

Bachelor's Thesis

Roles and Delegation in Ad-Hoc Business Processes

Mike Nagora

Supervision

Prof. Dr. Mathias Weske, Hasso-Plattner-Institute, Potsdam, Germany

M.Sc. Gero Decker, Hasso-Plattner-Institute, Potsdam, Germany

M.Sc. Harald Schubert, SAP AG, Walldorf, Germany

June 30, 2008

Abstract

Workflow management systems have become more important in recent years. Be it for the modelling, planning or the execution of processes in companies. But at the moment a big disadvantage of these systems is that they don't support enough flexible execution of processes. Therefore the employees are on their own when they have problems during the process execution and can not be supported by the system. Within the scope of the bachelor's project "Ad-Hoc Business Process" at the chair of 'Business Process Technology' at the Hasso-Plattner-Institut in Potsdam several concepts and prototypes have been developed in cooperation with the SAP AG to support flexibility.

This paper deals especially with one of our developed concepts, the delegation. Here first scenarios are presented, which clarify the necessity of such a function and give demonstration examples for the explanation of the concepts. Further the functionality and prototypical realization is explained in more detail. A further part of the paper consists of comparisons between our functionalities and concepts of the SAP AG.

Zusammenfassung

Workflowmanagementsysteme gewinnen in den letzten Jahren immer mehr an Bedeutung, sei es beim Modellieren und Planen von Arbeitsabläufen oder beim Ausführen dieser im Unternehmen. Ein großer Nachteil der Systeme besteht aber derzeit noch in der starren Ausführung dieser Prozesse, ohne ausreichende Flexibilität anzubieten. So sind die Mitarbeiter bei Ausnahmen im Prozess auf sich allein gestellt und können nicht vom System unterstützt werden. Im Rahmen des Bachelorprojektes "Ad-Hoc Business Processes" am Lehrstuhl 'Business Process Technology' des Hasso-Plattner Instituts sind in Kooperation mit der SAP AG verschiedene Konzepte und Prototypen entwickelt worden, welche den Mitarbeiter bei solchen Problemen unterstützen sollen.

Diese Arbeit beschäftigt sich speziell mit einem dieser neuen Konzepte, der Delegation. Dabei werden zu erst Szenarien vorgestellt, um die Notwendigkeit einer solchen Funktion zu verdeutlichen und für die Erläuterung der Konzepte einzelne Schaubispiele zu geben. Zudem wird auf die neue Funktionalität mit ihren Besonderheiten konzeptionell als auch auf ihre prototypische Umsetzung eingegangen. Ein weiterer Teil der Arbeit beschäftigt sich mit Vergleichen zu derzeit ähnlichen Konzepten der SAP AG.

Inhaltsverzeichnis

| | | |
|----------|--|-----------|
| 1 | Einleitung | 1 |
| 2 | Anwendungsfälle | 3 |
| 2.1 | Einstellungsantrag für einen Mitarbeiter | 3 |
| 2.2 | Kreditantrag | 4 |
| 3 | Delegation | 7 |
| 3.1 | Berechtigungen von Rollen | 8 |
| 3.2 | Einfache Delegation | 8 |
| 3.3 | Multiple Delegation | 12 |
| 4 | Technische Realisierung | 15 |
| 4.1 | Design-Entscheidung | 15 |
| 4.2 | Oryx | 16 |
| 4.3 | Task Lifecycle / Transformation | 19 |
| 4.4 | Ausführungsumgebung | 24 |
| 4.4.1 | Datenspeicherung | 24 |
| 4.4.2 | Auswertung von Rollen und Nutzer | 26 |
| 4.4.3 | Angewandte XSL Transformationen | 29 |
| 5 | Umsetzung von und Vergleich mit SAP Konzepten | 33 |
| 5.1 | Additional Approver | 33 |
| 5.2 | Forward | 35 |
| 5.3 | Shared Task | 35 |

| | |
|-----------------------------|-----------|
| 6 Zusammenfassung | 37 |
| A Anhang | 39 |
| Literaturverzeichnis | 43 |

1. Einleitung

In den letzten Jahren haben sich Geschäftsprozessmanagementsysteme immer weiter durchgesetzt. Sie tragen dazu bei, dass viele Prozesse im Unternehmen automatisch ablaufen können und somit die Durchlaufzeit stark verbessert wird. Ihre größte Schwäche besteht aber darin, dass alle Prozesse einer starren Ausführung unterliegen. Treten nun unvorhersehbare Probleme auf, wissen sich die meisten Mitarbeiter nicht anders zu helfen, als beim Kollegen Rat zu suchen. Sie umgehen damit das System und die getroffenen Entscheidungen werden nicht dokumentiert. Soetwas darf in der heutigen Zeit nicht passieren. Deshalb muss man den Mitarbeitern Mittel zur Hand geben, welche sie bei der Problemlösung unterstützen.

Im Rahmen dieses Bachelorprojekts 'Ad-Hoc Business Processes' wurde genau nach solchen Fähigkeiten eines Systems geforscht, verschiedene Anwendungsfälle betrachtet und Konzepte entwickelt, welche sich für heutige Systeme mehr als nützlich erweisen. Entstanden sind dabei Konzepte zur Delegation von Aufgaben, Reevaluierung und zur übersichtlicheren und kompakteren Modellierung von Prozessen, bei denen zur Laufzeit erst durch den Nutzer festgelegt wird, welche Ausführungsreihenfolge gewählt wird. Anschließend wurden diese Konzepte an zwei Prototypen evaluiert.

Das Bachelorprojekt fand in Kooperation mit der SAP AG und dem Lehrstuhl Business Process Technology des Hasso-Plattner Instituts in Potsdam unter Führung von Prof. Dr. Weske statt und wurde von M. Sc. Gero Decker und M. Sc. Harald Schubert betreut. Die Kooperation war darauf ausgelegt, dass wir von der SAP AG Anwendungsfälle erhalten, die vom Kunden wirklich gewünscht werden, als auch Ideen aus der SAP AG aufzugreifen und Konzepte für die SAP AG zu entwickeln, die an unserer Oryx-Plattform [4] evaluiert werden.

Im Zuge des Projekts sind sechs Arbeiten entstanden, in denen alle von uns eingeführte Konzepte erläutert werden. Da wäre zum Ersten die Arbeit von Philipp Maschke [11] zu nennen. Er beschäftigte sich mit dem Thema der Reevaluierung. Das Konzept des kompakteren Modellieren durch eine Ad-Hoc Task wird in der

Arbeit [9] von Stefan Krumnow beleuchtet. Seine Arbeit bezieht sich einerseits auf die Oryx-Plattform und andererseits auf die Umsetzung der Ad-Hoc Task im Process Composer. Die restlichen Implementierungen am SAP Galaxy sind in der Arbeit von Mathias Kleine [7] dokumentiert. Bei den zwei letzten Arbeiten handelt es sich zum einen um die Vorstellung der Architektur der Oryx Plattform und deren Modifikationen, welche von Lutz Gericke [6] in seiner Arbeit beschrieben werden. Zum anderen wird in der Arbeit von Alexander Koglin [8] auf die Umsetzung der Benutzeroberflächen mit Hilfe von XForms [2] näher eingegangen.

Bachelorarbeit

Das zentrale Thema dieser Bachelorarbeit ist das Konzept der Delegation. Um deren Notwendigkeit in der Praxis zu verdeutlichen, werden im zweiten Kapitel verschiedene Anwendungsfälle betrachtet und näher erläutert. Sie stellen gleichzeitig Schaubispiele für die Konzepte dar. Anschließend werden die Konzepte der Delegation und der Multiplen Delegation erklärt, bevor dann zu der technischen Realisierung im Prototypen Bezug genommen wird. Abschließend wird im letzten Kapitel ein Vergleich mit derzeitigen, ähnlichen Konzepten der SAP vorgenommen, wie auch die Umsetzung dieser Konzepte mit unseren Funktionen beschrieben.

2. Anwendungsfälle

Die gesamte Entwicklung unseres Prototyps basierte auf Anwendungsfällen, um die Notwendigkeit einzelner Funktionen aufzuzeigen und das System auf diese Fälle vorzubereiten. Diese Anwendungsfälle stammen einerseits von Kunden oder Entwicklern der SAP AG und andererseits vom Lehrstuhl. Im Allgemeinen haben wir uns mit mehreren Anwendungsfällen beschäftigt, dazu zählen das SRM-Szenario, EU-Dienstleistungsszenario, ein Antrag zum Umziehen in eine andere Stadt und ein Einstellungsantrag für einen Mitarbeiter.

In dieser Arbeit wird sich speziell auf den letztgenannten Anwendungsfall sowie auf ein Kreditantrag-Szenario bezogen und diese werden im nächsten Abschnitt des Kapitels näher erläutert.

2.1 Einstellungsantrag für einen Mitarbeiter

Das Szenario stellt einen allgemeinen, häufig in einem Unternehmen auftretenden Anwendungsfall dar. Täglich bekommen Personalabteilungen Bewerbungen von möglichen neuen Mitarbeitern. Diese müssen schnell bearbeitet werden können. In diesem Abschnitt soll es aber um einen spezielleren Fall des Szenarios gehen, nämlich um einen Einstellungsantrag für einen Hilfwissenschaftler am Hasso-Plattner-Institut (HPI).

Ein Student möchte am HPI am Lehrstuhl 'Business Process Technology' als studentische Hilfskraft arbeiten, daraufhin bewirbt er sich am Lehrstuhl und wird angenommen. Nun muss ein Einstellungsantrag angefertigt werden. Diese Aufgabe erledigt die zuständige Sekretärin. In diesem Formular müssen grundsätzliche Daten zum Studenten wie Name und Versicherungsnummer, den dann zu bekleidenden Jobtyp, eine Jobbeschreibung, das spätere Gehalt und den neuen Arbeitsraum aufgenommen werden. Sobald alles fertiggestellt ist, gibt die Sekretärin das Dokument an den Professor weiter, welcher diesen Auftrag

dann unterschreibt. Nachdem der Professor unterschrieben hat, wird das Formular zur Buchhaltung weitergeleitet, von dieser unterschrieben und abgeheftet. Modelliert ¹ sieht dieser Prozess wie folgt aus:

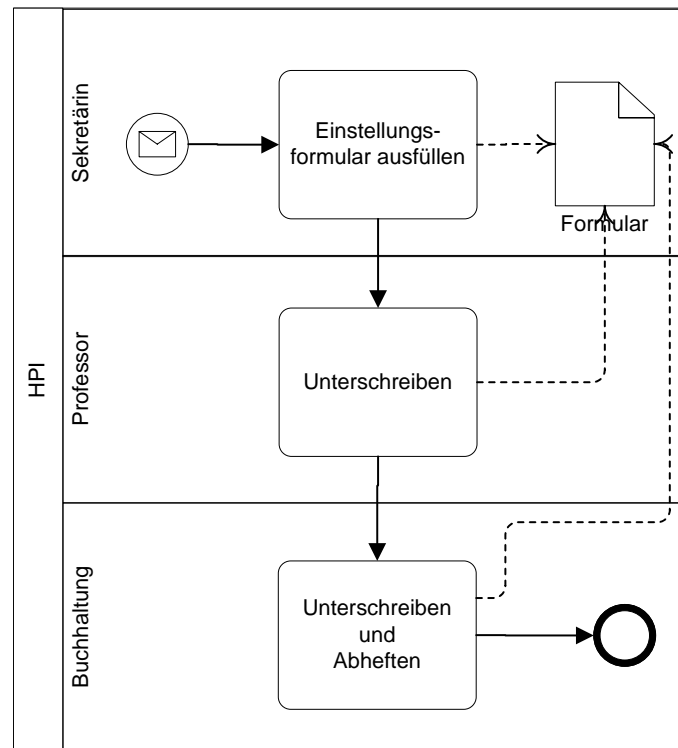


Abbildung 2.1: Einstellungsantrag für einen Mitarbeiter

Man beachte, dass dieser Prozess erst beim Erstellen des Formulars beginnt, da die Vorbetrachtungen zur Bewerbung nicht berücksichtigt werden.

Nachdem wir nun den Sachverhalt kennen, treffen wir die Annahme, dass der Prozess ohne Probleme durchgeführt werden kann. Doch die Realität sieht meist anders aus: Denn wie verhält es sich, wenn die Sekretärin beim Ausfüllen des Dokuments an einem Punkt ankommt, an dem sie nicht die Befugnis oder nicht die Kenntnis hat, das nächste Feld auszufüllen? Vielleicht ist sie sich in ihrer Entscheidung auch nicht sicher. All diese Probleme können auftreten und in heutigen Systemen nicht unterstützt werden.

2.2 Kreditantrag

Täglich kommt es in einer Bank dazu, dass ein Kunde einen Kredit beantragen möchte.

Ein Bankkunde kommt in eine Bank und möchte einen Kredit über 5.000 Euro beantragen. Er spricht dafür mit einem Bankangestellten am Schalter. Nach dem

¹Alle Prozesse wurde in der Business Process Modelling Notation (kurz BPMN [1]) modelliert

Gespräch füllt der Angestellte den Antrag aus, bestätigt diesen und stellt den Kredit aus.

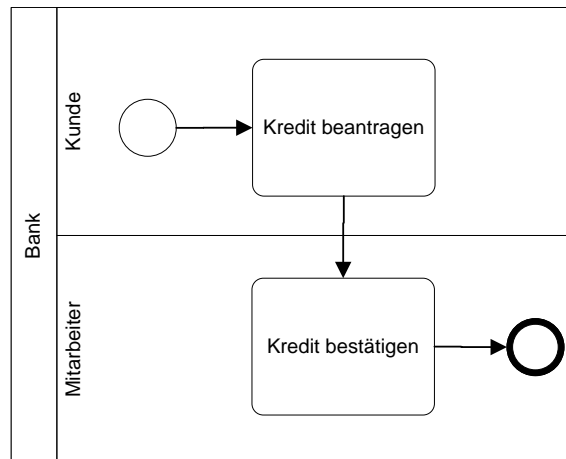


Abbildung 2.2: Kreditantrag

Bei einer Kreditsumme von 5.000 Euro mag der Kredit noch vom Bankangestellten bestätigt werden, doch ändert sich die Summe auf 20.000 Euro, übersteigt es die nötigen Befugnisse des Angestellten und er darf diesen Kredit nicht bestätigen. Dann müsste der Abteilungsleiter zu Rate gezogen werden, welche die Berechtigungen für diese Summe hätte. Aber selbst seine Berechtigungen reichen nicht aus, um einen Kredit in Höhe von 100.000 Euro auszustellen. Was damit zum Ausdruck gebracht werden soll ist, dass man von Anfang an nicht sagen kann, dass ein Angestellter einen Kredit ausstellen kann. Für die meisten Fälle ist er berechtigt, aber in Ausnahmefällen übersteigt es seine Kompetenzen.

3. Delegation

An den verschiedenen Ausnahmen in den Anwendungsfällen hat man gesehen, dass zusätzliche Funktionen benötigt werden, um diesen Einzelfällen gerecht zu werden.

Um nun der Mitarbeiterin beim Erstellen eines Einstellungsformulars eine Hilfe zu geben, benötigt man eine Funktion *Delegieren*. Mit dieser Funktion könnte sie einzelne Felder an Leute delegieren, die diese Felder ausfüllen können, da sie das nötige Wissen darüber besitzen. Genauso könnte sie einzelne Felder des Antrags, bei denen sie sich der Eintragung nicht sicher ist, an Personen delegieren, die sich sicher sind. Es würde ebenfalls den Fall abdecken, dass der nicht befugte Bankangestellte im Anwendungsfall *Kreditantrag* diese Aufgabe an eine Person mit höheren Berechtigungen delegieren kann.

Aufgrund dieser vielen Anwendungsfällen in jeder Branche haben wir für unseren Prototyp genau so eine Funktion 'Delegieren' vorgesehen.

Diese Funktion wird im ersten Teil des Kapitels näher erläutert. Der zweite Teil besteht in der Beschreibung einer zusätzlichen Funktion, der "*Multiple Delegation*". Bei dieser Methode handelt es sich um die Möglichkeit, an mehrere Personen gleichzeitig zu delegieren. Diese Funktion ist nötig, um dem Nutzer in verschiedenen Einzelfällen zu erlauben, dass er seine Aufgabe aufteilen kann und diese Teilaufgaben an mehrere Personen delegieren kann. Solch ein Fall könnte im Anwendungsfall *Einstellungsantrag für einen Mitarbeiter* auftreten, wenn die Sekretärin zwei verschiedene Felder nicht ausfüllen kann, aber weiß, dass diese von zwei unterschiedlichen Personen ausgefüllt werden können. Dadurch müsste sie beide Felder delegieren können. Die Aufteilung einer Aufgabe in mehrere Teilaufgaben erfolgt dabei über Sichtbarkeitsregeln, welche man für einzelne Formularfelder setzen kann.

Nur um Leute im Prozess anzusprechen und ihnen Aufgaben delegieren zu können, benötigt man zusätzlich ein *Rollenkonzept*, um Nutzer einem System bekannt zu machen. Dies soll im kommenden Abschnitt näher erläutert werden.

3.1 Berechtigungen von Rollen

Heutzutage werden immer mehr Menschen in Geschäftsprozesse eingebunden. Um diese Menschen einem Computersystem bekannt zu machen, müssen sie sich als Nutzer registrieren.

Nun könnte man sich vorstellen, dass jeder Nutzer alle Berechtigungen besitzt, um beispielsweise eine Aktivität auszuführen. Nur wäre dies den heutigen Verhältnissen nicht entsprechend. Erinnern wir uns an das Szenario *Kreditantrag*. Dort ist es so, dass der Angestellte nur bei einem minderen Kreditbetrag einen Kredit ausstellen darf. Würde man jetzt keine Unterscheidung in der Frage der Befugnis machen, könnte man auch keine Einschätzung treffen, wer in der Firmenhierarchie mehr Befugnisse hat als zum Beispiel der Angestellte. Dadurch würde man den Chef auf eine Ebene mit dem Angestellten stellen. Dies verdeutlicht, dass man unterschiedliche Rechte für jeden Nutzer festlegen können muss. Um jetzt aber nicht jedem Nutzer immer wieder unterschiedliche als auch gleiche Rechte zuteilen zu müssen, gibt es das Konzept von Rollen. Jede Rolle entspricht einer Ansammlung von Rechten, die alle Nutzer dieser Rolle besitzen.

Rechte

Anhand des Anwendungsfalles *Einstellungsantrag für einen Mitarbeiter* wird nun aufgezeigt, welche Rechte für die Bearbeitung einer Task benötigt werden.

Zu allererst brauchen wir ein Recht, das vorgibt, ob eine Person eine Task starten darf. In unserem Fall müsste die Mitarbeiterin die Befugnis haben, den Einstellungsantrag auszufüllen und damit die Task zu allokatieren. Bei der Bearbeitung des Formulars entsteht das Problem, dass die Mitarbeiterin ein Feld nicht ausfüllen kann, somit muss sie die Aufgabe an eine andere Person delegieren. Sprich sie muss das Recht besitzen, delegieren zu können. Die delegierte Person muss die Aufgabe erfüllen und darf diese Aktivität nicht überspringen. Somit darf ihr nicht das Recht auf Überspringen der Task gegeben werden. Andere Rechte stellen das Unterbrechen und wieder Aufnehmen einer Task dar, als auch die Möglichkeiten des Beenden und des Reviewen einer Task.

3.2 Einfache Delegation

Nachdem nun erklärt wurde, wie sich Nutzer am System registrieren und sich von anderen Nutzern unterscheiden, sind nun alle technischen Voraussetzungen erläutert worden. Insgesamt fehlt aber noch die Motivation für solch eine Funktion *Delegation*. Mit dieser Motivation soll nun der Abschnitt begonnen werden, bevor dann das Konzept des Delegierens erläutert wird.

In der Einleitung wurde deutlich, dass heutige Workflow-Management-Systeme mit Ausnahmen in verschiedenen Fällen nicht umzugehen wissen. In diesen Szenarien ist der Mitarbeiter mit dem Lösen der Probleme auf sich alleine gestellt. Ihm wird keine Hilfe angeboten, welche ihm bei der Lösungsfindung hilft. Dabei kann ein

Problem ganz schnell gelöst werden, indem man dem Mitarbeiter ein Mittel zur Hand stellt, welches ihm ermöglicht andere Personen an der Aktivität zu beteiligen. Bekommt er diese Möglichkeit nicht, wird der Mitarbeiter weiter den Weg über die Kommunikation mit anderen Mitarbeiter gehen, sei es per Email, Telefonanruf oder direktem Gespräch. Dabei ist aber auch nicht gesagt, dass er über Gespräche mit Kollegen zum Erfolg kommt. Es kann sein, dass er für die Problemlösung auf Daten zurückgreifen müsste, für die er keine Befugnis hat. Im schlimmsten Falle ergeben sich für ihn immer wieder neue Probleme, welche er lösen müsste. Bei dem Findungsprozess geht dem Mitarbeiter viel Arbeitszeit verloren, welche er sinnvoller mit der Bewältigung anderer Aufgaben hätte verwenden können. Außerdem wird bei diesem Lösungsprozess keine Entscheidung im System festgehalten. Aufgrund dessen könnte nie zurückverfolgt werden, was der jeweilige Mitarbeiter in der Situation gemacht hat. Das wäre für spätere, gleiche Situationen äußerst unproduktiv, da ein neuer Mitarbeiter den ganzen Prozess nochmals durchlaufen müsste.

Angenommen im Anwendungsfall *Einstellungsantrag eines Mitarbeiters* weiß die Sekretärin nicht, welches Gehalt für den Job-Type Entwickler bezahlt wird. Im Normalfall besitzt sie eine Liste mit Job-Typen und deren Gehälter, nur fehlt der Job-Typ Entwickler. Zudem besitzt sie keine Information, wer dies genau weiß. Um das Gehalt trotzdem im Formular einzutragen, telefoniert sie mit einer Sekretärin B. Wenn sie Glück hat, kennt sie die nötige Person C, welche die Information besitzt. Nach dem Gespräch mit der Person C kennt die Sekretärin das Gehalt und kann es eintragen. Doch durch ihre gewählten Umwege über das Telefon würden die Information verloren gehen. So wird nirgends festgehalten, wer diese Information besitzt und wer für spätere weitere Fragen zur Verfügung steht. Andererseits kann es auch sein, dass Person C im Moment keine Zeit hat, um die Frage von der Sekretärin zu beantworten. Sie hinterlässt ihr dafür eine Nachricht mit den Daten des Bewerbers, vergisst aber den Job-Typen. Bei der Bearbeitung der Aufgabe stellt sich nun Person C das Problem, dass sie nicht auf den Job-Typ des Bewerbers zugreifen kann. Sie hat dafür nicht die nötigen Berechtigungen. Somit muss Person C wieder ein Telefonat mit der Sekretärin führen. Der ganze Umstand kostet Zeit und Nerven für alle Beteiligten. Dabei benötigt die Sekretärin nur eine Funktionalität, um die Aufgabe an Person C zu delegieren, welche diese dann bearbeitet.

Gründe für das Delegieren können wie im betrachteten Fall Unkenntnis über Informationen, keine Berechtigung für Daten oder Unsicherheit sein. In der Präsentation [14] werden drei Hauptgründe für das Delegieren festgestellt.

- Fehlen von Ressourcen
- Kompetenz
- Spezialisierung

Wie man erkennen kann, ähneln sich diese stark zu den gezeigten Gründen aus dem Anwendungsfall.

Erkennbar an den gerade aufgezeigten Motiven und Problemen ist, dass eine Funktion zum Delegieren immer wichtiger wird. Ohne diese bliebe eine Ausführung starr und der Bearbeiter einer Task wäre immer auf sich allein gestellt. Es sei denn, er verfällt wieder in die eben beschriebenen Probleme.

Im restlichen Abschnitt soll nun genau das Konzept des Delegieren erläutert werden.

Bei der einfachen Delegation geht man davon aus, dass die delegierende Person ihre Aufgabe an eine weitere Person übergibt. Man trifft keine Annahme, ob sie weiß an wen sie diese Aktivität delegiert. Sie könnte also auch ihre Aufgabe an eine Putzfrau weiterreichen. Auf dieses Sicherheitsproblem wird aber in dieser Arbeit nicht eingegangen, sondern wir sehen in unserem Prototypen den Fall vor, dass die delegierende Person genau weiß, an wen sie die Aufgabe weiterreicht.

Angeregt durch dieses Problem ist die Delegation eng mit der Frage der Übergabe von Verantwortung verknüpft, da man sich immer wieder die Frage nach dem Delegieren stellen muss: Bin ich jetzt noch für diese Aufgabe verantwortlich? Einerseits könnte man argumentieren, dass man sich als delegierende Person sicher ist, dass der Delegierte diese Aufgabe bewältigen kann. Somit vertraut man dieser Person und schaut nicht nochmal über die Arbeit des Anderen. Andererseits könnte sich wiederum die übergebene Person nicht sicher sein und nochmal die Arbeit des Delegierten begutachten. Andere Gründe könnten darin liegen, dass die Aufgabe sehr wichtig ist oder anders, dass sich die delegierende Person Informationen merken möchte, um diese später selbst eintragen zu können. All diese Beweggründe zeigen, dass man zusätzlich zur Delegation auch eine Review Funktion benötigt.

In unserem Prototypen wollen wir diese beiden Funktionen unterstützen und zusätzlich eine neue Funktionalität einführen, um mehr Möglichkeiten beim Delegieren zu besitzen. Dabei handelt sich um das Festlegen von Sichtbarkeitsregeln für einzelne Formularfelder. Unter Sichtbarkeitsregeln verstehen wir drei Eigenschaften für Felder. So kann man festlegen, dass ein Feld für eine delegierte Person editierbar(writable) oder nur lesbar(readonly) ist, oder erst garnicht angezeigt wird. Dies ist daher sinnvoll, dass man manche Felder den Delegierten anzeigen muss, damit er seine Entscheidungen treffen kann. Dies trifft in dem Fall des zusätzlichen Genehmigers (siehe Kapitel 5.1) zu. Andere Felder möchte man aber dem Delegierten verbergen, weil es sich um sensible (private oder streng vertrauliche) Daten handelt.

Durch diese Sichtbarkeitsregeln erschließen sich uns neue Möglichkeiten, da man nun nicht mehr nur komplette Aufgaben delegieren muss, sondern auch nur einzelne Felder delegieren kann. Dies erhöht den Wirkungsgrad einer Person erheblich, da für sie mehr Möglichkeiten bestehen.

Nun stellt sich die Frage, wie wir diese Funktionen dem Nutzer zugänglich machen. Dafür haben wir verschiedene Masken für die Ausführung entwickelt. Nachfolgend wird nur die Reihenfolge jener und die Betätigung einzelner Buttons bei der Ausführung einer Delegation aufgezeigt. Die Entwicklung der einzelnen Formulare wird in der Arbeit von Alexander Koglin erläutert (siehe [8]).

Die erste Ansicht, welchen dem Nutzer angeboten wird, ist die Submit-Ansicht (3.1) zum Beenden einer Task. Entscheidet der User sich nun zum Delegieren, bestätigt er nicht die Task, sondern gelangt über Drücken des Interact-Buttons in die Delegationsansicht (3.2).

In dieser kann man nun auswählen, an welche Person die Aufgabe delegiert wird. Zudem kommen hier unsere eingeführten Sichtbarkeitsregeln zum Einsatz. Man kann nun für jedes Feld, durch Drücken des danebenstehenden Buttons, entscheiden, welchen Status dieses Feld beim Delegierten haben wird. Standardeinstellung ist, dass alle Felder für den Delegierten sichtbar und editierbar sind. Das ist dahergehend sinnvoll, da eine Person in der Vielzahl die komplette Task delegieren möchte. Entscheidet er sich dagegen, so kann er den Button einmal drücken und das Feld wird dem Delegierten nur angezeigt, sprich er kann den Inhalt des Feldes nur sehen aber nicht ändern. Beim zweiten Drücken würde das Feld der übergebenen Person nicht mehr angezeigt werden. Außerdem kann der Nutzer in dieser Maske markieren, ob er nach der Bearbeitung ein Review wünscht. Dafür müsste er die Chechbox bei Responsible bestätigen.

Abbildung 3.1: Submit-Ansicht

Abbildung 3.2: Delegationsansicht

Nachdem der Nutzer dann diese Ansicht mit Bestätigen des 'Delegate'-Buttons abschickt, bekommt die delegierte Person die Task in ihre Inbox und kann sie bearbeiten (siehe 3.3). Ihr wird einerseits nur die für sie delegierten Felder angezeigt und andererseits nicht mehr die Möglichkeit gegeben, diese Task noch einmal an eine andere Person weiterzudelegieren.

Nach dem Beenden der Task erscheint die bearbeitete Task entweder nochmal in der Inbox des Delegierenden (Review gewünscht (siehe 3.4)) oder sie ist gänzlich beendet.

Aber greifen wir nun nochmal den Punkt auf, dass die delegierte Person keine Möglichkeit besitzt, ihre Task nochmals weiterzugeben. Würde man dies erlauben, so könnte theoretisch jede delegierte Person eine weitere Delegation vornehmen, welche dann wieder delegiert. Zur Zeit wird dieser Fall noch nicht behandelt, weil sich dabei

| Name | Values |
|---------------------|----------------------|
| Name | <input type="text"/> |
| Job-Type | <input type="text"/> |
| Job-Beschreibung | <input type="text"/> |
| Gehalt | <input type="text"/> |
| Raumnummer | <input type="text"/> |
| Versicherungsnummer | <input type="text"/> |

Abbildung 3.3: delegierte Submit-Ansicht

| Name | Values |
|---------------------|----------------------|
| Name | <input type="text"/> |
| Job-Type | <input type="text"/> |
| Job-Beschreibung | <input type="text"/> |
| Gehalt | <input type="text"/> |
| Raumnummer | <input type="text"/> |
| Versicherungsnummer | <input type="text"/> |

Abbildung 3.4: Review-Ansicht

Probleme bei der Implementierung ergeben. Im nächsten Kapitel wird dazu Bezug genommen.

3.3 Multiple Delegation

Häufig ist es nötig an mehrere Personen Aufgaben zu delegieren, weil sie entweder für eine Person zu schwierig sind oder dem Zeitaufwand nicht entsprechend wären.

Dafür haben wir die Multiple Delegation entwickelt. Mit dieser Funktion wird es dem Bearbeiter möglich eine Aufgabe an mehrere Personen zu delegieren. Realisiert wird dies wieder über die oben erläuterten Sichtbarkeitsregeln.

Nachfolgend wird am Anwendungsfall *Einstellungsantrag für einen Mitarbeiter* auf die Reihenfolge der einzelnen Sichten eingegangen, welche für eine multiple Delegation nötig sind. In dem beschriebenen Einzelfall kennt die Sekretarin die Versicherungsnummer vom Studenten, sowie die Raumnummer für den späteren Arbeitsplatz nicht. Sie weiß nur, dass der Student die Versicherungsnummer kennt und die Raumnummer immer der Admin zuteilt. Deshalb teilt sie ihre Aufgabe und delegiert nur den Versicherungsnummereintrag zum Studenten und den Raumnummereintrag zum Admin.

Der Nutzer wählt in der submit-Ansicht erneut die Interaktion. Demnach wird ihm wieder die Delegationsansicht angezeigt. Der Unterschied zur normalen Delegation besteht nun darin, dass es einen zusätzlichen Button ('+') gibt. Wenn nun die Sekretärin das Feld für die Versicherungsnummer an den Studenten (Adam) und das Feld der Raumnummer an den Admin (Bert) delegieren möchte, füllt sie erst die nicht delegierten Felder aus. Das ist wichtig, da sie nach dem Delegieren keine

Möglichkeit mehr besitzt, diese Werte zu ändern. Danach kennzeichnet sie das Feld der Versicherungsnummer als editierbar, und delegiert dies über den '+'-Button an den Studenten. Durch das Drücken dieses Buttons wird die Sicht erneut angezeigt und die erste Teilaufgabe an den Studenten delegiert. Beim wiederholten Öffnen des Formulars legt sie nun die restlichen Sichtbarkeitseigenschaften für den Admin auf dem Raumnummerfeld fest und drückt 'delegate'. Dadurch bekommt der letzte Delegierte seine Aufgabe in seine Inbox und die Sekretärin ist mit der Delegation fertig, bekommt dementsprechend die Ansicht nicht nochmals angezeigt. Nachdem der Student und der Admin ihre Felder bearbeitet haben, wird die Task beendet. Es sei denn, die Sekretärin wünscht ein Review. Im UI betrachtet sieht dies wie folgt für den ersten Delegierten (siehe 3.5 und 3.6) aus und für den zweiten Delegierten so (siehe 3.7 und 3.8) aus.

The screenshot shows the 'Delegate task' form. At the top, the 'Delegate' dropdown is set to 'Adam'. Below it are fields for 'Message' and 'Deadline'. A 'Stay responsible for process' checkbox is checked. The 'Set all rights to' section has 'Writable', 'Readonly', and 'Invisible' buttons. The main table has two columns: 'Values' and 'Visibility Rights'. The 'Values' column contains input fields for Name (Adam), Job-Type (Entwickler), Job-Beschreibung (Web-Shops entwickeln), Gehalt (1000), Raumnummer, and Versicherungsnummer. The 'Visibility Rights' column contains buttons: 'Invisible' for Name, Job-Type, Job-Beschreibung, Gehalt, and Raumnummer; and 'Writable' for Versicherungsnummer. At the bottom are '+', 'Delegate', and 'Cancel' buttons.

Abbildung 3.5: Delegationsansicht für Adam

The screenshot shows the 'Execute delegated task' form. It displays 'Delegated from: Sekretärin'. Below this is a 'Values' section with a 'Versicherungsnummer' input field and a 'Submit' button.

Abbildung 3.6: Adam's Ausführungsansicht der delegierten Task

The screenshot shows the 'Delegate task' form. The 'Delegate' dropdown is set to 'Bert'. The 'Set all rights to' section has 'Writable', 'Readonly', and 'Invisible' buttons. The 'Values' column contains input fields for Name (Adam), Job-Type (Entwickler), Job-Beschreibung (Web-Shops entwickeln), Gehalt (1000), Raumnummer, and Versicherungsnummer. The 'Visibility Rights' column contains buttons: 'Invisible' for Name, Job-Type, Job-Beschreibung, and Gehalt; 'Writable' for Raumnummer and Versicherungsnummer. At the bottom are '+', 'Delegate', and 'Cancel' buttons.

Abbildung 3.7: Delegationsansicht für Adam

The screenshot shows the 'Execute delegated task' form. It displays 'Delegated from: Sekretärin'. Below this is a 'Values' section with a 'Raumnummer' input field and a 'Submit' button.

Abbildung 3.8: Bert's Ausführungsansicht der delegierten Task

4. Technische Realisierung

Bevor nun auf die Beschreibung der technischen Realisierung eingegangen wird, wird anhand des Architekturbildes aus Abb. 4.1 ein kurzer Überblick über die uns zur Verfügung stehende Architektur am Lehrstuhl gegeben. Einen detaillierteren Überblick gibt Lutz Gericke in seiner Arbeit (siehe [6]).

Grundsätzlich gibt es drei wichtige Komponenten: den Oryx, der als Modellierungsumgebung für Prozesse in der Business Process Modelling Notation (BPMN [1]) dient; einen BPMN Konverter, welcher mit dem Oryx modellierte Prozesse in Petri-netze - die in der Petri Net Markup Language (PNML, siehe [3]) ausgedrückt werden - umwandelt und eine Ausführungsumgebung, die diese konvertierten Petri-netze ausführt. Weiter erkennt man auf dem Architekturbild, dass es zusätzlich eine Worklist gibt, die auf der Ausführungsumgebung aufsetzt und dem Nutzer ein einheitliches UI zur Ausführung von Tasks zur Verfügung stellt

In diesem Kapitel wird in der eben beschriebenen Reihenfolge auf getätigte Änderungen in den einzelnen Komponenten eingegangen. Anfangen möchte ich aber mit Designentscheidungen, welche ich im Rahmen meiner Implementierungen getroffen habe.

4.1 Design-Entscheidung

Als Vorgabe galt es ein Rollenkonzept zu implementieren, um darauf aufsetzend die Delegation und die Multiple Delegation umzusetzen. Um es am Anfang einfach zu halten, habe ich mich dafür entschieden einzelne Rollen und User, welche wir für unsere Szenarien benötigen, fest in die Engine zu implementieren. So wurde von Anfang an festgelegt, zu welchen Rollen ein Nutzer gehört.

Dem System bekannt wird dieser Nutzer über ein Login Applet, welches vor dem Öffnen der Worklist erscheint. Aufgrund der vorgegebenen Zuweisung von Usern zu Rollen ist es nicht nötig, dem Nutzer beim Login einer Rolle zuzuordnen. Wollte man

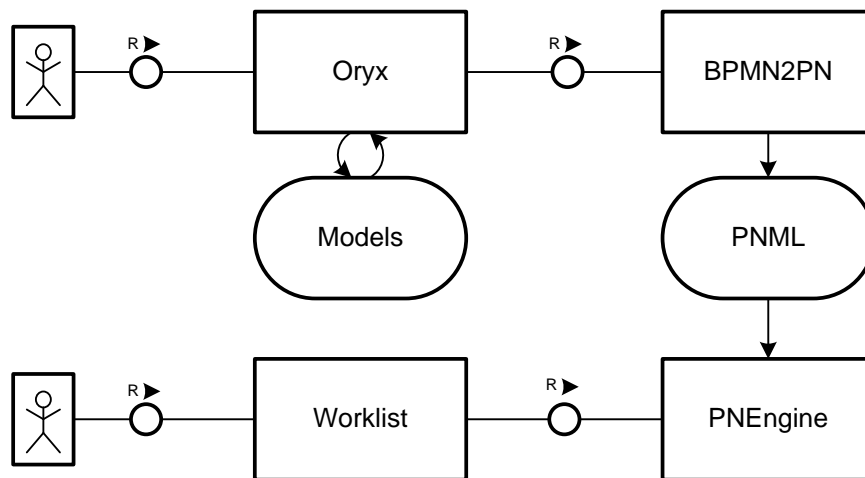


Abbildung 4.1: Architektur am Lehrstuhl

dies tun, erfordere dies ein zusätzliche grafische Oberfläche, welche nicht im Rahmen dieses Bachelorprojekts lag. Zudem sollte die Möglichkeit im Oryx bestehen, an alle Tasks Rollen zu annotieren.

4.2 Oryx

Im Oryx bestand die Aufgabe, dass man für eine Task eine Rolle festlegen kann. Aufgrund der Tatsache, dass es für eine Task eine Attributliste gibt und eine Rolle für eine Task auch nur ein Attribut ist, habe ich die Liste um ein zusätzliches Rollenattribut erweitert - ein kleiner Ausschnitt zur Umsetzung im Oryx (4.2).

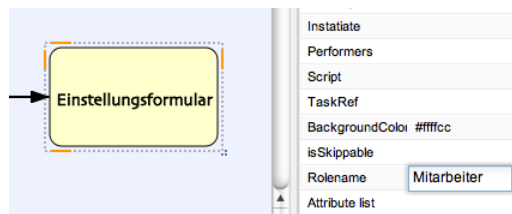


Abbildung 4.2: Rolle im Oryx festlegen

Nun wird der Prozess beschrieben, wie der Rollename im Oryx für jede Task in ein Dokument in PNML wie diese (Auszug für eine Transition) überführt wird. Beim Beginn der Überführung wird dabei die Einführung eines neuen Rollenattributs im Oryx erläutert.

```

<transition id="transition_tr_delegate_resource1" type="receive">
  <name>
    <value>tr_delegate_resource1</value>
    <text>tr_delegate_resource1</text>
  </name>
  <toolspecific tool="Petri Net Engine" version="1.0">
    <output>
      <model href="http://localhost:3000/">
    </output>
  </toolspecific>
</transition>

```

```

<form href="http://localhost:3000/" />
<bindings href="http://localhost:3000/" />
<worklist action="delegate" task="TaskA" />
<guard>
  <expr>place_pl_context_resource1.isDelegated == 'true'</
    expr>
</guard>
<role>Mitarbeiter</role>
<contextPlaceID>place_pl_context_resource1</contextPlaceID>
</toolspecific>
</transition>

```

Der erste Schritt für die besagte Überführung besteht darin, dass man in dem von uns angepassten Stencilset “bpmnexecutable” im Typ Node mit der id Task eine Eigenschaft Rolle hinzufügt. Nähere Informationen zur Stencilset-Spezifikation findet man in der Arbeit von Nicolas Peters [12].

```

{
  "id": "rolename",
  "type": "String",
  "title": "Rolename",
  "value": false,
  "description": "This role has the permission to execute this task.",
  "readonly": false,
  "optional": false
},

```

Damit sich später dieser Rollename im PNML wiederfindet, muss zusätzlich der BPMNRDFImporter und der Konverter angepasst werden.

Im BPMNRDFImporter wird der im Oryx gesetzte Rollename über das Stencilset, das intern in das Format RDF (Resource Description Framework) exportiert wird, ausgelesen und als Task-Attribut über die Methode setRolename() gespeichert. Falls kein Rollename für eine Task angegeben wird, wird der Rollennamen auf Default gesetzt.

```

if (attribute.equals("rolename")) {
  String roleValue = getContent(n);
  if (roleValue != null) {
    task.setRolename(roleValue);
  }
  else {
    task.setRolename("Default");
  }
}

```

Nun befindet sich der Name in einem Task-Attribut und kann im Konverter benutzt werden.

Bevor der Konverter vorgestellt wird, muss gesagt sein, dass wir uns am Anfang unserer Arbeit am Oryx dafür entschieden haben, eigene Unterklassen zu erstellen, um den Code konzeptionell sauber zu halten. Deshalb existiert für jede angepasste Klasse eine Exec- Klasse, die die ursprüngliche Klasse erweitert. Dies erkennt man auch an unserem veränderten Konverter, welcher den Namen ExecConverter trägt. Näheres zur Hierarchie findet man in der Arbeit von Lutz Gehricke (siehe [6]).

In dem ExecConverter wird eine Variable mit dem Wert des Rollennamen für eine Task definiert.

```
String rolename = task.getRolename();
```

Danach wird für jede Transition über die Funktion setRolename() ein Rollename gesetzt. Dies ist möglich, da jedem Node diese Funktion zur Verfügung steht und eine Transition ein erweiterter Node ist.

Beispiel für die Transition tr_allocate:

```
exTask.tr_allocate.setRolename(rolename);
```

ExTask ist ein Objekt der ExecTask-Klasse, welche, wie vorhin beschrieben eine Erweiterung der Task-Klasse ist. Diese enthält als Elemente Objekte von der Transition-Klasse, zu der auch tr_allocate zählt.

Nun fehlt nur noch, dass der Rollename ins PNML geschrieben wird, dies geschieht im letzten Schritt im ExecPNMLExporter. Hier wird für jede Transition überprüft, ob diesem ein Rollename übergeben wurde. Wenn dem so ist, wird dieser Name dem ToolspecificPNMLHelper übergeben.

```
if (transition.getRolename() != null) {
    tsHelper.setRolename(doc, ts, transition.getRolename());
}
```

Die Klasse ToolspecificPNMLHelper stellt eine Hilfsklasse dar, die festlegt, wie später einzelne Objekte in PNML gesetzt werden. In unserem Fall übergeben wir der Funktion setRolename() verschiedene Parameter. Dazu zählt: in welches Dokument später der Wert geschrieben werden soll, das Elternelement im späteren PNML und der Wert, der in dieses Element (Tag) im XML Dokument geschrieben werden soll.

```
void setRolename(Document doc, Element parent, String roleText) {
    Element role = (Element)parent.appendChild(doc.createElement("role"
    ));
    role.setTextContent(roleText);
}
```

4.3 Task Lifecycle / Transformation

Das war der Vorgang um ein Attribut im Oryx zu modellieren und dann ins PNML zu überführen. Nun stelle man sich vor, dass dieser Prozess für alle weiteren Attribute und Transitionen vollführt werde. Nach dieser Transformation aus einer modellierten Task entsteht ein Petrinetz, welches mittlerweile 13 Transitionen beinhaltet. Um diesen 'Task Lifecycle' soll es in diesem Abschnitt gehen. Dabei wird näher auf die Änderungen eingegangen, die durch diese Arbeit eingeflossen sind. Weitere Informationen findet man in der Arbeit von Philipp Maschke (siehe [11]).

Den kompletten Task Lifecycle kann man im Anhang (siehe A.1) finden.

Notation

In diesem Petri net (siehe A.1) gibt es einzelne Transitionen (z.B. `tr_allocate`), Stellen (z.B. `p_running`) zwischen Transition und deren verbindende Kanten. Zudem können zu jeder Transition Guards (z.B. `[if Delegated]`) definiert werden, die entscheiden, ob die jeweilige Transition schalten darf. Nähere Informationen zur Auswertung dieser Guards werden im weiteren Verlauf dieses Kapitels (Abschnitt 4.4.2. unter Guards) erläutert.

Desweiteren erkennt man eine Kontextstelle in der Darstellung. Diese ist mit jeder Transition verbunden und beinhaltet verschiedene Informationen zur Ausführung einer Task, wie z.B. ihren Eigner, den Status oder Information über ausgeführte Aktionen. Näheres gibt es in der Arbeit von Philipp Maschke (siehe [11]).

Zusätzlich zur Kontextstelle gibt es auch Datenstellen, die Informationen zum übergebenen Datenobjekt beinhalten. Vorausgesetzt es wird im Modell ein Datenobjekt an eine Task gehängt. Weitere Informationen hierüber findet man in der Arbeit von Alexander Koglin (siehe [8]).

Nachdem nun die Notation erklärt wurde, werden die einzelnen Teile des Lifecycles beschrieben.

Im vorherigen Abschnitt wurde erläutert, wie ein Rollenname aus dem Oryx in ein PNML überführt wurde. Dieser Name befindet sich nun an jeder Transition, ausgenommen der `tr_enable`, `tr_complete` und der `tr_finish` Transition. Diese Transitionen schalten automatisch und brauchen deshalb kein Rollenattribut.

Das Konzept der Rechte ist eng verbunden mit dem Task Lifecycle, weil jede Transition bzw. Aktion¹ der Transition auf ein Recht abgebildet wird. Das heißt, dass zum Beispiel das Recht auf Ausführen einer Task mit der Transition `tr_allocate` und der verbundenen Aktion 'allocate' verknüpft ist. Diese Annahme wurde für den Fall getroffen, bei dem eine Person eine Task allokkieren möchte und dadurch auch notwendigerweise die Transition `tr_allocate` ausführen muss. Hat sie nun nicht das Recht auf Schalten der `allocate`-Transition, wird ihr in der Worklist die Aktion 'Allokieren' nicht angeboten. Somit kann sie die Task nicht allokkieren und ausführen. Dies verhält sich bei den anderen Transitionen gleich (A.3 im Anhang).

¹Aktionen stellen die Ausführungsmöglichkeiten in der Worklist dar. Sie werden an jede Transition gebunden. Mehr Infos darüber findet man in der Arbeit von Lutz Gehricke [6]

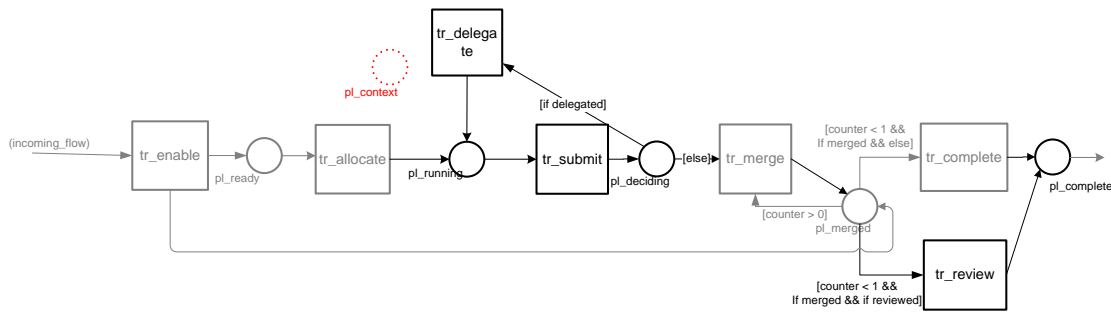


Abbildung 4.3: Delegation im Task Lifecycle

Da nun alle Vorkonzepte der Rollen und ihrer Berechtigungen erläutert wurden, werden nun die Änderungen am Task Lifecycle, welche durch die beiden Hauptkonzepte der Delegation und der multiplen Delegation bedingt sind, erläutert. Dafür werden einzelne Ausschnitte für die Funktionen aus dem Task Lifecycle entnommen und gezeigt.

Delegation

Die Delegation erkennt man im vollständigen Petri net (A.1 im Anhang) an der blauen Einfärbung, sprich diese Transitionen, Stellen und Guards wurden hinzugefügt, um die Funktionalität des Delegierens zu unterstützen. Dadurch dass die Review Funktion eng mit der Delegation verbunden ist, wird diese hier miterläutert. Dementsprechend wurde eine Transition `tr_delegate` und `tr_review` eingeführt. Zusätzlich wurde die Guards `[if delegated]` und `[if Reviewed]` hinzugeführt, um die richtige Ausführung der Funktionen zu Gewähr leisten.

Ansetzend an das vorherige Kapitel mit der Betrachtung der UIs wird nun die Ausführungsreihenfolge beschrieben. (siehe 4.3)

Für die Transition `tr_submit` wird eine Form hinterlegt, in der der Nutzer sagen kann, dass er die Task nicht alleine ausführen (bestätigen des 'Interact'-Buttons), sondern eine weitere Person damit betrauen möchte. Nun wird aufgrund dieser Entscheidung im Token der Wert des Elements im XML (Tag) `<isDelegated>` auf 'true' gesetzt. Diese Zuweisung wird durch einzelnes Feuern der Buttons ausgelöst. Dementsprechend ist der 'Interact'-Button mit der Zuweisung verbunden, dass der Wert des Elements `<isDelegated>` auf 'true' gesetzt wird. In der XForm sieht dies so aus:

```
<x:setvalue bind="isDelegated">true</x:setvalue>
```

Dabei wird für das Element `isDelegated` der Wert 'true' gebunden. Nähere Informationen zu diesen Zuweisungen finden sich in der Arbeit von Alexander Koglin (siehe [8]).

Dadurch, dass der Wert von `<isDelegated>` gleich 'true' ist, wird im nächsten Schritt die Guard in Richtung `tr_delegate` zu 'true' ausgewertet und der Nutzer

kann die Transition schalten. Dort wird ihm ein neues Formular angezeigt, worin er die Task an eine neue Person delegieren kann. Zusätzlich kann er in dieser auch die genannten Sichtbarkeitsregeln für den nächsten Delegierten definieren, die sich dann im Token wiederfinden. Entscheidet sich der Nutzer nun dafür, einzelne Felder dem Delegierten nicht anzuzeigen, wird durch Ändern des Buttons das jeweilige Attribut des Feldes geändert. Beispielsweise möchte die Sekretärin im Anwendungsfall *Einstellungsantrag für einen Mitarbeiter* nach dem Delegieren dem Studenten nicht das Gehalt anzeigen. Demnach wählt sie die Eigenschaft 'invisible' für das Feld 'Gehalt' aus. Im Token wird nun das Attribut auf `writable = 'false'` and `readonly = 'false'` gesetzt. Nach dem Drücken des 'Delegate'-Buttons und dem gleichzeitigen Schalten der Transition wird der delegierte Nutzer im Token gespeichert. Dies geschieht wieder über eine Zuweisung an dem Button, die in diesem Fall so aussehen würde:

```
<x:setvalue bind="owner" value="instance('output-token')/data/metadata/delegate" />
```

Hier wird der neue Wert für das 'owner' Element aus dem ausgehenden Token genommen und an das Element gebunden.

Nun kann auf diesen gespeicherten, neuen Bearbeiter eine Überprüfung stattfinden (nähere Erläuterung im Abschnitt 4.4.2) und die Transition `tr_submit` kann aufgrund dieser Auswertung nur von dem angegebenen Bearbeiter geschaltet werden. Loggt sich dieser ein, wird ihm diese Task in seiner Worklist angezeigt. Öffnet er diese, werden ihm die für ihn sichtbaren Formularfelder angezeigt und er kann die Aktivität bearbeiten und beenden.

Falls der ursprüngliche Bearbeiter keinen Review des Formulars wollte, wird diese Task über `tr_merge` durch `tr_complete` beendet. Wäre es der Fall, dass er einen Review wollte, so würde im Token der Wert des Elements `<isReviewed>` auf 'true' geändert. Somit würde `tr_review` schaltbereit sein und er kann das Formular nochmals begutachten. Die `consume`-Transition konsumiert dann schlussendlich auch das letzte Token, das beim Schalten von `tr_delegate` produziert wurde. Auch hier ist mit dem 'Delegate'-Button eine Zuweisung erfolgt, wodurch das Element `<isDelegated>` auf 'false' gesetzt wird und somit die Guard zu 'true' ausgewertet und `tr_consume` automatisch schaltet.

Multiple Delegation

Die grüne Einfärbung demonstriert die Multiple Delegation im Task Lifecycle (siehe A.1 im Anhang). Dementsprechend wird eine `consume`-Transition, `resetUser`-Transition und `merge`-Transition dem Petri net hinzugefügt, zuzüglich verschiedener Guards.

Nach dem gleichen Schema wie gerade eben wird nun die Schaltreihenfolge für eine mögliche Ausführung der multiplen Delegation betrachtet. Angesetzt wird wieder

Transition der Nutzer im Token auf den ursprünglichen Nutzer gesetzt werden und die Sichtbarkeits-eigenschaften zurückgesetzt werden. Wenn die getroffenen Eigenschaften ins produzierte Token übernommen werden würden, könnte der Delegierende bei der nächsten Delegation die noch ausstehenden Felder nicht mehr delegieren, weil sie ihm nicht mehr angezeigt werden. Zudem sollen delegierte Felder mit der Eigenschaft 'writeable' für die nächste Delegation nicht mehr angeboten werden. Das heißt in dem Anwendungsfall *Einstellungsantrag für einen Mitarbeiter*, wenn die Sekretärin an den Studenten das Formularfeld für die Versicherungsnummer delegiert, darf sie dieses Feld nicht mehr zum Delegieren angezeigt bekommen. Dementsprechend wird das Attribut 'writeable' für das Feld gleich 'false' gesetzt. Zur Umsetzung dieser Vorgaben existiert ein XSLT an der resetUser-Transition. Dieses XSLT, genannt reset_visibility_rights.xsl, wird in dem Abschnitt 4.4.3 näher erläutert.

Nachdem dieser Teilprozess abgeschlossen ist, kann die delegierende Person die Task ein weiteres Mal delegieren. Sie bekommt im neuen Formular nur noch die übrig gebliebenen Felder angezeigt. Wenn der Nutzer die Task an die letzte Person delegiert, wird der Wert des Elements `<isDelegated>` im Token auf 'false' gesetzt und das Token, welches den Teilprozess anstößt, konsumiert. Dadurch ist keine weitere Delegation möglich.

Nun wurde die Task an zwei Personen delegiert, beide bearbeiten ihre Teilaufgaben und beenden diese. Dadurch liegen zwei Token auf der Stelle `pl_deciding`. Gefordert für unsere Petri net-Beschreibung ist aber, dass nur ein Token aus dieser Taskinstanz herauskommt. Somit müssen diese beiden Token zusammengefasst werden. Dieser Vorgang passiert in der `merge_xslt`, welche an der `merge`-Transition angehängen wurde. Zur genauen Zusammenführung wird im Abschnitt 4.4.3 Bezug genommen. Nach dem erfolgreichen Mergen beider Token wird nun noch ausgewertet, ob die erste Person einen Review wollte. Falls ja, wird ihr wieder das Ergebnisformular angezeigt und sie kann die Task beenden, ansonsten passiert dies wieder automatisch über `tr_complete`.

Ein Problem im Petri net besteht aber noch, nämlich könnte `tr_complete` oder `tr_review` ein Eingangstoken von `tr_merge` konsumieren, aufgrund von positiv ausgewerteten Guards. Wenn dies passieren würde, käme es dazu, dass ein Token vor `tr_merge` im Netz liegen bleibt. Das Problem wird zum einem durch einem zusätzlichen Zähler (`numberOfDelegate`) abgefangen, welcher auch gleichzeitig als Guard-Bedingung dient. Wird nun die Task an zwei Personen delegiert, wird der Zähler für jeden Delegierten um eins erhöht. Dies geschieht wieder als Aktion auf das Drücken des '+'- oder 'Delegate'-Buttons. Verringert wird der Zähler bei jedem Schalten der `merge`-Transition über das `merge_xslt`. Zum anderen wird ein Element `<merged>` eingeführt, welches dann auf 'true' gesetzt wird, wenn mindestens einmal die `merge`-Transition geschaltet hat. Sie ist nötig, da beim initialen Belegen der `pl_merge` Stelle der Zähler noch gleich null ist, da diese Task noch nicht delegiert oder abgeschlossen wurde. Dadurch würde das Token konsumiert werden und es bliebe wieder ein Token im Netz liegen.

Das beschriebene Verhalten wird nun von der Ausführungsumgebung umgesetzt.

4.4 Ausführungsumgebung

Im folgenden Abschnitt wird nun die letzte Komponente aus unserer Architektur (Abbildung 4.1) näher erläutert. Dafür wird im einzelnen auf die Datenspeicherung, einerseits in der Datenbank und andererseits der Speicherung im Token selbst; die Auswertung von Rollen, Nutzer und deren Rechten und die angewandten XSLT-Transformation eingegangen.

4.4.1 Datenspeicherung

Zur Speicherung wird an Technologien eine Postgres-Datenbank und als Object-Relation-Mapper Active Record eingesetzt. Insbesondere letztere ermöglicht eine einfache Speicherung von Objekten in der Datenbank, da als Konzept dahinter steht, dass jede Datenbanktabelle eine Klasse und einzelne Datensätze Objekte dieser Klasse darstellen.

Am Beispiel der Rollen und Nutzer wird das Speichern derer in der Datenbank näher vorgestellt. Wenn nun eine Rolle in der PNML an einer Transition steht, so wird diese von der Klasse `pnml_parser` geparkt und in die Datenbanktabelle der Transition in den Eintrag der gerade bearbeiteten Transition gespeichert.

```
role = toolspecific.get_elements('./role').first
if role
  role = CGI::unescapeHTML(role.text)
  transition.role = role
end
```

Es wird aber lediglich der Name der Rolle gespeichert. Nun braucht man aber auch eine Tabelle in der die einzelnen Rollen gespeichert werden. Zusätzlich möchte man auch Nutzer speichern, wofür man auch wieder eine Tabelle benötigt. Zu dem möchte man eine m:n Beziehung zwischen Nutzer und Rollen haben, das heißt, dass man einer Rolle mehrere Nutzer und einem Nutzer mehrere Rollen zuordnen möchte. Dies erfolgt über eine weitere Tabelle (siehe 4.5) welche die Beziehungen zwischen den Nutzer und Rollen herstellt.

Zur Zeit ist es noch so, dass wir gesagt haben, dass man von Anfang an zum Durchsetzen unserer Use Cases Rollen und Nutzer fest implementieren, sprich wir fügen am Anfang jeder Schemamigrierung Rollen und Nutzer in der jeweiligen Datenbanktabelle ein.

Im Verlauf unseres Projektes kam mir die Idee einzelne Nutzer fest an eine Transition zu binden und diese in der Datenbank mit zu speichern. Dies würde sich bei der Performance positiv auswirken, da wir beim Auswerten von Nutzern direkt auf die Datenbank zugreifen könnten und nicht jedes Mal eine TokenXML durchparsen müssten, um den Nutzer zu erhalten.

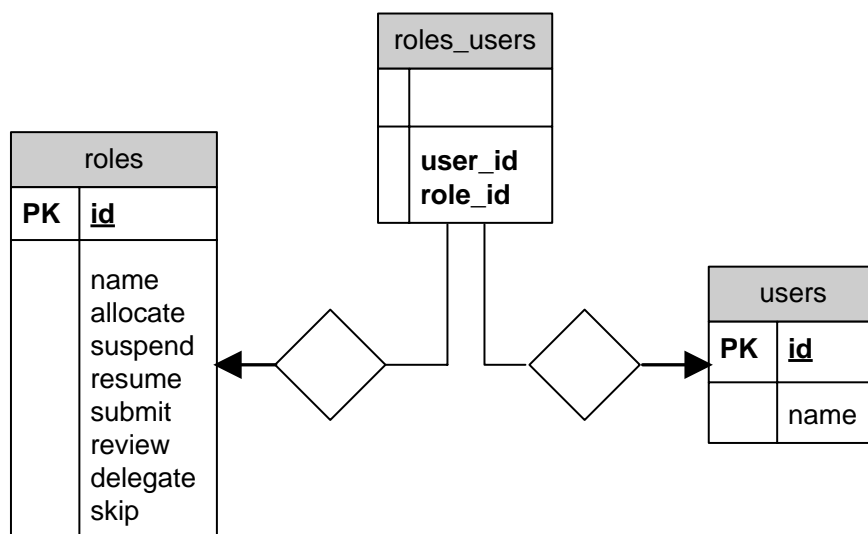


Abbildung 4.5: ER Diagramm zur Relation von Rollen und Nutzer

Aber gerade in dem festen Speichern an einer Transition lag das Problem. Man könnte beispielsweise für die Multiple Delegation nicht mehrere Eigner an der submit-Transition festlegen, da sie immer genau einer Person gehören würde. Im Anwendungsfall bedeutet das, wenn die Sekretärin eine Teilaufgaben an den Studenten und eine an den Admin delegiert, müsste der Student und der Admin als Eigner der Transition gespeichert werden. Da nun aber immer nur eine Person an einer Transition festgelegt werden kann, würde jede Auswertung auf den Nutzer für den zweiten Delegierten fehlschlagen. Er dürfe nie die submit-Transition schalten, obwohl er berechtigt sein müsste (mehr zur Auswertung im nächsten Abschnitt).

Ein großes Problem besteht auch dann, wenn man im gleichen Modell zwei Instanzen besitzt. Nun würde beispielsweise eine Instanz ihre Task delegieren. So würde die submit-Transition im Netz wie auch im Modell dem delegierten Nutzer gehören. Würde jetzt die zweite Instanz an den Punkt ankommen, wo die Transition `tr_submit` für sie schaltbereit wäre, so würde die Nutzerauswertung für diese Instanz zu 'false' ausgewertet werden und die Instanz könnte die Transition nicht schalten, obwohl sie es können müsste.

Aufgrund dieser Probleme haben wir uns darauf geeinigt, Nutzer im Token zu speichern und die Nutzerdaten aus dem Token bei der Nutzerprüfung zu verwenden.

Die Nutzerwerte werden erst zum Ende einer Task wieder aus dem Token entfernt, so dass sie für eine nachkommende Task neu ausgewertet werden können. Im Token werden die Nutzer unter dem Element

```

<executiondata>
  <owner>Testuser</owner>
</executiondata>
  
```

gespeichert.

Das Problem des mehrmaligen Parsen einer XML wird durch das Konzept der Locators verbessert. Locators binden Bezeichner an XPath-Ausdrücke², welche wiederum auf die Adresse des Elements im XML verweisen. Dadurch kann man einfach den Wert des Elements im XML für diese gebundene XPath-Adresse zurückgeben. Zusätzlich kann man diese XPath-Abfragen cachen, was dann das mehrmalige Parsen unnötig macht.

4.4.2 Auswertung von Rollen und Nutzer

Zur Auswertung einzelner Rollen und Nutzer habe ich zwei Funktionen implementiert. Beide werden benutzt, um zu erkennen, ob die eingeloggte Person die Berechtigung für verschiedene Aktionen in der Worklist besitzt, zum Beispiel um eine Task zu delegieren. Je nach Auswertungsergebnis werden die Aktionen dem Nutzer dann angezeigt oder bleiben nicht ausführbar. Diese Auswertung ist nötig, da nur der Delegierte seine Aufgabe bearbeiten und beenden kann, und nicht die delegierende Person oder eine Andere.

Bei der ersten implementierten Funktion handelt es sich um `eval_role()`. Diese hat den Nutzen, dass an jeder Transition ausgewertet werden kann, ob ein Nutzer der Rolle zugehörig ist, die die Transition schalten darf.

```
def eval_role
  return true unless role

  if $user
    $user.roles.each do |roles|
      if (roles.name == role)
        return roles.canFireTransition(self.action)
      end
    end
    return false
  end
end
```

In dieser Funktion wird geprüft, ob eine Rolle an einer Transition gesetzt wurde. Wenn dem nicht so ist, gibt die Funktion 'true' zurück. Das ist in dem Fall sinnvoll, wenn man keine Einschränkungen für eine Task zur Modellierzeit trifft, dass jeder diese Task auch ausführen darf. Danach werden alle Rollen des Nutzers betrachtet, und mit der Rolle der Transition verglichen. Hat der Nutzer die geforderte Rolle, wird in der Funktion `canFireTransition()` überprüft, ob die Rolle die Berechtigungen besitzt, um diese Transition zu schalten.

²<http://www.w3.org/TR/xpath>

```
def canFireTransition(transitionAction)
  case transitionAction
  when "trInitProcess"
    return tr_initProcess
  when "allocate"
    return allocate
  when "submit"
    return submit
  when "suspend"
    return suspend
  when "resume"
    return resume
  when "delegate"
    return delegate
  when "review"
    return review
  when "skip"
    return skip
  else
    return true
  end
end
```

In dieser Methode wird die übergebene Aktion mit den festgelegten Rechten verglichen und der Wert des übergebenen Rechtes für diese Rolle aus der Datenbank zurückgegeben. Wenn die Rolle nun das Recht besitzt, wertet die ganze Funktion zu 'true' aus.

Die zweite Funktion ist `evalOwner()`. Wie der Name schon sagt, wertet diese Funktion aus, ob der Nutzer die Transition schalten darf. Nötig wird diese Funktion bei der Delegation. Man kann eine Task an eine Person delegieren, welche nicht der Rolle der Task entspricht. Dadurch würde dann aber die Method `eval_role()` zu 'false' auswerten und der Nutzer dürfte die Task nicht bearbeiten. Deshalb braucht man noch eine zweite Funktion, welche unabhängig von der Rolle auf den Nutzer auswertet.

Nun wird diese Funktion nacheinander erklärt, dafür wird der Code von Außen nach Innen durchgegangen.

```
def eval_owner
  t_combinations = self.token_combinations

  if !(t_combinations.empty?)
    t_combinations.each do |comb|

      – siehe unteren Auszug –

    end
    return eval_role
  else
```

```

    return eval_ownerFromContextPlace
  end
end

```

Als erstes wird überprüft, ob eine Tokenkombination vor einer Transition anliegt. Tokenkombination stellen dazu ein Array von möglichen Kombinationen von Token dar. Befindet sich nun beispielsweise ein Token A auf der ersten Eingangsstelle und ein Token B auf der zweiten Stelle vom gleichen Case, dann wird eine neue Kombination dieser beiden Token in dem Feld gespeichert - nähere Informationen zu Tokenkombinationen findet man im Paper von Kai Schlichting und Andreas Lüders([10]).

Im ersten Fall der Bedingung kann es sein, dass sich kein Token vor einer Transition befindet, diese aber schaltbereit ist, weil sie die erste Transition im Netz ist. Dies bedeutet, dass in den Tokenkombinations-Array eine leere Tokenkombination vorliegt. Dadurch würde die erste each-Schleife ohne Ergebnis zurückkommen und eval_role() ausgewertet und ihr Wert zurückgegeben. Gibt es jetzt aber keine Kombination, weil beispielsweise diese Transition (für welche geprüft wird) nicht schaltbereit ist, wird durch die Funktion eval_ownerFromContextPlace() auf den Nutzer ausgewertet. In dieser wird das Token auf der Kontextstelle auf den eingeloggten Nutzer überprüft. Sobald sich ein Token vor einer Transition befindet, wird die Auswertung über das Kontrollflußtoken³ vorgenommen.

```

if !(comb.tokens.empty?)
  comb.tokens.each do |token|

    if (token.place.contenttype == "flow")

      value = getValueFromXML("owner", token.value)

      if (value && (value != ""))
        if ($user.name == value)
          return true
        end
      else
        if ((value == "") || (!value))
          return eval_role
        end
      end
    end
  end
end
return false
end

```

Nun wird jedes einzelne anliegende Token überprüft, ob es sich um ein Kontrollflußtoken handelt oder ein Token von der Kontextstelle ist. Dies kann man leicht

³Wir unterscheiden Stellen durch einen speziellen Typen. Das heißt, dass das Token, welches von der Kontextstelle kommt, ein Kontexttoken ist. Dies ist übertragbar auf die beiden anderen Typen Datentoken und Kontrollflußtoken.

erfahren, da man auf den Typen eines Tokens prüfen kann. Wenn es sich um ein Kontrollflußtoken handelt, ermöglicht die Funktion `getValueFromXML()` eine Abfrage auf den Wert des Elements im Token. Dementsprechend übergibt man ihr das gesuchte Element und das `TokenXML`, welches geparkt werden soll. Danach wird der gefundene Wert mit dem Nutzernamen verglichen, stimmt er überein, wertet die komplette Funktion zu `'true'` aus. Stimmt er nicht, wird als nächstes überprüft, ob im Token noch kein Wert gesetzt wurde. Weil sich dieses Token beispielsweise vor dem schalten der `allocate`-Transition befindet, wo noch kein Nutzer im Token aufgeführt wird. Ist dies der Fall, wird auf die Rolle des Nutzers überprüft. Es wird aber auch der Fall abgefangen, dass es zwar ein Token gibt, dieses aber keinen Inhalt besitzt. Falls nun kein Token dem eingeloggten Nutzer gehört, gibt die Funktion `'false'` zurück.

Guards

Guards sind im allgemeinen Bedingungen für Transitionen, in denen überprüft wird, ob diese Transition schalten darf. In diesen Bedingungen kann jetzt beispielsweise geprüft werden, ob der Wert des Elements `<isDelegated>` aus dem `EingangsXML` gleich `'true'` ist. Dafür werden wieder Locators benötigt, um die Adresse der Elemente im XML zu erhalten und schneller darauf zu greifen zu können. So setzt sich eine Guard-Bedingung aus der Stelle, wo der Locator definiert wurde und den vergleichenden Wert zusammen.

Dies würde in unserem Task Lifecycle für Ausdrücke wie `isDelegated == 'true'` oder `isReviewed == 'true'` nun so aussehen, dass für die Attribute `isDelegated` und `isReviewed` einzelne Locators auf der Kontextstelle definiert werden und auf diese Werte ausgewertet wird. Im PNML würden sich diese Guard im `Toolspecific`-Teil für diese Transition wiederfinden.

```
<guard>  
  <expr>place_pl_context_resource1.isDelegated = true</expr>  
</guard>
```

Wir beziehen nun den Locator `isDelegated` von der Stelle `place_pl_context_resource1` und überprüfen, ob dieser Wert des Elements gleich `'true'` ist. Wenn dem so ist, wird die Transition, für welche diese Guard definiert wurde, schaltbereit.

4.4.3 Angewandte XSL Transformationen

Nun kann nicht jedes XML ohne weiteres weitergegeben werden, sondern es muss verändert werden. Dafür gibt es die Sprache XSLT⁴, welche Transformationsregeln bereitstellt, um XML-Dokumente zu ändern. Allgemein kann man sich vorstellen, dass ein `XSLTProcessor` - eine spezielle Software um XSLT zu verarbeiten - jedes Element aus dem `EingangsXML` durchgeht und vergleicht, ob es für dieses Element

⁴<http://www.w3.org/TR/xslt>

eine Transformationsregel (xsl:template) gibt. Wenn eine solche existiert, wird das entsprechende Element aus dem eingehenden XML nach der Regel geändert und in das neue XML geschrieben.

In unserem Task Lifecycle gibt es zwei Stellen, an denen wir diese unbedingt für die Multiple Delegation benötigen. Einerseits brauchen wir sie an der Transition `tr_resetUser` und andererseits bei `tr_merge`.

reset_visibility_rights.xsl an der Transition `tr_resetUser`

Das Ziel dieser XSL ist, dass nach dem Schalten dieser Transition ein Token herauskommen soll, welches den ursprünglichen/delegierenden Nutzer als Nutzer im Token besitzt, als auch alle Felder mit der Sichtbarkeitseigenschaft 'writeable' auf 'true' gesetzt hat. Jedoch sollen die Felder, die als 'writeable' gekennzeichnet sind, auf 'false' gesetzt werden, damit sie nicht nochmals delegiert werden können. Die Umsetzung wird nun an der XSL erläutert.

```
<xsl:template match="executiondata/owner">
  <executiondata>
    <owner>
      <xsl:copy-of select="metadata/firstOwner"/>
    </owner>
  </executiondata>
</xsl:template>
```

Bei dieser Regel wird auf den Tag `executiondata/owner` des Tokens verglichen. Gibt es eine Übereinstimmung, wird ins neue Token die Struktur

```
<executiondata>
  <owner>Sekretaerin</owner>
</executiondata>
```

und der delegierende Nutzer, welcher zu der Zeit im Tag `firstOwner` gehalten wird, hinzugefügt. Würden wir diese Struktur nicht aufbauen, würde sie im neuen XML nicht erscheinen.

Um zu verdeutlichen, wie die Sichtbarkeiten zurückgesetzt werden, wird jetzt kurz der Dateninhalt des Tokens betrachtet. Für den Anwendungsfall *Einstellungsantrag für einen Mitarbeiter* sehe ein Kontrollfluss-Token wie folgt aus:

```
<data>
  <executiondata>
    <owner>Student</owner>
  </executiondata>
  <processdata>
    <name writeable="false" readonly="false">Peter</name>
```

```

    <jobType writeable='false' readonly='false'>Entwickler</jobType
    >
    <jobBeschreibung writeable='false' readonly='false'>entwickeln<
    /jobBeschreibung>
    <gehalt writeable='true' readonly='false'>1000Euro</gehalt>
    <raumnummer writeable='false' readonly='false'>123</raumnummer>
    <versicherungsnummer writeable='false' readonly='false' />
  </processdata>
</data>

```

Dieser Inhalt würde zu Stande kommen, wenn die Sekretärin eine Teilaufgabe an den Studenten mit den Einschränkungen delegieren würde, dass der Student nur das Gehalt eintragen kann. Den Rest des Formular sieht er nicht.

Nach der Transformation sollte das Token folgende Struktur besitzen:

```

<data>
  <executiondata>
    <owner>Sekretaerin</owner>
  </executiondata>
  <processdata>
    <name writeable="true" readonly="false">Peter</name>
    <jobType writeable='true' readonly='false'>Entwickler</jobType
    >
    <jobBeschreibung writeable='true' readonly='false'>entwickeln<
    /jobBeschreibung>
    <gehalt writeable='true' readonly='false'>1000Euro</gehalt>
    <raumnummer writeable='true' readonly='false'>123</raumnummer>
    <versicherungsnummer writeable='true' readonly='false' />
  </processdata>
</data>

```

Alle Sichtbarkeitseigenschaften 'writeable' sollten nun wieder auf 'true' gesetzt sein. Dies geschieht im XSL so, dass man eine Regel definiert, welche auf alle Kindknoten von <processdata> mit der Bedingung `writeable = 'false'` vergleicht. Danach wird bei allen gefundenen Elementen dieselbe Struktur aufgebaut und die Eigenschaften editierbar auf 'true' und lesbar auf 'false' gesetzt.

```

<xsl:template match="processdata/*[@writeable = 'false']">
  <xsl:element name="{name()}">
    <xsl:attribute name="writeable">true</xsl:attribute>
    <xsl:attribute name="readonly">false</xsl:attribute>
    <xsl:value-of select="current()" />
  </xsl:element>
</xsl:template>

```

Außerdem müssen noch alle Eigenschaften mit `writeable = 'true'` auf 'false' gesetzt werden.

```
<xsl:template match="processdata/*[@writeable = 'true']">
  <xsl:element name="{name()}">
    <xsl:attribute name="writeable">false</xsl:attribute>
    <xsl:attribute name="readonly">false</xsl:attribute>
    <xsl:value-of select="current()" />
  </xsl:element>
</xsl:template>
```

Danach wären alle Vorgaben ans XSLT umgesetzt.

merge_xsl an der Transition tr_merge

In dieser Transition sollte es darum gehen, zwei von unterschiedlichen Nutzern produzierte Token zu einem Token zusammenzufassen.

Eine Idee bestand darin, alle im Token `writeable = 'true'` markierten Felder ins neue Token zu übernehmen. Dadurch würde festgelegt werden, dass die delegierende Person alle Felder an verschiedene oder eine Person delegieren muss. Sie selbst dürfte keine Eingaben tätigen. Diese würden nämlich im neuen Token nicht als Attribut `writeable = 'true'` sondern `writeable = 'false'` haben.

Dieser Ansatz schränkt aber unsere Möglichkeiten durch die Sichtbarkeitseigenschaften stark ein. Der Delegierende kann keine Felder selber bearbeiten oder selbst schon teilweise Felder vorbereiten, um darauf diese zu delegieren. Um dies auch zu ermöglichen müsste das Merge_xslt erweitert werden. Im derzeit umgesetzten Fall wird dem Delegierenden die Möglichkeit gelassen, in Felder, die er nicht delegieren möchte, Eingaben vorzunehmen. Jedoch unterstützt dieses Merge-Verfahren noch nicht die Tatsache, dass die delegierende Personen Eingaben in später delegierende Felder tätigen darf. Dies könnte in Zukunft noch hinzugefügt werden. Dafür wäre nur eine Anpassung in der `merge_xslt` nötig.

In der XSL sieht unsere Realisierung nun so aus, dass auf alle Kindknoten von `<processdata>` mit der Bedingung `writeable = 'true'` verglichen wird und jedes übereinstimmende Element, in welchem auch ein Wert gesetzt wurde, in das neue Token übernommen wird.

```
<xsl:template match="processdata/*[@writeable = 'true']">
  <xsl:if test="node()">
    <xsl:copy-of select="." />
  </xsl:if>
</xsl:template>

<xsl:template match="processdata/*[@writeable = 'false']" />
```

Als Voraussetzung für dieses Verfahren gilt, dass nach dem letzten Delegierten alle Sichtbarkeitseigenschaften auf `writeable = 'true'` gesetzt werden. Dadurch wird möglich, dass selbst die bearbeiteten Felder von der delegierenden Person betrachtet werden. Würde dies nicht geschehen, würden die geänderte Felder von dem Delegierenden immer auf `writeable = 'false'` gesetzt sein, und sie würden beim Merge-Verfahren nicht berücksichtigt werden.

5. Umsetzung von und Vergleich mit SAP Konzepten

In diesem Kapitel wird Bezug zu den SAP Konzepten genommen, die in unserem Projekt gelöst werden sollten und denen, welchen unseren Funktionen ähnlich erscheinen. Spezieller soll es im ersten Abschnitt um den Anwendungsfall gehen, indem man einen zusätzlichen Genehmiger benötigt. Im nächsten Abschnitt wird die Forward Funktion von SAP mit der Delegation unseres Projektes verglichen. Im letzten Abschnitt wird das Konzept der Shared Task näher erläutert und Parallelen zu unserem Multiple Delegation gesucht.

5.1 Additional Approver

Der Anwendungsfall eines zusätzlichen Genehmigers kommt vermehrt im SRM-Bereich der SAP AG vor, dort besteht der größte Bedarf. Er tritt in den meisten Fällen beim Einkauf von Waren auf. Dabei geht es darum, dass sich eine Person beim Genehmigen eines Warenkorbs aufgrund der Summe nicht sicher ist, ob sie diesen genehmigen darf. Deshalb möchte sie beispielsweise zusätzlich ihren Chef den Warenkorb genehmigen lassen.

In dem von uns betrachteten Anwendungsfall 'Einstellung eines Mitarbeiters' wird nun erläutert, wo das Problem in diesem Fall auftritt und wie wir dies mit unseren neuen Funktionen lösen. Dieses Szenario weicht von ursprünglichen SRM-Szenarien ab, soll aber als Demonstration unserer Umsetzung dienen.

Als erstes füllt die Sekretärin den Einstellungsantrag aus und kommt an dem Punkt an, an dem sie das Gehalt festlegen soll und sie sich dessen nicht sicher ist. Deshalb möchte sie ihr eingetragenes Gehalt von ihrem Chef bestätigen lassen. Nach dem sie diese Genehmigung eingeholt hat, kann sie den Einstellungsantrag komplett ausfüllen und diesen an den Professor weitergeben. Dadurch würde der Prozess wie folgt modelliert werden (siehe 5.1)

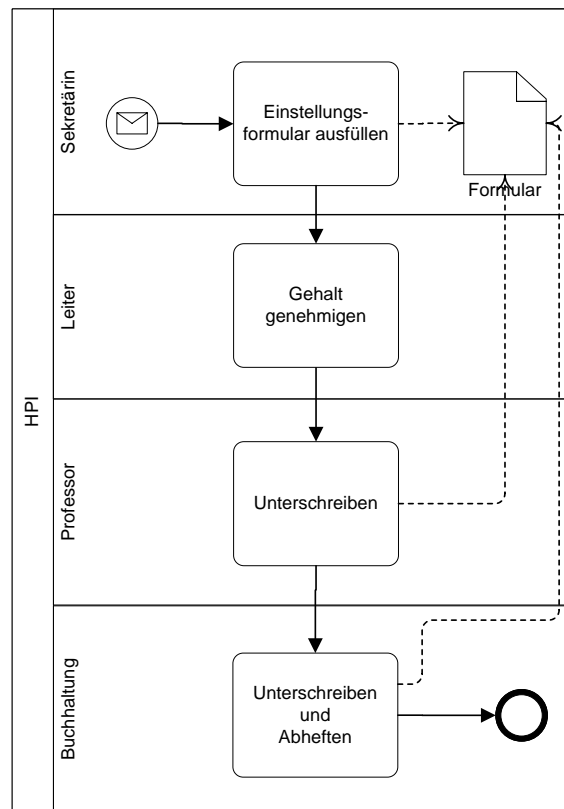


Abbildung 5.1: zusätzliche Task

Unsere Lösung dieses zusätzlichen Genehmigers besteht in der Funktion 'Delegation' und ihrer Sichtbarkeitseigenschaften. Mit dieser Funktion ist es der Sekretärin nun möglich, nur das Formularfeld Gehalt an ihren Chef zu delegieren, ohne dass er von den anderen Feldern weiß.

Speziell in diesem Beispiel sehe diese Einstellung wie folgt im UI aus. (siehe 5.2 und 5.3)

Man beachte, dass nur das Formularfeld 'Gehalt' für den delegierten editierbar markiert wird. Dadurch bekommt er auch nur dieses Feld angezeigt. Es besteht nun auch die Möglichkeit dem Delegierten mehr als nur das Gehaltsfeld anzeigen zu lassen. Es könnte auch sein, dass er für seine Entscheidungen weitere Felder sehen müsste, beispielsweise das Feld 'Job-Typ', damit er genau für diesen Job-Typ ein Gehalt abwägen könnte. Auch dies kann man über das UI lösen, in dem man dieses Feld nur als sichtbar aber nicht änderbar markiert.

Mit dieser Lösung haben wir bewusst nicht den Weg gewählt, neue Tasks in eine laufende Instanz hinzuzufügen. Damit wollten wir Probleme aus vorhergehenden Projekte vorbeugen. Dort war es meist so realisiert, dass man zur Modellierzeit kleine Anker an die Task hängen konnte, um zu kennzeichnen, dass man dort eine zusätzliche Task hinzufügen kann. Das hätte dann wieder zur Folge, dass man allen Aktivitäten einen Anker annotieren müsste, da man sich nicht sicher sein kann, wann

| | Values | Visibility Rights |
|---------------------|----------------------|-------------------|
| Name | <input type="text"/> | Invisible |
| Job-Type | Entwickler | Readonly |
| Job-Beschreibung | <input type="text"/> | Invisible |
| Gehalt | 1000 | Writable |
| Raumnummer | <input type="text"/> | Invisible |
| Versicherungsnummer | <input type="text"/> | Invisible |

+ Delegate Cancel

Abbildung 5.2: zusätzlichen Genehmiger bestimmen

bei der Ausführung eine zusätzliche Task gebraucht wird. Potentiell wäre dies an jeder Stelle im Modell. Es gibt zwar auch Möglichkeiten zu erfahren, an welcher Stelle zur Ausführung immer eine Task hinzugefügt wird. Nur kann man nie vollständig sagen, dass an einer Stelle nie eine Task hinzugefügt wird, da noch niemand es zuvor getan hat. Es ist in der Zukunft aber potentiell möglich, dass soetwas noch passiert.

5.2 Forward

Beim Forwarden geht es darum, dass man eine komplette Task an eine andere Person weitergibt, ohne diese bearbeitet zu haben. Diese Funktion ist sehr ähnlich zu unserer Delegation, wo wir auch eine Task an andere Personen weiterreichen können. Der Unterschied in beiden Funktionen besteht aber darin, dass man beim Forward die Task nicht bearbeiten kann und zusätzlich weitergeben kann. Es ist nur möglich, dass man die komplette Task ohne Bebearbeitung weiterreicht. Bei der Delegation ist beides möglich. Wir können die Task bearbeiten und dann mit dem veränderten Inhalt weiterdelegieren, oder die komplette Task weiterreichen.

5.3 Shared Task

Seit geraumer Zeit wird bei der SAP am Konzept der Shared Task (siehe [5]) entwickelt. Dieses Konzept ist von der Grundlage sehr ähnlich zu dem Multiple Delegation. Beide wollen mehrere Bearbeiter an einer Task-Instanz teilnehmen lassen. Doch gibt es kleine Unterschiede bei der Umsetzung, welche in diesem Abschnitt erläutert werden.

Bevor nun aber auf die Unterschiede gezielt eingegangen wird, werden einzelne Grundlagen beider näher erklärt.

Delegated from: Sekretarin

Values

Job-Type: Entwickler

Gehalt: 1000

Submit

Abbildung 5.3: zusätzlicher Genehmiger - Ausführung

Bei der SAP AG hat man verschiedene Task-Rollentypen, dazu zählen potentielle Bearbeiter, ein aktueller Bearbeiter, Beitragende, ein Initiator und ein Administrator. In unserem Prototypen wurde nicht speziell nach diesen Rollentypen unterschieden. Trotzdem kann man Parallelen finden. In unserer Entwicklung zählen alle Mitglieder einer Rolle zu den potentiellen Bearbeitern einer Task mit dieser Rolle. Den aktuellen Bearbeiter kann man mit dem ausführenden Bearbeiter einer Task gleich setzen. Die Beitragenden wären dann die Delegierten. Das Einladen von weiteren Beitragenden durch den aktuellen Bearbeiter würde bei uns das Delegieren darstellen.

In unserem Prototyp besteht der große Unterschied, dass wir die Beitragenden erst während der Ausführung der Task mit ihrer Aufgabe betrauen. In unserer Entwicklung gibt es deshalb auch nur genau so viele Beitragende, an die auch ein Teil der Task delegiert wird. Weitere Unterschiede sind, dass der aktuelle Bearbeiter in unserem Prototyp die delegierte Task nicht mehr in seiner Worklist sieht, wie dies bei der Shared Task der Fall wäre. Die Task würde dem aktuellen Delegierenden erst wieder angezeigt werden, wenn er einen Review der Task möchte. Zusätzlich können die Delegierten bei der Shared Task auf mehr Informationen zugreifen als der Delegierende und andere Beitragende einladen. Vergleichbar wäre diese letzte Option damit, dass eine delegierte Person ihre Aufgabe weiterdelegieren könnte. Dieses Konzept stellt aber in unserem Prototypen noch eine Schwierigkeit (siehe Kapitel 4.3 unter Multiple Delegation) dar, weshalb wir die Möglichkeit eines Delegierten nicht anbieten. Außer diesen genannten Unterschieden funktionieren die beiden Konzepte sehr ähnlich. Genauere Informationen zur Umsetzung der multiple Delegation findet man im vorherigen Kapitel.

6. Zusammenfassung

Im Laufe dieses Bachelorprojektes wurden viele Design-Entscheidungen getroffen. Manche davon waren durch Anforderungen bedingt, andere hatten prototypischen Charakter. Zu letzteren zähle ich insbesondere die initiale Festlegung von Nutzern und Rollen, oder die fehlende Möglichkeit delegierte Personen aus einer angebotenen Menge von Nutzer auszuwählen. Am Ende des Projekts stellten sich einige Entscheidungen, als nicht optimal dar. Im Task Lifecycle hätte man beispielsweise die delegate-Transition mit der pl_running Stelle verbinden können, wodurch das Netz kleiner geworden wäre. Damals haben wir uns aber aufgrund der XForms für die heutige Umsetzung entschieden.

Betrachtet man die abgelaufene Dauer, haben wir in den Anfangsmonaten viel Zeit für Definitionen und das Besprechen von verschiedenen, vorhandenen Konzepten verbraucht, bevor sich später drei Konzepte herauskristallisiert haben, welchen wir dann prototypisch umsetzen wollten. Vielleicht hätte man sich früher auf verschiedene Funktionen festlegen sollen, wodurch man mehr Zeit zum Implementieren gewonnen hätte. Damit hätte man aber wieder riskiert, dass die Konzepte nicht ausreichend gut durchdacht wären. Aber auch verschiedene technische Probleme haben Zeit gekostet. Sei es am Anfang das Einrichten des SAP Servers oder spätere Probleme mit der Petri Net Engine gewesen.

Schlußendlich kann man sagen, dass die entstandenen Lösungen neue Ansätze für die weitere Forschung mit sich bringen.

Ausblick

Gerade bei den Entwicklungen, welche nur prototypisch umgesetzt sind, kann man mit weiterer Arbeit ansetzen. So könnte man eine Möglichkeit schaffen, dass man neue Rollen mit Rechten erstellen und diesen dann Nutzern zuordnen kann, als auch bestehende Rollen und Nutzer editieren zu können. In der Worklist ist dafür bereits ein Menue vorgesehen. Zudem könnten in der Worklist noch mehrere Nutzer-

spezifische Eigenschaften angezeigt werden. Denkbar wären dort, dass man den eingeloggtten Nutzer anzeigt sowie dessen Rolle und Rechte. Weiter sollte es später möglich sein, einen Delegierten aus einer dem Nutzer angebotenen Sicht auswählen zu können. Dort könnte man sich vorstellen, dass sich ein Menü öffnet, in welchem zuerst alle Teilnehmer einer Rolle aufgelistet werden. Er könnte die Auswahl beliebig vergrößern oder verkleinern. Außerdem wird zur Zeit nur eine Delegation angeboten, sprich der Delegierte besitzt keine Möglichkeit seine Aufgabe weiterzudelegieren. In der Präsentation [13] spricht man dabei von einer Multi-step Delegation. Zu dem könnte man weitere Sicherheitseigenschaften festlegen, um zu vermeiden, dass eine Person eine Task an eine Person delegiert, welche nicht befugt ist, diese überhaupt auszuführen. Dadurch könnte man spätere Deadlock Situationen vermeiden.

A. Anhang

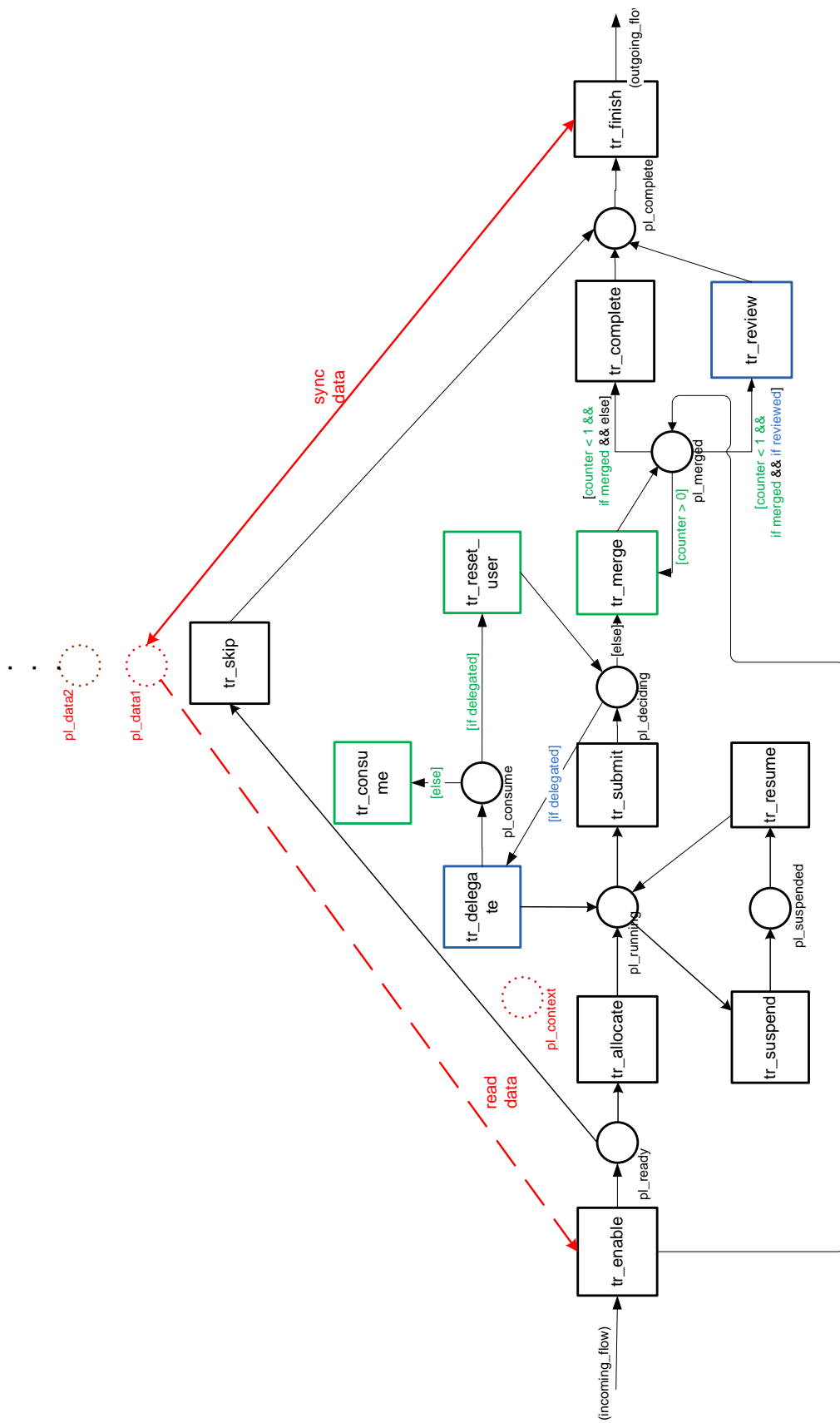


Abbildung A.1: Task Lifecycle

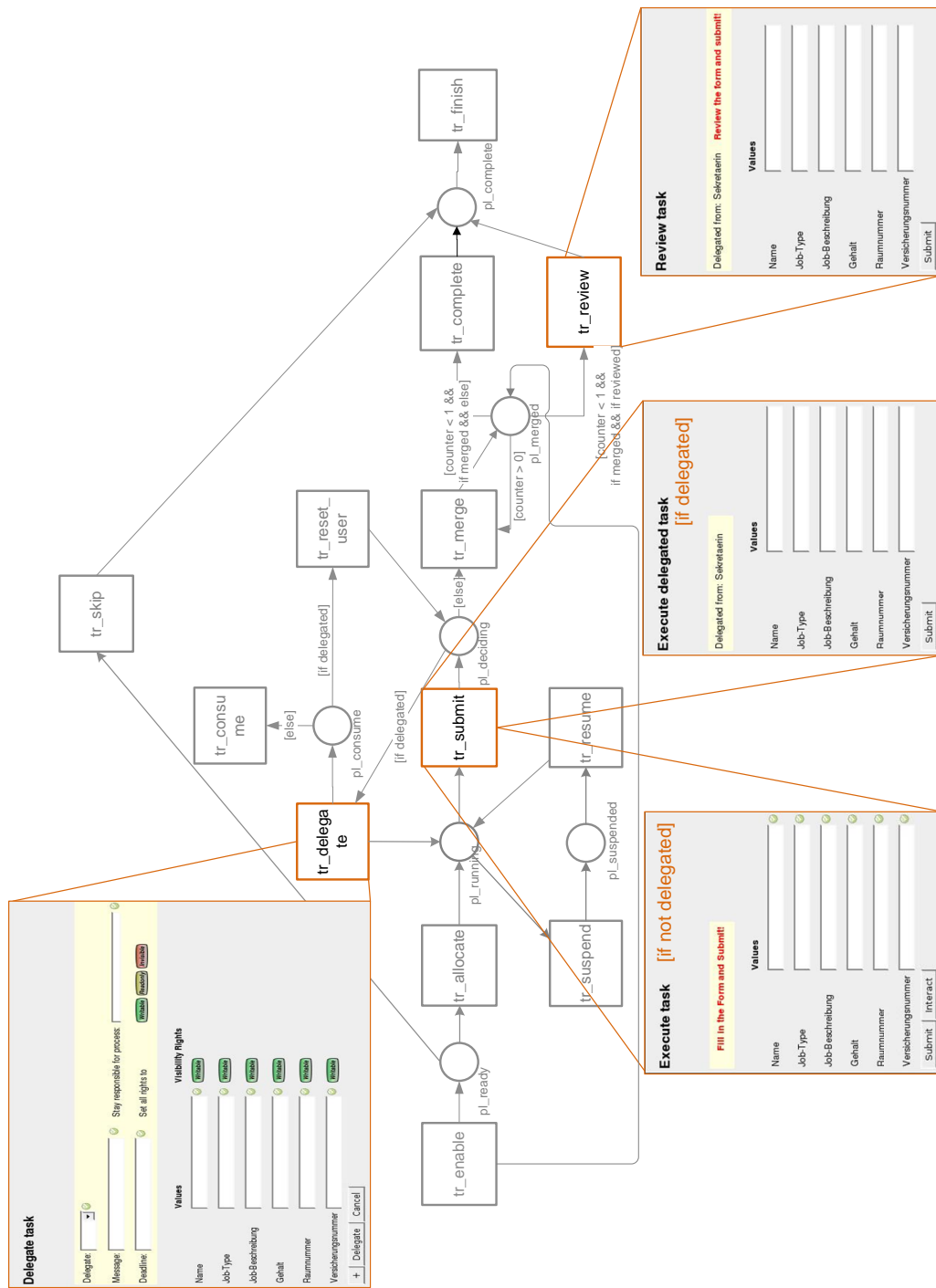


Abbildung A.2: Task Lifecycle mit Formularen

Alle Rechte sind an Transitionen und deren Aktionen gebunden. Sie gelten alle für Tasks.

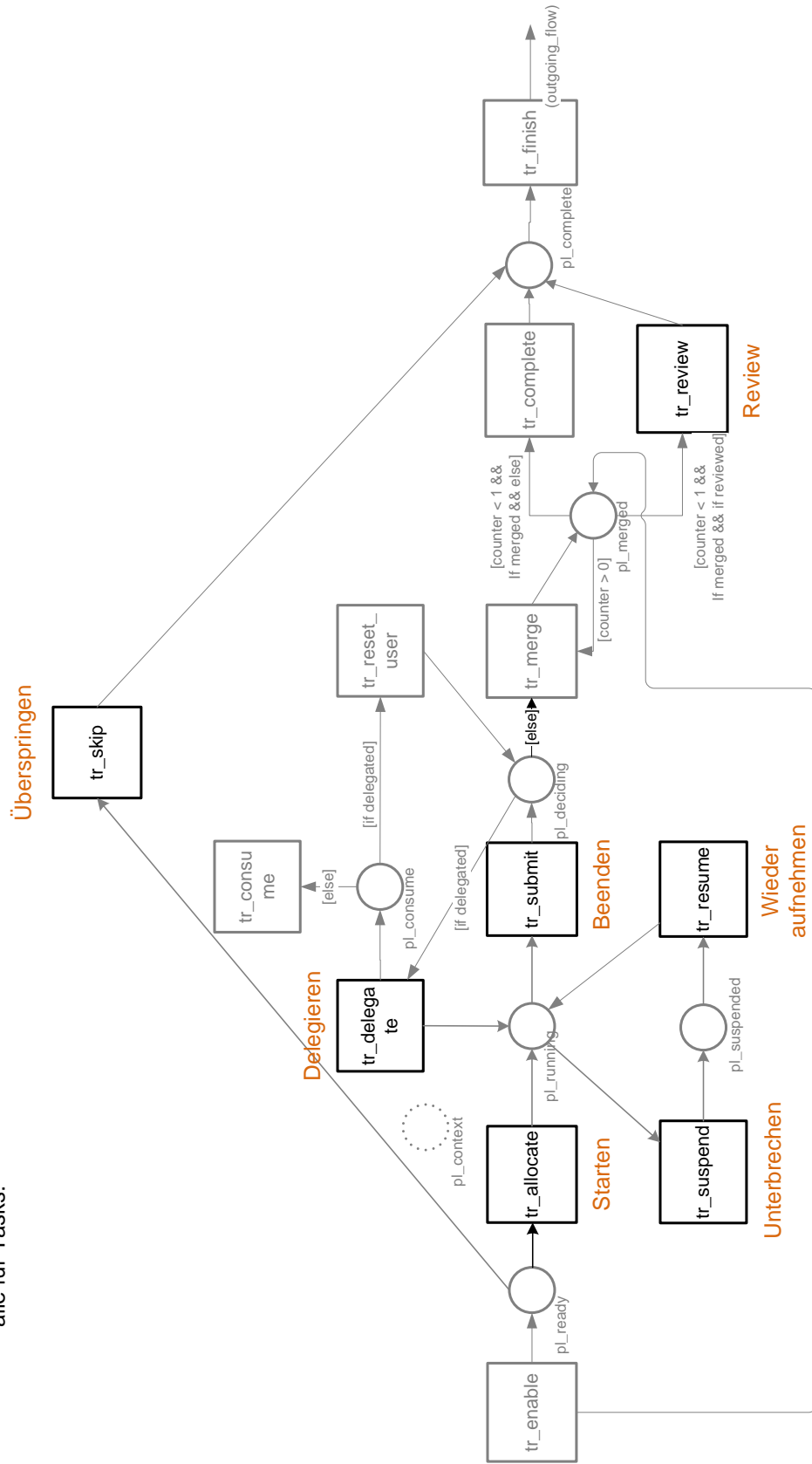


Abbildung A.3: Task Lifecycle mit Rechten

Literaturverzeichnis

- [1] Business Process Modeling Notation (BPMN) Specification, Final Adopted Specification. Technical report, Object Management Group (OMG), February 2006. <http://www.bpmn.org/>.
- [2] XForms 1.0 (Third Edition). Technical report, W3C, 2007. <http://www.w3.org/TR/xforms/>.
- [3] Jonathan Billington, Søren Christensen, Kees M. van Hee, Ekkart Kindler, Olaf Kummer, Laure Petrucci, Reinier Post, Christian Stehno, and Michael Weber. The Petri Net Markup Language: Concepts, Technology, and Tools. In *24th International Conference on the Applications and Theory of Petri Nets (ICATPN)*, LNCS, pages 483–505, Eindhoven, The Netherlands, June 2003.
- [4] Gero Decker, Lutz Gericke, Stefan Krumnow, and Mathias Weske. Prozessmodellierung und -ausführung im Web. Technical report, Hasso-Plattner Institute at the University of Potsdam, Potsdam, 2008.
- [5] BPEM EhP2. Shared Tasks. Technical report, SAP AG, 2008.
- [6] Lutz Gericke. Execution Perspective for Ad-Hoc Business Processes. Bachelor's Thesis, Hasso-Plattner Institute at the University of Potsdam, June 2008.
- [7] Matthias Kleine. Extending Galaxy Composer with Ad-Hoc Constructs. Bachelor's Thesis, Hasso-Plattner Institute at the University of Potsdam, June 2008.
- [8] Alexander Koglin. Scenarios, Usability and XForms in Ad-Hoc Business Processes. Bachelor's Thesis, Hasso-Plattner Institute at the University of Potsdam, June 2008.
- [9] Stefan Krumnow. Modeling and Executing Ad-Hoc Subprocesses in Different Environments. Bachelor's Thesis, Hasso-Plattner Institute at the University of Potsdam, June 2008.
- [10] Alexander Lüders and Kai Schlichting. RESTful Petri Net Engine. Project Report, Hasso-Plattner Institute at the University of Potsdam, February 2008.

- [11] Philipp Maschke. Execution and Re-evaluation of BPMN Processes. Bachelor's Thesis, Hasso-Plattner Institute at the University of Potsdam, June 2008.
- [12] Nicolas Peters. Oryx - Stencil Set Specification. Bachelor's Thesis, Hasso Plattner Institute at the University of Potsdam, July 2007.
- [13] Ezedin S.Barka and Ravi Sandhu. A Role-Based Delegation Model and some extensions. Technical report, George Mason University.
- [14] Andreas Schaad and Jonathan Moffett. Delegation of Obligations. Technical report, Department of Computer Science University of York, UK.