Albert Ludwigs Universität - Freiburg

Institut für Informatik und Gesellschaft
Abteilung Telematik

Bachelor Thesis

zur Erlangung des akademischen Grades Bachelor of Science(B. Sc.) im Studiengang Embedded Systems Engineering

Inter-Instance Constraints

Author:
Regina König

Gutachter:
Prof. Dr. Dr. h.c. Günter Müller
Betreuer:
Adrian Lange

Abgabedatum:
2. August 2015

Erklärung

Hiermit erkläre ich, Regina KÖNIG, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Ort, Datum: _____

Unterschrift: _____

"Companies spend millions of dollars on firewalls, encryption and secure access devices, and it's money wasted, because none of these measures address the weakest link in the security chain."

Kevin Mitnick

ALBERT-LUDWIGS-UNIVERSITÄT FREIBURG

Abstract

Abteilung Telematik
Institut für Informatik und Gesellschaft

Bachelor of Science

Inter-Instance Constraints

by Regina KÖNIG

The Thesis Abstract is written here (and usually kept to just this page). The page is kept centered vertically so can expand into the blank space above the title too...

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Symbole

W	Prozessschema
W_i	Prozessschema i
W_i^k	k-te Instanz des Prozessschema i
$T = \{t_1, t_2, ..., t_n\}, n \in \mathbb{N}$	Menge von Aktivitäten
D	Menge der Abhängigkeiten der Aktivitäten
t_{ij}	Aktivität j aus Prozessschema i
t_{ij}^k	Aktivität j aus der k-ten Instanz des Prozessschema i
R	Menge der Rollen
U	Menge der Nutzer
$\mathcal{T}$	Menge aller Zeitpunkte
TP	Zeitpunkt Symbole
TS	Zeitspannen Symbole
E	Event Logs

Kapitel 1

Einleitung

1.1 Motivation

Um Sicherheit in Unternehmen gewährleisten zu können, reicht es nicht aus, in teure Software zu investieren, die das Unternehmen gegen externe Angriffe absichert. Ein nicht zu unterschätzender Teil (Quelle) der Bedrohung befindet sich im Inneren des Unternehmens. Angestellte können ihre Position und ihr Wissen ausnutzen, um sich oder Anderen Vorteile zu verschaffen. Es müssen aber nicht immer betrügerische Absichten hinter der Handlung eines Angestellten stehen. Auch menschliches Versagen kann bei dem Unternehmen Schäden verursachen. Das einfachste Konzept, um zu verhindern, dass Angestellte internen Betrug oder Fehler begehen, ist das *Separation of Duty* (**SOD**) Prinzip (Quelle). Dabei werden kritische Aufgaben in kleinere Teile zerlegt, die nicht von einer einzigen Person ausgeführt werden dürfen. Somit wird ein gewisses Maß an Kontrolle geboten [1]. Jedoch beziehen sich die meisten Konzepte auf Aufgaben aus derselben Instanz eines Prozesses. Neuere Forschungsergebnisse (Quelle) weisen darauf hin, das dies nicht ausreicht, und man ebenfalls Einschränkungen auf Aufgaben definieren muss, die aus verschiedenen Prozessinstanzen stammen.

Ein Prozess (Arbeitsablauf) in einem Unternehmen besteht aus mehreren, in sich abgeschlossenen Aufgaben. Diesen Aufgaben werden Rollen zugewiesen, die potentiell dazu berechtigt sind, die Aufgabe auszuführen. Jeder Mitarbeiter besitzt ebenfalls eine potentialle Menge an Rollen, in denen er agieren kann. Sobald ein Mitarbeiter in einer authorisierten Rolle eine Aufgabe annimt, kann sie nur von ihm durchgeführt werden [2].

Abbildung 1.1 stellt eine schematische Darstellung der Aufgaben in einer Bank dar, nachdem ein Kunde einen Kreditwunsch geäußert hat.

Sobald ein Kreditantrag bei der Bank eingeht, muss dieser geprüft werden. Je nach Ergebnis der Prüfung, wird der Antrag entweder bewilligt oder abgelehnt. In jedem Fall muss der Kunde wieder kontaktiert werden. Da es nicht erwünscht ist, dass der selbe Mitarbeiter, der den Antrag aufnimmt, ihn auch prüft, wird hier das *Separation of Duty Prinzp* angewendet. Im einfachsten

[1] Das Konzept versagt natürlich, wenn sich alle beteiligten Personen gemeinsam dazu entschließen, einen Betrug zu begehen.

[2] Es gibt die Möglichkeit, Aufgaben wieder abzugeben oder an einen anderen Mitarbeiter weiterzuleiten

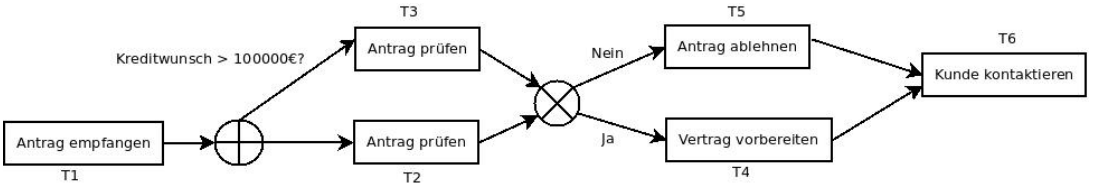

Abbildung 1.1: Arbeitsablauf in einer Bank, nachdem ein Kreditantrag eingegangen ist.

Fall würde der Prüfer die Fehler oder Falschangaben im Antrag entdecken und ihn zurückweisen. Ein schwierigeres Szenario ist, wenn sich zwei Angestellte absprechen und sich gegenseitig ihre Anträge bewilligen. Ein *SOD* Modell, welches jeden Arbeitsablauf für sich betrachtet, würde diesen Betrug nicht erkennen. Es wird eine Lösung benötigt, auch Einschränkungen zwischen mehreren Instanzen eines Arbeitsablaufs zu definieren. Um den Schaden gering zu halten, könnte man verbieten, dass zwei Mitarbeiter mehr als drei Mal an T1 und T2 gemeinsam arbeiten.

1.2 Ziel der Arbeit

Ziel der Arbeit ist es, eine Definitionssprache zu entwickeln, die es ermöglicht, Einschränkungen und Regeln zur Ausführung von Aufgaben durch bestimmte Angestellte sowohl innerhalb von Prozessinstanzen als auch zwischen mehreren Instanzen zu definieren. Dazu muss untersucht werden, welche Arten von Einschränkungen und Regeln es gibt und wie die Spannweite einer Regel definiert werden kann. Anschließend wird ein Modelchecker entwickelt, der *Eventlogs* auf die Einhaltung dieser Regeln untersucht.

1.3 Aufbau der Arbeit

Folgende Konventionen werden hier verwendet:
kursive Begriffe bezeichnen Fachbegriffe
Beim ersten Vorkommen werden Fachbegriffe fett geschrieben
`Blockschrift kennzeichnet Pseudocode, Klassennamen und ...`

In Kapitel 2 wird einerseits ein Überblick über die Grundlagenliteratur gegeben als auch verwandte Arbeit vorgestellt. Dabei wird kurz darauf eingegangen, inwiefern sich diese Arbeit von den Anderen unterscheidet. In Kapitel 3 werden wichtige Begriffe erläutert. In Kapitel 4 widmen wir uns der Herleitung von Einschränkungen und der Definition einer entsprechenden Grammatik. Diese wird in Kapitel 5 in ein Programm integriert, welches Event Logs auf die Verletzung von Einschränkungen prüft. Dazu wird der Algorithmus vorgestellt und der Aufbau des Programms vorgestellt. In Kapitel 6 wird ein Beispiel zur Funktionsweise des Programms vorgestellt. Letztendlich wird die Arbeit in Kapitel 7 mit ein paar abschließenden Bemerkungen beendet.

Kapitel 2

Verwandte Arbeit

2.1 Related work

2.1.1 Geschichte

Chinese Wall Modell Brewer/ Nash
Business Rules und Auditierung ?? Um die Zuverlässigkeit von Abläufen zu gewährleisten, wird schon lange Auditierung betrieben. Ursprünglich ging es darum, den korrekten Ablauf nachzuweisen. Man kann damit aber auch die zulässige Ausführung von Ereignissen prüfen.

2.1.2 Heute

Übergang zu Inter/instance Constraints Heute: verschiedene Forschungszweige: TBAC, Inter-Instance Constraints Allgemein eher den Inhalt auswerten, statt eine Liste von verwandten Arbeiten anzugeben Arten Unterscheiden, wie man Constraints spezifizieren kann Als logische Prädikate, GL4 Welche Constraint-Richtungen werden behandelt? ZB nur Entailment, Cardinality.

2.1.3 warner inter-Instance

Diese Arbeit bezieht sich hauptsächlich auf die Arbeit von Warner und Atluri [2]. In dieser Arbeit wird der theoretische Ansatz von Bertino et al. um inter instance constraints erweitert und ist der Ausgangspunkt meiner Arbeit. Allerdings gehen sie davon aus, dass die Constraints für Authorisierungszwecke gedacht sind, und unterscheiden deswegen zwischen statischer Analyse zur Entwurfszeit und dynamischer Analyse während des Ablaufs. Bei der dynamischen Analyse werden die Informationen nach jeder Aktion aktualisiert und unter Umständen die Liste von verbotenen Nutzern / Rollen angepasst. Da ich Auditierung benutze, wird statically checked weggelassen und weitere Regeln, die zur Laufzeit nötig sind.
Das Paper ist allerdings nur ein theoretischer Ansatz und behandelt zB die verschiedenen Eventtypen nicht ausreichen. Des weiteren wird nicht darauf eingegangen, was passiert, wenn manche Informationen fehlen.

2.1.4 leitner instance-spanning

Leitner et al. [6] stellen in ihrer Arbeit das Identification and Unification of Process Constraints (IUPC) compliance framework vor, dessen Fokus ebenfalls nicht nur auf Regeln liegt, deren Aktivitäten zur selben Prozessinstanz gehören. In der Arbeit beschäftigen sie sich mit der Frage, auf welche Aktivitäten sich eine Einschränkung bezieht (sind alle Aktivitäten involviert oder nur diejenigen aus einer bestimmten Prozessinstanz) und welche Spannweite eine Einschränkung hat. Sie unterscheiden dabei zwischen *intra-instance*, *inter-instance*, *inter-process*, *inter-organizational* und *trans-organizational*. Auf eine explizite Unterscheiden zu Einschränkungen zwischen Organisationen wird bei meiner Arbeit verzichtet, da der Organisationsname im verwendeten Logmodell als Metaattribut gespeichtert werden würde und somit einfach über die bereits vorhandenen Ausdrücke definiert werden könnte.

2.1.5 tan consistency

Es geht eher um die Prüfung auf Konsistenz, es werden aber globale Constraints erwähnt.

2.1.6 ProM - SCIFFchecker

Sciff wurde von Marco Alberti, Federico Chesani und Marco Gavanelli im Rahmen des SOCS Projekts entwickelt. Es arbeitet ebenfalls mit Prolog, kann aber keine Inter Instance Constraints.

Kapitel 3

Grundlagen

3.1 Grundlagen und Definitionen

In diesem Kapitel werden wichtige Begriffe vorgestellt, die im Verlauf der weiteren Arbeit von Bedeutung sind.

3.1.1 Prozess Schemata und Instanzen

Ein Prozessschema $\mathbf{W} = \{T, D\}$ mit n $\in \mathbb{N}$ ist eine Menge von Aktivitäten $\mathbf{T} = \{t_1, t_2, ..., t_n\}$ und einer Menge **D** von Abhängigkeiten, welche bestimmen, in welcher Reihenfolge die einzelnen Aktivitäten ausgeführt werden bzw von welchen Parametern abhängt, ob sie ausgeführt werden müssen, oder nicht. Die Menge der Aktivitäten muss mindestens eine Startaktivität haben, kann aber mehrere terminierende Aktivitäten beinhalten, die gleichberechtigt den Prozess abschließen. t_{ij} bezeichnet hierbei die Aktivität t_j aus dem Prozessschema W_i.

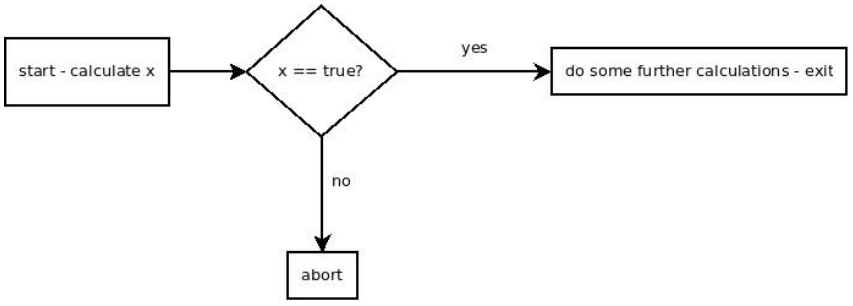

Abbildung 3.1: einfaches Beispiel eines Prozessschemas
Ein einfaches Beispiel für das Schema eines Prozesses mit einem Start und zwei finalen Aktivitäten.In Abhängigkeit davon, welcher Wert für x berechnet wird, wird der Prozess entweder sofort abgebrochen, oder es werden weitere Aktivitäten ausgeführt.

Ein Prozessschema kann mehrere Instanzen besitzen, die mit $\mathbf{W}_i^k$ gekennzeichnet werden. Eine Prozessinstanz W_i^k ist eine Menge von Instanzen T_i^k der zugehörigen Aktivitäten. Dabei ist

$|T_i| \subseteq |T|$ eine Teilmenge aller möglichen Aktivitäten des zugehörigen Schemas, da einzelne Aktivitäten unter Umständen aufgrund der Abhängigkeiten nicht ausgeführt werden müssen.

3.1.2 Aktivitäten

Eine Aktivität ist ein atomares Event, dh. eine in sich abgeschlossene Aufgabe im Kontext eines Prozesses. Sie haben eine definierte Menge von potentiellen Rollen und Nutzern, die die Erlaubnis besitzen, diese Aufgabe auszuführen. Sobald sie einem Nutzer zugewiesen wurde, kann kein weiterer Nutzer mehr die Aufgabe annehmen. Das bedeutet, dass jede Aktivität einen eindeutigen Nutzer und eine eindeutige Rolle hat, welcher sie ausgeführt hat.

Des weiteren besitzen die Aktivitäten einen eindeutigen Zeitstempel τ, zu dessen Zeitpunkt sie ausgeführt wurden. Diese Zeitstempel bestimmen eine Ordnung $< T, \leq >$. Es gilt nämlich $t_1 < t_2$, wenn t_1 vor t_2 ausgeführt wurde, dh. $\text{timestamp}(t_1) < \text{timestamp}(t_2)$.

Aktivitäten besitzen 7 verschiedene Zustände: *new, scheduled, assigned, active, suspended, completed, aborted*. Die Aktivitäten werden durch **Events** von einem Zustand in den nächsten geführt. Die Übergänge können sehr fein gegliedert sein oder sich nur auf elementare *Events* beschränken. In dieser Arbeit wird von 12 *Event*-Typen ausgegangen: *schedule, assign, reassign, start, complete, resume, suspend, autoskip, manualskip, withdraw, ate_abort, pi_abort*.

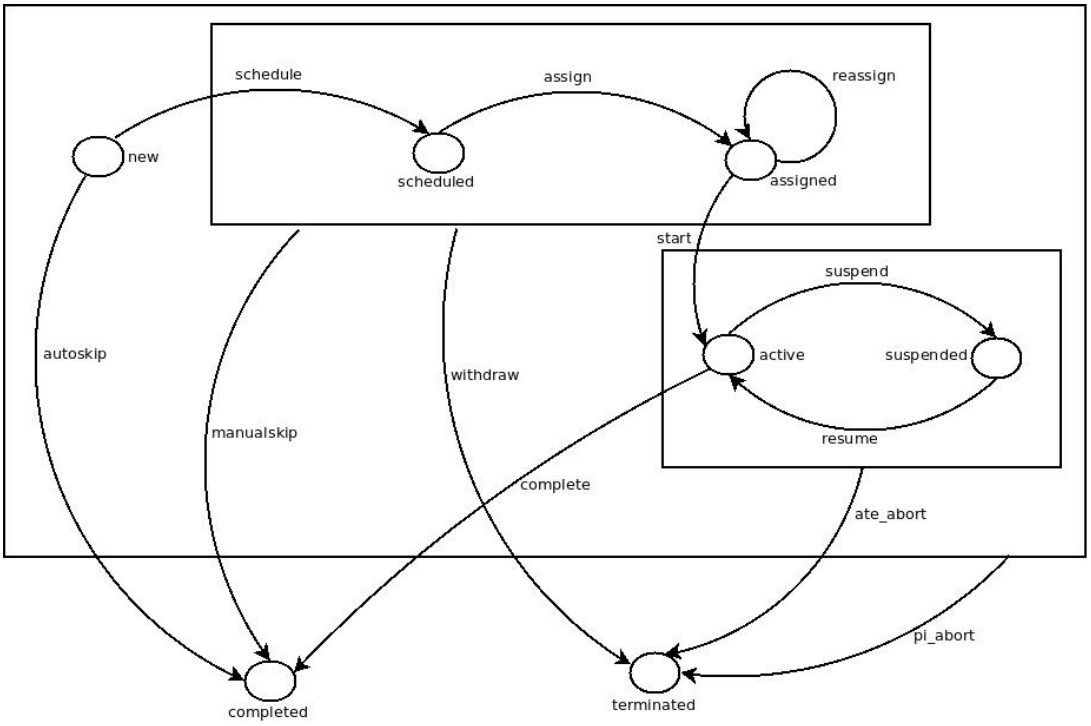

Abbildung 3.2: Aktivitäten und Events
Eine Aktivität und ihre möglichen Zustände. Der Startzustand ist *new*. Die neue Aktivität kann entweder sofort verworfen werden oder wird einem Nutzer zugewiesen und gestartet. Es gibt die Möglichkeit, die Aufgabe einem anderen Nutzer zuzuweisen oder zu pausieren und wieder zu starten. Aus jedem Zustand heraus kann die Aktivität abgebrochen werden.

3.1.3 Rollenmodell und Authorisierung

Sei $\mathbf{T} = \{t_1, t_2, ...t_m\}$, $m \in \mathbb{N}$ eine Menge von Tasks, $\mathbf{R} = \{r_1, r_2, ...t_n\}$, $n \in \mathbb{N}$ eine Menge von Rollen, und $\mathbf{U} = \{u_1, u_2, ...u_l\}$,$l \in \mathbb{N}$ eine Menge von Usern.

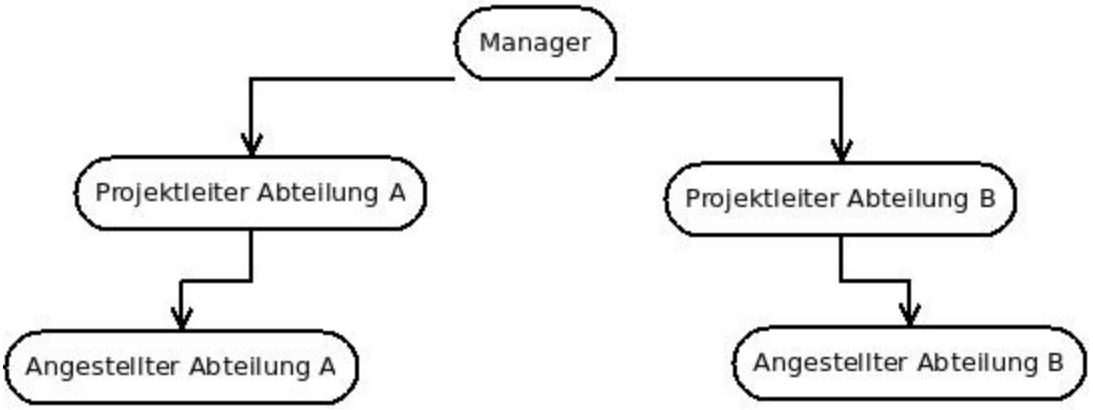

Abbildung 3.3: Beispiel Rollenmodell

Eine **Authorisierung** ist eine Menge von potentiellen Nutzer und Rollen, denen es erlaubt ist, einen Task auszuführen. Eine Authorisierung besteht aus den Tupeln

$$\mathbf{TR} = (T \times R)$$

und

$$\mathbf{UR} = (U \times R)$$

, welche eine n:m-Beziehung zwischen Tasks und Rollen, bzw zwischen Usern und Rollen kennzeichnen. Das bedeutet, dass User mit Rollen in der User-Rollen Beziehung assoziiert werden und Tasks mit Rollen in der Tast-Rollen Beziehung assoziiert werden.
Sei nun

$$\mathbf{R(t)} = \{r_m \in R : \exists(t_k, r_m) \in TR(t)\}$$

$$\mathbf{U(t)} = \{u_n \in U : \exists(u_n, r_m) \in UR, r_m \in R(t)\}$$

Mit anderen Worten ist R(t) die Menge aller Rollen, die authorisiert sind, einen Task auszuführen und U(t) die Menge alles User, die authorisiert sind, einem Task zugeteilt zu werden. Eine **Zuweisung** ist die konkrete Ausführung eines Tasks durch einen User.

Ein **hierarchisches Rollenmodell** ist eine geordnete Menge von Beziehungen zwischen Rollen $< R, \leq >$. Wenn $r_1, r_2 \in R$ und $r_1 < r_2$, dann dominiert die Rolle r_2 die Rolle r_1 in Bezug auf die organisatorische Rollenhierarchie. In Abb. 3.3 dominiert die Rolle Projektleiter"die Rolle Ängestellter", das bedeutet, dass der Projektleiterälle Tasks ausführen darf, die der Rolle Ängestellterßugeordnet wurde. Die Rolle und all ihre Elternrollen bis zur Wurzel können einem Task zugewiesen werden.

[1]

3.1.4 Zeitmodell

Das Zeitmodell ist ein Tupel T = $(\mathcal{T}; \leq)$.
$\mathcal{T}$ ist eine Menge von *Zeitpunkten* τ und $\leq$ eine total Ordnung auf $\mathcal{T}$. Ein *Zeitinterval* $[\tau_a, \tau_b]$ ist eine Menge von Zeitpunkten $\tau \in \mathcal{T}$ mit $\tau_a \leq \tau \leq \tau_b$. Ein Zeitinterval $\tau' = [\tau_a, \tau_b]$ wird als leer bezeichnet, falss $\tau_b \leq \tau_a$.

TS wird als die Menge aller Zeitpunkt Symbole wie 30.07.1999, 14:55 definiert und **TP** bezeichnet die Menge aller Zeitspannen wie 3 Tage 5 Stunden, 5 Monate. **TV** ... Variablen für Zeitpunkte TODO

[2]

3.1.5 Event Logs

EventLogs sind Abbildungen von Prozessen W_i auf eine Teilmenge $E \subseteq W_i$. Ein EventLog kann durchaus unvollständig sein bzw Fehler beinhalten. Jedoch wird in dieser Arbeit von einer *Closed World* ausgegangen, was bedeutet, dass eine Aktivität, die nicht geloggt wurde, auch nicht stattgefunden hat.

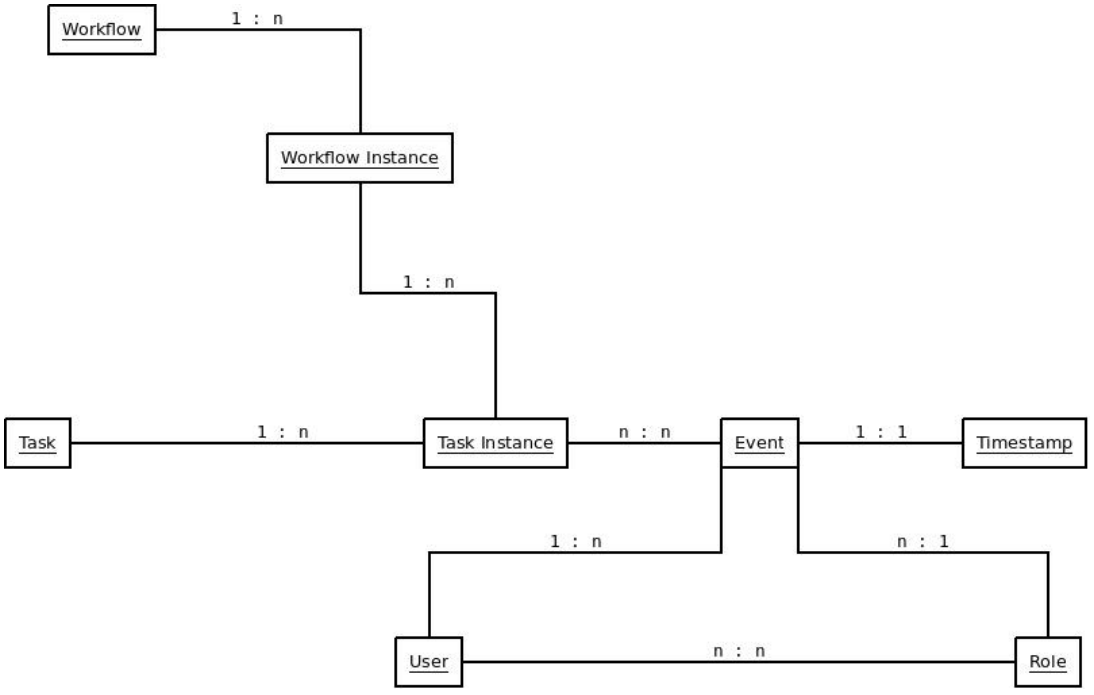

Abbildung 3.4: Ontologie eines Prozesses // TODO nochmal Beziehungen überdenken

// TODO Darstellung der additional Data, es ein bisschen verändern, damit man sieht, dass eine task von jedem element genau 1 hat

EventLogs repräsentieren die Instanzen eines Prozesses durch Auflistung der einzelnen Aktivitäten. In Tabelle 3.1 wird ein exemplarischer Log dargestellt. Die *caseID* kennzeichnet die

Instanz des Prozesses, da ein Prozessschema mehrere Instanzen besitzen kann. Zu einer Prozessinstanz können mehrere Aktivitäten (hier **Task**) gehören, die jedoch einen eindeutigen Zeitstempel, Eventtypen, Nutzer und dessen Rolle haben. Zudem kann eine Aktivität beliebig viele zusätzliche Informationen besitzen, die in dieser Tabelle nur angedeutet werden.

caseID	Task	User	Role	Timestamp	EventType	DataAttributes
0	Approach check	'Mark'	'Admin'	1999-12-13T12:22:15	start	(amount, 3000)
0	Pay check	'Theo'	'Azubi'	1999-12-13T12:22:16	start	(amount, 3000Euro)
1	Approach check	'Lucy'	'Azubi'	1999-12-13T12:22:17	start	(customer, Max Muster)
1	Pay check	'Mark'	'Admin'	1999-12-13T12:22:18	abort	()
0	Revoke check	'Theo'	'Clerk'	1999-12-13T12:22:19	start	()

Das ist nur ein Auszug und kein vollständiger Log. Es können auch weitere Daten vorhanden sein, die hier nicht dargestellt werden.

Tabelle 3.1: Beispiel Log Einträge.

3.1.6 Schutzziele

Um bestimmen zu können, welche Regeln und Einschränkungen für den Ablauf eines Prozesses notwendig sein könnten, muss zuletzt noch auf die Schutzziele von Unternehmen (besseres Wort) eingegangen werden.

Müller et al. [3] sammeln unter anderem[1] folgende Punkte:

(1) Autorisierung
Welche Subjekte bzw Rollen dürfen auf welche Ressourcen zugreifen? Da es in größeren Unternehmen komplex werden kann, wurde das Rollenmodell zur Vereinfachung entwickelt.
(2) Nutzungskontrolle
Regelt die Art der Nutzung. RWX, begrenzte Anzahl an Zugriffen, bzw Verpflichutung einer Löschoperation (weiteren, resultierenden Handlungen) nach Zugriff(Kontrollfluss)
(3) Interessenskonflikt
Unterbindung von unzulässiger Ausnutzung von insider-Wissen. Chinese Wall Modell.
(4) Funktionstrennung
Bestimmte Aufgaben dürfen nicht vom selben Subjekt, Rolle, Abteilung, ausgeführt werden. Unterbindung von kriminellen Handlungen und Betrug.
(5) Aufgabenbindung
Gegenteil von Funktionstrennung
(6) Mehr-Augen-Kontrolle
kritische Aktivität im Prozess darf nicht von einer einzelnen Person ausgeführt werden. Das wäre bei mir Cardinality Constraints, wenn eine Aktivität aus mehreren Tasks besteht.

// TODO: was kann man noch hinzufügen?

[1] Ein weiteres Schutzziel ist "Data Integrityünd Nutzungskontrolle", welches für diese Arbeit nicht relevant ist.

3.2 Herleitung der Einschränkungen

3.2.1 Ein praktisches Beispiel mit vielen Einschränkungen

In diesem Kapitel werden verschiedene Arten von Einschränlungen und Regeln betrachtet. Constraints sind Regeln bzw Einschränkungen, die aus dem Verlauf von vorhergehenden Tasks resultieren. Einschränkungen und Regeln werden hier synonym verwendet. Einschränkungen haben den Zweck Betrug, aber auch menschliches Versagen zu verhindern. Um sich ein besseres Bild von Einschränkungen zu machen, betrachten wir vor der eigentlichen Analyse ein Beispiel, welches im Verlauf der Arbeit auf die Einhaltung der Regeln untersucht wird. Das folgende Beispiel wurde in leicht veränderter Form [1] entnommen. // TODO stimmt das überhaupt?

Der Prozess in Abbildung 3.5 stellt die Bearbeitungsschritte eines Kreditantrages in einer Bank dar. In (T1 - Antrag empfangen) müssen zuerst alle erforderlichen Daten des Kunden aufgenommen werden. Daraufhin muss der Antrag geprüft werden, wie zum Beispiel die Kreditwürdigkeit des Kunden (T2 - Antrag prüfen). Sollte der gewünschte Betrag des Kredites 100000 Euro übersteigen, muss der Antrag zur Sicherheit von einer weiteren Person geprüft werden (T3 - Antrag prüfen). Nun muss entweder ein Vertrag vorbereitet werden (T4) oder, falls die Prüfung negativ verlaufen ist, ein Schreiben vorbereitet werden, welches dan Antrag ablehnt (T5). Unabhängig von dem Ergebnis wird der Kunde zuletzt noch einmal kontaktiert (6).

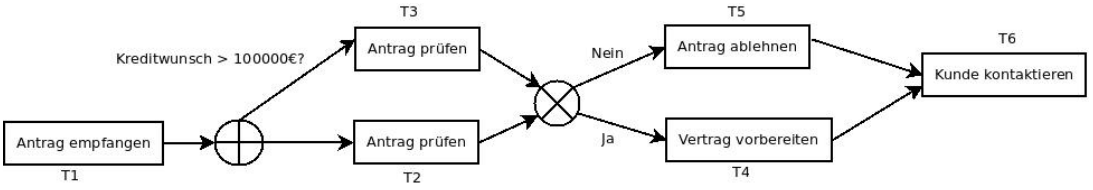

Abbildung 3.5: Bearbeitung eines Kreditantrags in der Bank

Um einen sicheren und reibungslosen Ablauf zu gewährleisten, werden folgende Anforderungen gestellt:

1. (Der Kontakt mit Kunden) T1 und T6 muss vom Kundenberater erledigt werden
2. Um den Kunden nicht zu lange warten zu lassen, sollte der Kunde spätestens 3 Tage nach erstem Kontakt über das Ergebnis informiert werden (timestamp(T6) < timestamp(T1) + 3D)
3. Um zu verhindern, dass Fehler durch Überlastung passieren, darf jeder Mitarbeiter am Tag höchstens 100 Tasks bearbeiten.
4. Den Antrag annehmen (T1) und den Antrag prüfen(T2, T3) sollten von verschiedenen Personen erledigt werden (4 Augen Prinzip).
5. Ferner sollten auch die zwei Prüfungen von verschiedenen Mitarbeitern vollzogen werden. T3 muss durch den Bank Manager erfolgen.
6. Wenn ein Mitarbeiter 5x einem Task zugewiesen wird und ihn dann abbricht, darf er nicht mehr an dem Task arbeiten.

7. Es dürfen keine Anträge von Verwandten geprüft werden.
8. Es dürfen auch höchstens 3 mal Mitarbeiter an den Anträgen des jeweils anderen Verwandten arbeiten.
9. Ein Mitarbeiter darf bei dem selben Kunden höchstens Kredite bis 100000 Euro prüfen.
10. Es dürfen hochstens 3 mal die selben Personen an T2 und T3 arbeiten

3.2.2 Arten von Constraints - Herleitung

Generell kann man verschiedene Arten von Regeln erkennen.

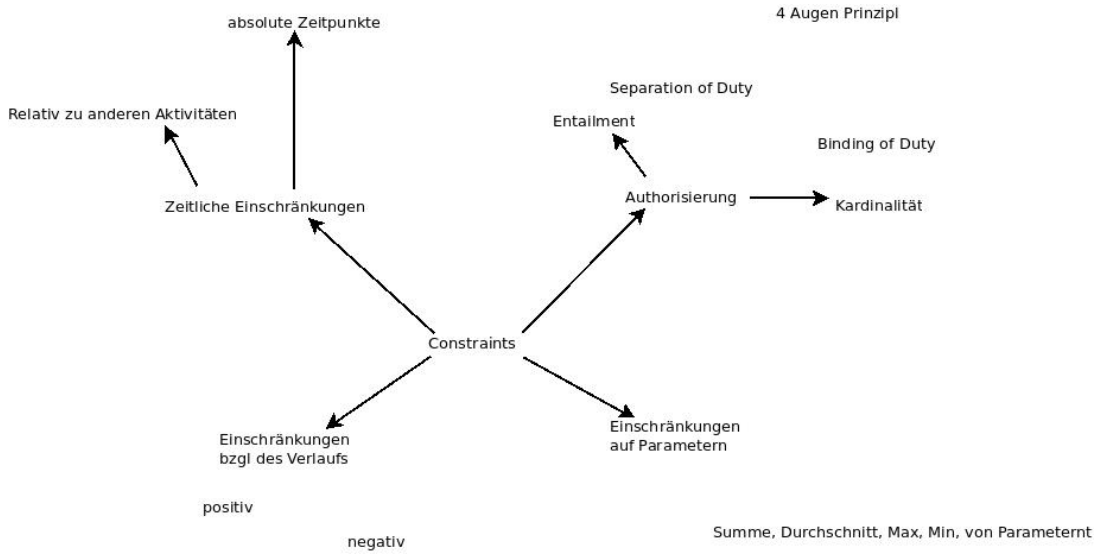

Abbildung 3.6: Typen von Einschränkungen und regeln

Sich gegenseitig ausschließende Tasks (Conflicting Tasks) TODO: kommt das zur Constraint Sammlung? In manchen Fällen kann man Tasks nicht verschiedenen Rollen zuweisen ohne die exisitierenden Rollen derart zu segmentieren, dass die schwer zu verwalten sind in Bezug auf das organsiatorische Modell. Außerdem können Rollenhierarchien dazu verwendet werden, in zwei verschiedenen Rollen zu agieren, die eigentlich getrennt waren. ZB könnte ein Manager als ein Clerk und gleichzeitig als sein eigenener Supervisor handeln.
Deswegen definieren wir **TC** $\subset$ T als eine Menge von **kollidierenden Tasks**. TC beinhaltet Tasks, deren Allokation von der Allokation von vorhergehend ausgeführten Tasks aus TC abhängt. Diese Abhängigkeit wird als Abhängigkeit zwischen Tasks aus TC beschrieben. $t_c \in TC$ gilt als entailed Task von $t_m \in TC$, wenn die Allokation von t_n von der Allokation von t_m eingeschränkt wird, mit $t_m < t_n$. [1]

Zeitliche Beschränkungen
–absolute Einschränkungen
Das sind Einschränkungen, die sich auf einen absoluten Zeitpunkt beziehen, wie zum Beispiel dass besondere Kredite nicht mehr vergeben werden dürfen, nachdem das Angebot erloschen ist.

Sollte nach dem festgelegten Datum trotzdem dieser Kredit vergeben werden, stellt das einen Regelbruch dar. –relative Einschränkungen
Relative Einschränkungen beziehen sich auf vorher erfüllte Aufgaben und schränken die weiteren für einen bestimmten Zeitraum ein.

Einschränkungen auf Parametern
Einschränkungen, die mit der Akkumulation von Aktivitäten arbeiten, geben oft eine Grenze für einen Parameter in einem bestimmten Zeitraum vor. Diese Art der Einschränkungen ist oft mit zeitlichen Einschränkungen verbunden.

Entailment Constraint c=(TC, n_u,m_{th}) mit n_u als minimale Anzahl an verschiedenen Usern, die einem Task zugewiesen werden müssen. Wenn t_{ki} eine Instanz des Tasks t_k ist, dann ist m_th der Grenzwert von der Summe der Task Instanzen, denen ein User zugewiesen sein darf. [1]
Separation of Duty
...
Binding of Duty Constraints
...
Cardinality Constraints
..
Workflow Soundness
Das sind Regeln, die allgemein den korrekten Ablauf eines Prozesses sicherstellen sollen. Diese Regeln beziehen sich im Allgemeinen nicht darauf, welcher Nutzer eine Aktivität ausgeführt hat (keine Authorisierungs-Einschränkungen) sondern betrachten die Tatsache, ob eine Aktivität zu einem korrekten Zeitpunkt oder in einem korrekten Zusammenhang ausgeführt wurde.

TODO Zum ein oder anderen Vielleicht Quellenangabe

3.2.3 Gültigkeitsbereich von Constraints

//TODO Welche der Einschränkungen gehört wohin **Intra-Instanz**
Die meisten Ansätze beziehen sich auf Regeln, die in Bezug zu einer Prozessinstanz stehen. Hier gilt, dass die Prozessinstanz für alle Aktivitäten die selbe ist. Sollten zwei Aktivitäten zu verschiedenen Instanzen gehören, werden sie nicht gegeneinander betrachtet. Seien A und B zwei Aktivitäten, dann gilt A.caseID = B.caseID.
Inter Instanz
Wie bereits erkannt wurde, reicht es nicht aus, nur Einschränkungen innerhalb eines Prozesses zu definieren, da sich eine kriminelle Handlung auch über mehrere Instanzen bemerkbar machen kann. TODO Weitere Erklärung oder kleineres Beispiel. Die Einschränkung von oben wird hier aufgehoben, es gilt jedoch noch, dass die betrachteten Tasks zum selben Prozessschema gehören müssen, dh A.case = B.case
Inter-Prozess
Die Regeln beziehen sich auf alle Aktivitäten, unabhängig davon, zu welcher Prozessinstanz

oder welchem Prozessschema sie gehören. Diese Einschränkungen betrachten häufig die Akkumulation von verschiedenen Aktivitäten, ihre Anzahl bzw Operationen auf die Aggregation ihrer Parameter.

Kapitel 4

Grammatik

In diesem Kapitel wird die Grammatik vorgestellt, die es ermöglichen soll, Regeln innerhalb von Prozessinstanzen aber auch Instanzübergreifend zu definieren. Zuerst muss geklärt werden, welche Anforderungen an die Grammatik gestellt werden und welche Fragen auftauchen. Im zweiten Teil wird die Syntax und Semantik der Grammatik erläutert. Schließlich wird aufgezeigt, wie die vorgestellten Regeln aus Kapitel 3.2.1 mithilfe der zuvor definierten Grammatik beschrieben werden können.

4.1 Anforderungen an die Grammatik

Um eine ausdrucksstarke Grammatik zu definieren, die möglichst viele Fälle abdeckt, muss zuerst untersucht werden, welche genauen Anforderungen an sie gestellt wird.

Grundsätzlich ist das Ziel, Regeln zur Ausführung bestimmter Aktivitäten auf Basis bereits abgeschlossener Aktivitäten aufzustellen. Es muss also einen Teil geben, in dem man die Bedingungen festlegen kann um dann anzugeben, welche gewünschte Aktion daraus resultiert. Die Bedingungen bilden in den meisten Fällen eine Disjunktion. Werden alle Bedingungen erfüllt, tritt die Regel in Kraft. Zur Vollständigkeit wird hier auch die Option mit aufgenommen, absolute Regeln aufzustellen, die in jedem Fall gelten sollen.

Für alle Regeltypen, die in 3.2.2 gefunden wurden, ist Disjunktion von positiven Bedingungen erlaubt. Um dem Nutzer größtmögliche Freiheit zu lassen, sollte Konjunktion und Negation von Bedingungen ebenfalls angeboten werden.

Das Ziel der Arbeit ist es, Regeln in Instanz-übergreifendem Kontext aufstellen zu können. Trotzdem muss es auch möglich sein, Regeln wie in bisherigen Compliancecheckern auch innerhalb einer Instanz zu prüfen. Es ist deswegen eine eindeutige Konvention notwendig, die deutlich macht, in welchem Kontext die Regel arbeitet.

Im Rollenbasierten Authorisierungsmodell existiert das einfache Rollenmodell, in welchem dem Nutzer nur die Rollen zur Verfügung stehen, die ihm explizit zugewiesen wurden. Es gibt keine

eindeutig festgelegten Hierarchien zwischen den Rollen. Im hierarchischen Rollenmodell hingegen kann ein Nutzer jede Rolle annehmen, die gleich oder unter der ihm zugewiesenen Rolle steht. Die Grammatik sollte beide Modelle behandeln können.

Da es Bedingungen in Bezug auf Zeitpunkte, Zeitunterschiede und Werte von Attributen gibt, müssen zumindest grundlegende arithmetische Operationen erlaubt sein.

Eventlogs speichern zu jeder Aktivität jedes einzelne Event. Das könnte entweder die Eingabe der Regeln unnötig kompliziert machen (indem der Anwender zu jeder einzelnen Aktivität immer das Event angeben muss) oder es würde zu multiplen Ergebnissen führen. Es muss deshalb eine Einigung geben, auf welches Event sich eine Bedingung bezieht und gleichzeitig die Möglichkeit offenlassen, auch weitere Events zu untersuchen. Zum Beispiel besteht die Frage, auf welches Event einer Aktivität sich der Zeitpunkt der Aktivität bezieht.

Unter Berücksichtigung aller gestellten Forderungen sollte trotzdem eine möglichst intuitive, leicht zu lernende und leicht zu verstehende Notation gewährleistet sein. Im folgenden wird die entwickelte Grammatik vorgestellt und im Anschluss geklärt, inwiefer die hier aufgeführten Fragen und Anforderungen gelöst wurden.

4.2 Definition der Grammatik

Für ein besseres Verständnis der Definition wird hier zuerst ein kleines, intuitiv zu verstehendes Beispiel aufgezeigt (Abb. 4.1). Am Anfang wird spezifiziert, dass Mark Maier und Max Mueller Verwandte sind. In der nächsten Zeile wird die Regel aufgestellt, dass Verwandte nicht gemeinsam and den Aktivitäten 'Kredit beantragen' und 'Kredit prüfen' arbeiten dürfen. Sollte Mark Maier den Kredit beantragt haben, wird Max Mueller die Prüfung des Kredits untersagt.

```
SET 'Mark Maier' is related to 'Max Mueller'

DESC "'Kredit beantragen' und 'Kredit prüfen' darf nicht von Verwandten ausgeführt werden"
IF USER_A executed 'Kredit beantragen' AND USER_A is related to USER_B
  THEN USER_B cannot execute 'Kredit prüfen'
```

Abbildung 4.1: Beispiel Spezifikation einer einfachen Regel

Das Ziel der Grammatik ist es, gewünschte oder unerwünscht Aktionen in Abhängigkeit von zuvor stattgefundenen Aktivitäten zu definieren. Zu diesem Zweck werden die Bedingungen als eine Disjunktion von Prädikaten über den Verlauf gebildet. Sollten diese Aussagen alle zu einem positiven Ergebnis führen, tritt die Einschränkung in Kraft. Diese Einschränkung macht eine Aussage darüber, ob eine bestimmte Aktivität von einem Nutzer / Rolle ausgeführt werden muss bzw dass sie von einem Nutzer/Rolle nicht ausgeführt werden darf. Es gibt auch Regeln, die aufzeigen, wann ein Verlauf nicht der Spezifikation entsprach (`illegal_execution`).
In den nächsten Abschnitten werden zuerst die Variablen, Konstanten und Prädikate vorgestellt, bevor dann im Anschluss die Syntax und Semantik der Grammatik genauer erläutert wird.

4.2.1 Argumente - Variablen und Konstanten

Prädikate sind Aussagen über Parameter einer Aktivität oder über Beziehungen zwischen Rollen oder Nutzern. Außer dem Prädikat `illegal execution` muss jedem mindestens ein Argument übergeben werden. Die Argumente sind entweder Variablen oder Konstanten in Form einer Zeichenkette oder einem numerischen Wert. Der Typ des Argumentes wird in der Grammatik nicht explizit deklariert, sondern erschließt sich aus dem Kontext des jeweiligen Prädikates. In diesem Abschnitt werden alle verwendbaren Typen vorgestellt. Sollte ein Argument als Variable übergeben werden, beginnt das Argument für jeden Typ mit einem Großbuchstaben. Die Form der Konstanten hängt von dem entsprechenden Typ ab. Grundsätzlich gilt, dass alle Typen außer den Zeiten und numerischen Werten einen String als Konstante haben, der mit einfachen Anführungszeichen umschlossen sein muss. Innerhalb der Anführungszeichen sind alle Zeichen erlaubt.

Typ	**Beschreibung**
UT	Variablen und Konstanten über Nutzer. Als Konstante sind Nutzer ein String.
RT	Variablen und Konstanten über Rollen. Als Konstante sind Rollen ein String.
TT	Variablen und Konstanten über Aktivitäten. Als Konstante sind Aktivitäten ein String.
WT	Variablen und Konstanten über Prozesse. Dieser Typ bezeichnet das Prozessschema. Als Konstante sind Prozesse ein String.
WIT	Variablen und Konstanten über Prozessinstanzen. Als Konstante sind Prozessinstanzen ein String.
ET	Variablen und Konstanten über Eventtypen. Als Konstante sind Events aus der Menge {'started', 'completed',..} (siehe 3.1.2).
TP	Variablen und Konstanten über Zeitpunkte. Als Konstante // TODO ISO und Verweise auf genaue Definition.. oder doch lieber hier genau deifnieren?.
TS	Variablen und Konstanten über Zeitspannen. Als Konstante // TODO.
NT	Variablen und Konstanten über numerische Werte. Als Konstante sind numerische Werte eine Zahl größer Null.

Tabelle 4.1: Argument Typen, die bei Prädikaten vorkommen können

4.2.2 Prädikate

Prädikate sind Aussagen über bestimmte Zustände. (TODO Ich wiederhole mich hier!!) Es gibt verschiedene Typen. Externe Informationen, Spezifikation, Status, Enforcement und Konditionell.

Externe Informationen (Tabelle 4.2) sind Aussagen, die nicht direkt mit der Workflow Spezifikation zu tun haben aber dennoch relevant für den Ablauf sein könnten. Diese Prädikate müssen explizit in den Regeln gesetzt werden, da es sonst zu keinem Ergebnis führt.

Spezifikationsprädikate (Tabelle 4.3) bestimmen die Beziehungen und Zugehörigkeit zwischen Nutzern, Rollen und Tasks. Sie Sollten genauso gesetzt werden, wie die Spezifikation war, als der Workflow ausgeführt wurde.

Prädikat	**Beschreibung**
UT is related to UT	Beide User sind verwandt
UT is partner fo UT	Beide Akteure sind Partner
UT is in same group as UT	Beide Akteure sind in der selben Gruppe, Abteilung

Tabelle 4.2: Prädikate für externe Informationen. Das sind nur drei Vorlagen. Dem Programmierer ist selbst überlassen, wie er diese Prädikate interpretieren will.

Prädikat	**Beschreibung**
'role' RT 'can execute' TT	RT ist in R(TT)
'user' UT 'can execute' TT	UT ist in U(TT)
'user' UT 'belongs to role' RT	(UT,RT) ist in UR
RT 'is glb of' TT	greatest lower bound. TT muss mindestens mit Rolle RT ausgeführt werden
RT 'is lub' TT	lowest upper bound. TT darf höchstens mit Rolle RT ausgeführt werden
RT 'dominates' RT	Rolle 1 dominiert Rolle 2
'critical_task_pair(' TT ',' TT ')'	Die beiden Tasks sind ein kritisches Paar. Die User werden markiert, die dieses Paar ausführen

In Bezug auf die jeweilige Rollenhierarchie

Tabelle 4.3: Prädikate für die Spezifikation von ...

Statusprädikate (Tabelle 4.4) sind Aussagen über Aktivitäten. Diese werden später mit den Informationen aus den Logs verglichen.

Prädikat	**Beschreibung**
('user')? UT 'executed' TT	Ut hat TT so ausgeführt, dass TT completed ??
'role' RT 'executed' TT	RT hat TT ausgeführt
UT 'is assigned to' TT	UT wurde TT zugewiesen
TT 'aborted'	TT ist abgebrochen
TT 'succeeded'	TT hat geklappt
UT 'is collaborator of' UT	UT sind alle Akteure, die an criticalTaskPair gearbeitet haben

Tabelle 4.4: Prädikate, um Aussgen über den Status in die Regeln mit einbeziehen zu können

Konditionelle Prädikate (Tabelle 4.5)

Vergleiche (Tabelle 4.6)

Operationen (Tabelle 4.7)

Kopfprädikate. Diese müssen im Kopf einer Regel stehen, und dürfen nicht im Körper vorkommen. (Tabelle 4.8)

4.2.3 Regeln

Regeln (Constraints sind Schlussfolgerungen, die sich aus Vorbedingungen ergeben. Es gibt positive und negative. Die positiven sagen, dass etwas passieren muss, die negativen verbieten,

Prädikat	**Beschreibung**
number where body is RES	Zählt die Anzahl der verschiedenen Lösungen für body
number of VAR where body is RES	Zählt die Anzahl der verschiedenen Lösungen für VAR, die body erfüllen. VAR muss mindestens einmal in body vorkommen.
sum of NT where body is RES	Gibt die Summe von NT zurück. NT muss im body enthalten sein und zählt alle Lösungen mit. NT muss
avg of NT ?TauT? where body is RES	Gibt den Durchschnitt von NT zurück.
min of NT ?TauT? where body is RES	Gibt das Minimum von NT zurück.
max of NT ?TauT? where body is RES	Gibt das Maximum von NT zurück.

Das Resultat wird in der Variable gespeichtert, die anstelle von RES definiert wurde. Body ist eine Konjunktion von Status- , Externen und Spezifikationsprädikaten.

Tabelle 4.5: Prädikate für Aussagen über die Akkumulation von Werten

Prädikat	**Beschreibung**
= |! =	ww
< | <= | > | >=	dd
...	

Tabelle 4.6: Vergleiche

Prädikat	**Beschreibung**
+|−	bb
∗|/	bb
...	

Tabelle 4.7: Operationsn

Prädikat	**Beschreibung**
UT cannot execute TT	ww
UT must execute TT	dd
RT cannot execute TT	
RT must execute TT	
illegal execution	
...	

Tabelle 4.8: Prädikate für den Kopf einer Regel

dass etwas passiert)
Ableitung,...
Die Köper der Regel werden als Konjunktion (Disjunktion ist ebenfalls möglich, jedoch nicht zwingend notwendig - siehe Kapitel 4.3.1) von Prädikaten gebildet.

```
IF body THEN head
```

Der Körper - body ...
Der Kopf der Regel - head darf nur aus

4.2.4 Grammatik - Syntax und Semantik

Reservierte Keywords

Konstanten sind als 'String' , Variablen ohne
Welche Zeichen darf man wo verwenden?
Verfügbare Literale
Syntax
Fakten: SET extern—workflow
Regeln: IF (..—..) (AND (..—..)*)
THEN (..—..)
WHERE t.name = t2.name AND ...

4.3 Verwendung der Grammatik, Erklärungen

Wenn sich etwas auf jede Instanz einzeln beziehen muss, muss Task1.workflow.instance = Task2.workflow.instance
Eignet sich für hierarchisches und auch für normales Rollenmodell. Hängt nur davon ab, ob die Hierarchie als Fakt gesetzt wurde.
User und ihre Rollenzuweisungen müssen nicht explizit angegeben werden. Nur wenn man das für die Vergleiche braucht.
Beziehung zwischen critical task pair und collaborateur.

4.3.1 Disjunktion

Es wird wenige Fälle geben, in denen eine Disjunktion von Klauseln notwendig ist. Um besser nachvollziehen zu können, zu welchem Fall eine Regelverletzung gehört, ist es oft sogar sinnvoller, eine Regel, die mehrere Fälle erlaubt, und mehrere Regeln aufzuteilen. Jedoch kann es verwendet werden, um Kardinalitätsaussagen einfacher darstelen zu können (TODO: mein Beispiel auf einem Zettel suchen), deswegen wird Disjunktion in der Grammatik erlaubt. Um die ??Assoziativität?? deutlich zumachen, müssen Disjunktionen in einer Klammer stehen. Disjunktion ohne umgebende Klammern ist nicht erlaubt. Disjunktionen von Konjunktionstermen ist nicht erlaubt.

```
IF (User executed 'task1' OR USER executed 'task2')
THEN User cannot execute 'task3'
```

Abbildung 4.2: Disjunktion im Körper einer Regel. Man beachte, dass jede Disjunktion von umschließenden Klammern umgeben sein muss.

Im Kopf einer Regel ist Disjunktion nicht erlaubt. Die Regel muss in zwei getrennte Regeln gespalten werden.

```
IF User executed 'task1'
THEN User cannot execute 'task2'

IF User executed 'task1'
THEN User cannot execute 'task3'
```

Abbildung 4.3: Disjunktion im Kopfbereich ist nicht erlaubt. Es müssen zwei getrennte Regeln erstellt werden.

4.3.2 Umgang mit verschiedenen Rollenmodellen

In den meisten Systemen wird ein hierarchisches Rollenmodell eingesetzt. Man ist bei der Analyse mit diesem Modelchecker nicht daran gebunden.

4.3.2.1 Definition eigener Prädikate

Es kann nötig erscheinen, eigene Prädikate zu definieren. Zum Beispiel um weitere Personengruppen anzulegen oder um Abkürzungen für Zusammenhänge zu erstellen. Dafür gibt es die Möglichkeit, ein Prädikat mittels des Schlüsselwortes "Defßu Beginn der Regeldefinitionen zu setzen. In der Definition wird der Typ der Argumente gesetzt. Man hat die Wahl zwischen UT,RT,TT,... Um die eigenen Prädikate nutzen zu können, muss man jedoch beachten, dass sie bei der Definition noch keinen "Wert"haben. Entweder muss man sie dann mit SSETßetzen oder mann kann ihnen eine Regel erstellen. Um die Prädikate für den Parser und die weitere Verarbeitung eindeutig sind, sind sie an eine spezielle Form gebunden: predicate_name(ARG-Type,...).

```
DEF suspicious(UT)
DEF skipped(TT)
DEF aborted(UT,TT)

SET suspicous('Max Neuer')
SET suspicous('Tom Weisser')

IF EventType(ACTIVITY).'skipped' THEN skipped(ACTIVITY)
IF ACTOR executed ACTIVITY AND EventType(ACTIVITY).'aborted' THEN aborted(ACTOR, ACTIVITY)
```

Abbildung 4.4: Definition eigener Prädikate. .. wurde gesetzt, .. hat eine eigene Regel

4.3.3 Negation

Negation wird ebenfalls für die Bildung der meisten Regeln nicht benötigt, wird hier aber der Vollständigkeit halber erlaubt. Dabei gilt hier das Prinzip der **negation as failure**. Da nicht

garantiert werden kann, dass ein Arbeitsablauf vollständig und korrekt geloggt wurde, muss man sich hier darauf einigen, dass das nicht-vorhandenseinëiner Klausel trotzdem mit der Negation dieser Klausel gleichzusetzen ist. Sollte das ein Fehler sein, entsteht ein False-positive Eintrag, dh. es wird ein Fehler zuviel angezeigt.

4.4 konkretes Beispiel

Die zuvor gefundenen Beispiele wären mit der neuen Grammtik:

```
/* C1 */
IF NUMBER OF USER executed TASK IS N AND N > 5
  THEN USER cannot execute TASK
```

Abbildung 4.5: Regeln für unsere gefundenen Beispiele

Kapitel 5

Implementierung

In diesem Kapitel wird darauf eingegangen, aus welchen Komponenten der im Rahmen dieser Arbeit entwickelte IICMChecker (**I**nter-**I**nstance **C**onstraints **M**odel **C**hecker) zur Überprüfung auf Einhaltung von Einschränkungen bei Prozessen auf Log-Basis aufgebaut ist und welche Funktion sie haben. Das Programm ermöglicht die Eingabe von Einschränkungen und Regeln im Intra-Instance sowie im Inter-Instance Kontext in einer intuitiven, an die natürliche Sprache angelehnte Eingabesprache. Diese Regeln werden in einem Container abgelegt, der sie in Klauseln im Prolog Format konvertiert. Ebenfalls werden einige Optimierungen und zusätzliche Prädikate hinzugefügt, die sicher stellen, dass die Regeln im richtigen Kontext angewandt werden.

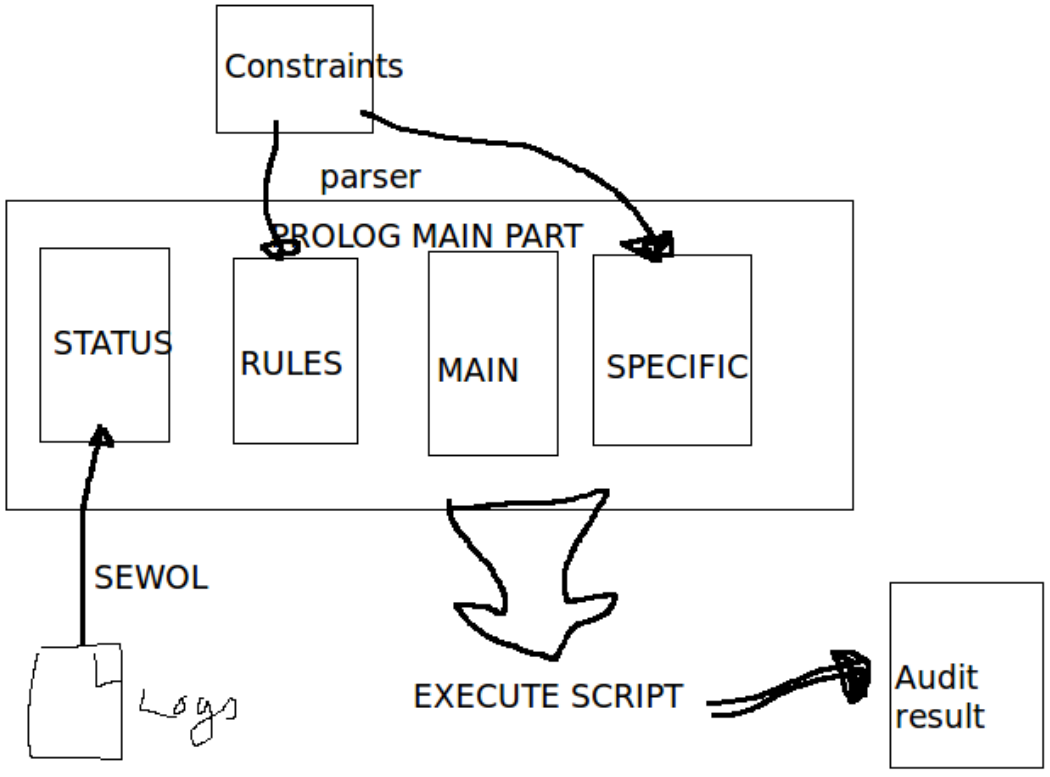

Abbildung 5.1: Aufbau des IICMCheckers

Das Programm besteht aus drei Teilen die über Prologdateien miteinander kommunizieren, wobei die Lesenden Teile in Java implementiert sind und der ausführende Teil (der eigentliche Modelchecker) in Prolog ausgeführt wird. Der Logtransformer liest Logs im SEWOL [4] Format ein und übersetzt sie in Status Prädikate (siehe Abschnitt 5.2.1). Die Constraints werden ebenfalls vom Constraintreader (siehe Abschnitt 5.2.2) eingelesen und in Regelprädikate übersetzt. Der Compliancechecker selbst ist in Prolog geschrieben und liest dazu die zuvor erstellten Statusprädikate und Regeln (für näheres siehe Abschnitt 5.2.3. Im letzten Abschnitt 5.2.4 wird noch kurz darauf eingegangen, wie die Ergebnisse des compliancecheckers interpretiert werden müssen.

5.1 Verwendete Hilfsmittel

5.1.1 SEWOL

SEWOL (**SE**curity-oriented **WO**rkflow **L**ib) wird am Institut für Informatik und Gesellschaft an der Universität Freiburg entwickelt. SEWOL ist eine Bibliothek, die das Arbeiten mit Prozesslogs unterstützt. Es bietet unter anderem die Möglichkeit, Logs in einfacher Text-Form, MXML und XES Format zu parsen und auch zu serialisieren. Ein Log besteht aus mehreren *traces*, wechels wiederum eine Liste von Log Einträgen enthält. Ein einzelner Eintrag repräsentiert dabei ein Event einer einzelnen Aktivität in einem Prozess. Zu jedem Logeintrag können zusätzliche Informationen hinzugefügt werden, wie der Zeitpunkt der Ausführung, der Nutzer (die Person oder das System, welcher für die Ausführung verantwortlich ist), die Rolle des Nutzers, den Typen des Events (start, complete,..) und weiteren Meta Informationen, die in Form von *DataAttributes* hinzugefügt werden können. [4] // TODO: ACHTUNG! SATZSTELLUNGEN ÄNDERN

5.1.2 ANTLR

ANTLR (**AN**other **T**ool for **L**anguage **R**ecognition) ist ein objektorientierter Parser Generator, der dazu geeignet ist, strukturierten Text zu lesen, zu bearbeiten, zu übersetzen und auszuführen. Die Applikation wird seit 1989 von Terence Parr an der Universität von San Francisco entwickelt und ist als freie Software verfügbar.
Eine vom Nutzer definierte Grammatik wird zuerst von einem Lexer verarbeitet und der entstehende Tokenstream anschließend geparst. Die Grammatik selbst ist in Parser und Lexer Regeln aufgeteilt. Zu der Grammatik erstellt ANTLR einen *parse tree*, welcher eine Datenstruktur ist, die repräsentiert, wie eine Grammatik den gelesenen Text interpretiert. Zudem wird ein entsprechender *parse tree walker* und *listener* erzeugt, die den *parse tree* durchlaufen und gemäß der Implementierung des *listeners* verarbeiten.
ANTLR selbst in in Java geschrieben.[5]

5.1.3 Programmiersprachen

Die Regeldefinition erfolgt in der eigens dafür entwickelten Grammatik (Kapitel 4). Das Einlesen der Logs sowie deren Konvertierung als auch das Übersetzen der Grammatik wurde in Java implementiert. Alle Informationen werden in Prolog Klauseln und Regeln übersetzt, welche mit dem Compliancechecker in Prolog ausgewertet werden.

5.2 Architektur und Umsetzung der Analyse

5.2.1 Erstellen der Wissensbasis - Einlesen der Logs

Um dem Compliancechecker die Informationen in einem verwendbaren Format als Prolog Klauseln liefern zu können, werden die Logs mittels SEWOL eingelesen. Der Logtransformer erstellt daraufhin für jeden einzelnen Eintrag je nach enthaltenen Informationen Klauseln aus sieben möglichen Prädikaten her, die in der Datei *Status.pl* im Ordner *prologfiles* gespeichert werden. Die **taskID** wird dabei automatisch generiert, um zusammengehörige Einträge zu identifizieren. Die restlichen Argumente entsprechen den Informationen, die zu den jeweiligen Aktivitäten herausgelesen werden konnten. Sollte ein Eintrag nicht vorhanden sein, wird er nicht gesetzt und gilt bei der späteren Untersuchung bei einer Anfrage **false** zurück.

Die ersten 2 Zeilen aus Tabelle 3.1 werden somit in die Wissensbasis aus Tabelle 5.2 konvertiert.

5.2.2 Übersetzung der Regeln

Der Parser und Listener für die Grammatik wurde mit ANTLR 4.5 [5] erzeugt. Die Grammatik selbst wird in einfachen Textdateien ohne Endung im Ordner *rulefiles* abgelegt [1]. Während der Parse Tree Walker den Parse Tree durchläuft, fügt der Listener die gefundenen Klauseln und Regeln den entsprechenden Containern hinzu. Gleichzeitig werden weitere Klauseln gesetzt, die notwendig sind, um den entsprechenden Kontext sicherzustellen. Sobald alle Regeln gesetzt sind, lässt der StorageHelper alle Container die enthaltenen Informationen in geordneter Reihenfolge in Prologdateien ablegen. Aus der Grammatik entstehen zwei Dateien *externspec.pl*(hier stehen alle Klauseln aus den `SET` Operationen) und *rules.pl* (die Regeln im `IF` body `THEN` head Abschnitt) im Ordner *prologfiles*.

Container Datenstruktur

Die Container sind eine Datenstruktur, in der alle Klauseln und Regeln zuerst abgelegt werden, um sie später geordnet und geprüft als Prologklauseln in Dateien ablegen zu können. Es gibt drei Container: `ExternAndSpecificationContainer`, `StatusContainer` und der `RuleContainer`. Im `StatusContainer` liegen alle Informationen, die aus den Logs gelesen werden.

[1] Im Anhang B - Argumente und Konfiguration wird beschrieben, wie sich der Quellordner und Dateinamen manuell anpassen lassen.

Prädikat	**Beschreibung**
workflow_name(caseID,workflowName)	Weist einer Case ID einen eindeutigen Namen hinzu, welcher der Workflow Spezifikation entspricht. Die Case ID kennzeichnet die Instanz, konkrete Ausführung eines Workflows. Dieses Prädikat dient jedoch nur zur Vollständigkeit. Sollte es für den Arbeitsablauf keinen eindeutigen Namen geben, wird dieses Prädikat nicht gesetzt. Es kann im Modelchecker dann auch nicht darauf zugegriffen werden.
task_workflow(taskID, caseID)	Setzt fest, zu welcher case ID die Activity gehört. TaskID wird intern vom Transformer gesetzt.
task_name(taskID,taskName)	Legt den Namen der Activity fest. Dieser ist im MXML Modell als WorkflowModelElement zu finden.
timestamp(taskID,timestamp)	Zeitpunkt der Aktivität in ms nach 1970.
eventtype(taskID,eventtype)	Events, wie sie in Abschnitt 3.1.2 - Aktivitäten vorgestellt werden. Es ist möglich, die Events neu zu definieren.
executed_user(user,taskID)	Der Nutzer, der die Activity ausgeführt hat.
executed_group(group,taskID)	Die Rolle, in der die Activity ausgeführt wurde. Man beachte, dass ein Nutzer mehrere Rollen haben kann.
task_attribute(taskID, attrName, attrValue)	Alle weiteren Attribute, die in SEWOL unter MetaAttributes stehen. Das zweite Argument ist der Name des Attributs und das dritte Argument der Wert des Attributs. Die Werte werden entweder als String gespeichert, oder - wenn möglich - als Nummer, um später Vergleiche und arithmetische Operationen zu ermöglichen.

Diese Statusprädikate werden aus den Logs extrahiert und sind ausschließlich dem Modelchecker sichtbar. Sie sind nicht zu verwechseln mit den Prädikaten, die in der Grammatik verwendet werden.

Tabelle 5.1: Statusprädikate

task_workflow(0,0)	task_workflow(1,0)
task_name(0,'Approach check')	task_name(1,'Pay check')
timestamp(0, 123)	timestamp(1, 123)
eventtype(0, start)	eventtype(1, start)
executed_user(0, 'Mark')	executed_user(1, 'Theo')
executed_role(0, 'Admin')	executed_role(1, 'Azubi')
task_attribute(0, 'amount', 3000)	task_attribute(0, 'amount', '3000Euro')

Tabelle 5.2: Aus einem Log herausgelesene Klauseln in Prolog. Die TaskID wird für jeden Eintrag automatisch generiert. Die restlichen Informationen stammen aus dem Log. Diejenigen Attributswerte, die eine valide Zahl darstellen, werden auch als Zahl gespeichert, um arithmetische Operationen zu erlauben. Die restlichen Werte werden als String im Prolog Format (zwei einfach Striche) gespeichert.

Der `ExternAndSpecificationContainer` enthält alle Klauseln, die in den Rulefiles gesetzt werden. Es sind nur Prädikate aus den Tabellen 4.2 und 4.3 erlaubt. Zusätzlich ist es möglich, eigene Prädikate zu definieren, die ebenfalls in diesem Container abgelegt werden. Zuletzt wird der `RuleContainer` mit den eigentlichen Einschränkungen befüllt. Diese widerum können nur Prädikate aus Tabelle 4.8 im Kopf der Regel enthalten, und werden nach diesen fünf Prädikaten gegliedert (`UserCannotDoRule`, `UserMustDoRule`, `RoleCannotDoRule`, `RoleMustDoRule` und

`IllegalExecutionRule`), die alle die abstrakte Klasse `Rule` erweitern. Jede Regel enthält einen `Rule_Body` in welchem alle Klauseln abgelegt werden, die in dem Körper der Regeln gesetzt wurden.

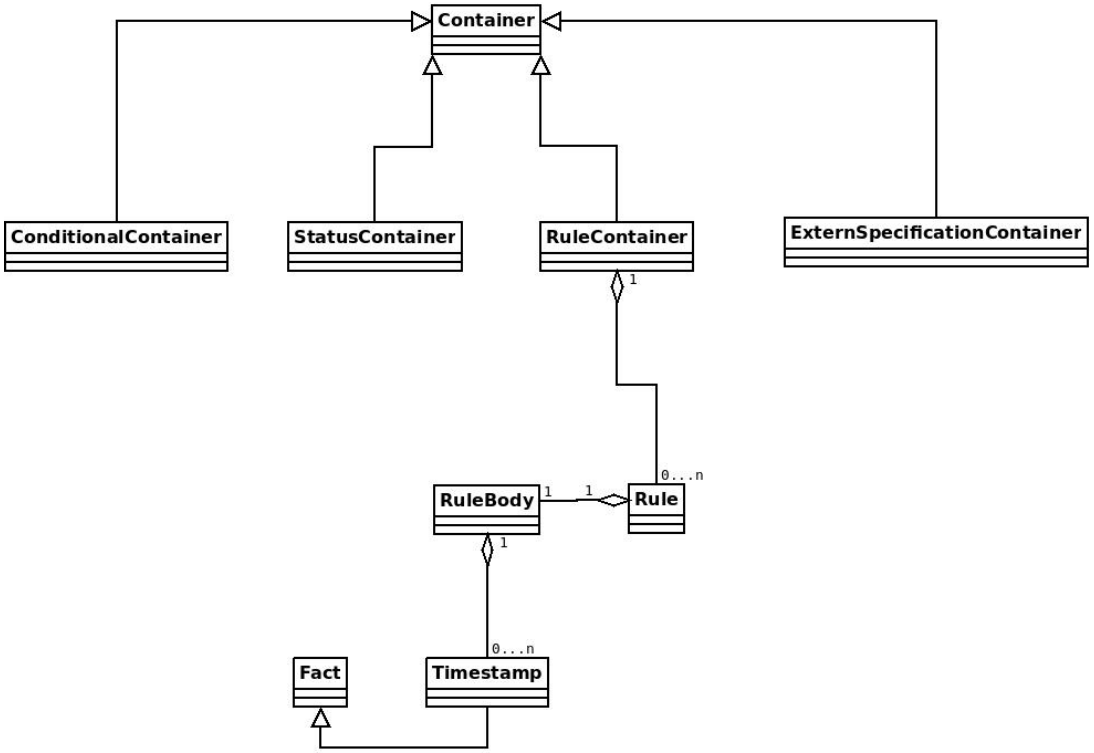

Abbildung 5.2: Interne Datenstruktur
TODO ganzes Bild stimmt nicht mehr

Transformation und Ergänzung der Regeln
Nachfolgend wird ein Einblick gegeben, welche Transformationen und Ergänzungen der Listener vollzieht. Die Auflistung ist nicht vollständig sondern skizziert nur die wichtigsten Anpassungen.

Um dem Nutzer eine möglichst intuitive Definition der Regeln zu ermöglichen, wird in den Regeln direkt der Name der Aktivität verwendet. Da es jedoch theoretisch erlaubt ist, mehrere Aktivitäten mit dem selben Namen zu versehen, wird intern für jede Aktivität eine taskID generiert, über die zusammengehörige Informationen identifiziert werden. So müssen alle Klauseln, die einen Namen für die Aktivität tragen, in zwei Klauseln geteilt werden.
`'Mark' executed 'Auftrag annehmen'` wird somit zu
`user_executed(taskID, 'Mark'), task_name(taskID, 'Auftrag annehmen')`.

Die wichtigste Aufgabe des Listeners ist es, den Regeln entsprechende Klauseln hinzuzufügen, die sicherstellen, dass die Regeln im richtigen Kontext angewandt werden - Intra-Instance, Inter-Instance oder Inter-Process. Intra-Instance Regeln gelten für Activities in der selben Prozessinstanz. Inter-Instance Regeln betrachten auch Aktivitäten, die aus unterschiedlichen Prozessinstanzen gehören, jedoch aus dem selben Prozessschema. Inter-Process Regeln gelten für alle Aktivitäten, unabhängig davon, zu welchem Prozess sie gehören...

Da Logs die Events einer Aktivität speichern, die Untersuchung in den meisten Fällen aber nicht die einzelnen Events einzeln betrachtet, werden bei den executed-Klauseln das Event start betrachtet.

5.2.3 Algorithmus Compliance Checker

In diesem Abschnitt wird betrachtet, wie der Compliance Checker die Logs auf die Einhaltung der definierten Regeln prüft. Der Checker selber besteht aus *start.pl* und *main.pl*. Die Datei *start.pl* enthält die Routine zum Starten der Untersuchung. In *main.pl* sind alle nötigen Funktionen, um implizite Beziehungen herzustellen und die einzelnen Routinen für die verschiedenen Regeltypen.

```
3. Add Collaborators
4. Adddominates
   Addrelated
5. For each cannot_execute_user(Actor, Activity):
6. If exists executed_user(Actor, Activity)
7.  then write trace
8. For each cannot_execute_role(Role, Activity):
9.  If exists executed_role(Role, Activity)
10. then write trace
11. For each must_execute_user(Actor, Activity):
12. If not exists executed_user(Actor, Activity)
13.  then write trace
14. For each must_execute_role(Role, Activity):
15.  If not exists executed_role(Role, Activity)
16. then write trace
17. For each illegal_execution write trace
```

Abbildung 5.3: Pseudocode des Modelchecker

Bevor die eigentliche Untersuchung beginnt, werden zuerst noch einige implizierte Klauseln die Prolog-Datenbank hinzugefügt. Es werden alle *collaborators* berechnet, die an critical_task_pairs gearbeitet haben, weitere Beziehung aus dem Rollenmodell und Verwandschaften ermittelt.... Für jede cannot execute Regel wird gefragt, ob der User das ausgeführt hat,...

5.2.4 Darstellung der Ergebnisse

Sollte eine ungewollte Aktivität im Log gefunden sein,

5.2.5 Struktur der Pakete

Hilfsklassen wie ... befinden sich im Paket utils

Kapitel 6

Ergebnisse und Diskussion

6.1 Evaluation

6.1.1 Mein Beispiel mit ein paar Logs untersuchen

Zur Auswertung wurden für den in ... beschrieben Workflow wurden für jede Regel jeweils 3 Logs manuell erstellt, die zusammen genau eine Regelverletzung beinhalten. Des weiteren gibt einen Log für 3 Cases, in denen keine Regel verletzt wurde. Um den Reibungslosen Ablauf auch bei mehreren Regelbrüchen zu prüfen, werden zum Schluss alle Logs gemeinsam eingelesen. Es wird erwartet, dass die selben Fehler gefunden werden, die bereits in den einzelnen Logs auftraten (+ natürlich ein paar weitere, wenn es um Akkumulation geht).

Erklärung, woher diese Logs kommen und wie sie erzeugt wurden. Die Logs wurden direkt mit SEWOl erzeugt ohne logfile
Zeigen, wie der Output aussieht.
Beweismethoden??
Wann gibt es false Negatives, false Positives?
Gibt es Fehler? Wieviele? Laufzeit? Bild vom Output

Wie werden die Sicherheitsziele behandelt?

Kapitel 7

Ausblick

Wie kann man die restlichen Constraints implementieren
Compliance Checking:
Cannot do und Must execute vergleichen.
Optimierung??
Vergleich mit verwandter Arbeit

Anhang A

Weitere Beispiele für die Benutzung der Grammatik

Als Orientierungshilfe zum Erstellen eigener Regeln werden hier ein paar weitere Beispiele für Regeln gezeigt, die während der Thesis nicht behandelt wurden.

Intra Instance

```
DESC "Task1 und Task2 dürfen nicht vom selben User ausgeführt werden"
IF USER executed 'Task1' THEN USER cannot execute 'Task2'
```

```
DESC "Wenn etwas bestellt wurde, sollte es innerhalb von 3 Tagen verschickt werden."
IF 'receive order' succeeded AND ...

DESC "30 Tage, nachdem das Gewinnspiel begonnen hat,
darf kein neuer Teilnehmer mehr angenommen werden."

SET 'asd' is related to 'asdas'
DESC "Mitarbeiter aus dem Unternehmen dürfen nicht an dem Gewinnspiel teilnehmen"
IF P1 executed...

DESC "task1 und task2 müssen von verschiedenen Gruppen erledigt werden."
IF role R executed 'task1'
THEN role R cannot execute 'task2'
```

Inter Instance

Inter Process

Anhang B

Argumente und Konfiguration

Input Dateien für die Constraints: Wo sind sie? Wie kann man mehrere einlesen? (Dadurch sollen die Regeln wiederverwendbar werden)
Kann man vielleicht eine textdatei anlegen, wo die Inputfiles drinstehen? Input Dateien MXML - wo sind sie? Wird der ganze Ordner eingelesen oder gibt man ein bestimmtes File an?
Output Logger - Konsole oder File. Einstellung Level
Unterscheiden zwischen Parser Logger und Visitor Logger?
Konfigurationsdatei für die vorkommenden Events
Output Ordner

Parameter	Argument	Default	Beschreibung
–constraintfolder	abs oder rel Pfad zum Ordner	rulefiles	..
–logfolder	abs oder rel Pfad zum Ordner	logfiles	..
–outputfolder	abs oder rel Pfad zum Ordner	prologfiles	..

Tabelle mit allen möglichen Parametern

Tabelle B.1: Kommandozeilenparameter

Anhang C

Anleitung zur Verwendung

Benötigte Pakete

1. ANTLR 4.5 ...
2. SEWOL 1.0.0
   (a) TOVAL 1.0.0
   (b) JAGAL 1.0.0

Es wurde auf folgendem System getestet

1. java7 .. 8??
2. SWI-Prolog 6
3. Linux Ubuntu 12....

Wie startet man das programm?

Anhang D

Grammatik im BNF Format

<file>	::=	(<define>)* (<explicitSetting>)* (<assignment>)* <EOF>
<define>	::=	<def-symbols> <clause> ”(”<arg-type> (”,”<arg-type>)* ”)”
<explicitSetting>	::=	<set-symbols> <settableClauses> (<konj> <settableClauses>)*
<settableClauses>	::=	<extern> | <specification> | <definedClause>
<assignment>	::=	<if> <assignmentBody> <then> <assignmentHead>
<description>	::=	<desc> <constant>
<assignmentBody>	::=	(<neg>)? <clauses> (<konj> (<neg>)? <clauses>)*
<assignmentHead>	::=	<enforcement> | <definedClause>
<clauses>	::=	<atoms> | ”(”<atoms> (<disj> <atoms>)* ”)”
<atoms>	::=	<specification>
		| <status>
		| <comparison>
		| <conditional>
		| <extern>
		| <definedClause>
<definedClause>	::=	<clause> ”(”(<const> | <var>) (”,”(<const> | <var>))* ”)”
<var>	::=	<uppercase letter> | <variable><character>
<lowercase letter>	::=	a | b | c | ... | x | y | z
<uppercase letter>	::=	A | B | C | ... | X | Y | Z
<numeral>	::=	<digit> | <numeral><digit>
<digit>	::=	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<character>	::=	<lowercase letter> | <uppercase letter> | <digit> | <special>
<special>	::=	+ | - | * | / | || | : | . | ? | |#| $ | &
<string>	::=	<character> | <string><character>

Literaturverzeichnis

[1] Christian Wolter and Andreas Schaad. Modeling of task-based authorization constraints in bpmn. In Gustavo Alonso, Peter Dadam, and Michael Rosemann, editors, *Business Process Management*, volume 4714 of *Lecture Notes in Computer Science*, pages 64–79. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-75182-3.

[2] Janice Warner and Vijayalakshmi Atluri. Inter-instance Authorization Constraints for Secure Workflow Management. In *Proceedings of the Eleventh ACM Symposium on Access Control Models and Technologies*, SACMAT '06, pages 190–199, New York, NY, USA, 2006. ACM. ISBN 1-59593-353-0.

[3] Günter Müller and Rafael Accorsi. Why are business processes not secure? In Marc Fischlin and Stefan Katzenbeisser, editors, *Number Theory and Cryptography*, volume 8260 of *Lecture Notes in Computer Science*, pages 240–254. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-42000-9.

[4] Sewol documentation. http://doku.telematik.uni-freiburg.de/sewol. Accessed: 2015-07-30.

[5] Terence Parr. *The Definitve ANTLR Reference - Building Domain-Specific Languages*. The Pragmatic Programmers, 2013.

[6] Maria Leitner, Juergen Mangler, and Stefanie Rinderle-Ma. Definition and Enactment of Instance-Spanning Process Constraints. In X.Sean Wang, Isabel Cruz, Alex Delis, and Guangyan Huang, editors, *Web Information Systems Engineering - WISE 2012*, volume 7651 of *Lecture Notes in Computer Science*, pages 652–658. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-35062-7.