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:
21. August t2015

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

TODO: Muss hier auch eine genaue Quelle angegeben werden?

ALBERT-LUDWIGS-UNIVERSITÄT FREIBURG
Institut für Informatik und Gesellschaft
Abteilung Telematik

Kurzfassung

Bachelor of Science

Inter-Instance Constraints

by Regina König

Compliance Checker werden in Unternehmen dazu eingesetzt, um Geschäftsprozesse auf Basis ausgeführter Aktivitäten in Form von Logs daraufhin zu prüfen, ob Regeln wie das Separation of Duty Prinzip tatsächlich eingehalten wurden. Dadurch sollen Fehler von Angestellten und möglicher Betrug erkannt werden. Allerdings werden die Prozessinstanzen unabhängig voneinander überprüft. Neuere Forschungen weisen darauf hin, dass dies nicht ausreicht, und es ebenso notwendig ist, Regeln zum Arbeitsablauf zwischen mehreren Instanzen zu definieren und die Aktivitäten daraufhin zu analysieren. In dieser Arbeit wird eine Regelbeschreibungssprache entwickelt, die auch Instanz-übergreifende Regeln definieren lässt und erläutert, wie Logs auf die Einhaltung dieser Regeln getestet werden können.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Symbole

Definition von Prozessen	
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
$\tau \in \mathcal{T}$	Zeitpunkt
Θ	Menge aller Zeitintervalle
$\theta \in \Theta$	Zeitintervall
E	Menge aller Informationen in Event Logs
Prädikate	
L	Menge aller Prädikate außer den Kopfprädikaten
H	Menge der Kopfprädikate
K	Menge aller Disjunktionen von Prädikaten aus L
N	Menge aller Negationen von Konjunktionen von Prädikaten aus L
Argumenttypen	
UT	Variablen und Konstanten über U
RT	Variablen und Konstanten über R
TT	Variablen und Konstentn über T
WT	Variablen und Konstanten über W und W_i
TP	Variablen und Konstanten über $\mathcal{T}$
TS	Variablen und Konstanten über Θ
ET	Variablen und Konstanten über Events

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 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 [1] [2]. 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 [3] [4] 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 auch die Möglichkeit, Aufgaben wieder abzugeben oder an einen anderen Mitarbeiter weiterzuleiten.

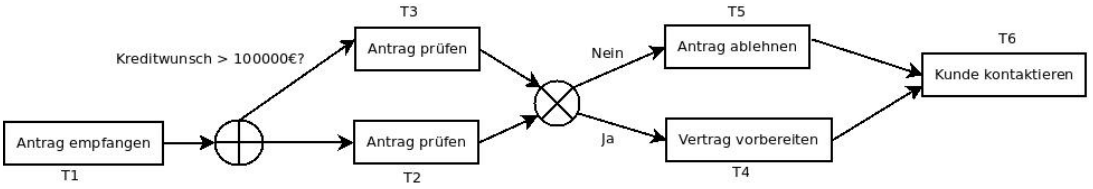


Abbildung 1.1: Arbeitsablauf in einer Bank, nachdem ein Kreditantrag eingegangen ist.

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 in mehr als drei Arbeitsabläufen gemeinsam an T1 und T2 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 eingehalten:
kursive Begriffe bezeichnen Fachbegriffe
Definitionen und Abkürzungen werden beim ersten Vorkommen fett geschrieben
`Blockschrift kennzeichnet Pseudocode, Klassennamen, Code und Programmbefehle`

Diese Arbeit setzt ein elementares Verständnis von Logik und logischer Programmierung voraus, da hier nicht näher darauf eingegangen wird.

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 und der Aufbau des Programms vorgestellt. In Kapitel 6 wird ein Beispiel zur Funktionsweise des Programms präsentiert. Schließlich wird die Arbeit in Kapitel 7 noch einmal zusammengefasst.

Kapitel 2

Verwandte Arbeit

Dieses Kapitel soll sowohl einen Überblick über den aktuellen Forschungsstand zu Instanz-übergreifenden Regeln geben, als auch verschiedene Ansätze zur Implementierung von Compliance Checkern vorstellen.

2.1 Untersuchung Instanz-übergreifender Regeln

Bereits 1999 untersuchten Bertino et al. [5], wie man Einschränkungen auf die Authorisierung von Nutzern und Rollen zu Aktivitäten in Geschäftsprozessen als Klauseln in einem logischen Programm darstellen kann. Die Klauseln, die vorgestellt werden, stellen keine End-User Sprache dar, sondern repräsentieren die Einschränkungen intern in einer Regeldatenbank. Der Körper einer Regel besteht dabei aus einer Konjunktion von logischen Literalen. Da ihr Ansatz die Authorisierung bei *Workflow Management Systemen* (**WfMS**) erweitern soll, unterscheiden sie zwischen statischen und dynamischen Einschränkungen. Statische Einschränkungen können vor Beginn eines Arbeitsablaufs ausgewertet werden. Dynamische Regeln aktualisieren die Datenbank nach jedem Ausführen einer Aktivität.

2006 griffen Warner et al. [3] diesen Ansatz auf und erweiterten ihn um eine Notation, die Instanz-übergreifende Einschränkungen erlaubt, da sie erkennen, dass der vorherige Ansatz nicht ausreichend ist, um alle Betrugsmöglichkeiten abzudecken. Ebenso erlaubt die neu entwickelte Notation Einschränkungen zu definieren, die *alle* bisherigen Instanzen in die Untersuchung mit einbeziehen, und somit nicht an ein Arbeitsablauf-Schema gebunden sind (*Inter-Process*).

Diese Arbeit greift den Ansatz von [3] zur Darstellung als Klauseln in einem logischen Programm auf, um die Regeln und Logs in einer internen Wissensbasis zu repräsentieren. Um die Eingabe der Regeln für den Anwender möglichst intuitiv zu gestalten, wird dafür eine eigene Eingabesprache entwickelt. Im Unterschied zur Arbeit von [3] ist das Ziel dieser Arbeit die Untersuchung von Logs *nach* Abschluss der Prozesse. Aus diesem Grund werden einige Prädikate und Regeln weggelassen, die bei [3] zur Berechnung während der Laufzeit notwendig sind. Zusätzlich werden arithmetische Operationen auf Variablen und Disjunktion erlaubt.

2012 stellen Leitner et al. in ihrer Arbeit [4] das "**I**dentification and **U**nification of **P**rocess **C**onstraints (**IUPC**) compliance Framework"vor, welches *Intra-Instance*, *Inter-Instance*, *Inter-Process* und *Inter-Organizational* Regeln bei Geschäftsprozessen erzwingt. *Inter-Organizational* Regeln beziehen sich auf Prozesse, die von mehreren Organisationen ausgeführt werden. In dieser Arbeit wird dieser Fall nicht berücksichtigt, da die Logs nicht zwingend auf eine Organisation beschränkt sein müssen. Des weiteren basieren die Berechnung zur Einhaltung der Regeln in [4] auf *Linear Time Logic* (**LTL**) und *Compliance Rule Graphs*(**CRG**). Diese Abschlussarbeit arbeitet mit Fakten und Regeln in einem logischen Programm.

2.2 SCIFF - SCIFFChecker - ProM

SCIFF ist ein *Framework* zur Überprüfung von Logs bezüglich der Einhaltung von Regeln. Es basiert auf Forschungergebnissen des SOCS (**S**ocieties **O**f **C**omputee**S**) Projekts und wurde am **ENDIF** an der Universität Ferrara und am **DEIS** an der Universität Bologna entwickelt. *SCIFF* bildet positive und negative Erwartungen aus einer Spezifikation auf logische Prädikate ab, um sie dann gegen eine konkrete Ausführung, die in Form von MXML Logs geliefert wird, zu testen. Diese *Traces* werden dann als Regelkonform oder nicht Regelkonform bewertet. SCIFF läuft mit SICSTUS Prolog oder SWI-Prolog. Es wird auch als Plugin für das *ProM Framework* in Form des SCIFFCheckers angeboten. Obwohl diese Arbeit hauptsächlich auf [3] basiert, weist sie viele Parallelen zu SCIFF auf. Jedoch untersucht SCIFF nur einzelne Instanzen von Prozessen. In dieser Arbeit wird ein Programm entwickelt, das die Untersuchung von Regeln zwischen mehreren Instanzen erlaubt. [6] [7][8]

2.3 Weitere Ansätze

Neben dem Einsatz von logischer Programmierung zur Auswertung von Logs in Bezug auf Einhaltung von Regeln gibt es weitere Ansätze, die hier kurz skizziert werden. Casati et al. stellen in ihrer Arbeit [9] ein Framework vor, das aktive Datenbank Systeme ausnutzt, um Instanz-, Zeit- und Verlaufsabhängige Authorisierungsregeln aufzustellen. Die Regeln werden dabei als ECA (**E**vent - **C**ondition - **A**ction) Regeln aufgestellt. In [10] vergleicht Wagner neben weiteren Beschreibungssprachen wie *UML*, *OCL*, wie sich Prozessregeln in *Prolog* und *SQL* darstellen lassen. *SQL* Regeln sind dabei sehr interessant, weil gleichzeitig eine SQL Datenbank dazu verwendet werden kann, um die Regeln auszuwerten.
Der LTLChecker ist ein weiteres Plugin für das *ProM Framework* und verifiziert *MXML* Logs gegenüber einer Spezifikation mithilfe von *LTL* (**L**inear **T**emporal **L**ogic). [11]

Kapitel 3

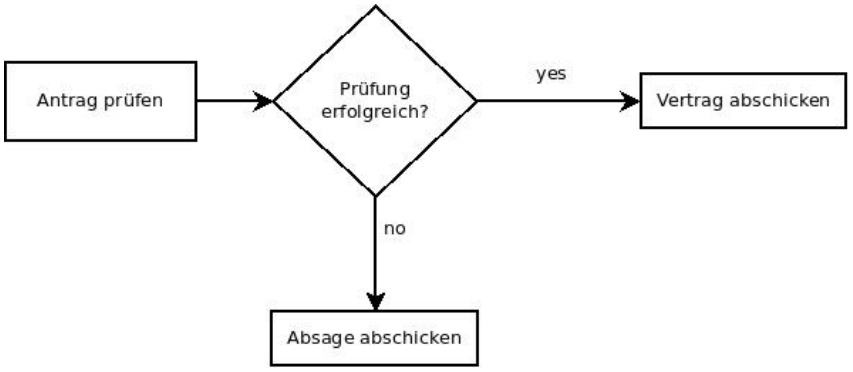
Grundlagen

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.



Ein einfaches Beispiel für das Schema eines Prozesses mit einem Start und zwei möglichen finalen Aktivitäten. Abhängig von dem Ergebnis der Prüfung wird entweder ein Vertrag vorbereitet oder eine Absage verschickt.

Abbildung 3.1: einfaches Beispiel eines Prozessschemas

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

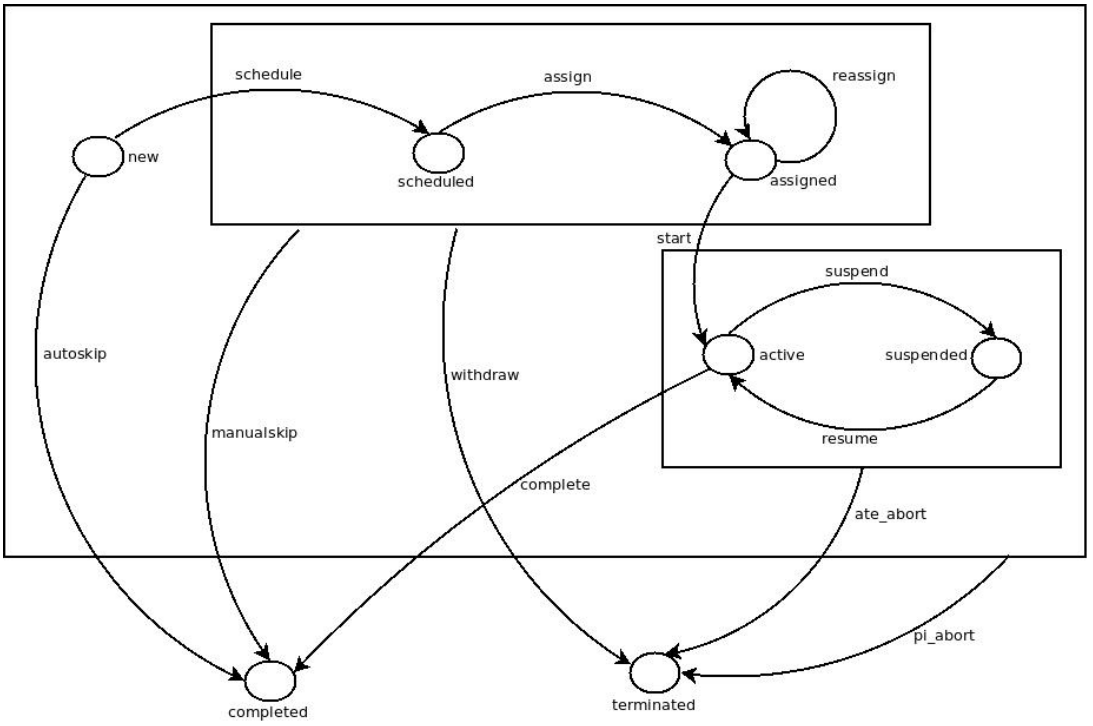
$|T_i| \subseteq |T|$ eine Teilmenge aller möglichen Aktivitäten des zugehörigen Schemas, da einzelne Aktivitäten unter Umständen aufgrund der Abhängigkeiten nicht ausgeführt werden müssen.

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. timestamp$(t_1) <$ 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*.



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.

Abbildung 3.2: Aktivitäten und Events

3.1.3 Rollenmodell und Authorisierung

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

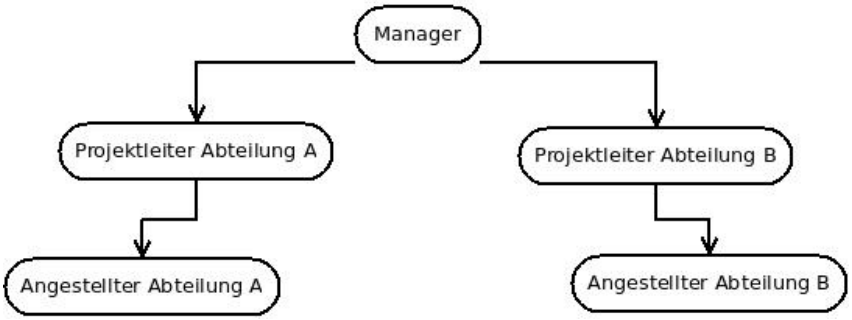


Abbildung 3.3: Beispiel Rollenmodell

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

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

und

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

welche eine n:m-Beziehung zwischen Aktivitätem und Rollen, bzw zwischen Nutzern und Rollen kennzeichnen. Das bedeutet, dass Nutzer mit Rollen in der Nutzer-Rollen Beziehung assoziiert werden und Aktivitätem mit Rollen in der Aktivitäten-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, eine Aktivität t auszuführen und U(t) die Menge aller Nutzer, die authorisiert sind, einer Aktivität t 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 'Angestellter', das bedeutet, dass der 'Projektleiter' alle Tasks ausführen darf, die der Rolle 'Angestellter' zugeordnet wurde. Die berechtigte Rolle und all ihre Elternrollen bis zur Wurzel können einem Task zugewiesen werden.

[12]

3.1.4 Zeitmodell

Das Zeitmodell ist ein Tupel $\mathbf{T} = (\mathcal{T}; \leq)$.
$\mathcal{T}$ ist eine Menge von *Zeitpunkten* τ und $\leq$ eine total Ordnung auf $\mathcal{T}$. Ein *Zeitinterval* $\theta = [\tau_a, \tau_b]$ (mit $\theta \in \Theta$ mit Θ = Menge aller Zeitintervalle) 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 $\tau_b \leq \tau_a$. [3]

TS wird als die Menge aller Zeitpunkt-Variablen und Konstanten wie 30.07.1999, 14:55 definiert und **TP** bezeichnet die Menge aller Zeitspannen wie 3 Tage 5 Stunden, 5 Monate.
Es gilt:

$$TP - TP = TS$$

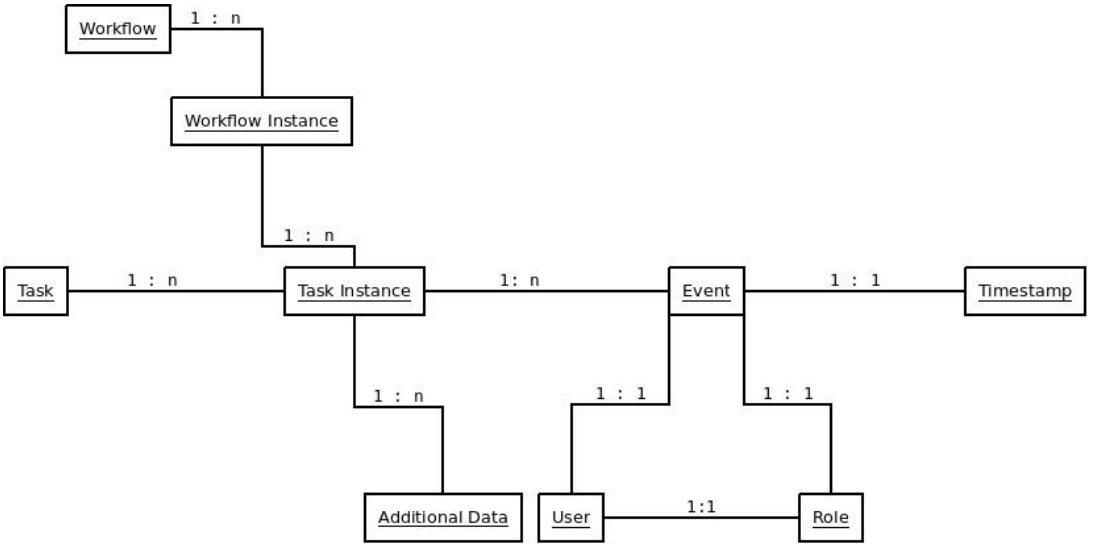
und

$$TP + TS = TP$$

Die Differenz zwischen zwei Zeitpunkten ist eine Zeitspanne (30.07.1999 - 25.7.1999 = 5D) und die Summe von einem Zeitpunkt und einer Zeitspanne ist wieder ein Zeitpunkt (25.07.1999 + 5D = 25.07.1999).

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.



Ein Prozessschema kann mehrere Instanzen haben, welche wiederrum aus Instanzen von mehreren Aktivitäten bestehen können. Die Instanz besitzt mehrere Events (*start, complete,...*), die einen eindeutigen Zeitstempel, einen eindeutigen Nutzer und eine eindeutige ausführende Rolle besitzen. Der Nutzer hat im Moment der Ausführung ebenfalls eine eindeutige Rolle. Eine Aktivität kann zudem mehrere zusätzliche Attribute haben.

Abbildung 3.4: Ontologie eines Prozesses

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.

3.2 Herleitung der Einschränkungen

3.2.1 Ein praktisches Beispiel

In diesem Kapitel werden verschiedene Arten von Einschränlungen und Regeln betrachtet. Einschränkungen und Regeln sind Erwartungen, die aus dem Verlauf von vorhergehenden Aktivitäten 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.

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 den Antrag ablehnt (T5). Unabhängig von dem Ergebnis wird der Kunde zuletzt noch einmal kontaktiert (T6).

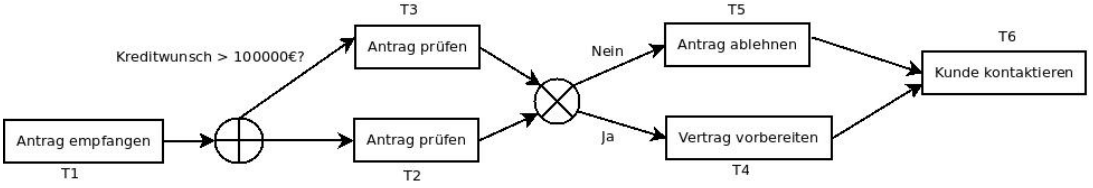


Abbildung 3.5: Bearbeitung eines Kreditantrags in der Bank

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

(C1) Der Kontakt mit Kunden (T1 und T6) muss durch den Kundenberater erfolgen.
(C2) 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).
(C3) Den Antrag annehmen (T1) und den Antrag prüfen(T2, T3) sollten von verschiedenen Personen erledigt werden (4 Augen Prinzip).
(C4) Ferner sollten auch die zwei Prüfungen von verschiedenen Mitarbeitern vollzogen werden. T3 muss durch den Bank Manager erfolgen.
(C5) Es dürfen keine Anträge von Verwandten geprüft werden.

(C6) Ein Mitarbeiter darf bei dem selben Kunden höchstens Kredite bis 100000 Euro prüfen.
(C7) Ein Mitarbeiterpaar darf höchstens 3mal gemeinsam an T1 und T2 arbeiten.

(C8) Wenn ein Mitarbeiter 5x einem Task zugewiesen wird und ihn dann abbricht, darf er nicht mehr an dem Task arbeiten.
(C9) Um zu verhindern, dass Fehler durch Überlastung passieren, darf jeder Mitarbeiter am Tag höchstens 100 Tasks bearbeiten.

3.2.2 Arten von Einschränkungen

Generell kann man Einschränkungen bezüglich des Verlaufs, zeitliche Einschränkungen, Abhängigkeiten von Parametern und Authorisierungsregeln unterscheiden. Jedoch sind sie nicht klar voneinander getrennt, sondern oft innerhalb einer Regel vermischt. Im folgenden werden die Arten von Regeln kurz erläutert.

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

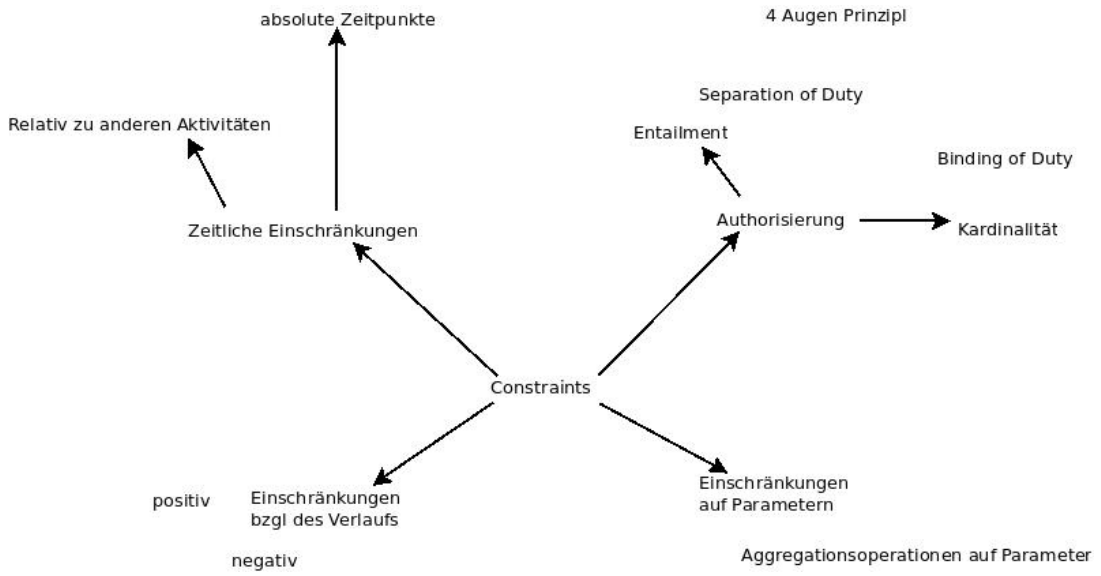


Abbildung 3.6: Typen von Einschränkungen und Regeln

Entailment Constraint

Entailment-Constraints machen Aussagen darüber, in welcher Beziehung die Ausführenden zweier Aktivitäten zueinander stehen. *Rollenbasierte Entailment-Constraints* haben die Form $(R, \{t_1, t_2\}, pred)$ mit **R** als Teilmenge aller Rollen. Die Rollen aus R sind zugelassen, t_1 und t_2 auszuführen. *pred* ist ein Prädikat aus der Menge $\{=, \neq, <, \leq, >, \geq\}$ und gibt an, in welchem Verhältnis gemäß der Rollenhierarchie die ausführenden Rollen zueinander stehen müssen.
Nutzerbasierte Constraints $(U, \{t_1, t_2\}, pred)$ legen fest, welches Verhältnis die Nutzer in Bezug auf ihre Rollen bei der Ausführung von t_1 und t_2 haben müssen. [12] [13] [14]

Cardinality Constraints

Kardinalitätseinschränkungen haben die generelle Form (T, k, n_u, n_r) und besagen, dass eine Menge von Aktivitäten **T** von n_u verschiedenen Personen und n_r verschiedenen Rollen ausgeführt werden muss. Man unterscheidet zwischen *lokalen* und *globalen* Einschränkungen. *Lokale* Einschränkungen beziehen sich auf verschiedene Instanzen der selben Aktivität in einer Prozessinstanz. Sie haben die Form (t, k, n_u, n_r) und können zusätzlich eine Aussage darüber treffen, wieviele Instanzen einer Aktivität es geben muss (k). *Globale* Kardinalitätseinschränkungen sind Einschränkungen auf die Instanzen von einer gegebenen Menge von Aktivitäten T in der selben Prozessinstanz. [12] [13] [14]

Separation of Duty

Das Separation of Duty-Prinzip ist ein Speziallfall der Kardinalitätseinschränkungen und kann für zwei Aktivitäten t_1 und t_2 als $(\{t_1, t_2\}, 2)$ ausgedrückt werden. Das Separation of Duty Prinzip ist eine Form von Aufgabentrennung, in der kritische Aufgaben zerlegt werden und dann von verschiedenen Nutzern ausgeführt werden müssen. [1][2]

Binding of Duty

Binding of Duty - Aufgabenbindung ist das Gegenteil von *Separation of Duty*, und ist ebenfalls ein Spezialfall von *Cardinality Constraints*. Es hat die Form $(\{t_1, t_2\}, 1)$. Bei diesem Prinzip müssen zwei Aufgaben von der selben Person erledigt werden.

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.

3.2.3 Gültigkeitsbereich von Constraints

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. 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

Entwicklung einer Definitionssprache für Regeln

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

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 Konjunktion. Werden alle Bedingungen erfüllt, tritt die Regel in Kraft. Zur Vollständigkeit wird hier auch die Option mit aufgenommen, absolute Regeln ohne Bedingungen aufzustellen, die in jedem Fall gelten sollen.

Für alle Regeltypen, die in 3.2.2 gefunden wurden, ist Konjunktion von positiven Bedingungen ausreichend. Um dem Nutzer größtmögliche Freiheit zu lassen, sollte Disjunktion 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 eine Einigung geben, ob Events explizit angegeben werden müssen, oder sich Prädikate wie `timestamp(TASK)` auf ein konkretes Event bezieht.

Unter Berücksichtigung aller gestellten Forderungen sollte trotzdem eine leicht zu lernende und leicht zu verstehende Notation gewährleistet sein. Im folgenden wird die entwickelte Grammatik vorgestellt und gleichzeitig geklärt, inwiefern die hier aufgeführten Fragen und Anforderungen gelöst werden.

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' dürfen
nicht von Verwandten ausgeführt werden"
IF user USER_A executed 'Kredit beantragen' AND USER_A is related to USER_B
  THEN user 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 Konjunktion 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 Elemente aus der Menge {'started', 'completed',..} (siehe 3.1.2).
TP	Variablen und Konstanten über Zeitpunkte. Als Konstante Zeitpunkt nach ISO 8601.
TS	Variablen und Konstanten über Zeitspannen. Als Konstante Zeitspanne nach ISO 8601.
NT	Variablen und Konstanten über numerische Werte. Als Konstante sind numerische Werte eine Zahl größer Null. Negative Werte sind nicht erlaubt.

Tabelle 4.1: Argument Typen, die bei Prädikaten vorkommen können

Zeitpunkte nach ISO 8601 werden im Format JJJJ-MM-DD'T'hh:mm:ss.f , wobei f ein dezimaler Bruchteil für Sekunden beliebiger Genauigkeit ist. Die Datums und Uhrzeit-Angabe wird von einem 'T' getrennt. Die Werte werden ohne Leerzeichen notiert. Es ist erlaubt, Werte mit geringerer Genauigkeit anzugeben, jedoch darf die Angabe immer nur von rechts (mit dem kleinsten Wert beginnend) weggelassen werden. Gültige Beispiele sind `2015-08-21T11:23:45.23526` , `2015-08-21T11`, `2015-08`. Eine ungültige Angabe hingegen wäre `2015-21T23`. Hier wurde der Monat und die Uhrzeit weggelassen, obwohl dahinter noch das Datum und die Minuten standen.

Zeitspannen im ISO 8601 Format sind 'P'JJJJ'Y'MM'M'DD'D"T'hh'h'mm'm'ss.f's'. Die Zeitspanne beginnt mit einem vorangestellten 'P', dahinter folgen Werte für Jahr, Monat, Tag, Stunden, Minuten, Sekunden. Datum und Zeitangaben werden hier ebenfalls voneinander mit dem Trennsymbol 'T' abgegrenzt. Im ISO 8601 sind auch Wochenangaben erlaubt. Da diese sich aber auch durch Monate und Tage darstellen lassen, wird hier darauf verzichtet. Bei Zeitspannen dürfen auch Werte in der Mitte"weggelassen werden. Um trotzdem eindeutig identifizieren

zu können, welche Einheit eine Angabe besitzt, muss hinter jedem Wert das zugehörige Symbol stehen: **Y** für Jahr, **M** Monat, **D** Tage, **h** Stungen, **m** Minuten, **s** Sekunden. Beispiele für Zeitspannen sind `P2Y6M1DT16h35m2s` (2 Jahre 6 Monate 1 Tag 16 Stunden 35 Minuten 2 Sekunden), `P1D` (1 Tag) und `P2Y1DT16h35m2s` (2 Jahre 1 Tag 16 Stunden 35 Minuten 2 Sekunden).

Zeitzonenangaben sind nicht erlaubt.

4.2.2 Prädikate

In diesem Abschnitt werden alle Verfügbaren Prädikate vorgestellt. Es gibt 7 verschiedene Typen: *Externe Informationen, Spezifikation des Prozesses und der Authorisierung, Status, Prädikate für den Kopf einer Regel, Aggregationsprädikate, Vergleiche und arithmetische Operationen.*

Eine genaue Auflistung aller Prädikate befindet sich in Anhang A.

Externe Informationen (Tabelle A.1) 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 sie nicht aus den Logs herausgelesen werden und bei einer Anfrage immer *false* zurückgeben würden.

Spezifikationsprädikate (Tabelle A.2) bestimmen die Beziehungen und Zugehörigkeit zwischen Nutzern, Rollen und Tasks. Um korrekte Ergebnisse zu erhalten, sollten sie genauso gesetzt werden wie in der Spezifikation der Authorisierung zur Ausführungszeit des Prozesses.

Statusprädikate (Tabelle A.3) sind Aussagen über Aktivitäten. Diese können in den Bedingungen einer Regel eingesetzt werden und werden später mit den Informationen aus den Logs verglichen.

Aggregations Prädikate (Tabelle A.4) (*NUMBER, SUM, AVG, MIN, MAX*) geben einen Wert über die Aggregation von einer Variablen zurück, die in den Klauseln des Körpers dieses Prädikates stehen. `MIN OF X WHERE 'T1' completed AND timestamp of 'T1' is X IS N` berechnet das Minimum von allen Zeitpunkten, an denen die Aktivität 'T1' abgeschlossen wurde und schreibt es in die Variable N. X ist die Variable, mit der gerechnet wird und mindestens ein Mal in Körper dieses Prädikates vorkommen muss. Die beiden Literale `'T1' completed` und `timestamp of 'T1' is X` bilden den Körper. '`MIN OF` Var `WHERE` body `IS` Var ist das Aggregations-Prädikat selbst.

Vergleiche (Tabelle A.5) dienen dazu, festzustellen, in welchem Verhältnis zwei Werte zueinander stehen. Es können immer nur zwei Werte des gleichen Typs verglichen werden. Bei Gleichheitsabfragen sind alle Typen erlaubt, bei Abfragen über Größenverhätltnisse sind außer den numerischen Werten und Zeiten auch Rollen bezüglich der Hierarchie erlaubt.

Arithmetische Operationen (Tabelle A.6) sind grundsätzlich nur für numerische Werte möglich. Eine Ausnahme bildet die Summe aus einem Zeitpunkt und einer Zeitspanne (Das Ergebnis ist ein Zeitpunkt) und die Differenz aus zwei Zeitpunkten (das Ergebnis ist eine Zeitspanne). Jede einzelne arithmetische Operation muss in Klammern stehen.

Kopfprädikate bestimmen das gewünschte Resultat einer Regel. Sie machen Aussagen darüber, ob eine Aktivität von einem Nutzer/Rolle ausgeführt werden darf oder muss. Diese müssen im Kopf einer Regel stehen, und dürfen nicht im Körper vorkommen. `illegal execution` kann dazu verwendet werden, eine Ausführung als regelwidrig zu kennzeichnen, sofern die Spezifikation im Körper der Regel erfüllt wurde. (Tabelle A.7)

4.2.3 Regeln

Eine Regel besteht aus einem Körper, der die Bedingungen enthält (eingeleitet durch das Schlüsselwort 'IF') und einem Kopf (nach dem Schlüsselwort 'THEN'):

```
IF body THEN head
```

Sei
L := $\{P_1, P_2, ..., P_n\},\ n \in \mathbb{N}$ die Menge aller zuvor definierter Prädikate außer den Kopfprädikaten,
K := $\{K_1\ OR\ K_2\ OR\ ...\ OR\ K_m\},\ m \in \mathbb{N},\ K_j \in L,\ j \leq m$ die Menge aller Disjunktionen der Prädikate aus L,
N := $\{NOT(N_1\ AND\ N_2\ AND\ ...\ AND\ N_l)\},\ l \in \mathbb{N},\ N_k \in\ L,\ k \leq l\}$ die Menge der Negation aller Konjunktionen von Prädikaten aus L,
dann gilt

$$body := A_1\ AND\ A_2\ AND\ ...\ A_N,\ N \in \mathbb{N},\ A_i \in \{L \vee\ K \vee\ N\},\ i \leq N$$

Ein Körper besteht in der Regel aus `AND` - Verknüpfungen von Literalen. Ein Literal kann allerdings selbst entweder eine `OR`-Verknüpfung aus Literalen sein [1] oder die Negation von `AND` - Verknüpfungen von Literalen.
Folgendes ist somit ein valides Beispiel:

$$body = B_1\ AND\ B_2\ AND\ (B_3\ OR\ B_4)\ AND\ NOT(B_5\ AND\ B_6),\ B_i \in\ L, i \leq 6$$

Um auch absolute Regeln aufstellen zu können, ist ein leerer Körper ebenfalls erlaubt. Die Regel tritt dann in jedem Fall in Kraft:

```
IF
THEN role 'Azubi' cannot execute 'Kreditantrag prüfen'
```

Die Aufgabe 'Kreditantrag prüfen' darf in keinem Fall von einem Azubi ausgeführt werden.

Abbildung 4.2: Absolute Regel mit einem leeren Körper

Neben den eigentlichen Regeln kann man Regeln für selbst definierte Prädikate aufstellen. Dies wird im Abschnitt 4.3.1 genauer erläutert.

[1] Diese `OR` - Verknüpfung muss in einer Klammer stehen und hat damit größere Bindung als die `AND`-Verknüpfungen. Ohne Klammer sind `OR`-Verbindungen nicht erlaubt, da man die Regel einfach in zwei Regeln aufteilen kann: $body = A_1\ AND\ A_2\ OR\ A_3\ AND\ A_4$ kann man schreiben als: $body1 = A_1\ AND\ A_2$ und $body2 = A_3\ AND\ A_4$

4.3 Grammatik - Syntax und Semantik

In diesem Abschnitt werden die wichtigsten Merkmale der Definitionssprache an Beispielen erläutert. Eine genaue Definition in **BNF**-Form [2] befindet sich in Anhang E.

```
DEF same_departement(UT, UT)

SET 'Erika Haas' is related to 'Stephan Heinzmann'
SET 'Erika Haas' is related to 'Tim Müller'
SET same_departememement('Erika Haas' , 'Stephan Heinzmann')

// Einzeiliger Kommentar
/*
Mehrzeiliger Kommentar
*/
DESC "Wenn zwei Mitarbeiter verwandt sind und in der selben Abteilung arbeiten,
dürfen sie nicht gemeinsam an 'T1' uns 'T2' arbeiten."
IF user USER_A executed 'T1' AND USER_A is related to USER_B
AND same_departement(USER_A, USER_B)
THEN user USER_B cannot execute 'T2'
```

Abbildung 4.3: Beispiel für die Definition von Regeln

Zu Beginn einer Regel-Datei kann man eigene Prädikate definieren (siehe Abschnitt 4.3.1). Als nächstes können Fakten aus Prädikaten in den Tabellen A.1 und A.2 und selbst definierten Prädikaten gesetzt werden (Abschnitt 4.3.2). Im letzten Abschnitt werden die eigentlichen Regeln eingetragen (Abschnitt 4.3.3). Die drei Teile müssen in dieser Reihenfolge stattfinden und können nicht gemischt werden.

4.3.1 Definition eigener Prädikate

Es kann nötig werden, 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 "DEF" zu Beginn der Regeldefinitionen zu setzen. In der Definition wird der Typ der Argumente festgelegt. Verfügbar sind 'UT', 'RT', 'TT', 'WT', 'TS', 'TP', 'NT' und 'STRING_VAR'. 'STRING_VAR' bezeichnet dabei einen beliebigen String. 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 'SET' setzen oder man kann ihnen eine Regel erstellen. Damit die Prädikate für den Parser und die weitere Verarbeitung eindeutig sind, sind sie an eine spezielle Form gebunden: <predicatename>(ARGType (, ARGType)*). <predicatename> kann nur aus Kleinbuchstaben a-z und Unterstrichen _ (nicht am Anfang des Namens) bestehen.

[2] Backus-Naur-Format

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

Definition eigener Prädikate. *suspicious* wird mit `SET` gesetzt. *skipped* und *aborted* erhalten eine Regel, wann sie gelten.

Abbildung 4.4: Definition eigener Prädikate.

Die definierten Prädikate dienen nur zur Gruppierung von Nutzern oder Rollen bzw um Abkürzungen für Beziehungen zwischen Aktivitäten herzustellen. Sie werden bei der späteren Auswertung *nicht* gegen die Logeinträge verglichen und sind somit nicht dazu geeignet, weitere Erwartungstypen zu definieren.

4.3.2 Setzen von Fakten

Fakten aus den Tabellen A.1 (externe Informationen) und A.2 (Spezifikation des Prozesses) und selbst definierte Prädikate werden nach dem Definieren eigener Prädikate und vor dem Notieren der Regeln gesetzt. Fakten können keine Variablen enthalten.

```
SET 'Manager' dominates 'Sekretärin'
SET 'Tom Hoffman' belongs to role 'Administrator'
```

Spezifikation, dass die Rolle 'Manager' die 'Rolle' Sekretär dominiert sowie Zuweisung der Rolle 'Administrator' zu 'Tom Hoffman'.

Abbildung 4.5: Setzen von Fakten.

4.3.3 Notation bei Regeln

Die Regeln repräsentieren die Einschränkungen, welche in einem Prozess gelten. Eine Regel besteht aus einem Kopf und einem Körper. Der Körper wird durch 'IF' und der Kopf durch 'THEN' eingeleitet. Bezüglich der Leerzeichen und Zeilenumbrüche exisitieren keine Einschränkungen. Insofern sie nicht direkt in einem Literal gesetzt werden, können sie an beliebiger Stelle platziert werden. Bei Literalen ist genau ein Leerzeichen zwischen den Wörtern einzuhalten. Abweichungen werden nicht toleriert.

Alle Klauseln (sowohl im Kopf als auch im Körper) können als Argumente entweder Konstanten oder Variablen enthalten. Konstanten sind entweder *Strings* (dann müssen sie in jedem Fall von zwei einfach Anführungsstrichen umschlossen sein - zB 'Simon Pabst', 'start'), numerische Werte (Zahlen n $\geq$ 0), Zeitpunkte oder Zeitstempel (siehe Abschnitt 4.2.1). Variablen müssen mit einem Großbuchstaben beginnen. Dannach können sie alle Zeichen [A-Za-z0-9_]enthalten. Es ist zu beachten, dass reservierte Schlüsselwörter als Variable nicht erlaubt sind (Abschnitt 4.3.8). Werden die selben Variablen in unterschiedlichen Klauseln eingesetzt, dann entsprechen sie dem selben Wert. Unterschiedliche Variablen *können* bei der Auswertung den selben Wert annehmen. Sollte es ausdrücklich gewünscht sein, unterschiedliche Werte für unterschiedliche Variablen zu erhalten, empfiehlt es sich, das Prädikat VAR != VAR zu setzen.

Die definierten Variablen gelten innerhalb einer Regel. Man kann die selben Variablen in verschiedenen Regeln verwenden, ohne dass sie sich auf den selben Wert beziehen müssen.

```
IF user A executed 'T1' AND user B executed 'T2' AND A != B
    THEN user A must execute 'T3'
```

Abbildung 4.6: Unterschiedliche Werte für unterschiedliche Variablen

Zu einer Regel kann eine Beschreibung mittels `DESC "Beschreibung"` oberhalb der Regel hinzugefügt werden. Obwohl es nicht obligatorisch ist, wird es dennoch empfohlen, da diese Beschreibung bei der späteren Auswertung der Logs im Falle gefundener Regelverletzungen dem Ergebnis hinzugefügt wird, und es dem Anwender erleichtert, die Verletzungen den definierten Regeln zuzuordnen.

Es ist erforderlich zu beachten, dass Konsistenz in Bezug auf die Kontextspezifische Notation gilt (siehe Abschnitt 4.3.4 - Spezifikation des Kontexts). Es ist nicht erlaubt, verschiedene Notationen innerhalb einer Regel zu mischen. Die Regeln selbst dürfen allerdings verschiedene Geltungsbereiche besitzen.

4.3.4 Spezifikation des Kontexts

Wie in Abschnitt 3.2.3 festgestellt, kann eine Regel drei Spannweiten der untersuchten Aktivitäten annehmen. Im *Intra-Instance* Bereich werden Aktivitäten jeweils aus der selben Prozessinstanz betrachtet. Jede Prozessinstanz wird einzeln untersucht. Bei *Inter-Instance* Regeln werden alle Aktivitäten aus aus dem selben Prozessschema mit einbezogen. Es wird nicht beachtet, aus welcher Instanz sie stammen. *Inter-Process* Regeln betrachten alle Aktivitäten, unabhängig davon, zu welchem Prozessschema sie gehören.

Folgende Konventionen herrschen in der Notation, um den Kontext einer Regel zu definieren:

Intra-Instance Aktivitäten sind einfache Konstanten oder Variablen: `'Auftrag prüfen'` oder `Task_A`

Inter-Instance Aktivitäten bestehen aus zwei Teilen: <taskName>. <processInstance>. Sowohl <taskName> als auch <processInstance> können unabhängig voneinander Variablen oder Konstanten sein: `Task_A.PI_1` oder `'Auftrag annehmen'.PI_2`.

Inter-Process Aktivitäten sind dreiteilig: <taskName>. <processName>. <processInstance>

```
// Intra-Instance - wenn Mark T1 ausführt, darf Tom nicht mehr an T3 arbeiten
IF user 'Mark' executed 'T1'
    THEN user 'Tom' must execute 'T3'

// Inter-Instance - ein Mitarbeiter darf höchstens 3 mal an T1 arbeiten
IF NUMBER WHERE user A executed 'T1'.TID_1 is N AND N > 3
    THEN user A cannot execute 'T1'.TID_2

// Inter-Process - ein Nutzer darf 'T3' höchstens 100 mal ausführen
IF NUMBER WHERE user A executed 'T3'.T_1.TID_1 is N AND N > 100
    THEN user A cannot execute 'T3'
```

Abbildung 4.7: Beispiele für verschiedene Geltungsbereiche

4.3.5 Events

Obwohl es zu erhöhtem Aufwand führt, den Eventtyp zu den in einer Regel enthaltenen Aktivitäten explizit anzugeben, wurde beschlossen, den Eventtypen für Prädikate wie zB `user U executed T` oder auch `timestamp of T is N` offenzulassen, da somit größtmögliche Freiheit gewährleistet wird.

Es gibt vier vordefinierte Prädikate, welche den Eventtyp ausdrücken: `UT is assigned to TT`, `TT aborted`, `TT completed` und `TT started`. Für alle weiteren Events ist das allgemeine Prädikat `eventtype of TT is ET` zu verwenden.

Sollte der Eventtyp nicht angegeben werden, ist mit multiplen Ergebnissen zu rechnen, da für jede Aktivität jedes Event einzeln betrachtet wird. Weitere Nachteile entstehen dadurch nicht.

4.3.6 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, auf mehrere Regeln aufzuteilen. Jedoch kann es verwendet werden, um Kardinalitätsaussagen einfacher darstellen zu können. Um die Stärke der Bindung deutlich zu machen, müssen Disjunktionen in einer Klammer stehen. Disjunktion ohne umgebende Klammern ist nicht erlaubt. Disjunktionen von Konjunktionstermen sind nicht erlaubt.

Beispiel: Anzahl an Rollen muss mindestens 2 sein, die an 3 Tasks gearbeitet haben.

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

Abbildung 4.8: 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 4.9: Disjunktion im Kopfbereich ist nicht erlaubt. Es müssen zwei getrennte Regeln erstellt werden.

4.3.7 Kollaborateure und kritische Aufgabenpaare

Die beiden Prädikate `UT is collaborator of UT` und `critical_task_pair(TT,TT)` gehören zueinander. Kritische Aufgabenpaare müssen mit `SET` gesetzt werden. Kollaborateure sind Nutzerpaare, die an den kritischen Aufgabenpaaren gemeinsam gearbeitet haben.
`UT is collaborator of UT` muss vom Anwender *nicht* exlipzit gesetzt werden, sondern wird intern im Analyseprogramm berechnet.

4.3.8 reservierte Schlüsselwörter

Variablen müssen immer mit einem Großbuchstaben beginnen. Folgende Schlüsselwörter sind jedoch nicht erlaubt:
SET, IF, THEN, NOT, AND, OR, DEF, DESC, UT, RT, TT, WT, TS, TP, NT, STRING_VAR, WHERE
Wörter, die mit **GENERATED** oder **RULEID** beginnen.

4.4 Konkretes Beispiel

Die Regeln in Abschnitt 3.2.1 können mit der entwickelten Definitionssprache wie folgt ausgedrückt werden:
TODO aktualisieren, wenn Programm fertig

```
SET related, critical task pairs,...
SET critical_task_pair('T2', 'T3')

TODO sortieren, was zu welchem Kontext gehört.
// C1
```

```
DESC "T1 muss vom Kundenberater ausgeführt werden"
IF
    THEN role 'Kundenberater' must execute 'T1'

DESC "T6 muss vom Kundenberater ausgeführt werden."
IF
    THEN role 'Kundenberater' must execute 'T6'

// C2
DESC "T6 sollte 3 Tage nach T1 erfolgen"
IF 'T1' started AND timestamp of 'T1' is N1
  AND 'T6' started AND timestamp of 'T6' is N2
  AND N2 > ( N1 + P3D )
    THEN illegal execution

// C3 TODO funktioniert das hier überhaupt? Es muss sich immer auf einen Task beziehen
DESC "Ein Mitarbeiter darf pro Tag höchstens 100 Tasks bearbeiten"
IF NUMBER WHERE (
  user U executed Task_B
  AND MIN OF X WHERE ( timestamp of Task_B IS X) IS N1
  AND MAX OF Y WHERE (timestamp of Task_B IS Y) IS N2
  AND ( N2 - N1 ) = 1D )
IS N
AND N > 100
    THEN user U cannot execute Task_B

// C4
DESC "T1 und T2 muss von verschiedenen Personen erledigt werden"
IF user U executed 'T1'
    THEN user U cannot execute 'T2'

DESC "T1 und T3 muss von verschiedenen Personen erledigt werden"
IF user U executed 'T1'
    THEN user U cannot execute 'T3'

// C5
DESC "T2 und T3 müssen von verschiedenen Personen erledigt werden"
IF user U executed 'T2' AND user U executed 'T3'
    THEN illegal execution

DESC "T3 muss vom Bank Manager durchgeführt werden"
IF
  role 'Manager' must execute 'T3'

// C6 TODO das passt auch nicht, oder doch?
IF NUMBER WHERE (user U executed T AND T aborted ) IS N
  AND N > 5
  THEN user U cannot execute T

// C7
DESC "Es dürfen keine Anträge von Verwandten bearbeitet werden"
IF user U executed T AND attribute 'customer' of T is C
```

```
  AND U is related to C
  THEN illegal execution

// C8 TODO
DESC "Mitarbeiter dürfen höchstens 3 mal gemeinsam an T2 und T3 arbeiten"
IF user U1 executed 'T2' AND NUMBER WHERE (U1 is collaborator of U2) IS N
  THEN U2 cannot execute 'T3'

// C9
DESC "Ein Mitarbeiter darf bei dem selben Kunden höchstens Kredite
bis 100000 Euro prüfen"
IF SUM OF X WHERE (
  user U executed 'Antrag prüfen'.P1
  AND 'Antrag prüfen'.P1 completed
  AND attribute 'loanamount' OF T.P1 is X
) IS N
AND N > 100000
    THEN user U cannot execute 'Antrag prüfen'.P2
```

Abbildung 4.10: 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.

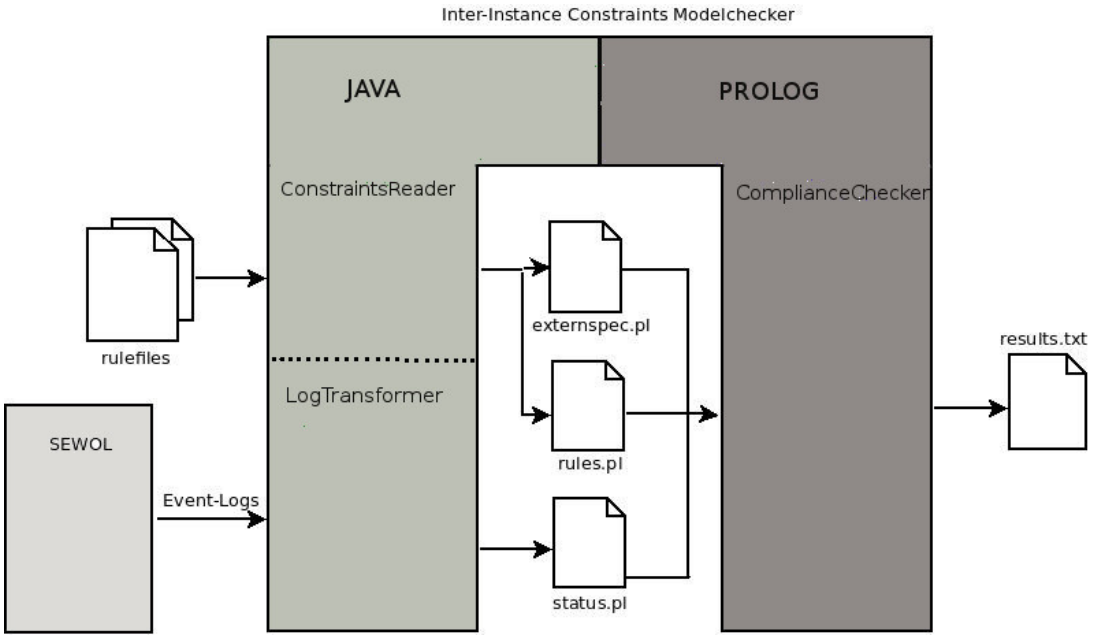


Abbildung 5.1: Aufbau des IICMCheckers

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 [15] 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. [15]

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.[16]

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.

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

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

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(1, 'amount', '3000Euro')

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 einfache Anführungszeichen) gespeichert.

Tabelle 5.2: Wissensbasis

5.2.2 Übersetzung der Regeln

Der Parser und Listener für die Grammatik wurde mit ANTLR 4.5 [16] 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.
Der `ExternAndSpecificationContainer` enthält alle Klauseln, die in den Rulefiles gesetzt werden. Es sind nur Prädikate aus den Tabellen A.1 und A.2 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 A.7 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.

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

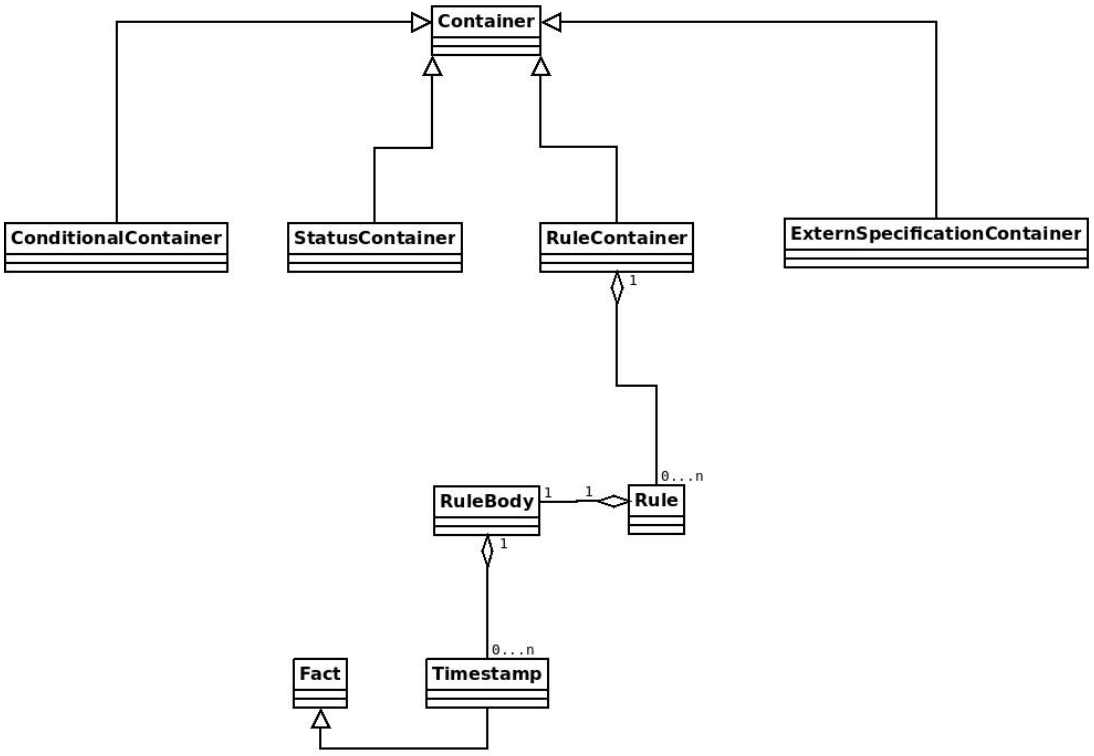


Abbildung 5.2: Interne Datenstruktur
TODO Grafik fertig machen, wenn Programm fertig

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 Ergänzungen.

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 Aktivitäten 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. Aus diesem Grund werden Intra-Instanz Regeln um Klauseln erweitert, die jeder Aktivität die selbe Prozessinstanz aufzwingen.

Zeitpunkte und Zeitspannen werden in ms nach 1970 übersetzt um arithmetische Operationen zu ermöglichen.

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.

Bevor die eigentliche Untersuchung beginnt, werden zuerst noch einige implizierte Klauseln zu der Prolog-Datenbank hinzugefügt. Es werden alle *collaborators* berechnet, die an critical_task_pairs gearbeitet haben, weitere Beziehungen aus dem Rollenmodell und Verwandschaften ermittelt.

Schließlich wird für jede Regel geprüft, ob es eine Klausel gibt, die sie bricht.
TODO vernünftiger erklären

Im Anhang F befindet sich der Pseudocode für die Analyseprozedur.

5.2.4 Darstellung der Ergebnisse

Sollte eine Aktivität im Log gefunden werden, die eine Regel bricht, wird der User/Rolle mit Task Namen und Beschreibung der Regel in *results.txt* geschrieben.

```
Modelchecker startet...
Searching for illegal executions by user...

"blabla"
Illegal execution found:
User Mark executed Task 10

Searching for illegal executions by role...
Searching for missed executions by user...
Searching for missed executions by role...
Searching for illegal executions...
Modelchecker finished...
```

TODO aktualisieren, wenn Programm fertig

Abbildung 5.3: Beispiel Output

Kapitel 6

Auswertung

6.1 Evaluation

Sämtliche hier vorgestellte Regeln wurden mit speziell erstellen Logs in SEWOL erfolgreich getestet. Es wurden alle Regelverletzungen gefunden und keine *false negatives* vermerkt.

Im Folgenden wird ein vereinfachtes Beispiel für den Analysevorgang vorgestellt und der genaue Ablauf der Auswertung besprochen. Gegeben seien ein Log und eine Regelspezifikation.

caseID	Task	User	Role	Timestamp	EventType	DataAttributes
0	Antrag empfangen	Mark	Kundenberater	2015-08-03T10:23:14	start	()
0	Antrag empfangen	Mark	Kundenberater	2015-08-03T11:47:23	complete	()
0	Antrag prüfen	Jenny	Sekretär	2015-08-04T07:02:36	start	()
0	Antrag prüfen	Jenny	Sekretär	2015-08-04T07:02:03	complete	()

Tabelle 6.1: Einfacher Log

```
SET 'Manager' dominates 'Abteilungsleiter'
SET 'Abteilungsleiter' dominates 'Kundenberater'
SET 'Abteilungsleiter' dominates 'Sekräter'

DESC "Die Rolle, die Antrag prüfen ausführt, muss die Rolle,
die den Antrag annimmt, dominieren"
IF role R1 executed 'Antrag empfangen' AND 'Antrag empfangen' completed
AND (R1 dominates R2 OR R1 = R2)
  THEN R2 cannot execute 'Antrag prüfen'
```

Abbildung 6.1: Einfache Regelspezifikation

Phase 1: Einlesen des Logs
Der Log wird vom LogTransformer in eine interne Wissensbasis übersetzt.

```
task_workflow(0,0),task_workflow(1,0),task_workflow(2,0),task_workflow(3,0),
task_name(0,'Antrag empfangen'),task_name(1,'Antrag empfangen'),
task_name(2,'Antrag prüfen'),task_name(3,'Antrag prüfen'),executed_user(0,'Mark'),
executed_user(0,'Mark'),executed_user(1,'Mark'),executed_user(2,'Jenny'),
executed_user(3,'Jenny'),executed_role(0,'Kundenberater'),
executed_role(1,'Kundenberater'),executed_role(2,'Sekretär'),
executed_role(3,'Seketär'), timestamp(0,TODO),timestamp(1,TODO),
timestamp(2,TODO),timestamp(3,TODO),eventtype(0, 'start'),eventtype(1, 'complete'),
eventtype(2, 'start'),eventtype(3, 'complete')
```

Abbildung 6.2: Beispiel Wissensbasis

Phase 2: Einlesen der Regeln
Aus der Regelspezifikation werden die gesetzten Fakten und Regeln übersetzt.

```
dominates('Manager','Abteilungsleiter'),
dominates('Abteilungsleiter', 'Kundenberater'),
dominates('Abteilungsleiter', 'Sekretär')

role_cannot_execute(R2, GENERATED1) :-
    task_name(GENERATED1, 'Antrag prüfen'),
    role_executed(R1, GENERATED2),
    task_name(GENERATED2, 'Antrag empfangen'),
    eventtype(GENERATED2, 'complete'),
// Die folgenden zwei Zeilen sollen sicherstellen,
// dass nur Aufgaben aus der selben Prozessinstanz betrachtet werden.
    task_workflow(GENERATED1, GENERATED3),
    task_workflow(GENERATED2, GENERATED3),
    ( dominates(R1, R2) ; R1 = R2).
```

Die Regel wurde hier auch vereinfacht und enthält in der originalen Version weitere Verfahren zum ...

Abbildung 6.3: Beispiel Wissensbasis

Phase 3: Erweiterung der Wissensbasis um implizite Informationen
Gemäß der Transitivität von Rollenhierarchien wird die Wissensbasis um `dominates('Manager','Kundenberater')` und `dominates('Manager','Sekretär')` erweitert.

Phase 4: Auswertung der Regel
Der Körper der Regel `role_cannot_execute` wird ausgewertet, und es wird festgestellt, dass die Rolle 'Sekretär' und 'Kundenberater' keine Instanzen von 'Auftrag prüfen' ausführen darf.

Da Jenny als Sekretärin diesen Auftrag aber begonnen und fertiggestellt hat, wird dies als Regelbruch vermerkt.

Phase 5: Ausgabe
Weil für die Aufgabe 'Antrag prüfen' kein Eventtyp angegeben wurde, wird der Regelverstoß zweimal registriert.

```
"Die Rolle, die Antrag prüfen ausführt, muss die Rolle,
die den Antrag annimmt, dominieren"
Illegal execution found:
User Jenny executed Task 2

"Die Rolle, die Antrag prüfen ausführt, muss die Rolle,
die den Antrag annimmt, dominieren"
Illegal execution found:
User Jenny executed Task 3
```

Abbildung 6.4: Output nach der Analyse

6.2 Diskussion

Die entwickelte Definitionssprache stellt einige vordefinierte Prädikate zur Verfügung, die ausdrücken, ob die Ausführung einer Aktivität durch einen bestimmten Nutzer oder Rolle gewünscht oder unerwünscht ist. Diese Erwartungen sind daran gebunden, ob der Körper der Regel durch vorhergehende Aktivitäten erfüllt wurde. Ist dies der Fall, so tritt die Regel in Kraft. Die Erwartung wird anschließend gegen die Informationen aus den Logs verglichen. Wenn die tatsächliche Ausführung nicht der Spezifikation entspricht, wird dies verzeichnet.

Der Vorteil am implementierten Programm ist, dass es sich leicht erweitern lässt, um weitere Erwartungstypen zu definieren. Um zB ein Prädikat `expected execution time of TT is TP` zu implementieren, muss dieses Prädikat in der Grammatik Definition hinzugefügt werden und ein weiterer Regeltyp im Container zugelassen werden. Diese Regel wird dem Container hinzugefügt, sobald der Listener diesen Typ entdeckt und `main.pl` um eine Routine erweitert werden, die die tatsächliche Ausführungszeit mit der erwarteten Zeit vergleicht.

Für alle nicht vordefinierten Erwartungen kann man `illegal execution` verwenden, um zu beschreiben, welcher Verlauf einen Regelbruch darstellt. Somit ist das Programm nicht an die definierten Kopf-Prädikate gebunden und kann alle Einschränkungstypen abdecken.

Durch die eindeutige Notation von Aktivitäten bezüglich des Gültigkeitsbereich wird schon während dem parsen erkannt, welche Spanne eine Regel besitzt. Der Regel werden intern weitere Prädikate hinzugefügt, die den jeweiligen Kontext erzwingen. Da im Namen der Aktivität bei Instanzübergreifenden Regeln bereits ein Platz für die Instanzvariable vorhanden ist, kann man leicht definieren, ob sich einzelne Aktivitäten in einer Regel vielleicht doch in der selben

Instanz befinden müssen. Ferner kann man bei Betrachtung der status.pl die genauen Instanz Ids ermitteln, um bei Bedarf eine eindeutige Instanz für die Untersuchung zu erzwingen.

Ein weiterer Vorteil an dem entwickelten Programm ist die Flexibilität bezüglich der verwendeten Prädikate. Um Nutzer und Rollen zu gruppieren, stehen einige Vordefinierte Prädikate zur Verfügung. Mittels des Schlüsselwortes "DEF" lassen sich eigene Prädikate definieren, die entweder gesetzt werden oder denen man sogar eine Regel erstellen kann.

Kapitel 7

Zusammenfassung

Das Ziel der Arbeit war es, eine Beschreibungssprache für Regeln zu entwickeln, die sowohl für die Spezifikation von Regeln zu Aktivitäten innerhalb einer Prozessinstanz als auch zwischen mehreren Instanzen eingesetzt werden kann. Das wurde gelöst, indem eine für die jeweilige Spannweite eindeutige Notation für Aktivitäten gewählt wurde, die es dem Parser erlaubt, den Kontext sofort zu erkennen und die Regeln automatisch so zu erweitern, dass der richtige Kontext gewährleistet ist. Das nimmt dem Programmierer die Arbeit ab, in den Regeln selbst definieren zu müssen, ob sich die Tasks auf die selbe Instanz beziehen. Da die Variablen für die Aktivitäten auch Werte für die Prozessinstanz (im Instanz-übergreifenden Kontext) und auch für das Prozessschema (im Prozess-übergreifendem Kontext) enthalten, ist es dennoch möglich, einzelne Aktivitäten bei Bedarf auf spezielle Instanzen festzulegen bzw zu erzwingen, dass sich mehrere Aktivitäten in der selben Instanz befinden müssen. Die Struktur der Definitionssprache ist so gewählt, dass sie einerseits an die natürliche Sprache angelehnt ist, um das Verständnis bei Lesen der Regeln zu erleichtern. Gleichzeitig lässt sie durch ihre Verwandtschaft zur logischen Programmierung keine Mehrdeutigkeit zu.

Die Regeln und Fakten werden auf Prologklauseln abgebildet. Die flexiblen Container - die interne Datenstruktur, welche alle Klauseln zwischenspeichert um sie schließlich zusammen in Prologklauseln zu konvertieren - ermöglichen es, eigene Prädikate in der Regelspezifikation zu definieren und sogleich im weiteren Verlauf zu verwenden.

Der Compliancechecker ist in Prolog implementiert und berechnet inwiefern die aus den Logs extrahierten Klausen der Spezifikation entsprechen. Es wurden zahlreiche Regeln aus den verschiedenen Bereichen in Abschnitt 3.2.2 getestet. Die Regelbeschreibungssprache besitzt vordefinierte Prädikate, die es möglich machen, auszudrücken, ob die Ausführung einer Aktivität durch einen Nutzer / Rolle verboten oder zwingend ist. Alle weiteren Erwartungen, wie zB die Ausführung zu einem bestimmten Zeitpunkt, können dadurch getestet werden, dass der unerwünschte Fall im Körper der `illegal execution` Regel beschrieben wird. Die Struktur des Programms und der Grammatik macht die Definitionssprache leicht erweiterbar, um weitere Prädikate oder Regeltypen fest zu integrieren, ohne dass die Definitionssprache an Übersichtlichkeit verliert.

Anhang A

Prädikate

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

Die Beschreibungen sind nur Vorschläge für den Einsatz. Dem Programmierer ist selbst überlassen, wie er diese Prädikate interpretieren und einsetzen möchte.

Tabelle A.1: Prädikate für externe Informationen.

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 (links) dominiert Rolle 2 (rechts)
'critical_task_pair(' TT ',' TT ')'	Die beiden Aufgaben sind ein kritisches Paar. Die Nutzer, die diese beiden Aufgaben ausführen, werden als *collaborators* in die interne Datenbank eingetragen.

Prädikate für die Spezifikation der Authorisierung, Rollenbeziehungen und Bestimmung kritischer Aufgabenpaare.

Tabelle A.2: Spezifikation

Prädikat	**Beschreibung**
'user' UT 'executed' TT	Ut hat TT ausgeführt. Der Event-Typ ist unbestimmt und muss extra angegeben werden.
'role' RT 'executed' TT	RT hat TT ausgeführt. Der Event-Typ ist unbestimmt und muss extra angegeben werden.
UT 'is assigned to' TT	UT wurde TT zugewiesen. Entspricht dem Event-Typ 'assign'.
TT 'started'	TT wurde gestartet. Entspricht dem Event-Typ 'start'.
TT 'aborted'	TT wurde abgebrochen. Entspricht dem Event-Typ 'ate_abort'.
TT 'completed'	TT wurde erfolgreich abgeschlossen. Entspricht dem Event-Typ 'complete'.
'eventtype of ' TT ' is ' ET	Prädikat, um einer Aktivität einen beliebigen Event-Typ zuweisen zu können.
UT 'is collaborator of' UT	UT sind alle Akteure, die an criticalTaskPair gearbeitet haben. Collaborator muss nicht extra wie criticalTaskPair gesetzt werden, sondern wird in der Analysephase vom Model-Checker berechnet.
'timestamp of' TT 'is' VAR	Der Zeitpunkt von TT wird in eine Variable an der Stelle von VAR geschrieben.
'timeinterval of' TT 'and' TT 'is' VAR	Das Zeitinterval zwischen TT 1 (links) und TT 2 (rechts) wird in eine Variable an der Stelle VAR geschrieben.
'attribute' VAR1 'of' TT 'is' VAR2	TODO

Da sich die beiden *executed* Prädikate nicht auf ein bestimmtes Event beziehen, ist es notwendig, das Event extra anzugeben, um keine multiplen Ergebnisse zu erhalten. Es werden vier Prädikate für die Event-Typen *assign*, *start*, *abort* und *complete* angeboten. Für alle weiteren kann man das allgemeine Event-Prädikat 'Event(' TT ')' verwenden. Für das Event *skip* wäre es EventType(TT).'skip'. Das Event muss in zwei einfachen Anführungszeichen stehen.

Tabelle A.3: Status Prädikate

Prädikat	**Beschreibung**
'NUMBER WHERE (' <body> ') IS' RES	Zählt die Anzahl der verschiedenen Lösungen für <body> und speichert sie in einer Variablen an der Stelle von RES
'NUMBER OF' VAR 'WHERE (' <body> ') IS' RES	Zählt die Anzahl der verschiedenen Lösungen für VAR, die in <body> vorkommen, wenn <body> erfüllt wird. VAR kann eine beliebige Variable sein, muss aber mindestens einmal in <body> vorkommen.
'SUM OF' NT|TP|TS 'WHERE (' <body> ') IS' RES	Gibt die Summe einer Variablen aus <body> zurück. Diese Variable sollte für einen numerischen Wert stehen (zB in den Attributen einer Aktivität) oder für Zeitpunkte oder Zeitintervalle.
'AVG OF' NT|TP|TS 'WHERE (' <body> ') IS' RES	Gibt den Durchschnitt einer Variablen aus <body> zurück.
'MIN OF' NT|TP|TS 'WHERE (' <body> ') IS' RES	Gibt das Minimum einer Variablen aus <body> zurück.
'MAX OF' NT|TP|TS 'WHERE (' <body> ') IS' RES	Gibt das Maximum einer Variablen aus <body> zurück.

Das Resultat wird in der Variable gespeichtert, die anstelle von RES definiert wurde. <body> ist eine Konjunktion von Status- , Externen und Spezifikationsprädikaten. Für <body> gelten die selben Regeln wie für den Körper einer Regel bezüglich verwendbarer Prädikate, Negation und Disjunktion.

Tabelle A.4: Aggregationsprädikate

Prädikat	**Beschreibung**
= |! =	Gleichheit und Ungleichheit kann man auf alle Argumenttypen anwenden, solange auf der rechten und linken Seite der selbe Typ steht. Es können jeweils Strings, numerische Werte, Zeitpunkte und Zeitstempel untereinander verglichen werden.
< | <= | > | >=	Ungleichheit kann nur auf Rollen (um Beziehungen zwischen Rollen bezüglich der Hierarchie festzustellen), numerische Werte, Zeitpunkte und Zeitstempel angewendet werden.

'Manager' > 'Azubi' ergibt ein positives Ergebnis, wenn der 'Manager' in der Hierarchie über dem 'Azubi' steht.

Tabelle A.5: Vergleiche

Prädikat	**Beschreibung**
+	Numerische Werte und TP + TS (= TP)
−	Numerische Werte und TP - TP (= TS)
∗|/	Als Argumente sind nur numerische Werte erlaubt.

2015-08-03 + P5D liefert das Ergebnis 2015-08-08.

Tabelle A.6: Arithmetische Operationen

Prädikat	**Beschreibung**
UT cannot execute TT	Nutzer UT darf TT nicht ausführen.
UT must execute TT	Nutzer UT muss TT ausführen.
RT cannot execute TT	Rolle RT darf TT nicht ausführen.
RT must execute TT	Rolle RT muss TT ausführen.
illegal execution	Die Prozessinstanz hat den Körper der Regel erfüllt (was aber nicht erwünscht war) und hat somit die Spezifikation gebrochen.

Im Körper von `illegal execution` kann alles beschrieben werden, was zu einer ?? Ausführung führt. Wenn `IF USER executed 'T1' AND USER executed 'T2' THEN illegal execution` *true* zurückgibt, bedeutet es, dass ein Nutzer sowohl T1 als auch T2 ausgeführt hat, dies aber nicht erlaubt war.

Tabelle A.7: Prädikate für den Kopf einer Regel

Anhang B

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.
TODO Einfügen, wenn Programm fertig

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 C

Anleitung zur Verwendung

TODO Fertig machen, wenn Programm fertig

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 entwickelt und getestet

1. Linux Ubuntu 14.04 LTS 64-Bit
2. Java version 1.7.0_79
3. SWI-Prolog version 6.6.4 for amd64

Ausführen des Programms per Kommandozeile
Entweder muss sich der Ordner libraries oder alle benötigten Pakete im Classpath befinden. Alle Ordner (rule, log, output,..) müssen im bin-Ordner sein.

```
java -cp .:../../libraries/* main.Main
```

Das Paket

Einrichten des Programms in Eclipse

Einbinden in eigenes Projekt
Das Paket `iicmchecker` muss sich samt allen Abhängigkeiten im Build-Path des Projekts befinden. Gestartet wird es über das Kommando `run()`.

```
IICMChecker checker = new IICMChecker();
checker.run();
```

Beispiel laufen lassen

Individuelle Einstellungen

Der IICMChecker besitzt diverse Methoden, um individuelle Einstellungen vorzunehmen. Folgende Optionen sind möglich (im Kommentar steht die Klasse des Arguments):

```
checker.setLoggerLevel(loggerLevel);     // java.util.logging.Level
checker.setRuleLocation(rulelocation);   // java.lang.String
checker.setRuleFiles(rulefiles);   // java.lang.String[]
checker.setLogLocation(loglocation); // java.lang.String
checker.setLogFiles(logfiles); // java.lang.String[]
checker.setOutputLocation(outputlocation); // java.lang.String
```

Anhang D

Paketstruktur

Der Java Teil des entwickelten Programm befindet sich im Paket `iicmchecker`. Es besteht aus den Paketen `iicmchecker.compliancechecker`, `iicmchecker.constraintReader`, `iicmchecker.logtransformer`, `iicmchecker.storage`, `iicmchecker.utils`.

In `iicmchecker.compliancechecker` findet man die Klasse `iicmchecker.compliancechecker.Compliancechecker` welche für das Starten des Prolog Teils zuständig ist.

`iicmchecker.constraintReader` beinhaltet alle Klassen und benötigte Dateien für den *Lexer, Parser* und den *Listener*.

In `iicmchecker.logtransformer` liegt die Klasse `iicmchecker.logtransformer.LogTransformer` die Logs im *SEWOL* Format in Prolog-Fakten übersetzt.

`iicmchecker.storage` enthält die Container - eine interne Datenstruktur zum temporären speichern aller Informationen und anschließendem Konvertieren ins Prolog-Format.

Im Paket `iicmchecker.utils` liegen alle Helfer-Klassen zur String-Konvertierung und Überprüfung, *Logging* und *Exceptions*.

Das Programm wird über die Klasse `iicmchecker.IICMChecker` gestartet.

`main.Main` ist eine Beispielklasse für den Aufruf für den `IICMChecker`.

Im Ordner `prologfiles` befinden sich alle generierten Prologdateien sowie `main.pl` und `start.pl`.

Die Defaultordner für die Regeln und `results.txt` sind `rulefiles` und `outputfiles`.

Anhang E

Grammatik im BNF Format

TODO fertig machen, wenn Programm fertig

<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>))* ”)”
<def-symbols>	::=	DEF...
<set-symbols>	::=	SET...
<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>

Anhang F

Algorithmus Compliance Checker

Algorithm 1 Algorithmus Model Checker

1: **for** each task pairs $t_i, t_j \in critical\_task\_pair$ **do**
2: get u_i: u_i executed t_i
3: get u_j: u_i executed t_j
4: add collaborator(u_i, u_j)
5: **end for**
6: **for** each task pairs $t_i, t_j \in critical\_task\_pair$ **do**
7: add critical_task_pair(u_j, u_i)
8: **end for**
9: **for** each user pairs u_i, u_j and $u_j, u_k \in related$ **do**
10: add related(u_i, u_k)
11: **end for**
12: **for** each user pair $u_i, u_j \in related$ **do**
13: add related(u_j, u_i)
14: **end for**
15: **for** each role pairs r_i, r_j and $r_j, r_k \in dominates$ **do**
16: add dominates(r_i, r_k)
17: **end for**
18: **for** each rule $user\_cannot\_execute(u_i, t_i)$ **do**
19: **if** $\exists$ $user\_executed(u_i, t_i)$ **then**
20: **return** Illegal execution
21: **end if**
22: **end for**
23: **for** each rule $role\_cannot\_execute(r_i, t_i)$ **do**
24: **if** $\exists$ $role\_executed(r_i, t_i)$ **then**
25: **return** Illegal execution
26: **end if**
27: **end for**

28: **for** each rule $user\_must\_execute(u_i, t_i)$ **do**
29: **if** $\nexists$ $user\_executed(u_i, t_i)$ **then**
30: **return** Illegal execution
31: **end if**
32: **end for**
33: **for** each rule $role\_must\_execute(r_i, t_i)$ **do**
34: **if** $\nexists$ $role\_executed(r_i, t_i)$ **then**
35: **return** Illegal execution
36: **end if**
37: **end for**
38: **for** each rule *illegalexecution* **do**
39: **if** body of rule is satisfied **then**
40: **return** Illegal execution
41: **end if**
42: **end for**

Literaturverzeichnis

[1] V.D. Gligor, S.I. Gavrila, and David Ferraiolo. On the formal definition of separation-of-duty policies and their composition. In *Security and Privacy, 1998. Proceedings. 1998 IEEE Symposium on*, pages 172–183, May 1998. doi: 10.1109/SECPRI.1998.674833.

[2] Reinhardt A. Botha and Jan H. P. Eloff. Separation of duties for access control enforcement in workflow environments. *IBM Syst. J.*, 40(3):666–682, March 2001. ISSN 0018-8670. doi: 10.1147/sj.403.0666. URL http://dx.doi.org/10.1147/sj.403.0666.

[3] 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.

[4] 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.

[5] E. Bertino, E. Ferrari, and V. Atluri. An authorization model for supporting the specification and enforcement of authorization constraints in workflow management systems. In *in Workflow Management Systems. ACM Transactions on Information System Security*, pages 65–104, 1999.

[6] Sciff documentation. http://www.lia.deis.unibo.it/research/climb/tools-SCIFF.html, 2015. Accessed: 2015-08-02.

[7] Federico Chesani, Paola Mello, Marco Montali, Fabrizio Riguzzi, Maurizio Sebastianis, and Sergio Storari. Checking compliance of execution traces to business rules. In Danilo Ardagna, Massimo Mecella, and Jian Yang, editors, *Business Process Management Workshops*, volume 17 of *Lecture Notes in Business Information Processing*, pages 134–145. Springer Berlin Heidelberg, 2009.

[8] Marco Alberti, Federico Chesani, Marco Gavanelli, Evelina Lamma, Paola Mello, and Paolo Torroni. Verifiable agent interaction in abductive logic programming: the sciff framework. *ACM Transactions on Computational Logic (TOCL)*, 9(4):29, 2008.

[9] Fabio Casati, Silvana Castano, and Mariagrazia Fugini. Managing Workflow Authorization Constraints Through Active Database Technology. *Information Systems Frontiers*, 3(3):

319–338, September 2001. ISSN 1387-3326. doi: 10.1023/A:1011461409620. URL http://dx.doi.org/10.1023/A:1011461409620.

[10] Gerd Wagner. How to design a general rule markup language. In *In Invited talk at the Workshop XML Technologien für das Semantic Web (XSW 2002*, pages 24–25, 2002.

[11] H. T. De Beer and B. F. Van Dongen. Process mining and verification of properties: An approach based on temporal logic. In *On the Move to Meaningful Internet Systems 2005: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2005, volume 3760 of Lecture Notes in Computer Science*, pages 130–147. Springer-Verlag, 2005.

[12] 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.

[13] Jason Crampton and Royal Holloway. On the satisfiability of constraints in workflow systems, 2004.

[14] K. Tan, J. Crampton, and C.A. Gunter. The consistency of task-based authorization constraints in workflow. In *Computer Security Foundations Workshop, 2004. Proceedings. 17th IEEE*, pages 155–169, June 2004. doi: 10.1109/CSFW.2004.1310739.

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

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