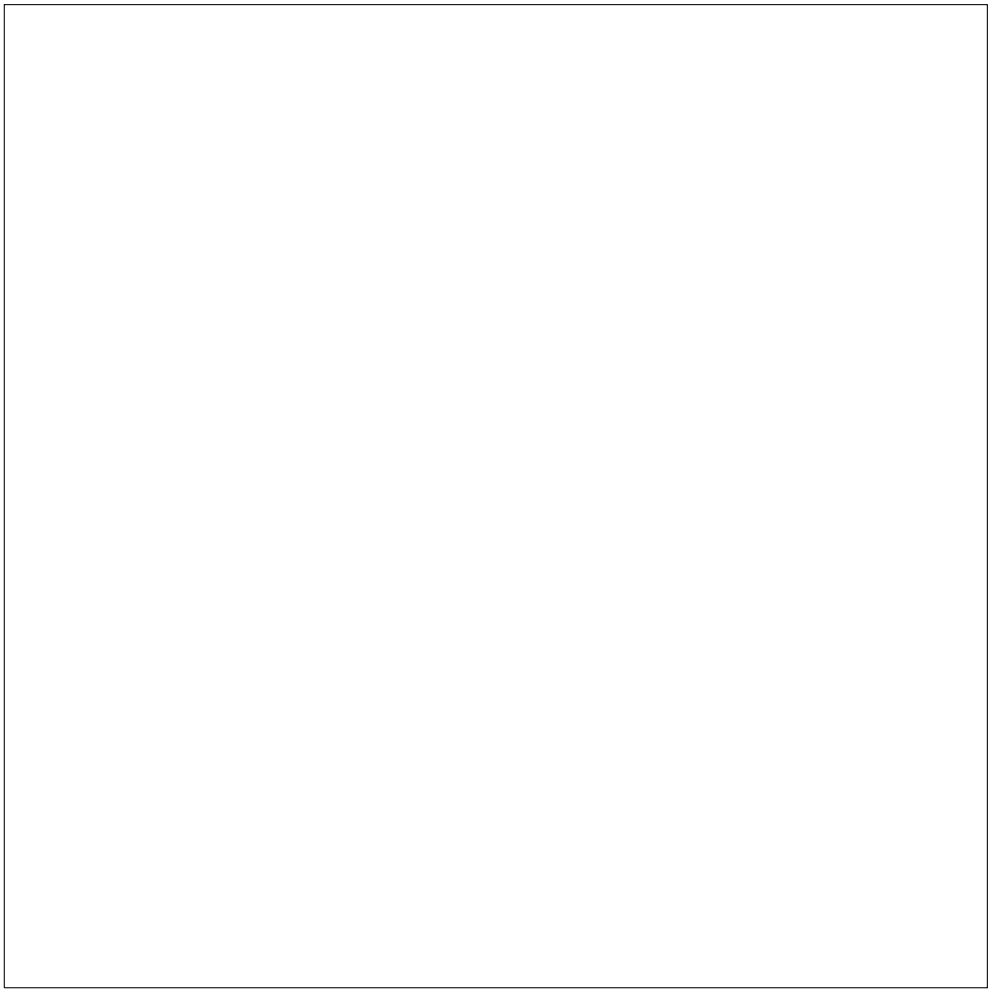
```
1     package delegation : delegation = 'http://www.dfki.de/model/highlevel/
          delegation'
2 {
3 class Delegation
4 {
5 annotation _'Security-Policy-Constraints'
6 (
7 constraints = 'maxLevelCheck prerequisiteRoleGrantorCheck
      prerequisiteRoleDelegateCheck'
8 );
9 property forwardings#backtrace : Delegation[*|1] { ordered };
10 property grantor#delegationsGrantor : highlevel::Subject[1];
11 property delegate#delegationsDelegate : highlevel::Subject[1];
12 property backtrace#forwardings : Delegation[?];
13 property role : highlevel::rbac::Role[1];
14 property duration : Duration[?] { composes };
15 property maxLevel : MaxLevel[?] { composes };
16 property prerequisiteRoleGrantor : PrerequisiteRoleDelegationGrantor[1];
17 property prerequisiteRoleDelegate : PrerequisiteRoleDelegationDelegate[1];
18 invariant
19 maxLevelCheck: -- Supervise that the number of levels in the delegation tree,
      i.e. number of how often a delegation was further delegated, doesn't
      exceed the limit.
20 -- Only the root of the delegation tree should have an optional MaxLevel
      instance. If other instances in the delegation tree have a MaxLevel
      instance, it will be ignored.
21 -- The algorithm checks the path length to the root of the delegation tree iff
       the current Delegation instance is a leaf of the delegation tree (i.e.
      the delegation was NOT further delegated) and compares this number with
      the number from the MaxLevel instance of the root.
22
23 -- Check, if this instance is leaf in the delegation tree
24 if self.forwardings->isEmpty() then
25 let traceNodes : Set(Delegation) = Set{self}->closure(backtrace)->union(Set{
      self})->selectByType(delegation::Delegation) in
26 let roots : Set(Delegation) = traceNodes->select(n | n.backtrace->isEmpty())
      in
27 if roots->size() = 1 then
28 let root : Delegation = roots->any(true) in
29 if root.maxLevel->size() = 1 then
30 traceNodes->size() <= root.maxLevel.maxLevels
31 else
32 if root.maxLevel->size() = 0 then
33 true
34 else
35 false
36 endif
37 endif
38 else
39 false
40 endif
41 else
42 true
43 endif;
44 invariant matchingPrerequisiteRoleGrantorCheck: self.role.name = self.
      prerequisiteRoleGrantor.role.name;
```

```
45 invariant
46 prerequisiteRoleGrantorCheck: self.grantor.getAllAssignedPermissions()->
       selectByType(highlevel::rbac::Role)->union(self.grantor.
       getAllAssignedPermissions()->selectByType(highlevel::rbac::Role)->closure(
       children))->includesAll(self.prerequisiteRoleGrantor.requiredRoles);
47 invariant matchingPrerequisiteRoleDelegateCheck: self.role.name = self.
       prerequisiteRoleDelegate.role.name;
48 invariant
49 prerequisiteRoleDelegateCheck: self.delegate.getAllAssignedPermissions()->
       selectByType(highlevel::rbac::Role)->union(self.delegate.
       getAllAssignedPermissions()->selectByType(highlevel::rbac::Role)->closure(
       children))->includesAll(self.prerequisiteRoleDelegate.requiredRoles);
50 invariant
51 differentSubjects: -- Check that the grantor and delegate are not the same
       object.
52 self.grantor<>self.delegate;
53 invariant
54 sameSubjectType: -- Check that the grantor and delegate are an instance of the
        same type (both Person or both Login)
55 (self.grantor.oclIsTypeOf(permissions::Person) and self.delegate.oclIsTypeOf(
       permissions::Person)) or (self.grantor.oclIsTypeOf(permissions::Login) and
        self.delegate.oclIsTypeOf(permissions::Login));
56 invariant
57 noLoginsForPersons: -- Check that the grantor and delegate have no logins, if
       both are Person instances.
58 if self.grantor.oclIsTypeOf(permissions::Person) and self.delegate.oclIsTypeOf
       (permissions::Person) then
59 self.grantor.oclAsType(permissions::Person).logins->isEmpty() and self.
       delegate.oclAsType(permissions::Person).logins->isEmpty()
60 else
61 true
62 endif;
63 invariant
64 differentPersonWithLoginSubject: if self.grantor.oclIsTypeOf(permissions::
       Login) and self.delegate.oclIsTypeOf(permissions::Login) then
65 let persons : Set(permissions::Person) = permissions::Person.allInstances() in
66 let grantorPersons : Set(permissions::Person) = persons->select(person |
       person.logins->includes(self.grantor.oclAsType(permissions::Login)))->
       asSet() in
67 let delegatePersons : Set(permissions::Person) = persons->select(person |
       person.logins->includes(self.delegate.oclAsType(permissions::Login)))->
       asSet() in
68 if grantorPersons->isEmpty() and delegatePersons->isEmpty() then
69 -- No Person object used on both sides (grantor and delegate) => valid system
       state
70 true
71 else
72 if grantorPersons->isEmpty() xor delegatePersons->isEmpty() then
73 -- Person objects used on one side => invalid system state
74 false
75 else
76 -- Both, grantorPersons and delegatePersons, are not empty
77 -- It is not allowed that one person delegates permission from one of its
       logins to one of its other logins (even if also other persons using the
       same login)
78 grantorPersons->intersection(delegatePersons)->isEmpty()
79 endif
80 endif
81 else
82 -- Constraint is only relevant if grantor and delegate are Login objects
83 true
```

```
84 endif;
85 invariant
86 noPrerequisiteRoleDelegationGrantorDuplicates: -- Check, if no multiple
       PrerequisiteRoleDelegationGrantor instances exist that have the same Role
       instances as 'role'
87
88 let allPrerequisiteRoleDelegationGrantors : Set(delegation::
       PrerequisiteRoleDelegationGrantor) = delegation::
       PrerequisiteRoleDelegationGrantor.allInstances()->asSet() in
89 -- Check, if the number of PrerequisiteRoleDelegationGrantor objects is the
       same as number of different roles (Set of role) used inside the
       PrerequisiteRoleDelegationGrantor objects; if it's not the case, a role
       was used multiple times (invalid system state)
90 allPrerequisiteRoleDelegationGrantors->size() =
       allPrerequisiteRoleDelegationGrantors.role->asSet()->size();
91 invariant
92 noPrerequisiteRoleDelegationDelegateDuplicate: -- Check, if no multiple
       PrerequisiteRoleDelegationDelegate instances exist that have the same Role
        instances as 'role'
93
94 let allPrerequisiteRoleDelegationDelegates : Set(delegation::
       PrerequisiteRoleDelegationDelegate) = delegation::
       PrerequisiteRoleDelegationDelegate.allInstances()->asSet() in
95 -- Check, if the number of PrerequisiteRoleDelegationDelegate objects is the
       same as number of different roles (Set of role) used inside the
       PrerequisiteRoleDelegationDelegate objects; if it's not the case, a role
       was used multiple times (invalid system state)
96 allPrerequisiteRoleDelegationDelegates->size() =
       allPrerequisiteRoleDelegationDelegates.role->asSet()->size();
97 invariant differentPersons;
98 }
99 class MaxLevel extends highlevel::Constraint
100 {
101 attribute maxLevels : ecore::EInt[1];
102 }
103 class Duration extends highlevel::Constraint
104 {
105 property startTime : highlevel::Timestamp[1] { composes };
106 property endTime : highlevel::Timestamp[1] { composes };
107 property systemTime : highlevel::SystemClock[1];
108 invariant
109 notExpired: -- Check, if the current system time is not expired.
110 self.systemTime.currentTime.timestamp <= self.endTime.timestamp;
111 invariant NotExpired;
112 }
113 class MultipleDelegationCheck
114 {
115 annotation _'Security-Policy-Constraints'
116 (
117 constraints = 'maxMultipleDelegation'
118 );
119 attribute maxMultipleDelegation : ecore::EInt[1] = '1';
120 property subject : highlevel::Subject[1];
121 property role : highlevel::rbac::Role[1];
122 invariant
123 maxMultipleDelegation: -- Check, if the allowed number of delegation is NOT
       infinity
124 if self.maxMultipleDelegation <> -1 then
125 -- Check, if the maximum number of delegations for the subject-role-
       combination of this instance is NOT exceeded
```

```
126 self.subject.delegationsGrantor->select(role.name=self.role.name)->size() <=
        self.maxMultipleDelegation
127 else
128 true
129 endif;
130 }
131 class PrerequisiteRoleDelegationGrantor extends highlevel::rbac::
        PrerequisiteRole;
132 class PrerequisiteRoleDelegationDelegate extends highlevel::rbac::
        PrerequisiteRole;
133 }
```

B CD



CD-Inhalt

- Masterarbeit (dieses Dokument in digitaler Form)
- Quellcode der entwickelten Software Ecore2USE
- Originales Ecore-Modell, Ecore-Modell inklusive des Delegationskonzepts, transformiertes USE-Modell (ohne nachträgliche Modifikation; fehlerhaft laut USE) und transformiertes USE-Modell mit nachträglichen Korrekturen und Anpassungen für die Modellvalidierung.
- Beispiele für die dynamische Modellvalidierung (`*.soil`- und `*.properties`-Dateien)