ulm university universität ulm

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften und Informatik

Authorization Constraints in Workflow-Management-Systemen

Diplomarbeit an der Universität Ulm

Vorgelegt von:
Florian Kelbert
florian.kelbert@uni-ulm.de

Gutachter:
1. Prof. Dr. Manfred Reichert
2. Prof. Dr. Peter Dadam

Juni 2010

Danksagung

Während meines Studiums und der Erstellung meiner Diplomarbeit wurde ich von vielen Menschen begleitet und unterstützt. Mein Dank gilt daher:

Herrn Prof. Dr. Manfred Reichert und Herrn Prof. Dr. Peter Dadam vom Institut für Datenbanken und Informationssysteme der Universität Ulm für die Überlassung des interessanten Themas und die Begutachtung dieser Arbeit.

Michael Predeschly für die intensive Betreuung während der zurückliegenden sechs Monate mit vielen hilfreichen Diskussionen und Gesprächen. Danke für Anregungen, konstruktive Kritik und das allseits offene Ohr.

David Knuplesch für die hilfreichen Diskussionen zu theoretischen Fragestellungen der Informatik während der Erstellung dieser Arbeit.

Allen Kommilitonen für das gemeinsame Meistern der letzten Jahre. Ein Studium im Alleingang wäre wohl kaum möglich gewesen.

Allen Freunden, Bekannten und Verwandten, die mich während der letzten Monate und Jahre geprägt, unterstützt und motiviert haben. Danke für die tolle Freizeitgestaltung und den benötigten Ausgleich.

Mein abschließender und besonderer Dank gilt meinen Eltern, die mir dieses Studium ermöglicht haben und ohne die ein solcher Bildungsweg nicht möglich gewesen wäre. Danke für Unterstützung, Motivation, Verständnis, Geduld und alles andere.

Inhaltsangabe

Workflow-Management-Systeme (WfMS) werden zunehmend zur Modellierung und Ausführung umfangreicher Geschäftsprozesse eingesetzt. Da die Ausführung der Prozesse durch viele unterschiedliche Benutzer erfolgt, entsteht die Notwendigkeit komplexer Sicherheitsanforderungen, auch Authorization Constraints genannt.

In heutigen WfMS werden an Benutzer Berechtigungen vergeben, die zur Ausführung einzelner Aktivitäten eines Prozesses berechtigen. Authorization Constraints ermöglichen dagegen den Einsatz komplexerer Berechtigungen. Bekannte Beispiele sind Separation of Duty (SoD) und Binding of Duty (BoD). Während SoD die Ausführung zweier oder mehrerer Aktivitäten durch unterschiedliche Benutzer fordert, müssen diese bei BoD durch denselben Benutzer ausgeführt werden.

Diese Arbeit beschäftigt sich mit dem Einsatz von Authorization Constraints in WfMS und geht dabei insbesondere auf deren Durchsetzung und Validierung ein. Ziel der Durchsetzung ist zu gewährleisten, dass alle Authorization Constraints zur Modellierzeit und zur Laufzeit des Prozesses eingehalten werden. Die Validierung hat zum Ziel, Widersprüche zwischen Authorization Constraints zu finden. Außerdem beschäftigt sich die Validierung mit der Frage, ob die Ausführung aller Aktivitäten eines Prozesses, bei gleichzeitiger Durchsetzung aller Authorization Constraints, möglich ist.

Zunächst werden hierzu die in WfMS relevanten Authorization Constraints vorgestellt und kategorisiert. Um die Durchsetzung und Validierung der unterschiedlichen Authorization Constraints auf eine einheitliche Weise zu ermöglichen, wird ein Modell namens MΘnK eingeführt. MΘnK ermöglicht die einheitliche Modellierung der unterschiedlichen Authorization Constraints. Die Durchsetzung und Validierung wird dann auf Basis sogenannter MΘnK-Constraints vorgenommen. Dabei ist zu beachten, dass unterschiedliche Arten von Authorization Constraints existieren, deren Durchsetzung und Validierung teilweise zur Modellierzeit, teilweise aber auch erst zur Laufzeit des Prozesses möglich ist. Daher werden in dieser Arbeit Vorgehensweisen vorgestellt, um MΘnK-Constraints sowohl zur Modellierzeit, als auch zur Laufzeit eines Prozesses durchsetzen und validieren zu können.

Exemplarisch wurden die Modellierung der Authorization Constraints durch MΘnK-Constraints, sowie die zur Modellierzeit durchführbaren Teile der Durchsetzung und Validierung implementiert.

Inhaltsverzeichnis

Teil I

Einleitung und Grundlagen für Authorization Constraints

1 Einleitung

1.1 Motivation

Der Einsatz von Workflow-Management-Systemen zur Modellierung und Ausführung von Geschäftsprozessen hat in den letzten Jahren einen wichtigen Stellenwert in vielen Organisationen, wie z.B. Unternehmen oder Behörden, eingenommen. Neben der Modellierung von Prozessen ermöglicht ein Workflow-Management-System auch die Erzeugung und Ausführung von Instanzen der modellierten Prozesse. Dabei werden die elementaren Aktivitäten des Prozesses durch unterschiedliche Benutzer ausgeführt. Das Ziel ist hierbei eine möglichst optimale und konfliktfreie Ausführung der modellierten Geschäftsprozesse.

Dabei sollen Workflow-Management-Systeme eine sichere Ausführung der Prozesse gewährleisten. Die Kompromittierung eines Prozesses durch, beabsichtigte oder unbeabsichtigte, Fehler soll ausgeschlossen werden. Heutige Workflow-Management-Systeme unterstützen eine sichere Ausführung der Prozesse nur unzureichend. So ist die Vergabe einfacher Berechtigungen zur Ausführung der Aktivitäten möglich, jedoch fehlt die Möglichkeit zur Umsetzung komplexerer Sicherheitsanforderungen.

Gerade in großen Organisationen mit umfangreichen Prozessen und vielen Mitarbeitern besteht der Wunsch zur Umsetzung weitergehender Sicherheitsanforderungen. Derartige bekannte Sicherheitsanforderungen sind beispielsweise, dass bestimmte Aktivitäten von unterschiedlichen Benutzern (Separation of Duty) oder von demselben Benutzer (Binding of Duty) ausgeführt werden müssen. Ein weiteres Beispiel für eine komplexe Sicherheitsanforderung ist, dass kritische Berechtigungen nur an eine bestimmte Anzahl von Mitarbeitern vergeben werden dürfen. Viele Berechtigungen sind dabei von Ereignissen und Kontextinformationen abhängig, die erst während der Ausführung einer Prozessinstanz stattfinden bzw. vorliegen.

Derartige Sicherheitsanforderungen sind mit herkömmlichen Verfahren zur Berechtigungsvergabe nicht umsetzbar, so dass der Einsatz zusätzlicher Konzepte notwendig wird. Ein

solches Konzept zur Umsetzung komplexer Sicherheitsanforderungen sind Authorization Constraints.

Mit dem Einsatz von Authorization Constraints ergeben sich jedoch weitere Fragestellungen. So stellen Authorization Constraints, verglichen mit den herkömmlichen Verfahren zur Berechtigungsvergabe, komplexe Gebilde dar, deren Einhaltung durch das Workflow-Management-System gewährleistet werden muss. Auf Grund dieser Komplexität ist es notwendig, dass dem Constraintmodellierer eine einfache Handhabung der Authorization Constraints ermöglicht wird.

Trotzdem kann nicht ausgeschlossen werden, dass widersprüchliche Authorization Constraints formuliert werden, oder dass diese eine erfolgreiche Ausführung des Prozesses unmöglich machen. Aus diesen Gründen ist es notwendig, die formulierten Authorization Constraints verschiedenen Prüfungen zu unterziehen. Das Ziel dieser Überprüfungen ist die Feststellung der Widerspruchsfreiheit der Authorization Constraints und damit die Ermöglichung einer konfliktfreien Ausführung des Prozesses.

Diese Arbeit beschäftigt sich mit dem Einsatz und der Umsetzung von Authorization Constraints in Workflow-Management-Systemen. Dabei wird insbesondere auf die Formulierung, Validierung und Durchsetzung der Authorization Constraints eingegangen.

1.2 Ziel dieser Arbeit

Ziel dieser Arbeit ist die Umsetzung komplexer Sicherheitsanforderungen in Workflow-Management-Systemen in Form von Authorization Constraints.

Hierfür sind zunächst die für Workflow-Management-Systeme relevanten Authorization Constraints zu identifizieren und gemäß ihrer Gemeinsamkeiten und Unterschiede zu kategorisieren.

Ziel ist anschließend der Entwurf eines Konzeptes zur Umsetzung der relevanten Authorization Constraints in Workflow-Management-Systemen. Hierbei muss zunächst die Einhaltung der in einem Prozess formulierten Authorization Constraints gewährleistet werden. Um eine konfliktfreie Ausführung eines Prozesses zu ermöglichen, ist außerdem eine Validierung der Authorization Constraints notwendig. Ziel dieser Validierung ist es, die Entstehung von Konflikten zu verhindern oder diese zumindest frühzeitig zu erkennen. Sowohl

die Durchsetzung als auch die Validierung sind dabei, je nach Art der Authorization Constraints, zur Modellierzeit und zur Laufzeit des Prozesses vorzunehmen. Da Authorization Constraints komplexe Gebilde darstellen, sollen darüber hinaus die Constraintmodellierer bei deren Einsatz unterstützt werden.

Schließlich sollen die erarbeiteten Konzepte prototypisch implementiert und an Beispielszenarien demonstriert und getestet werden.

1.3 Wissenschaftlicher Beitrag dieser Arbeit

Im Hinblick auf die Durchsetzung, Validierung und einfache Handhabung von Authorization Constraints, wurde im Rahmen dieser Arbeit ein Modell namens MΘnK zur Modellierung von Authorization Constraints erarbeitet. Dabei setzt MΘnK eine rollenbasierte Berechtigungsvergabe voraus. Besonderen Wert wurde darauf gelegt, dass eine einheitliche Modellierung vieler unterschiedlicher Authorization Constraints möglich ist. MΘnK ermöglicht sowohl die statische, als auch die dynamische Einschränkung von Berechtigungen. Somit lassen sich mit MΘnK alle in Workflow-Management-Systemen relevanten Authorization Constraints einheitlich auf sogenannte MΘnK-Constraints abbilden.

Für die in Workflow-Management-Systemen wichtigsten Authorization Constraints wurden außerdem Templates für MΘnK erarbeitet. Durch diese Templates wird die Modellierung der Authorization Constraints für den Constraintmodellierer deutlich vereinfacht. Die Templates stellen Regeln dar, nach denen bekannte Authorization Constraints auf MΘnK-Constraints abgebildet werden. Der Einsatz solcher Template-unterstützter Authorization Constraints gestaltet sich somit für den Constraintmodellierer besonders einfach, da keine manuelle Abbildung des gewünschten Authorization Constraints auf MΘnK-Constraints vorgenommen werden muss.

Durch die Abbildung der Authorization Constraints auf MΘnK-Constraints, wird deren Durchsetzung und Validierung auf eine einheitliche Weise möglich. Es wurde ein Konzept erarbeitet, das die zuverlässige Durchsetzung und Einhaltung der MΘnK-Constraints gewährleistet. Ein besonderes Augenmerk liegt dabei auf der Unterscheidung zwischen zur Modellierzeit und zur Laufzeit durchsetzbaren MΘnK-Constraints. Um Konflikte durch Authorization Constraints zu entdecken, wurden außerdem Betrachtungen zur Validierung der MΘnK-Constraints angestellt. Dabei ist die Frage, ob eine Menge von MΘnK-Constraints

widerspruchsfrei ist, NP-vollständig. Aus diesem Grund wurden einfache Prüfungen betrachtet, mit denen zur Modellierzeit bereits ein großer Teil der Konflikte in einer Menge von MΘnK-Constraints effizient gefunden werden kann. Weitere Betrachtungen zur Validierung der MΘnK-Constraints zur Laufzeit haben zum Ziel, potentielle Konflikte möglichst frühzeitig zu erkennen und so eine konfliktfreie Ausführung des Prozesses zu ermöglichen.

1.4 Gliederung dieser Arbeit

Die vorliegende Arbeit gliedert sich in zwei Hauptteile, Teil I und Teil II.

Teil I: Einleitung und Grundlagen für Authorization Constraints

Teil I führt in Kapitel 2 grundlegende Begrifflichkeiten zu Workflow-Management-Systemen ein. Dabei wird zunächst in Kapitel 2.1 auf die Modellierung und Ausführung von Prozessen, sowie auf Sicherheitsaspekte und -anforderungen in diesen Systemen eingegangen. Kapitel 2.2 erläutert anschließend Role-Based Access Control als Grundlage für die Vergabe von Berechtigungen an Benutzer. Zum Abschluss von Teil I wird in Kapitel 3 zunächst Allgemeines zu Authorization Constraints erläutert (Kapitel 3.1), bevor schließlich verschiedene Authorization Constraints vorgestellt (Kapitel 3.3 – 3.6) und kategorisiert (Kapitel 3.7) werden.

Teil II: Formulierung, Durchsetzung und Validierung von Authorization Constraints

In Teil II wird die Formulierung, Durchsetzung und Validierung von Authorization Constraints thematisiert. Hierzu werden in Kapitel 4 Voraussetzungen und Annahmen für die weiteren Betrachtungen formuliert. Kapitel 5 führt mit MΘnK ein Modell zur einheitlichen Modellierung der verschiedenartigen Authorization Constraints ein. Hierzu wird in Kapitel 5.2 die allgemeine Funktionsweise von MΘnK vorgestellt. Anschließend werden in Kapitel 5.3 die in Workflow-Management-Systemen relevanten Authorization Constraints auf MΘnK-Constraints abgebildet.

Auf Basis dieser einheitlichen Modellierung der Authorization Constraints mit MΘnK, wird in Kapitel 6 deren Durchsetzung zur Modellierzeit (Kapitel 6.1) und zur Laufzeit (Kapitel 6.2, 6.3) des Prozesses betrachtet. Kapitel 7 beschäftigt sich mit der Validierung der

MΘnK-Constraints. Dabei wird sowohl auf die Validierung einzelner MΘnK-Constraints (Kapitel 7.2), als auch auf die MΘnK-Constraint-übergreifende Validierung (Kapitel 7.3) eingegangen. Auch bei der Validierung der MΘnK-Constraints wird bei den Betrachtungen zwischen der Modellierzeit (Kapitel 7.4) und der Laufzeit (Kapitel 7.5) des Prozesses unterschieden. Details zu den im Rahmen dieser Arbeit vorgenommenen Implementierungen werden anschließend in Kapitel 8 betrachtet. Eine Zusammenfassung (Kapitel 9) und ein Ausblick (Kapitel 10) schließen Teil II ab.

2 Grundlagen

Im Folgenden werden die Grundlagen für Authorization Constraints in Workflow-Management-Systemen (WfMS) vorgestellt. Hierzu wird zunächst auf die Funktionsweise von WfMS eingegangen und grundlegende Begrifflichkeiten eingeführt. Dabei wird insbesondere auf die Modellierung und Ausführung von Prozessen, sowie auf Sicherheitsanforderungen im Zusammenhang mit der Ausführung eingegangen. Anschließend wird Role-Based Access Control (RBAC), ein rollenbasiertes Modell zur Berechtigungsvergabe und Zugriffskontrolle, vorgestellt. RBAC wird im weiteren Verlauf dieser Arbeit eine zentrale Rolle spielen, da dieses Modell zur Vergabe von Berechtigungen an Benutzer dienen wird.

2.1 Workflow-Management-Systeme (WfMS)

Workflow-Management-Systeme (**WfMS**) dienen der Modellierung und Ausführung von Geschäftsprozessen in Organisationen, wie z.B. Unternehmen oder Behörden. Ein **Geschäftsprozess** (kurz: **Prozess**) beschreibt dabei eine Folge von Aktivitäten, die schrittweise ausgeführt werden, um in der Gesamtheit ein Ergebnis zu erzielen.

2.1.1 Modellierung von Prozessen

Zur **Prozessmodellierung** müssen zunächst die einzelnen Aktivitäten des Prozesses identifiziert und anschließend in die richtige Ausführungsreihenfolge gebracht werden. Die **Aktivitäten** stellen dabei elementare Teilaufgaben des Prozesses dar, deren weitere Untergliederung nicht möglich, oder im jeweiligen Kontext nicht sinnvoll ist. Diese Aktivitäten müssen üblicherweise in einer bestimmten Reihenfolge abgearbeitet werden. Diese Reihenfolge wird durch den **Kontrollfluss** beschrieben, der somit auch die Abhängigkeiten zwischen den Aktivitäten beschreibt.

Für die grafische Modellierung solcher Prozesse bieten sich daher gerichtete Graphen an, wobei Knoten Aktivitäten darstellen und gerichtete Kanten den Kontrollfluss beschreiben. Abb. 2.1 zeigt einen einfachen Prozess mit zwei Aktivitäten, die mittels einer gerichteten Kante verbunden sind. Der Kontrollfluss besagt, dass „Aktivität 1“ vor „Aktivität 2“ ausgeführt werden muss.

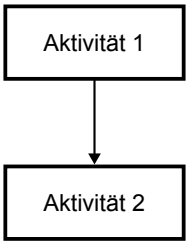


Abb. 2.1: Zwei mittels einer gerichteten Kante verbundene Aktivitäten.

Da die einzelnen Aktivitäten durch Mitarbeiter der Organisation ausgeführt werden müssen, ist zusätzlich eine entsprechende Zuweisung der Mitarbeiter an Aktivitäten notwendig. Diese Zuweisung geschieht üblicherweise auf Basis des Organisationsmodells der Organisation. Dieses **Organisationsmodell** bildet den strukturellen, meist hierarchischen, Aufbau einer Organisation ab. Das Ziel des Organisationsmodells ist es, Mitarbeiter gemäß ihrer Verantwortlichkeiten, Qualifikationen und Tätigkeiten auf eine sinnvolle Weise zu gruppieren. Diese Gruppierung findet in sogenannte **Organisationseinheiten** statt, die dann wiederum durch das Organisationsmodell in Beziehung zueinander gesetzt werden.

Im weiteren Verlauf dieser Arbeit wird an Stelle von Mitarbeitern von **Benutzern** gesprochen, da insbesondere diejenigen Mitarbeiter von Interesse sind, die gleichzeitig Benutzer des betrachteten WfMS sind.

In den einzelnen Aktivitäten werden üblicherweise Daten verarbeitet, so dass ein Prozess außerdem durch einen **Datenfluss** erweitert wird. Dieser beschreibt die Ein- und Ausgangsdaten der einzelnen Aktivitäten und inwiefern die Ausgangsdaten einer früheren Aktivität mit den Eingangsdaten späterer Aktivitäten zusammenhängen.

Bei der Prozessmodellierung entsteht somit ein sogenanntes **Prozessschema**, das im Kern aus Aktivitäten, Kontrollfluss, Datenfluss und Mitarbeiterzuordnungen besteht. Der Zeitraum der Prozessmodellierung wird im weiteren Verlauf dieser Arbeit als **Modellierzeit** bezeichnet. Dieser Begriff weist ausdrücklich darauf hin, dass bestimmte Vorgänge vor der

Ausführung des Prozesses stattfinden. Diese Ausführung von Prozessen wird im Folgenden betrachtet.

2.1.2 Ausführung von Prozessen

Nachdem ein Prozess modelliert wurde, und entsprechend ein Prozessschema vorliegt, ermöglichen WfMS eine computergestützte Ausführung der Prozesse. Von einem Prozessschema können hierbei mehrere **Prozessinstanzen** erzeugt und ausgeführt werden. Diese Ausführung mehrerer Prozessinstanzen kann zeitgleich erfolgen. Es ist dann die Aufgabe des WfMS sicher zu stellen, dass die einzelnen Aktivitäten einer Prozessinstanz in der richtigen Reihenfolge durch entsprechend berechtigte Benutzer ausgeführt werden.

Befindet sich eine Prozessinstanz in der Ausführung, so wird der Zeitraum vom Start bis zur Beendigung der Instanz als **Ausführungszeit** oder **Laufzeit** bezeichnet. Ist im weiteren Verlauf dieser Arbeit vereinfachend von der Ausführung eines Prozesses die Rede, so ist damit stets die Ausführung einer Prozessinstanz gemeint. Die entsprechenden Erläuterungen beziehen sich dann auf alle potentiellen Instanzen des Prozesses.

Jeder Prozessinstanz ist außerdem ein **Status** (auch **Prozessstatus**) zugeordnet. Dieser hält alle zum jeweils aktuellen Zeitpunkt relevanten Informationen. Hierzu gehören insbesondere Informationen über den aktuellen Zustand der einzelnen Aktivitäten der Prozessinstanz, sowie weitere Informationen, beispielsweise über vorliegende Daten. Alle im bisherigen Verlauf der Ausführung einer Prozessinstanz angefallenen Informationen (z.B. welche Aktivität durch welchen Benutzer ausgeführt wurde, oder Informationen über verarbeitete Daten) werden dagegen als **Ausführungshistorie** der Prozessinstanz bezeichnet. Alle in Prozessstatus und Ausführungshistorie gespeicherten Informationen werden gemeinsam auch als **Kontext** der Prozessinstanz bezeichnet.

Zur Ausführungszeit eines Prozesses verwaltet das WfMS **Arbeitslisten** für die einzelnen Benutzer. Eine solche Arbeitsliste enthält alle Aktivitäten, die zu dem jeweiligen Zeitpunkt durch den jeweiligen Benutzer ausgeführt werden können. Der Benutzer kann dann einen der Einträge der Arbeitsliste auswählen und somit die entsprechende Aktivität ausführen. Die entsprechenden Arbeitslisteneinträge werden dann durch das WfMS aus den Arbeitslisten aller anderen potentiellen Benutzer ausgetragen, so dass diese nicht dieselbe Aktivität ausführen können.

2.1.3 Sicherheit in WfMS

Ganz allgemein werden unter **Sicherheitsanforderungen** solche Anforderungen an ein System verstanden, die als Voraussetzung dafür dienen, dass das System als sicher gilt. Im Rahmen dieser Arbeit werden Sicherheitsanforderungen in WfMS betrachtet, die die sichere Ausführung der Prozesse zum Ziel haben. Ein Prozess wird dabei sicher ausgeführt, wenn die einzelnen Aktivitäten eines Prozesses nur durch entsprechend berechtigte Benutzer ausgeführt werden. Diese Sicherheitsanforderungen sind eng mit dem jeweiligen Prozess und dem vorliegenden Organisationsmodell verbunden. Dabei werden sowohl Sicherheitsanforderungen betrachtet die einzelne Aktivitäten betreffen, als auch solche die sich auf mehr als eine Aktivität beziehen. Diese Sicherheitsanforderungen werden in einer **Sicherheitsrichtlinie** beschrieben, deren Einhaltung vom WfMS gewährleistet werden muss.

Die Sicherheitsrichtlinie resultiert dann in einer **Berechtigungsvergabe**, bei der auf Basis der Prozessbeschreibung und des Organisationsmodells Berechtigungen an die Benutzer vergeben werden. Das WfMS muss dann vor der Ausführung einer Aktivität durch einen Benutzer sicherstellen, dass dies durch die Sicherheitsrichtlinie erlaubt wird. Dies geschieht durch Überprüfung der an die Benutzer vergebenen Berechtigungen.

Heutige WfMS beschränken sich auf Sicherheitsanforderungen, die sich auf einzelne Aktivitäten beziehen, also auf die Fragestellung, ob Benutzer zur Ausführung einzelner Aktivitäten berechtigt sind. Diese Berechtigungen für die einzelnen Aktivitäten werden dabei zur Modellierzeit an einzelne Benutzer vergeben und sind dann zur Ausführungszeit des Prozesses nicht mehr änderbar.

Darüber hinaus ist jedoch auch die Umsetzung komplexerer Sicherheitsanforderungen, durch die eine differenziertere Berechtigungsvergabe an die Benutzer ermöglicht wird, wünschenswert. Diese komplexen Sicherheitsanforderungen werden mit Hilfe von Authorization Constraints formuliert und beziehen sich in der Regel auf einen größeren Kontext, beispielsweise auf mehrere Aktivitäten. Derartige Authorization Constraints schränken dann wahlweise die zur Modellierzeit vergebbaren Berechtigungen ein oder ermöglichen die Modifikation der vergebenen Berechtigungen zur Ausführungszeit des Prozesses.

Authorization Constraints und deren Umsetzung in WfMS stehen nach den folgenden Betrachtungen zu Role-Based Access Control im Fokus dieser Arbeit.

2.2 Role-Based Access Control (RBAC)

Sicherheitsrichtlinien in Organisationen drücken Berechtigungen üblicherweise mit Hilfe von Rollen aus und binden Berechtigungen nicht direkt an einzelne Benutzer [2]. **Role-Based Access Control** (**RBAC**; dt. Rollenbasierte Zugriffskontrolle) [13] ist eine weit verbreitete Möglichkeit, derartige Sicherheitsrichtlinien systematisch zu modellieren und zu erzwingen. Die Zugriffskontrolle geschieht dann auf Basis von Rollen und nicht auf Basis einzelner Benutzer. Die Entscheidung ob ein Benutzer bestimmte Berechtigungen besitzt, basiert somit auf der Frage, ob dieser einer bestimmten Rolle angehört.

Wir betrachten zunächst einfaches RBAC. Dieses legt den Grundstein für die Vergabe von Berechtigungen mittels Rollen und führt alle grundlegenden, und für RBAC unverzichtbaren, Konzepte ein.

2.2.1 Einfaches RBAC

Im Rahmen dieser Arbeit werden unter **Einfachem RBAC** [13] die grundlegenden Konzepte von RBAC verstanden. RBAC macht dabei Gebrauch von fünf Mengen, namentlich „Benutzer“, „Rollen“, „Berechtigungen“, „Objekte“ und „Operationen“. Diese Mengen werden durch RBAC in Beziehung zueinander gesetzt und ermöglichen so die Modellierung von Sicherheitsrichtlinien.

In diesem RBAC-Grundkonzept werden zunächst Benutzer passenden Rollen zugeordnet. Diese Zuordnung von Benutzern zu Rollen geschieht unter Berücksichtigung der Verantwortlichkeiten und Qualifikationen der Benutzer [2, 5, 6]. Die **Rollen** repräsentieren dabei Berufs- oder Tätigkeitsbeschreibungen innerhalb der Organisation und entsprechen somit entweder Organisationseinheiten oder bestimmten Vorgängen. RBAC orientiert sich somit eng an der vorliegenden Organisationsstruktur. Damit sind mit Rollen implizit auch gewisse Semantiken und Erwartungshaltungen verbunden.

Darauf aufbauend werden den Rollen solche Berechtigungen zugeordnet, die zur Erfüllung der durch die Rolle beschriebenen Tätigkeit erforderlich sind. Eine Berechtigung gibt dabei jeweils an, welche Operationen auf welchen Objekten erlaubt sind. Dieser Zusammenhang wird in Abb. 2.2 veranschaulicht. Die genaue Form dieser Operationen und Objekte ist dabei vom jeweiligen Einsatzbereich des RBAC-Modells abhängig.

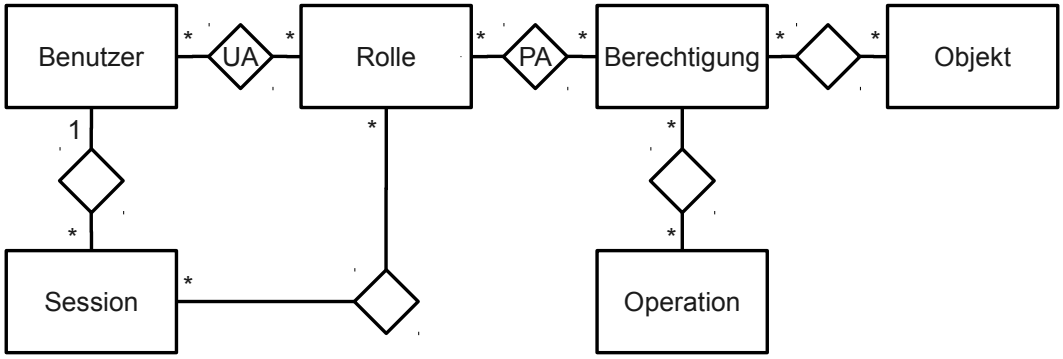


Abb. 2.2: Zusammenwirken der Grundkonzepte bei Einfachem RBAC mit Sessions.

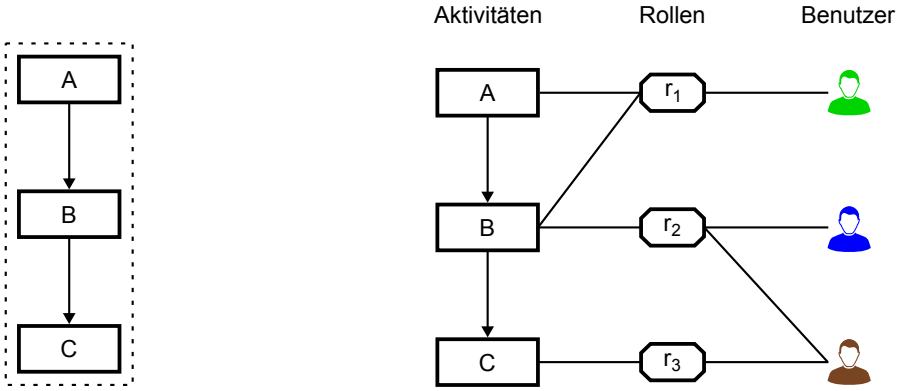
Benutzerzuordnung und Berechtigungszuordnung

RBAC teilt somit die Vergabe von Berechtigungen an Benutzer in zwei logisch getrennte Teile: das Zuordnen von Benutzern zu Rollen (User Assignment (UA); dt. **Benutzerzuordnung** bzw. **Benutzer-Rollen-Zuordnung**) und das Zuordnen von Berechtigungen zu Rollen (Permission Assignment (PA); dt. **Berechtigungszuordnung**) [30]. Diese Zuordnungen werden von einem Sicherheitsadministrator vorgenommen. Dabei kann ein Benutzer mehreren Rollen zugeordnet sein und eine Rolle kann mehreren Benutzern zugeordnet sein. Dasselbe gilt für die Kardinalitäten zwischen den Rollen und den Berechtigungen. RBAC beschreibt damit sowohl eine $n : m$-Beziehung zwischen Benutzern und Rollen, als auch eine $n : m$-Beziehung zwischen Rollen und Berechtigungen [13, 35]. Dieser Zusammenhang wird in Abb. 2.2 verdeutlicht.

Im Zusammenhang mit WfMS kann das Zuordnen von Rollen zu Aktivitäten als Berechtigungszuordnung angesehen werden. Wird eine Rolle einer Aktivität zugeordnet, so erhalten alle der Rolle zugeordneten Benutzer die Berechtigung zur Ausführung der entsprechenden Aktivität [7]. Diese Zuordnung zwischen Aktivitäten und Rollen wird im weiteren Verlauf dieser Arbeit als **Aktivitäten-Rollen-Zuordnung** bezeichnet. Benutzer können folglich alle Aktivitäten ausführen, denen eine Rolle zugeordnet ist, deren Mitglied sie sind.

Abb. 2.3a zeigt einen Prozess mit drei Aktivitäten A, B und C. Abb. 2.3b verdeutlicht den Zusammenhang zwischen Aktivitäten, Rollen und Benutzern in Form eines Berechtigungsgraphen nach [1]. So erhält der grüne Benutzer über Rolle r_1 Zugriff auf die Aktivitäten A und B, während der blaue Benutzer über Rolle r_2 die Berechtigung für Aktivität B erhält.

Der braune Benutzer erhält über Rolle r_2 die Berechtigung für Aktivität B und über Rolle r_3 die Berechtigung für Aktivität C.



(a) Prozess bestehend aus den Aktivitäten A, B und C.

(b) Berechtigungsgraph nach [1].

Abb. 2.3: Prozess und zugehöriger Berechtigungsgraph mit Rollen und Benutzern.

2.2.2 RBAC Sessions

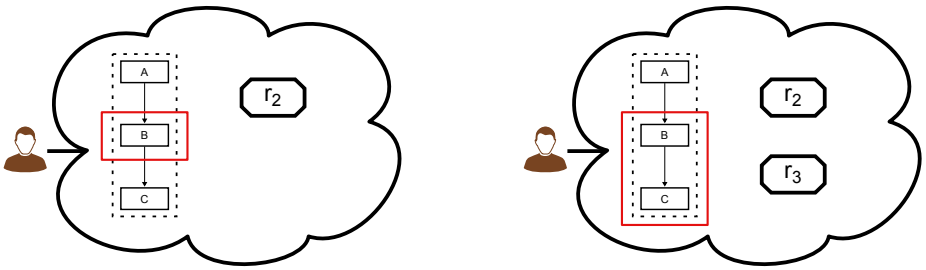
RBAC macht zusätzlich Gebrauch von einem **Session**-Konzept, wobei jedem Benutzer mehrere Sessions zugeordnet sein können, eine Session jedoch genau einem Benutzer zugeordnet ist [13]. Eine Session hält Informationen über alle Rollen, die der jeweilige Benutzer innerhalb des Zeitraums der Gültigkeit dieser Session aktiviert hat. Abb. 2.2 zeigt diesen Zusammenhang zwischen Benutzern, Rollen und Sessions.

Es ist also durchaus möglich, dass ein Benutzer mehrere Rollen gleichzeitig annimmt und damit aktiviert hat. Dieser Vorgang wird auch als **Rollenaktivierung** bezeichnet. Diese Menge der aktivierten Rollen ist eine Teilmenge aller dem Benutzer zugeordneten Rollen. Ist eine Rolle aktiviert, so bedeutet dies, dass der Benutzer über die der Rolle zugeordneten Berechtigungen verfügt. Betrachtet man alle zu einem Benutzer gehörenden Sessions und die darin enthaltenen aktivierten Rollen, so kann ermittelt werden, über welche Berechtigungen der Benutzer zu diesem Zeitpunkt verfügt.

Im Zusammenhang mit Sessions kommt dabei zwangsläufig die Frage zu den Gültigkeitszeiträumen der Sessions auf. In der in diesem Kapitel zitierten Fachliteratur werden hierzu jedoch keine Vorschläge diskutiert. Im Rahmen dieser Arbeit werden die Gültigkeitszeiträume der Sessions daher mit der Ausführungszeit der jeweiligen zugehörigen Prozessinstanz

synchronisiert. Da RBAC in dieser Arbeit ausschließlich im Zusammenhang mit Prozessabläufen eingesetzt wird, ist dies ein naheliegender Ansatz. Das bedeutet, dass pro Prozessinstanz und Benutzer eine Session instanziert wird. Die innerhalb einer Prozessinstanz aktivierten Rollen eines Benutzers werden dann in der entsprechenden Session vermerkt. Bei Beendigung der jeweiligen Prozessinstanz verlieren die zugehörigen Sessions ihre Gültigkeit.

Aktiviert der braune Benutzer in Abb. 2.3b die Rolle r_2, so wird diese Rolle in einer Session vermerkt. Diese Session ist wiederum dem aktivierenden Benutzer zugeordnet und bezieht sich auf die entsprechende Prozessinstanz (Abb. 2.4a). Durch Aktivierung der Rolle r_2 ist der Benutzer dann zur Ausführung von Aktivität B berechtigt. Wird zusätzlich die Rolle r_3 von demselben Benutzer aktiviert, so wird auch diese in der Session vermerkt und der Benutzer ist zusätzlich zur Ausführung von Aktivität C berechtigt (Abb. 2.4b). Die Session verliert ihre Gültigkeit bei Beendigung der Prozessinstanz, also nach Ausführung der Aktivität C.



(a) Session mit aktivierter Rolle r_2.

(b) Session nach zusätzlicher Aktivierung der Rolle r_3.

Abb. 2.4: Session des braunen Benutzers nach Aktivierung der Rolle r_2 bzw. r_3.

Das beschriebene RBAC-Grundmodell mit Sessions kann durch weitere Konzepte erweitert werden, um die Ausdrucksmächtigkeit zu erhöhen und den Modellierungs- und Administrationsaufwand zu verringern. Dabei ist die Hierarchisierung der Rollen die wichtigste und am meisten Verwendung findende Erweiterung.

2.2.3 Hierarchisches RBAC

Hierarchisches RBAC erweitert das einfache RBAC-Modell dahingehend, dass die Rollen in hierarchischen Beziehungen zueinander stehen können. Dies ermöglicht die Strukturie-

rung von Rollen und die Abbildung von Verantwortlichkeiten und Autoritäten der Organisation [13]. Im Rahmen dieser Arbeit werden wir uns auf baumartige **Rollenhierarchien** (**RH**) beschränken, da Baumstrukturen in den meisten verbreiteten Systemen die Regel sind und auch Systemressourcen typischerweise in dieser Form visualisiert werden [12]. Abb. 2.5 veranschaulicht diese zusätzliche Beziehung zwischen Rollen.

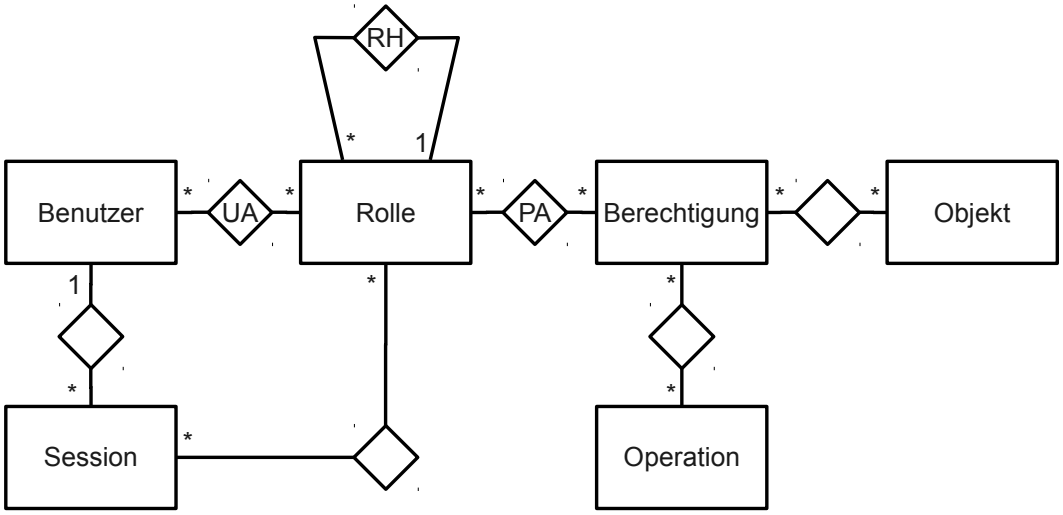


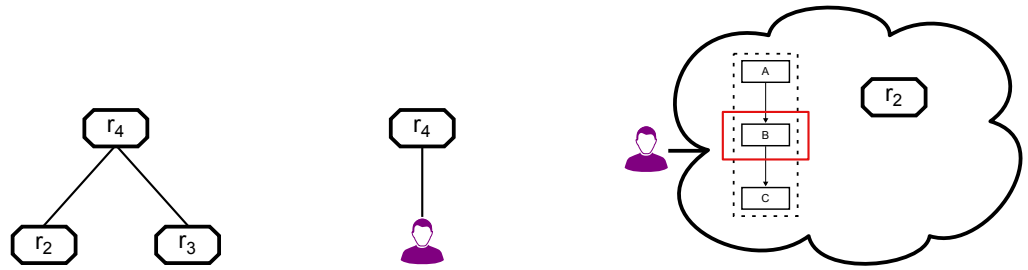
Abb. 2.5: Hierarchisches RBAC: Eine zusätzliche Beziehung zwischen den Rollen.

Zusätzlich besteht zwischen den Rollen meist eine partielle Ordnung $\succ$, die die Rollen im Sinne der Organisationsstruktur der Organisation anordnet [5]. Eine Rolle r_i **dominiert** dabei eine Rolle r_j, falls $r_i \succ r_j$. r_i erbt dann sämtliche Berechtigungen von r_j und wird als mächtigere der beiden Rollen oberhalb von r_j notiert. Berechtigungen werden also stets „bottom-up“ vererbt. Gleichzeitig sind alle Benutzer der Rolle r_i automatisch auch Benutzer der Rolle r_j, wodurch Benutzer-Rollen-Zuordnungen „top-down“ vererbt werden. Da nur baumartige Rollenhierarchien betrachtet werden, ist eine Mehrfachvererbung nicht möglich. Daher hat jede Rolle höchstens eine direkte Vorgängerrolle.

Bei der Rollenaktivierung soll r_j dann üblicherweise eine höhere Priorität als die dominierende r_i erhalten, falls die Berechtigungen von r_j für die Erledigung der entsprechenden Aktivität ausreichend sind. Dadurch, dass die Rolle mit den geringeren Berechtigungen eine höhere Priorität erhält, wird das **Least-Privilege-Prinzip** durchgesetzt [6]. Das bedeutet, dass stets nur die Rolle aktiviert wird, die für die Erledigung einer Aktivität tatsächlich notwendig ist. Weiterreichende Berechtigungen von übergeordneten Rollen werden damit

nicht an den Benutzer weitergegeben und können somit auch nicht missbraucht werden. Eine Rolle r_i wird also nur dann aktiviert, wenn die Berechtigungen einer untergeordneten Rolle r_j ($r_i \succ r_j$) nicht zur Erledigung einer Aktivität ausreichen.

Wir betrachten hierzu erneut die Aktivitäten und Rollenzuordnung aus Abb. 2.3b. Es wird eine zusätzliche Rolle r_4 eingeführt, die die Berechtigungen der Rollen r_2 und r_3 erbt (Abb. 2.6a). Rolle r_4 dominiert also die Rollen r_2 und r_3. Wird ein Benutzer dieser neuen Rolle zugeordnet (Abb. 2.6b), so erhält dieser folglich die Berechtigungen der Rollen r_2 und r_3 und ist damit zur Ausführung der Aktivitäten B und C berechtigt. Führt der Benutzer die Aktivität B aus, so wird nicht die ihm zugeordnete Rolle r_4 aktiviert. Da bereits die Berechtigungen der Rolle r_2 ausreichen, wird lediglich diese untergeordnete Rolle aktiviert und somit in der entsprechenden Session vermerkt (Abb. 2.6c). Da somit nicht auch automatisch die Rolle r_3 aktiviert wird, erhält der Benutzer nur die Berechtigungen, die für die Ausführung der Aktivität tatsächlich notwendig sind. Auf diese Weise kann mit Hilfe der Rollenhierarchien und des Session-Konzeptes das Least-Privilege-Prinzip durchgesetzt werden.



(a) Rolle r_4 dominiert die Rollen r_2 und r_3.

(b) Benutzer der Rolle r_4.

(c) Session des Benutzers bei Ausführung der Aktivität B.

Abb. 2.6: Rollenvererbung und Least-Privilege-Prinzip bei Ausführung der Aktivität B.

Durch die Möglichkeit der Vererbung von Berechtigungen wird der Administrationsaufwand beim Erstellen einer Sicherheitsrichtlinie deutlich reduziert [6]. Berechtigungen die an unterschiedlichen Stellen gemeinsam vergeben werden sollen, lassen sich so in einer Rolle bündeln. Diese Rolle kann dann dazu genutzt werden, um die gebündelten Berechtigungen an andere Rollen zu vererben.

2.2.4 Vorteile von RBAC

Mit RBAC wird die Zugriffskontrolle deutlich vereinfacht, wodurch dies ein mächtiges Instrument ist, um Kosten und potentielle Fehler zu verringern [12]. So ist in der Regel die Anzahl der Rollen sehr viel geringer als die Anzahl der Benutzer [5]. Außerdem sind Organisationsstrukturen, und damit die den Organisationseinheiten entsprechenden Rollen, deutlich seltener von Änderungen betroffen als einzelne Benutzer [23]. Bei großen Organisationen mit vielen Benutzern wird somit der Administrationsaufwand mit Hilfe von Rollen deutlich reduziert. Zusätzlich kann auf Veränderungen bei Benutzern schnell reagiert werden: Soll ein Benutzer zukünftig andere Aufgaben wahrnehmen, so müssen diesem lediglich die bisherigen Rollen entzogen und die neuen gewünschten Rollen zugeordnet werden.

Gerade in Organisationen mit vielen Rollen und tiefen Hierarchien nimmt hierarchisches RBAC einen großen Teil des Administrationsaufwandes ab und ermöglicht eine einfache und natürliche Modellierung der Organisationsstruktur. Mit dem Konzept der hierarchischen Rollen ist es außerdem möglich, überlappende Berechtigungen zu gruppieren und dann an andere Rollen weiter zu vererben.

RBAC bietet außerdem den Vorteil Richtlinien-neutral [20] zu sein. Das bedeutet, dass mit RBAC sehr viele unterschiedliche Sicherheitsrichtlinien modelliert und erzwungen werden können. Der Einsatz von RBAC ist damit in vielen unterschiedlichen Szenarien möglich.

Gegenüber herkömmlichen Zugriffskontrollmodellen wie Mandatory Access Control (MAC) [25] oder Discretionary Access Control (DAC) [24] bietet RBAC damit viele Vorteile, wodurch dieses Modell in vielen heutigen Systemen zum Einsatz kommt. Auch WfMS verwenden RBAC als Grundgerüst für die Rechteverwaltung, stoßen jedoch bei komplexen Szenarien auf die im Folgenden beschriebenen Einschränkungen.

2.2.5 Unzulänglichkeiten von RBAC

Trotz der vielen Vorteile gegenüber anderen Zugriffskontrollmechanismen ist RBAC für WfMS, mit denen auch komplexe Szenarien abgebildet werden sollen, nicht ausreichend. So können mit RBAC lediglich statische Berechtigungen vergeben werden, die unabhängig von der Ausführungshistorie und anderen Kontextinformationen einzelner Prozessinstanzen sind. Die Umsetzung weitergehender, in WfMS gewünschter, Sicherheitsanforderungen wie Authorization Constraints ist mit RBAC daher nicht möglich.

3 Authorization Constraints

Authorization Constraints (im Rahmen dieser Arbeit kurz **Constraints**) haben, wie der Name bereits andeutet, zum Ziel Berechtigungen einzuschränken. Bei Verwendung von RBAC ist die Zuordnung eines Benutzers zu einer Rolle daher nach [11] eine notwendige, aber nicht immer auch hinreichende Bedingung, um Berechtigungen zu erlangen. Für die tatsächlich gültigen Berechtigungen müssen zusätzlich zu den Benutzer-Rollen-Zuordnungen und den aktivierten Rollen auch weiterführende Sicherheitsaspekte, wie Einschränkungen durch Constraints, berücksichtigt werden. Nach [29] soll mit Constraints die sichere Ausführung eines Workflows erzwungen werden können.

Dieses Kapitel beleuchtet zunächst, welche Ziele mit Constraints verfolgt werden und wie diese umgesetzt werden. Anschließend folgt ein Überblick über die in der Literatur am meisten diskutierten Authorization Constraints. Diese werden jeweils an einem Fallbeispiel erläutert. Zu den betrachteten Constraints gehören Entailment Constraints, Separation of Duty Constraints, Binding of Duty Constraints und Cardinality Constraints. Separation of Duty wird wiederum in statisches und dynamisches Separation of Duty unterteilt. Das Vier-Augen-Prinzip kann außerdem als wichtiger Spezialfall von Separation of Duty angesehen werden.

3.1 Einsatz von Constraints

Neben den durch Constraints verfolgten Zielen, wird im Folgenden auch auf die Umsetzung der Constraints eingegangen. Besondere Beachtung findet dabei das Zusammenwirken der Constraints mit der Berechtigungsvergabe durch RBAC.

3.1.1 Zielsetzung von Constraints

Mit Constraints soll es möglich werden, Berechtigungen auf eine dynamische, flexible und systematische Weise einzuschränken. Ziel ist es, auf Ereignisse in einzelnen Prozessinstanzen reagieren und die Berechtigungen entsprechend anpassen zu können. Berechtigungen sollen also nicht mehr nur instanzunabhängig vergeben und durchgesetzt werden, sondern sich in verschiedenen Instanzen desselben Prozesses unterscheiden können. Hierbei spielen Kontextinformationen wie Status und Ausführungshistorie des Prozesses eine wichtige Rolle. Erst durch diese Informationen werden dynamische und instanzabhängige Sicherheitsanforderungen formulierbar und durchsetzbar.

Es soll an dieser Stelle ausdrücklich darauf hingewiesen werden, dass Constraints nicht nur die Einschränkung sondern auch die Ausweitung vorhandener Berechtigungen ermöglichen. Diese Arbeit beschränkt sich jedoch auf die Einschränkung von Berechtigungen, da diese Anwendung in der Natur der Constraints liegt. In den meisten Fällen ist jedoch auf eine ähnliche Weise die Ausweitung der Berechtigungen durch Constraints möglich.

3.1.2 Umsetzung von Constraints

Die Sicherheitsrichtlinien in Organisationen werden heutzutage üblicherweise mit Hilfe von RBAC ausgedrückt. Da dieses Konzept in allen in dieser Arbeit betrachteten Constraint-Modellen Verwendung findet, legen wir bei den folgenden Betrachtungen stets eine Berechtigungsvergabe durch RBAC zugrunde.

Um die dynamische und instanzspezifische Einschränkung von Berechtigungen zu ermöglichen, wird den Constraints eine höhere Priorität als den mittels RBAC vergebenen Berechtigungen zugewiesen. Soll also ermittelt werden, ob Berechtigungen für eine bestimmte Operation vorliegen, so werden zunächst die mittels RBAC vergebenen Berechtigungen daraufhin untersucht. Existieren zusätzlich Constraints die in der entsprechenden Situation Anwendung finden, so schränken diese anschließend die mittels RBAC vergebenen Berechtigungen ein. Constraints formulieren also Bedingungen, bei deren Eintreten die RBAC-Berechtigungen überschrieben werden [32]. In vielen Fällen beziehen sich die in Constraints formulierten Bedingungen auf Kontextinformationen, die erst im Verlauf der jeweiligen Prozessinstanz bekannt werden. Daher kann oft erst zur Laufzeit des Prozesses festgestellt werden, ob die entsprechenden Constraints eingehalten werden.

In [7] werden fünf wichtige Eigenschaften eines Modells zur Spezifikation von Constraints identifiziert. Diese werden in Kapitel 5 von enormer Bedeutung sein. Demnach sollte ein solches Modell

- keine Doppeldeutigkeiten zulassen,
- eine große Ausdrucksmächtigkeit besitzen, mit der eine Vielzahl unterschiedlicher Constraints und Sicherheitsanforderungen formuliert werden können,
- so einfach sein, dass es von Anwendungsentwicklern und Administratoren aus betrieblichen Fachbereichen genutzt werden kann,
- unabhängig von dem darunterliegenden Workflow und eingesetzten Referenzmonitor sein,
- eine einfache Analyse der Constraints ermöglichen.

Alle in dieser Arbeit betrachteten Constraints werden an einem Fallbeispiel illustriert. Dieses wird im Folgenden mit einer rollenbasierten Berechtigungsvergabe eingeführt. Im weiteren Verlauf dieser Arbeit werden dann Constraints auf diesem Fallbeispiel formuliert.

3.2 Fallbeispiel: Warenbestellung

Im Fallbeispiel soll eine Warenbestellung mit anschließendem Waren- und Rechnungseingang sowie Rechnungsprüfung und -bezahlung durchgeführt werden. Der Prozess besteht daher aus den folgenden einzelnen Aktivitäten:

- Bestellung schreiben
- Bestellung prüfen
- Bestellung aufgeben
- Rechnung erfassen
- Wareneingang erfassen
- Rechnung prüfen
- Bezahlung veranlassen

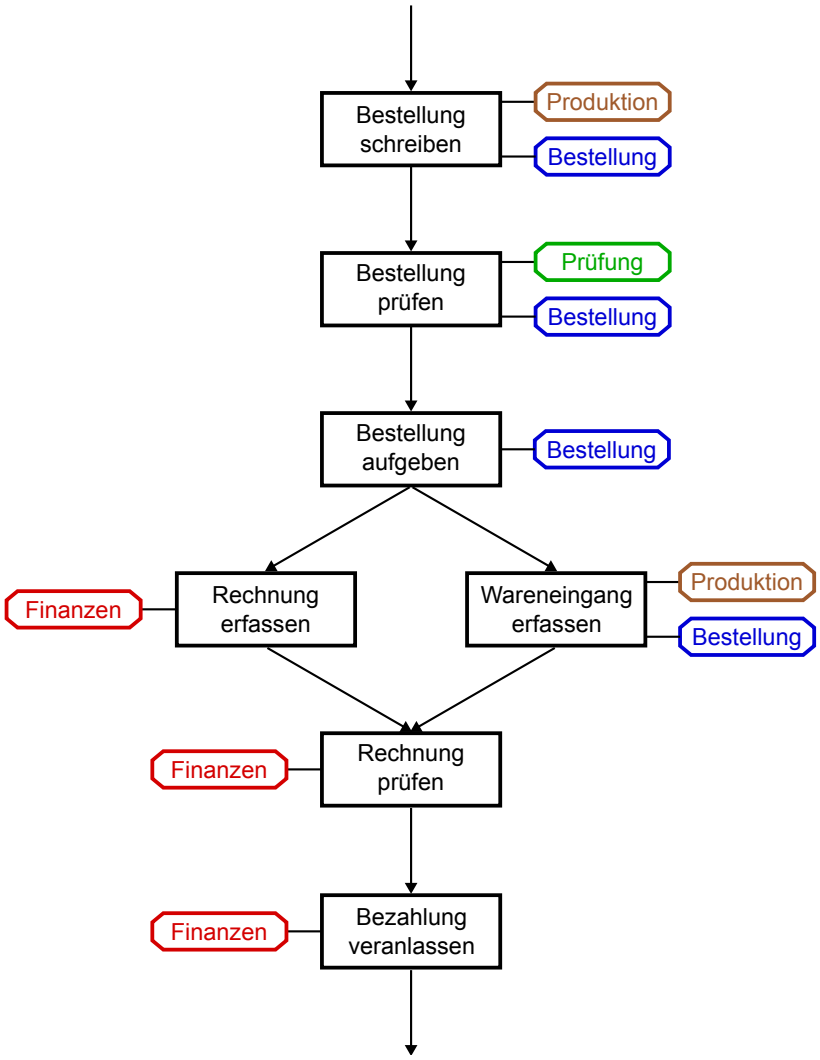


Abb. 3.1: Prozessgraph des „Warenbestellung“-Fallbeispiels mit Rollenzuordnung.

Um eine rollenbasierte Berechtigungsvergabe für diese Aktivitäten zu ermöglichen, werden außerdem die folgenden vier Rollen eingeführt:

- Bestellung
- Finanzen
- Prüfung
- Produktion

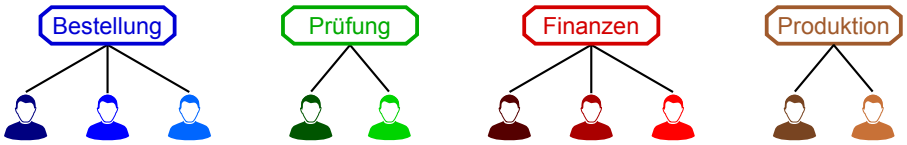


Abb. 3.2: Benutzer-Rollen-Zuordnung im Fallbeispiel.

Abb. 3.1 veranschaulicht den Ablauf der Warenbestellung. Aktivitäten sind dabei als Rechtecke dargestellt, Rollen als Achtecke. Ist eine Rolle mittels einer Kante mit einer Aktivität verbunden, so berechtigt diese Rolle zur Ausführung der entsprechenden Aktivität. Zur leichteren Unterscheidung und Gruppierung sind die Rollen unterschiedlich eingefärbt. Abb. 3.2 zeigt die den Rollen zugeordneten Benutzer. Diese entsprechen in der Farbgebung jeweils der entsprechenden Rolle. Ist ein Benutzer innerhalb einer Aktivität abgebildet, so bedeutet dies, dass die Aktivität durch diesen Benutzer ausgeführt wird (Abb. 3.3).

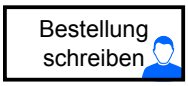


Abb. 3.3: Ausführung von „Bestellung schreiben“ durch den abgebildeten Benutzer.

Einen Überblick über die Aktivitäten-Rollen-Zuordnung gibt auch Tabelle 3.1. Dabei bedeutet ein X, dass eine Zuordnung zwischen Aktivität und Rolle besteht. Ein der Rolle zugeordneter Benutzer ist demnach zur Ausführung der entsprechenden Aktivität berechtigt.

Nachdem nun Grundlagen zum Einsatz von Authorization Constraints erörtert und ein Fallbeispiel vorgestellt wurde, werden im Folgenden konkrete Authorization Constraints vorgestellt.

	Bestellung	Prüfung	Finanzen	Produktion
Bestellung schreiben	X			X
Bestellung prüfen	X	X		
Bestellung aufgeben	X			
Rechnung erfassen			X	
Wareneingang erfassen	X			X
Rechnung prüfen			X	
Bezahlung veranlassen			X	

Tabelle 3.1: Aktivitäten-Rollen-Zuordnung im Fallbeispiel „Warenbestellung“.

3.3 Entailment Constraints

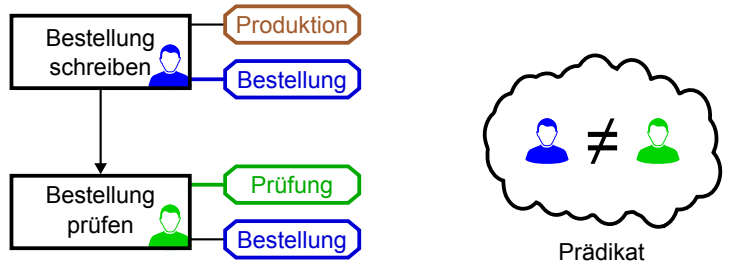
In [29] wird ein **Entailment Constraint** definiert als ein Constraint, der sich auf zwei Aktivitäten bezieht. Ein Entailment Constraint fordert, dass eine Aktivität von einer anderen Aktivität eingeschränkt wird. Damit dies möglich ist, muss die Ausführungsreihenfolge dieser beiden Aktivitäten vorab feststehen. Die Aktivitäten werden als **einschränkende Aktivität** und **eingeschränkte Aktivität** bezeichnet.

Ein Entailment Constraint beschreibt, welchen Einfluss die einschränkende Aktivität auf die eingeschränkte Aktivität hat [7, 8]. Neben diesen beiden Aktivitäten wird dem Constraint eine sogenannte **Domain** zugeordnet, die einer Menge von Benutzern entspricht. Beim Einsatz von RBAC kann sich diese Domain auch auf Rollen beziehen, so dass alle der Rolle zugeordneten Benutzer in der Domain enthalten sind [8, 29]. Mit Hilfe dieser Domain wird es möglich, den Anwendungsbereich des Constraints einzuschränken. Der Constraint wird nur dann durchgesetzt, wenn die einschränkende Aktivität durch einen Benutzer ausgeführt wird, der in der Domain enthalten ist. Ist dies nicht der Fall, findet der Constraint keine Anwendung.

Muss der Entailment Constraint durchgesetzt werden, findet eine weitere Komponente des Constraints, das sogenannte **Prädikat**, Anwendung. Dieses Prädikat beschreibt, in welcher Beziehung die ausführenden Benutzer der einschränkenden und der eingeschränkten Aktivität stehen müssen. Damit der Constraint eingehalten wird, müssen diese beiden Benutzer in der vom Prädikat geforderten Beziehung zueinander stehen.

Entailment Constraint im Fallbeispiel: Die Aktivitäten „Bestellung schreiben“ und „Bestellung prüfen“ sollen stets von zwei unterschiedlichen Benutzern ausgeführt werden, wie in Abb. 3.4a gezeigt. Hierbei ist „Bestellung schreiben“ die einschränkende Aktivität und

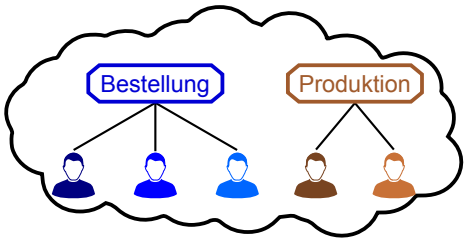
„Bestellung prüfen“ die eingeschränkte Aktivität. Die Domain enthält alle potentiellen Benutzer der Aktivität „Bestellung schreiben“ (Abb. 3.4c). Damit die beiden Aktivitäten immer von unterschiedlichen Benutzern ausgeführt werden, muss das Prädikat die Tatsache formulieren, dass die Benutzer der beiden Aktivitäten nicht gleich sein dürfen (Abb. 3.4b).



(a) Einschränkende Aktivität „Bestellung schreiben“ und eingeschränkte Aktivität „Bestellung prüfen“.

Prädikat

(b) Forderung der Ungleichheit der beiden Benutzer.



Domain

(c) Die Domain bezieht sich auf alle Benutzer der einschränkenden Aktivität.

Abb. 3.4: Entailment Constraint: Einschränkende Aktivität „Bestellung schreiben“, eingeschränkte Aktivität „Bestellung prüfen“, Prädikat und Domain.

Mit einem Entailment Constraint ist es also möglich, die Menge der potentiellen Benutzer einer Aktivität einzuschränken, wenn zuvor eine andere Aktivität von einem Benutzer ausgeführt wurde, der der Domain angehört. Zu beachten ist, dass im Prädikat auch solche Beziehungen zwischen Benutzern formulierbar sind, die erst mit Hilfe von Kontextinformationen zur Laufzeit auswertbar sind. In [34] werden Entailment Constraints außerdem dahingehend erweitert, dass diese sich auf mehr als zwei Aktivitäten beziehen.

3.4 Separation of Duty Constraints

Erstmals erwähnen Saltzer und Schroeder 1975 in [22] **Separation of Duty** (**SoD**), damals noch unter dem Namen „Separation of Privilege“. Das Ziel von SoD ist es, Interessenskonflikte und unbeabsichtigte Fehler zu vermeiden, sowie Betrug auszuschließen oder zu erschweren [29].

Um dieses Ziel zu erreichen muss sichergestellt werden, dass kein Benutzer den Prozess alleine zu Ende führen kann. In diesem Fall trägt dann kein Benutzer die alleinige Verantwortung für den Gesamtprozess [18]. Um einen Betrug zu begehen, müssten dann mehrere Benutzer zusammenarbeiten. Da die Zuordnung der Rollen zu Benutzern auf Grundlage ihrer Verantwortlichkeiten und Qualifikationen beruht (vgl. Kapitel 2.2.1), ist Separation of Duty eng an die jeweilige Anwendungssemantik gebunden [28]. Durch SoD sind an der Erledigung eines Prozesses folglich mindestens zwei Benutzer beteiligt, wodurch unbeabsichtigte Fehler entdeckt und Betrug durch einzelne Benutzer ausgeschlossen oder zumindest erschwert werden kann.

In der Literatur [6, 11, 16, 17, 28] wird Separation of Duty in zwei Kategorien unterschieden: Das zur Modellierzeit durchsetzbare Statische Separation of Duty und das erst zur Laufzeit auswertbare Dynamische Separation of Duty. Beide Möglichkeiten werden im Folgenden vorgestellt.

3.4.1 Statisches Separation of Duty

Die einfachste Form von Separation of Duty ist **Statisches Separation of Duty** (**SSoD**; engl.: **Static Separation of Duty**). SSoD ermöglicht die Formulierung von SoD-Constraints, die von einer statischen Beschaffenheit sind. Das bedeutet, dass die durch diese Constraints auferlegten Einschränkungen unveränderlich sind und bedingungslos während der gesamten Laufzeit Gültigkeit besitzen.

Bei SSoD werden zwei Rollen als **streng exklusiv** oder als sich **gegenseitig ausschließend** bezeichnet, wenn ein Benutzer niemals beide Rollen aktivieren darf [12, 16, 28]. Das heißt, dass zwei streng exklusiven Rollen niemals ein gemeinsamer Benutzer zugeordnet sein darf. SSoD kann damit allein durch Einschränkung der Benutzer-Rollen-Zuordnung durchgesetzt werden [28].

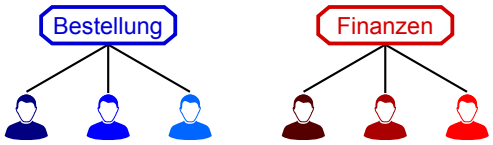


Abb. 3.5: Statisches Separation of Duty: Die Benutzer der Rollen „Bestellung“ und „Finanzen“ sind disjunkt.

Statisches Separation of Duty im Fallbeispiel: Die Rollen „Bestellung“ und „Finanzen“ repräsentieren unterschiedliche Abteilungen der Organisation. Da ein Benutzer jedoch nur einer Abteilung angehört, soll kein Benutzer beiden Rollen zugeordnet werden können. Die Rollen „Bestellung“ und „Finanzen“ sind daher streng exklusiv. Wie Abb. 3.5 zeigt, kann ein Benutzer daher nicht beiden Rollen zugeordnet werden. Die Benutzer der beiden Rollen sind disjunkt.

Unzulänglichkeiten von Statischem Separation of Duty

SSoD wird in der Literatur stets auf Rollen definiert [12, 16, 28]. Im Folgenden werden die Unzulänglichkeiten von SSoD auf Rollen im Kontext von WfMS demonstriert.

Abb. 3.6: Statisches Separation of Duty: Die Benutzer der Rollen „Bestellung“, „Prüfung“ und „Produktion“ sind paarweise disjunkt.

Unzulänglichkeit von Statischem Separation of Duty im Fallbeispiel: Um Fehler bei der Warenbestellung rechtzeitig erkennen zu können, sollen die Aktivitäten „Bestellung schreiben“ und „Bestellung prüfen“ von zwei unterschiedlichen Benutzern ausgeführt werden. Die Rollen „Bestellung“, „Prüfung“ und „Produktion“ sind hierzu paarweise streng exklusiv (Abb. 3.6). SSoD macht jedoch keine Einschränkung bezüglich der Zuordnung von Aktivitäten zu Rollen, so dass eine Aktivitäten-Rollen-Zuordnung wie in Abb. 3.7a erlaubt ist. Da die Rolle „Bestellung“ beiden Aktivitäten „Bestellung schreiben“ und „Bestellung prüfen“ zugeordnet ist, kann ein Benutzer dieser Rolle in einer Prozessinstanz beide Aktivitäten ausführen (Abb. 3.7b).

(a) Bei SSoD erlaubte Aktivitäten-Rollen-Zuordnung.

(b) Ausführung der beiden Aktivitäten durch denselben Benutzer.

Abb. 3.7: Aushebelung des gewünschten Separation of Duty auf den beiden Aktivitäten durch entsprechende Aktivitäten-Rollen-Zuordnung.

Trotz des eingehaltenen SSoD-Constraints auf den Rollen „Bestellung“, „Prüfung“ und „Produktion“, ist also offenbar eine Durchsetzung von Separation of Duty auf den beiden Aktivitäten „Bestellung schreiben“ und „Bestellung prüfen“ nicht möglich. In WfMS soll jedoch gerade ein Separation of Duty auf Aktivitäten umgesetzt werden können. SSoD ist ohne weitere Einschränkungen hierzu offenbar nicht geeignet. Hierbei ist anzumerken, dass derartige unerwünschte Aktivitäten-Rollen-Zuordnungen bei komplexen Prozessen nicht derart offensichtlich sein müssen und unter Umständen nur schwer erkannt werden können.

1-step Strict Static Separation of Duty

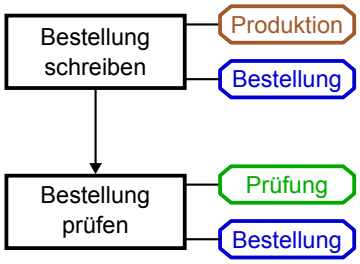
Um ein zuverlässiges Separation of Duty auf einer Menge von Aktivitäten zu ermöglichen, wird SSoD in [14] zu einem strengeren Modell namens **1-step Strict Static Separation of Duty** (**1sSSSoD**) erweitert. Übertragen auf den Workflow-Kontext werden dabei die folgenden, zu SSoD zusätzlichen, Einschränkungen gefordert:

- zwei streng exklusive Rollen dürfen nicht für dieselben Aktivitäten aus der gegebenen Menge berechtigen
- jede Rolle darf höchstens für eine Aktivität aus der gegebenen Menge berechtigen

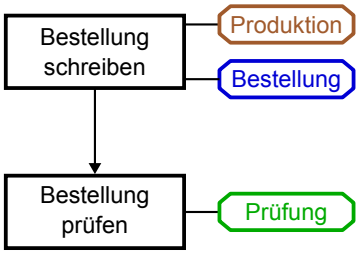
Das bedeutet, dass bei 1sSSSoD neben den Benutzern auch die Aktivitäten zweier streng exklusiver Rollen disjunkt sein müssen. Zusätzlich wird ausgeschlossen, dass ein gewünschtes Separation of Duty auf Aktivitäten durch entsprechende Aktivitäten-Rollen-Zuordnungen unterwandert wird. Dies wird dadurch bewerkstelligt, dass eine Rolle höchstens für eine der Aktivitäten berechtigen darf, die Gegenstand des SoD-Constraints sind.

Damit setzt 1sSSSoD, durch die Einschränkung der statischen Zuordnungen, ein zuverlässiges Separation of Duty um, das vollkommen zur Modellierzeit durchgesetzt werden kann.

1-step Strict Static Separation of Duty im Fallbeispiel: Mit 1sSSSoD kann erzwungen werden, dass die Aktivitäten „Bestellung schreiben“ und „Bestellung prüfen“ von unterschiedlichen Benutzern ausgeführt werden. Die Rollen „Bestellung“, „Prüfung“ und „Produktion“ sind hierzu erneut streng exklusiv, wie in Abb. 3.6 gezeigt. Da bei 1sSSSoD die Zuordnung einer Rolle zu mehr als einer der spezifizierten Aktivitäten nicht erlaubt ist, darf die Rolle „Bestellung“ nicht für beide Aktivitäten berechtigen. Abb. 3.8a zeigt daher eine im Zuge von 1sSSSoD nicht erlaubte Aktivitäten-Rollen-Zuordnung. Wird diese Zuordnung derart geändert, dass die Rolle „Bestellung“ nur noch für „Bestellung schreiben“ berechtigt (Abb. 3.8b), so wird die weitere Forderung nach der Disjunktheit der Aktivitäten zweier streng exklusiver Rollen nicht eingehalten. Dieser Konflikt kann dadurch aufgelöst werden, dass die Rolle „Produktion“ nicht mehr länger für „Bestellung schreiben“ berechtigt (Abb. 3.9). Alternativ können die Forderungen von 1sSSSoD dadurch erfüllt werden, dass die strenge Exklusivität der Rollen „Produktion“ und „Bestellung“ aufgelöst wird (Abb. 3.10).



(a) Nicht erlaubte Aktivitäten-Rollen-Zuordnung bei 1sSSSoD.



(b) Nicht erlaubte Aktivitäten-Rollen-Zuordnung bei 1sSSSoD.

Abb. 3.8: Nicht erlaubte Aktivitäten-Rollen-Zuordnungen bei 1sSSSoD und strenger Exklusivität der Rollen „Bestellung“, „Produktion“ und „Prüfung“.

Abb. 3.9: Erlaubte Aktivitäten-Rollen-Zuordnung bei 1sSSSoD und strenger Exklusivität der Rollen „Bestellung“, „Produktion“ und „Prüfung“.

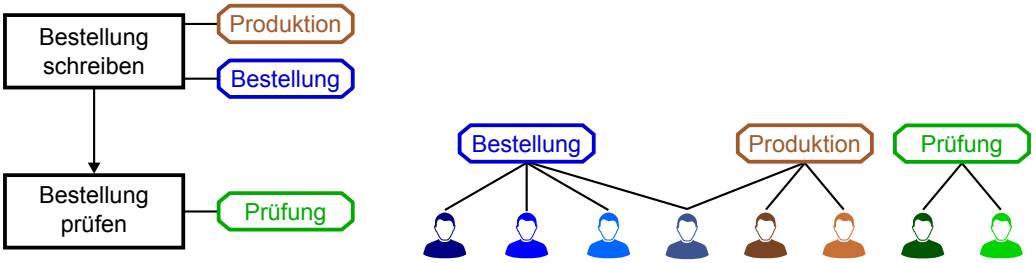


Abb. 3.10: Erlaubte Aktivitäten-Rollen-Zuordnung bei 1sSSSoD, bei Auflösung der strengen Exklusivität zwischen den Rollen „Bestellung“ und „Produktion“.

Zusammenfassung

Wie gesehen sind SSoD und 1sSSSoD mächtige Instrumente, um Interessenkonflikte, Betrug und unbeabsichtigte Fehler auszuschließen. Derartige (1sS)SSoD-Constraints beziehen sich ausschließlich auf die mittels RBAC vergebenen Berechtigungen zwischen Benutzern, Rollen und Aktivitäten. Sie müssen daher bereits zur Modellierzeit des Prozesses durchgesetzt werden.

Obwohl mit (1sS)SSoD ein Separation of Duty formulierbar ist, wird diese Herangehensweise als zu restriktiv angesehen [13, 18, 31]. So könnte zwar die gleichzeitige Verwendung zweier Rollen oder Aktivitäten durch einen Benutzer zu einem Sicherheitsrisiko führen, deren unabhängige Verwendung jedoch ungefährlich sein [11]. Mit (1sS)SSoD kann dem möglichen Sicherheitsrisiko bei gleichzeitiger Verwendung der Rollen oder Aktivitäten zwar vorgebeugt werden, jedoch ist dann auch deren unabhängige und ungefährliche Verwendung nicht mehr möglich. Dies stellt eine unnötige Einschränkung dar.

Die restriktive Art von (1sS)SSoD hat ihren Ursprung darin, dass es mit statischen Constraints nicht möglich ist, den bisherigen Verlauf der Prozessinstanz bei der Berechtigungsvergabe einzubeziehen. Das im Folgenden beschriebene Dynamische Separation of Duty umgeht diese restriktiven Ausprägungen von (1sS)SSoD.

3.4.2 Dynamisches Separation of Duty

Dynamisches Separation of Duty (**DSoD**; engl.: **Dynamic Separation of Duty**) ermöglicht die Formulierung weit weniger restriktiver Constraints als es mit SSoD möglich ist. So kann ein Benutzer mehreren Rollen zugeordnet werden, die bei gemeinsamer Inanspruchnahme einen Interessenkonflikt erzeugen würden. Werden diese Rollen unabhängig voneinander aktiviert, so liegt jedoch kein Interessenkonflikt vor [23]. Solche Rollen werden auch als **schwach exklusiv** bezeichnet [28].

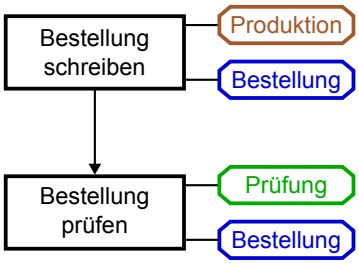
Mit SSoD ist die Formulierung solcher Sicherheitsanforderungen nicht möglich, da dort nur das Prinzip der streng exklusiven Rollen Verwendung findet. Die Zuordnung eines Benutzers zu zwei streng exklusiven Rollen ist bei SSoD verboten. Wie auch SSoD hat DSoD zum Ziel, die Zuordnung zwischen Benutzern und Aktivitäten einzuschränken. DSoD und SSoD unterscheiden sich jedoch darin, wann und in welchem Umfang diese Einschränkungen auferlegt werden.

Bei DSoD wird im Gegensatz zu SSoD nicht die statische Zuordnung von Rollen zu Benutzern oder Aktivitäten eingeschränkt. Die schwache Exklusivität zweier Rollen wird dadurch ermöglicht, dass die Aktivierung der Rollen eingeschränkt wird [13, 23]. Die Zuordnung eines Benutzers zu einer Rolle bedeutet im Rahmen von DSoD also nicht automatisch, dass diese Rolle auch jederzeit aktiviert werden kann. Ob dies möglich ist, hängt vom konkreten Verlauf der jeweiligen Prozessinstanz und den bis dahin aktivierten Rollen des jeweiligen Benutzers ab.

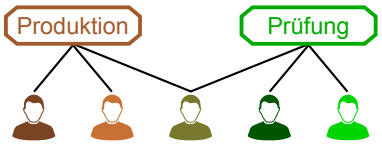
Simple Dynamic Separation of Duty (**SDSoD**) besagt, dass ein Benutzer eine zusätzliche Rolle nur dann aktivieren kann, sofern diese mit keiner anderen bereits durch ihn aktivierten Rolle schwach exklusiv ist [11]. Das bedeutet, dass in einer Prozessinstanz zu keinem Zeitpunkt ein Benutzer zwei Rollen aktiviert hat, die gemäß der vorliegenden Sicherheitsrichtlinie schwach exklusiv zueinander sind [14].

Simple Dynamic Separation of Duty im Fallbeispiel: Die Rollen „Produktion“ und „Prüfung“ seien schwach exklusiv. Damit ist es möglich, dass Benutzer beiden Rollen zuge-

ordnet sind (Abb. 3.11b). Diesen ist es jedoch nicht gestattet, beide Rollen gleichzeitig zu aktivieren. Damit ist eine flexiblere Ausführung der Aktivitäten „Bestellung schreiben“ und „Bestellung prüfen“ (Abb. 3.11a) möglich, da Benutzer sowohl in der Rolle „Produktion“ als auch in der Rolle „Prüfung“ aktiv werden können. Dabei herrscht lediglich die Einschränkung, dass ein Benutzer nicht beide Rollen gleichzeitig aktivieren darf.



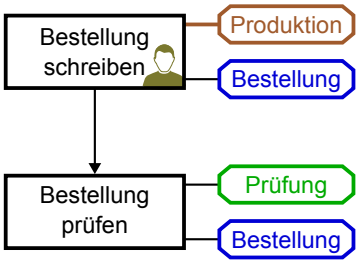
(a) Prozessausschnitt mit „Bestellung schreiben“ und „Bestellung prüfen“.



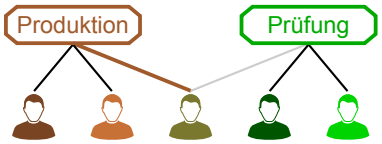
(b) Schwach exklusive Rollen „Produktion“ und „Prüfung“.

Abb. 3.11: Prozessausschnitt und schwach exklusive Rollen.

Entspricht der Gültigkeitszeitraum der Sessions, und damit der Gültigkeitszeitraum der Aktivierungen, der Dauer der jeweiligen Prozessinstanz, so kann ein Benutzer während der gesamten Warenbestellung höchstens eine der Rollen „Produktion“ oder „Prüfung“ aktivieren. Führt einer der Benutzer die Aktivität „Bestellung schreiben“ in der Rolle „Produktion“ aus (Abb. 3.12a), so ist dieser nicht mehr in der Lage die Rolle „Prüfung“ zu aktivieren (Abb. 3.12b). Die Aktivität „Bestellung prüfen“ kann somit nicht von demselben Benutzer in der Rolle „Prüfung“ ausgeführt werden.



(a) Ausführung von „Bestellung schreiben“.



(b) Aktivierung der Rolle „Produktion“.

Abb. 3.12: Ausführung von „Bestellung schreiben“ durch einen Benutzer in der Rolle „Produktion“.

Wie Abb. 3.11a entnommen werden kann, erlaubt DSoD die Zuordnung der Rolle „Bestellung“ zu beiden Aktivitäten „Bestellung schreiben“ und „Bestellung prüfen“. Somit ist denkbar, dass ein Benutzer der Rolle „Bestellung“ beide Aktivitäten ausführt. Ein Separation of Duty auf den beiden Aktivitäten ist somit nicht gewährleistet. Um tatsächlich ein Separation of Duty auf den beiden Aktivitäten zu erzwingen, ist also, ähnlich dem 1sSSSoD, die Forderung der Disjunktheit der Rollen der beiden Aktivitäten notwendig.

Zusätzlich zu den vorgestellten statischen und dynamischen Separation of Duty Constraints, werden in der Literatur viele weitere, oft sehr spezielle, SoD-Constraints diskutiert [14]. Auf deren weitere Betrachtung wird an dieser Stelle jedoch verzichtet, da die wichtigsten Aussagen in WfMS bereits mit den vorgestellten SoD-Constraints getroffen werden können.

3.4.3 Vier-Augen-Prinzip

Das **Vier-Augen-Prinzip** (**4Eyes**) ist ein Spezialfall des SoD-Konzeptes [34]. Es besagt, dass wichtige, fehleranfällige oder folgenschwere Aktivitäten von zwei Benutzern durchgeführt werden müssen. Demnach muss beim Vier-Augen-Prinzip eine solche Aktivität innerhalb einer Prozessinstanz zweimal von unterschiedlichen Benutzern ausgeführt werden. Durch die Mehrfachausführung der Aktivität wird somit das Risiko verringert, dass bei dieser Aktivität Fehler auftreten bzw. unerkannt bleiben.

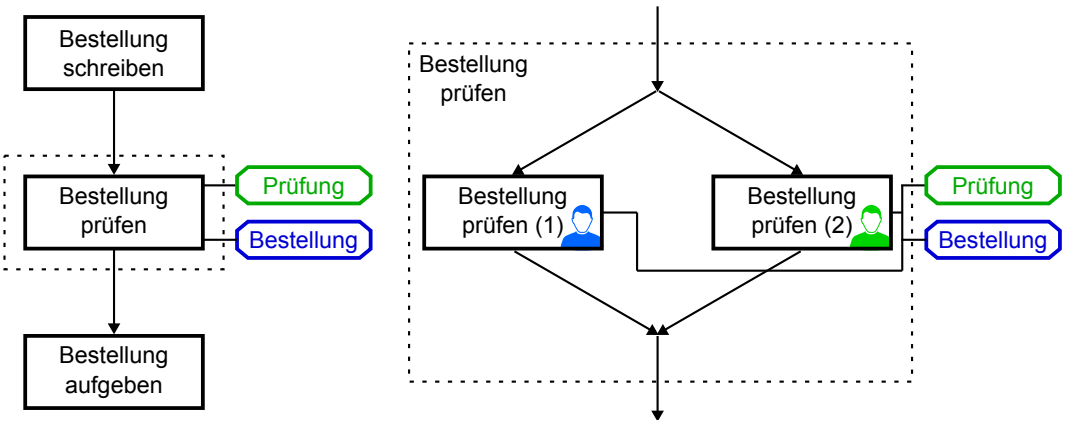


Abb. 3.13: Vier-Augen-Prinzip: Die Aktivität „Bestellung prüfen“ wird durch einen Subprozess mit zwei Ausprägungen der Aktivität ersetzt. Diese werden von zwei unterschiedlichen Benutzern ausgeführt.

Vier-Augen-Prinzip im Fallbeispiel: Die Aktivität „Bestellung prüfen“ soll bei Bestellungen ab einer bestimmten Größenordnung mehrmals von unterschiedlichen Benutzern durchgeführt werden. So soll sichergestellt werden, dass bei Bestellungen in größerem Umfang keine Fehler unterlaufen. Um dies zu bewerkstelligen, wird die Aktivität „Bestellung prüfen“ durch einen Subprozess, wie in Abb. 3.13 gezeigt, ersetzt. In diesem Subprozess werden zwei Ausprägungen dieser Aktivität parallel ausgeführt. Gemäß der Spezifikation des Vier-Augen-Prinzips müssen diese beide Ausprägungen von zwei unterschiedlichen Benutzern ausgeführt werden.

3.4.4 Binding of Duty

Auch **Binding of Duty** (**BoD**) kann als eine Variante von SoD angesehen werden [34]. Ähnlich den SoD Constraints, schränken auch BoD Constraints die Menge der Benutzer ein, die zur Ausführung einer bestimmten Aktivität berechtigt sind [8]. Im Gegensatz zu SoD fordern BoD Constraints jedoch, dass zwei oder mehr Aktivitäten eines Prozesses von demselben Benutzer ausgeführt werden [29, 31]. Eine solche Forderung kann beispielsweise gewünscht sein, wenn ein Benutzer durch Ausführung einer früheren Aktivität Kenntnisse erlangt, die bei nachfolgenden Aktivitäten nützlich oder erforderlich sind. Weiter kann so erreicht werden, dass sensible Informationen, die unter Umständen bei mehreren Aktivitäten anfallen, nur einem Benutzer bekannt werden.

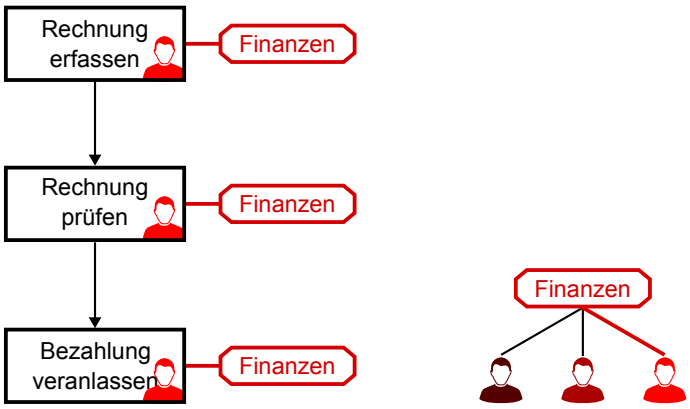


Abb. 3.14: Binding of Duty: Die Aktivitäten „Rechnung erfassen“, „Rechnung prüfen“ und „Bezahlung veranlassen“ werden von demselben Benutzer ausgeführt.

Binding of Duty im Fallbeispiel: Die Aktivitäten „Rechnung erfassen“, „Rechnung prüfen“ und „Bezahlung veranlassen“ sollen von demselben Benutzer ausgeführt werden. Somit ist nur ein Benutzer pro Prozessinstanz in die Bearbeitung der Rechnung einbezogen. Die Rechnungsdaten sind daher nur einem Benutzer zugänglich und die Verantwortlichkeit ist klar geregelt. Sobald einer der Benutzer der Rolle „Finanzen“ die Aktivität „Rechnung erfassen“ ausführt, muss dieser auch die beiden anderen Aktivitäten ausführen. Alle anderen Benutzer sind dann in dieser Prozessinstanz nicht zur Ausführung der Aktivitäten „Rechnung prüfen“ und „Bezahlung veranlassen“ berechtigt. Abb. 3.14 zeigt die Benutzer der Rolle „Finanzen“ und den entsprechenden Prozessausschnitt aus dem Fallbeispiel. Dabei werden die drei spezifizierten Aktivitäten von demselben Benutzer ausgeführt.

3.5 Cardinality Constraints

Wie der Name bereits andeutet, ermöglichen **Cardinality Constraints** (**CC**) die Formulierung von Einschränkungen, die sich darauf beziehen, wie oft eine bestimmte Aktion geschehen darf oder muss. Mit Cardinality Constraints steht ein Mechanismus zur Verfügung, der es ermöglicht, dass sensible Zugriffsrechte keiner großen Anzahl an Benutzern zur Verfügung stehen [9]. Die in der Literatur [7, 8, 9, 10, 11, 15, 28] beschriebenen Cardinality Constraints lassen sich dabei in zwei verschiedene Arten unterteilen: Rollen-Cardinality-Constraints und Aktivitäten-Cardinality-Constraints.

3.5.1 Rollen-Cardinality-Constraints

Rollen-Cardinality-Constraints (**RoleCC**) beziehen sich darauf, wie oft eine Rolle zu einem oder mehreren Benutzern zugeordnet oder durch diese aktiviert werden darf oder muss.

Im einfachsten Fall bezieht sich ein Rollen-Cardinality-Constraint auf die statische Benutzer-Rollen-Zuordnung. Eine Rolle hat hierbei eine maximal erlaubte Anzahl an Benutzerzuordnungen [10, 11]. Ein neuer Benutzer kann der Rolle nur dann zugewiesen werden, wenn die maximale Anzahl an Benutzerzuordnungen noch nicht erreicht wurde [11]. Ebenso ist es denkbar, dass einer Rolle eine Mindestanzahl an Benutzern zugeordnet werden muss.

Statischer Rollen-Cardinality-Constraint im Fallbeispiel: Da die Rolle „Prüfung“ auch zum Prüfen von Bestellungen mit enormen Größenordnungen berechtigt, soll die maximale

Anzahl der ihr zugeordneten Benutzer eingeschränkt werden. In Abb. 3.15 verbietet ein Rollen-Cardinality-Constraint die Zuordnung eines dritten Benutzers zur Rolle „Prüfung“. Mit Hilfe eines solchen Rollen-Cardinality-Constraints kann sichergestellt werden, dass die Anzahl der Benutzer mit weitreichenden Berechtigungen übersichtlich bleibt.

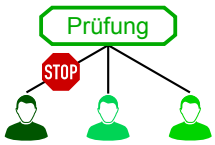


Abb. 3.15: Statischer Rollen-Cardinality-Constraint: Die Zuordnung eines dritten Benutzers zur Rolle „Prüfung“ ist nicht erlaubt.

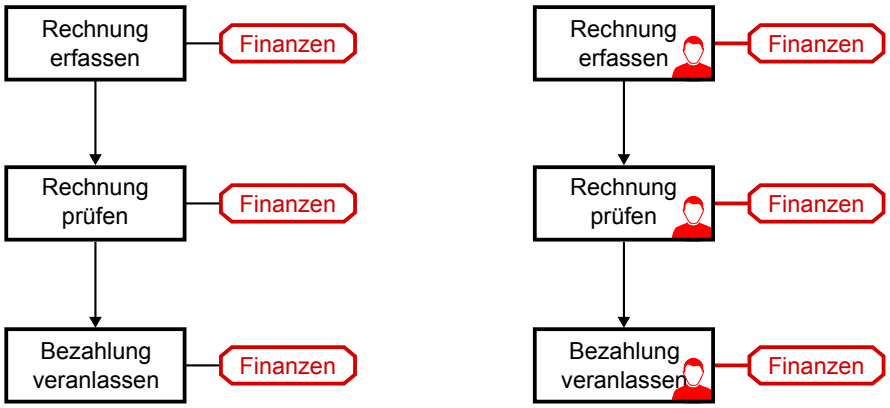
Darüber hinaus können sich Rollen-Cardinality-Constraints auf die Aktivierung von Rollen beziehen. Damit wird es möglich, der Rolle zugeordneten Benutzern das Aktivieren dieser Rolle zu untersagen [28]. In [15] wird diese Art von Rollen-Cardinality-Constraints in zwei Typen unterschieden: n Gesamtaktivierungen und maximal n gleichzeitige Aktivierungen.

Bei n **Gesamtaktivierungen** wird eine Rolle derart eingeschränkt, dass diese insgesamt höchstens n mal aktiviert werden kann. Hierbei kann weiter unterschieden werden, ob sich die Gesamtaktivierungen auf alle Benutzer gemeinsam beziehen, oder ob es jedem zugeordneten Benutzer erlaubt ist, diese Rolle n mal zu aktivieren. In letzterem Fall ist es sogar denkbar, dass verschiedenen Benutzern unterschiedliche erlaubte Grenzwerte zugeordnet werden. [15]

n **Gesamtaktivierungen im Fallbeispiel:** Auf den Aktivitäten „Rechnung erfassen“, „Rechnung prüfen“ und „Bezahlung veranlassen“ in Abb. 3.16a kann mit n Gesamtaktivierungen ein Binding of Duty formuliert werden. Dieser Constraint kann durch einen Rollen-Cardinality-Constraint wie folgt durchgesetzt werden:

- Die Rolle „Finanzen“ darf höchstens ein Mal von allen Benutzern gemeinsam aktiviert werden.

Grundsätzlich kommen für die Ausführung der drei Aktivitäten alle in Abb. 3.17a gezeigten Benutzer der Rolle „Finanzen“ in Frage. Aktiviert jedoch einer dieser Benutzer die Rolle „Finanzen“, so muss dieser alle drei Aktivitäten „Rechnung erfassen“, „Rechnung prüfen“ und „Bezahlung veranlassen“ ausführen (Abb. 3.16b). Da gemäß diesem Constraint höchstens eine Aktivierung der Rolle zulässig ist, darf kein weiterer Benutzer die Rolle „Finanzen“ aktivieren (Abb. 3.17b). Somit ist nur ein Benutzer in der Lage alle drei Aktivitäten auszuführen.



(a) Prozessausschnitt mit den Aktivitäten.

(b) Ausführung der beiden Aktivitäten durch denselben Benutzer.

Abb. 3.16: Rollen-Cardinality-Constraint: Binding of Duty auf den Aktivitäten „Rechnung erfassen“, „Rechnung prüfen“ und „Bezahlung veranlassen“.

(a) Benutzer der Rolle „Finanzen“.

(b) Aktivierung der Rolle durch einen Benutzer und resultierendes Verbot für die anderen Benutzer.

Abb. 3.17: Rollen-Cardinality-Constraint: Durchsetzung des Constraints nach Aktivierung der Rolle „Finanzen“ durch einen Benutzer.

Maximal n gleichzeitige Aktivierungen schränkt dagegen die Anzahl der gleichzeitigen Aktivierungen einer Rolle ein. Zu jedem Zeitpunkt kann die Rolle daher höchstens n mal aktiviert sein. [15]

Wie in [28] angedeutet wird, ist es darüber hinaus zusätzlich denkbar, dass Rollen-Cardinality-Constraints auch Untergrenzen für die Anzahl an Benutzerzuordnungen, Gesamtaktivierungen oder gleichzeitigen Aktivierungen festlegen.

Zusätzlich zu der Einschränkung der Rollenzuordnungen oder -aktivierungen ist es auch möglich, Cardinality Constraints für Aktivitäten zu formulieren. Diese Art der Cardinality Constraints wird im Folgenden betrachtet.

3.5.2 Aktivitäten-Cardinality-Constraints

Aktivitäten-Cardinality-Constraints (**TaskCC**) legen nach [8] fest, wie oft eine bestimmte Aktivität ausgeführt werden darf oder muss. Dazu wird jeder Aktivität ein Intervall $[n, m]$ zugeordnet. Die Aktivität muss dann innerhalb einer Prozessinstanz mindestens n mal und höchstens m mal ausgeführt werden. So besagt das Intervall $[0, 1]$, dass die entsprechende Aktivität höchstens einmal ausgeführt werden kann (optionale Aktivität), während $[2, \infty)$ bedeutet, dass die Aktivität mindestens zweimal ausgeführt werden muss. Zu beachten ist, dass durch Aktivitäten-Cardinality-Constraints keine Einschränkung bezüglich der Rolle gemacht wird, in der eine Aktivität ausgeführt wird.

Aktivitäten-Cardinality-Constraints können nach [7] zusätzlich dahingehend erweitert werden, dass eine Aktivität von mindestens k verschiedenen Benutzern ($k \leq m$) ausgeführt werden muss.

Aktivitäten-Cardinality-Constraint im Fallbeispiel: Mit einem Aktivitäten-Cardinality-Constraint kann ausgedrückt werden, dass die Aktivität „Bestellung prüfen“ genau zwei Mal ausgeführt werden muss. Formuliert man zusätzlich die Forderung dass dies von zwei unterschiedlichen Benutzern geschehen muss, so entspricht dieser Constraint dem Vier-Augen-Prinzip (vgl. Kapitel 3.4.3). Eine solche Mehrfachausführung der Aktivität “Bestellung prüfen“ durch unterschiedliche Benutzer ist beispielsweise sinnvoll wenn es sich um eine Bestellung mit einem großen Umfang handelt. So kann die Wahrscheinlichkeit von Fehlern oder Missbrauch durch Benutzer reduziert werden.

3.6 Weitere Authorization Constraints

Zusätzlich existieren viele weitere Authorization Constraints, die jedoch im weiteren Verlauf nicht weiter betrachtet werden. Der Fokus liegt im Folgenden auf den bisher betrachteten Constraints. Mit diesen kann bereits ein beträchtlicher Teil der gewünschten Sicherheitsanforderungen umgesetzt werden. Der Vollständigkeit halber wird jedoch an dieser Stelle auf folgende weitere Authorization Constraints hingewiesen:

Inter-Instance Constraints [33] beziehen sich auf mehrere Prozessinstanzen und werden auch in diesem Umfang erzwungen. Alle bisher betrachteten Constraints waren dagegen **Intra-Instance Constraints** [33], also Constraints, die innerhalb einzelner Prozessinstanzen Gültigkeit besitzen und umgesetzt werden. Inter-Instance Constraints sind dabei nicht notwendigerweise auf ein Prozessschema beschränkt, sondern können sich auch auf mehrere Instanzen unterschiedlicher Prozessschemas beziehen. Inter-Instance Constraints ermöglichen in erster Linie die Formulierung von Separation of Duty Constraints über mehrere Prozessinstanzen hinweg. Inter-Instance Constraints können dabei keine Intra-Instance Constraints ersetzen, diese jedoch ergänzen.

Mit sogenannten **Temporal Constraints** [3, 4, 15] ist es möglich, Berechtigungen zeitlich einzuschränken. So ist es vorstellbar, dass Rollen nur zu vordefinierten Zeitpunkten oder nur in bestimmten Zeitintervallen aktiviert werden können. Weitere Einschränkungen betreffen die zeitliche Beschränkung von Berechtigungen, so dass eine Rolle durch einen Benutzer beispielsweise maximal 30 Minuten aktiviert werden kann. Hierbei kann wiederum unterschieden werden, ob sich diese Zeitbeschränkung auf jede einzelne Aktivierung bezieht, oder ob Mehrfachaktivierungen derselben Rolle aggregiert werden.

3.7 Kategorisierung

Im Folgenden werden die Gemeinsamkeiten und Unterschiede der in den vorhergehenden Kapiteln betrachteten Constraints herausgearbeitet. Hierzu werden diese auf unterschiedliche Weisen kategorisiert.

3.7.1 Statische und dynamische Constraints

Eine erste übliche Kategorisierung der Constraints ist die Unterscheidung in statische und dynamische Constraints [5, 6, 19, 28]. Hierbei werden die Constraints dahingehend unterschieden, ob diese zur Modellierzeit oder zur Laufzeit des Prozesses durchgesetzt und analysiert werden können. Zusätzlich gibt es **hybride Constraints** [5], die jeweils teilweise zur Modellierzeit und zum Teil erst zur Laufzeit ausgewertet werden können.

Zu den zur Modellierzeit auswertbaren **statischen Constraints** gehören demnach:

- Statisches Separation of Duty (SSoD)
- 1-step Strict Static Separation of Duty (1sSSSoD)
- Statische Rollen-Cardinality-Constraints (SRoleCC)

Alle diese Constraints treffen Aussagen darüber, inwiefern Benutzer, Rollen und Aktivitäten mittels RBAC einander zugeordnet werden müssen oder dürfen. So fordert SSoD die Disjunktheit der Benutzermengen zweier Rollen, während 1sSSSoD zusätzlich die Disjunktheit der Rollen zweier Aktivitäten fordert. SRoleCC schränkt dagegen ein, wie oft eine Rolle zugeordnet werden darf oder muss.

Zu den **dynamischen Constraints** gehören dagegen:

- Simple Dynamic Separation of Duty (SDSoD)
- Vier-Augen-Prinzip (4Eyes)
- Binding of Duty (BoD)
- Aktivierungs-Rollen-Cardinality-Constraints (DRoleCC)
- Aktivitäten-Cardinality-Constraints (TaskCC)

Die Einhaltung dieser dynamischen Constraints kann erst zur Laufzeit des Prozesses erzwungen bzw. überwacht werden. So ist bei SDSoD erst zur Laufzeit bekannt, durch welchen Benutzer eine Aktivität ausgeführt wird und somit, welche Aktivität derselbe Benutzer im weiteren Verlauf nicht ausführen darf. Auch das Vier-Augen-Prinzip kann nur zur Laufzeit durchgesetzt werden, da für beide Ausprägungen der Aktivität unterschiedliche Benutzer erzwungen werden müssen, während ihnen statisch dieselben Benutzer zugeordnet sind. Ebenso kann BoD erst erzwungen werden, wenn eine der spezifizierten Aktivitäten durch einen Benutzer ausgeführt wurde. Auch bei den beiden Arten von Cardinality Constraints ist klar, dass diese erst zur Laufzeit durchgesetzt werden können, da auch die entsprechenden Aktivierungen der Rollen bzw. Ausführungen der Aktivitäten erst zur Laufzeit stattfinden können.

3.7.2 Entailment Constraints

Ein weiterer interessanter Indikator ist die Anzahl der Aktivitäten, auf die sich ein Constraint bezieht. Bezieht sich ein Constraint auf zwei Aktivitäten, so kann dieser, gemäß den Ausführungen in Kapitel 3.3, als Entailment Constraint bezeichnet werden. Zu dieser Art von Constraints gehören:

- 1-step Strict Static Separation of Duty (1sSSSoD)
- Simple Dynamic Separation of Duty (SDSoD)
- Binding of Duty (BoD)
- Vier-Augen-Prinzip (4Eyes)

1sSSSoD, SDSoD und BoD beziehen sich in ihrer Grundform jeweils auf zwei Aktivitäten, die entsprechend durch unterschiedliche (1sSSSoD, SDSoD) bzw. durch denselben (BoD) Benutzer ausgeführt werden müssen. Sie gehören somit zur Klasse der Entailment Constraints. Da Entailment Constraints dahingehend erweitert werden können, dass diese sich auf mehr als zwei Aktivitäten beziehen, können auch 1sSSSoD-, SDSoD- und BoD-Constraints mit mehr als zwei Aktivitäten als Entailment Constraint bezeichnet werden. Da beim Vier-Augen-Prinzip zwei Ausprägungen einer Aktivität von unterschiedlichen Benutzern ausgeführt werden müssen, kann auch dieses als Entailment Constraint angesehen werden.

3.7.3 Cardinality Constraints

Wie in Kapitel 3.5 beschrieben, beziehen sich Cardinality Constraints darauf, wie oft eine Rolle zugeordnet oder aktiviert werden muss oder darf, oder wie oft eine Aktivität ausgeführt werden muss. Gemäß ihrer Definition sind somit

- Statische Rollen-Cardinality-Constraints,
- Aktivierungs-Rollen-Cardinality-Constraints und
- Aktivitäten-Cardinality-Constraints

Cardinality Constraints. Ebenso wurde bereits in Kapitel 3.5.2 motiviert, dass das Vier-Augen-Prinzip als Aktivitäten-Cardinality-Constraint angesehen werden kann, da eine Aktivität hierbei zwei Mal ausgeführt werden muss.

Bei genauerer Betrachtung können auch

- Statisches Separation of Duty (SSoD)
- 1-step Strict Static Separation of Duty (1sSSSoD)
- Simple Dynamic Separation of Duty (SDSoD)
- Binding of Duty (BoD)

als Cardinality Constraints aufgefasst werden. So besagt SSoD, dass den spezifizierten Rollen keine (also genau 0) gemeinsamen Benutzer zugeordnet werden dürfen. 1sSSSoD und DSoD können dagegen insofern als Cardinality Constraint ausgedrückt werden, dass jeweils zwei Aktivitäten von genau zwei unterschiedlichen Benutzern ausgeführt werden müssen. Bei BoD müssen hingegen zwei Aktivitäten durch genau einen Benutzer ausgeführt werden.

Wie gesehen, können verschiedene Constraints in mehrere der Kategorien

- statische und dynamische Constraints,
- Entailment Constraints und
- Cardinality Constraints

eingeordnet werden. Die Zugehörigkeit der Constraints zu diesen Kategorien spielt im weiteren Verlauf dieser Arbeit eine zentrale Rolle.

Teil II

Formulierung, Durchsetzung und Validierung von Authorization Constraints

4 Voraussetzungen und Annahmen

Für die weiteren Betrachtungen in Teil II dieser Arbeit werden einige Voraussetzungen und Annehmen getroffen. Diese werden im Folgenden beschrieben. Zunächst wird auf die Berechtigungsvergabe mittels RBAC eingegangen. Dabei wird auf Rollenhierarchien und Rollen ohne Benutzerzuordnungen verzichtet. Außerdem wird die im Folgenden verwendete Notation für RBAC-Berechtigungen eingeführt. Anschließend werden die im weiteren Verlauf betrachteten Authorization Constraints motiviert.

4.1 Role-Based Access Control

RBAC bietet sehr flexible Möglichkeiten zur Vergabe von Berechtigungen an Benutzer und wird in vielen heutigen WfMS eingesetzt. Daher dient RBAC auch im weiteren Verlauf dieser Arbeit zur Berechtigungsvergabe und bildet so eine solide Basis zur Modellierung von Authorization Constraints. RBAC wurde zu diesem Zweck bereits in Kapitel 2.2 vorgestellt. Dennoch werden im Folgenden einige Besonderheiten und Einschränkungen des RBAC-Modells vorgestellt, die im weiteren Verlauf dieser Arbeit von Bedeutung sind. Damit wird die Komplexität der weiteren Betrachtungen verringert.

4.1.1 Verzicht auf Rollenhierarchie

Der Verzicht auf Rollenhierarchien (Kapitel 2.2.3) scheint zunächst eine große Einschränkung zu sein. Im Rahmen von WfMS repräsentieren Rollen Organisationseinheiten oder Tätigkeitsbereiche. Dabei ist davon auszugehen, dass die Organisationseinheiten zum größten Teil unterschiedliche Aufgaben erfüllen. Somit unterscheiden sich auch die benötigten Berechtigungen der verschiedenen Rollen in weiten Teilen. Auf eine Vererbung von Berechtigungen mittels Rollen wird unter dieser Annahme verzichtet. Auf diese Weise verringert sich auch die Komplexität der weiteren Betrachtungen.

4.1.2 Verzicht auf Rollen ohne Benutzerzuordnung

Eine weitere Forderung ist der Verzicht auf Rollen ohne zugeordnete Benutzer (**leere Rollen**). Derartige Rollen würden zwar für Aktivitäten berechtigen, jedoch ohne dass diese durch Benutzer in Anspruch genommen werden könnten. Leere Rollen haben somit keine Daseinsberechtigung, sodass als Konsequenz jeder Rolle mindestens ein Benutzer zugeordnet sein muss.

4.1.3 Umsetzung und Notation von RBAC-Berechtigungen

Da RBAC auf unterschiedliche Weisen umgesetzt werden kann, wird im Rahmen dieser Arbeit von der konkreten Umsetzung abstrahiert. Die RBAC-Berechtigungsvergabe wird in Form von Abbildungen erfolgen. Dabei finden grafische Zuordnungen zwischen Aktivitäten und Rollen statt. Eine solche grafische Zuordnung hat die Semantik, dass die entsprechenden Rollen zur Ausführung der jeweiligen Aktivitäten berechtigen. Zusätzlich werden, ebenfalls in Form von Abbildungen, den Rollen Benutzer zugeordnet. Diese Benutzer sind zur Inanspruchnahme der Rolle berechtigt und somit in der Lage, die der Rolle zugeordneten Aktivitäten auszuführen.

Zusätzlich wird eine einfache Schreibweise für die Ermittlung der jeweils gültigen RBAC-Berechtigungen eingeführt. Die folgenden Funktionen haben jeweils eine Menge als Ergebnis. Dabei steht

- $roles(user)$ für alle Rollen, denen der Benutzer $user$ zugeordnet ist
- $roles(task)$ für alle Rollen, die für die Aktivität $task$ berechtigen
- $users(role)$ für alle Benutzer, die der Rolle $role$ zugeordnet sind
- $users(task)$ für alle Benutzer, die mittels mindestens einer Rolle zur Ausführung der Aktivität $task$ berechtigt sind
- $tasks(role)$ für alle Aktivitäten, die in der Rolle $role$ ausgeführt werden können
- $tasks(user)$ für alle Aktivitäten, für die der Benutzer $user$ mittels mindestens einer Rolle zur Ausführung berechtigt ist
- $allRoles$ für alle in der aktuellen Prozessinstanz vorkommenden Rollen
- $allUsers$ für alle in der aktuellen Prozessinstanz vorkommenden Benutzer
- $allTasks$ für alle in der aktuellen Prozessinstanz vorkommenden Aktivitäten

Dabei gilt mit den Operatoren der Mengenlehre:

- $users(task) = \bigcup_{r \in roles(task)} users(r)$
- $tasks(user) = \bigcup_{r \in roles(user)} tasks(r)$

4.2 Wichtigste Constraints

In Kapitel 3 wurden bereits mehrere unterschiedliche Authorization Constraints vorgestellt. Die weiteren Betrachtungen in Teil II dieser Arbeit beschränken sich jedoch auf die in WfMS wichtigsten Authorization Constraints. Im Folgenden wird motiviert, welche der bereits betrachteten Authorization Constraints im Rahmen von WfMS von besonderer Bedeutung sind.

4.2.1 Separation of Duty (SoD)

Im weiteren Verlauf dieser Arbeit werden drei Varianten von SoD ausführlich betrachtet:

- Statisches Separation of Duty auf Rollen (RoleSSoD)
- Statisches Separation of Duty auf Aktivitäten (TaskSSoD)
- Dynamisches Separation of Duty auf Aktivitäten (TaskDSoD)

Statisches Separation of Duty auf Rollen (RoleSSoD)

RoleSSoD wurde bereits in Kapitel 3.4.1 als einfachste Form des Separation of Duty vorgestellt, wobei zwei oder mehr, sogenannte streng exklusive, Rollen keine gemeinsamen Benutzer haben dürfen. So kann es an vielen Stellen gewünscht sein, dass Benutzer nicht mehreren Organisationseinheiten, und somit Rollen, angehören dürfen. Auf diese Weise kann ausgeschlossen werden, dass Berechtigungen zusammengeführt werden, die keinesfalls auf einzelne Benutzer entfallen dürfen. Derartige Einschränkungen sind von einer sehr restriktiven Art, ermöglichen aber auf eine einfache Weise die Umsetzung erster und weitreichender Sicherheitsanforderungen.

Statisches Separation of Duty auf Aktivitäten (TaskSSoD)

Eine beliebte Forderung bei der Ausführung von Prozessen ist, dass zwei oder mehr Aktivitäten nicht von demselben Benutzer ausgeführt werden dürfen. Auf diese Weise sind Sicherheitsanforderungen durchsetzbar, die die Ausführung verschiedener Aktivitäten durch unterschiedliche Benutzer fordern. Mit TaskSSoD ist die Durchsetzung dieser Forderung auf eine sehr restriktive Art möglich: ein Benutzer darf höchstens für eine der spezifizierten Aktivitäten berechtigt sein. Gerade aufgrund der Restriktivität hat diese Art des Separation of Duty den Vorteil, dass die Einhaltung des Constraints bereits zur Modellierzeit des Prozesses sichergestellt werden kann. Es kann somit nicht passieren, dass dieser Constraint zur Laufzeit des Prozesses Konflikte verursacht.

Dynamisches Separation of Duty auf Aktivitäten (TaskDSoD)

Auch TaskDSoD begegnet Sicherheitsanforderungen, nach denen mehrere Aktivitäten durch unterschiedliche Benutzer ausgeführt werden müssen. Jedoch können bei TaskDSoD einzelne Benutzer prinzipiell zur Ausführung mehrerer Aktivitäten berechtigt sein. Erst zur Laufzeit des Prozesses wird sichergestellt, dass bei Ausführung einer der Aktivitäten die anderen nicht durch denselben Benutzer ausgeführt werden können.

TaskDSoD hat im Gegensatz zu TaskSSoD den Vorteil, dass es weit weniger restriktiv ist. Ein Benutzer wird dabei nicht bereits zur Modellierzeit zur Ausführung einer bestimmten der spezifizierten Aktivitäten verpflichtet. Gegenüber TaskSSoD hat dies jedoch den Nachteil, dass zur Laufzeit des Prozesses Konflikte auftreten können, die die weitere Ausführung des Prozesses unmöglich machen.

Unter Beachtung dieser Vor- und Nachteile von TaskSSoD und TaskDSoD muss also im jeweiligen Anwendungsfall entschieden werden, welche dieser Lösungen die bessere Alternative darstellt. Aus diesem Grund werden im weiteren Verlauf dieser Arbeit beide Varianten betrachtet.

4.2.2 Binding of Duty (BoD)

Binding of Duty (**BoD**) wurde bereits in Kapitel 3.4.4 vorgestellt und fordert, dass zwei oder mehr Aktivitäten durch denselben Benutzer ausgeführt werden müssen. Eine solche

Forderung ist beispielsweise notwendig, wenn eine Aktivität nur erfolgreich ausgeführt werden kann, wenn der Benutzer hierzu Kenntnisse aus einer früheren Aktivität benötigt. Ein weiteres Beispiel für den Einsatz von BoD wäre, dass sensible Informationen, die bei der Ausführung mehrerer Aktivitäten anfallen, nur einem Benutzer zugänglich sein sollen, um so die Gefahr des Missbrauchs dieser Informationen zu verringern.

4.2.3 Vier-Augen-Prinzip

Der Einsatz des **Vier-Augen-Prinzips** ist sinnvoll, wenn eine Aktivität mehrfach durch unterschiedliche Benutzer ausgeführt werden soll. Auf diese Weise können beabsichtigte oder unbeabsichtigte Fehler bei der Ausführung dieser Aktivität verhindert oder zumindest erschwert werden.

5 Constraint-Formulierung mit MΘnK

Ziel dieses Kapitels ist die einheitliche Modellierung der in Kapitel 4.2 motivierten Constraints mit MΘnK. Hierzu wird zunächst das MΘnK-Modell motiviert und dessen Funktionsweise erläutert. Anschließend erfolgt die Modellierung der Constraints mit Hilfe von MΘnK.

5.1 Motivation für MΘnK

Wie in Kapitel 4.2 gesehen, sollen in einem WfMS unterschiedliche Arten von Constraints umgesetzt werden können. Neben diesen wichtigsten Constraints ist es außerdem denkbar, dass auch deutlich komplexere Constraints Verwendung finden sollen. Alle diese Constraints adressieren unterschiedliche Sicherheitsaspekte und besitzen daher auch verschiedene Semantiken und Wirkungsweisen. So schränken manche der Constraints die per RBAC vergebbaren Berechtigungen ein, während andere die Inanspruchnahme der vergebenen Berechtigungen einschränken. Ob und wann diese Einschränkungen tatsächlich gültig sind, kann dabei wiederum von Kontextinformationen des Prozesses abhängig sein.

Obwohl also große Unterschiede zwischen den gewünschten Constraints bestehen, sollen diese dennoch innerhalb eines WfMS formuliert und später durchgesetzt und validiert werden. Dieser Herausforderung begegnet **MΘnK**. MΘnK ist dabei ein Modell zur Spezifikation von Constraints und wurde in Anlehnung an [27] entworfen. MΘnK beruht auf der Beobachtung, dass sich alle der gewünschten Constraints als Cardinality Constraints (Kapitel 3.5) formulieren lassen.

Ein Statisches Separation of Duty auf den beiden Rollen „Bestellung“ und „Finanzen“ im Fallbeispiel lässt sich beispielsweise formulieren mit „Jeder Benutzer darf höchstens einer der beiden Rollen zugeordnet werden“. Mit dieser Forderung in Form eines Cardinality

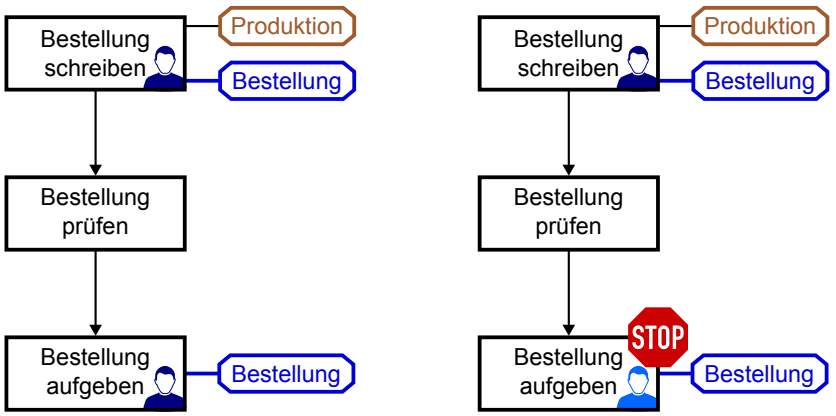
(a) Erlaubte Benutzerzuordnung: Jeder Benutzer ist höchstens einer der beiden Rollen zugeordnet.

(b) Verbotene Benutzerzuordnung: Ein Benutzer ist mehr als einer der beiden Rollen zugeordnet.

Abb. 5.1: Erlaubte und verbotene Benutzerzuordnung bei statischem Separation of Duty auf den beiden Rollen „Bestellung“ und „Finanzen“.

Constraints sind also die Benutzermengen der beiden Rollen disjunkt. Abb. 5.1a zeigt entsprechend eine erlaubte Benutzerzuordnung, während Abb. 5.1b die verbotene Zuordnung eines Benutzers zu beiden Rollen zeigt.

Ein Binding of Duty lässt sich dagegen formulieren mit „Die Aktivitäten ‚Bestellung schreiben‘ und ‚Bestellung aufgeben‘ müssen von genau einem Benutzer ausgeführt werden“. In Abb. 5.2a ist eine erlaubte Ausführung der beiden Aktivitäten durch denselben Benutzer abgebildet, während Abb. 5.2b die verbotene Ausführung durch zwei unterschiedliche Benutzer zeigt.



(a) Erlaubte Ausführung: Beide Aktivitäten werden durch denselben Benutzer ausgeführt.

(b) Verbotene Ausführung: Die beiden Aktivitäten werden durch mehr als einen Benutzer ausgeführt.

Abb. 5.2: Erlaubte und verbotene Ausführung der Aktivitäten „Bestellung schreiben“ und „Bestellung aufgeben“ bei Binding of Duty.

Durch MΘnK wird also eine einheitliche Modellierung der gewünschten Constraints in Form von Cardinality Constraints ermöglicht. Mit Hilfe dieser einheitlichen Modellierung wird die

spätere Durchsetzung und Validierung der verschiedenartigen Constraints auf eine einheitliche Weise möglich.

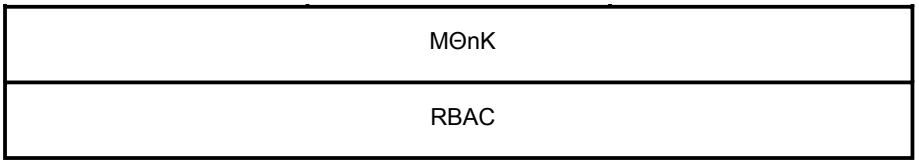


Abb. 5.3: MΘnK basiert auf einer Berechtigungsvergabe mittels RBAC.

MΘnK wird hierzu eine Berechtigungsvergabe mittels RBAC zugrunde gelegt. Dabei ist es unerheblich, wie diese konkret umgesetzt wird. Dieser Zusammenhang wird auch in Abb. 5.3 verdeutlicht. Zu beachten ist außerdem, dass MΘnK die wichtigsten Eigenschaften eines Modells zur Spezifikation von Constraints, wie in Kapitel 3.1.2 beschrieben, aufweist. An dieser Stelle sei besonders darauf hingewiesen, dass MΘnK eine große Ausdrucksmächtigkeit besitzt, mit der viele unterschiedliche Constraints modelliert werden können. Durch die einheitliche Modellierung der verschiedenartigen Constraints wird in Kapitel 6 und Kapitel 7 deren einheitliche Durchsetzung und Validierung möglich.

Im Folgenden wird die Funktionsweise von MΘnK und dessen Verwendung als Modell zur Spezifikation von Constraints verdeutlicht.

5.2 Funktionsweise von MΘnK

Das MΘnK-Modell besteht aus sogenannten MΘnK-Constraints. Im Folgenden wird zunächst die allgemeine Funktionsweise dieser MΘnK-Constraints vorgestellt. Anschließend werden diesen MΘnK-Constraints insgesamt drei verschiedene Semantiken in Form von assignMΘnK-Constraints, activateMΘnK-Constraints und entailmentMΘnK-Constraints zugewiesen.

5.2.1 Allgemeine Funktionsweise

Ein sogenannter **MΘnK-Constraint** entspricht einem 4-Tupel (M, Θ, n, K), wobei die einzelnen Elemente dieses Tupels die folgende Bedeutung haben:

- M ist eine nichtleere Menge bestehend aus Benutzern, Rollen oder Aktivitäten
- Θ ist ein Vergleichsoperator, $\Theta \in \{\leq, =, \geq\}$
- n ist eine natürliche Zahl, $n \in \mathbb{N}_0$
- K ist eine nichtleere Menge bestehend aus Benutzern, Rollen oder Aktivitäten

Ein solcher MΘnK-Constraint wird wie folgt notiert:

- $MonK(M, \Theta, n, K)$

Abb. 5.4 veranschaulicht einen ersten, abstrakten MΘnK-Constraint. Dabei sind

- m_i Elemente aus der Menge M,
- k_i Elemente aus der Menge K.

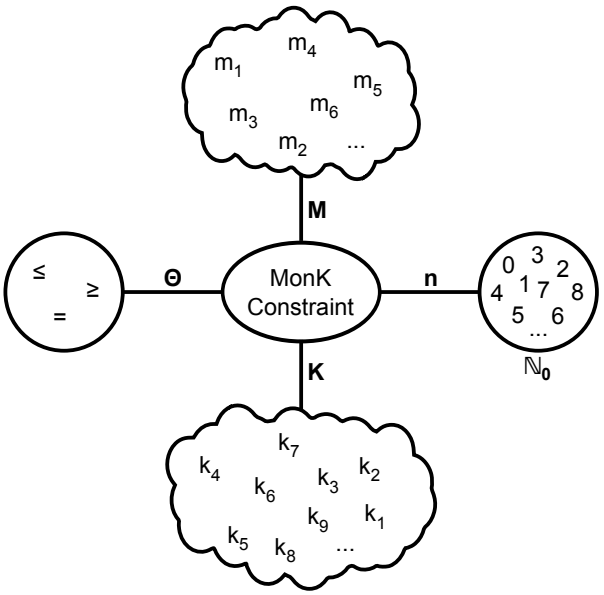


Abb. 5.4: Die Elemente eines MΘnK-Constraints

Durch die zugrunde gelegte RBAC-Berechtigungsvergabe existieren bereits Mengen von Benutzern, Rollen und Aktivitäten. Diese durch RBAC zur Verfügung gestellten Elemente werden auch innerhalb der MΘnK-Constraints verwendet.

Den Mengen M und K ist zunächst jeweils ein sogenannter **Typ** zugeordnet. Dieser typ_M bzw. typ_K gibt an, von welcher Art die Elemente der jeweiligen Menge M bzw. K sind. Zur Auswahl stehen hierfür die Typen „Benutzer“, „Rolle“ oder „Aktivität“. Ist also beispielsweise $typ_M =$ „Benutzer“, so befinden sich in der Menge M ausschließlich Benutzer. Ist dagegen $typ_K =$ „Rolle“, so enthält die Menge K ausschließlich Rollen.

Ein MΘnK-Constraint trifft dann durch Θ und n eine Aussage darüber, in welchem Verhältnis die Elemente der Menge M zu den Elementen der Menge K stehen. Die Semantik eines solchen MΘnK-Constraints ist wie folgt: Jedem Element $m \in M$ werden

- höchstens n viele Elemente aus K zugeordnet, falls $\Theta =$ ‘ $\leq$ ‘,
- genau n viele Elemente aus K zugeordnet, falls $\Theta =$ ‘ $=$ ‘,
- mindestens n viele Elemente aus K zugeordnet, falls $\Theta =$ ‘ $\geq$ ‘.

Im Folgenden ist die Schreibweise hierfür, dass jedem Element $m \in M$ Θ n viele Elemente aus K zugeordnet werden müssen.

Abb. 5.5 veranschaulicht eine solche Zuordnung für die folgenden Bestandteile eines MΘnK-Constraints:

- $M = \{m_1, m_2\}$
- $\Theta =$ ‘ $=$ ‘
- $n = 2$
- $K = \{k_1, k_2, k_3, k_4, k_5, k_6, k_7\}$

Die Notation dieses MΘnK-Constraints ist dabei wie folgt:

- $MonK(\{m_1, m_2\}, =, 2, \{k_1, k_2, k_3, k_4, k_5, k_6, k_7\})$.

Die Semantik dieses MΘnK-Constraints ist dann, dass sowohl m_1 als auch m_2 jeweils genau zwei Elemente aus der Menge K zugeordnet werden müssen.

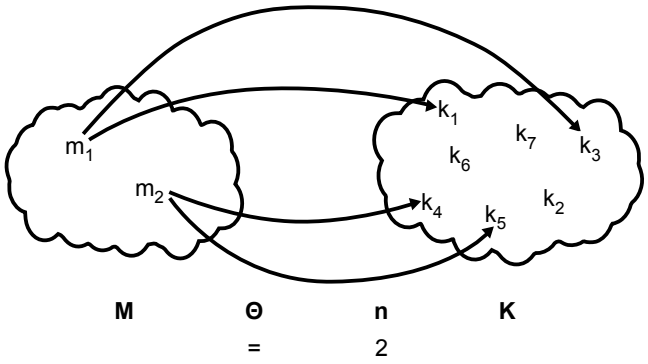


Abb. 5.5: Erlaubte Zuordnungen zwischen M und K für $\Theta =$ ‘ $=$ ‘ und $n = 2$.

Dabei ist zu beachten, dass jedes Element der Menge K höchstens einem Element der Menge M zugeordnet werden darf. Eine Mehrfachzuordnung der Elemente aus K an verschiedene Elemente aus M ist also nicht erlaubt. Ein Element $k \in K$ darf somit nicht zwei

Elementen $m_1, m_2 \in M$ ($m_1 \neq m_2$) zugeordnet werden. Soll eine solche Mehrfachzuordnung ausdrücklich erwünscht sein, so kann die Menge K als Multimenge aufgefasst werden, in der das entsprechende Element mehrfach enthalten ist. Die Menge K könnte dann wie folgt aussehen:

- $K = \{k_1, k_1, k_2, k_3, k_3, k_4, k_5\}$ bzw.
- $K = \{k_1^2, k_2^1, k_3^2, k_4^1, k_5^1\}$.

Dabei bedeutet k_i^j, dass das Element k_i genau j mal in der Menge K enthalten ist.

Ein MΘnK-Constraint trifft also immer eine Aussage darüber, welche „Verbindungen“ zwischen den Elementen der Menge M und den Elementen der Menge K bestehen. Daher ist es leicht ersichtlich, dass die Typen der Mengen M und K stets unterschiedlich sein müssen. Eine Aussage zwischen zwei Elementen desselben Typs macht keinen Sinn. Daher muss für jeden MΘnK-Constraint $typ_M \neq typ_K$ gelten.

Es ist außerdem wichtig zu erwähnen, dass die Mengen M und K auch durch Operationen der Mengenlehre gebildet werden können. Hiervon wird später häufig Gebrauch gemacht werden. Dabei werden insbesondere die folgenden Operatoren häufige Verwendung finden:

- $\cup$ und $\bigcup$ für die Vereinigung zweier bzw. mehrerer Mengen
- $\cap$ und $\bigcap$ für die Schnittmenge zweier bzw. mehrerer Mengen
- $A \backslash B$ für die Differenzmenge der beiden Mengen A und B
- $|A|$ für die Anzahl der Elemente in der Menge A

Die im Rahmen dieser Arbeit betrachteten Constraints formulieren sowohl Einschränkungen statischer als auch dynamischer Natur. Damit wird es nötig, auch den MΘnK-Constraint statische und dynamische Semantiken zuzuordnen. Ein MΘnK-Constraint kann daher entweder ein assignMΘnK-Constraint, ein activateMΘnK-Constraint oder ein entailmentMΘnK-Constraint sein. Abb. 5.6 zeigt den Zusammenhang zwischen diesen verschiedenen MΘnK-Constraints. Diese entsprechen in Form und Aufbau den vorherigen Ausführungen, besitzen jedoch unterschiedliche Semantiken. Diese werden im Folgenden spezifiziert.

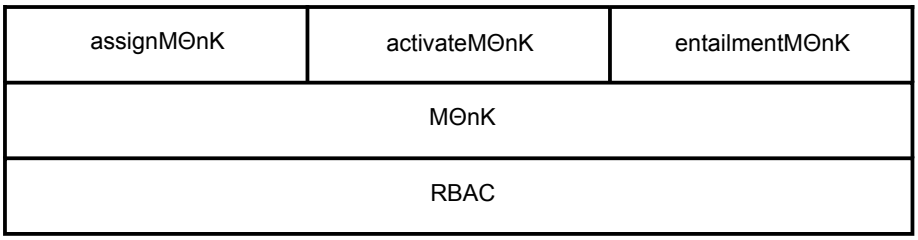


Abb. 5.6: Zusammenhang der MΘnK-Constraints mit unterschiedlicher Semantik.

5.2.2 assignMΘnK: MΘnK-Constraint mit statischer Zuordnungssemantik

assignMΘnK-Constraints (Abb. 5.7) sind MΘnK-Constraints mit einer statischen Zuordnungssemantik. Durch assignMΘnK-Constraints kann die statische Berechtigungsvergabe durch RBAC eingeschränkt werden. Die durch assignMΘnK-Constraints formulierten Einschränkungen sind also stets statischer Natur und schränken die mittels RBAC vergebbaren Berechtigungen ein. Somit müssen alle Berechtigungen die mittels RBAC an Benutzer vergeben werden den jeweiligen assignMΘnK-Constraints genügen.

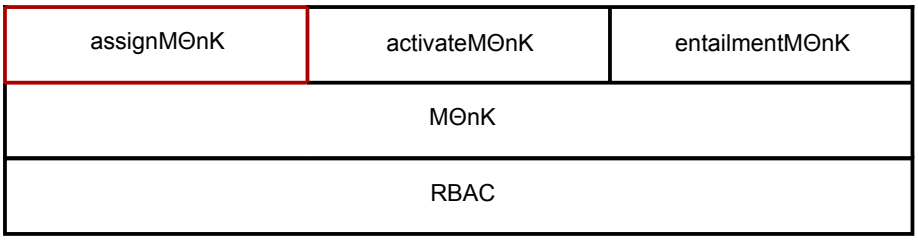


Abb. 5.7: assignMΘnK: MΘnK-Constraint mit statischer Zuordnungssemantik.

Analog zu den abstrakten MΘnK-Constraints werden assignMΘnK-Constraints wie folgt notiert:

- $assignMonK(M, \Theta, n, K)$

So besagt im Fallbeispiel

- $assignMonK(\{Bestellung\}, =, 2, \{Bestellung\ schreiben,\ Bestellung\ aufgeben\})$

mit $typ_M =$ „Rolle“ und $typ_K =$ „Aktivität“, dass die Rolle „Bestellung“ genau zwei der Aktivitäten „Bestellung schreiben“ und „Bestellung aufgeben“ zugeordnet werden muss. Das

bedeutet, dass die Rolle zur Ausführung beider Aktivitäten berechtigen muss. Damit dieser assignMΘnK-Constraint nicht verletzt wird, müssen die entsprechenden Zuordnungen per RBAC vorgenommen werden. Hierbei handelt es sich also um eine Art des Binding of Duty, bei der eine Rolle für mehrere Aktivitäten berechtigen muss. Abb. 5.8 veranschaulicht den assignMΘnK-Constraint und die einzige im Beispiel mögliche Zuordnung zwischen den Elementen aus M und K.

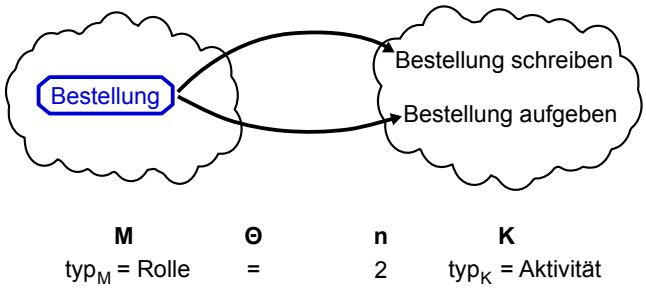


Abb. 5.8: Einzige mögliche Zuordnung zwischen M und K.

Der assignMΘnK-Constraint

- $assignMonK(\{Alice\}, =, 0, \{Finanzen, Produktion\})$

besagt dagegen, dass Benutzer Alice keiner der Rollen „Finanzen“ oder „Produktion“ zugeordnet sein darf. Damit ist Alice niemals in der Lage Aktivitäten auszuführen, deren Ausführung ausschließlich mittels dieser Rollen möglich ist. Abb. 5.9 veranschaulicht diesen assignMΘnK-Constraint.

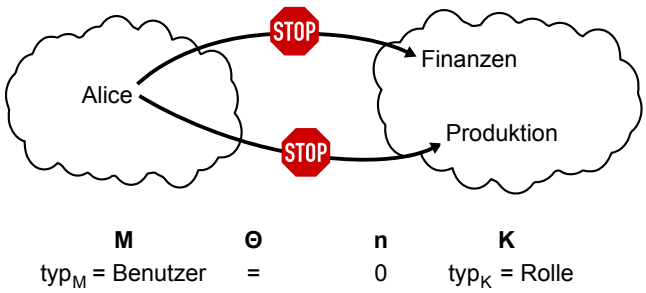


Abb. 5.9: Verbot der Zuordnung von Alice zu den Rollen „Finanzen“ und „Produktion“.

5.2.3 activateMΘnK: MΘnK-Constraint mit dynamischer Aktivierungssemantik

activateMΘnK-Constraints (Abb. 5.10) schränken hingegen die Aktivierung von Rollen und die Ausführung von Aktivitäten ein. Auf diese Weise kann zur Laufzeit verhindert werden, dass mittels RBAC vergebene Berechtigungen tatsächlich in Anspruch genommen werden. So kann mit activateMΘnK-Constraints beispielsweise die Aktivierung einer Rolle durch bestimmte Benutzer verhindert werden. Derartige activateMΘnK-Constraints sind daher eng mit dem Session-Konzept von RBAC verknüpft. Durch activateMΘnK-Constraints können also Constraints mit einer dynamischen Semantik formuliert werden.

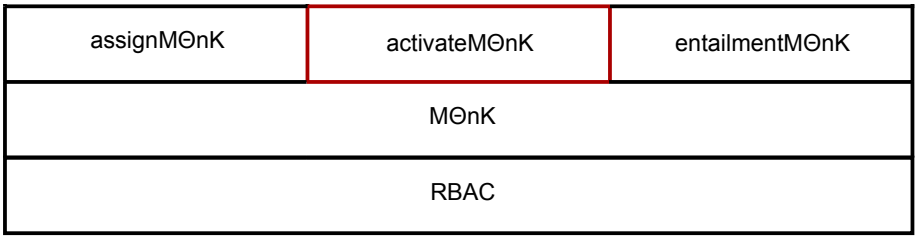


Abb. 5.10: activateMΘnK: MΘnK-Constraint mit dynamischer Aktivierungssemantik.

Auch activateMΘnK-Constraints werden analog zu den abstrakten MΘnK-Constraints wie folgt notiert:

- $activateMonK(M, \Theta, n, K)$

So besagt im Fallbeispiel

- $activateMonK(\{Alice\}, \leq, 1, \{Bestellung, Prüfung\})$

dass Benutzer Alice höchstens eine der beiden Rollen „Bestellung“ und „Prüfung“ aktivieren darf. Es wird somit ausgeschlossen, dass Alice beide Rollen aktiviert. Es handelt sich also bei diesem activateMΘnK-Constraint um eine Art des dynamischen Separation of Duty auf den beiden Rollen „Bestellung“ und „Prüfung“. Durch die dynamische Aktivierungssemantik des activateMΘnK-Constraints ist Alice jedoch nicht vorab auf eine der beiden Rollen festgelegt. Erst wenn zur Laufzeit eine der Rollen durch Alice aktiviert wird, wird die Aktivierung der entsprechenden anderen Rolle durch Alice ausgeschlossen. Abb. 5.11 zeigt diese Situation nach Aktivierung der Rolle „Bestellung“ durch Alice. Die Rolle „Prüfung“ darf dann nicht mehr von Alice aktiviert werden.

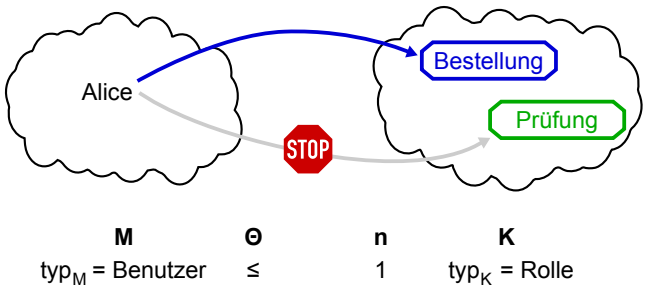


Abb. 5.11: Situation nach Aktivierung der Rolle „Bestellung“ durch Alice.

Der activateMΘnK-Constraint

- $activateMonK(\{Bob\}, \leq, 0, \{Rechnung\ prüfen\})$

besagt dagegen, dass Benutzer Bob nicht die Aktivität „Rechnung prüfen“ ausführen darf. Dieser activateMΘnK-Constraint wird in Abb. 5.12 veranschaulicht.

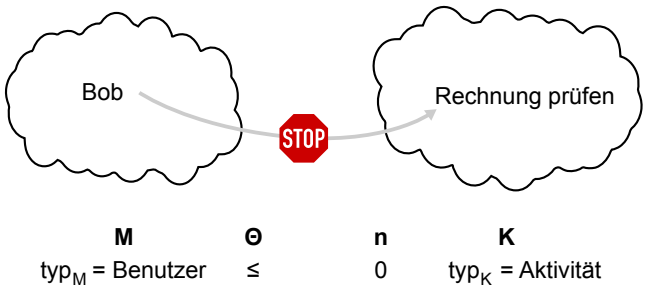


Abb. 5.12: Verbot der Ausführung von „Rechnung prüfen“ durch Bob.

Im Rahmen von activateMΘnK-Constraints ist zu beachten, dass Θ keinen der Werte aus $\{=, \geq\}$ annehmen darf. Mit anderen Worten nimmt Θ stets den Wert ‘ $\leq$ ‘ an. Der Grund hierfür liegt in der Dynamik der activateMΘnK-Constraints und in der Art der Auswertung dieser dynamischen MΘnK-Constraints. So kann während der Laufzeit des Prozesses sichergestellt werden, dass beispielsweise eine Rolle höchstens n mal aktiviert wird. Jedoch kann erst bei Beendigung des Prozesses festgestellt werden, ob eine Rolle genau n mal (falls $\Theta =$ ‘ $=$ ‘) oder mindestens n mal (falls $\Theta =$ ‘ $\geq$ ‘) aktiviert wurde. Für $\Theta \in \{=, \geq\}$ kann also erst bei Prozessende festgestellt werden, ob der entsprechende activateMΘnK-Constraint tatsächlich eingehalten wurde. Dies ist jedoch aus offensichtlichen Gründen vollkommen unpraktikabel und wird daher durch die Einschränkung $\Theta \notin \{=, \geq\}$ bzw. $\Theta \in \{\leq\}$ verhindert.

5.2.4 entailmentMΘnK: MΘnK-Constraint in Abhängigkeit der Ausführungshistorie

Um Constraints auch in Abhängigkeit von der Ausführungshistorie des Prozesses formulieren zu können, ist außerdem ein zu Entailment Constraints (vgl. Kapitel 3.3) ähnliches Konstrukt nötig. Konkret wird dies durch **entailmentMΘnK-Constraints** (Abb. 5.13) modelliert. Ein entailmentMΘnK-Constraint besteht aus zwei activateMΘnK-Constraints und wird wie folgt notiert:

- $activateMonK_1(M_1, =, 1, K_1) \Rightarrow activateMonK_2(M_2, \leq, n_2, K_2)$

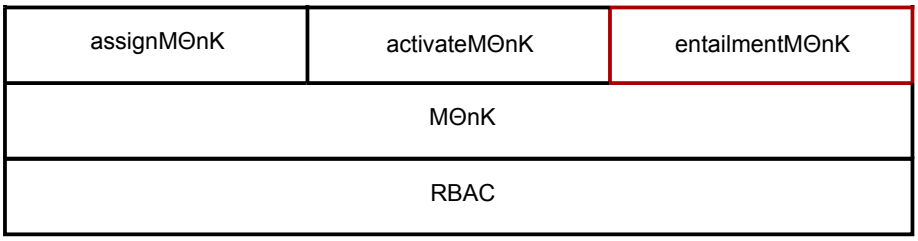


Abb. 5.13: entailmentMΘnK: MΘnK-Constraint in Abhängigkeit der Ausführungshistorie.

Im Gegensatz zu den Ausführungen in Kapitel 5.2.3 ist zu beachten, dass bei $activate-MonK_1$ immer gilt $\Theta_1 = \text{`} = \text{`}$. Dies ist an dieser Stelle legitim, da bei entailmentMΘnK-Constraints eine „wenn-dann"-Semantik vorliegt. Die genaue Semantik ist dabei wie folgt:

Wird zur Laufzeit des Prozesses einem $m \in M_1$ ein $k \in K_1$ zugeordnet, so wird der **Folgeconstraint** $activateMonK_2$ aktiv. Dieser Folgeconstraint muss dann im weiteren Verlauf des Prozesses eingehalten werden. Eine derartige Zuordnung zwischen m und k liegt in diesem Zusammenhang dann vor, wenn beispielsweise k durch m ausgeführt oder aktiviert wird. Solange jedoch keine solche Zuordnung zwischen einem Element $m \in M_1$ und einem Element $k \in K_1$ existiert, muss der activateMΘnK-Constraint $activateMonK_2$ nicht eingehalten werden.

Auf diese Weise lassen sich also activateMΘnK-Constraints in Abhängigkeit der Ausführungshistorie des Prozesses formulieren. Dabei ist zu beachten, dass, den Ausführungen in Kapitel 5.2.3 entsprechend, für den Folgeconstraint $activateMonK_2$ stets $\Theta = \text{`} \leq \text{`}$ gilt. In $activateMonK_2$ können zusätzlich die Platzhalter $activatingUser$, $executingUser$, $activatedRole$ und $executedTask$ verwendet werden. Mit diesen kann zur Laufzeit ein Be-

zug zu den in $activateMonK_1$ zugeordneten Elementen hergestellt werden. Die Semantik dieser Platzhalter wird in Tabelle 5.1 beschrieben.

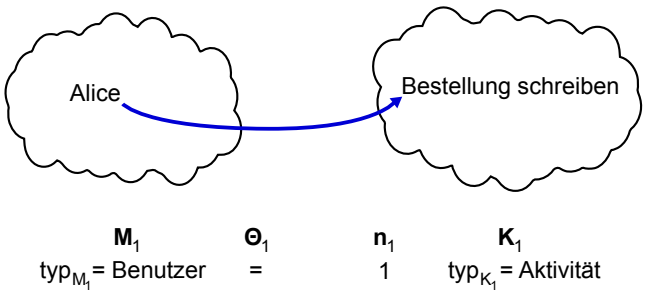
Platzhalter	Semantik	möglich bei	
		typ_{M_1}	typ_{K_1}
$activatingUser$	der eine Rolle aktivierende Benutzer	Benutzer	Rolle
		Rolle	Benutzer
$executingUser$	der eine Aktivität ausführende Benutzer	Benutzer	Aktivität
		Aktivität	Benutzer
$activatedRole$	die durch einen Benutzer aktivierte Rolle	Benutzer	Rolle
		Rolle	Benutzer
	die eine Aktivität ausführende Rolle	Aktivität	Rolle
		Rolle	Aktivität
$executedTask$	die durch einen Benutzer ausgeführte Aktivität	Benutzer	Aktivität
		Aktivität	Benutzer
	die mittels einer Rolle ausgeführte Aktivität	Aktivität	Rolle
		Rolle	Aktivität

Tabelle 5.1: Semantik der Platzhalter bei $activateMonK_2$.

Beispielsweise drückt im Fallbeispiel der entailmentMΘnK-Constraint

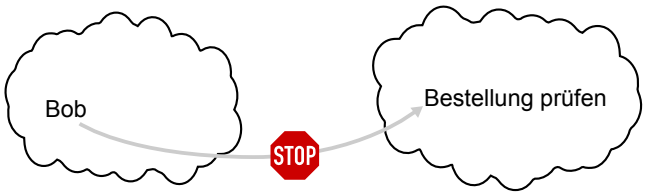
- $activateMonK_1(\{Alice\}, =, 1, \{Bestellung\ schreiben\})$
  $\Rightarrow activateMonK_2(\{Bob\}, \leq, 0, \{Bestellung\ prüfen\})$

aus, dass Benutzer Bob nicht die Aktivität „Bestellung prüfen“ ausführen darf, falls die Aktivität „Bestellung schreiben“ durch Alice ausgeführt wurde. Durch diesen entailment-MΘnK-Constraints wird also verhindert, dass Bob Bestellungen prüft, die von Alice geschrieben wurden. Abb. 5.14a zeigt dabei $activateMonK_1$ bei der Ausführung von „Bestellung schreiben“ durch Alice. Abb. 5.14b zeigt den daraus resultierenden Folgeconstraint $activateMonK_2$. Dieser verhindert die Ausführung von „Bestellung prüfen“ durch Bob und muss ab dem Zeitpunkt der Ausführung von „Bestellung schreiben“ durch Alice eingehalten werden.



$\mathbf{M_1}$	$\mathbf{\Theta_1}$	$\mathbf{n_1}$	$\mathbf{K_1}$
typ_{M_1} = Benutzer	=	1	typ_{K_1} = Aktivität

(a) Ausführung von „Bestellung schreiben“ durch Alice.



$\mathbf{M_2}$	$\mathbf{\Theta_2}$	$\mathbf{n_2}$	$\mathbf{K_2}$
typ_{M_2} = Benutzer	≤	0	typ_{K_2} = Aktivität

(b) Resultierendes Verbot für Bob zum Prüfen der Bestellung.

Abb. 5.14: Ausführung von „Bestellung schreiben“ durch Alice und der einzuhaltende Folgeconstraint.

5.2.5 Einordnung von assignMΘnK, activateMΘnK und entailmentMΘnK

Wie gesehen, werden alle drei Konzepte assignMΘnK, activateMΘnK und entailmentMΘnK auf MΘnK-Constraints mit unterschiedlichen Semantiken zurückgeführt. Die verschiedenen MΘnK-Constraints unterscheiden sich lediglich in der Semantik der Zuordnungen zwischen den Elementen aus M und den Elementen aus K.

So besitzen assignMΘnK-Constraints eine statische Zuordnungssemantik und schränken die Berechtigungsvergabe durch RBAC bereits vor der Ausführung des Prozesses ein. Dagegen schränken activateMΘnK-Constraints die Aktivierung von Rollen oder die Ausführung von Aktivitäten zur Laufzeit des Prozesses ein. entailmentMΘnK-Constraints erweitern das Konzept der activateMΘnK-Constraints um eine Folgerung („⇒“) und ermöglichen auf diese Weise die Formulierung von activateMΘnK-Constraints, die vom Verlauf des Prozesses abhängig sind. Somit liegen drei konkrete Ausprägungen der abstrakten MΘnK-

Constraints vor, die sich zwar in ihrer Semantik, nicht jedoch in ihrer grundlegenden Syntax, unterscheiden. Dieser Zusammenhang wird in Abb. 5.15 illustriert.

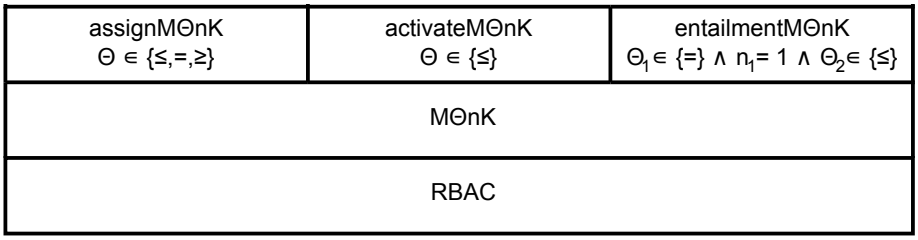


Abb. 5.15: Einordnung und Besonderheiten von assignMΘnK, activateMΘnK und entailmentMΘnK.

5.3 Modellierung der wichtigsten Constraints mit MΘnK

In Kapitel 4.2 wurden bereits die für WfMS wichtigsten Constraints vorgestellt. Diese werden im Folgenden mittels den in Kapitel 5.2 vorgestellten MΘnK-Constraints in einer einheitlichen Form modelliert. Die Constraints werden somit auf MΘnK-Constraints abgebildet. Hierbei ist zu beachten, dass ein Constraint unter Umständen auf mehrere MΘnK-Constraints abgebildet werden muss.

Statische Constraints bestehen dabei lediglich aus assignMΘnK-Constraints, während sich dynamische Constraints aus activateMΘnK-Constraints und entailmentMΘnK-Constraints zusammensetzen. Hybride Constraints werden dagegen auf assignMΘnK-Constraints und activateMΘnK-Constraints bzw. entailmentMΘnK-Constraints abgebildet. Dieser Zusammenhang zwischen den verschiedenen Constraints wird in Abb. 5.16 veranschaulicht.

Im Folgenden wird die Abbildung der wichtigsten Constraints auf MΘnK-Constraints vorgenommen. Diese Abbildungen werden als **Template** bezeichnet. Wann immer die entsprechenden Constraints Anwendung finden sollen, können diese Templates zur einfachen Abbildung der Constraints auf MΘnK-Constraints verwendet werden. Auf diese Weise ist für jede Art von Constraint nur eine einmalige Abbildung auf MΘnK-Constraints nötig. Bei der Anwendung der Templates sind dann lediglich die im konkreten Fall zutreffenden Benutzer, Rollen und Aktivitäten zu spezifizieren. Diese Templates bilden also eine Art Zwischenschicht zwischen den statischen, hybriden und dynamischen Constraints auf der einen

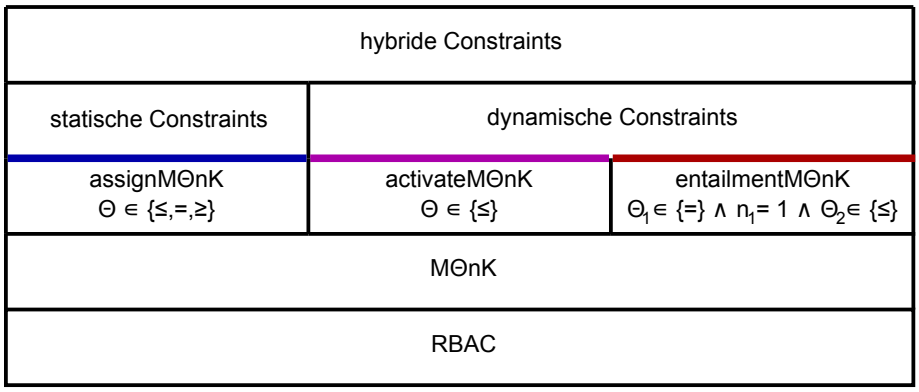


Abb. 5.16: Abbildung von statischen, hybriden und dynamischen Constraints auf MΘnK-Constraints und Einordnung der Templates.

Seite und den drei verschiedenen Arten von MΘnK-Constraints auf der anderen Seite. In Abb. 5.16 werden die Templates als Zwischenschicht farblich angedeutet.

5.3.1 Statisches Separation of Duty auf Rollen

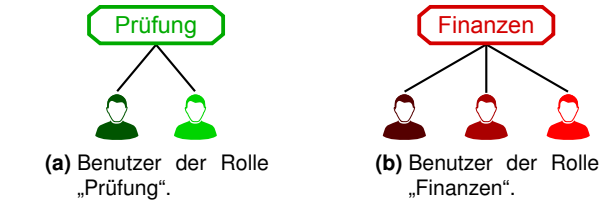
Bei einem statischen Separation of Duty auf Rollen (RoleSSoD) sollen den gegebenen Rollen keine gemeinsamen Benutzer zugeordnet werden. Mit Hilfe eines solchen Constraints können also beispielsweise Abteilungen eines Unternehmens modelliert werden, so dass kein Mitarbeiter in mehreren Abteilungen tätig sein kann. Ein derartiger Constraint ist von einer statischen Natur und kann allein durch assignMΘnK-Constraints ausgedrückt werden.

Eine sinnvolle Forderung im Fallbeispiel wäre, dass kein Benutzer mehr als einer der Rollen „Bestellung“, „Prüfung“ und „Finanzen“ zugeordnet sein darf. Dieser Sachverhalt kann durch drei assignMΘnK-Constraints zum Ausdruck gebracht werden. Bei diesen assignMΘnK-Constraints ist jeweils $typ_M =$ „Rolle“ und $typ_K =$ „Benutzer“. Allen Rollen in der Menge M müssen somit $\Theta\ n$ viele Benutzer aus der Menge K zugeordnet werden:

`RoleSSoD(Bestellung, Prüfung, Finanzen)`:

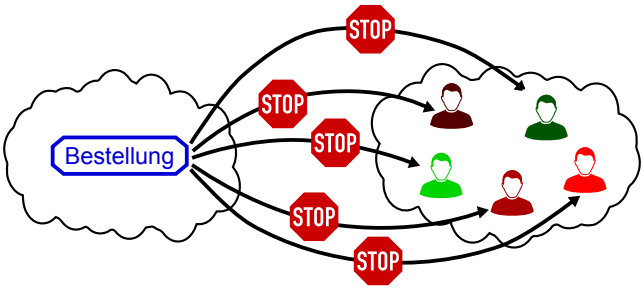
- $assignMonK(\{Bestellung\}, =, 0, users(Prüfung) \cup users(Finanzen))$
- $assignMonK(\{Prüfung\}, =, 0, users(Bestellung) \cup users(Finanzen))$
- $assignMonK(\{Finanzen\}, =, 0, users(Bestellung) \cup users(Prüfung))$

So besagt der erste der drei assignMΘnK-Constraints, dass der Rolle „Bestellung“ kein Benutzer aus der Menge $users(Prüfung) \cup users(Finanzen)$ zugeordnet sein darf. Diese Menge ist die Vereinigungsmenge aller Benutzer der Rolle „Prüfung“ und der Rolle „Finanzen“. Sie entspricht somit allen Benutzern, die mindestens einer dieser beiden Rollen zugeordnet sind. Anders ausgedrückt darf ein Benutzer nicht der Rolle „Bestellung“ zugeordnet werden, wenn dieser bereits mindestens einer der Rollen „Prüfung“ oder „Finanzen“ angehört. Abb. 5.17a und Abb. 5.17b zeigen die Benutzerzuordnungen der Rollen „Prüfung“ und „Finanzen“, während Abb. 5.17c den beschriebenen assignMΘnK-Constraint veranschaulicht. Die Argumentation für die beiden anderen assignMΘnK-Constraints erfolgt analog.



(a) Benutzer der Rolle „Prüfung“.

(b) Benutzer der Rolle „Finanzen“.



M	Θ	n	K
typ_M = Rolle	=	0	typ_K = Benutzer

(c) Eine Zuordnung von Benutzern der Rollen „Prüfung“ und „Finanzen“ an die Rolle „Bestellung“ ist nicht erlaubt.

Abb. 5.17: assignMΘnK-Constraint bei statischem Separation of Duty auf Rollen.

Ist ganz allgemein eine Menge von Rollen $\{r_1, \ldots, r_k\}$ gegeben und soll auf diesen Rollen ein SSoD formuliert werden, so ist dies durch Abbildung auf k viele assignMΘnK-Constraints möglich. Für jede Rolle r_i $(1 \leq i \leq k)$ wird dabei ein assignMΘnK-Constraint gebildet, wobei wiederum gilt $typ_M =$ „Rolle“ und $typ_K =$ „Benutzer“:

`RoleSSoD(`$r_1, \ldots, r_k$`)`:

- $assignMonK(\{r_i\}, =, 0, \bigcup_{1 \leq j \leq k,\ i \neq j} users(r_j)), \quad \forall 1 \leq i \leq k$

Analog zur Argumentation bei obigem Beispiel, erhält also kein Benutzer Berechtigungen für eine Rolle r_i, wenn dieser bereits zu den Benutzern einer anderen Rolle r_j ($i \neq j$) gehört, die ebenfalls Bestandteil des RoleSSoD-Constraints ist. In jedem der k assignMΘnK-Constraints enthält hierzu die Menge M genau eine Rolle, während die Menge K jeweils gerade der Vereinigungsmenge der Benutzer aller anderen Rollen entspricht.

Die Funktion **RoleSSoD($r_1, \ldots, r_k$)** ist also ein Template für SSoD-Constraints auf Rollen. Soll ein RoleSSoD-Constraint in einem konkreten Prozess Anwendung finden, so sind lediglich die entsprechenden Rollen zu spezifizieren. Die Abbildung auf assignMΘnK-Constraints findet dann automatisch mit Hilfe der **RoleSSoD($r_1, \ldots, r_k$)**-Funktion statt.

5.3.2 Statisches Separation of Duty auf Aktivitäten

Bei einem statischen Separation of Duty auf Aktivitäten (TaskSSoD) dürfen den gegebenen Aktivitäten keine gemeinsamen Benutzer zugeordnet werden. Mit Hilfe eines solchen Constraints kann also statisch ausgeschlossen werden, dass ein Benutzer jemals mehr als eine der gegebenen Aktivitäten ausführt. Auch dieser Constraint kann ausschließlich durch assignMΘnK-Constraints ausgedrückt werden.

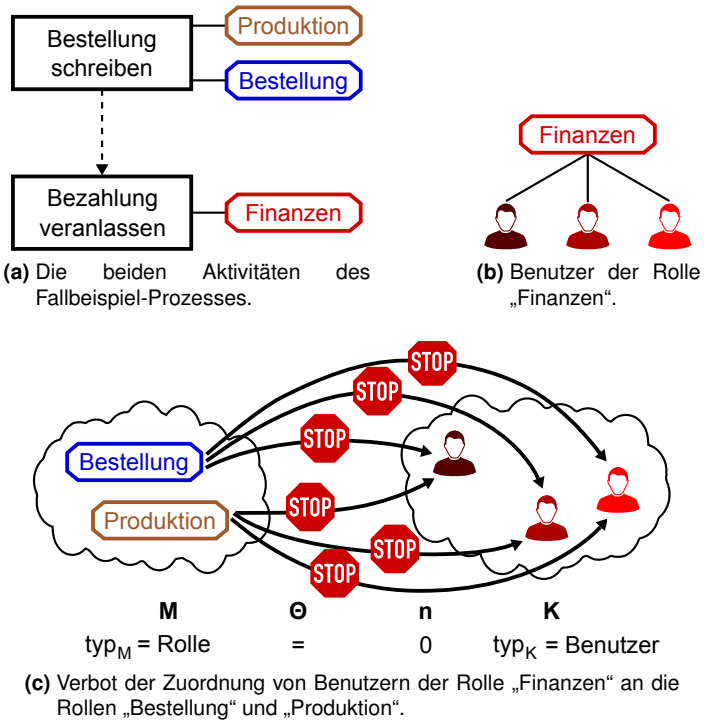
Im Fallbeispiel wäre eine Forderung, dass ein Benutzer entweder die Aktivität „Bestellung schreiben“ oder die Aktivität „Bezahlung veranlassen“ ausführen darf. Ein Benutzer darf also nur die Berechtigung für eine der beiden Aktivitäten erhalten. Somit ist es ihm niemals möglich die andere Aktivität auszuführen. Dies kann durch die beiden folgenden assignMΘnK-Constraints ausgedrückt werden, bei denen jeweils gilt $typ_M =$ „Rolle“ und $typ_K =$ „Benutzer“:

TaskSSoD(Bestellung schreiben, Bezahlung veranlassen):

- $assignMonK(roles(Bestellung\ schreiben), =, 0, users(Bezahlung\ veranlassen))$
- $assignMonK(roles(Bezahlung\ veranlassen), =, 0, users(Bestellung\ schreiben))$

Der erste der assignMΘnK-Constraints wird im Folgenden exemplarisch im Detail betrachtet. In der Menge M sind alle Rollen enthalten, die für die Aktivität „Bestellung schreiben“ berechtigen. Die Menge K entspricht dagegen allen Benutzern, die mittels mindestens einer Rolle zum Ausführen der Aktivität „Bezahlung veranlassen“ berechtigt sind. Insgesamt besagt dieser assignMΘnK-Constraint daher, dass jeder der Aktivität „Bestellung schreiben“ zugeordnete Rolle ($roles(Bestellung\ schreiben)$) kein Benutzer aus der Menge

$users(Bezahlung\ veranlassen)$ zugeordnet sein darf. Mit anderen Worten darf also keine Rolle existieren die für „Bestellung schreiben“ berechtigt und der gleichzeitig Benutzer zugeordnet sind, die zusätzlich einer Rolle der Aktivität „Bezahlung veranlassen“ angehören.



(a) Die beiden Aktivitäten des Fallbeispiel-Prozesses.

(b) Benutzer der Rolle „Finanzen“.

(c) Verbot der Zuordnung von Benutzern der Rolle „Finanzen“ an die Rollen „Bestellung“ und „Produktion“.

Abb. 5.18: assignMΘnK-Constraint bei statischem SoD auf Aktivitäten.

Abb. 5.18a und Abb. 5.18b zeigen den entsprechenden Ausschnitt aus dem Fallbeispiel mit den zugehörigen Rollen und den relevanten Benutzerzuordnungen. Es gilt also

- $roles(Bestellung\ schreiben) = \{Bestellung, Produktion\}$ und
- $roles(Bezahlung\ veranlassen) = \{Finanzen\}$,

so dass der erste der beiden assignMΘnK-Constraints wie in Abb. 5.18c veranschaulicht werden kann.

Für den zweiten der beiden assignMΘnK-Constraints kann auf dieselbe Weise argumentiert werden. Insgesamt erhält so kein Benutzer Berechtigungen für die Ausführung beider Aktivitäten.

Soll ganz allgemein auf einer Menge von Aktivitäten $\{t_1, \ldots, t_k\}$ ein TaskSSoD formuliert werden, so kann dieser Sachverhalt durch k assignMΘnK-Constraints ausgedrückt werden. Für jede Aktivität t_i wird hierzu ein assignMΘnK-Constraint gebildet:

TaskSSoD($t_1, \ldots, t_k$):

- $assignMonK(roles(t_i), =, 0, \bigcup_{1 \leq j \leq k,\ i \neq j} users(t_j)), \quad \forall 1 \leq i \leq k$

Einer Rolle die für Aktivität t_i berechtigt darf somit kein Benutzer zugeordnet werden, der über eine andere Rolle Berechtigungen für eine Aktivität t_j ($i \neq j$) besitzt. Auf diese Weise sind die Benutzer zweier Aktivitäten aus $\{t_1 \ldots t_k\}$ stets disjunkt. Ein Benutzer kann also immer nur zur Auführung einer Aktivität aus $\{t_1 \ldots t_k\}$ berechtigt sein.

Die Funktion **TaskSSoD($t_1, \ldots, t_k$)** entspricht somit dem Template für ein statisches Separation of Duty auf Aktivitäten. In einem konkreten Anwendungsfall sind somit nur die entsprechenden Aktivitäten zu spezifizieren. Eine Abbildung auf assignMΘnK-Constraints findet dann mit Hilfe dieser Funktion automatisch statt.

5.3.3 Dynamisches Separation of Duty auf Aktivitäten

Auch mit dynamischem Separation of Duty auf Aktivitäten (TaskDSoD) soll verhindert werden, dass zwei oder mehr Aktivitäten durch denselben Benutzer ausgeführt werden. Jedoch wird bei TaskDSoD ein Benutzer nicht statisch für eine der Aktivitäten festgelegt. Stattdessen kann ein Benutzer statisch durchaus zur Ausführung mehrerer der Aktivitäten berechtigt sein. Erst zur Laufzeit wird dann durchgesetzt, dass ein Benutzer nicht mehr als eine der Aktivitäten ausführt. Derartige TaskDSoD-Constraints werden mittels entailmentMΘnK-Constraints formuliert.

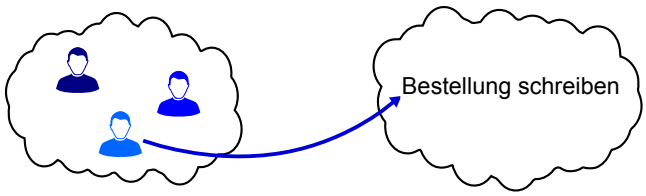
Im Fallbeispiel könnte auf den Aktivitäten „Bestellung schreiben“ und „Bestellung prüfen“ ein TaskDSoD-Constraints gewünscht sein. Damit ist ein Benutzer prinzipiell in der Lage sowohl Bestellungen zu schreiben als auch Bestellungen zu prüfen. Jedoch soll ein Benutzer in einer Prozessinstanz niemals beide Aktivitäten ausführen können. Dieser Sachverhalt kann durch den folgenden entailmentMΘnK-Constraint mit $typ_M =$ „Benutzer“ und $typ_K =$ „Aktivität“ ausgedrückt werden:

- $activateMonK_1(users(Bestellung\ schreiben), =, 1, \{Bestellung\ schreiben\})$
  $\Rightarrow activateMonK_2(\{executingUser\}, \leq, 0, \{Bestellung\ prüfen\})$

In der Menge M_1 sind alle Benutzer enthalten, die zur Ausführung der Aktivität „Bestellung schreiben“ berechtigt sind ($users(Bestellung\ schreiben)$), während K_1 genau diese Aktivität enthält. Führt ein Benutzer die Aktivität „Bestellung schreiben“ aus, so ist dieser auch in $users(Bestellung\ schreiben)$ enthalten. Damit ist $activateMonK_1$ erfüllt und der Folgeconstraint $activateMonK_2$ wird aktiv. Dieser besagt, dass derjenige Benutzer der die Aktivität „Bestellung schreiben“ ausgeführt hat ($executingUser$), nicht die Aktivität „Bestellung prüfen“ ausführen darf. Dieser Folgeconstraint muss im weiteren Verlauf des Prozesses eingehalten werden. Auf diese Weise wird verhindert, dass die beiden Aktivitäten durch denselben Benutzer ausgeführt werden.

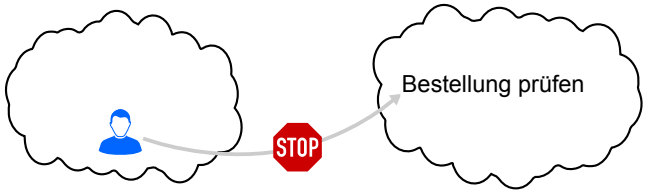
(a) Ausschnitt des Fallbeispiel-Prozesses.

(b) Benutzer der Rolle Bestellung.



M	Θ	n	K
typ_M = Benutzer	=	1	typ_K = Aktivität

(c) Erfüllter entailmentMΘnK-Constraint bei Ausführung von „Bestellung schreiben“.



M	Θ	n	K
typ_M = Benutzer	≤	0	typ_K = Aktivität

(d) Resultierendes Verbot für denselben Benutzer zum Prüfen der Bestellung.

Abb. 5.19: Ausführung von „Bestellung schreiben“ und der einzuhaltende Folgeconstraint.

Abb. 5.19a zeigt den relevanten Ausschnitt aus dem Fallbeispiel-Prozess, Abb. 5.19b die zugehörigen Benutzer der Rolle „Bestellung“. Dabei wird „Bestellung schreiben“ durch einen Benutzer der Rolle „Bestellung“ ausgeführt. Mit dieser Ausführung ist der activateMΘnK-Constraint $activateMonK_1$ des entailmentMΘnK-Constraints erfüllt (Abb. 5.19c). Der daraus resultierende Folgeconstraint (Abb. 5.19d) verbietet demselben Benutzer die Ausführung von „Bestellung prüfen“. Hierbei ist zu beachten, dass in diesem Fall die Ausführungsreihenfolge der beiden Aktivitäten vorab bekannt ist. Ist dies nicht der Fall, so ist ein zusätzlicher entailmentMΘnK-Constraint nötig. Dieser allgemeinere Fall wird mit der folgenden Betrachtung abgedeckt.

Soll ein Benutzer aus einer Menge von Aktivitäten $\{t_1, \ldots, t_k\}$ höchstens eine ausführen dürfen, so kann dies durch k entailmentMΘnK-Constraints ausgedrückt werden:

`TaskDSoD(`$t_1, \ldots, t_k$`)`:

- $activateMonK_1(users(t_i), =, 1, \{t_i\})$
  $\Rightarrow activateMonK_2(\{executingUser\}, \leq, 0, \{t_1, \ldots, t_k\} \backslash \{t_i\})$

Mit diesen k entailmentMΘnK-Constraints gilt daher für jede Aktivität t_i ($1 \leq i \leq k$), dass der die Aktivität t_i ausführende Benutzer keine Aktivität aus $\{t_1, \ldots, t_k\} \backslash \{t_i\}$ ausführen darf. Mit anderen Worten darf ein Benutzer höchstens eine der Aktivitäten aus $\{t_1, \ldots, t_k\}$ ausführen.

Auch an dieser Stelle dient **`TaskDSoD(`$t_1, \ldots, t_k$`)`** als Template. Soll in einem Prozess ein dynamisches Separation of Duty auf zwei oder mehr Aktivitäten formuliert werden, so kann die Abbildung dieses Constraints auf MΘnK-Constraints automatisch mittels dieses Templates geschehen.

5.3.4 Binding of Duty

Binding of Duty (BoD) fordert, dass mehrere Aktivitäten durch denselben Benutzer ausgeführt werden müssen. Eine plausible zusätzliche Einschränkung ist, dass alle diese Aktivitäten in derselben Rolle ausgeführt werden müssen. Für die Formulierung eines solchen BoD-Constraints müssen daher zusätzlich zu den Aktivitäten auch die Rollen bekannt sein, durch die die Aktivitäten ausgeführt werden können. Gegeben ist also eine Menge von Rollen und eine Menge von Aktivitäten.

Eine Forderung im Fallbeispiel könnte lauten, dass die Aktivitäten „Rechnung prüfen“ und „Bezahlung veranlassen“ von demselben Benutzer in der Rolle „Finanzen“ oder „Finanzchef“ ausgeführt werden müssen. Gegeben sind im diesen Fall also die Aktivitäten

- $tasks = \{Rechnung\ pr\ddot{u}fen,\ Bezahlung\ veranlassen\}$

und die Rollen

- $roles = \{Finanzen,\ Finanzchef\}$.

$tasks$ entspricht also der Menge der Aktivitäten des BoD-Constraints, während $roles$ für die Rollen steht, in denen diese Aktivitäten ausgeführt werden können. Im Folgenden wird ein Template für einen solchen BoD-Constraint **`TaskBoD(tasks;roles)`** entwickelt.

Mittels $|roles| + 1$ assignMΘnK-Constraints wird zunächst sichergestellt, dass nur die spezifizierten Rollen $roles$ zur Ausführung der Aktivitäten berechtigen:

- $assignMonK(\{r\}, =, |tasks|, tasks), \qquad \forall r \in roles$
- $assignMonK(allRoles \backslash roles, =, 0, tasks)$

Der erste assignMΘnK-Constraint wird dabei für jede Rolle $r \in roles$ formuliert. Diese insgesamt $|roles|$ viele assignMΘnK-Constraints besagen, dass jede Rolle aus $roles$ für alle Aktivitäten $tasks$ berechtigen muss. Dabei ist $|tasks|$ die Anzahl der Aktivitäten in $tasks$. Durch diese Forderung wird sichergestellt, dass alle Aktivitäten des BoD-Constraints durch dieselbe Rolle ausgeführt werden können. Abb. 5.20 zeigt diesen assignMΘnK-Constraint für die Rolle „Finanzen“.

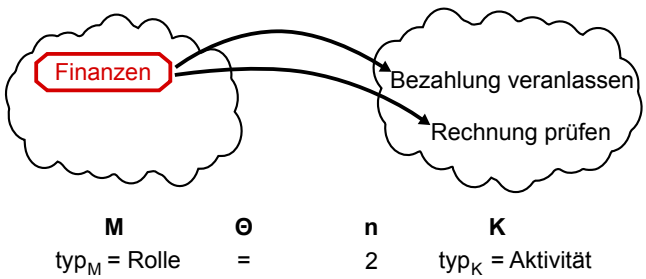


Abb. 5.20: Zuordnung aller Aktivitäten des BoD-Constraints an die Rolle „Finanzen“.

Der assignMΘnK-Constraint

- $assignMonK(allRoles \backslash roles, =, 0, tasks)$

stellt dagegen sicher, dass keine andere Rolle zur Ausführung der Aktivitäten des BoD-Constraints berechtigt. Auf diese Weise wird ausgeschlossen, dass Benutzer anderer Rollen die Aktivitäten des Constraints ausführen. Abb. 5.21 zeigt diesen assignMΘnK-Constraint.

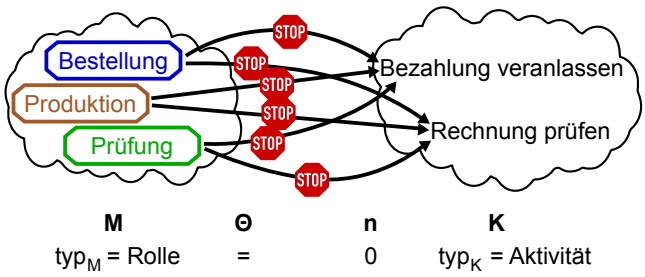
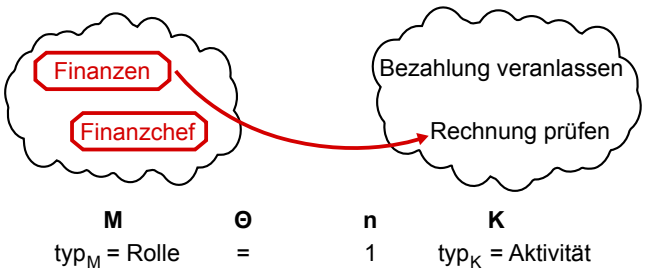


Abb. 5.21: Verbot der Zuordnung aller anderen Rollen an Aktivitäten des BoD-Constraints.

Diese $|roles| + 1$ assignMΘnK-Constraints drücken die notwendige Bedingung für BoD-Constraints aus. Durch diese assignMΘnK-Constraints existieren keine Benutzer, die nur für einen Teil der Aktivitäten $tasks$ des BoD-Constraints berechtigt sind. Jedoch ist auf diese Weise nicht garantiert, dass die Aktivitäten tatsächlich durch denselben Benutzer ausgeführt werden. Um die Ausführung der Aktivitäten durch denselben Benutzer zu erzwingen, sind zusätzlich zwei entailmentMΘnK-Constraints notwendig:

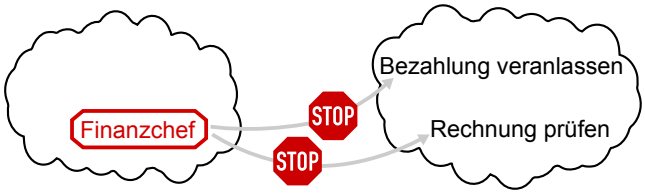
- $activateMonK_1(roles, =, 1, tasks)$
  $\Rightarrow activateMonK_2(roles \backslash \{activatedRole\}, \leq, 0, tasks)$
- $activateMonK_1(\bigcup_{r \in roles} users(r), =, 1, tasks)$
  $\Rightarrow activateMonK_2(\bigcup_{r \in roles} users(r) \backslash \{executingUser\}, \leq, 0, tasks)$

Der erste der beiden entailmentMΘnK-Constraints stellt sicher, dass alle Aktivitäten in $tasks$ durch dieselbe Rolle ausgeführt werden. Denn sobald eine der Aktivitäten aus $tasks$ mittels einer Rolle aus $roles$ ausgeführt wird, wird der Folgeconstraint $activateMonK_2$ aktiv und muss eingehalten werden. Dieser verbietet die Ausführung von Aktivitäten aus $tasks$ durch alle anderen Rollen. Somit können alle weiteren Aktivitäten des BoD-Constraints nur noch in derselben Rolle ($activatedRole$) ausgeführt werden. Dieser entailmentMΘnK-Constraint wird in Abb. 5.22a und Abb. 5.22c veranschaulicht. Abb. 5.22b und Abb. 5.22d zeigen die zugehörigen Benutzer der Rollen „Finanzen“ und „Finanzchef“.



(a) Ausführung von „Rechnung prüfen“ durch einen Benutzer der Rolle Finanzen.

(b) Benutzer der Rolle „Finanzen“.



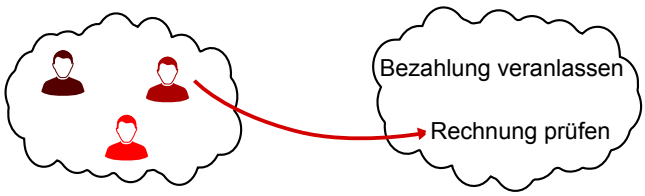
(c) Resultierender Folgeconstraint: Keine der Aktivitäten darf durch einen Benutzer in der Rolle „Finanzchef“ ausgeführt werden.

(d) Benutzer der Rolle „Finanzchef“.

Abb. 5.22: Erzwingen derselben Rolle für alle Aktivitäten des BoD-Constraints.

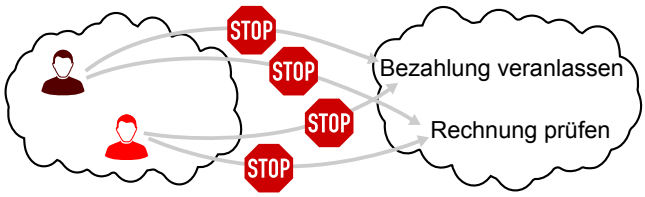
Der zweite entailmentMΘnK-Constraint erzwingt, dass alle Aktivitäten durch denselben Benutzer ausgeführt werden. Wird eine der Aktivitäten $tasks$ durch einen Benutzer ausgeführt, so verbietet der aktiv werdende Folgeconstraint die Ausführung der anderen Aktivitäten durch einen anderen als diesen Benutzer. Alle weiteren Aktivitäten $tasks$ des BoD-Constraints können somit nur durch denselben Benutzer ($executingUser$) ausgeführt werden. Dieser entailmentMΘnK-Constraint wird in Abb. 5.23a und Abb. 5.23b veranschaulicht.

Ein BoD-Constraint wird also sowohl auf assignMΘnK-Constraints, als auch auf entailmentMΘnK-Constraints abgebildet. Während die assignMΘnK-Constraints zur Modellierzeit durchgesetzt werden, ist die Auswertung der entailmentMΘnK-Constraints erst zur Laufzeit des Prozesses möglich. BoD-Constraints gehören daher zur Klasse der hybriden Constraints.



M	Θ	n	K
typ_M = Benutzer	=	1	typ_K = Aktivität

(a) Ausführung von „Rechnung prüfen“ durch einen Benutzer der Rolle „Finanzen“.



M	Θ	n	K
typ_M = Benutzer	≤	0	typ_K = Aktivität

(b) Resultierender Folgeconstraint: Keine der Aktivitäten darf durch einen anderen Benutzer ausgeführt werden.

Abb. 5.23: Erwingen desselben Benutzers für alle Aktivitäten des BoD-Constraints.

Zusammenfassend kann ein Template für einen BoD-Constraint, gemäß den vorherigen Ausführungen, wie folgt angegeben werden:

`TaskBoD` $(tasks; roles)$:

- $assignMonK(\{r\}, =, |tasks|, tasks), \qquad \forall r \in roles$
- $assignMonK(allRoles \backslash roles, =, 0, tasks)$
- $activateMonK(roles, =, 1, tasks)$
  $\Rightarrow activateMonK(roles \backslash \{activatedRole\}, \leq, 0, tasks)$
- $activateMonK(\bigcup_{r \in roles} users(r), =, 1, tasks)$
  $\Rightarrow activateMonK(\bigcup_{r \in roles} users(r) \backslash \{executingUser\}, \leq, 0, tasks)$

Bei Einsatz eines BoD-Constraints müssen somit lediglich die entsprechenden Aktivitäten $tasks$ und Rollen $roles$ spezifiziert werden. Eine Abbildung auf MΘnK-Constraints geschieht dann automatisch unter Verwendung der Funktion **`TaskBoD`** $(tasks; roles)$.

5.3.5 Vier-Augen-Prinzip

Nach dem Vier-Augen-Prinzip soll dieselbe Aktivität mehrmals von unterschiedlichen Benutzern ausgeführt werden. Als Abgrenzung zu herkömmlichem Separation of Duty sind dabei zusätzliche Einschränkungen denkbar. Eine solche Einschränkung wäre, dass die beiden Ausprägungen der Aktivität durch unterschiedliche Benutzer in unterschiedlichen, vorab spezifizierten, Rollen ausgeführt werden müssen.

Das Vier-Augen-Prinzip wird im Folgenden für das Fallbeispiel formuliert. So ist es sinnvoll, dass die Aktivität „Bestellung prüfen“ ab einem bestimmten Warenwert mehrfach durch unterschiedliche Benutzer ausgeführt wird. Auf diese Weise kann die Gefahr von Fehlbestellungen in großem Umfang vermieden werden. Neben einem Benutzer der Rolle „Prüfung“ soll daher zusätzlich ein Benutzer der Rolle „Bestellung“ die Bestellung prüfen. Gegeben ist also die Aktivität „Bestellung prüfen“ und die beiden Rollen „Prüfung“ und „Bestellung“, die zur Ausführung dieser Aktivität berechtigen. Die Aktivität soll nun durch einen Benutzer der Rolle „Prüfung“ und durch einen Benutzer der Rolle „Bestellung“ ausgeführt werden. Hierzu wird im Folgenden eine Abbildung auf MΘnK-Constraints entwickelt.

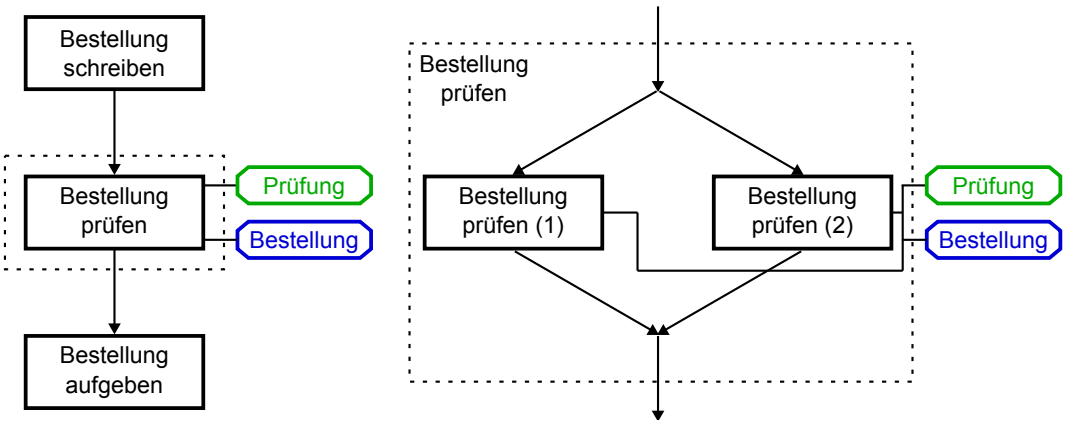


Abb. 5.24: Modifikation des Prozesses zur Umsetzung des Vier-Augen-Prinzips.

Zur Formulierung des Vier-Augen-Prinzips muss zunächst eine zweite Ausprägung der Aktivität „Bestellung prüfen“ in den Prozess eingefügt werden. Die beiden Ausprägungen werden im Folgenden als „Bestellung prüfen (1)“ und „Bestellung prüfen (2)“ bezeichnet. Zusätzlich werden beiden Ausprägungen beide Rollen „Prüfung“ und „Bestellung“ zugeordnet. Abb. 5.24 zeigt diese beiden Ausprägungen der Aktivität. Statisch sind somit die Benutzer

beider Rollen zur Ausführung beider Ausprägungen der Aktivität berechtigt. Dieser Sachverhalt kann mit drei assignMΘnK-Constraints ausgedrückt werden:

- $assignMonK(\{Bestellung\ prüfen\ (1)\}, =, 2, \{Prüfung, Bestellung\})$
- $assignMonK(\{Bestellung\ prüfen\ (2)\}, =, 2, \{Prüfung, Bestellung\})$
- $assignMonK(\{Bestellung\ prüfen\ (1), Bestellung\ prüfen\ (2)\}, =, 0,$
  $allRoles \backslash \{Prüfung, Bestellung\})$

Die ersten beiden assignMΘnK-Constraints drücken aus, dass für beide Ausprägungen der Aktivität beide Rollen „Prüfung“ und „Bestellung“ zur Ausführung berechtigen müssen. Abb. 5.25 zeigt exemplarisch den assignMΘnK-Constraint für „Bestellung prüfen (1)“. Der dritte assignMΘnK-Constraint bringt dagegen zum Ausdruck, dass für diese beiden Ausprägungen der Aktivität keine anderen als die beiden spezifizierten Rollen berechtigen dürfen. Dieser assignMΘnK-Constraint wird in Abb. 5.26 veranschaulicht.

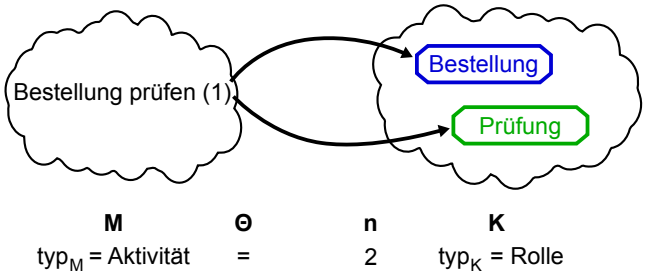


Abb. 5.25: Zuordnung von „Bestellung prüfen (1)“ an beide Rollen „Prüfung“ und „Bestellung“.

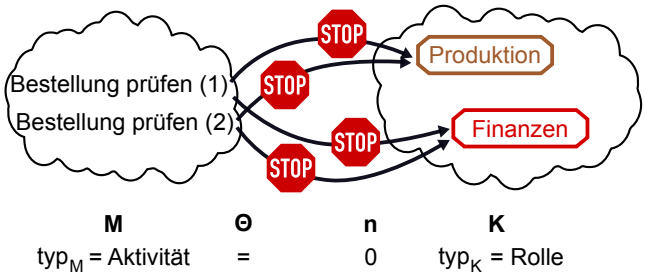


Abb. 5.26: Verbot der Zuordnung aller anderen Rollen zu den beiden Aktivitäten „Bestellung prüfen (1)“ und „Bestellung prüfen (2)“.

Zur Laufzeit des Prozesses muss dann sichergestellt werden, dass die beiden Ausprägungen von „Bestellung prüfen“ tatsächlich durch unterschiedliche Benutzer in den unter-

schiedlichen Rollen ausgeführt werden. Dies wird mit den folgenden entailmentMΘnK-Constraints möglich:

- $activateMonK(\{Prüfung, Bestellung\}, =, 1,$
  $\{Bestellung\ prüfen\ (1), Bestellung\ prüfen\ (2)\})$
  $\Rightarrow activateMonK(\{activatedRole\}, \leq, 0,$
  $\{Bestellung\ prüfen\ (1), Bestellung\ prüfen\ (2)\} \backslash \{executedTask\})$
- $activateMonK(users(Prüfung) \cup users(Bestellung), =, 1,$
  $\{Bestellung\ prüfen\ (1), Bestellung\ prüfen\ (2)\})$
  $\Rightarrow activateMonK(\{executingUser\}, \leq, 0,$
  $\{Bestellung\ prüfen\ (1), Bestellung\ prüfen\ (2)\} \backslash \{executedTask\})$

Der erste entailmentMΘnK-Constraints stellt sicher, dass die beiden Ausprägungen der Aktivität in unterschiedlichen Rollen ausgeführt werden. Wird beispielsweise „Bestellung prüfen (1)“ ($executedTask$) durch einen Benutzer in der Rolle „Bestellung“ ($activatedRole$) ausgeführt (Abb. 5.27a), so ist ab diesem Zeitpunkt der Folgeconstraint einzuhalten. Dieser schließt in diesem Fall aus, dass die andere Ausprägung „Bestellung prüfen (2)“ ebenfalls durch einen Benutzer in der Rolle „Bestellung“ ausgeführt wird. Mit diesen Werten für $executedTask$ und $activatedRole$ sieht der konkrete, und ab sofort einzuhaltende, Folgeconstraint folgendermaßen aus (vgl. auch Abb. 5.27b):

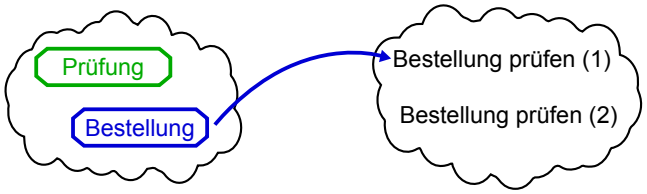
- $activateMonK(\{Bestellung\}, \leq, 0, \{Bestellung\ prüfen\ (2)\})$

Der zweite entailmentMΘnK-Constraint stellt auf eine ähnliche Weise sicher, dass die beiden Ausprägungen der Aktivität durch unterschiedliche Benutzer ausgeführt werden: Führt Benutzer Alice ($executingUser$) die Aktivität „Bestellung schreiben (1)“ ($executedTask$) aus, so wird durch den Folgeconstraint sichergestellt, dass Alice nicht auch „Bestellung schreiben (2)“ ausführt. Der folgende activateMΘnK-Constraint ist also nach der Ausführung von „Bestellung schreiben (1)“ durch Alice einzuhalten:

- $activateMonK(\{Alice\}, \leq, 0, \{Bestellung\ prüfen\ (2)\})$

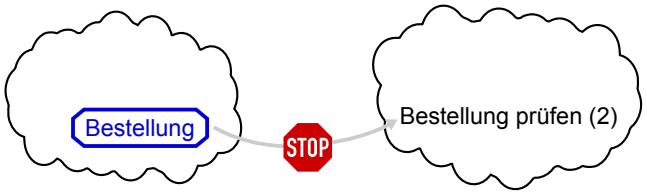
Abb. 5.28a zeigt die Ausführung von „Bestellung prüfen (1)“ durch einen Benutzer der Rolle „Bestellung“ gemäß dem obigen zweiten entailmentMΘnK-Constraint. Der daraus resultierende Folgeconstraint wird in Abb. 5.28b veranschaulicht.

Zusammenfassend wird nun, analog zu den obigen Ausführungen, ein allgemeines Template für ein Vier-Augen-Prinzip auf einer Aktivität $task$ mit zwei Ausprägungen $task_1$ und $task_2$ angegeben. Dabei soll eine der Ausprägungen in Rolle $role_1$ und die andere Ausprä-



M	Θ	n	K
typ_M = Rolle	=	1	typ_K = Aktivität

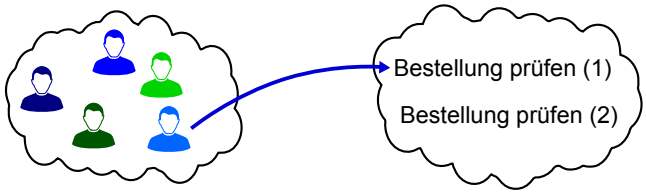
(a) Ausführung von „Bestellung prüfen (1)“ durch einen Benutzer in der Rolle „Bestellung“.



M	Θ	n	K
typ_M = Rolle	≤	0	typ_K = Aktivität

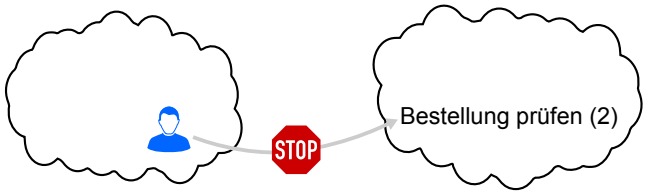
(b) Resultierender Folgeconstraint: Verbot der Ausführung von „Bestellung prüfen (2)“ durch Benutzer in der Rolle „Bestellung“.

Abb. 5.27: Erzwingen unterschiedlicher Rollen für die beiden Ausprägungen der Aktivität „Bestellung prüfen“.



M	Θ	n	K
typ_M = Benutzer	=	1	typ_K = Aktivität

(a) Ausführung von „Bestellung prüfen (1)“ durch einen Benutzer der Rolle „Bestellung“.



M	Θ	n	K
typ_M = Benutzer	≤	0	typ_K = Aktivität

(b) Resultierender Folgeconstraint: „Bestellung prüfen (2)“ darf nicht durch denselben Benutzer ausgeführt werden.

Abb. 5.28: Ausführung von „Bestellung prüfen (1)“ durch einen Benutzer und der einzuhaltende Folgeconstraint.

gung in Rolle $role_2$ ausgeführt werden. $task_1$ und $task_2$ dürfen dabei nicht von demselben Benutzer ausgeführt werden:

4Eyes($task$; $role_1$; $role_2$):

- $assignMonK(\{task_1\}, =, 2, \{role_1, role_2\})$
- $assignMonK(\{task_2\}, =, 2, \{role_1, role_2\})$
- $assignMonK(\{task_1, task_2\}, =, 0, allRoles \backslash \{role_1, role_2\})$
- $activateMonK(\{role_1, role_2\}, =, 1, \{task_1, task_2\})$
  $\Rightarrow activateMonK(\{activatedRole\}, \leq, 0, \{task_1, task_2\} \backslash \{executedTask\})$
- $activateMonK(users(task_1) \cup users(task_2), =, 1, \{task_1, task_2\})$
  $\Rightarrow activateMonK(\{executingUser\}, \leq, 0, \{task_1, task_2\} \backslash \{executedTask\})$

Soll das beschriebene Vier-Augen-Prinzip in einem Prozess Anwendung finden, so müssen mit diesem Template lediglich die entsprechende Aktivität und die beiden Rollen spezifiziert werden. Die Abbildung auf MΘnK-Constraints wird dann durch dieses Template vorgenommen.

5.4 Modellierung weiterer Constraints

Im vorherigen Kapitel wurden die wichtigsten Constraints auf die verschiedenen Arten von MΘnK-Constraints abgebildet. Diese bisher betrachteten Constraints werden auch in der Fachliteratur an vielen Stellen erörtert (vgl. Kapitel 3). Jedoch existieren viele weitere Constraints, deren Einsatz in WfMS in speziellen Fällen sinnvoll und gewünscht sein kann. Diese können deutlich komplexere Strukturen aufweisen und sehr viel individueller auf den jeweiligen Prozess abgestimmt sein.

Auch derartige individuellere Constraints sind mit MΘnK formulierbar bzw. auf die unterschiedlichen MΘnK-Constraints abbildbar. Einem Prozess- bzw. Constraintmodellierer steht es daher offen, individuelle Constraints auf MΘnK-Constraints abzubilden und somit Templates wie in Kapitel 5.3 zu erstellen.

Exemplarisch wird an dieser Stelle ein spezieller Dynamic Separation of Duty Constraint auf MΘnK-Constraints abgebildet werden. Dieser drückt aus, dass ein Benutzer aus einer Menge von n vielen Aktivitäten höchstens $n-1$ viele ausführen darf. Ein solcher Constraint wird nach [14] auch **Operational Dynamic Separation of Duty** (**OpDSoD**) genannt.

Aus einer Menge gegebener Aktivitäten ($tasks$) soll ein Benutzer also maximal $|tasks| - 1$ viele ausführen können. Auf diese Weise wird verhindert, dass ein Benutzer alle der spezifizierten Aktivitäten ausführt. Dieser Sachverhalt kann mit dem folgenden entailment-MΘnK-Constraint ausgedrückt werden:

- $activateMonK_1(\bigcup_{t \in tasks} users(t), =, 1, tasks)$
  $\Rightarrow activateMonK_2(\{executingUser\}, \leq, |tasks| - 2, tasks \backslash \{executedTask\})$

Führt also ein Benutzer eine der Aktivitäten aus $tasks$ aus, so wird der Folgeconstraint $activateMonK_2$ aktiv. Dieser besagt, dass der ausführende Benutzer ($executingUser$) nur noch höchstens $|tasks|-2$ viele der verbleibenden Aktivitäten ($tasks\backslash\{executedTask\}$) ausführen. Insgesamt wird so erreicht, dass kein Benutzer alle der spezifizierten Aktivitäten ausführen kann.

5.5 Zusammenfassung

Wie gesehen, können die verschiedensten Authorization Constraints mittels MΘnK formuliert werden. Dabei werden alle Constraints auf die MΘnK-Struktur (M, Θ, n, K) abgebildet. Durch die unterschiedlichen Semantiken von assignMΘnK-Constraints, activateMΘnK-Constraints und entailmentMΘnK-Constraints ist es möglich statische, dynamische und hybride Constraints auf MΘnK-Constraints abzubilden.

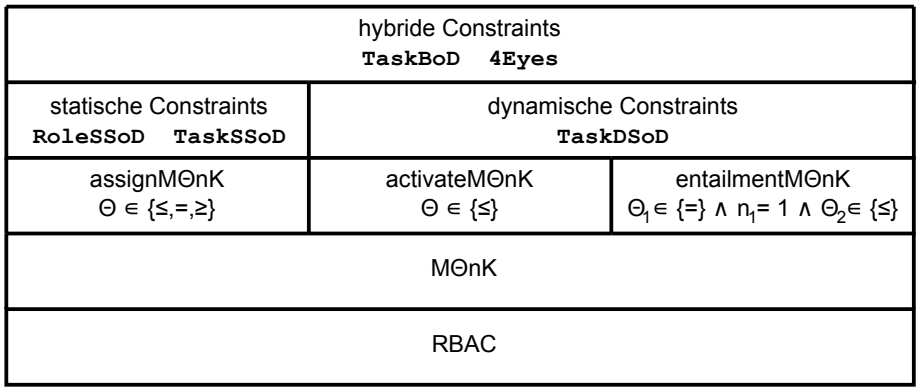


Abb. 5.29: Einordnung der wichtigsten Constraints und ihr Zusammenhang mit den unterschiedlichen MΘnK-Constraints.

Für die wichtigsten Constraints wurden außerdem Templates entwickelt, die die Abbildung dieser Constraints auf MΘnK-Constraints vornehmen. Mit Hilfe dieser Templates wird die Verwendung dieser Constraints deutlich vereinfacht wird, da die Abbildung auf MΘnK-Constraints automatisiert geschehen kann. Eine Einordnung dieser wichtigsten Constraints und ihr Zusammenhang mit den unterschiedlichen MΘnK-Constraints ist in Abb. 5.29 abgebildet.

Die Abbildung der Authorization Constraints auf MΘnK-Constraints bildet die Grundlage für die Betrachtungen in den folgenden Kapiteln 6 und 7. Alle Constraints werden dabei zunächst immer, wie im vorliegenden Kapitel gezeigt, auf MΘnK-Constraints abgebildet. Weitere Betrachtungen der unterschiedlichen Constraints finden dann auf Basis der MΘnK-Constraints statt. Auf diese Weise können diese Betrachtungen losgelöst von den ursprünglichen Constraints stattfinden und sind durch die Fokussierung auf MΘnK-Constraints auf eine einheitliche Weise möglich.

6 Durchsetzung der MΘnK-Constraints

Nachdem in Kapitel 5.3 die gewünschten Constraints durch MΘnK-Constraints formuliert wurden, müssen diese durch das WfMS durchgesetzt werden. Das bedeutet, dass Vorgänge wie die Berechtigungsvergabe, Aktivitätenausführungen und Rollenaktivierungen den MΘnK-Constraints entsprechen müssen. Das Durchsetzen der MΘnK-Constraints kann teilweise zur Modellierzeit, zum Teil jedoch erst zur Laufzeit des Prozesses geschehen. Dabei lassen sich die assignMΘnK-Constraints zur Modellierzeit durchsetzen, während activateMΘnK-Constraints und entailmentMΘnK-Constraints erst zur Laufzeit durchsetzbar sind.

Im Folgenden ist stets eine RBAC-Berechtigungszuordnung und eine Menge verschiedener MΘnK-Constraints gegeben. Es gilt nun, die Einhaltung der MΘnK-Constraints zu überwachen bzw. diese durchzusetzen.

6.1 Durchsetzung von assignMΘnK-Constraints zur Modellierzeit

assignMΘnK-Constraints schränken nach Kapitel 5.2.2 die Berechtigungen ein, die mittels RBAC vergeben werden können. Aus diesem Grund können und müssen assignMΘnK-Constraints zur Modellierzeit des Prozesses durchgesetzt werden. Dabei wird zunächst überprüft, ob die vergebenen RBAC-Berechtigungen den assignMΘnK-Constraints entsprechen. Ist dies nicht der Fall, so müssen die assignMΘnK-Constraints durchgesetzt werden. Das heißt, dass die mittels RBAC vergebenen Berechtigungen an die assignMΘnK-Constraints angepasst werden müssen. Im Folgenden wird beleuchtet, wie die Einhaltung der assignMΘnK-Constraints überprüft werden kann.

Die im Folgenden betrachteten assignMΘnK-Constraints resultieren aus der Abbildung von Authorization Constraints auf MΘnK-Constraints, gemäß den Ausführungen in Kapitel 5.3.

Die RBAC-Berechtigungen und assignMΘnK-Constraints sind dann auf ihre Konformität hin zu überprüfen. Abb. 6.1 zeigt den Kontext dieser Überprüfung.

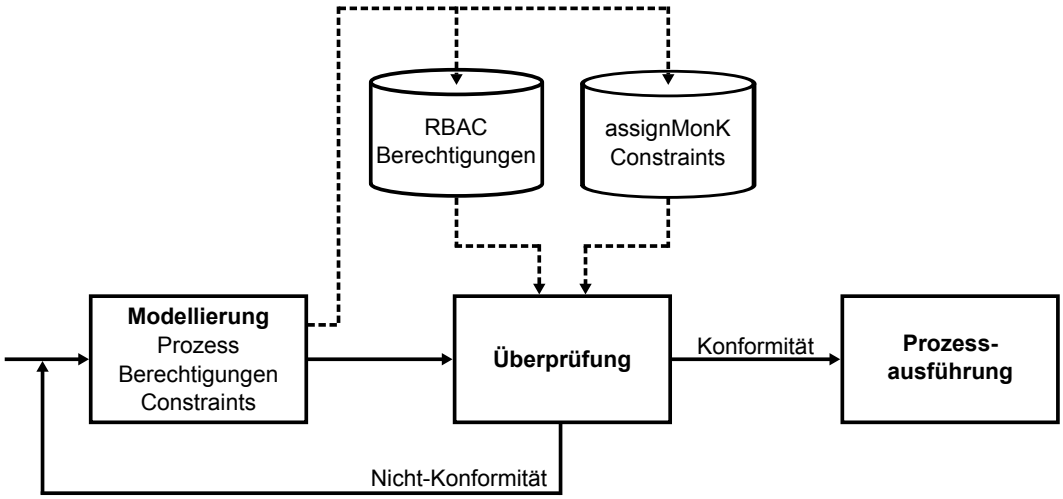


Abb. 6.1: Überprüfung der Konformität von RBAC-Berechtigungen und assignMΘnK-Constraints.

6.1.1 Formale Durchsetzung

Insgesamt existieren sechs verschiedene Arten von assignMΘnK-Constraints, die durch Kombination der Typen der Mengen M und K, typ_M und typ_K, entstehen. Wie in Kapitel 5.2 beschrieben, gilt dabei für jeden assignMΘnK-Constraint $typ_M \neq typ_K$.

Trotz dieser verschiedenen Arten von assignMΘnK-Constraints, ist die Überprüfung der Konformität aller assignMΘnK-Constraints mit den RBAC-Berechtigungen auf eine einheitliche Weise möglich. Damit ein assignMΘnK-Constraint $assignMonK(M, \Theta, n, K)$ durch die vergebenen RBAC-Berechtigungen eingehalten wird, müssen zwei Bedingungen erfüllt sein:

- **i)** jedem Element $m \in M$ werden Θ n viele Elemente aus K zugeordnet
- **ii)** jedes Element $k \in K$ wird höchstens einem Element $m \in M$ zugeordnet

i) Zunächst werden für jedes Element $m \in M$ die ihm, mittels RBAC, zugeordneten Elemente vom Typ typ_K ermittelt. Damit ensteht für jedes Element $m \in M$ eine Menge K_m, für die gilt

- $K_m = users(m)$, falls $typ_K =$ „Benutzer“,
- $K_m = roles(m)$, falls $typ_K =$ „Rolle“,
- $K_m = tasks(m)$, falls $typ_K =$ „Aktivität“.

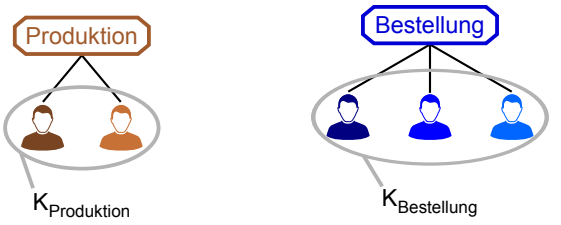
So gilt beispielsweise bei Betrachtung des abstrakten assignMΘnK-Constraints

- $assignMonK(\{Produktion, Bestellung\}, \Theta, n, K)$

mit $typ_M =$ „Rolle“, $typ_K =$ „Benutzer“ und den RBAC-Berechtigungen aus dem Fallbeispiel

- $K_{Produktion} = users(Produktion)$ und
- $K_{Bestellung} = users(Bestellung)$.

$K_{Produktion}$ enthält somit alle Benutzer der Rolle „Produktion“ (Abb. 6.2a), $K_{Bestellung}$ enthält alle Benutzer der Rolle „Bestellung“ (Abb. 6.2b). Somit entstehen $|M|$ viele Mengen K_m.



(a) Benutzer der Rolle „Produktion“.

(b) Benutzer der Rolle „Bestellung“.

Abb. 6.2: $K_{Produktion}$ und $K_{Bestellung}$ bei $assignMonK(\{Produktion, Bestellung\}, \Theta, n, K)$ mit $typ_K =$ „Benutzer“.

Dann muss für jedes $m \in M$ gelten, dass die Größe der Schnittmenge der beiden Mengen K_m und K Θ n entspricht: $|K_m \cap K|$ Θ n, $\forall$ $m \in M$. Ist dies erfüllt, so sind Θ n viele Elemente der Menge K_m auch in der Menge K enthalten.

ii) Zusätzlich darf jedes $k \in K$ höchstens einem $m \in M$ zugeordnet werden. Die Einhaltung dieser Forderung wird überprüft, indem für jedes Element $k \in K$ die ihm, mittels RBAC, zugeordneten Elemente vom Typ typ_M ermittelt werden. Das heißt, dass für jedes $k \in K$ eine Menge M_k entsteht, für die gilt

- $M_k = users(k)$, falls $typ_M =$ „Benutzer“,
- $M_k = roles(k)$, falls $typ_M =$ „Rolle“,
- $M_k = tasks(k)$, falls $typ_M =$ „Aktivität“.

Dann darf für jedes $k \in K$ höchstens ein Element der Menge M_k auch in der Menge M enthalten sein. Dies bedeutet, dass die Schnittmenge dieser beiden Mengen höchstens ein Element enthalten darf: $|M_k \cap M| \leq 1, \ \forall\, k \in K$.

Werden die Bedingungen i) und ii) eingehalten, so entsprechen die vergebenen RBAC-Berechtigungen dem assignMΘnK-Constraint. In Tabelle 6.1 werden diese Bedingungen für die Einhaltung der sechs verschiedenen Arten von assignMΘnK-Constraints ausformuliert.

assignMΘnK-Constraint		**Bedingung für Einhaltung**	
$\boldsymbol{typ_M}$	$\boldsymbol{typ_K}$	$\forall\, \boldsymbol{m} \in \boldsymbol{M}$	$\forall\, \boldsymbol{k} \in \boldsymbol{K}$
Benutzer	Rolle	$\lvert roles(m) \cap K \rvert \ \ \Theta\ n$	$\lvert users(k) \cap M \rvert \ \leq 1$
Aktivität	Rolle	$\lvert roles(m) \cap K \rvert \ \ \Theta\ n$	$\lvert tasks(k) \cap M \rvert \ \leq 1$
Rolle	Benutzer	$\lvert users(m) \cap K \rvert \ \ \Theta\ n$	$\lvert roles(k) \cap M \rvert \ \leq 1$
Rolle	Aktivität	$\lvert tasks(m) \cap K \rvert \ \ \Theta\ n$	$\lvert roles(k) \cap M \rvert \ \leq 1$
Benutzer	Aktivität	$\lvert tasks(m) \cap K \rvert \ \ \Theta\ n$	$\lvert users(k) \cap M \rvert \ \leq 1$
Aktivität	Benutzer	$\lvert users(m) \cap K \rvert \ \ \Theta\ n$	$\lvert tasks(k) \cap M \rvert \ \leq 1$

Tabelle 6.1: Arten von assignMΘnK-Constraints und die einzuhaltenden Bedingungen.

Die Durchsetzung der assignMΘnK-Constraints zur Modellierzeit wurde im Rahmen dieser Arbeit implementiert. Details zur vorgenommenen Implementierung werden in Kapitel 8 betrachtet. Nach diesen formalen Betrachtungen wird im Folgenden die Durchsetzung einiger assignMΘnK-Constraint am Fallbeispiel illustriert.

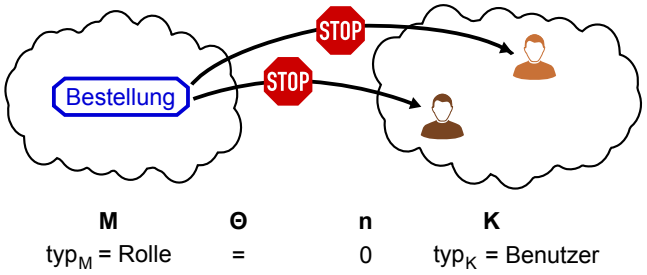
6.1.2 Durchsetzung am Fallbeispiel

Im Folgenden wird exemplarisch die Einhaltung von Constraints, die gemäß den Ausführungen in Kapitel 5.3 auf MΘnK-Constraints abgebildet wurden, überprüft. Zu beachten ist, dass an dieser Stelle nur die jeweiligen assignMΘnK-Constraints betrachtet werden. Die Durchsetzung der activateMΘnK-Constraints und entailmentMΘnK-Constraints werden anschließend in Kapitel 6.2 und Kapitel 6.3 betrachtet. Weiter beschränken wir uns auf die Betrachtung der Elemente der Menge M (Fall i)). Die Überprüfung ob jedes Element aus K höchstens einem $m \in M$ zugeordnet ist (Fall ii)) erfolgt analog.

a) **`RoleSSoD(Bestellung, Produktion)`**:

1. $assignMonK(\{Bestellung\}, =, 0, users(Produktion))$
2. $assignMonK(\{Produktion\}, =, 0, users(Bestellung))$

Die assignMΘnK-Constraints des Constraints a) fordern, dass den beiden Rollen „Bestellung“ und „Produktion“ keine Benutzer der jeweils anderen Rolle zugeordnet werden. Der erste der beiden assignMΘnK-Constraints wird in Abb. 6.3a veranschaulicht. Abb. 6.3b zeigt ergänzend die Benutzer der Rolle „Produktion“.

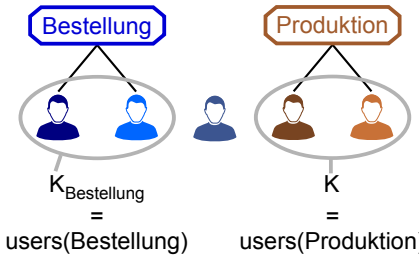


(a) Einer der assignMΘnK-Constraints von RoleSSoD(Bestellung, Produktion).

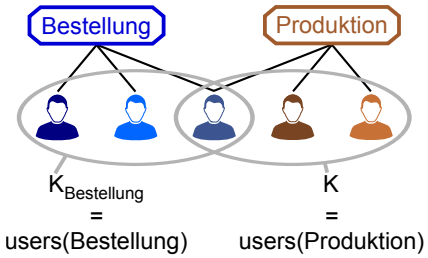
(b) Benutzer der Rolle „Produktion“.

Abb. 6.3: assignMΘnK-Constraint bei Constraint a) und die relevanten Benutzer.

Zur Durchsetzung der beiden assignMΘnK-Constraints muss sichergestellt werden, dass die Schnittmenge der Benutzer der beiden Rollen leer ist. Formal sind, gemäß Kapitel 6.1.1, für den ersten der beiden assignMΘnK-Constraints alle Benutzer der Rolle „Bestellung“ zu ermitteln ($K_{Bestellung} = users(Bestellung)$). Von diesen darf keiner in der Menge der Benutzer der Rolle „Produktion“ enthalten sein (da $K = users(Produktion)$). Abb. 6.4a zeigt hierzu eine erlaubte Benutzerzuordnung, da $|users(Bestellung) \cap users(Produktion)| = 0$.



(a) Erlaubte Benutzerzuordnung.



(b) Nicht erlaubte Benutzerzuordnung.

Abb. 6.4: Erlaubte und nicht erlaubte Benutzerzuordnung bei Constraint a).

Abb. 6.4b zeigt dagegen die nicht erlaubte Zuordnung eines Benutzers zu beiden Rollen, da somit gilt $|users(Bestellung) \cap users(Produktion)| \neq 0$. Die Argumentation für den zweiten assignMΘnK-Constraint erfolgt analog.

b) **`TaskSSoD(Bestellung schreiben, Bestellung prüfen)`**:

1. $assignMonK(\{Bestellung,\ Produktion\}, =, 0, users(Bestellung) \cup users(Prüfung))$
2. $assignMonK(\{Bestellung,\ Prüfung\}, =, 0, users(Bestellung) \cup users(Produktion))$

Bei der Durchsetzung von Constraint b) wird dagegen immer, unabhängig von der Benutzerzuordnung, ein Konflikt auftreten. Beide assignMΘnK-Constraints fordern, dass der Rolle „Bestellung“ keine Benutzer aus der jeweiligen Menge K zugeordnet sind. In K sind jedoch jeweils alle Benutzer der Rolle „Bestellung“ enthalten ($users(Bestellung) \subseteq K$). Dies ist jedoch nur möglich, wenn die Rolle „Bestellung“ leer ist, d.h. wenn ihr keine Benutzer zugeordnet sine. Leere Rollen sind jedoch wenig praktikabel und wurden daher in Kapitel 4.1.2 ausgeschlossen.

Formal wird dieser Widerspruch, exemplarisch für den ersten der beiden assignMΘnK-Constraints, wie folgt aufgedeckt: Wegen $\Theta \in \{=\}$ und $n = 0$, muss gemäß den Ausführungen in Kapitel 6.1.1 gelten

- $|K_{Bestellung} \cap K| = 0$ und
- $|K_{Prüfung} \cap K| = 0$.

Wegen $typ_K =$ „Benutzer“ gilt $K_{Bestellung} = users(Bestellung)$.
Mit $K = users(Bestellung) \cup users(Prüfung)$ muss also gelten

- $|users(Bestellung) \cap (users(Bestellung) \cup users(Prüfung))| = 0$.

Dies ist jedoch nur möglich, wenn die Rolle „Bestellung“ leer ist, d.h. $users(Bestellung) = \emptyset$. Dieser Widerspruch wird auch in Abb. 6.5 deutlich. Er resultiert daraus, dass die Rolle „Be-

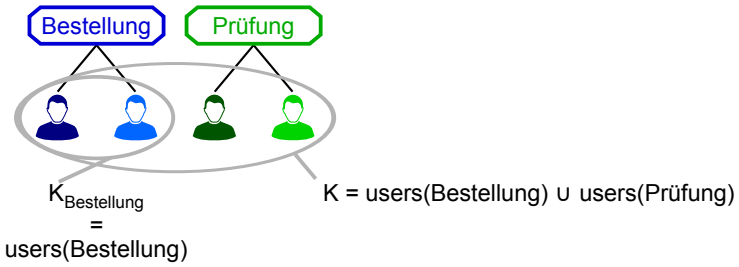


Abb. 6.5: Die Mengen $K_{Bestellung}$ und K bei Constraint b) im Fallbeispiel.

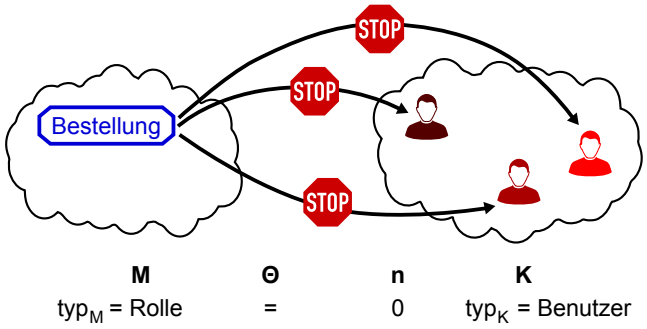
stellung“ für beide Aktivitäten „Bestellung schreiben“ und „Bestellung aufgeben“ berechtigt. Es gibt somit keine RBAC-Berechtigungsvergabe, die diesem Constraint genügt. Es wird Ziel der Validierung in Kapitel 7 sein, derartige Widersprüche bereits frühzeitig zu erkennen.

c) **`TaskSSoD(Bestellung aufgeben, Bezahlung veranlassen)`**:
   1. $assignMonK(\{Bestellung\}, =, 0, users(Finanzen))$
   2. $assignMonK(\{Finanzen\}, =, 0, users(Bestellung))$

Constraint c) besagt, dass die Aktivitäten „Bestellung aufgeben“ und „Bezahlung veranlassen“ nicht von derselben Person ausgeführt werden dürfen. Die assignMΘnK-Constraints kommen dabei mittels der Aktivitäten-Rollen-Zuordnungen aus dem Fallbeispiel zustande, wonach gilt

- $roles(Bestellung\ aufgeben) = \{Bestellung\}$ und
- $roles(Bezahlung\ veranlassen) = \{Finanzen\}$.

Dabei darf nach dem ersten der beiden assignMΘnK-Constraints der Rolle „Bestellung“ kein Benutzer zugeordnet sein, der über die Rolle „Finanzen“ für „Bezahlung veranlassen“ berechtigt ist. Abb. 6.6a veranschaulicht den ersten der beiden assignMΘnK-Constraints, Abb. 6.6b ergänzt die zugehörigen Benutzer der Rolle „Finanzen“.



(a) Einer der assignMΘnK-Constraints von Constraint c).

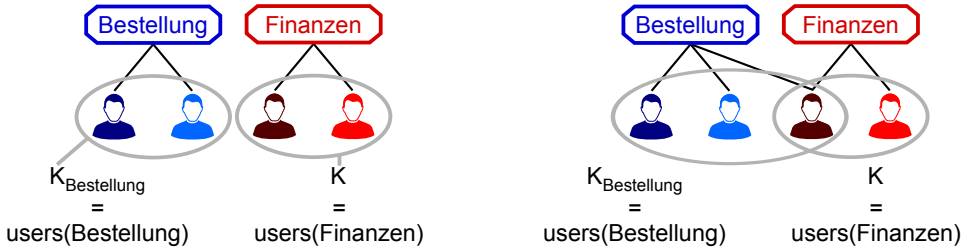
(b) Benutzer der Rolle „Finanzen“.

Abb. 6.6: Verbot der Zuordnung von Benutzern der Rolle „Finanzen“ an die Rolle „Bestellung“ bei Constraint c).

Wie bei Constraint a) gilt auch hier, dass die den Rollen zugeordneten Benutzer disjunkt sein müssen. Exemplarisch sind für den ersten der beiden assignMΘnK-Constraints alle Benutzer der Rolle „Bestellung“ zu ermitteln ($K_{Bestellung} = users(Bestellung)$). Von diesen

darf keiner in der Menge der Benutzer der Rolle „Finanzen“ ($K = users(Finanzen)$) enthalten sein, d.h. es muss gelten $|K_{Bestellung} \cap K| = 0$. Abb. 6.7a illustriert eine erlaubte, Abb. 6.7b eine nicht erlaubte (da $|K_{Bestellung} \cap K| \neq 0$) Zuordnung der Benutzer zu den entsprechenden Rollen.

Die Überprüfung der Einhaltung des zweiten assignMΘnK-Constraints erfolgt analog.



(a) Erlaubte Benutzerzuordnung.

(b) Nicht erlaubte Benutzerzuordnung.

Abb. 6.7: Erlaubte und nicht erlaubte Benutzerzuordnung bei Constraint c).

d) **`TaskBoD(Bestellung schreiben, Bestellung aufgeben; Bestellung)`**:

1. $assignMonK(\{Bestellung\}, =, 2, \{Bestellung\ schreiben,\ Bestellung\ aufgeben\})$
2. $assignMonK(\{Prüfung, Finanzen, Produktion\}, =, 0,$
   $\{Bestellung\ schreiben,\ Bestellung\ aufgeben\})$

Constraint d) fordert, dass die beiden Aktivitäten „Bestellung schreiben“ und „Bestellung aufgeben“ durch denselben Benutzer in der Rolle „Bestellung“ ausgeführt werden. Durch die beiden assignMΘnK-Constraints wird statisch durchgesetzt, dass die Rolle „Bestellung“ für beide Aktivitäten berechtigt, während alle anderen Rollen für keine der Aktivitäten berechtigen. Dabei stimmt der erste der beiden assignMΘnK-Constraints mit den im Fallbeispiel vergebenen Berechtigungen überein: Der Rolle „Bestellung“ sind unter anderem die spezifizierten Aktivitäten „Bestellung schreiben“ und „Bestellung aufgeben“ zugeordnet. Dieser assignMΘnK-Constraint wird in Abb. 6.8 veranschaulicht.

Formal wird die Einhaltung dieses assignMΘnK-Constraints wie folgt überprüft: Wegen $typ_K =$ „Aktivität“, gilt

- $K_{Bestellung} = tasks(Bestellung)$
  $= \{Bestellung\ schreiben,\ Bestellung\ prüfen,$
  $Bestellung\ aufgeben,\ Wareneingang\ erfassen\}.$

Mit $K = \{Bestellung\ schreiben,\ Bestellung\ aufgeben\}$ ergibt sich

- $|K_{Bestellung} \cap K| = |\{Bestellung\ schreiben,\ Bestellung\ aufgeben\}| = 2,$

womit der erste assignMΘnK-Constraint mit $\Theta = \text{`} = \text{`}$ und $n = 2$ erfüllt ist.

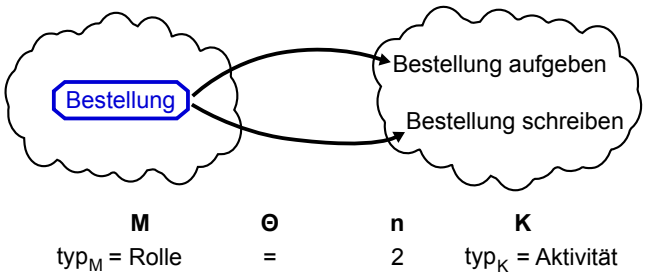


Abb. 6.8: Zuordnung beider Aktivitäten zur Rolle „Bestellung“ bei Constraint d).

Der zweite assignMΘnK-Constraint, abgebildet in Abb. 6.9, wird jedoch nicht eingehalten, da die Rolle „Produktion“ der Aktivität „Bestellung schreiben“ zugeordnet ist. Formal wird dieser Konflikt wie folgt festgestellt: Es gilt $typ_K =$ „Aktivität“ und somit insbesondere

- $K_{Produktion} = tasks(Produktion)$
  $= \{Bestellung\ schreiben,\ Wareneingang\ erfassen\}$

Mit $K = \{Bestellung\ schreiben,\ Bestellung\ aufgeben\}$ ergibt sich dann

- $|K_{Produktion} \cap K| = |\{Bestellung\ schreiben\}| = 1,$

womit der zweite assignMΘnK-Constraint mit $\Theta = \text{`} = \text{`}$ und $n = 0$ nicht erfüllt ist.

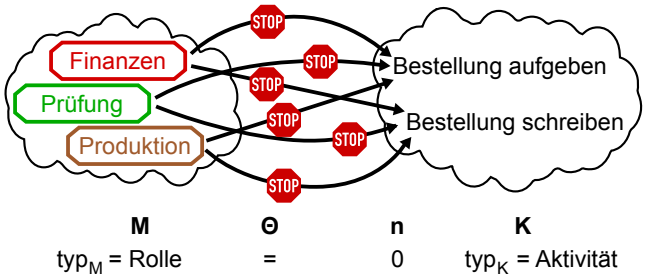


Abb. 6.9: Verbot der Zuordnung der Aktivitäten zu allen anderen Rollen bei Constraint d).

Tritt ein solcher Konflikt auf, muss der entsprechende assignMΘnK-Constraint durchgesetzt werden. Dies geschieht durch Anpassung der vergebenen RBAC-Berechtigungen. In diesem Falll müsste daher die RBAC-Zuordnung zwischen der Rolle „Produktion“ und der

Aktivität „Bestellung schreiben“ aufgelöst werden. Die Auflösung eines solchen Konflikts obliegt dabei in jedem Fall dem Prozess- bzw. Constraintmodellierer.

e) **`TaskBoD(Rechnungseingang erfassen, Bezahlung veranlassen; Finanzen)`**:

1. $assignMonK(\{Finanzen\}, =, 2,$
   $\{Rechnungseingang\ erfassen,\ Bezahlung\ veranlassen\})$
2. $assignMonK(\{Bestellung, Prüfung, Produktion\}, =, 0,$
   $\{Rechnungseingang\ erfassen,\ Bezahlung\ veranlassen\})$

Die assignMΘnK-Constraints von Constraint e) entsprechen dem bereits zur Modellierzeit durchsetzbaren Teil des BoD-Constraints. In diesem Fall entsprechen die vergebenen RBAC-Berechtigungen des Fallbeispiels den beiden assignMΘnK-Constraints: Die Rolle „Finanzen“ berechtigt, unter anderem, für beide Aktivitäten „Rechnungseingang erfassen“ und „Bezahlung veranlassen“. Daher wird der erste der beiden assignMΘnK-Constraint (Abb. 6.10) eingehalten.

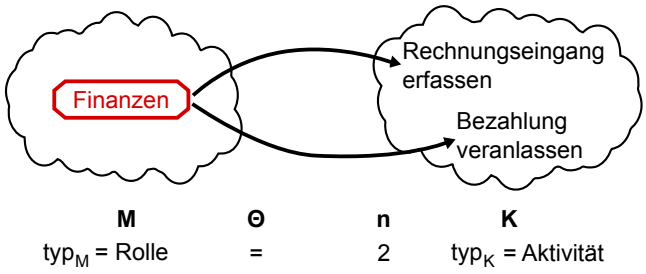


Abb. 6.10: Einer der assignMΘnK-Constraints bei Constraint e).

Formal gilt wegen $typ_K =$ „Aktivität“

- $K_{Finanzen} = \{Rechnungseingang\ erfassen,\ Rechnung\ prüfen,$
  $Bezahlung\ veranlassen\}.$

Mit $K = \{Rechnungseingang\ erfassen,\ Bezahlung\ veranlassen\}$ ist dann

- $|K_{Finanzen} \cap K| = 2,$

so dass der assignMΘnK-Constraint mit $\Theta \in$ ‘ $=$ ‘ und $n = 2$ erfüllt ist.

Alle anderen Rollen sind keiner der beiden Aktivitäten zugeordnet, so dass auch der zweite assignMΘnK-Constraint eingehalten wird.

Im Gegensatz zu assignMΘnK-Constraints können activateMΘnK-Constraints und entailmentMΘnK-Constraints erst zur Laufzeit des Prozesses durchgesetzt werden. Im Folgenden wird zunächst die Durchsetzung der activateMΘnK-Constraints betrachtet .

6.2 Durchsetzung von activateMΘnK-Constraints zur Laufzeit

Da im Folgenden stets der Prozess während der Ausführung betrachtet wird, wird von der vorherigen Durchsetzung, und der damit verbundenen Einhaltung, sämtlicher assignMΘnK-Constraints ausgegangen. Daher ist es an dieser Stelle interessant, die noch nicht durchgesetzten activateMΘnK-Constraints zu betrachten.

6.2.1 Rollenaktivierung und Aktivitätenausführung

Für die Durchsetzung der activateMΘnK-Constraints ist es zunächst notwendig, die Unterschiede zwischen der Aktivierung einer Rolle und der Ausführung einer Aktivität zu betrachten.

Im Fall der Rollenaktivierung ist stets davon auszugehen, dass ein bestimmter Benutzer eine, durch ihn spezifizierte, Rolle aktivieren möchte. Die Bedeutung von $activate(user, role)$ ist daher, dass der Benutzer $user$ beabsichtigt die Rolle $role$ zu aktivieren. Bei der Aktivitätenausführung beabsichtigt dagegen ein Benutzer die Ausführung einer bestimmten Aktivität in einer bestimmten Rolle. $execute(user, role, task)$ bedeutet daher, dass der Benutzer $user$ in der Rolle $role$ die Aktivität $task$ ausführen möchte.

An dieser Stelle kann bereits beobachtet werden, dass einer Aktivitätenausführung stets eine Rollenaktivierung vorausgehen muss. Da eine Aktivität immer durch einen Benutzer in einer bestimmten Rolle ausgeführt wird, muss der Benutzer immer zuerst die entsprechende Rolle aktivieren. Eine erfolgreiche Rollenaktivierung $activate(user, role)$ ist also stets Voraussetzung für eine erfolgreiche Aktivitätenausführung $execute(user, role, task)$.

6.2.2 Zeitpunkt der Durchsetzung

Im Fall der activateMΘnK-Constraints ist außerdem der Zeitpunkt ihrer Durchsetzung von enormer Bedeutung. Voraussetzung für eine erfolgreiche Rollenaktivierung oder Aktivitätenausführung ist stets, dass der Benutzer über entsprechende RBAC-Berechtigungen verfügt. Zunächst sind also die vergebenen RBAC-Berechtigungen zu überprüfen. Sind keine ausreichenden RBAC-Berechtigungen vorhanden, so wird die Aktivierung bzw. Ausführung sofort verboten.

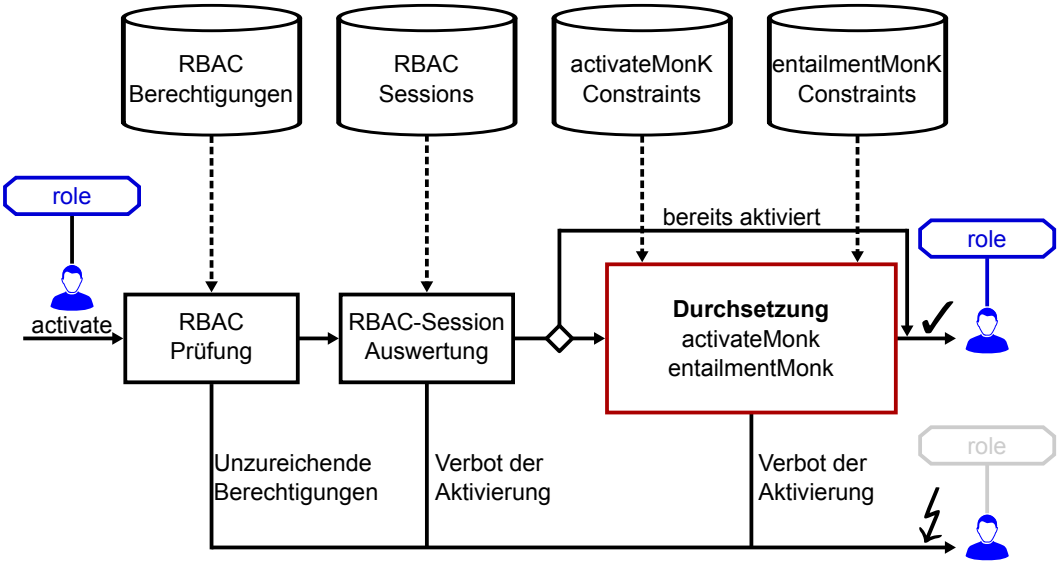


Abb. 6.11: Durchsetzung der activateMΘnK-Constraints bei Rollenaktivierung.

Wie bereits in Kapitel 5.2.3 erwähnt wurde, sind activateMΘnK-Constraints eng mit dem Session-Konzept von RBAC verknüpft. Insbesondere bei der Rollenaktivierung kann hierbei von den RBAC-Sessions profitiert werden. So kann bei einem Wunsch der Rollenaktivierung mit Hilfe der RBAC-Sessions zunächst überprüft werden, ob die entsprechende Rolle bereits aktiviert ist. Ist das der Fall, so besteht keine Notwendigkeit für weitere Prüfungen.

Sind ausreichende RBAC-Berechtigungen vorhanden und wird die Aktivierung oder Ausführung nicht ausdrücklich durch RBAC-Sessions verboten, so muss die Einhaltung der activateMΘnK-Constraints sicher gestellt werden. Wichtig ist hierbei, dass die activateMΘnK-Constraints vor der eigentlichen Rollenaktivierung bzw. Aktivitätenausführung durchgesetzt

werden. Nur wenn kein activateMΘnK-Constraint im Widerspruch zu der gewünschten Rollenaktivierung oder Aktivitätenausführung steht, kann diese tatsächlich stattfinden.

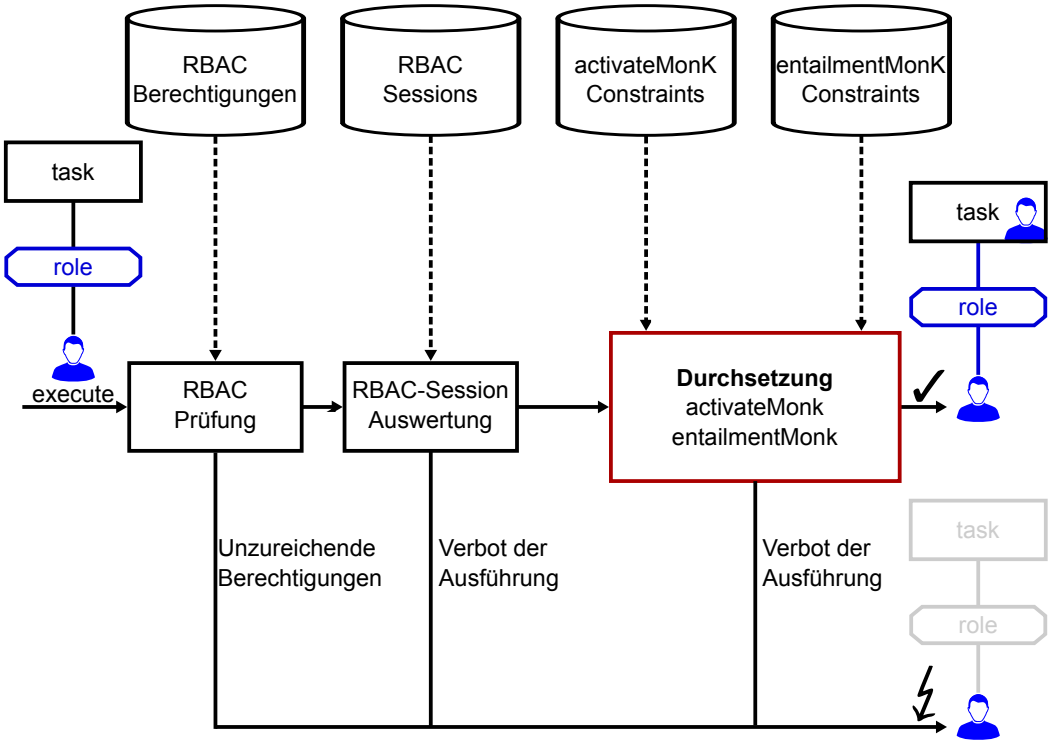


Abb. 6.12: Durchsetzung der activateMΘnK-Constraints bei Aktivitätenausführung.

Abb. 6.11 zeigt diesen Zeitpunkt der Durchsetzung der activateMΘnK-Constraints für eine Rollenaktivierung, Abb. 6.12 für eine Aktivitätenausführung. Der Unterschied besteht hierbei insbesondere in der positiven Entscheidungsfindung bei der Rollenaktivierung durch RBAC-Sessions. Wie die Durchsetzung der activateMΘnK-Constraints im Detail umgesetzt wird, wird in den beiden folgenden Kapiteln betrachtet. In den Abbildungen ist außerdem bereits die Durchsetzung der entailmentMΘnK-Constraints angedeutet, die Gegenstand der Betrachtungen in Kapitel 6.3 sein wird.

6.2.3 Formale Durchsetzung bei Rollenaktivierung

Möchte ein Benutzer eine Rolle aktivieren, so ist zuvor zu prüfen, ob durch die anstehende Rollenaktivierung activateMΘnK-Constraints verletzt würden. Wie bereits gesehen, ist bei

einer solchen Rollenaktivierung stets ein Benutzer $user$ und eine Rolle $role$ beteiligt. Es sind somit alle activateMΘnK-Constraints zu überprüfen, die eine Aussage über das Verhältnis der beiden Elemente $user$ und $role$ treffen. Bei der Rollenaktivierung können dabei genau zwei Arten von activateMΘnK-Constraints zutreffen:

1.) $activateMonK(M, \Theta, n, K)$ mit $user \in M \wedge role \in K$
2.) $activateMonK(M, \Theta, n, K)$ mit $role \in M \wedge user \in K$

Alle zutreffenden activateMΘnK-Constraints sind darauf hin zu überprüfen, ob sie die Aktivierung der Rolle $role$ durch den Benutzer $user$ verbieten. Wird dies durch keinen der activateMΘnK-Constraints verboten, so kann mit der tatsächlichen Aktivierung der Rolle fortgefahren werden. Das Aktivieren der Rolle wird dabei genau dann verboten, wenn bei mindestens einem der zutreffenden activateMΘnK-Constraints $n = 0$ gilt. In diesem Fall darf keine weitere Zuordnung zwischen Elementen der Menge M und Elementen der Menge K stattfinden. Alle der zutreffenden activateMΘnK-Constraints sind also auf diese Eigenschaft hin zu überprüfen.

Gilt für alle zutreffenden activateMΘnK-Constraints $n > 0$, so wird dem Benutzer $user$ die Aktivierung der Rolle $role$ gestattet. Dieser Vorgang hat Auswirkungen auf alle zutreffenden activateMΘnK-Constraints. Da jedem $m \in M$ höchstens Θ n viele Elemente aus K zugeordnet werden dürfen, sind die zutreffenden activateMΘnK-Constraints bei einer Rollenaktivierung für den weiteren Verlauf des Prozesses anzupassen. Aus jedem zutreffenden activateMΘnK-Constraint entstehen in Folge der Rollenaktivierung zwei neue activateMΘnK-Constraints, die diese Aktivierung widerspiegeln. In den beiden Fällen 1.) und 2.) entstehen jeweils die beiden folgenden activateMΘnK-Constraints:

1.) $activateMonK(\{user\}, \Theta, n-1, K \backslash \{role\})$
$activateMonK(M \backslash \{user\}, \Theta, n, K \backslash \{role\})$
2.) $activateMonK(\{role\}, \Theta, n-1, K \backslash \{user\})$
$activateMonK(M \backslash \{role\}, \Theta, n, K \backslash \{user\})$

Im Fall 1.) darf $user$ nur noch $n-1$ der verbleibenden Rollen ($K \backslash \{role\}$) aktivieren. Alle anderen Benutzer dürfen weiterhin n der verbleibenden Rollen aktivieren. Bei 2.) darf $role$ insgesamt nur noch $n-1$ mal durch die verbleibenden Benutzer ($K \backslash \{user\}$) aktiviert werden, während alle anderen Rollen der Menge M weiterhin n mal aktiviert werden dürfen. Dass aus der Menge K jeweils $user$ bzw. $role$ entfernt wird hat seine Begründung darin, dass jedes Element $k \in K$ höchstens einem Element $m \in M$ zugeordnet werden darf. Somit wird die mehrmalige Zuordnung eines Elementes $k \in K$ verhindert. Nach der Rollen-

aktivierung sind also diese neu entstehenden activateMΘnK-Constraints einzuhalten. Der zuvor gültige activateMΘnK-Constraint verliert seine Gültigkeit.

Entsteht dabei ein activateMΘnK-Constraint mit $M = \emptyset$ oder $K = \emptyset$, so entspricht dieser nicht der Definition der MΘnK-Constraints in Kapitel 5.2.1 und ist somit nicht weiter zu beachten. Dies hat seine Begründung darin, dass ein activateMΘnK-Constraint keine sinnvolle Aussage trifft, wenn eine der Mengen M oder K keine Elemente enthält. Ein activateMΘnK-Constraint mit $n < 0$ wird dagegen nicht entstehen, da in diesem Fall zuvor $n = 0$ gilt und somit die Rollenaktivierung nicht gestattet wird.

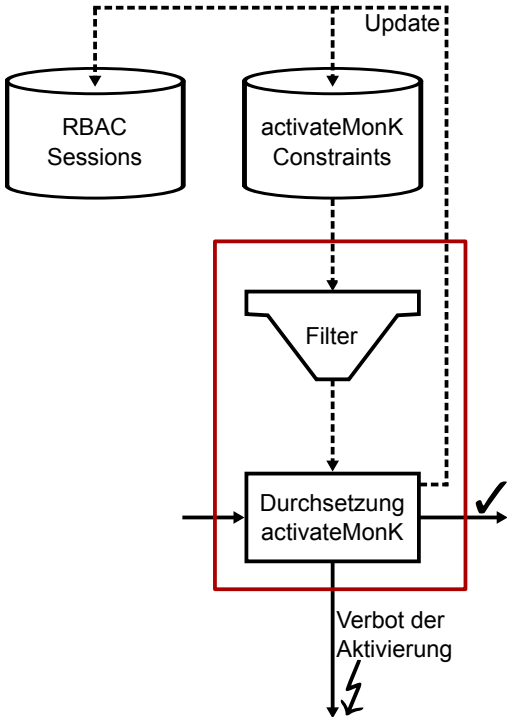


Abb. 6.13: Durchsetzung der activateMΘnK-Constraints vor der Rollenaktivierung bzw. Aktivitätenausführung.

Abb. 6.13 zeigt den genauen Ablauf der Durchsetzung der activateMΘnK-Constraints im Kontext von Abb. 6.11. Zunächst werden durch den Filter alle für die Rollenaktivierung relevanten activateMΘnK-Constraints ausfindig gemacht. Anschließend erfolgt die Durchsetzung der activateMΘnK-Constraints. Gilt für einen der zutreffenden activateMΘnK-Constraints $n = 0$, so ist die Aktivierung nicht möglich. Im erfolgreichen Fall werden dagegen im Anschluss an die Durchsetzung die RBAC-Sessions und die activateMΘnK-Constraints

aktualisiert. Die Aktualisierung der activateMΘnK-Constraints geschieht dabei gemäß den vorherigen Ausführungen, während in der entsprechenden RBAC-Session die aktivierte Rolle aufgenommen wird.

An dieser Stelle sei besonders auf die in Kapitel 5.3 modellierten Constraints hingewiesen. Für die dort aus den entailmentMΘnK-Constraints entstehenden activateMΘnK-Constraints gilt stets $n = 0$. Daher ist bei Einsatz dieser Constraints nicht zu befürchten, dass die Anzahl der activateMΘnK-Constraints durch die beschriebene Art der Durchsetzung unverhältnismäßig ansteigt.

6.2.4 Formale Durchsetzung bei Aktivitätenausführung

Die Durchsetzung der activateMΘnK-Constraints vor der Ausführung einer Aktivität kann im Grundsatz ähnlich erfolgen wie bei der Aktivierung einer Rolle. Dennoch handelt es sich hierbei um eine komplexere Aufgabe, da neben einem Benutzer und einer Rolle zusätzlich eine Aktivität beachtet werden muss. Bevor die Aktivität $task$ tatsächlich durch den Benutzer $user$ in der Rolle $role$ ausgeführt werden kann, muss auch an dieser Stelle zunächst geprüft werden, ob durch die Ausführung activateMΘnK-Constraints verletzt würden.

Um die Aktivität $task$ ausführen zu können, muss der Benutzer $user$ die Rolle $role$ aktiviert haben. Im Folgenden wird daher vorausgesetzt, dass diese Rollenaktivierung erfolgreich stattgefunden hat. Der Aktivitätenausführung ist also stets eine Rollenaktivierung gemäß Kapitel 6.2.3 voran zu stellen. Bei der Aktivitätenausführung sind dann weitere activateMΘnK-Constraints zu betrachten. Vor der Ausführung sind daher alle activateMΘnK-Constraints zu prüfen, die entweder eine Aussage über das Verhältnis von $role$ und $task$ oder über das Verhältnis von $user$ und $task$ treffen. Daher können bei der Aktivitätenausführung vier Arten von activateMΘnK-Constraints zutreffen:

1.) $activateMonK(M, \Theta, n, K)$ mit $user \in M \wedge task \in K$
2.) $activateMonK(M, \Theta, n, K)$ mit $task \in M \wedge user \in K$
3.) $activateMonK(M, \Theta, n, K)$ mit $role \in M \wedge task \in K$
4.) $activateMonK(M, \Theta, n, K)$ mit $task \in M \wedge role \in K$

Wie bei der Rollenaktivierung wird die Ausführung der Aktivität genau dann verboten, wenn für mindestens einen der zutreffenden activateMΘnK-Constraints $n = 0$ gilt. In den Fällen 1.) und 2.) bedeutet dies, dass dem Benutzer $user$ die Ausführung der Aktivität $task$ verbo-

ten wird. In den Fällen 3.) und 4.) ist dagegen die Ausführung der Aktivität $task$ in der Rolle $role$ untersagt.

Verbietet keiner der zutreffenden activateMΘnK-Constraints die Ausführung, d.h. $n > 0$ für alle zutreffenden activateMΘnK-Constraints, so werden diese angepasst. Jeder zutreffende activateMΘnK-Constraint wird dann durch zwei neue activateMΘnK-Constraints ersetzt, die die Ausführung der Aktivität widerspiegeln. Für die Fälle 1.) bis 4.) bedeutet dies:

1.) $activateMonK(\{user\}, \Theta, n-1, K \backslash \{task\})$
$activateMonK(M \backslash \{user\}, \Theta, n, K \backslash \{task\})$
2.) $activateMonK(\{task\}, \Theta, n-1, K \backslash \{user\})$
$activateMonK(M \backslash \{task\}, \Theta, n, K \backslash \{user\})$
3.) $activateMonK(\{role\}, \Theta, n-1, K \backslash \{task\})$
$activateMonK(M \backslash \{role\}, \Theta, n, K \backslash \{task\})$
4.) $activateMonK(\{task\}, \Theta, n-1, K \backslash \{role\})$
$activateMonK(M \backslash \{task\}, \Theta, n, K \backslash \{role\})$

Im Folgenden sind diese neu entstehenden activateMΘnK-Constraints einzuhalten, während der ursprüngliche activateMΘnK-Constraint ungültig wird. Wie bei der Rollenaktivierung gilt, dass ein neu entstehender activateMΘnK-Constraint $activateMonK(M, \Theta, n, K)$ mit $M = \emptyset$ oder $K = \emptyset$ nicht der Definition von MΘnK-Constraints gemäß Kapitel 5.2.1 entspricht. Ebenso kann auch hier kein activateMΘnK-Constraint mit $n = 0$ entstehen, da andernfalls bereits zuvor die Ausführung der Aktivität verboten würde. Abb. 6.13 zeigt wiederum den Ablauf der Durchsetzung der activateMΘnK-Constraint, dieses Mal im Kontext von Abb. 6.12.

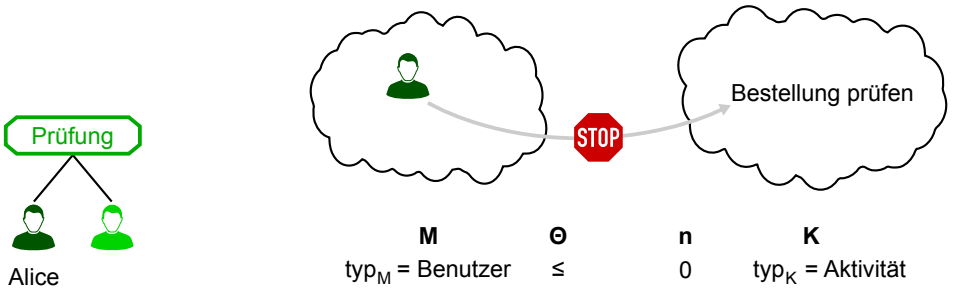
6.2.5 Durchsetzung am Fallbeispiel

Im Folgenden werden einige activateMΘnK-Constraints, wie sie im Fallbeispiel auftreten können, auf ihre Einhaltung hin überprüft. Diese activateMΘnK-Constraints stehen in der Regel nicht bereits vor Ausführung des Prozesses fest, sondern entstehen erst zur Laufzeit durch entailmentMΘnK-Constraints. Die betrachteten activateMΘnK-Constraints werden daher als gegeben angesehen. Wie diese im Fallbeispiel zustande kommen können, ist Gegenstand der Betrachtungen in Kapitel 6.3.

Nach activateMΘnK-Constraint

a) $activateMonK(\{Alice\}, \leq, 0, \{Bestellung\ prüfen\})$

darf Alice nicht die Aktivität „Bestellung prüfen“ ausführen. Möchte Alice diese Aktivität in der Rolle „Prüfung“ ausführen ($execute(Alice, Prüfung, Bestellung\ prüfen)$), so ist zur Laufzeit sicher zu stellen, dass dies nicht gestattet wird. Bei einem solchen Ausführungswunsch wird zunächst geprüft, ob Alice der Rolle „Prüfung“ angehört. Ist dies nicht der Fall, so wird die Ausführung bereits bei dieser Überprüfung der RBAC-Berechtigungen verboten. Ist Alice dagegen ein Benutzer der Rolle „Prüfung“ (Abb. 6.14a) und wird die Ausführung auch nicht durch die RBAC-Sessions verboten, so wird schließlich der zutreffende activateMΘnK-Constraint a) (Abb. 6.14b) durchgesetzt. Somit wird die Ausführung verboten und eine Aktualisierung der activateMΘnK-Constraints entfällt.



(a) Alice als ein Benutzer der Rolle „Prüfung“.

(b) activateMΘnK-Constraint a) verbietet die Ausführung von „Bestellung prüfen“ durch Alice.

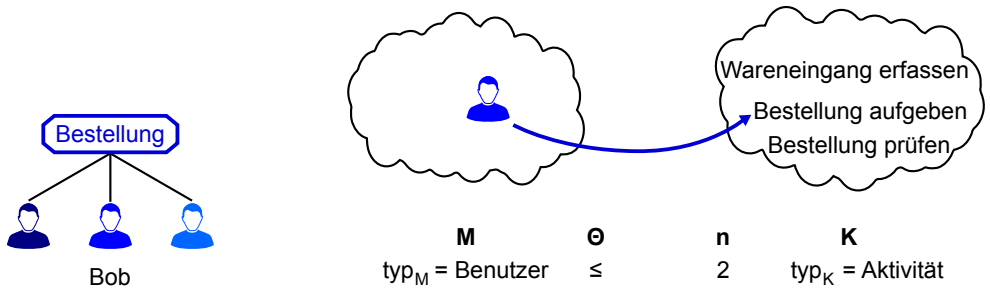
Abb. 6.14: Verbot der Ausführung von „Bestellung prüfen“ durch Alice.

Sind dagegen die beiden activateMΘnK-Constraints

b_1) $activateMonK(\{Bob\}, \leq, 0, \{Bestellung\ prüfen\})$

b_2) $activateMonK(\{Bob\}, \leq, 2, \{Bestellung\ prüfen,\ Bestellung\ aufgeben,$
$Wareneingang\ erfassen\})$

gegeben, so verbietet b_1), analog zu activateMΘnK-Constraint a), den Ausführungswunsch $execute(Bob, Bestellung, Bestellung\ prüfen)$. Zu beachten ist, dass bei diesem Ausführungswunsch zwar beide activateMΘnK-Constraints b_1) und b_2) zutreffen, bei der tatsächlichen Ausführung jedoch nur b_1) verletzt würde. Dass b_2) die Ausführung gestatten würde ist unerheblich, da bereits ein zutreffender activateMΘnK-Constraint mit $n = 0$ ausreicht, damit die Ausführung nicht stattfinden darf.



(a) Bob als ein Benutzer der Rolle „Bestellung“.

(b) activateMΘnK-Constraint b_2) bei Ausführung von „Bestellung aufgeben“ durch Bob.

Abb. 6.15: Ausführung von „Bestellung aufgeben“ durch Bob.

Möchte Bob dagegen die Aktivität „Bestellung aufgeben“ in der Rolle „Bestellung“ ausführen ($execute(Bob, Bestellung, Bestellung\ aufgeben)$), so ist zunächst zu prüfen, ob Bob der Rolle „Bestellung“ angehört. Ist das der Fall (Abb. 6.15a), und wird die vorausgehende Rollenaktivierung $activate(Bob, Bestellung)$ gemäß Kapitel 6.2.3 erfolgreich abgeschlossen, so ist der auf diese Aktivitätenausführung zutreffende activateMΘnK-Constraint b_2) (Abb. 6.15b) durchzusetzen. Da bei diesem activateMΘnK-Constraint $n = 2$ gilt, wird die Ausführung der Aktivität durch Bob gestattet. Durch die Ausführung verliert der activateMΘnK-Constraint b_2) seine Gültigkeit. Stattdessen entstehen, gemäß den Ausführungen in Kapitel 6.2.4, die zwei neuen activateMΘnK-Constraints

- $activateMonK(\{Bob\}, \leq, 1, \{Bestellung\ prüfen,\ Wareneingang\ erfassen\})$ und
- $activateMonK(\{\}, \leq, 2, \{Bestellung\ prüfen,\ Wareneingang\ erfassen\})$.

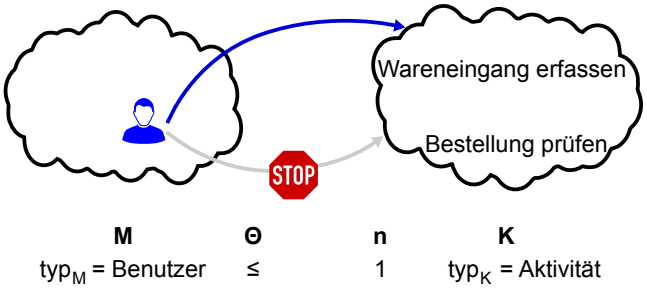


Abb. 6.16: Neu entstehender activateMΘnK-Constraint: Bob darf höchstens eine der zwei Aktivitäten aus K ausführen.

Dabei trifft der zweite der activateMΘnK-Constraints wegen $M = \emptyset$ keine sinnvolle Aussage und muss somit nicht weiter beachtet werden. Der erste dieser beiden entstehenden activateMΘnK-Constraints wird in Abb. 6.16 veranschaulicht.

Neben den activateMΘnK-Constraints können auch entailmentMΘnK-Constraints erst zur Laufzeit des Prozesses durchgesetzt werden. Diese werden im Folgenden betrachtet.

6.3 Durchsetzung von entailmentMΘnK-Constraints zur Laufzeit

6.3.1 Formale Durchsetzung

Gemäß Kapitel 5.2.4 hat ein entailmentMΘnK-Constraint die Form

- $activateMonK_1(M_1, =, 1, K_1) \Rightarrow activateMonK_2(M_2, \leq, n_2, K_2)$

Dabei muss der activateMΘnK-Constraint $activateMonK_2$ ab dem Zeitpunkt eingehalten werden, wenn einem Element $m \in M_1$ genau ein Element aus K_1 zugeordnet wird. Im Gegensatz zu activateMΘnK-Constraints werden entailmentMΘnK-Constraints erst dann durchgesetzt, wenn eine Rollenaktivierung ($activate(user, role)$) oder eine Aktivitätenausführung ($execute(user, role, task)$) keinen activateMΘnK-Constraints widerspricht und somit tatsächlich stattfindet.

In Abhängigkeit davon ob eine Rollenaktivierung oder eine Aktivitätenausführung stattfindet, sind unterschiedliche entailmentMΘnK-Constraints durchzusetzen. Im Fall einer erfolgreichen Rollenaktivierung sind alle entailmentMΘnK-Constraints durchzusetzen mit

- $user \in M_1 \wedge role \in K_1$ oder
- $role \in M_1 \wedge user \in K_1$.

Findet dagegen eine Aktivitätenausführung statt, so sind zusätzlich alle entailmentMΘnK-Constraints durchzusetzen mit

- $role \in M_1 \wedge task \in K_1$ oder
- $task \in M_1 \wedge role \in K_1$ oder
- $user \in M_1 \wedge task \in K_1$ oder
- $task \in M_1 \wedge user \in K_1$.

Bei allen zutreffenden entailmentMΘnK-Constraints wird in Folge der Rollenaktivierung oder Aktivitätenausführung die Bedingung in $activateMonK_1$ erfüllt, da einem Element $m \in M$ ein Element $k \in K$ zugeordnet wird. Alle zutreffenden entailmentMΘnK-Constraints müssen also durchgesetzt werden. Dies wird folgendermaßen realisiert:

Im weiteren Verlauf des Prozesses sind alle activateMΘnK-Constraints $activateMonK_2$ der zutreffenden entailmentMΘnK-Constraints einzuhalten. Dabei ist zu beachten, dass in den activateMΘnK-Constraints $activateMonK_2$ die Platzhalter

- $executingUser$,
- $activatingUser$,
- $executedRole$,
- $executedTask$

auftreten können. Diese sind zuvor durch die entsprechenden, in $activateMonK_1$ zugeordneten, Elemente zu ersetzen. Da entailmentMΘnK-Constraints eine „wenn-dann"-Semantik verkörpern und die Bedingung (die Zuordnung eines Elements $m \in M_1$ an ein Element $k \in K_1$) somit erfüllt wurde, muss der ursprüngliche entailmentMΘnK-Constraint nicht weiter beachtet werden. Die Durchsetzung der neu eingesetzten activateMΘnK-Constraints $activateMonK_2$ geschieht im weiteren Verlauf des Prozesses auf Basis der Ausführungen in Kapitel 6.2. Die neu eingesetzten activateMΘnK-Constraints müssen somit bei auf sie zutreffenden Rollenaktivierungen oder Aktivitätenausführungen durchgesetzt werden.

Abb. 6.17 zeigt die Durchsetzung der entailmentMΘnK-Constraints im Anschluss an die Durchsetzung der activateMΘnK-Constraints. Diese Durchsetzungen sind dabei in den Kontext der Aktivitätenausführung in Abb. 6.12 zu setzen. Hierbei macht wiederum ein Filter alle auf die Aktivitätenausführung zutreffenden entailmentMΘnK-Constraints ausfindig. Wird die Ausführung durch die activateMΘnK-Constraints nicht verboten, so werden die Mengen der activateMΘnK-Constraints und entailmentMΘnK-Constraints angepasst: Die zutreffenden entailmentMΘnK-Constraints werden entfernt und die entstehenden activateMΘnK-Constraints $activateMonK_2$ werden zur Menge der durchzusetzenden activateMΘnK-Constraints hinzugefügt. Die Durchsetzung der entailmentMΘnK-Constraints bei einer Rollenaktivierung erfolgt analog.

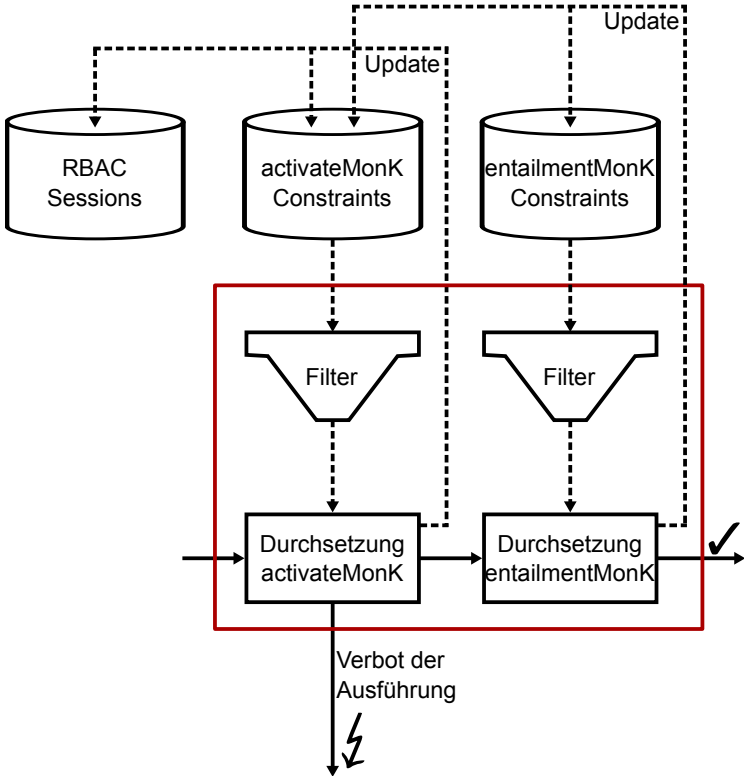


Abb. 6.17: Durchsetzung der entailmentMΘnK-Constraints im Anschluss an die Durchsetzung der activateMΘnK-Constraints.

6.3.2 Durchsetzung am Fallbeispiel

In Kapitel 6.2.5 wurden verschiedene activateMΘnK-Constraints als gegeben angesehen. Diese activateMΘnK-Constraints entstehen aus entailmentMΘnK-Constraints zur Laufzeit des Prozesses. Im Folgenden werden entailmentMΘnK-Constraints im Fallbeispiel formuliert. Finden dann bestimmte Rollenaktivierungen oder Aktivitätenausführungen statt, so entstehen die exemplarisch in Kapitel 6.2.5 betrachteten activateMΘnK-Constraints.

Der entailmentMΘnK-Constraint

a) $activateMonK_1(users(Bestellung\ schreiben), =, 1, \{Bestellung\ schreiben\})$
$\Rightarrow activateMonK_2(\{executingUser\}, \leq, 0, \{Bestellung\ pr\ddot{u}fen\})$

formuliert gemäß Kapitel 5.3.3 ein dynamisches Separation of Duty auf „Bestellung schreiben“ und „Bestellung prüfen“. Aus Abb. 6.18 geht hervor dass gilt

- $users(Bestellung\ schreiben) = users(Bestellung) \cup users(Produktion)$

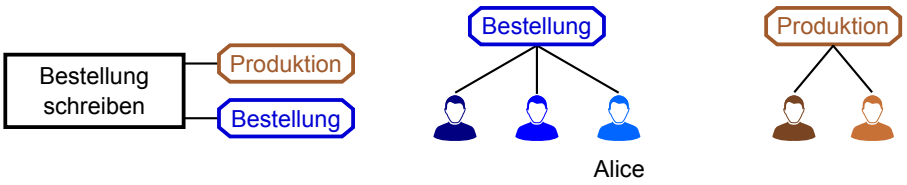


Abb. 6.18: Prozessausschnitt und die Benutzer der Rollen „Bestellung“ und „Produktion“.

Bei dem Ausführungswunsch $execute(Alice, Bestellung, Bestellung\ schreiben)$ ist dann zunächst, gemäß den Ausführungen in Kapitel 6.2, zu prüfen, ob dieser den bestehenden RBAC-Berechtigungen und activateMΘnK-Constraints entspricht. Verlaufen diese Prüfungen erfolgreich, so darf Alice die Aktivität ausführen. Es gilt dann für den entailmentMΘnK-Constraint a)

- $Alice \in M_1 = users(Bestellung\ schreiben)$ (siehe Abb. 6.18) und
- $Bestellung\ schreiben \in K_1 = \{Bestellung\ schreiben\}$.

Somit trifft der entailmentMΘnK-Constraint a) auf diese Aktivitätenausführung zu und muss durchgesetzt werden. Abb. 6.19 zeigt den entailmentMΘnK-Constraint a) bei der Ausführung von „Bestellung schreiben“ durch Alice.

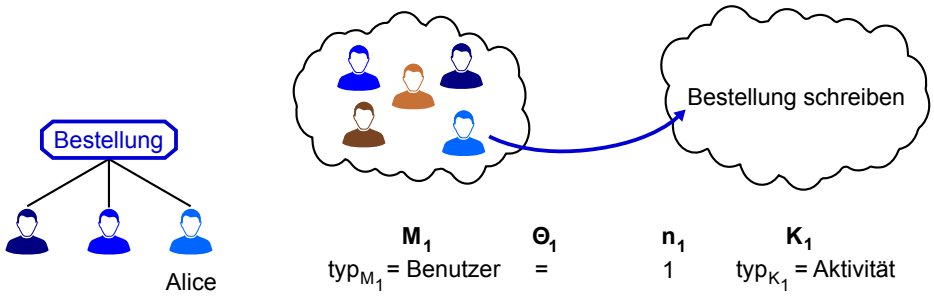


Abb. 6.19: entailmentMΘnK-Constraint a) bei Ausführung von „Bestellung schreiben“ durch Alice.

Der entailmentMΘnK-Constraint a) wird dann durchgesetzt, indem zunächst die Platzhalter in $activateMonK_2$ ersetzt werden. $executingUser$ entspricht dabei dem Benutzer Alice, da

die Aktivität durch diesen Benutzer ausgeführt wird. Der resultierende activateMΘnK-Constraint ist somit

- $activateMonK_2(\{Alice\}, \leq, 0, \{Bestellung\ prüfen\})$

Dieser muss im weiteren Verlauf des Prozesses eingehalten werden. Die Menge der durchzusetzenden activateMΘnK-Constraint wird also um diesen resultierenden activateMΘnK-Constraint erweitert. Die weitere Durchsetzung dieses activateMΘnK-Constraints erfolgt dann gemäß Kapitel 6.2.5. Der ursprüngliche entailmentMΘnK-Constraint a) wird dagegen aus der Menge der durchzusetzenden entailmentMΘnK-Constraints entfernt.

Weiter wird der komplexere entailmentMΘnK-Constraint

b) $activateMonK_1(\bigcup_{t \in tasks} users(t), =, 1, tasks)$
$\Rightarrow activateMonK_2(\{executingUser\}, \leq, |tasks| - 2, tasks \backslash \{executedTask\})$ mit
- $tasks = \{Bestellung\ schreiben,\ Bestellung\ prüfen,$
$Bestellung\ aufgeben,\ Wareneingang\ erfassen\}$

betrachtet, der Operational Dynamic Separation of Duty gemäß Kapitel 5.4 modelliert. Ein Benutzer darf durch diesen entailmentMΘnK-Constraint höchstens drei der vier Aktivitäten aus $tasks$ ausführen. Dabei gilt im Fallbeispiel

- $\bigcup_{t \in tasks} users(t) = users(Bestellung) \cup users(Produktion) \cup users(Prüfung),$

was auch aus Abb. 6.20 hervorgeht.

Führt Bob „Bestellung schreiben“ aus ($execute(Bob, Bestellung, Bestellung\ schreiben)$), so wurde wiederum bereits zuvor sichergestellt, dass die nötigen RBAC-Berechtigungen existieren und keine activateMΘnK-Constraints verletzt werden. Es gilt also

- $Bob \in users(Bestellung\ schreiben) \subseteq M_1 = \bigcup_{t \in tasks} users(t)$ und
- $Bestellung\ schreiben \in K_1 = tasks,$

womit der entailmentMΘnK-Constraint b) auf $execute(Bob, Bestellung, Bestellung\ schreiben)$ zutrifft und durchgesetzt werden muss. Abb. 6.21a zeigt den entailmentMΘnK-Constraint b) bei der Ausführung von „Bestellung schreiben“ durch Bob, während Abb. 6.21b die Rolle „Bestellung“ mit den zugeordneten Benutzern zeigt.

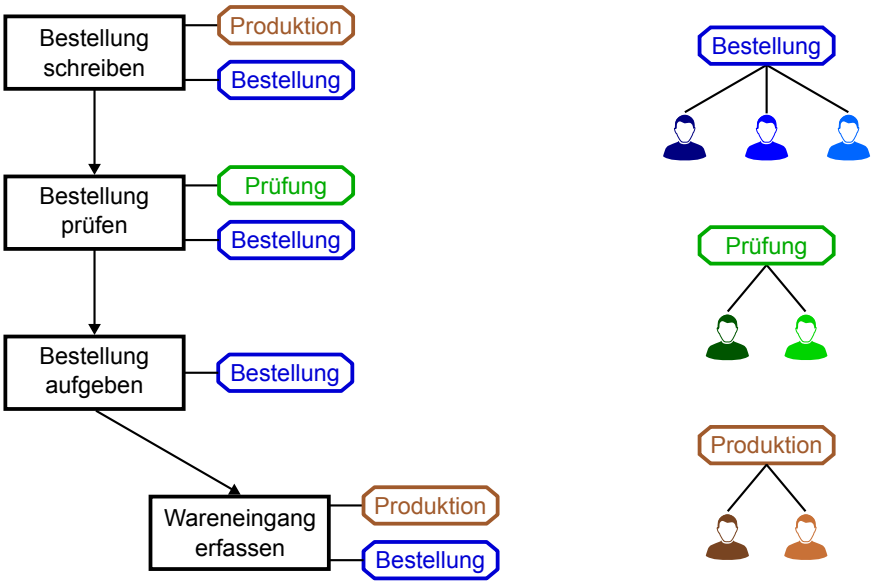
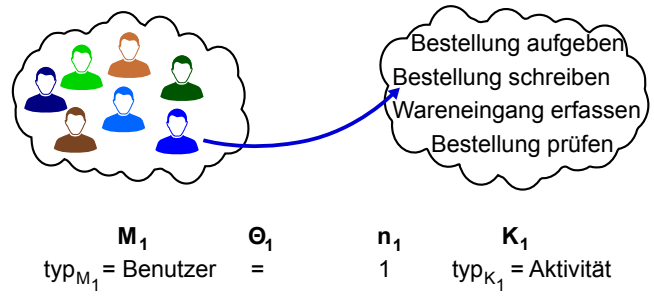
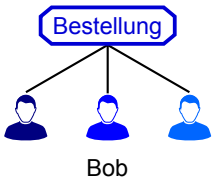


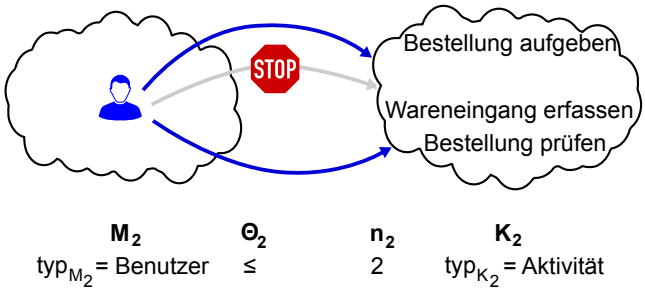
Abb. 6.20: Ausschnitt aus dem Fallbeispiel-Prozess und zugehörige Rollenzuordnungen.



(a) entailmentMΘnK-Constraint b) bei Ausführung von „Bestellung schreiben“ durch Bob.



(b) Bob als ein Benutzer der Rolle „Bestellung“.



(c) Aus entailmentMΘnK-Constraint b) resultierender Folgeconstraint.

Abb. 6.21: entailmentMΘnK-Constraint b) bei Ausführung von „Bestellung schreiben“ durch Bob und der resultierende Folgeconstraint.

Die Platzhalter in activateMΘnK-Constraint $activateMonK_2$ entsprechen dann dem ausführenden Benutzer Bob ($executingUser = Bob$) und der ausgeführten Aktivität „Bestellung schreiben“ ($executedTask = Bestellung\ schreiben$). Der resultierende activateMΘnK-Constraint ist somit

- $activateMonK_2(\{Bob\}, \leq, 2, \{Bestellung\ prüfen,\ Bestellung\ aufgeben,\ Wareneingang\ erfassen\})$

Dieser muss im weiteren Verlauf des Prozesses gemäß Kapitel 6.2 durchgesetzt werden und wird hierzu zur Menge der durchzusetzenden activateMΘnK-Constraints hinzugefügt. Dieser activateMΘnK-Constraint wird in Abb. 6.21c veranschaulicht. Der entailmentMΘnK-Constraint b) wurde dagegen erfüllt und wird aus der Menge der durchzusetzenden entailmentMΘnK-Constraints entfernt.

6.4 Zusammenfassung

Ziel der Durchsetzung ist es, die Einhaltung aller MΘnK-Constraints eines Prozesses zu erzwingen. Vorgänge wie die Berechtigungsvergabe, Aktivitätenausführungen und Rollenaktivierung müssen daher den MΘnK-Constraints entsprechen.

Zur Modellierzeit ist die Durchsetzung der assignMΘnK-Constraints, durch die die Berechtigungsvergabe eingeschränkt wird, möglich. Die mittels RBAC vergebenen Berechtigungen müssen somit den assignMΘnK-Constraints entsprechen. Die Überprüfung der Konformität der Berechtigungen mit den assignMΘnK-Constraints erfolgt mit Hilfe der Mengenlehre.

Zur Laufzeit des Prozesses müssen dagegen die activateMΘnK-Constraints und entailmentMΘnK-Constraints durchgesetzt werden. Bei Vorliegen eines Ausführungs- oder Aktivierungswunsches sind daher alle zutreffenden activateMΘnK-Constraints durchzusetzen. Verbietet einer der activateMΘnK-Constraints die gewünschte Ausführung oder Aktivierung, so wird der Vorgang abgebrochen. Andernfalls entstehen neue activateMΘnK-Constraints, die die genehmigte Ausführung oder Aktivierung widerspiegeln und im weiteren Verlauf des Prozesses eingehalten werden müssen. Ebenso entstehen in diesem Fall aus allen zutreffenden entailmentMΘnK-Constraints neue activateMΘnK-Constraints. Auch diese müssen bei weiteren Ausführungen oder Aktivierungen eingehalten werden.

7 Validierung der MΘnK-Constraints

Dieses Kapitel betrachtet die Validierung der MΘnK-Constraints. Dabei wird zunächst auf die Ziele der Validierung eingegangen. Anschließend wird die Validierung einzelner MΘnK-Constraints und die MΘnK-Constraint-übergreifende Validierung betrachtet. Dabei können wiederum assignMΘnK-Constraints zur Modellierzeit validiert werden, während die Validierung der activateMΘnK-Constraints und entailmentMΘnK-Constraints zur Laufzeit des Prozesses erfolgt.

7.1 Ziel der Validierung

Beim Einsatz von Authorization Constraints in WfMS können unterschiedliche Arten von Konflikten und Widersprüchen auftreten. Diese Konflikte werden spätestens dann sichtbar, wenn widersprüchliche Constraints durchgesetzt werden sollen. Ziel der Validierung der MΘnK-Constraints ist es, diese Konflikte frühzeitig zu erkennen und somit einen möglichst konfliktfreien Ablauf des Prozesses zu gewährleisten. Je nach Art der MΘnK-Constraints werden diese, wie in Kapitel 6 beschrieben, entweder zur Modellierzeit oder zur Laufzeit durchgesetzt. Um Konflikte bei der Durchsetzung der Constraints zu vermeiden, muss die Validierung also bereits zuvor stattfinden.

Erste Konflikte können bereits bei der Abbildung der gewünschten Authorization Constraints auf MΘnK-Constraints auftreten. Da Constraints durch Constraintmodellierer manuell, also unabhängig von den Betrachtungen in Kapitel 5.3, auf MΘnK-Constraints abgebildet werden können, kann die Fehlerfreiheit bei der Modellierung einzelner Constraints nicht ausgeschlossen werden. So ist es möglich, dass sich die MΘnK-Constraints eines Constraints widersprechen. Derartige Konflikte sollen nach Möglichkeit nicht auftreten bzw. müssen frühzeitig erkannt werden.

Da in einem Prozess üblicherweise mehrere Constraints formuliert werden, können außerdem Widersprüche zwischen diesen Constraints bestehen. Ein weiteres Ziel ist es daher,

Constraint-übergreifende Widersprüche zu finden. Alle Maßnahmen zielen dabei darauf ab, einen mit Constraints angereicherten Prozess möglichst ohne Konflikte auszuführen zu können.

Gegeben ist im Folgenden eine RBAC-Berechtigungsvergabe und die MΘnK-Constraints des Prozesses: assignMΘnK-Constraints, activateMΘnK-Constraint und entailmentMΘnK-Constraints. Zu beachten ist, dass die Validierung der MΘnK-Constraints immer vor der Durchsetzung stattfindet. Dabei muss beachtet werden, dass assignMΘnK-Constraints bereits zur Modellierziet validiert werden müssen, während dies bei activateMΘnK-Constraints und entailmentMΘnK-Constraints erst zur Laufzeit des Prozesses möglich ist. Abb. 7.1 zeigt schematisch den Ablauf von Durchsetzung und Validierung der MΘnK-Constraints zur Modellierzeit und zur Laufzeit des Prozesses.

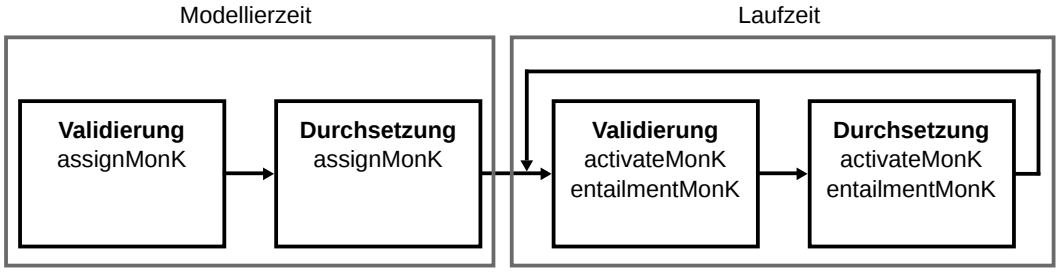


Abb. 7.1: Validierung und Durchsetzung der MΘnK-Constraints zur Modellierzeit und zur Laufzeit des Prozesses.

7.2 Validierung einzelner MΘnK-Constraints

Zunächst ist es sinnvoll, einzelne MΘnK-Constraints daraufhin zu überprüfen, ob deren Erfüllbarkeit möglich ist. Dabei ist ein MΘnK-Constraint dann **erfüllbar**, wenn dessen Einhaltung möglich ist. Das bedeutet zum einen, dass ein MΘnK-Constraint keine in sich widersprüchliche Aussage treffen darf und zum anderen, dass zur Erfüllung eines MΘnK-Constraints unter Umständen bestimmte RBAC-Berechtigungen gegeben sein müssen. Da bei der Ausführung eines Prozesses stets alle MΘnK-Constraints eingehalten werden müssen, ist es notwendig, dass jeder einzelne MΘnK-Constraints erfüllbar ist. Im Folgenden wird betrachtet, inwiefern derartige Widersprüche innerhalb einzelner assignMΘnK-Constraints, activateMΘnK-Constraints und entailmentMΘnK-Constraints auftreten können.

Erfüllbarkeit einzelner assignMΘnK-Constraints

Zur Betrachtung der Erfüllbarkeit einzelner assignMΘnK-Constraints wird eine Fallunterscheidung vorgenommen. Dabei wird jeweils ein assignMΘnK-Constraint

- $assignMonK(M, \Theta, n, K)$

betrachtet mit

i) $\Theta \in \{=, \geq\}$
ii) $\Theta \in \{\leq\}$

i) Die Aussage eines assignMΘnK-Constraints $assignMonK(M, \Theta, n, K)$ mit $\Theta \in \{=, \geq\}$ ist, dass jedem $m \in M$ genau n viele bzw. mindestens n viele Elemente aus K zugeordnet werden müssen. Zusätzlich darf jedes Element $k \in K$ höchstens einem $m \in M$ zugeordnet werden. Damit ein solcher assignMΘnK-Constraint erfüllt werden kann, muss die Menge K mindestens $|M| \cdot n$ viele Elemente enthalten: $|M| \cdot n \leq |K|$. Denn ist $|M| \cdot n > |K|$, so ist es nicht möglich, jedem $m \in M$ genau n viele bzw. mindestens n viele Elemente aus K zuzuordnen. Dadurch wäre der assignMΘnK-Constraint nie erfüllbar. Abb. 7.2 zeigt einen solchen nicht erfüllbaren assignMΘnK-Constraint, da die Menge K weniger als $|M| \cdot n = 4$ Benutzer enthält.

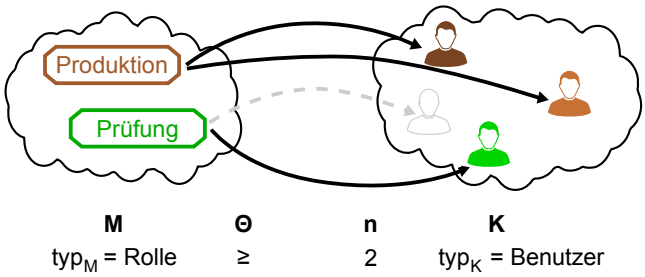


Abb. 7.2: Nicht erfüllbarer assignMΘnK-Constraint: K enthält zu wenig Benutzer.

ii) Für $\Theta \in \{\leq\}$ kann ein solcher Widerspruch nicht bestehen, da damit eine Maximalzuordnung ausgedrückt wird. Der assignMΘnK-Constraint kann somit stets dadurch erfüllt werden, dass keine Elemente aus M und K einander zugeordnet werden. Anders als in i) besteht damit keine Notwendigkeit, eine Mindestgröße für die Menge K zu fordern.

Weitere als die in i) beschriebenen Widersprüche können innerhalb einzelner assignMΘnK-Constraints nicht auftreten. Gilt für einen assignMΘnK-Constraint $assignMonK(M, \Theta, n, K)$ also entweder

- $\Theta \in \{=, \geq\} \wedge |M| \cdot n \leq |K|$ oder
- $\Theta \in \{\leq\}$,

so existieren immer Zuordnungen zwischen M und K, so dass dieser einzelne assignMΘnK-Constraint eingehalten werden kann. Ob die RBAC-Berechtigungen tatsächlich konform dem gegebenen assignMΘnK-Constraint sind, wird dann erst bei der, noch zur Modellierzeit stattfindenden, Durchsetzung des assignMΘnK-Constraints deutlich.

Die Validierung einzelner assignMΘnK-Constraints wurde im Rahmen dieser Arbeit implementiert. Details zur Implementierung können Kapitel 8 entnommen werden.

Erfüllbarkeit einzelner activateMΘnK-Constraints

Ein activateMΘnK-Constraint $activateMonK(M, \leq, n, K)$ schränkt die Aktivierung bzw. Ausführung der Elemente in M und K ein. Voraussetzung für solche Aktivierungen oder Ausführungen sind stets entsprechende RBAC-Berechtigungen. Bei einem einzelnen activateMΘnK-Constraint kann jedoch kein Widerspruch auftreten – unabhängig davon, ob zwischen den Elementen der Mengen M und K RBAC-Berechtigungen vorliegen oder nicht. Dies hat seine Begründung darin, dass ein activateMΘnK-Constraint durch die Einschränkung $\Theta \in \{\leq\}$ stets eine Maximalzuordnung ausdrückt. Der activateMΘnK-Constraint kann immer eingehalten werden, indem keine Aktivierungen oder Ausführungen vorgenommen werden, auf die dieser zutrifft.

Erfüllbarkeit einzelner entailmentMΘnK-Constraints

Da ein entailmentMΘnK-Constraint

- $activateMonK_1(M_1, =, 1, K_1) \Rightarrow activateMonK_2(M_2, \leq, n_2, K_2)$

eine „wenn-dann“-Semantik besitzt, kann auch bei diesem kein Widerspruch auftreten. Trifft eine Aktivierung oder Ausführung auf den activateMΘnK-Constraint $activateMonK_1$ zu, so wird der Folgeconstraint aktiv und muss im weiteren Verlauf des Prozesses eingehalten werden. Trifft der entailmentMΘnK-Constraint dagegen nie zu, so muss auch der Folgeconstraint nie betrachtet werden.

Wenn sichergestellt wurde, dass alle einzelnen MΘnK-Constraints erfüllt werden können, so sind diese anschließend auf MΘnK-Constraint-übergreifende Konflikte zu untersuchen.

7.3 MΘnK-Constraint-übergreifende Validierung

Um MΘnK-Constraint-übergreifende Widersprüche identifizieren zu können, muss zunächst herausgearbeitet werden, zwischen welchen verschiedenen MΘnK-Constraints überhaupt Widersprüche auftreten können. Ein Widerspruch zwischen zwei oder mehr MΘnK-Constraints liegt dabei dann vor, wenn unter keinen Umständen die Einhaltung aller betrachteten MΘnK-Constraints möglich ist.

assignMΘnK-Constraints

Da assignMΘnK-Constraints die mittels RBAC vergebbaren Berechtigungen einschränken und da $\Theta \in \{\leq, =, \geq\}$ gilt, lassen sich mit zwei oder mehr assignMΘnK-Constraints unumgehbare Widersprüche formulieren. So ist es leicht vorstellbar, dass ein assignMΘnK-Constraint die Zuordnung eines Benutzers zu einer Rolle fordert, während ein anderer assignMΘnK-Constraint genau diese Zuordnung verbietet. Die assignMΘnK-Constraints

- $assignMonK(\{Alice\}, =, 1, \{Bestellung\})$ und
- $assignMonK(\{Alice\}, =, 0, \{Bestellung\})$

formulieren einen derartigen Widerspruch. Diese werden in Kapitel 7.4 noch genauer betrachtet werden.

activateMΘnK-Constraints

Ein activateMΘnK-Constraint $activateMonK(M, \leq, n, K)$ schränkt die Aktivierung oder Ausführung der Elemente in M und K ein. Da bei allen activateMΘnK-Constraints $\Theta \in \{\leq\}$ gilt, werden dabei jedoch stets Maximalzuordungen ausgedrückt. Daher ist es nicht möglich, zwei oder mehr activateMΘnK-Constraints zu formulieren, so dass die Einhaltung aller dieser activateMΘnK-Constraints unter keinen Umständen möglich ist. Dies ist darin begründet, dass jeder activateMΘnK-Constraint $activateMonK(M, \Theta, n, K)$ erfüllt ist, solange keine Aktivierung oder Ausführung stattfindet, die sich auf Elemente in M und K

bezieht. Es ist daher niemals unmöglich, alle activateMΘnK-Constraints gleichzeitig zu erfüllen: durch den Verzicht auf jegliche Rollenaktivierung oder Aktivitätenausführung bleiben stets alle activateMΘnK-Constraints erfüllt.

Damit entsteht jedoch die Problematik, dass unter Umständen die Ausführung von Aktivitäten unmöglich wird und somit der Prozess nicht zu Ende gebracht werden kann. Weitere Betrachtungen hierzu werden in Kapitel 7.5 vorgenommen.

Ist eine Menge von widerspruchsfreien assignMΘnK-Constraints gegeben, so ist die Formulierung hierzu widersprüchlicher activateMΘnK-Constraint nicht möglich. Die Erfüllung jedes activateMΘnK-Constraints $activateMonK(M, \leq, n, K)$ ist unabhängig von allen assignMΘnK-Constraints möglich, indem auf jegliche Aktivierungen oder Ausführungen verzichtet wird.

entailmentMΘnK-Constraints

Ebenso kann zwischen zwei oder mehr entailmentMΘnK-Constraints kein Widerspruch auftreten, da bei diesen eine „wenn-dann"-Semantik vorliegt. Wird ein activateMΘnK-Constraint $activateMonK_1$ eines entailmentMΘnK-Constraints erfüllt, so geschieht dies unabhängig von allen anderen entailmentMΘnK-Constraints.

Auch wenn entailmentMΘnK-Constraints zusammen mit widerspruchsfreien assignMΘnK-Constraints und activateMΘnK-Constraints auftreten, entsteht kein Widerspruch. Schränken die assignMΘnK-Constraints oder activateMΘnK-Constraints die vergebenen Berechtigungen derart ein, dass der activateMΘnK-Constraint $activateMonK_1$ eines entailmentMΘnK-Constraints nicht erfüllt werden kann, so ist dies auf Grund der „wenn-dann"-Semantik des entailmentMΘnK-Constraints unerheblich. In diesem Fall wird lediglich niemals der activateMΘnK-Constraint $activateMonK_2$ des entailmentMΘnK-Constraints aktiv.

Zusammenfassend zeigt Abb. 7.3, dass nur zwischen assignMΘnK-Constraints solche Widersprüche auftreten können, die die Einhaltung aller MΘnK-Constraints unmöglich machen. Bei Auftreten aller anderen MΘnK-Constraint-Kombinationen können derartige Widersprüche nicht auftreten. Dies hat seinen Ursprung in der Einschränkung $\Theta \in \{\leq\}$ bei activateMΘnK-Constraints und in der „wenn-dann"-Semantik bei entailmentMΘnK-Constraints.

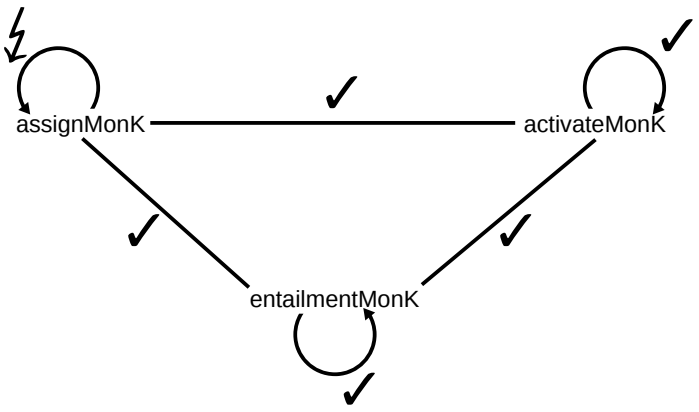


Abb. 7.3: MΘnK-Constraint-übergreifende Konflikte bei Einsatz unterschiedlicher MΘnK-Constraint-Kombinationen.

7.4 Validierung zur Modellierzeit

Liegt eine Menge von assignMΘnK-Constraints vor, so werden diese gemäß Kapitel 6.1 noch vor der Ausführung des Prozesses durchgesetzt. Da die Validierung noch vor der Durchsetzung stattfinden muss, findet auch diese zur Modellierzeit statt. Das Ziel ist es hierbei, sich widersprechende assignMΘnK-Constraints zu finden. Ohne eine derartige Validierung treten die Widersprüche erst bei der Durchsetzung der assignMΘnK-Constraints in Erscheinung. Durch eine vorherige assignMΘnK-Constraint-übergreifende Validierung soll sichergestellt werden, dass auch die Herkunft eventueller Widersprüche herausgefunden werden kann. Der Prozess- bzw. Constraintmodellierer kann somit besser auf die Konfliktsituation reagieren und die Widersprüche auflösen.

7.4.1 NP-Vollständigkeit von assignMΘnK

Gegeben sind bei der Validierung zur Modellierzeit jeweils mehrere assignMΘnK-Constraints. Das Ziel der Validierung ist es, Widersprüche in dieser Menge von assignMΘnK-Constraints zu finden.

Die Problematik bei der Validierung einer Menge von assignMΘnK-Constraints liegt darin, dass die Elemente der Mengen M und K der assignMΘnK-Constraints in vielen unterschiedlichen assignMΘnK-Constraints vorkommen können. Auf diese Weise können sich komplexe Abhängigkeitsbeziehungen zwischen den assignMΘnK-Constraints und deren

Elementen ergeben. Das Problem ob eine Menge von assignMΘnK-Constraints widerspruchsfrei ist, ist NP-vollständig. Mit anderen Worten ist kein deterministischer Algorithmus bekannt, der mit polynomialem Zeitaufwand entscheidet, ob eine Menge von assignMΘnK-Constraints widerspruchsfrei ist.

Gegeben ist eine Menge von assignMΘnK-Constraints. Von dieser Menge soll festgestellt werden, ob sie widerspruchsfrei ist. Der Nachweis der NP-Vollständigkeit des assignMΘnK-Problems (kurz: assignMΘnK) erfolgt in zwei Schritten.

1. assignMΘnK liegt in NP

Ein Entscheidungsproblem („Ist eine Menge von assignMΘnK-Constraints widerspruchsfrei?“) liegt in der Komplexitätsklasse NP, wenn eine geratene Lösung von einer deterministischen Turingmaschine mit polynomialem Zeitaufwand auf ihre Korrektheit hin überprüft werden kann. Der Nichtdeterminismus findet sich dabei im Raten der Lösung wieder. [26]

Dass dies bei assignMΘnK der Fall ist, ist leicht ersichtlich: Gegeben ist eine Lösung in Form von Zuordnungen, und damit Berechtigungen, zwischen den Benutzern, Rollen und Aktivitäten. Dann ist jeder der assignMΘnK-Constraints darauf hin zu überprüfen, ob er im Widerspruch zu den gegebenen Zuordnungen steht. Diese Überprüfung nimmt für jeden assignMΘnK-Constraint polynomiellen Zeitaufwand in Anspruch, so dass auch der Zeitaufwand zur Prüfung aller assignMΘnK-Constraints polynomiell ist.

2. assignMΘnK ist NP-hart

Um zu zeigen, dass assignMΘnK NP-hart ist, muss nachgewiesen werden, dass sich ein bereits bekanntes NP-hartes Problem mit polynomiellem Zeitaufwand auf assignMΘnK reduzieren lässt. Ein bekanntes NP-vollständiges, und damit auch NP-hartes, Problem ist Exact-3-SAT (X3SAT) [21]. Gegeben ist dabei eine boolesche Formel B in konjunktiver Normalform mit exakt drei Literalen pro Klausel. Das Entscheidungsproblem lautet „Ist B erfüllbar?“.

Zur polynomalien Reduzierung von X3SAT auf assignMΘnK muss also ein Verfahren f mit polynomalier Komplexität angegeben werden, so dass gilt $x \in$ X3SAT $\Leftrightarrow f(x) \in$ assignMΘnK. Gelingt es also, X3SAT auf assignMΘnK polynomial zu reduzieren, so ist damit der Nachweis erbracht, dass auch assignMΘnK NP-hart ist.

Gemäß X3SAT sind also gegeben

- boolesche Variablen $x_1, \ldots, x_l$ mit den zugehörigen Negationen $\overline{x_1}, \ldots, \overline{x_l}$ und
- die boolesche Formel $B = (z_{11} \vee z_{12} \vee z_{13}) \wedge \ldots \wedge (z_{m1} \vee z_{m2} \vee z_{m3})$, wobei
- $z_{ij} \in \{x_1, \ldots, x_l\} \cup \{\overline{x_1}, \ldots, \overline{x_l}\}$.

X3SAT wird dann wie folgt auf assignMΘnK reduziert:

Für jede boolesche Variable x_i ($\forall\ i \in \{1, \ldots, l\}$) drückt ein assignMΘnK-Constraint aus, dass diese Variable nur einen der Werte ‚Wahr' (W) oder ‚Falsch' (F) annehmen kann:

- $assignMonK(\{W, F\}, =, 1, \{x_i, \overline{x_i}\}), \quad \forall\, i \in \{1, \ldots, l\}$

Für jede Klausel ($z_{k1} \vee z_{k2} \vee z_{k3}$) wird ein assignMΘnK-Constraint wie folgt gebildet:

- $assignMonK(\{F\}, \leq, 2, \{z_{k1}, z_{k2}, z_{k3}\}), \quad \forall\, k \in \{1, \ldots, m\}$

Insgesamt entstehen so also $l + m$ viele assignMΘnK-Constraints, wobei l die Anzahl der Variablen und m die Anzahl der Klauseln des X3SAT-Problems ist.

Ist nun eine erfüllende Belegung für B gegeben, so sind zunächst alle assignMΘnK-Constraints

$$assignMonK(\{W, F\}, =, 1, \{x_i, \overline{x_i}\})$$

erfüllt, da innerhalb boolescher Ausdrücke jede Variable x_i immer genau einen der Werte W oder F annehmen muss. Erfüllt eine Belegung B, so nimmt außerdem jede der m Klauseln den Wert W an. Dies ist jedoch nur dann der Fall, wenn mindestens eines der Literale z_{k1}, z_{k2}, z_{k3} ($\forall\, k \in \{1, \ldots, m\}$) den Wert W annimmt. Anders ausgedrückt dürfen höchstens zwei der Literale einer Klausel den Wert F annehmen. Somit sind auch alle assignMΘnK-Constraints

$$assignMonK(\{F\}, \leq, 2, \{z_{k1}, z_{k2}, z_{k3}\})$$

erfüllt, da dort F höchstens zwei Elementen aus $\{z_{k1}, z_{k2}, z_{k3}\}$ zugeordnet werden darf.

Ist dagegen eine nicht erfüllende Belegung für B gegeben, so sind zunächst ebenfalls alle assignMΘnK-Constraints

$$assignMonK(\{W, F\}, =, 1, \{x_i, \overline{x_i}\})$$

erfüllt. Wird B nicht erfüllt, so muss jedoch mindestens eine Klausel ($z_{k1} \vee z_{k2} \vee z_{k3}$) existieren, die den Wert F annimmt. In dieser Klausel müssen hierfür alle Literale z_{k1}, z_{k2}, z_{k3} den Wert F annehmen. Da jedoch

$$assignMonK(\{F\}, \leq, 2, \{z_{k1}, z_{k2}, z_{k3}\})$$

die Zuordnung von F an höchstens zwei Elementen aus $\{z_{k1}, z_{k2}, z_{k3}\}$ erlaubt, steht die gegebene Belegung auch im Widerspruch zu assignMΘnK.

Das Problem ob eine Menge von assignMΘnK-Constraints widerspruchsfrei ist, ist also NP-vollständig. Es ist daher kein Algorithmus bekannt, der mit polynomiellem Zeitaufwand die Erfüllbarkeit einer Menge von assignMΘnK-Constraints überprüfen kann. Dennoch ist es möglich, die assignMΘnK-Constraints einfacheren Prüfungen zu unterziehen. Durch diese kann bereits ein Teil der Widersprüche gefunden werden.

7.4.2 Paarweise Validierung von assignMΘnK-Constraints

Eine solche einfache Prüfung ist die paarweise Validierung von assignMΘnK-Constraints. Das Ziel dieser paarweisen Validierung ist es, Widersprüche zwischen jeweils zwei betrachteten assignMΘnK-Constraints zu finden. Auf diese Weise kann sichergestellt werden, dass die gleichzeitige Erfüllung beider assignMΘnK-Constraints möglich ist.

Bei Betrachtung zweier assignMΘnK-Constraints

- $assignMonK_1(M_1, \Theta_1, n_1, K_1)$ und
- $assignMonK_2(M_2, \Theta_2, n_2, K_2)$

kann nur dann ein Widerspruch auftreten, wenn entweder

i) $typ_{M_1} = typ_{M_2} \wedge typ_{K_1} = typ_{K_2} \wedge M_1 \cap M_2 \neq \emptyset \wedge K_1 \cap K_2 \neq \emptyset$ oder
ii) $typ_{M_1} = typ_{K_2} \wedge typ_{M_2} = typ_{K_1} \wedge M_1 \cap K_2 \neq \emptyset \wedge M_2 \cap K_1 \neq \emptyset$

gilt. Denn ein Widerspruch kann nur dann auftreten, wenn die assignMΘnK-Constraints eine Aussage über dieselben Elemente treffen. Sind zwei assignMΘnK-Constraints gegeben die auf keine der beiden Bedingungen zutreffen, so kann kein Widerspruch bestehen. Im Folgenden werden daher jeweils zwei assignMΘnK-Constraints in den Fällen i) und ii) betrachtet.

i) $\boldsymbol{typ_{M_1} = typ_{M_2} \wedge typ_{K_1} = typ_{K_2} \wedge M_1 \cap M_2 \neq \emptyset \wedge K_1 \cap K_2 \neq \emptyset}$

Zunächst werden jeweils zwei assignMΘnK-Constraints mit $typ_{M_1} = typ_{M_2}$ und $typ_{K_1} = typ_{K_2}$ betrachtet. Das bedeutet, dass in den Mengen M und K dieselbe Art von Elementen (Benutzer, Rollen oder Aktivitäten) enthalten sind. Zusätzliche Einschränkungen für die

beiden betrachteten assignMΘnK-Constraints sind $M_1 \cap M_2 \neq \emptyset$ und $K_1 \cap K_2 \neq \emptyset$. Zunächst werden zwei assignMΘnK-Constraints betrachtet mit $M_1 = M_2 = M$, also

- $assignMonK_1(M, \Theta_1, n_1, K_1)$ und
- $assignMonK_2(M, \Theta_2, n_2, K_2)$.

Der Fall $M_1 \neq M_2$ wird später auf diesen Fall zurückgeführt.

Tabelle 7.1 listet auf, bei welchen Werten für Θ_1, Θ_2, n_1 und n_2 Widersprüche zwischen $assignMonK_1$ und $assignMonK_2$ auftreten können. Betrachtet wird dabei außerdem das Verhältnis zwischen den Mengen K_1 und K_2. Die Bedeutung der einzelnen Zeichen ist wie folgt:

- $\lightning$: Widerspruch tritt unbedingt auf
- $\natural$: Widerspruch tritt auf, falls $|K_1 \backslash K_2| < (n_1 - n_2) \cdot |M|$
- $\flat$: Widerspruch tritt auf, falls $|K_2 \backslash K_1| < (n_2 - n_1) \cdot |M|$

Fall	$\boldsymbol{\Theta_1}$	$\boldsymbol{\Theta_2}$	$\boldsymbol{n_1, n_2}$	$\boldsymbol{K_1 = K_2}$	$\boldsymbol{K_1 \subset K_2}$	$\boldsymbol{K_1 \supset K_2}$	$\boldsymbol{K_1 \not\subset K_2 \wedge K_1 \not\supset K_2}$
a)	$=$	$=$	$n_1 > n_2$	$\lightning$	$\lightning$	$\natural$	$\natural$
b)	$=$	$\leq$	$n_1 > n_2$	$\lightning$	$\lightning$	$\natural$	$\natural$
c)	$=$	$\geq$	$n_1 < n_2$	$\lightning$	$\flat$	$\lightning$	$\flat$
d)	$\leq$	$\geq$	$n_1 < n_2$	$\lightning$	$\flat$	$\lightning$	$\flat$

Tabelle 7.1: Widersprüche bei Betrachtung zweier assignMΘnK-Constraints.

An dieser Stelle werden exemplarisch drei der insgesamt 16 Fälle aus Tabelle 7.1 ausführlich betrachtet. Die ausführlichen Betrachtungen aller anderen Fälle der Tabelle verlaufen analog und sind dem Anhang A zu entnehmen. Dort finden sich außerdem ausführliche Betrachtungen für alle anderen Kombinationsmöglichkeiten von Θ_1, Θ_2, n_1 und n_2, sowie K_1 und K_2.

Fall a) mit $\boldsymbol{K_1 \supset K_2}$ und $\boldsymbol{n_1 > n_2}$

- $assignMonK_1(M, =, n_1, K_1)$
- $assignMonK_2(M, =, n_2, K_2)$

Gemäß $assignMonK_2$ müssen jedem $m \in M$ exakt n_2 viele Elemente aus K_2 zugeordnet werden. Aus K_2 werden folglich genau $|M| \cdot n_2$ viele Elemente an die Elemente aus M zugeordnet. Wegen $K_1 \supset K_2$ sind alle diese $|M| \cdot n_2$ Elemente auch aus K_1. Damit $assignMonK_1$ erfüllt ist, müssen jedem $m \in M$ genau n_1 viele Elemente aus K_1 zugeordnet werden. Da $n_1 > n_2$ ist, müssen also jedem $m \in M$ weitere $n_1 - n_2$ Elemente aus K_1

zugeordnet werden. Diese insgesamt $|M| \cdot (n_1 - n_2)$ Elemente müssen jedoch aus $K_1 \backslash K_2$ sein, da andernfalls $assignMonK_2$ verletzt würde. Somit folgt, dass beide assignMΘnK-Constraints nicht gleichzeitig erfüllbar sind, falls $|K_1 \backslash K_2| < (n_1 - n_2) \cdot |M|$ gilt. Abb. 7.4 zeigt zwei derartige unerfüllbare assignMΘnK-Constraints. Dabei kann $assignMonK_1$ nicht erfüllt werden, da $K_1 \backslash K_2$ nicht ausreichend viele Benutzer enthält.

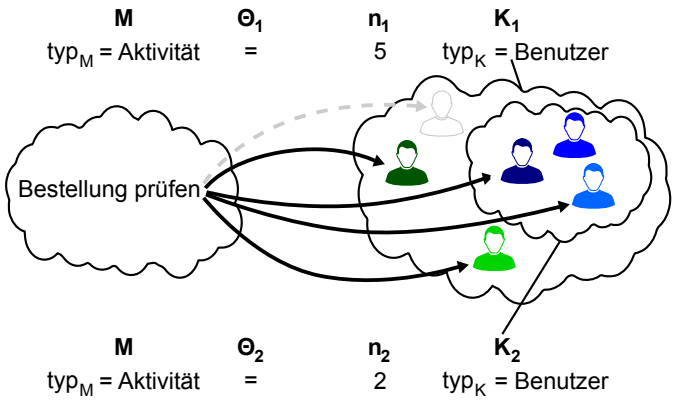


Abb. 7.4: Zwei nicht gleichzeitig erfüllbare assignMΘnK-Constraints im Fall a).

Fall b) mit $K_1 = K_2 = K$ und $n_1 > n_2$

- $assignMonK_1(M, =, n_1, K)$
- $assignMonK_2(M, \leq, n_2, K)$

Gemäß $assignMonK_1$ müssen jedem $m \in M$ genau n_1 viele Elemente aus K zugeordnet werden. Wegen $assignMonK_2$ dürfen außerdem jedem $m \in M$ höchstens n_2 viele Elemente aus K zugeordnet werden. Jedem $m \in M$ müssen also gleichzeitig genau n_1 viele und höchstens n_2 viel Elemente aus derselben Menge K zugeordnet werden. Gleichzeitig gilt jedoch $n_1 > n_2$, so dass $assignMonK_1$ und $assignMonK_2$ stets im Widerspruch zueinander stehen und niemals gleichzeitig erfüllt werden können. In Abb. 7.5 sind zwei derartige assignMΘnK-Constraints abgebildet. Die Erfüllung von $assignMonK_2$ ist dabei zwar möglich, nicht aber die gleichzeitige Erfüllung von $assignMonK_1$.

Fall c) mit $K_1 \not\subset K_2 \wedge K_1 \not\supset K_2$ und $n_1 < n_2$

- $assignMonK_1(M, =, n_1, K_1)$
- $assignMonK_2(M, \geq, n_2, K_2)$

Gemäß $assignMonK_1$ müssen jedem $m \in M$ exakt n_1 viele Elemente aus K_1 zugeordnet werden. Da $n_1 < n_2$ ist, müssen jedem $m \in M$ außerdem zur Erfüllung von $assignMonK_2$

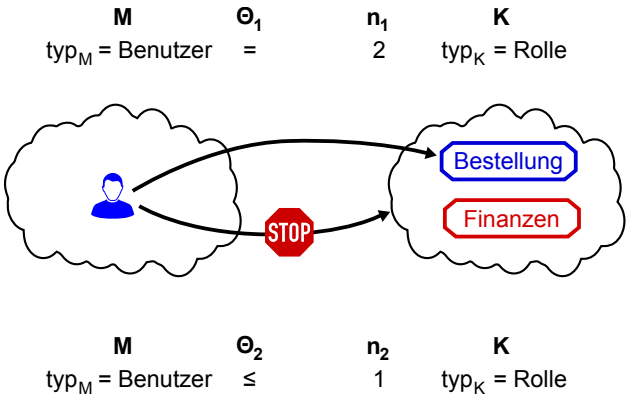


Abb. 7.5: Zwei nicht gleichzeitig erfüllbare assignMΘnK-Constraints im Fall b).

mindestens $n_2 - n_1$ viele Elemente aus $K_2 \backslash K_1$ zugeordnet werden. Insgesamt werden aus $K_2 \backslash K_1$ also immer $|M| \cdot (n_2 - n_1)$ viele Elemente zugeordnet. Somit folgt, dass beide assignMΘnK-Constraints nicht gleichzeitig erfüllbar sind, falls $|K_2 \backslash K_1| < (n_2 - n_1) \cdot |M|$. Abb. 7.6 zeigt zwei derartige assignMΘnK-Constraints. $assignMonK_2$ kann nicht erfüllt werden, da die Menge $K_2 \backslash K_1$ zu wenig Benutzer enthält.

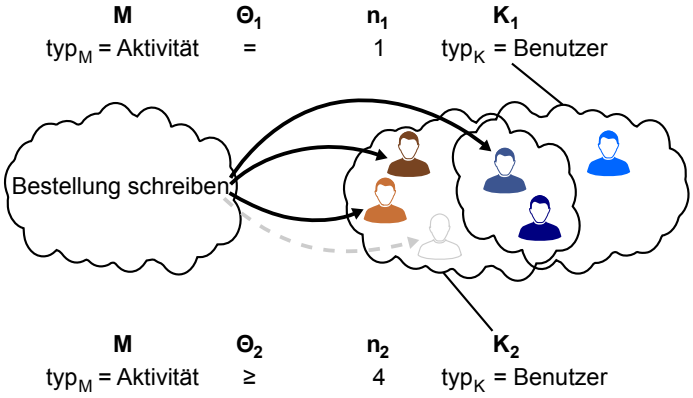


Abb. 7.6: Zwei nicht gleichzeitig erfüllbare assignMΘnK-Constraints im Fall c).

Bisher wurden lediglich assignMΘnK-Constraints $assignMonK_1$ und $assignMonK_2$ mit $M_1 = M_2 = M$ betrachtet. Das Auffinden von Widersprüchen bei Betrachtung zweier assignMΘnK-Constraints mit $M_1 \neq M_2$ lässt sich wie folgt auf die vorherigen Betrachtungen zurückführen. Gegeben sind zwei assignMΘnK-Constraints

- $C_1 : assignMonk(M_1, \Theta_1, n_1, K_1)$ und
- $C_2 : assignMonk(M_2, \Theta_2, n_2, K_2)$

mit $M_1 \neq M_2$ und $M_1 \cap M_2 \neq \emptyset$. Dann gilt:

a) Jedes $m \in M_1 \backslash M_2$ muss $\Theta_1 n_1$ vielen Elementen aus K_1 zugeordnet werden.

b) Jedes $m \in M_2 \backslash M_1$ muss $\Theta_2 n_2$ vielen Elementen aus K_2 zugeordnet werden.

c) Jedes $m \in M_1 \cap M_2$ muss $\Theta_1 n_1$ vielen Elemente aus K_1 und außerdem $\Theta_2 n_2$ vielen Elementen aus K_2 zugeordnet werden.

C_1 und C_2 lassen sich mit diesen Erkenntnissen durch die folgenden assignMΘnK-Constraints C_3, C_4, C_5 und C_6 ausdrücken:

- $C_3 : assignMonk(M_1 \backslash M_2, \Theta_1, n_1, K_1)$ realisiert a),
- $C_4 : assignMonk(M_2 \backslash M_1, \Theta_2, n_2, K_2)$ realisiert b),
- $C_5 : assignMonk(M_1 \cap M_2, \Theta_1, n_1, K_1)$ und
  $C_6 : assignMonk(M_1 \cap M_2, \Theta_2, n_2, K_2)$ realisieren c).

Die assignMΘnK-Constraints C_3 und C_4 widersprechen sich weder gegenseitig, noch widersprechen sie C_5 oder C_6. Dies hat seine Begründung darin, dass die jeweiligen Mengen M disjunkt sind. Da die Mengen M der beiden assignMΘnK-Constraints C_5 und C_6 gleich sind, können diese gemäß den vorherigen Ausführungen daraufhin untersucht werden, ob deren gleichzeitige Einhaltung möglich ist. Nur wenn zwischen C_5 und C_6 ein Widerspruch gefunden wird, besteht auch derselbe Widerspruch zwischen den gegebenen assignMΘnK-Constraints C_1 und C_2.

ii) $\boldsymbol{typ_{M_1} = typ_{K_2} \wedge typ_{M_2} = typ_{K_1} \wedge M_1 \cap K_2 \neq \emptyset \wedge M_2 \cap K_1 \neq \emptyset}$

Bei den vorherigen Betrachtungen zweier assignMΘnK-Constraints waren jeweils die Typen der Mengen M und K identisch. Weiter können bei der Betrachtung zweier assignMΘnK-Constraints

- $C_1 : assignMonK_1(M_1, \Theta_1, n_1, K_1)$ und
- $C_2 : assignMonK_2(M_2, \Theta_2, n_2, K_2)$

auch dann Widersprüche auftreten, wenn gilt $M_1 \cap K_2 \neq \emptyset$ und $M_2 \cap K_1 \neq \emptyset$. Ist eine dieser beiden Schnittmengen leer, so treffen die zwei assignMΘnK-Constraints keine Aussage über dieselben Elemente, so dass diese sich auch nicht widersprechen können.

Um Widersprüche zwischen $assignMonK_1$ und $assignMonK_2$ zu finden, wird dann wie folgt vorgegangen: Für jedes Element $k \in K_1$ wird ein assignMΘnK-Constraint

- $C_k : assignMonK_k(\{k\}, \leq, 1, M_1)$

formuliert. Dieser besagt ausdrücklich, dass jedes Element $k \in K_1$ höchstens einem Element $m \in M_1$ zugeordnet werden darf. Jeder assignMΘnK-Constraint C_k ist dann auf Widersprüche mit C_2 hin zu überprüfen. Dies geschieht gemäß den Ausfühungen in i) (Seite 124).

Dabei sei darauf hingewiesen, dass Widersprüche zwischen C_2 und C_k nur dann auftreten können, falls

- $\{k\} = M_2$ oder
- $\{k\} \subset M_2$

gilt. In beiden Fällen führen dann die Betrachtungen in i) dazu, dass für jedes $k \in K_1$ die beiden assignMΘnK-Constraints

- $assignMonK(\{k\}, \leq, 1, M_1)$ und
- $assignMonK(\{k\}, \Theta_2, n_2, K_2)$

auf Widersprüche hin zu untersuchen sind. Da die Betrachtungen in i) jedoch unabhängig von den konkreten Elementen in M sind, ist die Überprüfung zweier assignMΘnK-Constraints

- $assignMonK(M, \leq, 1, M_1)$ und
- $assignMonK(M, \Theta_2, n_2, K_2)$

für eine Menge M mit $|M| = 1$ ausreichend.

Analog wird mit allen Elementen der Menge K_2 verfahren. Dies hat gemäß den obigen Ausführungen zur Folge, dass zwei assignMΘnK-Constraints

- $assignMonK(M, \leq, 1, M_2)$ und
- $assignMonK(M, \Theta_1, n_1, K_1)$

für eine Menge M mit $|M| = 1$ auf Widersprüche gemäß i) überprüft werden müssen.

Zusammenfassung

Die paarweise Validierung von assignMΘnK-Constraints ermöglicht das Finden von Widersprüchen zwischen jeweils zwei assignMΘnK-Constraints mit geringem Zeitaufwand. Auf diese Weise können, trotz der NP-Vollständigkeit des assignMΘnK-Problems, viele Konflikte in einer Menge von assignMΘnK-Constraints effizient gefunden werden. Ist eine Menge von assignMΘnK-Constraints gegeben und werden diese gemäß den vorhergehenden Ausführungen paarweise betrachtet, so ist der hierzu nötige Zeitaufwand quadratisch in der Anzahl der assignMΘnK-Constraints.

Durch die paarweise Validierung von assignMΘnK-Constraints können jedoch nicht alle Widersprüche in einer Menge von assignMΘnK-Constraints gefunden werden. Um die tatsächliche Widerspruchsfreiheit einer Menge von assignMΘnK-Constraints zu überprüfen, sind weitere Konzepte, wie beispielsweise Backtracking, notwendig. Hierbei würden allen Berechtigungskombinationen ausprobiert, mit dem Ziel alle assignMΘnK-Constraints zu erfüllen. Ist es nicht möglich, eine solche Lösung zu finden, so sind die gegebenen assignMΘnK-Constraints nicht alle gleichzeitig erfüllbar. Weiter ist es denkbar, das Backtracking-Konzept nur auf Submengen der assignMΘnK-Constraints anzuwenden, um so einen Teil der assignMΘnK-Constraints auf Widersprüche hin zu überprüfen.

Die paarweise Validierung von assignMΘnK-Constraints wurde im Rahmen dieser Arbeit implementiert. Details zur Implementierung sind Kapitel 8 zu entnehmen.

Während assignMΘnK-Constraint bereits zur Laufzeit validiert werden können und müssen, ist die Validierung der activateMΘnK-Constraints und entailmentMΘnK-Constraints erst zur Laufzeit des Prozesses möglich. Diese werden im Folgenden betrachtet.

7.5 Validierung zur Laufzeit

In Kapitel 7.3 wurde gezeigt, dass zwischen activateMΘnK-Constraints und entailmentMΘnK-Constraints keine Widersprüche bestehen können, die die Einhaltung aller dieser MΘnK-Constraints unmöglich machen. Eine große Herausforderung beim Einsatz von Authorization Constraints in WfMS besteht jedoch darin sicher zu stellen, dass die Ausführung aller Aktivitäten des Prozesses möglich ist. Die vergebenen RBAC-Berechtigungen und MΘnK-Constraints dürfen also niemals die Ausführung einer oder mehrerer Aktivitäten un-

möglich machen. Es muss daher gewährleistet werden, dass alle noch auszuführenden Aktivitäten durch mindestens einen Benutzer ausgeführt werden können.

Wie Abb. 7.7 zeigt, sind zur Laufzeit des Prozesses nur noch activateMΘnK-Constraints und entailmentMΘnK-Constraints zu validieren. Durch die fortwährende Anpassung der activateMΘnK-Constraints (Kapitel 6.2) und das Hinzukommen neuer activateMΘnK-Constraints durch entailmentMΘnK-Constraints (Kapitel 6.3), ist eine einmalige Validierung der MΘnK-Constraints nicht ausreichend. Stattdessen müssen diese bei jeder Änderung neu validiert werden.

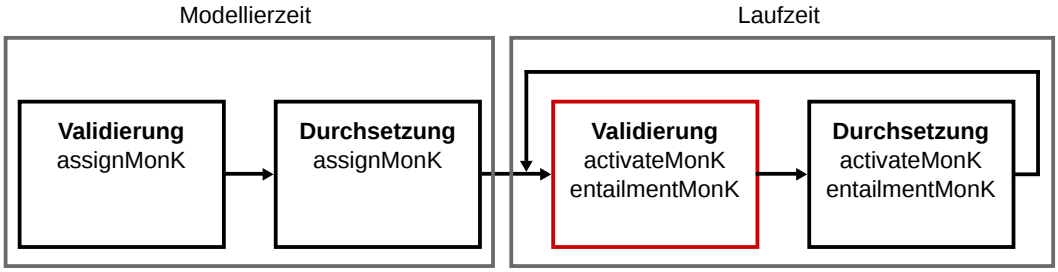


Abb. 7.7: Validierung der activateMΘnK-Constraints und entailmentMΘnK-Constraints zur Laufzeit des Prozesses.

Da die assignMΘnK-Constraints bereits zur Modellierzeit des Prozesses durchgesetzt worden sind, spiegeln diese sich nun in den RBAC-Berechtigungen wider und müssen daher bei der Validierung zur Laufzeit nicht weiter beachtet werden. entailmentMΘnK-Constraints nehmen wiederum auf Grund ihrer „wenn-dann“-Semantik eine Sonderstellung ein. Im Folgenden wird daher zunächst die Validierung der activateMΘnK-Constraints betrachtet. Anschließend folgt die zusätzliche Validierung der entailmentMΘnK-Constraints.

7.5.1 Validierung der activateMΘnK-Constraints

Die folgenden Betrachtungen finden stets zu einem beliebigen aber festen Zeitpunkt während der Laufzeit des Prozesses statt. Zu einem solchen Zeitpunkt existieren immer RBAC-Berechtigungen und eine Menge von activateMΘnK-Constraints, die die Verwendung der RBAC-Berechtigungen einschränken. Eine zusätzliche Voraussetzung ist, dass alle activateMΘnK-Constraints zu Beginn der Betrachtung eingehalten werden. Andernfalls wird mindestens ein activateMΘnK-Constraint verletzt, was aber auf keinen Fall geschehen darf.

Zu dem betrachteten Zeitpunkt gibt es außerdem eine Menge von Aktivitäten $tasks$, die im weiteren Verlauf des Prozesses noch auszuführen sind.

Ziel ist es nun herauszufinden, ob alle noch auszuführenden Aktivitäten $tasks$ mit den gegebenen RBAC-Berechtigungen und activateMΘnK-Constraints zum Zeitpunkt der Betrachtung von mindestens einem Benutzer ausgeführt werden können. Wird dabei mindestens eine Aktivität gefunden deren Ausführung nicht möglich ist, so kann der Prozess mit den gegebenen RBAC-Berechtigungen und activateMΘnK-Constraints nicht erfolgreich zu Ende gebracht werden. Es wird Teil der Betrachtungen in Kapitel 7.5.2 sein, derartige Situationen (Deadlocks) zu verhindern oder frühzeitig zu erkennen. Im vorliegenden Kapitel wird zunächst eine Lösung entwickelt um herauszufinden, ob alle noch auszuführenden Aktivitäten $tasks$ durch mindestens einen Benutzer ausgeführt werden können.

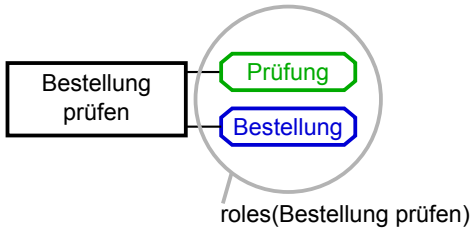
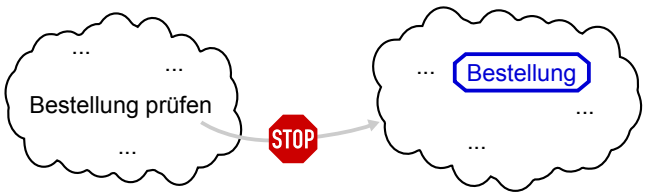


Abb. 7.8: Rollen der Aktivität „Bestellung prüfen“ im Fallbeispiel.

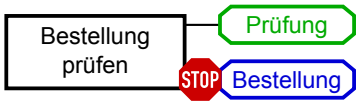
Für jede Aktivität $t \in tasks$ werden zunächst alle Rollen betrachtet, die durch die RBAC-Berechtigungen zur Ausführung von t berechtigen. Dies sind gerade alle Rollen $r \in roles(t)$. Abb. 7.8 zeigt beispielhaft die Rollen der Aktivität „Bestellung prüfen“ im Fallbeispiel. Für jede Rolle $r \in roles(t)$ werden dann sämtliche activateMΘnK-Constraints betrachtet, die eine Aussage über das Verhältnis zwischen der Aktivität t und der Rolle r machen. Dies sind alle activateMΘnK-Constraints für die gilt

- $r \in M \wedge t \in K$ oder
- $t \in M \wedge r \in K$.



M	Θ	n	K
typ_M = Aktivität	≤	0	typ_K = Rolle

(a) Verbot der Ausführung von „Bestellung prüfen“ in der Rolle „Bestellung“.



(b) Zur Ausführung von „Bestellung prüfen“ berechtigte Rollen mit dem activateMΘnK-Constraints aus Abb. 7.9a.

Abb. 7.9: activateMΘnK-Constraint und dessen Auswirkung auf die Rollen der Aktivität „Bestellung prüfen“.

Existiert ein darauf zutreffender activateMΘnK-Constraint mit $n = 0$, so wird durch diesen die Ausführung der Aktivität t durch die Rolle r verboten. Die Rolle r ist somit nicht weiter zu betrachten. Trifft dagegen kein activateMΘnK-Constraint auf eine dieser Bedingungen zu, oder gilt für alle der zutreffenden activateMΘnK-Constraints $n > 0$, so ist die Ausführung der Aktivität t durch einen Benutzer in der Rolle r möglich. In Abb. 7.9a ist ein activateMΘnK-Constraint abgebildet, der die Ausführung von „Bestellung prüfen“ durch die Rolle „Bestellung“ verbietet.

Nach den obigen Ausführungen können somit alle Rollen $roles$ ermittelt werden, die, auch unter Berücksichtigung der activateMΘnK-Constraints, zur Ausführung der Aktivität t berechtigen. Abb. 7.9b zeigt, dass mit dem activateMΘnK-Constraints aus Abb. 7.9a die Aktivität „Bestellung prüfen“ nur noch in der Rolle „Prüfung“ ausgeführt werden kann.

Die Aktivität t kann jedoch nur dann in einer Rolle $r \in roles$ ausgeführt werden, wenn mindestens ein Benutzer zur Aktivierung der Rolle r berechtigt ist. Hierbei ist es auch möglich, dass Benutzer existieren, die diese Rolle bereits aktiviert haben.

Für jede der für Aktivität t berechtigten Rollen $r \in roles$ werden dann alle Benutzer betrachtet, die mittels RBAC dieser Rolle zugeordnet sind. Dies sind gerade alle Benutzer $u \in users(r)$. Abb. 7.10 zeigt alle Benutzer der zur Ausführung von „Bestellung prüfen“ berechtigenden Rolle „Prüfung“. Ist die Rolle r für einen Benutzer u bereits aktiviert, so

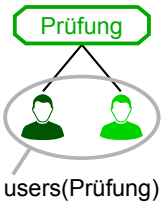


Abb. 7.10: Benutzer der Rolle „Prüfung“.

müssen die activateMΘnK-Constraints nicht ausgewertet werden. Andernfalls werden für jeden Benutzer u alle activateMΘnK-Constraints betrachtet, die die Aktivierung der Rolle r durch u einschränken. Dies sind alle activateMΘnK-Constraints für die gilt

- $u \in M \wedge r \in K$ oder
- $r \in M \wedge u \in K$.

Existiert ein activateMΘnK-Constraints der auf eine dieser Bedingungen zutrifft und gilt für diesen zusätzlich $n = 0$, so ist der Benutzer u nicht zur Aktivierung der Rolle r berechtigt. Trifft dagegen kein activateMΘnK-Constraint auf eine der Bedingungen zu, oder gilt für alle zutreffenden activateMΘnK-Constraints $n > 0$, so ist die Aktivierung der Rolle r durch den Benutzer u möglich. Abb. 7.11 zeigt für das Fallbeispiel einen solchen zutreffenden activateMΘnK-Constraint mit $n = 0$ für einen der Benutzer der Rolle „Prüfung“.

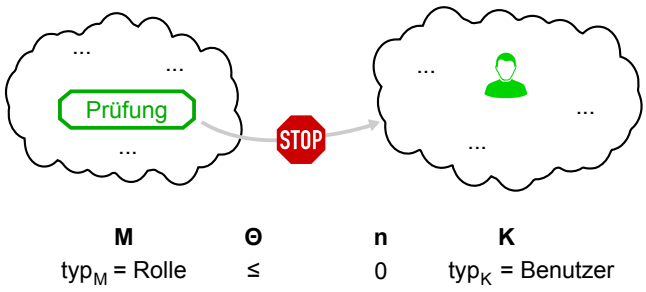


Abb. 7.11: Verbot der Aktivierung der Rolle „Prüfung“ für bestimmte Benutzer.

Auf die beschriebene Weise ist es möglich Benutzer zu finden, die zur Aktivierung einer Rolle berechtigt sind, welche wiederum zur Ausführung der Aktivität t berechtigt. Abb. 7.12a und Abb. 7.12b fassen die resultierenden Berechtigungen nach Berücksichtigung der beiden obigen activateMΘnK-Constraints im Fallbeispiel zusammen.

(a) Zur Ausführung von „Bestellung prüfen“ berechtigte Rollen.

(b) Zur Aktivierung von „Prüfung“ berechtigte Benutzer.

Abb. 7.12: Berechtigungen nach Beachtung beider activateMΘnK-Constraints.

Dennoch können auch activateMΘnK-Constraints existieren, die die Ausführung der Aktivität t durch einen Benutzer u, unabhängig von der verwendeten Rolle, verbieten. Dies sind alle activateMΘnK-Constraints für die gilt

- $u \in M \wedge t \in K$ oder
- $t \in M \wedge u \in K$.

Existiert ein activateMΘnK-Constraint mit $n = 0$ der auf eine dieser beiden Bedingungen zutrifft, so ist Benutzer u nicht zur Ausführung der Aktivität t berechtigt. Gibt es dagegen keinen activateMΘnK-Constraint der auf eine der Bedingungen zutrifft, oder gilt für alle zutreffenden activateMΘnK-Constraints $n > 0$, so kann t durch u ausgeführt werden. Abb. 7.13 zeigt einen im Fallbeispiel zutreffenden activateMΘnK-Constraint mit $n = 1$. Die Aktivität „Bestellung prüfen“ kann somit durch den Benutzer in der Rolle „Prüfung“ ausgeführt werden kann.

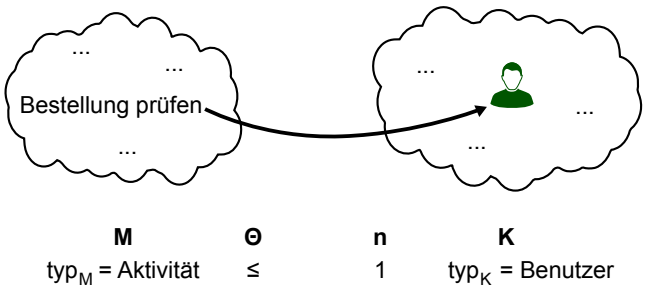


Abb. 7.13: Zutreffender activateMΘnK-Constraint mit $n = 1$.

Wird der beschriebene Vorgang für alle noch auszuführenden Aktivitäten ausgeführt, so kann herausgefunden werden, ob alle Aktivitäten des Prozesses mit den bestehenden RBAC-Berechtigungen und activateMΘnK-Constraints ausführbar sind.

Können alle noch auszuführenden Aktivitäten zu dem betrachteten Zeitpunkt von mindestens einem Benutzer ausgeführt werden, so bedeutet dies jedoch nicht, dass im weiteren Verlauf auch tatsächlich alle Aktivitäten ausgeführt werden können. Dies hat seine Begründung in den entailmentMΘnK-Constraints und den sich ändernden activateMΘnK-Constraints. Wird ein entailmentMΘnK-Constraint zur Laufzeit des Prozesses gemäß den Ausführungen in Kapitel 6.3 durchgesetzt, so entstehen neue activateMΘnK-Constraints, die es im weiteren Verlauf des Prozesses einzuhalten gilt. Auch bei der Durchsetzung der activateMΘnK-Constraints entstehen nach Kapitel 6.2 neue activateMΘnK-Constraints, die im weiteren Verlauf einzuhalten sind. Schon bei der nächsten Rollenaktivierung oder Aktivitätenausführung können daher solche activateMΘnK-Constraints entstehen, die die Ausführung zukünftiger Aktivitäten unmöglich machen. Diese Problematik wird im folgenden Kapitel betrachtet.

7.5.2 Validierung der entailmentMΘnK-Constraints

Zur Validierung der entailmentMΘnK-Constraints betrachten wir den Prozess bei Vorliegen eines Aktivierungswunsches ($activate(user, role)$) oder eines Ausführungswunsches ($execute(user, role, task)$). Gemäß Kapitel 6.2 und Kapitel 6.3 werden dann die RBAC-Berechtigungen und activateMΘnK-Constraints dahingehend überprüft, ob die Aktivierung oder Ausführung tatsächlich stattfinden kann. Findet die Aktivierung oder Ausführung statt, so müssen alle activateMΘnK-Constraints $activateMonK_2$ der zutreffenden entailmentMΘnK-Constraints im weiteren Verlauf des Prozesses durchgesetzt werden. An dieser Stelle können jedoch activateMΘnK-Constraints entstehen, die die Ausführung späterer Aktivitäten unmöglich machen.

Um die Entstehung solcher activateMΘnK-Constraints zu verhindern, ist die vorherige Validierung der entailmentMΘnK-Constraints notwendig. Abb. 7.14 veranschaulicht, dass die Validierung der entailmentMΘnK-Constraints gewissermaßen zeitgleich mit der Durchsetzung der activateMΘnK-Constraints stattfindet. Inwiefern dabei die Durchsetzung und Validierung der activateMΘnK-Constraints und entailmentMΘnK-Constraints zusammenwirken, wird im Folgenden betrachtet. Die Validierung der entailmentMΘnK-Constraints wird dabei auf die Validierung einer Menge von activateMΘnK-Constraints gemäß Kapitel 7.5.1 zurückgeführt. Dabei beschränken wir uns bei den folgenden Betrachtungen auf das Vorliegen eines Ausführungswunsches (Abb. 7.14). Das Vorgehen bei einer gewünschten Rollenaktivierung erfolgt analog.

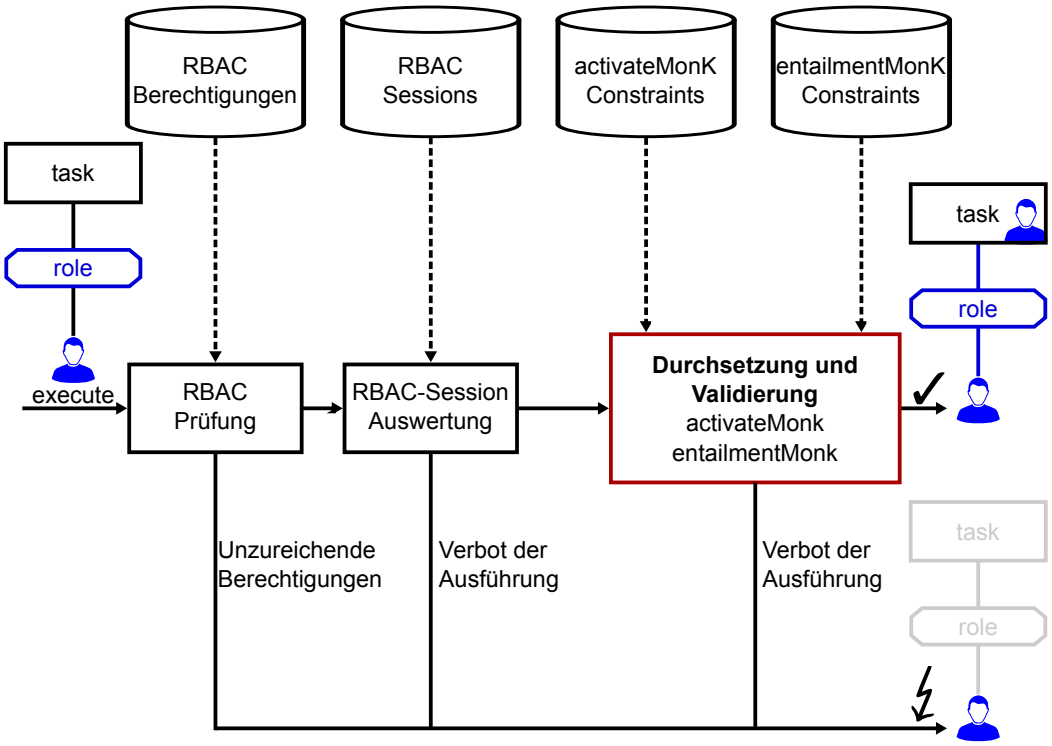


Abb. 7.14: Zeitpunkt der Validierung der entailmentMΘnK-Constraints bei einer gewünschten Aktivitätenausführung.

Soll eine Aktivitätenausführung stattfinden, so sind zunächst die RBAC-Berechtigungen und RBAC-Sessions zu überprüfen. Wird von diesen die Ausführung nicht verboten, sind anschließend die activateMΘnK-Constraints und entailmentMΘnK-Constraints zu betrachten. Die folgenden Ausführungen hierzu werden in Abb. 7.15 veranschaulicht. Dabei detailliert Abb. 7.15 die Durchsetzung und Validierung von activateMΘnK-Constraints und entailmentMΘnK-Constraints in Abb. 7.14.

Ähnlich zu Kapitel 6.2 werden zunächst alle zutreffenden activateMΘnK-Constraints durchgesetzt. Diese werden jeweils daraufhin geprüft, ob sie die gewünschte Ausführung verbieten. Ist das der Fall, so kann die gewünschte Ausführung auf Grund der bestehenden activateMΘnK-Constraints nicht stattfinden. Wird die Ausführung jedoch durch keinen der zutreffenden activateMΘnK-Constraints verboten, so entsteht, abweichend von Kapitel 6.2, eine neue Menge temporärer activateMΘnK-Constraints. In dieser Menge sind zunächst alle auf die Ausführung nicht-zutreffenden activateMΘnK-Constraints enthalten. Zusätzlich

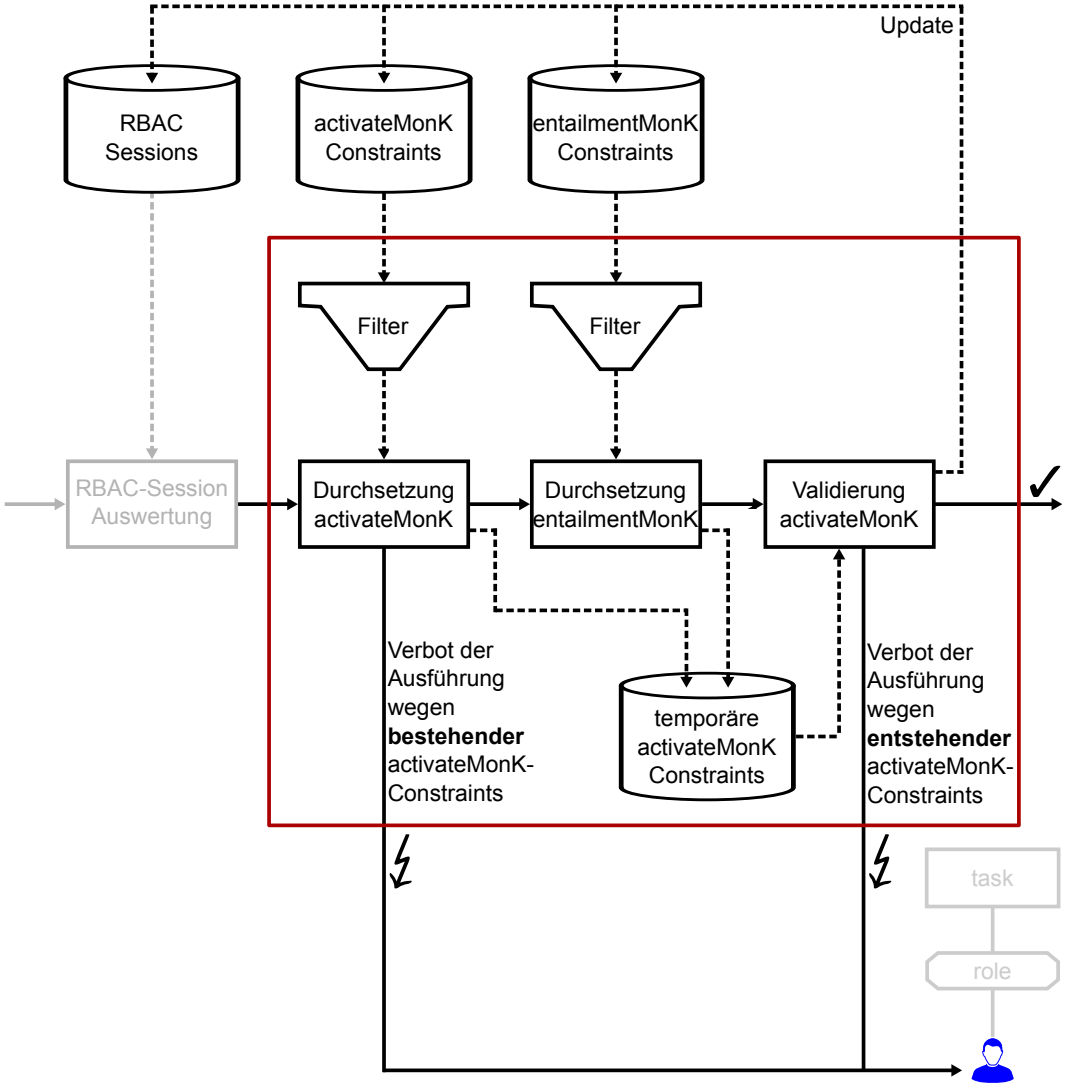


Abb. 7.15: Zusammenspiel von Validierung und Durchsetzung der activateMΘnK-Constraints und entailmentMΘnK-Constraints.

enthält diese Menge alle activateMΘnK-Constraints, die im Fall der erfolgreichen Ausführung aus den zutreffenden activateMΘnK-Constraints, gemäß den Ausführungen in Kapitel 6.2.4, hervorgehen würden.

Auch die Durchsetzung der entailmentMΘnK-Constraints geschieht leicht abweichend von Kapitel 6.3. Zunächst werden alle zutreffenden entailmentMΘnK-Constraints ausfindig gemacht. Die im Fall der erfolgreichen Ausführung entstehenden activateMΘnK-Constraints $activateMonK_2$ der zutreffenden entailmentMΘnK-Constraints werden dann zur Menge der temporären activateMΘnK-Constraints hinzugefügt. Auf diese Weise enthält die Menge der temporären activateMΘnK-Constraints genau diejenigen activateMΘnK-Constraints, die nach Zulassung des Ausführungswunsches durchzusetzen wären. Gewissermaßen kann mit dieser entstandenen Menge die Zulassung des Ausführungswunsches simuliert werden.

Die Menge der temporären activateMΘnK-Constraints wird dann gemäß den Ausführungen in Kapitel 7.5.1 validiert. Für alle noch auszuführenden Aktivitäten wird also auf Basis der temporären activateMΘnK-Constraints geprüft, ob diese nach der Zulassung des Ausführungswunsches noch ausführbar sind. Wird dabei festgestellt, dass mit diesen temporären activateMΘnK-Constraints nicht alle Aktivitäten ausgeführt werden können, so darf die gewünschte Ausführung nicht stattfinden. Ist dagegen die Ausführung aller noch auszuführenden Aktivitäten weiterhin möglich, so kann die gewünschte Ausführung stattfinden. In diesem Fall werden einmalig, gemäß den Ausführungen in Kapitel 6.2 und Kapitel 6.3, die Mengen der activateMΘnK-Constraints und entailmentMΘnK-Constraints, sowie die RBAC-Sessions aktualisiert.

7.6 Zusammenfassung

Ziel der Validierung ist es, Widersprüche zwischen Constraints frühzeitig zu erkennen und eine möglichst konfliktfreie Ausführung der Prozesse zu ermöglichen.

Ein Konflikt kann zunächst innerhalb von assignMΘnK-Constraints auftreten, da bei diesen insbesondere $\Theta \in \{=, \geq\}$ erlaubt ist. In diesen Fällen muss K eine Mindestanzahl an Elementen enthalten. Erreicht K diese Mindestanzahl nicht, so ist die Einhaltung des assignMΘnK-Constraints nicht möglich. Wegen der Einschränkung $\Theta \in \{\leq\}$ bei activate-

MΘnK-Constraints und der „wenn-dann“-Semantik bei entailmentMΘnK-Constraints, kann bei diesen ein solcher Konflikt nicht auftreten.

Bei Betrachtung mehrerer MΘnK-Constraints ist es möglich, dass Konflikte zwischen assignMΘnK-Constraints auftreten. Das Problem ob eine Menge von assignMΘnK-Constraints widerspruchsfrei ist, ist sogar NP-vollständig. Es ist somit kein Algorithmus bekannt, der dieses Problem mit polynomiellem Zeitaufwand löst. Mit Hilfe der paarweisen Validierung von assignMΘnK-Constraints kann jedoch ein Teil der Konflikte in einer Menge von assignMΘnK-Constraints effizient gefunden werden. Die Validierung der assignMΘnK-Constraints findet dabei vollständig zur Modellierzeit statt.

Zur Laufzeit des Prozesses muss dagegen gewährleistet werden, dass sämtliche activateMΘnK-Constraints und entailmentMΘnK-Constraints eingehalten werden und der Prozess möglichst konfliktfrei ausgeführt wird. Da assignMΘnK-Constraints bereits vor der Ausführung des Prozesses durchgesetzt werden, kann deren Einhaltung zur Laufzeit vorausgesetzt werden.

Liegt zur Laufzeit des Prozesses ein Wunsch zur Aktivitätenausführung oder Rollenaktivierung vor, so muss sicher gestellt werden dass durch die Genehmigung dieses Wunsches keine activateMΘnK-Constraints verletzt werden. Hierzu werden alle activateMΘnK-Constraints daraufhin überprüft, ob diese die Ausführung oder Aktivierung verbieten. Wird die Ausführung oder Aktivierung durch keinen activateMΘnK-Constraint verboten, so entstünden durch die Durchsetzung der activateMΘnK-Constraints und entailmentMΘnK-Constraints neue activateMΘnK-Constraints. Durch diese kann die Ausführung anderer Aktivitäten des Prozesses unmöglich werden. Daher sind die entstehenden activateMΘnK-Constraints vor Genehmigung des Ausführungs- oder Aktivierungswunsches daraufhin zu überprüfen, ob weiterhin alle Aktivitäten ausgeführt werden können. Nur wenn dies der Fall ist, wird der vorliegende Ausführungs- oder Aktivierungswunsch genehmigt.

8 Implementierung

Im Folgenden wird die im Rahmen dieser Arbeit erfolgte Implementierung beschrieben. Diese wurde auf Basis der Anwendung „graphicalrmsSimulator“ vorgenommen. „graphicalrms-Simulator“ baut wiederum auf dem Projekt „ADEPT2“ auf, ein am Institut für Datenbanken und Informationssysteme der Universität Ulm entwickeltes Workflow-Management-System.

Mit Hilfe des entwickelten Tools ist die Formulierung von statischen Separation of Duty Constraints und Binding of Duty Constraints in Prozessinstanzen möglich. Auf den formulierten Constraints können dann die folgenden Prüfungen vorgenommen werden:

- Überprüfung der Einhaltung der assignMΘnK-Constraints durch die vergebenen Berechtigungen (Kapitel 6.1)
- Überprüfung der Erfüllbarkeit der assignMΘnK-Constraints (Kapitel 7.2)
- Paarweise Validierung von assignMΘnK-Constraints (Kapitel 7.4.2)

8.1 Zentrale implementierte Klassen

Die implementierten Klassen und deren Zusammenhänge werden in Abb. 8.1 als UML2-Klassendiagramm veranschaulicht. Dabei sind die Klassen

- `Constraint`
- `MonkConstraint`
- `ConstraintTemplate`

abstrakte Oberklassen.

Die Klasse `InstanceConstraintCheck` wird für jede Prozessinstanz instanziert und verwaltet die Überprüfung der Constraints innerhalb der jeweiligen Prozessinstanz. Hierzu sind jeder Instanz der Klasse `InstanceConstraintCheck` beliebig viele Instanzen der Klasse `ConstraintTemplate` zugeordnet.

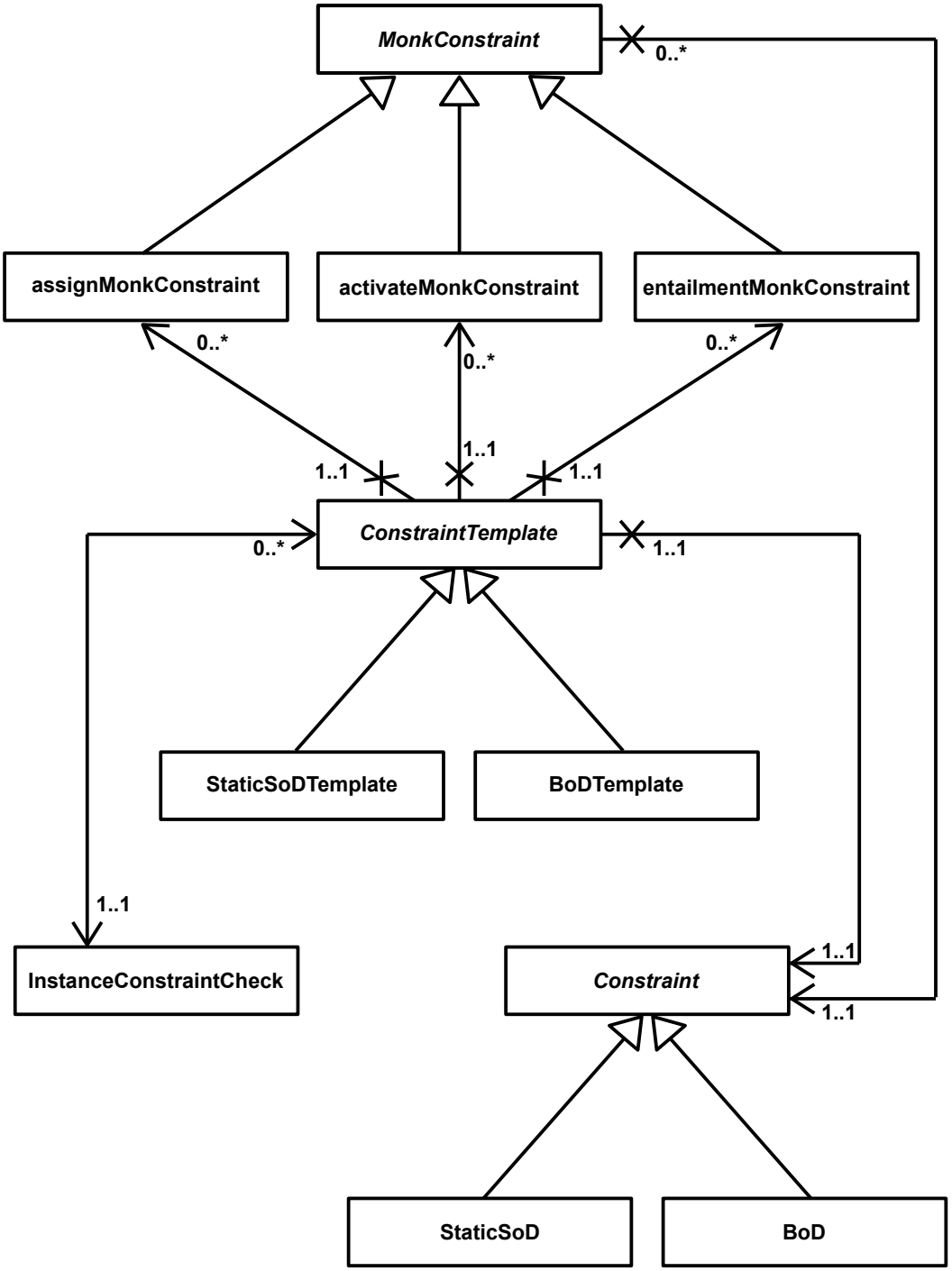


Abb. 8.1: UML2-Klassendiagramm der implementierten Klassen.

Eine Instanz der Klasse `ConstraintTemplate` verwaltet jeweils einen Constraint und alle diesen repräsentierenden MΘnK-Constraints. Hierzu sind einer Instanz der Klasse `ConstraintTemplate` beliebig viele Instanzen der Klassen `assignMonkConstraint`, `activateMonkConstraint` und `entailmentMonkConstraint` zugeordnet. Zusätzlich ist ihr genau eine Instanz der Klasse `Constraint` zugeordnet.

Beim Anlegen eines neuen Constraints in einer Prozessinstanz wird von der Klasse `InstanceConstraintCheck` ein `ConstraintTemplate` instanziert. Dieses instanziert wiederum alle den Constraint repräsentierenden `MonkConstraint`s.

Änderungen an Constraints werden mit Hilfe des Observer-Patterns propagiert. Wird eine Instanz der Klasse `Constraint` geändert, so wird das `ConstraintTemplate` hierüber informiert. Dieses nimmt die Anpassung der `MonkConstraint`s vor. Von diesen Änderungen im `ConstraintTemplate` wird wiederum die Klasse `InstanceConstraintCheck` unterrichtet, so dass diese die entsprechenden Constraint-Prüfungen erneut vornimmt.

8.2 Einsatz des entwickelten Tools am Fallbeispiel

Anhand des Fallbeispiels wird im Folgenden die Funktionsweise des entwickelten Tools vorgestellt. Hierzu wurde der bekannte Warenbestellungs-Prozess modelliert (Abb. 8.2) und in der Anwendung „graphicalrmsSimulator“ geladen (Abb. 8.3).

Der Dialog „Change Constraints“ ermöglicht zunächst das Anzeigen der Constraints einer Instanz des Warenbestellungs-Prozesses. Abb. 8.4 zeigt einen statischen SoD Constraint namens „SSoD_SchreibenPrüfen“ mit den Aktivitäten „Bestellung schreiben“ und „Bestellung prüfen“. In Abb. 8.5 wurde der BoD Constraint „BoD_Finanzen“ formuliert, der die Ausführung der Aktivitäten „Rechnung erfassen“, „Rechnung prüfen“ und „Bezahlung veranlassen“ durch denselben Benutzer fordert.

Das Anlegen eines Constraints geschieht über die Dialoge „Static SoD Configure Constraint“ bzw. „BoD Configure Constraint“, die über den Listeneintrag „+ add constraint +“ geöffnet werden. Abb. 8.6 und Abb. 8.7 zeigen das Anlegen der beiden Constraints „SSoD_SchreibenPrüfen“ und „BoD_Finanzen“ mittels dieser Dialoge.

Wurden die gewünschten Constraints angelegt, so ermöglicht das Tool die Durchführung von Prüfungen auf den statischen Teilen der formulierten Constraints. Die Ergebnisse die-

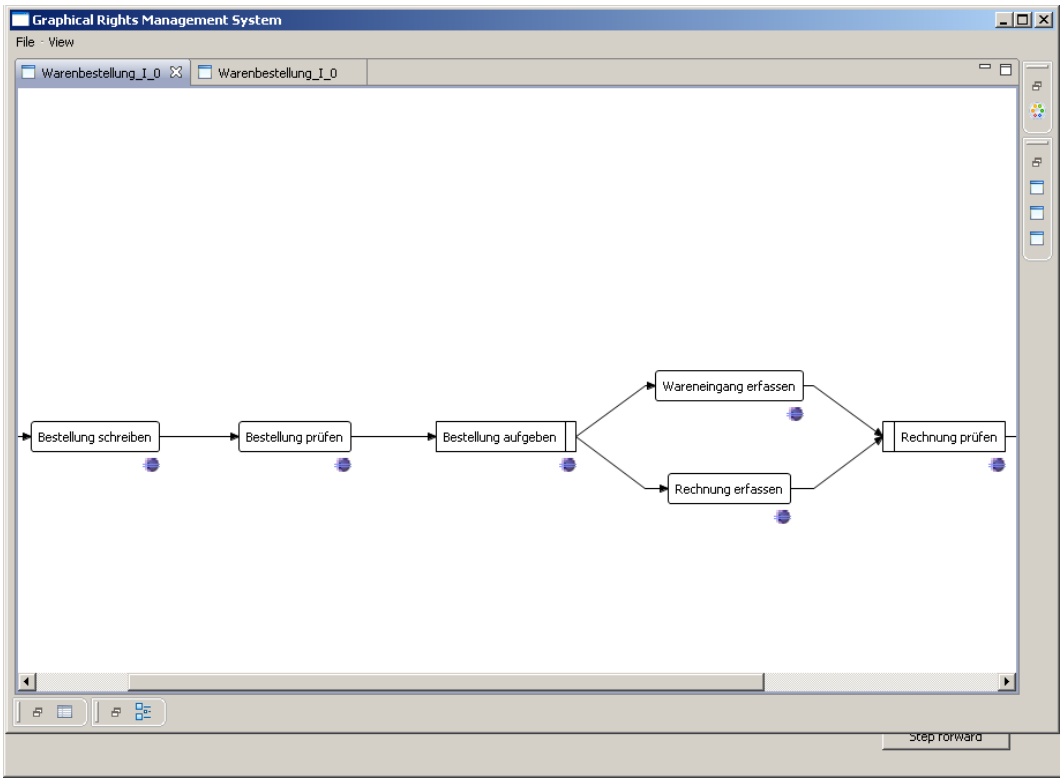


Abb. 8.2: Der Warenbestellungs-Prozess.

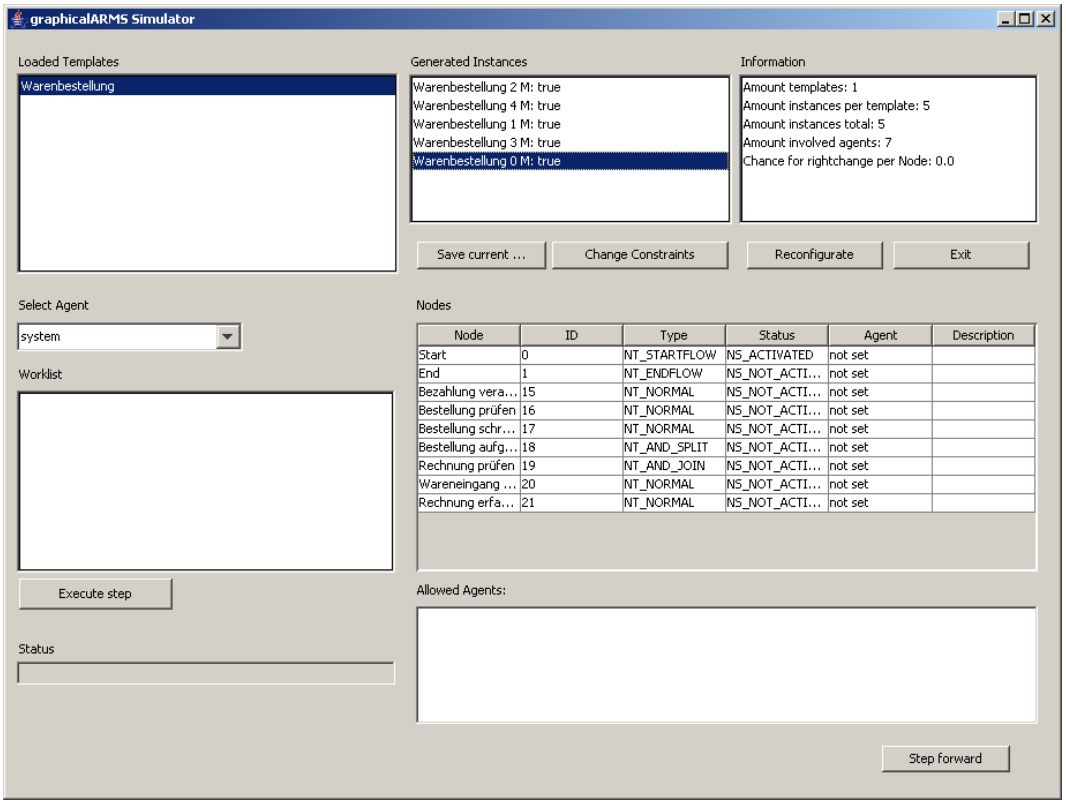


Abb. 8.3: „graphicalrmsSimulator“ mit geladenem Warenbestellungs-Prozess.

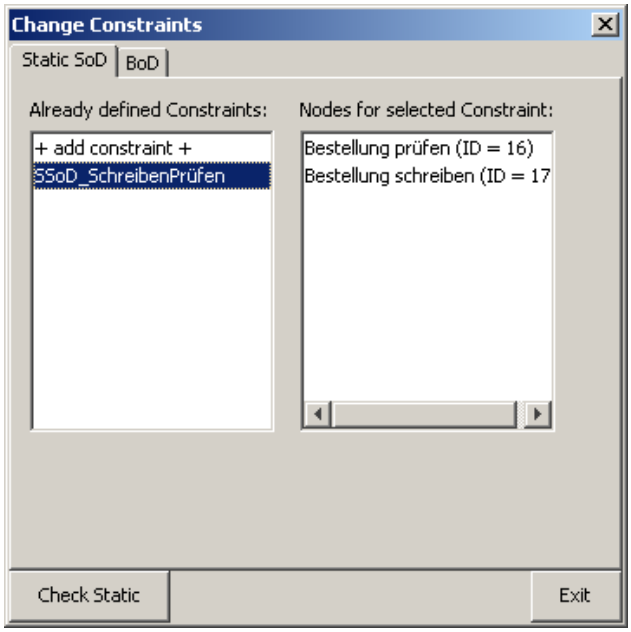


Abb. 8.4: SSoD Constraint „SSoD_SchreibenPrüfen“ mit den Aktivitäten „Bestellung schreiben“ und „Bestellung prüfen“.

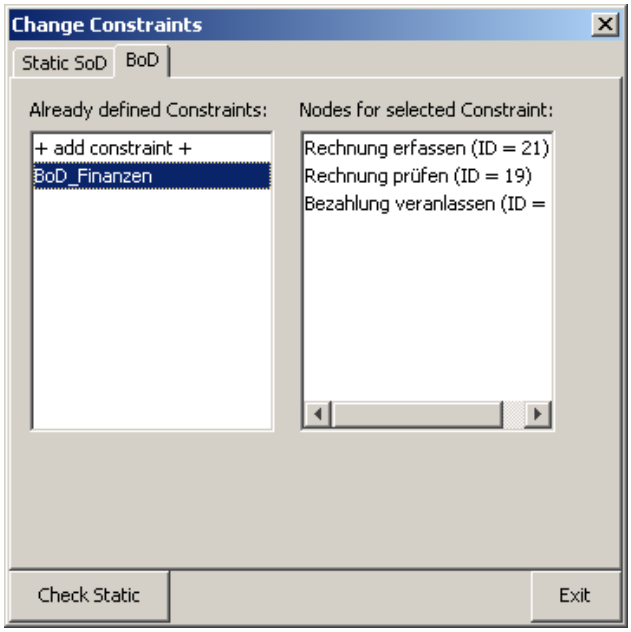


Abb. 8.5: BoD Constraint „BoD_Finanzen“ mit den Aktivitäten „Rechnung erfassen“, „Rechnung prüfen“ und „Bezahlung veranlassen“.

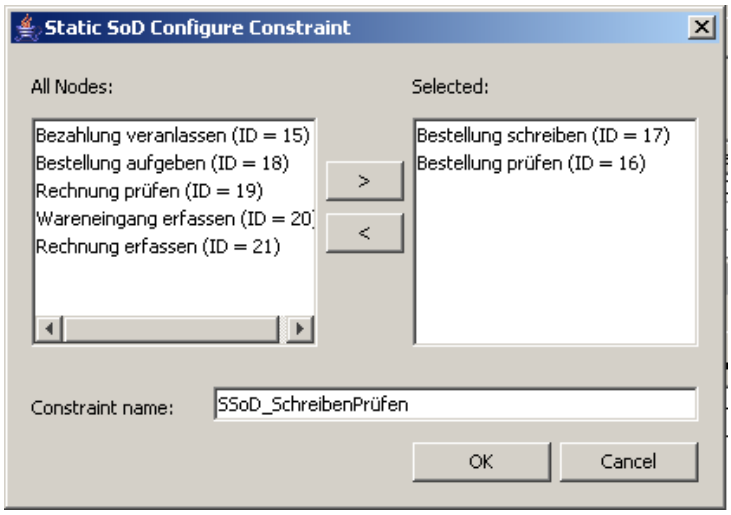


Abb. 8.6: Anlegen des SSoD Constraints „SSoD_SchreibenPrüfen“.

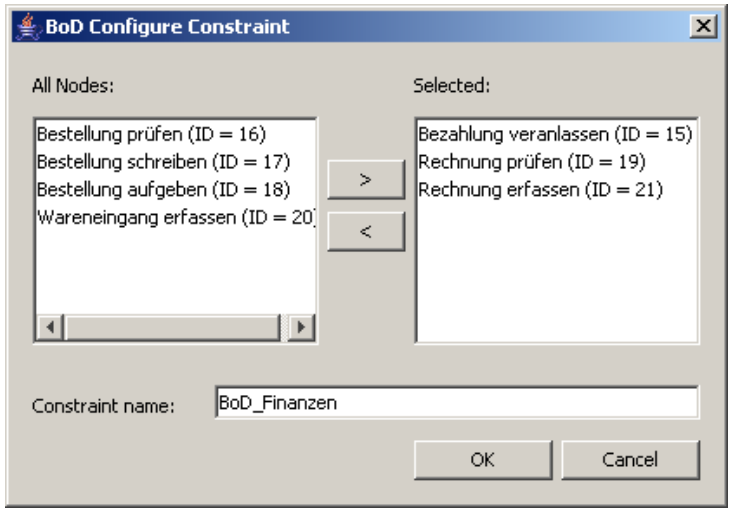


Abb. 8.7: Anlegen des BoD Constraints „BoD_Finanzen“.

ser Überprüfungen werden nach Betätigen der Schaltfläche „Check Static“ in einem neuen Dialog namens „Static Constraint Check Results“ angezeigt.

Abb. 8.8 zeigt alle für die Constraints „SSoD_SchreibenPrüfen“ und „BoD_Finanzen“ durch das jeweilige ConstraintTemplate automatisch erzeugten assignMΘnK-Constraints. Zusätzlich liefert Abb. 8.8 die Information, dass alle assignMΘnK-Constraints erfüllbar sind (✓). Mittels der Schaltflächen am unteren Rand des Dialoges ist es möglich, nur erfüllbare bzw. nur nicht-erfüllbare assignMΘnK-Constraints anzuzeigen.

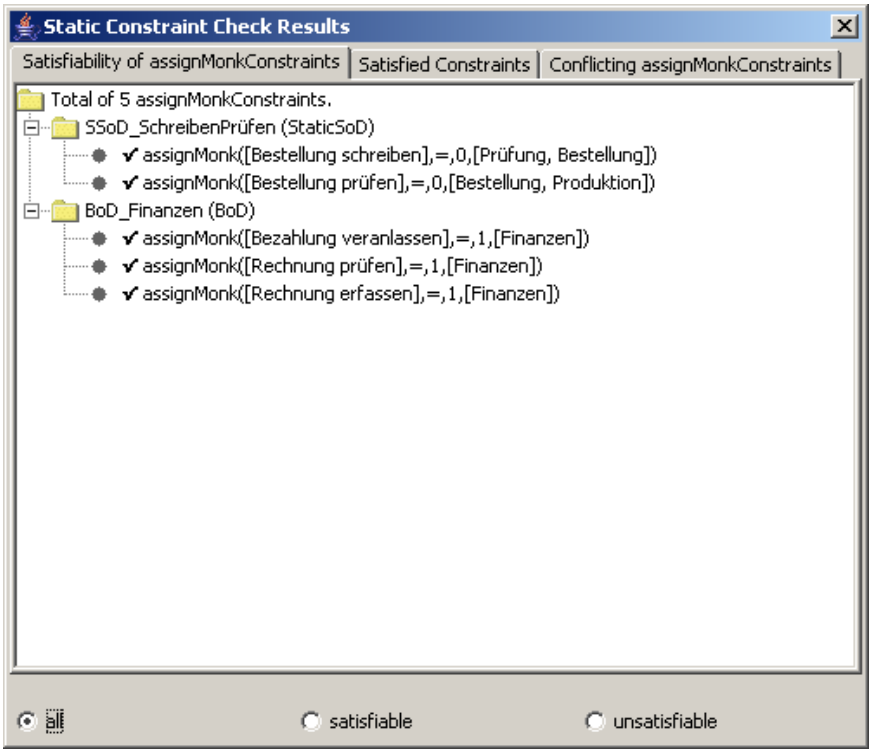


Abb. 8.8: assignMΘnK-Constraints der angelegten Constraints „SSoD_SchreibenPrüfen“ und „BoD_Finanzen“, sowie Informationen über deren Erfüllbarkeit.

Weiter ermöglicht das Tool die Überprüfung, ob die assignMΘnK-Constraints, und damit die ursprünglichen Constraints, eingehalten werden. Abb. 8.9 zeigt die Ergebnisse dieser Überprüfung für die Constraints „SSoD_SchreibenPrüfen“ und „BoD_Finanzen“. Im Beispiel wird der Constraint „BoD_Finanzen“ eingehalten, da, entsprechend der Berechtigungszuordnung in dieser Ausarbeitung, die Aktivitäten dieses Constraints jeweils nur der Rolle „Finanzen“ zugeordnet sind. Der Constraint „SSoD_SchreibenPrüfen“ wird da-

gegen nicht eingehalten. Das Tool liefert dann nähere Informationen darüber, welche assignMΘnK-Constraints des Constraints nicht eingehalten wurden („Failure Descriptions“). In diesem Fall resultiert die Nicht-Einhaltung des Constraints aus der Zuordnung der Rolle „Bestellung“ zu beiden Aktivitäten „Bestellung schreiben“ und „Bestellung prüfen“.

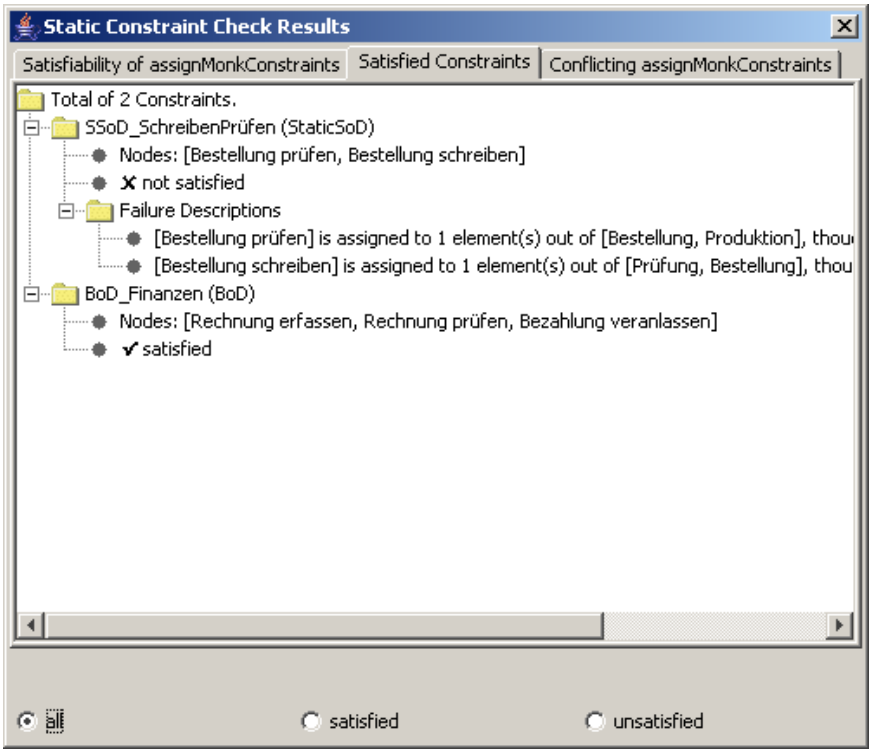


Abb. 8.9: Nicht-Einhaltung des Constraints „SSoD_SchreibenPrüfen“ und Einhaltung des Constraints „BoD_Finanzen“.

Zusätzlich wurde die paarweise Validierung von assignMΘnK-Constraints implementiert. Die Ergebnisse der paarweisen Validierung für die assignMΘnK-Constraints der Constraints „SSoD_SchreibenPrüfen“ und „BoD_Finanzen“ zeigt Abb. 8.10. Da die Aktivitäten der beiden Constraints disjunkt sind, ist die gleichzeitige Erfüllung beider Constraints möglich.

Zu Demonstrationszwecken wird ein zusätzlicher BoD Constraint „BoD_SchreibenPrüfen“ auf den Aktivitäten „Bestellung schreiben“ und „Bestellung prüfen“ angelegt (Abb. 8.11). Dieser steht offenbar im Widerspruch zu Constraint „SSoD_SchreibenPrüfen“. Abb. 8.12

zeigt den durch die paarweise Validierung aufgedeckten Widerspruch dieser beiden Constraints.

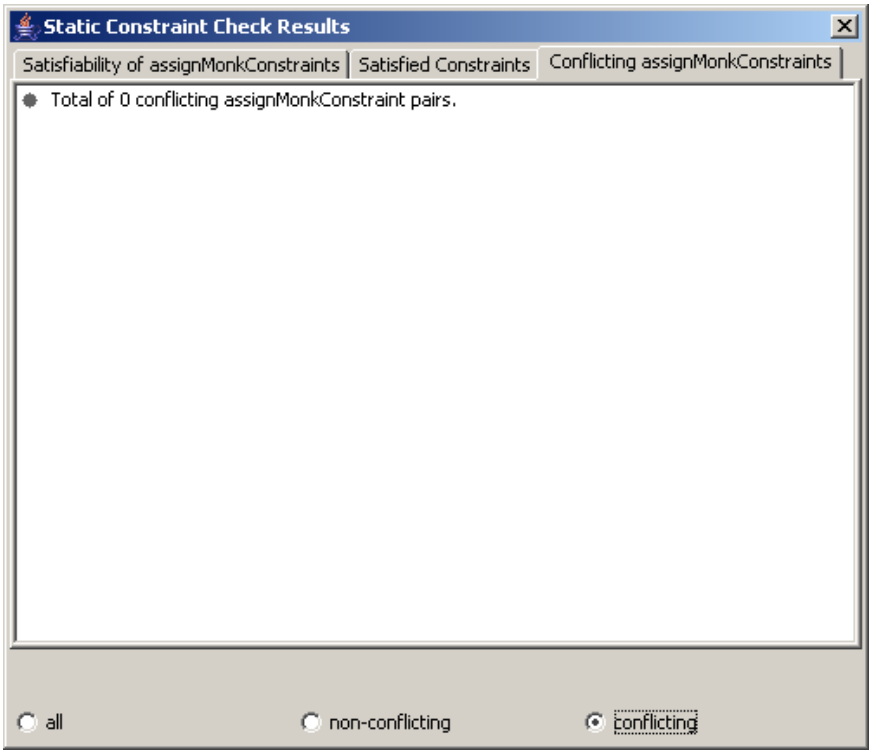


Abb. 8.10: Ergebnisse der paarweisen Validierung der assignMΘnK-Constraints der Constraints „SSoD_SchreibenPrüfen“ und „BoD_Finanzen“.

Die Quelltexte und deren ausführliche Dokumentation befinden sich auf der beiliegenden DVD (siehe Anhang B). Auf dieser ist ebenfalls eine virtuelle Maschine enthalten, in der die vorgenommene Implementierung gesichtet, erprobt und weiterentwickelt werden kann. Installation und Start der virtuellen Maschine, sowie der Start der Anwendung „graphicalrms-Simulator“ werden in Anhang C beschrieben. Details zur Bedienung des im Rahmen dieser Arbeit entwickelten Tools können Anhang D entnommen werden.

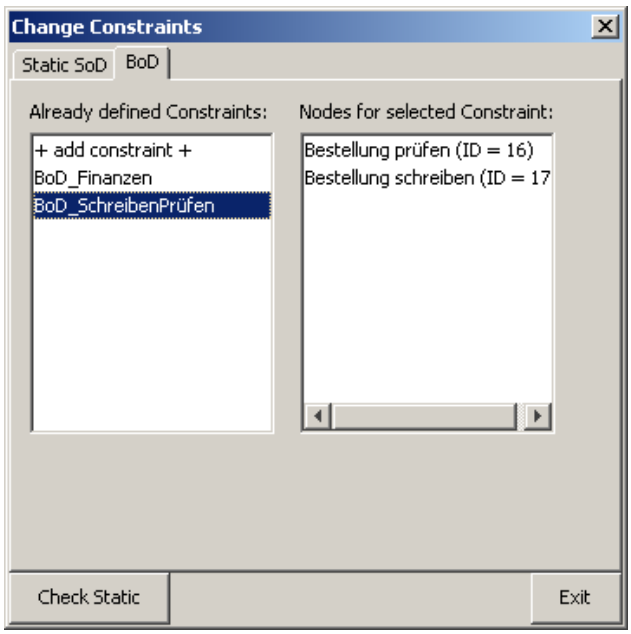


Abb. 8.11: BoD Constraint „BoD_SchreibenPrüfen“ mit den Aktivitäten „Bestellung schreiben“ und „Bestellung prüfen“.

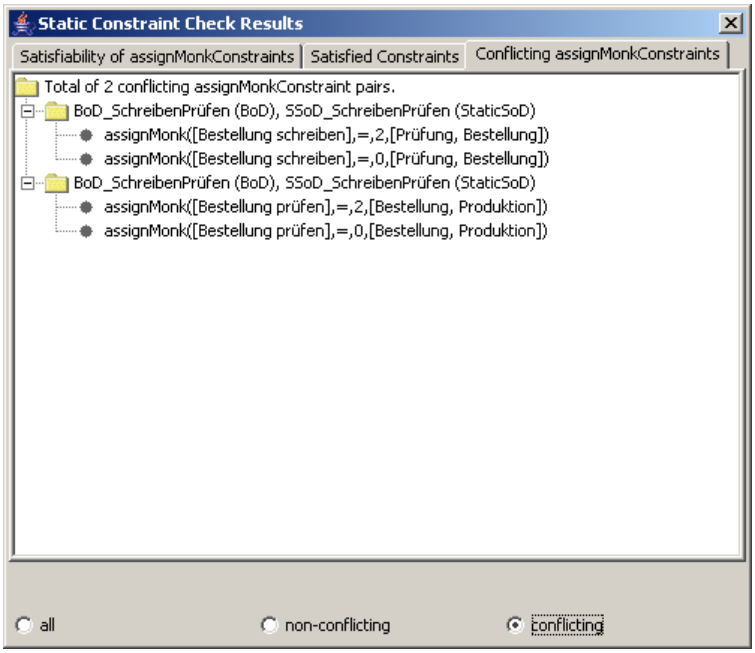


Abb. 8.12: Im Widerspruch zueinander stehende assignMΘnK-Constraints der Constraints „SSoD_SchreibenPrüfen“ und „BoD_SchreibenPrüfen“.

9 Zusammenfassung

In dieser Arbeit wurde ein Modell zum Einsatz von Authorization Constraints in Workflow-Management-Systemen vorgestellt. Durch Authorization Constraints können komplexe Sicherheitsanforderungen in Prozessen formuliert und durchgesetzt werden. Ein besonderes Augenmerk lag dabei auf der Durchsetzung und Validierung der Authorization Constraints.

Das in dieser Arbeit eingeführte MΘnK-Modell ermöglicht die Modellierung vieler unterschiedlicher Authorization Constraints auf eine einheitliche Weise. Durch die unterschiedlichen Semantiken von assignMΘnK-Constraints, activateMΘnK-Constraints und entailmentMΘnK-Constraints ist mit MΘnK sowohl die statische, als auch die dynamische Einschränkung von Berechtigungen möglich. Somit können alle in Workflow-Management-Systemen wichtigen Authorization Constraints auf MΘnK-Constraints abgebildet werden. Für die wichtigsten Authorization Constraints wurden Templates entwickelt, durch die eine automatisierte Abbildung auf MΘnK-Constraints stattfinden kann.

Durch die Modellierung der Authorization Constraints mit MΘnK-Constraints wird deren einheitliche Durchsetzung und Validierung ermöglicht. Diese Vorgänge finden dann ausschließlich auf Basis der MΘnK-Constraints statt, so dass die ursprünglichen Authorization Constraints mittels der MΘnK-Constraints durchgesetzt und validiert werden. Durchsetzung und Validierung finden somit für alle Authorization Constraints auf dieselbe Weise statt.

Die Durchsetzung der, auf assignMΘnK-Constraints abgebildeten, statischen Authorization Constraints findet vollständig zur Modellierzeit statt. Die Einhaltung der assignMΘnK-Constraints wird mit Hilfe einfacher Mengenoperationen überprüft. Durch die einheitliche Abbildung der dynamischen Authorization Constraints auf activateMΘnK-Constraints und entailmentMΘnK-Constraints, ist auch deren Durchsetzung zur Laufzeit auf eine einheitliche Weise möglich. Bei Vorliegen eines Wunsches zur Aktivierung einer Rolle oder Ausführung einer Aktivität sind alle zutreffenden activateMΘnK-Constraints lediglich auf $n = 0$ hin zu überprüfen. Bei Genehmigung des vorliegenden Wunsches geschieht die Durchsetzung

durch Ersetzen der zutreffenden activateMΘnK-Constraints. Auch die Durchsetzung zutreffender entailmentMΘnK-Constraints ist durch deren Löschen und die zukünftige Einhaltung des jeweiligen Folgeconstraints auf einfache Weise möglich.

Ebenso ist die Validierung der Authorization Constraints durch ihre Abbildung auf MΘnK-Constraints auf eine einheitliche Weise möglich. Dabei werden die assignMΘnK-Constraints der statischen Authorization Constraints zur Modellierzeit auf Widersprüche hin untersucht. Dabei hat sich herausgestellt, dass das Problem ob eine Menge von assignMΘnK-Constraints widerspruchsfrei ist, NP-vollständig ist. Aus diesem Grund wurden Vorgehensweisen vorgestellt, durch die ein Teil der Konflikte in einer Menge von assignMΘnK-Constraints effizient gefunden werden kann. So können mit der paarweisen Validierung Widersprüche zwischen zwei gegebenen assignMΘnK-Constraints mit Hilfe einfacher Mengenoperationen gefunden werden.

Durch die Validierung der activateMΘnK-Constraints und entailmentMΘnK-Constraints zur Laufzeit wird sichergestellt, dass stets alle Aktivitäten des Prozesses ausführbar sind. Liegt ein Aktivierungs- oder Ausführungswunsch vor, der die Ausführung späterer Aktivitäten unmöglich machen würde, so wird dieser Wunsch nicht genehmigt. Auf diese Weise besteht frühzeitig die Möglichkeit, dass ein anderer Benutzer die gewünschte Aktivierung oder Ausführung vornimmt und so nicht die Ausführung späterer Aktivitäten eingeschränkt wird.

Im Rahmen dieser Arbeit wurde die Abbildung von statischen SoD Constraints und BoD Constraints auf MΘnK-Constraints, sowie die zur Modellierzeit durchführbaren Teile der Validierung und Durchsetzung implementiert. Durch diese Implementierung wurde die Umsetzbarkeit der erarbeiteten Konzepte gezeigt.

Insgesamt ermöglicht MΘnK die einheitliche Modellierung vieler unterschiedlicher Authorization Constraints. Darauf basierend kann die Durchsetzung und Validierung der MΘnK-Constraints auf einheitliche Weise erfolgen. Durch die Abbildung auf MΘnK-Constraints werden die ursprünglichen Authorization Constraints mittelbar validiert und durchgesetzt.

10 Ausblick

An einigen Stellen dieser Arbeit wurden Einschränkungen, wie der Verzicht auf Rollenhierarchien, vorgenommen, um die Komplexität der Betrachtungen zu reduzieren. Der Verzicht auf diese Einschränkungen sollte Gegenstand weiterer Arbeiten sein. Interessante und wichtige Aspekte betreffen außerdem die Erweiterung des in dieser Arbeit vorgestellten MΘnK-Modells um weitere Sicherheitskonzepte, bzw. das Zusammenspiel von MΘnK mit diesen.

Rollenhierarchien und RBAC-Sessions

Zunächst wurde in Kapitel 4.1.1 auf das in RBAC bekannte Prinzip der Rollenhierarchien verzichtet. Ein interessanter Aspekt ist daher, inwiefern Rollenhierarchien mit dem erarbeiteten MΘnK-Modell umgesetzt werden können. Zwar sind die Sicherheitsrichtlinien prinzipiell auch ohne Rollenhierarchien formulierbar, jedoch kann deren Einsatz durchaus sinnvoll sein, um die Berechtigungsvergabe zu vereinfachen. Ein weiterer Aspekt betrifft das aus RBAC bekannte Session-Prinzip, von dem bei der Aktivierung von Rollen Gebrauch gemacht wurde. Weitere Überlegungen könnten eine weitergehende Verwendung der Sessions ermöglichen, so dass unter Umständen aufwendige Validierungen der MΘnK-Constraints entfallen können bzw. nicht wiederholt getätigt werden müssen.

Inter-Instance-Constraints

Intensive Betrachtungen sind außerdem auf dem Gebiet der Inter-Instance Constraints notwendig. Während die in Kapitel 5.3 modellierten Authorization Constraints stets nur innerhalb einzelner Prozessinstanzen umgesetzt werden, wurden bereits in Kapitel 3.6 Inter-Instance Constraints vorgestellt. Diese ermöglichen die Umsetzung von Prozessinstanzübergreifenden Sicherheitsanforderungen. Der Einsatz derartiger Constraints in WfMS ist wünschenswert, mit dem vorgestellten MΘnK-Konzept jedoch bisher nicht möglich.

Validierung

Weitere offene Arbeiten bestehen im Bereich der Validierung der MΘnK-Constraints. So wäre eine weitreichendere und tiefer gehende Validierung der activateMΘnK-Constraints und entailmentMΘnK-Constraints wünschenswert, so dass Konflikte zur Laufzeit frühzeitiger erkannt werden können. Mit der Validierung von assignMΘnK-Constraints zur Modellierzeit liegt dagegen ein NP-vollständiges Problem vor. Hierzu sind bekannte Lösungsansätze (z.B. Backtracking) auf ihre Eignung und Performanz hin zu untersuchen und zu vergleichen, um so Widersprüche möglichst effizient zu finden. In diesem Zusammenhang sind auch Untersuchungen sinnvoll, wie viele der Widersprüche bereits durch einfache und effiziente Überprüfungen, wie die betrachtete paarweise Validierung, gefunden werden können.

Konfliktauflösung zur Laufzeit

Nicht betrachtet wurde in dieser Arbeit die Auflösung von zur Laufzeit auftretenden Konflikten durch MΘnK-Constraints. Gegenstand weiterer Arbeiten wäre daher, inwiefern Konflikte zur Laufzeit automatisiert aufgelöst werden können, oder inwiefern dem Constraintmodellierer Vorschläge zur sinnvollen Auflösung der Konflikte durch das WfMS unterbreitet werden können.

Änderungen zur Laufzeit

Ein weiterer interessanter Punkt betrifft Änderungen, die während der Laufzeit des Prozesses auftreten können. Dabei können Änderungen

- am Prozess (z.B. Hinzufügen oder Löschen von Aktivitäten),
- an den mittels RBAC vergebenen Berechtigungen,
- an den Authorization Constraints

auftreten. Bei derartigen Änderungen ist darauf zu achten, dass alle Constraints weiterhin eingehalten werden und keine Widersprüche entstehen. Naive Ansätze würden bei jeder Änderung eine Neuvalidierung aller MΘnK-Constraints vornehmen und die gewünschten Änderungen entsprechend genehmigen oder verbieten. Ziel weiterer Arbeiten sollte es sein, effiziente Verfahren zu entwickeln, die eine Neuvalidierung aller MΘnK-Constraints unnötig machen.

Teil III

Verzeichnisse

Literaturverzeichnis

[1] BALDWIN, R. : Naming and grouping privileges to simplify security management in large databases. (1990). http://dx.doi.org/10.1109/RISP.1990.63844. – DOI 10.1109/RISP.1990.63844

[2] BERTINO, E. ; FERRARI, E. ; ATLURI, V. : A Flexible Model Supporting the Specification and Enforcement of Role-based Authorizations in Workflow Management Systems. (1997), 1. http://dx.doi.org/10.1145/266741.266746. – DOI 10.1145/266741.266746

[3] BERTINO, E. ; BETTINI, C. ; FERRARI, E. ; SAMARATI, P. : A Temporal Access Control Mechanism for Database Systems. In: *IEEE Trans. on Knowl. and Data Eng.* 8 (1996), Nr. 1, 67–80. http://dx.doi.org/10.1109/69.485637. – DOI 10.1109/69.485637. – ISSN 1041–4347

[4] BERTINO, E. ; BETTINI, C. ; FERRARI, E. ; SAMARATI, P. : An access control model supporting periodicity constraints and temporal reasoning. In: *ACM Trans. Database Syst.* 23 (1998), Nr. 3, S. 231–285. http://dx.doi.org/10.1145/293910.293151. – DOI 10.1145/293910.293151. – ISSN 0362–5915

[5] BERTINO, E. ; FERRARI, E. ; ATLURI, V. : The specification and enforcement of authorization constraints in workflow management systems. In: *ACM Trans. Inf. Syst. Secur.* 2 (1999), Nr. 1, S. 65–104. http://dx.doi.org/10.1145/300830.300837. – DOI 10.1145/300830.300837. – ISSN 1094–9224

[6] BOTHA, R. A. ; ELOFF, J. H. P.: Separation of duties for access control enforcement in workflow environments. In: *IBM Syst. J.* 40 (2001), Nr. 3, 666–682. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.83.6573. – ISSN 0018–8670

[7] CRAMPTON, J. : On the Satisfiability of Constraints in Workflow Systems. (2004). http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.114.7250

[8] CRAMPTON, J. : A reference monitor for workflow systems with constrained task execution. (2005), 38–47. http://dx.doi.org/10.1145/1063979.1063986. – DOI 10.1145/1063979.1063986. ISBN 1–59593–045–0

[9] CRAMPTON, J. ; KHAMBHAMMETTU, H. : Delegation in role-based access control. In: *Int. J. Inf. Secur.* 7 (2008), Nr. 2, S. 123–136. http://dx.doi.org/10.1007/s10207-007-0044-8. – DOI 10.1007/s10207–007–0044–8. – ISSN 1615–5262

[10] FERRAIOLO, D. F. ; BARKLEY, J. F. ; KUHN, D. R.: A role-based access control model and reference implementation within a corporate intranet. In: *ACM Trans. Inf. Syst. Secur.* 2 (1999), Nr. 1, 34–64. http://dx.doi.org/10.1145/300830.300834. – DOI 10.1145/300830.300834. – ISSN 1094–9224

[11] FERRAIOLO, D. F. ; CUGINI, J. ; KUHN, D. : Role-based access control (RBAC): Features and motivations. (1995), 11–15. http://brutus.ncsl.nist.gov/groups/SNS/rbac/documents/ferraiolo-cugini-kuhn-95.pdf

[12] FERRAIOLO, D. F. ; KUHN, D. R. ; CHANDRAMOULI, R. : *Role-Based Access Control*. Artech House http://books.google.de/books?hl=de&lr=&id=48AeIhQLWckC&oi=fnd&pg=PR15&dq=%22Role-Based+Access+Control%22+artech&ots=LJXIDCsRD2&sig=3WnBdZxr1tEUSk7Xs6Z_GKjkaVM

[13] FERRAIOLO, D. F. ; SANDHU, R. ; GAVRILA, S. ; KUHN, D. R. ; CHANDRAMOULI, R. : Proposed NIST standard for role-based access control. In: *ACM Trans. Inf. Syst. Secur.* 4 (2001), Nr. 3, S. 224–274. http://dx.doi.org/10.1145/501978.501980. – DOI 10.1145/501978.501980. – ISSN 1094–9224

[14] GLIGOR, V. ; GAVRILA, S. L. ; FERRAIOLO, D. : On the Formal Definition of Separation-of-Duty Policies and their Composition. (1998), 172–183. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.32.4744

[15] JOSHI, J. ; BERTINO, E. ; LATIF, U. ; GHAFOOR, A. : A generalized temporal role-based access control model. In: *IEEE Transactions on Knowledge and Data Engineering* 17 (2005), Nr. 1, 4–23. http://dx.doi.org/10.1109/TKDE.2005.1. – DOI 10.1109/TKDE.2005.1

[16] KUHN, D. : Mutual Exclusion of Roles as a Means of Implementing Separation of Duty in Role-Based Access Control Systems. (1997), S. 23. http://dx.doi.org/10.1145/266741.266749. – DOI 10.1145/266741.266749

[17] LI, N. ; TRIPUNITARA, M. V. ; BIZRI, Z. : On mutually exclusive roles and separation-of-duty. In: *ACM Trans. Inf. Syst. Secur.* 10 (2007), Nr. 2, S. 5. http://dx.doi.org/10.1145/1237500.1237501. – DOI 10.1145/1237500.1237501. – ISSN 1094–9224

[18] LIU, D.-R. ; WU, M.-Y. ; LEE, S.-T. : Role-based authorizations for workflow systems in support of task-based separation of duty. In: *J. Syst. Softw.* 73 (2004), Nr. 3, S. 375–387. http://dx.doi.org/10.1016/S0164-1212(03)00175-4. – DOI 10.1016/S0164–1212(03)00175–4. – ISSN 0164–1212

[19] NEUMANN, G. ; STREMBECK, M. : An approach to engineer and enforce context constraints in an RBAC environment. (2003), S. 65–79. http://dx.doi.org/10.1145/775412.775421. – DOI 10.1145/775412.775421. ISBN 1–58113–681–1

[20] OSBORN, S. ; SANDHU, R. ; MUNAWER, Q. : Configuring role-based access control to enforce mandatory and discretionary access control policies. In: *ACM Trans. Inf. Syst. Secur.* 3 (2000), Nr. 2, S. 85–106. http://dx.doi.org/10.1145/354876.354878. – DOI 10.1145/354876.354878. – ISSN 1094–9224

[21] PAPADIMITRIOU, C. : The NP-completeness of the bandwidth minimization problem. In: *Computing* 16 (1976), Nr. 3, S. 263–270. http://dx.doi.org/10.1007/BF02280884. – DOI 10.1007/BF02280884

[22] SALTZER, J. ; SCHROEDER, M. : The protection of information in computer systems. 63 (1975), 1278–1308. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.128.2905

[23] SANDHU, R. ; FERRAIOLO, D. ; KUHN, R. : The NIST model for role-based access control: towards a unified standard. (2000), S. 47–63. http://dx.doi.org/10.1145/344287.344301. – DOI 10.1145/344287.344301. ISBN 1–58113–259–X

[24] SANDHU, R. ; MUNAWER, Q. : How to do discretionary access control using roles. (1998), S. 47–54. http://dx.doi.org/10.1145/286884.286893. – DOI 10.1145/286884.286893. ISBN 1–58113–113–5

[25] SANDHU, R. ; SAMARATI, P. : Access control: principle and practice. In: *IEEE Communications Magazine* 32 (1994), Nr. 9, 40–48. http://www.list.gmu.edu/journals/commun/i94ac(org).pdf

[26] SCHÖNING, U. ; TORÁN, J. : *Berechenbarkeit und Komplexität*. Vorlesungsskript, 10 2005

[27] SEIPEL, D. ; GESKE, U. : Solving cardinality constraints in (constraint) logic programming. (2001). http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.22.1057

[28] SIMON, R. ; ZURKO, M. E.: Separation of Duty in Role-based Environments. (1997), 183. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.28.6661. ISBN 0–8186–7990–5

[29] TAN, K. ; CRAMPTON, J. ; GUNTER, C. A.: The Consistency of Task-Based Authorization Constraints in Workflow Systems. (2004), S. 155. http://dx.doi.org/10.1109/CSFW.2004.22. – DOI 10.1109/CSFW.2004.22. ISBN 0–7695–2169–X

[30] TOLONE, W. ; AHN, G.-J. ; PAI, T. ; HONG, S.-P. : Access control in collaborative systems. In: *ACM Comput. Surv.* 37 (2005), Nr. 1, 29–41. http://dx.doi.org/10.1145/1057977.1057979. – DOI 10.1145/1057977.1057979. – ISSN 0360–0300

[31] WAINER, J. ; BARTHELMESS, P. ; KUMAR, A. : W-RBAC-a workflow security model incorporating controlled overriding of constraints. In: *International Journal of Cooperative Information Systems* 12 (2003), Nr. 4, 455–485. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.9.5720

[32] WAINER, J. ; KUMAR, A. ; BARTHELMESS, P. : Security Management in the presence of delegation and revocation in workflow systems. (2001), 18. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.18.8533

[33] WARNER, J. ; ATLURI, V. : Inter-instance authorization constraints for secure workflow management. (2006), S. 190–199. http://dx.doi.org/10.1145/1133058.1133085. – DOI 10.1145/1133058.1133085. ISBN 1–59593–353–0

[34] WOLTER, C. ; SCHAAD, A. ; MEINEL, C. : Task-based entailment constraints for basic workflow patterns. (2008), S. 51–60. http://dx.doi.org/10.1145/1377836.1377844. – DOI 10.1145/1377836.1377844. ISBN 978–1–60558–129–3

[35] ZHANG, L. ; AHN, G.-J. ; CHU, B.-T. : A rule-based framework for role-based delegation and revocation. In: *ACM Trans. Inf. Syst. Secur.* 6 (2003), Nr. 3, 404–441. http://dx.doi.org/10.1145/937527.937530. – DOI 10.1145/937527.937530. – ISSN 1094–9224

Glossar

Authorization Constraint 21
Mechanismus zur dynamischen, flexiblen und systematischen Einschränkung von Berechtigungen.

Benutzer 10
Mitarbeiter der Organisation, der gleichzeitig Anwender des Workflow-Management-Systems ist.

Benutzer-Rollen-Zuordnung 14
Zuordnung zwischen Benutzern und Rollen bei Role-Based Access Control.

Benutzerzuordnung 14
siehe Benutzer-Rollen-Zuordnung.

Berechtigungsvergabe 12
Die Vergabe von Berechtigungen an Benutzer.

Berechtigungszuordnung 14
Zuordnung zwischen Berechtigungen und Rollen bei Role-Based Access Control.

Binding of Duty 36
Ein Constraint, der die Ausführung mehrerer Aktivitäten durch denselben Benutzer fordert.

Cardinality Constraint 37
Constraint der sich darauf bezieht, wie oft eine bestimmte Aktion geschehen darf oder muss.

Constraint 21
Im Rahmen dieser Arbeit stets ein Authorization Constraint.

Datenfluss 10
Beschreibung der Ein- und Ausgangsdaten der einzelnen Aktivitäten.

Domain 26
Eine Menge von Benutzern, durch die der Anwendungsbereich eines Entailment Constraints eingeschränkt wird.

Dynamische Constraints 42
Constraints, die erst zur Laufzeit analysiert und durchgesetzt werden können.

Dynamisches Separation of Duty 33
Ein Separation of Duty-Constraint, der von einer dynamischen Beschaffenheit ist und somit erst zur Ausführungszeit des Prozesses ausgewertet werden kann.

Dynamisches Separation of Duty auf Aktivitäten 52
Ein DSoD-Constraint, der die Ausführung der spezifizierten Aktivitäten durch unterschiedliche Benutzer fordert.

Eingeschränkte Aktivität 26
Die im Rahmen eines Entailment Constraints von der einschränkenden Aktivität beeinflusste Aktivität.

Einschränkende Aktivität 26
Die im Rahmen eines Entailment Constraints die eingeschränkte Aktivität beeinflussende Aktivität.

Entailment Constraint 26
Authorization Constraint der sich auf zwei Aktivitäten bezieht.

entailmentMΘnK-Constraint 65
MΘnK-Constraint, der die Formulierung von activateMΘnK-Constraints in Abhängigkeit der Ausführungshistorie ermöglicht.

Erfüllbarkeit 116
eines MΘnK-Constraints. Ein MΘnK-Constraint ist erfüllbar, wenn dessen Einhaltung möglich ist.

Folgeconstraint 65
activateMΘnK-Constraint, der im Rahmen von entailmentMΘnK-Constraints dynamisch zur Laufzeit des Prozesses in Abhängigkeit der Ausführungshistorie eingesetzt wird.

Gegenseitiger Ausschluss 28
von Rollen. Siehe Strenge Exklusivität.

Geschäftsprozess 9
Aus elementaren Aktivitäten bestehender Vorgang, mit dem ein bestimmtes Ergebnis erreicht werden soll.

Hierarchisches RBAC 16
Eine Form von Role-Based Access Control, bei der die Rollen in hierarchische Beziehungen zueinander gesetzt werden.

Hybride Constraints 42
Constraints, die zum Teil bereits zur Modellierzeit und zum Teil erst zur Laufzeit analysiert und durchgesetzt werden können.

Inter-Instance Constraint 41
Constraint, der sich auf mehrere Prozessinstanzen bezieht.

Intra-Instance Constraint 41
Constraint, der innerhalb einer Prozessinstanz Gültigkeit besitzt und auch nur in diesem Umfang umgesetzt wird.

Kontext 11
Die Gesamtheit der Informationen aus Prozessstatus und Ausführungshistorie.

Kontrollfluss 9
Beschreibung der Abarbeitungsreihenfolge der Aktivitäten.

Laufzeit 11
siehe Ausführungszeit.

Least-Privilege-Prinzip 17
Vergabe nur der notwendigsten Berechtigungen an Benutzer.

Leere Rolle 50
Rolle ohne zugeordnete Benutzer.

MΘnK 55
Ein Modell zur Modellierung von Authorization Constraint.

MΘnK-Constraint 57
Ein abstrakter Constraint des MΘnK-Modells, dessen exakte Semantik in Form von assignMΘnK-Constraints, activateMΘnK-Constraints und entailmentMΘnK-Constraints festgelegt wird.

Modellierzeit 10
Zeitraum der Prozessmodellierung; insbesondere der Zeitraum vor der Ausführungszeit.

Operational Dynamic Separation of Duty 85
Ein DSoD-Constraint, der einem Benutzer die Ausführung von maximal $n-1$ aus n gegebenen Aktivitäten erlaubt.

Organisationseinheit 10
Teil des Organisationsmodells, in dem Mitarbeiter entsprechend ihrer Verantwortlichkeiten, Qualifikationen und Tätigkeiten gruppiert werden.

Organisationsmodell 10
Abbildung des strukturellen Aufbaus einer Organisation durch einzelne Organisationseinheiten.

Prozess 9
siehe Geschäftsprozess.

Prozessinstanz 11
Ausführbare Instanz eines Prozesses.

Prozessmodellierung 9
Identifizierung der Aktivitäten und ihrer Abhängigkeiten innerhalb eines Prozesses.

Prozessschema 10
Ergebnis der Prozessmodellierung, im Kern bestehend aus Aktivitäten, Kontrollfluss, Datenfluss und Mitarbeiterzuordnungen.

Prozessstatus 11
Alle zum jeweils aktuellen Zeitpunkt relevanten Informationen einer Prozessinstanz.

Prädikat 26
Beschreibt, in welcher Beziehung die ausführenden Benutzer der einschränkenden Aktivität und der eingeschränkten Aktivität eines Entailment Constraints stehen müssen.

Role-Based Access Control 13
System zur Zugriffskontrolle, wobei die Berechtigungsvergabe mittels Rollen stattfindet.

Rolle 13
Essentielles Konzept bei Role-Based Access Control, mittels dem die Berechtigungsvergabe an Benutzer stattfindet. Eine Rolle entspricht meist einer Organisationseinheit.

Rollen-Cardinality-Constraint 37
Cardinality Constraint der sich darauf bezieht, wie oft eine Rolle zu einem oder mehreren Benutzern zugeordnet oder durch diese aktiviert werden darf oder muss.

Rollenaktivierung 15
Das tatsächliche Annehmen einer Rolle durch einen Benutzer im Rahmen von Role-Based Access Control, wodurch der Benutzer die der Rolle zugeordneten Berechtigungen erhält.

Rollenhierarchie 17
Hierarchische Anordnung der Rollen bei Role-Based Access Control; siehe Hierarchisches RBAC.

Schwache Exklusivität 33
von Rollen. Zwei schwach exklusive Rollen dürfen niemals zum gleichen Zeitpunkt von demselben Benutzer aktiviert sein.

Separation of Duty 28
Pflichtentrennung. Ein Constraint, der unterschiedliche Benutzer für mehrere Aktivitäten fordert.

Session 15
Konzept von Role-Based Access Control, wobei in einer Session alle Rollen vermerkt sind, die der Benutzer der Session innerhalb des Gültigkeitszeitraums der Session aktiviert hat.

Sicherheitsanforderungen 12
Anforderungen an das Workflow-Management-System, die erfüllt werden müssen, damit das System als sicher gilt.

Sicherheitsrichtlinie 12
Formulierung der Sicherheitsanforderungen, die vom Workflow-Management-System durchzusetzen sind.

Simple Dynamic Separation of Duty 33
Ein Dynamischer Separation of Duty Constraint, bei dem eine Rolle durch einen Benutzer nur dann aktiviert werden kann, wenn diese nicht schwach exklusiv zu einer anderen, bereits durch diesen Benutzer aktivierten, Rolle ist.

Statische Constraints 42
Constraints, die bereits zur Modellierzeit analysiert und durchgesetzt werden können.

Statisches Separation of Duty 28
Ein Separation of Duty-Constraint, der von einer statischen Beschaffenheit ist und somit bereits zur Modellierzeit des Prozesses ausgewertet werden kann.

Statisches Separation of Duty auf Aktivitäten 52
Ein SSoD-Constraint, der die Ausführung der spezifizierten Aktivitäten durch unterschiedliche Benutzer fordert.

Statisches Separation of Duty auf Rollen 51
Ein SSoD-Constraint, der für die spezifizierten Rollen disjunkte Benutzermengen fordert.

Status 11
siehe Prozessstatus.

Strenge Exklusivität 28
von Rollen. Zwei streng exklusiven Rollen darf kein gemeinsamer Benutzer zugeordnet sein.

Template 68
Eine vordefinierte Abbildungsfunktion von Authorization Constraint auf MΘnK-Constraints.

Temporal Constraint 41
Constraint, der die zeitliche Einschränkung von Berechtigungen ermöglicht.

Typ 58
Beschreibt im Zusammenhang mit MΘnK, von welcher Art die Elemente der Mengen M (typ_M) und K (typ_K) eines MΘnK-Constraints sind.

Vier-Augen-Prinzip 35
Ein Constraint, der die zweifache Ausführung einer Aktivität durch unterschiedliche Benutzer fordert.

Workflow-Management-System 9
Computersystem zur Modellierung und Ausführung von Geschäftsprozessen.

Abkürzungsverzeichnis

1sSSSoD	1-step Strict Static Separation of Duty
4Eyes	Vier-Augen-Prinzip
BoD	Binding of Duty
CC	Cardinality Constraint
DAC	Discretionary Access Control
DSoD	Dynamisches Separation of Duty
MAC	Mandatory Access Control
OpDSoD	Operational Dynamic Separation of Duty
PA	Berechtigungszuordnung (Permission Assignment)
RBAC	Role-Based Access Control
RoleCC	Rollen-Cardinality-Constraint
RH	Rollenhierarchie
SDSoD	Simple Dynamic Separation of Duty
SoD	Separation of Duty
SSoD	Statisches Separation of Duty
RoleSSoD	Statisches Separation of Duty auf Rollen
TaskCC	Aktivitäten-Cardinality-Constraint
TaskDSoD	Dynamisches Separation of Duty auf Aktivitäten
TaskSSoD	Statisches Separation of Duty auf Aktivitäten
UA	Benutzerzuordnung (User Assignment)
WfMS	Workflow-Management-System

Abbildungsverzeichnis

Tabellenverzeichnis

Teil IV

Anhang

A Paarweise Validierung von assignMΘnK-Constraints

Im Folgenden sind jeweils zwei assignMΘnK-Constraints

- $C_1 : assignMonk(M_1, \Theta_1, n_1, K_1)$
- $C_2 : assignMonk(M_2, \Theta_2, n_2, K_2)$

mit den zusätzlichen Voraussetzungen

- $type(M_1) = type(M_2)$ und
- $type(K_1) = type(K_2)$

gegeben. Außerdem wird vorausgesetzt, dass C_1 und C_2 bei individueller Betrachtung erfüllt werden können. Unter den folgenden Punkten i) bis xv) werden die beiden assignMΘnK-Constraints C_1 und C_2 auf Widersprüche hin untersucht. Hierzu werden alle Kombinationsmöglichkeiten für die Werte der beiden assignMΘnK-Constraints betrachtet.

i) $\boldsymbol{M_1 = M_2 = M, \quad \Theta_1 \in \{=\}, \Theta_2 \in \{=\}, \quad n_1 = n_2 = n, \quad K_1 \cap K_2 \neq \emptyset}$

- $C_1 : assignMonk(M, =, n, K_1)$
- $C_2 : assignMonk(M, =, n, K_2)$

a) $K_1 = K_2 = K$

Kein Widerspruch.

C_1 und C_2 sind exakt gleich.

b) $K_1 \subset K_2$

Kein Widerspruch.

Jedem $m \in M$ dürfen nur Elemente aus $K_1 \cap K_2 = K_1$ zugeordnet werden. Denn würde einem $m \in M$ ein Element aus $K_2 \backslash K_1$ zugeordnet werden, so müsste diesem m auch ein Element aus $K_1 \backslash K_2$ zugeordnet werden. Dies ist jedoch nicht möglich, da $K_1 \backslash K_2 = \emptyset$. C_2 ist also obsolet bzw. alle Elemente aus $K_2 \backslash K_1$ können ohnehin nicht zugeordnet werden, da sonst C_1 verletzt würde. Es besteht jedoch kein Widerspruch, da C_1 erfüllt werden kann und somit automatisch auch C_2 erfüllt ist.

c) $K_1 \supset K_2$

Kein Widerspruch.

Vorgehen analog zu b).

d) $K_1 \not\subset K_2 \wedge K_1 \not\supset K_2$

Kein Widerspruch.

Jedem $m \in M$ werden n_m viele Elemente aus $K_1 \cap K_2$ zugeordnet ($n_m \leq n$). Zusätzlich müssen jedem $m \in M$ jeweils $n - n_m$ viele Elemente aus $K_2 \backslash K_1$ (erfüllt C_2) sowie $K_1 \backslash K_2$ (erfüllt C_1) zugeordnet werden. Dies ist immer möglich, da C_1 und C_2 erfüllbar sind.

ii) $\boldsymbol{M_1 = M_2 = M, \quad \Theta_1 \in \{=\}, \Theta_2 \in \{=\}, \quad n_1 > n_2, \quad K_1 \cap K_2 \neq \emptyset}$

- $C_1 : assignMonk(M, =, n_1, K_1)$
- $C_2 : assignMonk(M, =, n_2, K_2)$

a) $K_1 = K_2 = K$

C_1 und C_2 widersprechen sich immer.

Jedem $m \in M$ müssen gemäß C_1 genau n_1 viele Elemente aus K und gemäß C_2 genau n_2 viele Elemente aus K zugeordnet werden. Dies ist jedoch niemals möglich, da $n_1 \neq n_2$.

b) $K_1 \subset K_2$

C_1 und C_2 widersprechen sich immer.

Jedem $m \in M$ werden gemäß C_1 genau n_1 viele Elemente aus K_1 zugeordnet. Da $K_1 \subset K_2$ gilt, werden diese Elemente immer auch aus K_2 zugeordnet. Damit wird jedoch C_2 verletzt, denn es werden n_1 viele Elemente aus K_2 zugeordnet. Dies sind jedoch mehr als erlaubt, da $n_1 > n_2$.

c) $K_1 \supset K_2$

Widerspruch, falls $|K_1 \backslash K_2| < (n_1 - n_2) \cdot |M|$.

Jedem $m \in M$ werden gemäß C_2 exakt n_2 viele Elemente aus K_2 zugeordnet. Aus K_2 werden also $|M| \cdot n_2$ viele Elemente zugeordnet. Da $K_1 \supset K_2$ gilt, werden alle diese Elemente auch aus K_1 zugeordnet. Um auch C_1 zu erfüllen, müssen jedem m zusätzlich $n_1 - n_2$ viele Elemente zugeordnet werden die in K_1, aber nicht in K_2 sind. Es muss also gelten $|K_1 \backslash K_2| \geq (n_1 - n_2) \cdot |M|$.

d) $K_1 \not\subset K_2 \wedge K_1 \not\supset K_2$

Widerspruch, falls $|K_1 \backslash K_2| < (n_1 - n_2) \cdot |M|$.

Jedem $m \in M$ werden gemäß C_2 genau n_2 viele Elemente aus K_2 zugeordnet. In K_1 müssen dann $(n_1 - n_2) \cdot |M|$ viele Elemente enthalten sein die nicht in K_2 enthalten sind. Denn andernfalls könnte C_1 nicht erfüllt werden, da jedem $m \in M$ genau $(n_1 - n_2)$ viele Elemente zugeordnet werden müssen die zwar in K_1, nicht aber in K_2 enthalten sind. Es muss daher gelten: $|K_1 \backslash K_2| \geq (n_1 - n_2) \cdot |M|$.

iii) $\boldsymbol{M_1 = M_2 = M, \quad \Theta_1 \in \{\geq\}, \Theta_2 \in \{\geq\}, \quad n_1 = n_2 = n, \quad K_1 \cap K_2 \neq \emptyset}$

- $C_1 : assignMonk(M, \geq, n, K_1)$
- $C_2 : assignMonk(M, \geq, n, K_2)$

a) $K_1 = K_2 = K$

Kein Widerspruch.

C_1 und C_2 sind exakt gleich.

b) $K_1 \subset K_2$

Kein Widerspruch.

Jedem $m \in M$ werden mindestens n Elemente aus K_1 zugeordnet. Diese Elemente werden immer auch aus K_2 zugeordnet. C_2 ist also immer erfüllt wenn C_1 erfüllt ist.

c) $K_1 \supset K_2$

Kein Widerspruch.

Vorgehen analog zu b).

d) $K_1 \not\subset K_2 \wedge K_1 \not\supset K_2$

Kein Widerspruch.

Jedem $m \in M$ werden n_m Elemente aus $K_1 \cap K_2$ zugeordnet ($n_m \leq n$). Zusätzlich müssen jedem $m \in M$ jeweils mindestens $n - n_m$ Elemente aus $K_2 \backslash K_1$ (erfüllt C_2) sowie $K_1 \backslash K_2$ (erfüllt C_1) zugeordnet werden. Dies ist immer möglich, da C_1 und C_2 erfüllbar sind.

iv) $\boldsymbol{M_1 = M_2 = M, \quad \Theta_1 \in \{\geq\}, \Theta_2 \in \{\geq\}, \quad n_1 > n_2, \quad K_1 \cap K_2 \neq \emptyset}$

- $C_1 : assignMonk(M, \geq, n_1, K_1)$
- $C_2 : assignMonk(M, \geq, n_2, K_2)$

a) $K_1 = K_2 = K$

Kein Widerspruch.

Jedem $m \in M$ werden mindestens n_1 viele Elemente aus K zugeordnet. Diese Zuordnungen erfüllen neben C_1 immer auch C_2, da $n_1 > n_2$ gilt und jedem m mindestens n_2 viele Elemente aus K zugeordnet werden müsssen.

b) $K_1 \subset K_2$

Kein Widerspruch.

Jedem $m \in M$ werden mindestens n_1 viele Elemente aus K_1 zugeordnet. Diese Zuordnungen erfüllen neben C_1 immer auch C_2, da $n_1 > n_2$ und $K_1 \subset K_2$ gilt und da jedem m mindestens n_2 viele Elemente aus K_2 zugeordnet werden müsssen.

c) $K_1 \supset K_2$

Kein Widerspruch.

Jedem $m \in M$ werden n_2 viele Elemente aus K_2 zugeordnet. Diese sind immer auch in K_1 enthalten. Zur Erfüllung von C_1 müssen jedem m zusätzlich $n_1 - n_2$ viele Elemente aus K_1 zugeordnet werden. Da auch weitere Elemente aus K_2 zugeordnet werden dürfen, ist es unerheblich ob diese Elemente auch in K_2 enthalten sind.

d) $K_1 \not\subset K_2 \wedge K_1 \not\supset K_2$

Kein Widerspruch.

Jedem $m \in M$ werden mindestens n_1 viele Elemente aus K_1 und mindestens n_2 viele Elemente aus K_2 zugeordnet. Dabei ist es unerheblich, ob diese Elemente aus $K_1 \cap K_2$, $K_1 \backslash K_2$ oder $K_2 \backslash K_1$ sind.

v) $\boldsymbol{M_1 = M_2 = M, \quad \Theta_1 \in \{\leq\}, \Theta_2 \in \{\leq\}, \quad n_1 = n_2 = n, \quad K_1 \cap K_2 \neq \emptyset}$

- $C_1 : assignMonk(M, \leq, n, K_1)$
- $C_2 : assignMonk(M, \leq, n, K_2)$

a) $K_1 = K_2 = K$

Kein Widerspruch.

C_1 und C_2 sind exakt gleich.

b) $K_1 \subset K_2$

Kein Widerspruch.

Jedem $m \in M$ werden höchstens n_1 viele Elemente aus K_1 und höchstens n_2 viele Elemente aus K_2 zugeordnet. Die gleichzeitige Erfüllung von C_1 und C_2 ist bei Zuordnung keiner Elemente immer möglich.

c) $K_1 \supset K_2$

Kein Widerspruch.

Vorgehen analog zu b).

d) $K_1 \not\subset K_2 \wedge K_1 \not\supset K_2$

Kein Widerspruch.

Jedem $m \in M$ werden n_m ($n_m \leq n$) viele Elemente aus $K_1 \cap K_2$ zugeordnet. Jedem $m \in M$ dürfen zusätzlich jeweils höchstens $n - n_m$ viele Elemente aus $K_1 \backslash K_2$ und $K_2 \backslash K_1$ zugeordnet werden. Die gleichzeitige Erfüllung von C_1 und C_2 ist bei Zuordnung keiner Elemente immer möglich.

vi) $\boldsymbol{M_1 = M_2 = M, \quad \Theta_1 \in \{\leq\}, \Theta_2 \in \{\leq\}, \quad n_1 > n_2, \quad K_1 \cap K_2 \neq \emptyset}$

- $C_1 : assignMonk(M, \leq, n_1, K_1)$
- $C_2 : assignMonk(M, \leq, n_2, K_2)$

a) $K_1 = K_2 = K$

Kein Widerspruch.

Jedem $m \in M$ dürfen höchstens n_2 viele Elemente aus K zugeordnet werden. Damit ist immer auch C_1 erfüllt.

b) $K_1 \subset K_2$

Kein Widerspruch.

Jedem $m \in M$ dürfen gemäß C_1 höchstens n_1 viele Elemente aus K_1 zugeordnet werden. Alle diese Elemente sind jedoch auch in K_2 enthalten, so dass diese immer auch aus K_2 zugeordnet werden. Mit C_1 ist also immer auch C_2 erfüllt.

c) $K_1 \supset K_2$

Kein Widerspruch.

Jedem $m \in M$ werden höchstens n_2 viele Elemente aus K_2 zugeordnet. Jedem m dürfen weitere Elemente aus K_1 zugeordnet werden. Eine Mindestanforderung an die Größe von $K_1 \backslash K_2$ besteht jedoch nicht, da diese Zuordnungen optional sind.

d) $K_1 \not\subset K_2 \wedge K_1 \not\supset K_2$

Kein Widerspruch.

Jedem $m \in M$ werden höchstens n_2 viele Elemente aus K_2 zugeordnet. Jedem m dürfen weitere Elemente aus $K_1 \backslash K_2$ zugeordnet werden. Eine Mindestanforderung an die Größe von $K_1 \backslash K_2$ besteht jedoch nicht, da diese Zuordnungen optional sind.

vii) $\boldsymbol{M_1 = M_2 = M, \quad \Theta_1 \in \{=\}, \Theta_2 \in \{\geq\}, \quad n_1 = n_2 = n, \quad K_1 \cap K_2 \neq \emptyset}$

- $C_1 : assignMonk(M, =, n, K_1)$
- $C_2 : assignMonk(M, \geq, n, K_2)$

a) $K_1 = K_2 = K$

Kein Widerspruch.

Jedem $m \in M$ werden gemäß C_1 exakt n viele Elemente aus K_1 zugeordnet. Damit ist auch C_2 erfüllt. Zuordnen weiterer Elemente aus K_2 (gemäß C_2) ist nicht möglich, da sonst C_1 verletzt würde.

b) $K_1 \subset K_2$

Kein Widerspruch.

Jedem $m \in M$ werden gemäß C_1 exakt n viele Elemente aus K_1 zugeordnet. Da $K_1 \subset K_2$ gilt, werden diese Elemente immer auch aus K_2 zugeordnet. C_2 erlaubt das Zuordnen weiterer Elemente aus $K_2 \backslash K_1$. Eine Mindestanforderung an die Größe von $K_2 \backslash K_1$ besteht jedoch nicht, da diese Zuordnungen optional sind.

c) $K_1 \supset K_2$

Kein Widerspruch.

Jedem $m \in M$ werden zunächst n viele Elemente aus K_2 zugeordnet. Diese sind jedoch auch in K_1 enthalten und somit ist damit auch C_1 erfüllt. Das Zuordnen weiterer Elemente aus K_2 ist nicht zulässig. Die gleichzeitige Erfüllung von C_1 und C_2 ist also durch Zuordnung von exakt n_2 vielen Elementen aus K_2 an jedes $m \in M$ möglich.

d) $K_1 \not\subset K_2 \wedge K_1 \not\supset K_2$

Kein Widerspruch.

Jedem $m \in M$ werden n_m ($n_m \leq n$) viele Elemente aus $K_1 \cap K_2$ zugeordnet. Jedem m müssen also exakt $n - n_m$ viele weitere Elemente aus $K_1 \backslash K_2$ und mindestens $n - n_m$ viele weitere Elemente aus $K_2 \backslash K_1$ zugeordnet werden.

viii) $\boldsymbol{M_1 = M_2 = M, \quad \Theta_1 \in \{=\}, \Theta_2 \in \{\geq\}, \quad n_1 > n_2, \quad K_1 \cap K_2 \neq \emptyset}$

- $C_1 : assignMonk(M, =, n_1, K_1)$
- $C_2 : assignMonk(M, \geq, n_2, K_2)$

a) $K_1 = K_2 = K$

Kein Widerspruch.

Jedem $m \in M$ werden gemäß C_1 genau n_1 viele Elemente aus K zugeordnet. Mit Erfüllung von C_1 ist damit immer auch C_2 erfüllt.

b) $K_1 \subset K_2$

Kein Widerspruch.

Jedem $m \in M$ werden genau n_1 viele Elemente aus K_1 zugeordnet. Diese Elemente sind immer auch in K_2 enthalten. Da $n_1 > n_2$ ist damit auch C_2 erfüllt. Das Zuordnen weiterer Elemente gemäß C_2 ist nur aus der Menge $K_2 \backslash K_1$ erlaubt. Eine Mindestanforderung an die Größe von $K_2 \backslash K_1$ besteht jedoch nicht, da diese Zuordnungen optional sind.

c) $K_1 \supset K_2$

Kein Widerspruch.

Jedem $m \in M$ werden zunächst n_2 viele Elemente aus K_2 zugeordnet. Diese sind immer auch in K_1 enthalten. Um zusätzlich C_1 zu erfüllen, müssen jedem $m \in M$ weitere $n_1 - n_2$ viele Elemente aus K_1 zugeordnet werden. Diese dürfen auch in K_2 enthalten sein ohne C_2 zu verletzen, so dass kein Widerspruch besteht.

d) $K_1 \not\subset K_2 \wedge K_1 \not\supset K_2$

Kein Widerspruch.

Jedem $m \in M$ werden zunächst mindestens n_m ($n_m \leq n$) viele Elemente aus $K_1 \cap K_2$ zugeordnet. Jedem m werden dann zunächst genau $n_2 - n_m$ viele weitere Elemente aus $K_2 \backslash K_1$ zugeordnet, so dass C_2 erfüllt ist. Zusätzlich werden jedem m exakt $n_1 - n_m$ viele Elemente aus $K_1 \backslash K_2$ zugeordnet, wodurch C_1 erfüllt wird. Das Zuordnen weiterer Elemente aus $K_2 \backslash K_1$ ist zulässig.

ix) $\boldsymbol{M_1 = M_2 = M, \quad \Theta_1 \in \{=\}, \Theta_2 \in \{\geq\}, \quad n_1 < n_2, \quad K_1 \cap K_2 \neq \emptyset}$

- $C_1 : assignMonk(M, =, n_1, K_1)$
- $C_2 : assignMonk(M, \geq, n_2, K_2)$

a) $K_1 = K_2 = K$

Widerspruch immer.

Da $n_1 < n_2$ gilt, ist es nicht möglich gleichzeitig genau n_1 und mindestens n_2 Elemente aus K zuzuordnen. C_1 und C_2 sind also gemeinsam nie erfüllbar.

b) $K_1 \subset K_2$

Widerspruch, falls $\boldsymbol{|K_2 \backslash K_1| < (n_2 - n_1) \cdot |M|}$.

Jedem $m \in M$ werden gemäß C_1 exakt n_1 viele Elemente aus K_1 zugeordnet. Um auch C_2 zu erfüllen, müssen jedem m zusätzlich $n_2 - n_1$ viele Elemente aus K_2 zugeordnet werden. Diese Elemente dürfen jedoch nicht auch in K_1 enthalten sein, da sonst C_1 verletzt würde. Damit muss gelten: $|K_2 \backslash K_1| \geq (n_2 - n_1) \cdot |M|$.

c) $K_1 \supset K_2$

Widerspruch immer.

Jedem $m \in M$ werden gemäß C_2 mindestens n_2 viele Elemente aus K_2 zugeordnet. Da alle diese Elemente auch in K_1 enthalten sind, wird damit jedoch C_1 verletzt. Denn C_1 erlaubt nur die Zuordnung von exakt n_1 vielen Elementen an jedes m, jedoch wurden durch C_2 bereits n_2 viele Elemente zugeordnet (und $n_2 > n_1$).

d) $K_1 \not\subset K_2 \wedge K_1 \not\supset K_2$

Widerspruch, falls $\boldsymbol{|K_2 \backslash K_1| < (n_2 - n_1) \cdot |M|}$.

In K_2 müssen mindestens $(n_2 - n_1) \cdot |M|$ viele Elemente sein die nicht in K_1 enthalten sind, damit beide Constraints erfüllt werden können. Es muss also gelten $|K_2 \backslash K_1| \geq (n_2 - n_1) \cdot |M|$.

x) $\boldsymbol{M_1 = M_2 = M, \quad \Theta_1 \in \{=\}, \Theta_2 \in \{\leq\}, \quad n_1 = n_2 = n, \quad K_1 \cap K_2 \neq \emptyset}$

- $C_1 : assignMonk(M, =, n, K_1)$
- $C_2 : assignMonk(M, \leq, n, K_2)$

a) $K_1 = K_2 = K$

Kein Widerspruch.

Jedem $m \in M$ werden gemäß C_1 genau n Elemente aus K zugeordnet. Damit ist immer auch C_2 erfüllt.

b) $K_1 \subset K_2$

Kein Widerspruch.

Jedem $m \in M$ werden gemäß C_1 genau n Elemente aus K_1 zugeordnet. Da C_2 die Zuordnung von höchstens n Elementen aus einer Obermenge erlaubt, ist C_2 automatisch mit C_1 erfüllt.

c) $K_1 \supset K_2$

Kein Widerspruch.

Jedem $m \in M$ werden gemäß C_1 genau n Elemente aus K_1 zugeordnet. C_2 erlaubt die Zuordnung von höchstens n Elementen aus einer Submenge und ist damit automatisch mit C_1 erfüllt.

d) $K_1 \not\subset K_2 \wedge K_1 \not\supset K_2$

Kein Widerspruch.

Jedem $m \in M$ werden gemäß C_1 exakt n viele Elemente aus K_1 zugeordnet. Für jedes m seien dabei n_m viele Elemente aus $K_1 \cap K_2$. Jedem m dürfen dann höchstens $n - n_m$ viele weitere Elemente aus $K_2 \backslash K_1$ zugeordnet werden. C_2 ist jedoch bereits automatisch mit C_1 erfüllt.

xi) $\boldsymbol{M_1 = M_2 = M, \quad \Theta_1 \in \{=\}, \Theta_2 \in \{\leq\}, \quad n_1 > n_2, \quad K_1 \cap K_2 \neq \emptyset}$

- $C_1 : assignMonk(M, =, n_1, K_1)$
- $C_2 : assignMonk(M, \leq, n_2, K_2)$

a) $K_1 = K_2 = K$

Widerspruch immer.

Jedem $m \in M$ werden gemäß C_1 genau n_1 viele und gemäß C_2 höchstens n_2 viele Elemente aus K zugeordnet. Da aber $n_1 > n_2$ gilt, widersprechen sich beide Constraints immer.

b) $K_1 \subset K_2$

Widerspruch immer.

Jedem $m \in M$ werden gemäß C_1 exakt n_1 viele Elemente aus K_1 zugeordnet. Diese werden immer auch aus K_2 zugeordnet, jedoch sind dort höchstens n_2 viele Zuordnungen erlaubt. Da $n_1 > n_2$ gilt, widersprechen sich die Constraints immer.

c) $K_1 \supset K_2$

Widerspruch, falls $\boldsymbol{|K_1 \backslash K_2| < (n_1 - n_2) \cdot |M|}$.

Jedem $m \in M$ werden gemäß C_2 höchstens n_2 viele Elemente aus K_2 zugeordnet. Um auch C_1 zu erfüllen, müssen jedem m mindestens $n_1 - n_2$ viele weitere Elemente zugeordnet werden die in K_1, nicht jedoch in K_2 enthalten sind. Damit muss gelten: $|K_1 \backslash K_2| \geq (n_1 - n_2) \cdot |M|$.

d) $K_1 \not\subset K_2 \wedge K_1 \not\supset K_2$

Widerspruch, falls $\boldsymbol{|K_1 \backslash K_2| < (n_1 - n_2) \cdot |M|}$.

Jedem $m \in M$ werden gemäß C_2 höchstens n_2 viele Elemente aus K_2 zugeordnet. Um auch C_1 zu erfüllen, müssen jedem m zusätzlich mindestens $n_1 - n_2$ viele Elemente zugeordnet werden die in K_1, nicht jedoch in K_2 enthalten sind. Damit muss gelten: $|K_1 \backslash K_2| \geq (n_1 - n_2) \cdot |M|$.

xii) $\boldsymbol{M_1 = M_2 = M, \quad \Theta_1 \in \{=\}, \Theta_2 \in \{\leq\}, \quad n_1 < n_2, \quad K_1 \cap K_2 \neq \emptyset}$

- $C_1 : assignMonk(M, =, n_1, K_1)$
- $C_2 : assignMonk(M, \leq, n_2, K_2)$

a) $K_1 = K_2 = K$

Kein Widerspruch.

Jedem $m \in M$ werden gemäß C_1 exakt n_1 viele Elemente aus K zugeordnet. Die Zuordnung zusätzlicher Elemente gemäß C_2 würde jedoch C_1 verletzen. C_2 ist jedoch immer erfüllt, wenn C_1 erfüllt ist.

b) $K_1 \subset K_2$

Kein Widerspruch.

Jedem $m \in M$ werden gemäß C_1 exakt n_1 viele Elemente aus K_1 zugeordnet. Das Zuordnen weiterer Elemente (bis maximal n_2) muss dann aus $K_2 \backslash K_1$ erfolgen, da sonst C_1 verletzt würde. Eine Mindestanforderung an die Größe von $K_2 \backslash K_1$ besteht jedoch nicht, da diese Zuordnungen optional sind.

c) $K_1 \supset K_2$

Kein Widerspruch.

Jedem $m \in M$ werden höchstens n_2 viele Elemente aus K_2 zugeordnet. Diese werden immer auch aus K_1 zugeordnet. Um also auch C_1 zu erfüllen, dürfen auch aus K_2 höchstens n_1 viele Elemente zugeordnet werden (da ja $n_1 < n_2$ gilt). Hiermit ist jedoch auch C_2 weiterhin erfüllt.

d) $K_1 \not\subset K_2 \wedge K_1 \not\supset K_2$

Kein Widerspruch.

Jedem $m \in M$ werden genau n_1 viele Elemente aus K_1 zugeordnet. C_2 erlaubt die Zuordnung weiterer Elemente aus $K_2 \backslash K_1$ (bis maximal n_2). Eine Mindestanforderung an die Größe von $K_2 \backslash K_1$ besteht jedoch nicht, da diese Zuordnungen optional sind.

xiii) $\boldsymbol{M_1 = M_2 = M, \quad \Theta_1 \in \{\leq\}, \Theta_2 \in \{\geq\}, \quad n_1 = n_2 = n, \quad K_1 \cap K_2 \neq \emptyset}$

- $C_1 : assignMonk(M, \leq, n, K_1)$
- $C_2 : assignMonk(M, \geq, n, K_2)$

a) $K_1 = K_2 = K$

Kein Widerspruch.

Jedem $m \in M$ werden mindestens n und höchstens n Elemente aus $K_1 = K_2$ zugeordnet. Damit werden C_1 und C_2 genau dann erfüllt, wenn jedem $m \in M$ exakt n Elemente aus K zugeordnet werden.

b) $K_1 \subset K_2$

Kein Widerspruch.

Jedem $m \in M$ werden zunächst n viele Elemente aus K_2 zugeordnet. Damit sind C_1 und C_2 erfüllt. Die Zuordnung weiterer Elemente aus $K_2 \backslash K_1$ ist möglich.

c) $K_1 \supset K_2$

Kein Widerspruch.

Jedem $m \in M$ werden mindestens n viele Elemente aus K_2 zugeordnet. Da $K_1 \supset K_2$ gilt, werden diese immer auch aus K_1 zugeordnet. C_1 erlaubt jedoch nur das Zuordnen von höchstens n vielen Elemente aus K_1. Damit werden C_1 und C_2 genau dann erfüllt, wenn jedem $m \in M$ exakt n Elemente aus K zugeordnet werden.

d) $K_1 \not\subset K_2 \wedge K_1 \not\supset K_2$

Kein Widerspruch.

Jedem $m \in M$ werden gemäß C_2 zunächst exakt n Elemente aus K_2 zugeordnet. Da jedoch wegen C_1 nicht mehr als n Elemente aus K_1 zugeordnet werden dürfen, dürfen maximal n Elemente aus $K_1 \cap K_2$ zugeordnet werden. Die Zuordnung weiterer Elemente aus $K_2 \backslash K_1$ ist möglich.

xiv) $\boldsymbol{M_1 = M_2 = M, \quad \Theta_1 \in \{\leq\}, \Theta_2 \in \{\geq\}, \quad n_1 > n_2, \quad K_1 \cap K_2 \neq \emptyset}$

- $C_1 : assignMonk(M, \leq, n_1, K_1)$
- $C_2 : assignMonk(M, \geq, n_2, K_2)$

a) $K_1 = K_2 = K$

Kein Widerspruch.

Jedem $m \in M$ werden gemäß C_2 zunächst n_2 viele Elemente aus K zugeordnet. Damit ist C_2 und auch C_1 erfüllt. Weitere Elemente aus K bis maximal n_1 dürfen zugeordnet werden.

b) $K_1 \subset K_2$

Kein Widerspruch.

Jedem $m \in M$ werden gemäß C_2 zunächst n_2 viele Elemente aus K_2 zugeordnet. Hiermit ist C_1 und C_2 erfüllt.

c) $K_1 \supset K_2$

Kein Widerspruch.

Jedem $m \in M$ werden gemäß C_2 zunächst n_2 viele Elemente aus K_2 zugeordnet. Hiermit ist C_1 und C_2 erfüllt. Das Zuordnen weiterer $n_1 - n_2$ Elemente aus K_1 ist möglich.

d) $K_1 \not\subset K_2 \wedge K_1 \not\supset K_2$

Kein Widerspruch.

Jedem $m \in M$ werden mindestens n_2 viele Elemente aus K_2 zugeordnet. Wegen C_1 dürfen jedoch nicht mehr als n_1 Elemente aus K_1 zugeordnet werden. Es dürfen also maximal n_1 viele Elemente aus $K_1 \cap K_2$ zugeordnet werden.

xv) $\boldsymbol{M_1 = M_2 = M, \quad \Theta_1 \in \{\leq\}, \Theta_2 \in \{\geq\}, \quad n_1 < n_2, \quad K_1 \cap K_2 \neq \emptyset}$

- $C_1 : assignMonk(M, \leq, n_1, K_1)$
- $C_2 : assignMonk(M, \geq, n_2, K_2)$

a) $K_1 = K_2 = K$

Widerspruch immer.

Jedem $m \in M$ werden laut C_2 mindestens n_2 und laut C_1 höchstens n_1 viele Elemente aus K zugeordnet. Da jedoch $n_1 < n_2$ ist, können C_1 und C_2 nie gleichzeitig erfüllt werden.

b) $K_1 \subset K_2$

Widerspruch, falls $|K_2 \backslash K_1| < (n_2 - n_1) \cdot |M|$.

Jedem $m \in M$ werden höchstens n_1 Elemente aus K_1 zugeordnet um C_1 zu erfüllen. Alle diese zugeordneten Elemente sind jedoch immer auch in K_2 enthalten. Um auch C_2 zu erfüllen, müssen jedem m mindestens $n_2 - n_1$ viele Elemente aus $K_2 \backslash K_1$ zugeordnet werden. Damit muss gelten $|K_2 \backslash K_1| \geq (n_2 - n_1) \cdot |M|$.

c) $K_1 \supset K_2$

Widerspruch immer.

Jedem $m \in M$ werden gemäß C_2 mindestens n_2 viele Elemente aus K_2 zugeordnet. Diese sind jedoch immer auch in K_1 enthalten. C_1 erlaubt jedoch die Zuordnung von höchstens n_1 vielen Elemente aus K_1. Da aber $n_1 < n_2$ gilt, widersprechen sich beide Constraints immer.

d) $K_1 \not\subset K_2 \wedge K_1 \not\supset K_2$

Widerspruch, falls $|K_2 \backslash K_1| < (n_2 - n_1) \cdot |M|$.

Jedem $m \in M$ werden mindestens n_2 viele Elemente aus K_2 zugeordnet. Gleichzeitig dürfen jedem m höchstens n_1 viele Elemente aus K_1 zugeordnet werden. Da $n_1 < n_2$ gilt, muss K_2 mindestens $(n_2 - n_1) \cdot |M|$ viele Elemente enthalten die nicht in K_1 enthalten sind, es muss also gelten $|K_2 \backslash K_1| \geq (n_2 - n_1) \cdot |M|$.

B Inhalt der beiliegenden DVD

Auf der beiliegenden DVD befinden sich die drei Verzeichnisse

- Ausarbeitung
- Implementierung
- Literatur

Ausarbeitung

In diesem Verzeichnis befindet sich die Ausarbeitung dieser Arbeit als PDF-Datei

- Authorization Constraints in Workflow Management Systemen.pdf

Implementierung

Dieses Verzeichnis enthält die drei Unterverzeichnisse

- doc
- src
- vm

doc

Dieses Verzeichnis enthält die mittels Javadoc[1] generierte Dokumentation im HTML-Format. Dokumentiert werden alle für die Implementierung relevanten Java-Packages und -Klassen. Besonders hervorzuheben sind die Packages

- constraints
- de.aristaflow.adept2.base.processmodel.rightmanagement.constraints

[1] http://java.sun.com/j2se/javadoc/

und die jeweils enthaltenen Klassen. Diese entsprechen dem im Rahmen dieser Arbeit entstandenen Programmcode.

src

Dieses Verzeichnis enthält den im Rahmen dieser Arbeit entstandenen Programmcode in den beiden Packages

- constraints
- de.aristaflow.adept2.base.processmodel.rightmanagement.constraints

Die Java-Quelldateien sind in entsprechenden Unterverzeichnissen abgelegt.

vm

Dieses Verzeichnis enthält die mit VirtualBox erstellte virtuelle Maschine in Form der drei Dateien

- arms.mf
- arms.ovf
- arms.vmdk

In dieser virtuellen Maschine können die im Rahmen dieser Arbeit implementierten Konzepte gesichtet, erprobt und weiterentwickelt werden. Instruktionen zur Verwendung der virtuellen Maschine finden sich in Anhang C.

Literatur

In diesem Verzeichnis befindet sich die BibTeX-Datei

- Bibliography.bib

Diese enthält alle relevanten Informationen zu der in dieser Arbeit verwendeten Literatur.

Im Unterverzeichnis

- Artikel

befinden sich alle mir in elektronischer Form vorliegenden Artikel der BibTeX-Datei im PDF-Format.

C Start des „graphicalrmsSimulator“

Dieses Kapitel beschreibt die Verwendung der auf DVD beiliegenden virtuellen Maschine (siehe Anhang B) mit dem Ziel die Anwendung „graphicalrmsSimulator“ zu starten. Die virtuelle Maschine wurde mit VirtualBox[1] (Version 3.1.6_OSE) erstellt und befindet sich im OVF-Format (Open Virtualization Format[2]) im Verzeichnis

- Implementierung/vm

Import und Start der virtuellen Maschine in VirtualBox

Zur Verwendung der virtuellen Maschine muss zunächst VirtualBox installiert[3] und gestartet werden. Hierzu ist ein Host-System mit mindestens 2GB Arbeitsspeicher empfehlenswert. Der Import der virtuellen Maschine geschieht über das Menü (File → Import Appliance...). Hier ist die Datei

- arms.ovf

aus obigem Verzeichnis zu importieren. Weiterführende Informationen zum Import virtueller Maschinen in VirtualBox finden sich im entsprechenden Handbuch[4]. Der Start der importierten virtuellen Maschine erfolgt durch Klick auf die Schaltfläche „Start“.

Starten der Anwendung „graphicalrmsSimulator“

Die Implementierungen wurden im Rahmen der Anwendung „graphicalrmsSimulator“ vorgenommen. Um diese zu starten, muss zunächst „Eclipse“ über die entsprechende Verknüpfung auf dem Desktop der virtuellen Maschine gestartet werden. Die Anwendung „graphicalrmsSimulator“ wird dann über das Menü (Run → Run History → graphicalrmsSimu-

[1] http://www.virtualbox.org/
[2] http://www.vmware.com/appliances/getting-started/learn/ovf.html
[3] http://www.virtualbox.org/manual/ch01.html#id2510843
[4] http://www.virtualbox.org/manual/ch01.html#ovf

lator.application) gestartet (Abb. C.1). Im sich öffnenden Fenster „graphicalARMS config“ (Abb. C.2) muss dann heruntergescrollt und auf „Start“ geklickt werden. Anschließend öffnet sich das Hauptfenster „graphicalARMS Simulator“ (Abb. C.3).

In diesem Hauptfenster muss ein Template (in der Terminologie dieser Arbeit ein Prozessschema) eines Prozesses ausgewählt werden („Loaded Templates“). Die virtuelle Maschine wurde so konfiguriert, dass hier ein Template des Warenbestellungsprozesses aus dieser Arbeit geladen wurde. Wird dieses Template ausgewählt, so werden fünf zuvor generierte Instanzen dieses Templates angezeigt („Generated Instances“). Nach Auswahl einer dieser Instanzen werden alle Aktivitäten der gewählten Prozessinstanz mit ihrem Status angezeigt („Nodes“) (Abb. C.4). Mit Hilfe der Schaltfläche „Change Constraints“ ist dann die Verwaltung der Constraints dieser Prozessinstanz möglich.

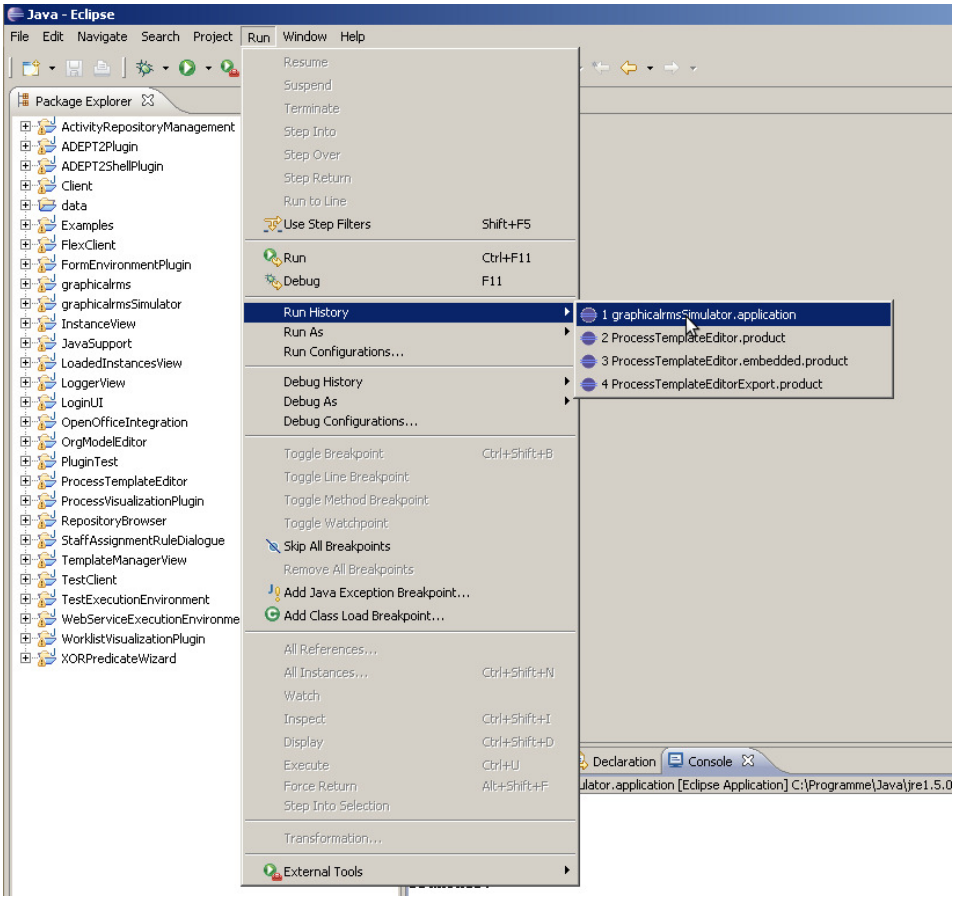


Abb. C.1: Starten von „graphicalrmsSimulator“ aus „Eclipse“.

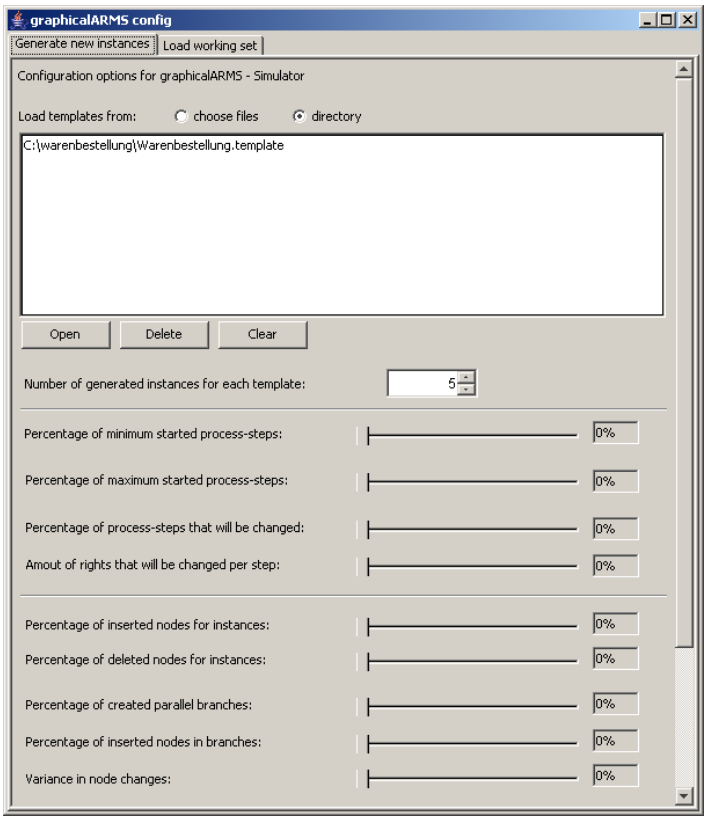


Abb. C.2: „graphicalARMS config“ Fenster.

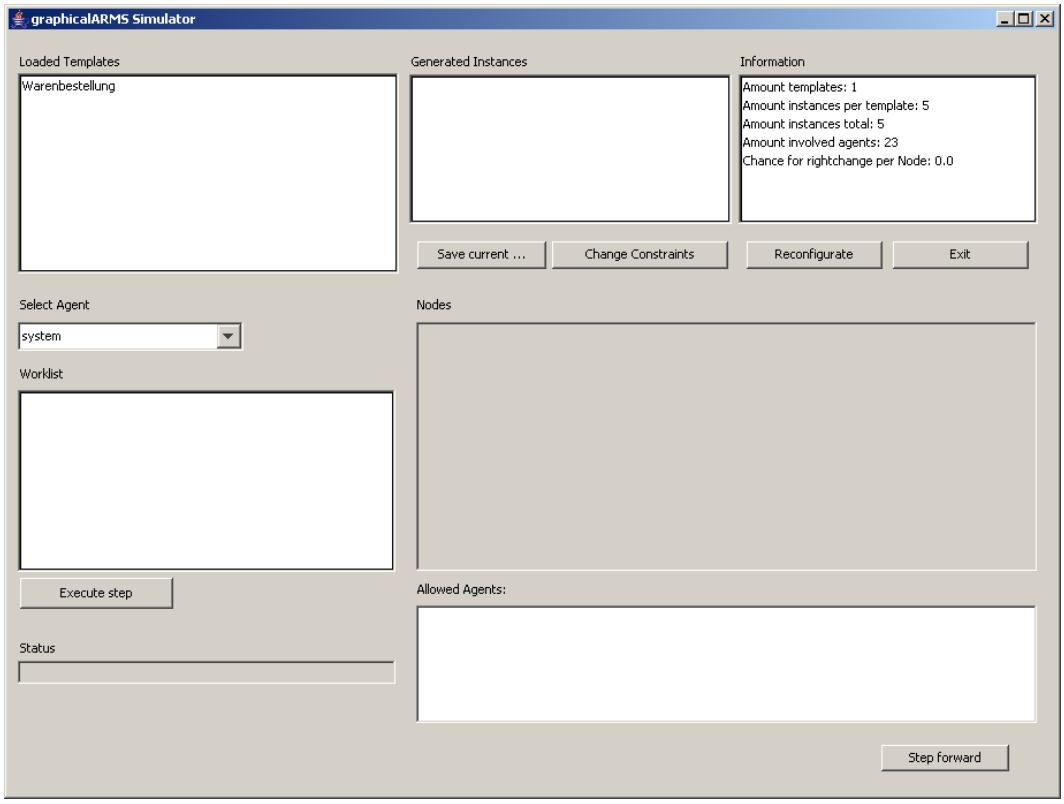


Abb. C.3: „graphicalARMS Simulator“ Hauptfenster.

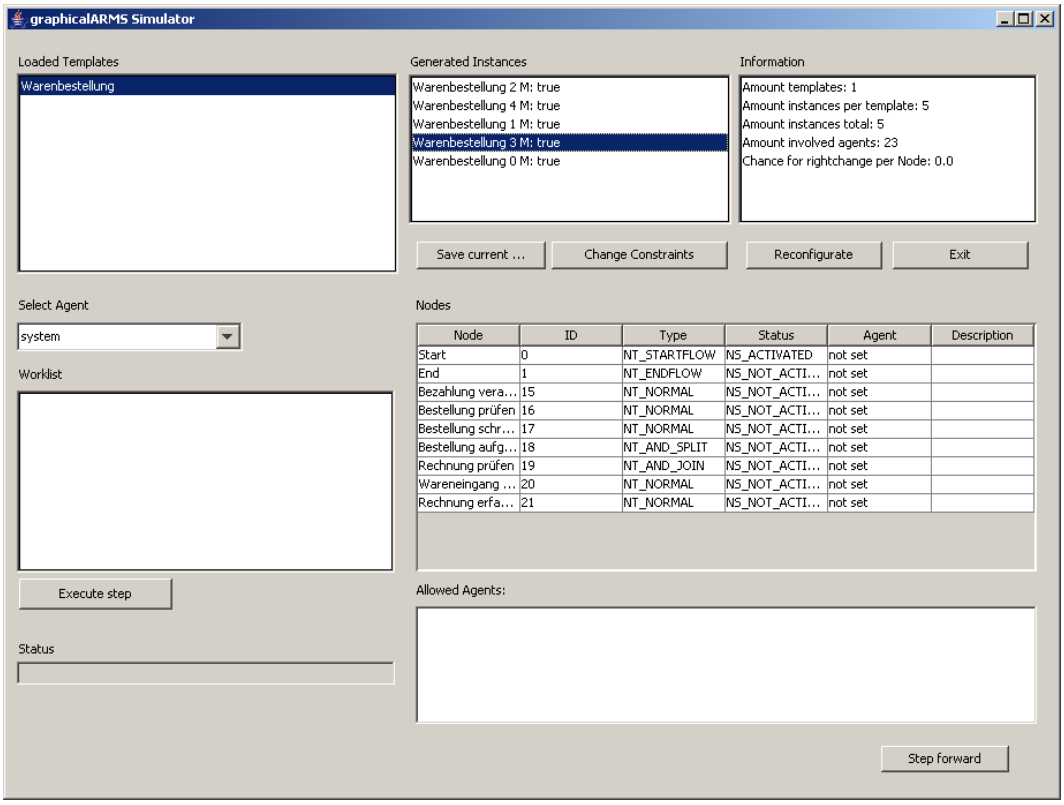


Abb. C.4: „graphicalARMS Simulator“ Hauptfenster nach Auswahl eines Templates und einer Instanz.

D Bedienung des entwickelten Tools

Anlegen und Ändern von Constraints

Ist die Anwendung „graphicalrmsSimulator“ gestartet und wurde eine Prozessinstanz geladen (Anhang C), so ist mit der Schaltfläche „Change Constraints“ das Ändern der Constraints für diese Prozessinstanz möglich. Der sich öffnende Dialog „Change Constraints“ (Abb. D.1) ermöglicht das Anlegen statischer Separation of Duty Constraints (Kapitel 5.3.2) und Binding of Duty Constraints (Kapitel 5.3.4). Hierzu existiert ein Reiter für jede Art von Constraints, wobei jeder Reiter zwei Listen enthält. Die linke Liste zeigt alle bereits angelegten Constraints („Already defined Constraints“). Die rechte Liste zeigt die Aktivitäten des in der linken Liste ausgewählten Constraints („Nodes for selected Constraint“).

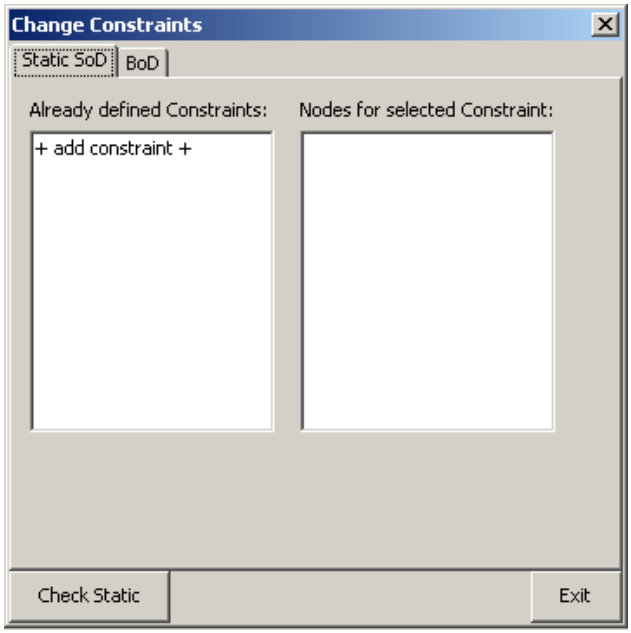


Abb. D.1: „Change Constraints“ Dialog.

Zum Anlegen eines statischen SoD Constraints muss doppelt auf „+ add constraint +“ geklickt werden. Damit öffnet sich der Dialog „Static SoD Configure Constraint“ (Abb. D.2). Hier finden sich auf der linken Seite alle Aktivitäten der Prozessinstanz („All Nodes“). Die rechte Liste enthält alle Aktivitäten des anzulegenden Constraints. Die Aktivitäten können nach Selektion mit Hilfe der Pfeil-Schaltflächen zum Constraint hinzugefügt bzw. aus diesem gelöscht werden. Nach Eingabe eines Namens für diesen Constraint (Abb. D.3) wird dieser mit „OK“ angelegt. Hiermit wird der Constraint, gemäß vorformulierter Constraint-Templates, auf assignMΘnK-Constraints abgebildet.

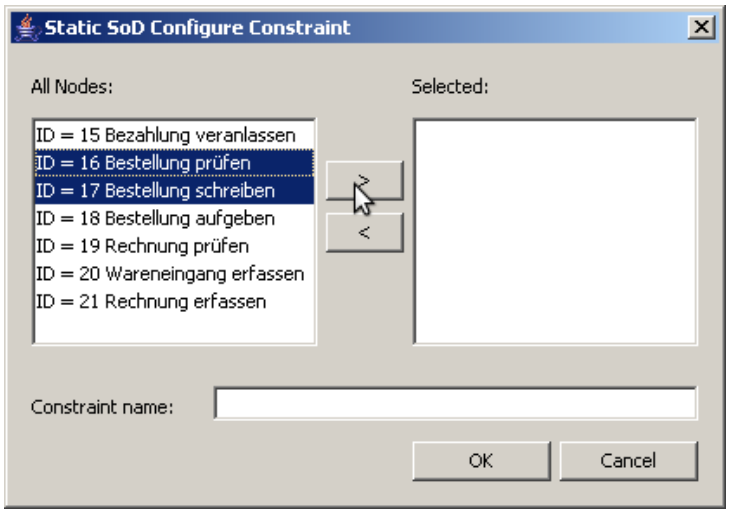


Abb. D.2: „Static SoD Configure Constraint“ Dialog.

Um einen bereits angelegten Constraint zu ändern, muss im Dialog „Change Constraints“ (Abb. D.4) auf den entsprechenden Constraint doppelt geklickt werden. Das Ändern des Constraints erfolgt dann wiederum mit Hilfe des „Static SoD Configure Constraint“ Dialogs (Abb. D.2). Das Anlegen und Ändern eines Binding of Duty Constraints erfolgt analog.

Betrachtung der Ergebnisse statischer Constraint-Prüfungen

Durch Betätigung der Schaltfläche „Check Static“ im Dialog „Change Constraints“ können die Ergebnisse der statischen Constraint-Prüfungen begutachtet werden. Hierzu öffnet sich

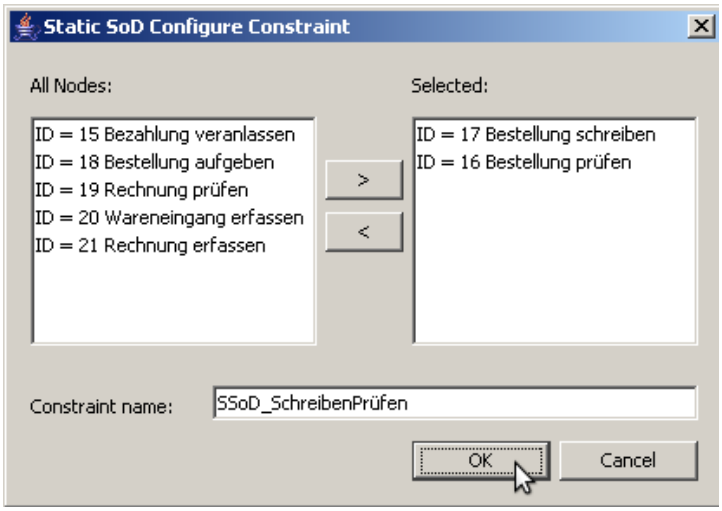


Abb. D.3: „Static SoD Configure Constraint“ Dialog.

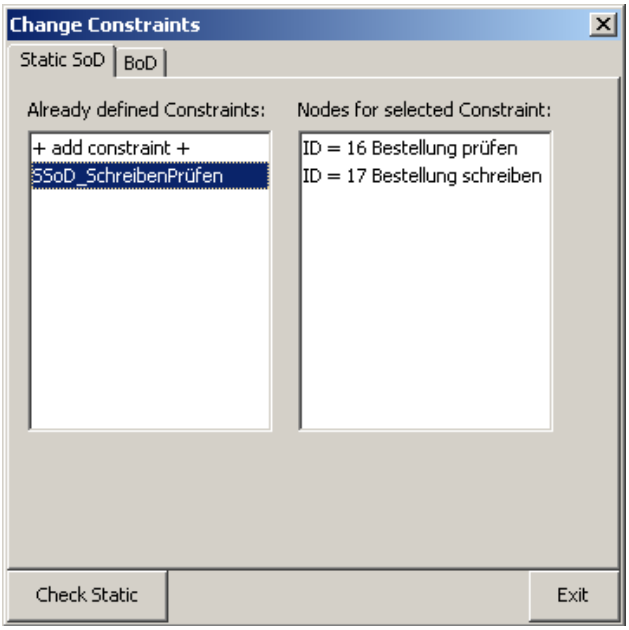


Abb. D.4: „Change Constraints“ Dialog mit angelegtem Static SoD Constraint.

der Dialog „Static Constraint Check Results“. Dieser stellt drei Reiter zur Verfügung, unter denen jeweils die Ergebnisse der Prüfungen zur

- Erfüllbarkeit einzelner assignMΘnK-Constraints
  („Satisfiability of assignMonkConstraints“)
- Einhaltung einzelner Constraints
  („Satisfied Constraints“)
- paarweisen Validierung von assignMΘnK-Constraints
  („Conflicting assignMonkConstraints“)

dargestellt werden. Mittels der RadioButtons am unteren Rand des Dialoges kann ausgewählt werden, ob alle, nur erfolgreiche, oder nur nicht erfolgreiche Prüfungen angezeigt werden sollen. Exemplarisch werden an dieser Stelle vier Ansichten beschrieben. Dabei wurden jeweils im Warenbestellungs-Prozess die Constraints „SSoD_SchreibenPrüfen“ und „BoD_Finanzen“ formuliert (vgl. Kapitel 8.2). Die Unterschiede zu den Abbildungen in Kapitel 8.2 ergeben sich durch die Selektion der RadioButtons.

Abb. D.5 zeigt ergänzend zu Abb. 8.8 (Seite 148) nur die erfüllbaren assignMΘnK-Constraints der Constraints „SSoD_SchreibenPrüfen“ und „BoD_Finanzen“. Es sind folglich alle assignMΘnK-Constraints erfüllbar. In Abb. D.6 werden nur die eingehaltenen der beiden Constraints „SSoD_SchreibenPrüfen“ und „BoD_Finanzen“ angezeigt. Im Gegensatz zu Abb. 8.9 ist daher nur der Constraint „BoD_Finanzen“ zu sehen. Abb. D.7 zeigt dagegen die nicht eingehaltenen Constraints. In diesem Fall entspricht dies dem Constraint „SSoD_SchreibenPrüfen“. Zu nicht eingehaltenen Constraints werden außerdem Fehlerbeschreibungen angezeigt („Failure Descriptions“). Diese beschreiben, welche assignMΘnK-Constraints nicht eingehalten wurden. Während Abb. 8.10 nur die sich widersprechenden assignMΘnK-Constraint-Paare der beiden Constraints zeigt (in diesem Fall sind dies keine), zeigt Abb. D.8 alle auf Widersprüche geprüften assignMΘnK-Constraint-Paare.

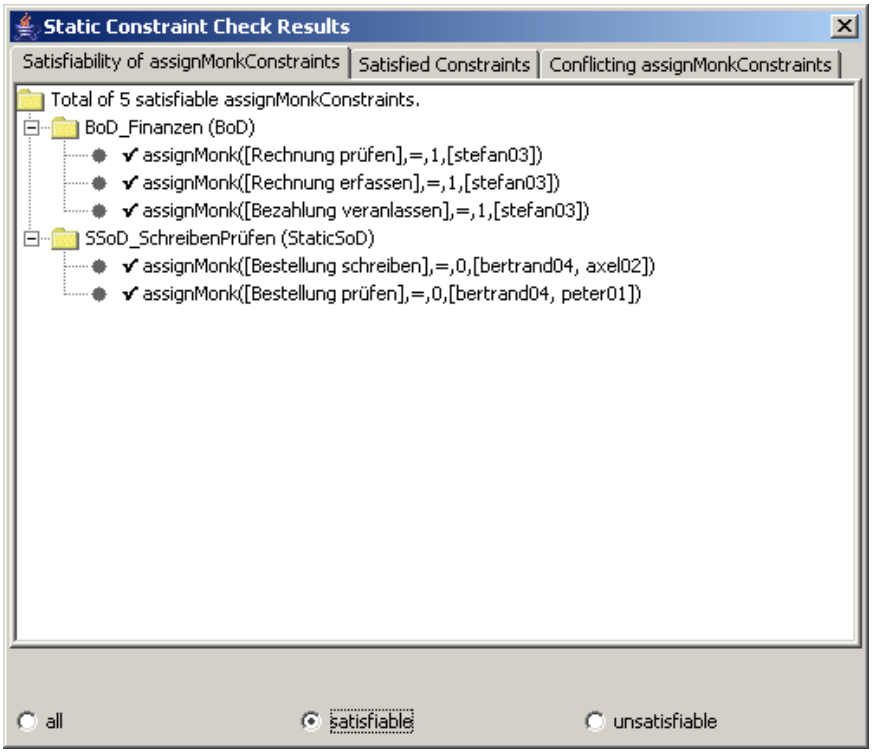


Abb. D.5: Erfüllbare assignMΘnK-Constraints der Constraints „SSoD_SchreibenPrüfen“ und „BoD_Finanzen“.

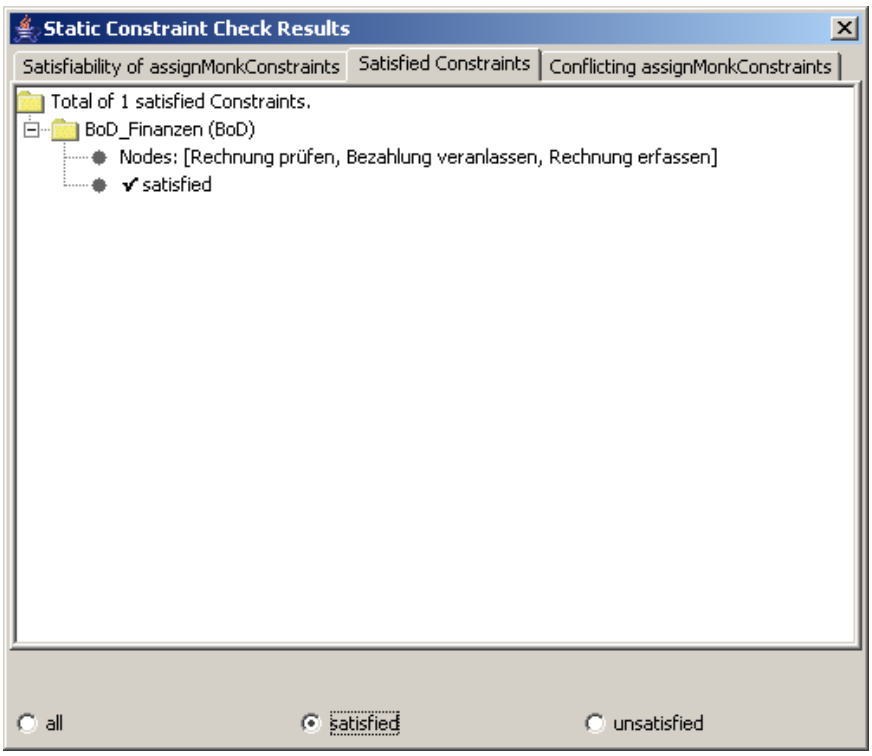


Abb. D.6: Eingehaltener Constraint „BoD_Finanzen“.

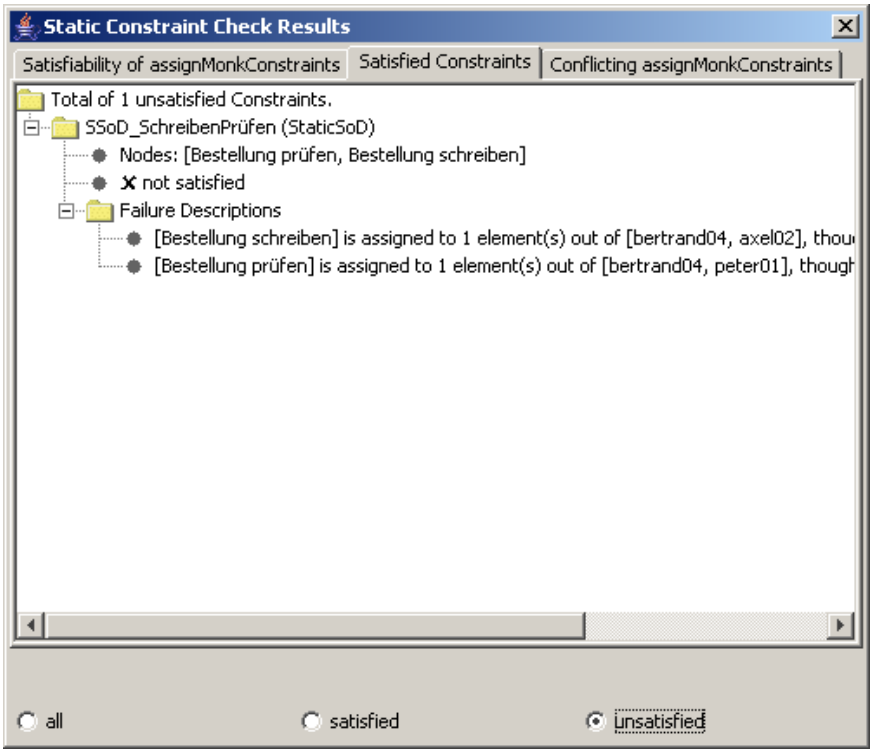


Abb. D.7: Nicht eingehaltener Constraint „SSoD_SchreibenPrüfen“.

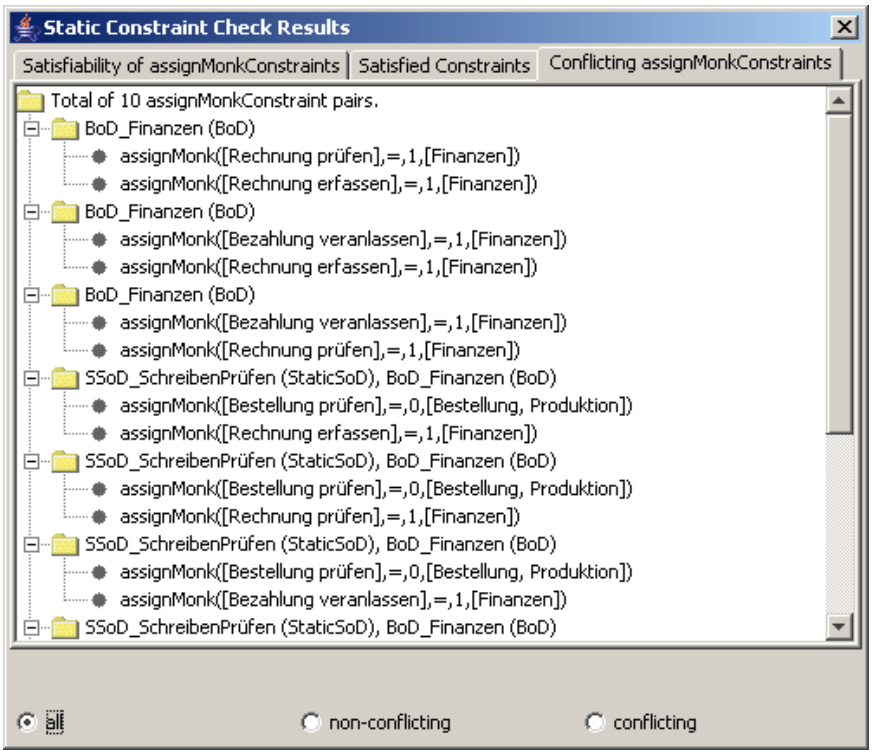


Abb. D.8: Alle auf Widersprüche untersuchten assignMΘnK-Constraint-Paare der Constraints „SSoD_SchreibenPrüfen“ und „BoD_Finanzen“.

Name: Florian Kelbert Matrikelnummer: 543210

Erklärung

Ich erkläre, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den

Florian Kelbert