Wissensnetzwerke im Grid (WisNetGrid) - Untersuchung und Vergleich von bestehenden Workflow Werkzeugen bzgl. der Anforderungen -

Bericht zu Arbeitspaket 3.3

Juni 2011

Sudhir Agarwal (KIT), René Jäkel (ZIH), Martin Junghans (KIT), Steffen Metzger (MPII) und Bernd Schuller (FZJ)

GEFÖRDERT VOM

Bundesministerium
für Bildung
und Forschung

Inhaltsverzeichnis

Kapitel 1

Einführung

Im D-Grid Verbund gibt es keinen einheitlichen Standard zur Generierung, Manipulation und Ausführung von Workflows. Mittlerweile gibt es eine Vielzahl von Projekten mit stark unterschiedlichem Nutzerkreis und Themenschwerpunkten und dementsprechend unterschiedlichen Herangehensweisen. Ebenfalls verschieden kann die Erwartung an die Nutzerinteraktion mit der Workflow-Engine ausfallen, sei es eine hohe Flexibilität bei der Workflow-Erstellung bzw. Interaktion zu ermöglichen, oder eben nur das Anbieten von fest definierten Prozessketten mit eingeschränkter Nutzerinteraktion zu realisieren.

Für unterschiedliche Nutzerkreise haben sich unterschiedliche Systeme zur Workflowerstellung im Grid etabliert, wenn diese auch in der Regel für einen sehr speziellen Einsatz konzipiert sind aund darüber hinaus auf einer vorgegebenen Plattform arbeiten.

Im wissenschaftlichen Bereich wird oft ein sehr direkter Ansatz zur Workflowerstellung oder -modifizierung gewählt, etwa durch die Formulierung des Workflows mittels Scripten. Diese werden dann direkt auf dem Zielsystem ausgeführt, bzw. über die im Projekt verwendete Middleware submittiert. Hierbei liegt der Fokus der Anwendbarkeit auf hoher Flexibilität in der Benutzung, sowohl in der Handhabung und auch in der Veränderbarkeit der Anwendungen. Dadurch ist oftmals ein größeres Vorwissen bzw. eine gewisse Einarbeitungsphase in die Benutzung des Workflowsystems notwendig.

In kommerziellen Projekten wird ein stärkeres Augenmerk auf die Verfügbarkeit von Diensten für einen (zuvor) festgelegten Nutzerkreis gelegt,

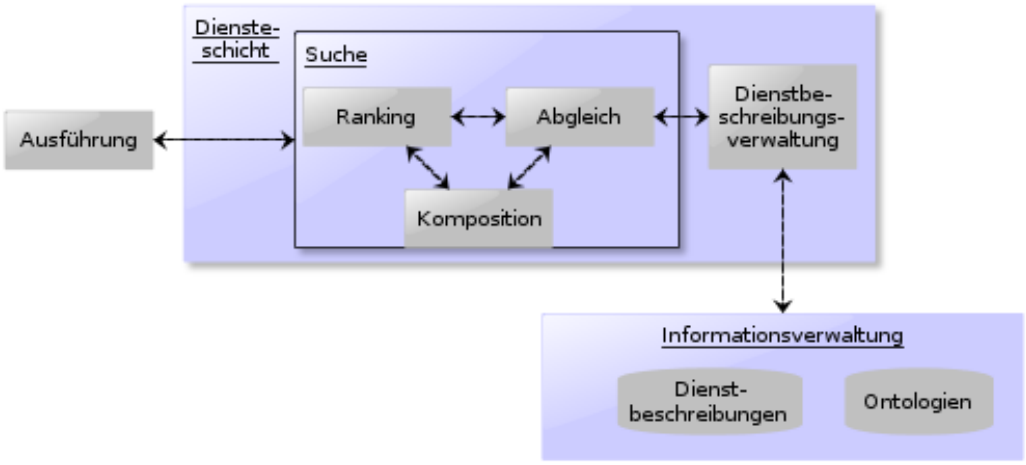


Abbildung 1.1: Konzeptionelle Darstellung der Architektur zum Einsatz von Diensten und Workflows am Beispiel der Suche.

bei gleichzeitiger Zusicherung der Eigenschaften von angebotenen Dienstleistungen. Durch diese Ausrichtung an den Nutzerkreis ergibt sich ein eingeschränkteres Anwendungsfeld. Beispielsweise lassen sich spezielle Workflows aus der Logistikbranche nicht in ein anderes Anwendungsfeld übertragen. Ebenso steht ein hohes Maß an Nuterinteraktion nicht im Vordergrund zur Anpassung des Workflows, da oftmals der Einsatz durch die Zielstellung klar vorgegeben ist.

Abbildung 1.1 illustriert das Zusammenspiel zwischen syntaktischen und semantischen Workflows. Die Ersten legen, wie oben beschrieben, starken Fokus auf die Ausführbarkeit, während die Letzteren auf das automatische Schlußfolgern über die Workflow Eigenschaften abzielen. Syntaktische Workflows werden in der Regel komplett manuell modelliert und es wird von dem Modellierer die Kenntnis über alle einzubettenden Dienste erwartet. Dies stellt eine wesentliche Herausforderung bei der Modellierung von syntaktischen Workflows dar. Die semantischen Workflows ermöglichen aufgrund von deren formalen Semantik automatische Techniken für Suche von passenden Diensten und die Komposition von Workflows. Die semantischen Workflows müssen jedoch in syntaktische Workflows transformiert werden, um im Grid ausgeführt werden zu können.

Dieses Dokuments ist wie folgt strukturiert. Im Kapitel 2 präsentieren wir einen Überblick über einige der gängigsten Sprachen für die Modellierung

von ausführbaren Workflows im Grid. Hierbei liegt der Fokus auf Verfahren, die hauptsächlich im D-Grid verwendet werden und einen generellen Ansatz aufgreifen. Wir zeigen anhand eines Beispielworkflows für Wissensextraktion aus dem TextGrid Projekt, wie das mit GWorkflowDL und mit UNICORE 6 modelliert werden kann. Die fehlende semantische Beschreibung von ausführbaren Workflows hindert die Entwicklung von automatischer Unterstützung für das Modellieren von Workflows. Demzufolge, werden solche Workflows oft vollständig manuell modelliert. Im Kapitel 3 geben wir erst einen Überblick über die im WisNetGrid entwickelte semantische Workflowsprache suprimePDL und präsentieren dann einen Ansatz für die automatische Komposition von suprimePDL Workflows. Um die automatisch komponierten Workflows ausführen zu können bedarf es entweder der Entwicklung von entsprechenden Workflowausführungsengines, oder einer Transformation in eine Sprache, für die es bereits solche gibt. In WisNetGrid wird der zweite Ansatz präferiert. Eine Transformation von suprimePDL in GWorkflowDL und in UNICORE 6 wird im Kapitel 3.4 präsentiert. Zum Schluss präsentieren wird eine Zusammenfassung sowie einen Ausblick im Kapitel 4.

Kapitel 2

Workflow-Modellierung und Ausführung im Grid

In diesem Kapitel werden wir einen Überblick über Workflowsprachen geben, die im Grid häufig verwendet werden, um ausführbare Workflows zu modellieren, zu modifizieren und auszuführen. Im Abschnitt 2.2 zeigen wir anhand eines Beispiels aus TextGrid, wie ein Workflow zur Extraktion von Fakten mit den Sprachmitteln von UNICORE 6 und GWES modelliert werden kann.

2.1 Syntaktische Modellierungssprachen

Inhalt Es existieren eine Vielzahl an Dialekten für die syntaktische Modellierung von Workflows. Neben Industriestandards wie BPEL oder XPDL werden oft Eigenentwicklungen wie GWES oder UNICORE 6 Workflows verwendet, die besser an die Bedürfnisse von Anwendern im Grid angepasst sind. In diesem Abschnitt werden die genannten vier Workflow-Dialekte und ihre Ausführungsumgebungen kurz vorgestellt.

2.1.1 BPEL BPEL-[S]WS

Die Business Process Execution Language (BPEL) wurde explizit zur Komposition von SOAP/WSDL-basierten Web Services entworfen. Ein BPEL Workflow wird statisch in XML beschrieben, und wiederum als Web Service zur Verfügung gestellt. BPEL ist eng an die Konzepte von WSDL gebunden. So müssen Variablen mittels XML Schema definiert werden, und die ausführbaren Elemente des Workflows sind Aufrufe von SOAP/WSDL Web Services. BPEL Workflows können somit als relativ starre Vorschriften zur Orchestrierung von Web Services aufgefasst werden.

Im D-Grid hat sich das BISGrid-Projekt mit der Komposition von Grid-Diensten mithilfe von BPEL befasst. Dabei wurden Globus4- und UNICORE-Web Services im Rahmen von BPEL Workflows aufgerufen. Die Grid-typischen Aufgaben, wie Job-Submission, Status-Monitoring oder das Verarbeiten der Ergebnisdateien, liessen sich als recht komplexe Teil-Workflows in BPEL darstellen.

Obwohl es somit möglich ist, Grid-Dienste und ihre Interaktionen mit dem Benutzer und untereinander in BPEL zu modellieren, sind die resultierenden Workflows sehr komplex, und verbergen dadurch etwas die wissenschaftlichen Fragestellungen, die ja mithilfe des Grid behandelt werden sollen.

2.1.2 XPDL

Die XML Process Definition Language ist als XML-Notation für die Beschreibung von allgemeinen Geschäftsprozessen entworfen, und beinhaltet Konzepte wie verschiedene Aktoren und Rollen sowie verschiedene Kontrollstukturen und Datentypen. XPDL ist nicht an ein bestimmtes Ausführungsmodell (etwa Web Services) gebunden, sondern kann an verschiedene, vom Bereitsteller der XPDL-Engine definierte Ausführungsmechanismen, gebunden werden. Damit ist XPDL eine abstrakte Beschreibung, die sich sowohl zur Modellierung von Geschäftsprozessen, als auch wissenschaftlicher Arbeitsabläufe, eignet. Dennoch spielt XPDL im Grid-Umfeld (vielleicht zu Unrecht) keine Rolle.

2.1.3 Generic Workflow Execution Service (GWES)

Der Generic Workflow Execution Service (GWES) wurde von Fraunhofer FIRST im Rahmen des europäischen "K-Wf Grid" Projektes entwickelt. Die Beschreibungssprache *Generic Workflow Description Language (GWorkflowDL)* ist eine generische Sprache zur Beschreibung von verteilten Workflows. GWorkflowDL basiert auf den höher-stufigen Perti-Netzen (HLPN, refer to ISO/IEC 15909-1), die aus einer Menge von Stellen (visualisiert durch Kreise), eine Menge von Transitionen (visualisiert durch Quadrate) und einer Menge von Flussbeziehungen (visualisiert durch Pfeile von Stellen nach Transitionen bzw. von Transitionen nach Stellen). Zudem wird eine Stelle mit der Angabe über die maximal erlaubte Zahl der Marken in der Stelle versehen.

Im Gegensatz zu gewöhnlichen Petri-Netzen sind die Marken der HLPNs voneinander unterscheidbar und können verwendet werden, um höher-stufige Werte, wie echte Ein- und Ausgabe-Daten, Referenzen auf Daten (z.B. Dateinamen oder URLs) und Boolesche Werte, die Nebeneffekte darstellen, zu modellieren. Die Verteilung von Marken auf den Stellen wird als Markierung bezeichnet und stellt den Zustand des verteilten Systems dar. Wenn eine aktivierte Transition erfolgt (feuert), dann wird jeweils eine Eingabe-Marke von jeder Eingang-Stelle entnommen und jeweils eine neue Ausgabe-Marke an jeder Ausgang-Stelle platziert.

Das Modul GWorkflowDL umfasst eine Java-Bibliothek für den Umgang mit Workflow-Darstellungen, wie dem Erstellen, Verfeinerung und Lesen/Schreiben von XML-Workflow Darstellungen. Insbesondere basieren die Operationen zur Ausführung von GWES auf der GworkflowDL Bibliothek. Der Entwurf von Schnittstellen- und Implementierungsklassen erleichtert den Austausch von Implementierungen sowie zukünftige Optimierungen.

Eine Besonderheit des GworkflowDL ist die Unterstützung von verschiedenen Abstraktionsebenen von Transitionen oder Operationen. Die Abstraktesten repräsentieren reine Kontrollfunktionalität und die am wenigsten abstrakte die ausführbare Grid-Operationen, die mit den Werkzeugen wie WCT, AAB und Scheduler gebaut werden. Es können derzeit SOAP Web Services und Globus4-Dienste verwendet werden, jedoch ist GWES durch eigene Plugins erweiterbar.

2.1.4 UNICORE 6 Workflows

In der UNICORE-Middleware ist ein integriertes Workflow-System verfügbar, das einen grafischen Editor und mehrere Web Services beinhaltet. Dieses erlaubt die Ausführung von Workflows, die aus einzelnen Grid-Jobs und diversen Kontrollkonstrukten (Schleifen, bedingter Ausführung, etc) zusammengesetzt sind. Es ist optimiert an die Verhältnisse im Grid und berücksichtigt Belange wie Sicherheit, Datentransfers und Batch-Jobs. Die Beschreibungssprache ist ein graphen-orientierer XML-Dialekt, der die einzelnen Prozess-Schritte und ihre Abhängigkeiten beschreibt. Der Vollständigkeit halber sei angemerkt, dass die Workflow-Engine im Prinzip auch die Verwendung anderer XML-Dialekte erlaubt, sofern ein passender Konverter zur Verfügung gestellt wird.

Das ausführende System besteht aus zwei logischen Schichten, die als Workflow-Engine und Service Orchestrator bezeichnet werden. Dabei kümmert sich die Workflow-Engine um den Ablauf des Prozesses, verwaltet Variablen und bietet dem Klienten Interfaces zur Submission und Monitoring von Workflows. Die eigentlichen Prozess-Schritte (Tasks) korrespondieren zu UNICORE-Jobs. Sie werden zum Service Orchestrator weitergeleitet, der sich um die alle Aspekte der Ausführung kümmert. Insbesondere wird hier auch ein geeignetes UNICORE-System zur Ausführung ausgewählt (resource brokering). Es ist auch möglich, die Systeme vorab auszuwählen.

Zur Zeit erlaubt das UNICORE Workflow-System nicht, beliebige Dienste (etwa SOAP/WSDL Web services) im Rahmen des Workflows auszuführen, sondern ist auf UNICORE-Jobs beschränkt.

Im D-Grid ist die UNICORE Workflow-Umgebung verfügbar und wird etwa im neuen "MMM@HPC" Projekt verwendet.

Die UNICORE Workflow-Beschreibung

Die UNICORE 6 Workflows bestehen aus sogenannten Activities verschiedener Art, die beispielsweise einem UNICORE-Job entsprechen können, sowie aus Kontrollkonstrukten, wie Schleifen, Variablen und bedingten Übergängen. Mithilfe dieser Elemente können beliebig komplexe Workflows beschrieben werden. Die Workflow-Beschreibungssprache ist in XML Sche-

ma definiert, hier soll ein kurzer Überblick über die verschiedenen Elemente gegeben werden.

- Activity. Eine Activity beschreibt einen Knoten des Workflow-Graphen. Es gibt verschiedene Activities, beispielsweise beschreibt eine Activity vom Typ *JSDL* einen UNICORE-Job. Jede Activity hat einen eindeutigen Bezeichner, der es erlaubt, die Activity zu referenzieren.
- Transition. Eine Transition beschreibt den Übergang zwischen Knoten (Activities oder SubWorkflows). Optional kann eine Übergangsbedingung angegeben werden, die von der Workflow Engine zur Laufzeit ausgewertet wird.
- SubWorkflow. Ein SubWorkflow bezeichnet einen Block von Elementen, ähnlich einer Kontrollstruktur in einer prozeduralen Programmiersprache wie Java oder C. Es gibt insbesondere Schleifen (while, for-each, repeat-until). Jeder SubWorkflow hat einen eindeutigen Bezeichner, der es erlaubt den SubWorkflow zu referenzieren.
- Daten. Da UNICORE Batchjob-orientiert arbeitet, werden müssen Eingangsdaten und Resultate als Dateien behandelt werden. Für jeden Batchjob werden die benötigten Eingangsdaten vor dem Job-Start zum ausführenden System transferiert. Die Ausgabedateien werden üblicherweise ebenfalls zu einem vordefinierten Speichersystem wegkopiert.

2.2 UNICORE 6 und GWES am Beispiel von Wissensextraktion

Als Beispiel wollen wir hier den Ablauf der Wissensextraktion[3] heranziehen. In vereinfachter Form lässt sich das Extraktionsverfahren auf abstrakter Ebene als 4-stufiger Workflow darstellen (siehe Abbildung 2.1). Startet ein Nutzer das Extraktionsverfahren auf einem Quelldokument, wird in einem ersten Schritt der im Dokument enthaltene Text tokenisiert. Der resultierende *Tokenstream* wird im nächsten Schritt zur Erkennung von Vorkommen benannter Entitäten, wie Personen, Ländern, Filmen etc. im Text verwendet. Hierbei wird einerseits ein *Index über die Vorkommen textueller Entitätsreferenzen*, inklusive der jeweiligen Position im Text, aufgebaut und andererseits werden bei mehrdeutigen Referenzen *mögliche Disambiguierungen*,

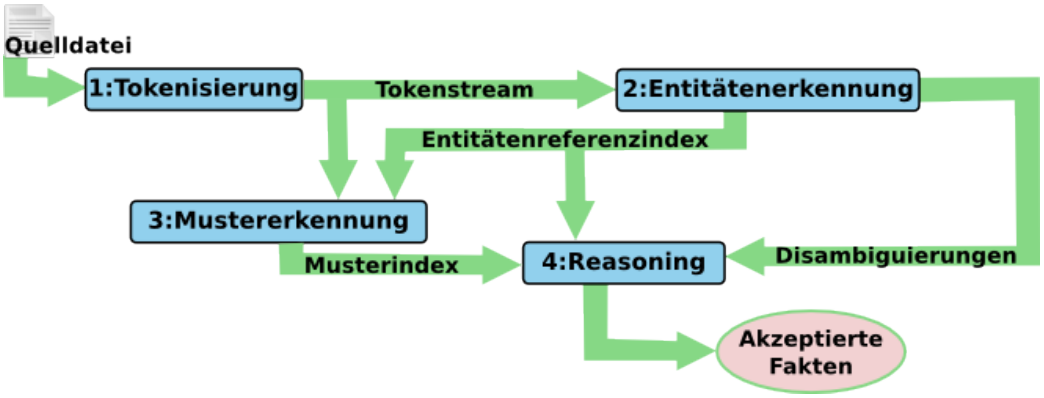


Abbildung 2.1: Workflow der WisNetGrid-Extraktionskomponenten

also Zuordnungen der Referenzen auf je eine eindeutige Entität, generiert. Aufbauend auf dem Index über die Entitätenvorkommen werden dann Textmuster, die möglicherweise eine Beziehung zweier Entitäten ausdrücken, auf dem urpsrünglichen Tokenstream identifiziert. Die Vorkommen solcher generischer Textmuster mit bestimmten Entitätsreferenzen im Text, sowie ihre möglichen Interpretationen als Relationen, werden sodann als *Musterindex* zusammen mit den zuvor schon erzeugten Vorschlägen zur Entitätsdisambiguierung und den Entitätenvorkommen einer Reasoningkomponte übergeben. Diese Komponente entscheidet einerseits welche Textmuster als Indikatoren für bestimmte Relationen akzeptiert werden können und welches die wahrscheinlichste Auflösung von mehrdeutigen Entitätsbezügen im Text unter dem gegebenen Kontext ist. Andererseits generiert die Komponente, mit Hilfe des erlangten Wissens um aussagekräftige Textmuster, aus den erkannten Vorkommen dieser Muster im aktuellen Text mögliche Fakten über die enthaltenen Entitäten. Nach einer logischen Überprüfung auf Konsistenz dieser generellen Faktenkandidaten wird eine Menge an neuen *akzeptierten Fakten* ausgegeben.

Tabelle ?? ist ein alternativer Vorschlag zur Illustration welcher Output wo wieder als Input landet

2.2.1 Wissensextraktion als UNICORE 6 Workflow

Abbildung 2.2 zeigt den Kontrollfluss und den Datenfluss unseres Wissensextraktionsbeispiels in der graphischen Notation von UNICORE 6. Lis-

Input	Bearbeitungsschritt	Output
Quelltext	Tokenisierung	Token-Stream
Quelltext, Token-Stream, Ontologisches Hintergrundwissen: Namensbedeutungen, Entitätsbeziehungen (Relationsintanzen)	Entitätenerkennung	Index über Entitätenvorkommen, Faktenvorschläge zur Entitätendisambiguierung
Token-Stream, Index über Entitätenvorkommen, Ontologisches Hintergrundwissen:Entitätstypen, Relationsparameterty-pen, Relationsinstanzen, Bekannte Muster	Mustererkennung	Index über Mustervorkommen, Faktenvorschläge für indikative Muster
Index über Mustervorkommen, Faktenvorschläge zu Entitätendisambiguierung und für indikative Muster, Ontologisches Hintergrundwissen: Entitätstypen, Relationsparameterty-pen, Entitätsbeziehungen, Bekannte Muster	Reasoning	Akzeptierte (und gegebenenfalls verworfene) Fakten

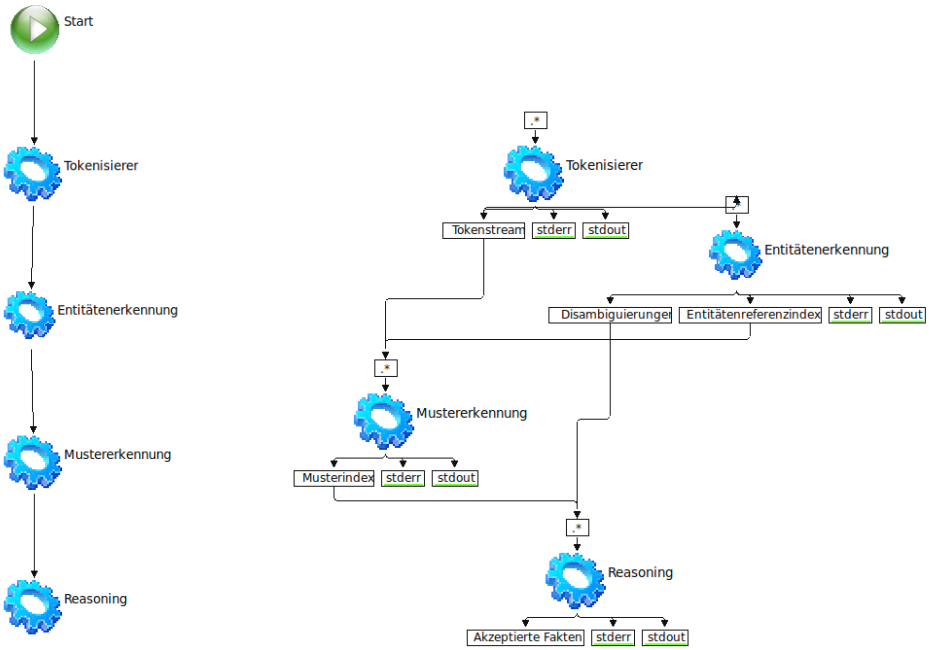
Tabelle 2.1: Workflowschritte des Extraktionssystems: Semantischer In- und Output

Input	Bearbeitungsschritt	Output
Textdokument(HTML, XML, TXT,...)	Tokenisierung	Tabseparierte-Werte Datei (.TSV)
Textdokument, .TSV Datei, Ontologiezugang	Entitätenerkennung	.TSV Datei (Entitätenvorkommen), .TSV Datei (Disambiguierungsvorschläge) (alternativ: Ontologieeinträge des Konzepts 'Entitätenvorkommen' und der Relation 'hatDisambiguierungsvorschlag')
.TSV Datei(Token-Stream), .TSV Datei(Entitätenvorkommen), Ontologiezugang	Mustererkennung	.TSV Datei (Mustervorkommen), .TSV Datei(Indikative Muster) (alternativ: Ontologieeinträge des Konzepts 'Mustervorkommen' und der Relation 'möglicherIndikatorFürRelation')
.TSV Datei(Mustervorkommen), .TSV Datei(Disambiguierungsvorschlä .TSV Datei(Indikative Muster), Ontologiezugang	Reasoning	UI-Ausgabe: Akzeptierte (und gegebenenfalls verworfene) Fakten (alternativ: Ontologieeinträge)

Tabelle 2.2: Workflowschritte des Extraktionssystems: Syntaktischer In- und Output

Input	Bearbeitungsschritt	Output
Quelltext	Tokenisierung	Token-Stream
Quelltext, Token-Stream, Namensbedeutungen (via Ontologie)	Entitätenerkennung	Index über Entitätenvorkommen, Faktenvorschläge zur Entitätendisambiguierung
Token-Stream, Index über Entitätenvorkommen, Ontologisches Hintergrundwissen	Mustererkennung	Index über Mustervorkommen, Faktenvorschläge für indikative Muster
Index über Mustervorkommen, Faktenvorschläge zur Entitätendisambiguierung und für indikative Muster, Ontologisches Hintergrundwissen	Reasoning	Akzeptierte Fakten (ggfs. verworfene Fakten)

Tabelle 2.3: Workflowschritte des Extraktionssystems: Semantischer In- und Output mit Argumenthighlighting



(a) Kontrollfluss des Workflows für Wissensextraktion

(b) Datenfluss des Workflows für Wissensextraktion

Abbildung 2.2: Wissensextraktion-Workflow als UNICORE6 Workflow

ting 2.2 zeigt den gesamten UNICORE 6 Workflow für das Wissensextraktionsbeispiel in XML.

Listing 2.1: XML Serialisierung des UNICORE 6 Workflows für Wissensextraktion

```
<?xml version="1.0" encoding="UTF-8"?>
<sim:Workflow xmlns:sim="http://www.chemomentum.org/workflow/simple">

    <sim:Activity Id="Start" Type="START" Name="START"/>

    <sim:Activity Id="Tokenisierer" Type="JSDL" Name="JSDL">
        <sim:JSDL id="Tokenisierer">
            <jsdl:JobDescription xmlns:jsdl="http://schemas.ggf.org/jsdl/2005/11/jsdl">
                <jsdl:JobIdentification>
                    <jsdl:JobName>Tokenisierer</jsdl:JobName>
                </jsdl:JobIdentification>
                <jsdl:Application>
                    <jsdl:ApplicationName>Tokeniser</jsdl:ApplicationName>
                    <jsdl1:POSIXApplication xmlns:jsdl1="http://schemas.ggf.org/jsdl
                        /2005/11/jsdl-posix">
                        <jsdl1:Environment name="INPUT">Quelldatei</jsdl1:Environment>
                    </jsdl1:POSIXApplication>
                </jsdl:Application>
                <jsdl:DataStaging>
                    <jsdl:FileName>Quelldatei</jsdl:FileName>
                    <jsdl:CreationFlag>overwrite</jsdl:CreationFlag>
                    <jsdl:Source>
                        <jsdl:URI>c9m:${WORKFLOW_ID}/Tokenisierer/Quelldatei</jsdl:URI>
                    </jsdl:Source>
                </jsdl:DataStaging>
                <jsdl:DataStaging>
                    <jsdl:FileName>Tokenstream</jsdl:FileName>
                    <jsdl:CreationFlag>overwrite</jsdl:CreationFlag>
                    <jsdl:Target>
                        <jsdl:URI>c9m:${WORKFLOW_ID}/Tokenisierer/Tokenstream</jsdl:URI>
                    </jsdl:Target>
                </jsdl:DataStaging>
            </jsdl:JobDescription>
        </sim:JSDL>
    </sim:Activity>

    <sim:Activity Id="Entitaetenerkennung" Type="JSDL" Name="JSDL">
        <sim:JSDL id="Entitaetenerkennung">
            <jsdl:JobDescription xmlns:jsdl="http://schemas.ggf.org/jsdl/2005/11/jsdl">
                <jsdl:JobIdentification>
                    <jsdl:JobName>Entitaetenerkennung</jsdl:JobName>
                </jsdl:JobIdentification>
                <jsdl:Application>
                    <jsdl:ApplicationName>Entitaetenerkennung</jsdl:ApplicationName>
                    <jsdl1:POSIXApplication xmlns:jsdl1="http://schemas.ggf.org/jsdl
                        /2005/11/jsdl-posix">
                        <jsdl1:Environment name="INPUT">Tokenisierer_Tokenstream</
                            jsdl1:Environment>
                    </jsdl1:POSIXApplication>
                </jsdl:Application>
                <jsdl:DataStaging>
                    <jsdl:FileName>Tokenisierer_Tokenstream</jsdl:FileName>
                    <jsdl:CreationFlag>overwrite</jsdl:CreationFlag>
                    <jsdl:Source>
                        <jsdl:URI>c9m:${WORKFLOW_ID}/Tokenisierer/Tokenstream</jsdl:URI>
                    </jsdl:Source>
                </jsdl:DataStaging>
                <jsdl:DataStaging>
                    <jsdl:FileName>Entitaetenreferenzindex</jsdl:FileName>
                    <jsdl:CreationFlag>overwrite</jsdl:CreationFlag>
                    <jsdl:Target>
                        <jsdl:URI>c9m:${WORKFLOW_ID}/Entitaetenerkennung/
                            Entitaetenreferenzindex</jsdl:URI>
                    </jsdl:Target>
                </jsdl:DataStaging>
                <jsdl:DataStaging>
                    <jsdl:FileName>Disambiguierungen</jsdl:FileName>
                    <jsdl:CreationFlag>overwrite</jsdl:CreationFlag>
                    <jsdl:Target>
                        <jsdl:URI>c9m:${WORKFLOW_ID}/Entitaetenerkennung/Disambiguierungen</
                            jsdl:URI>
                    </jsdl:Target>
                </jsdl:DataStaging>
            </jsdl:JobDescription>
        </sim:JSDL>
```

```
</sim:Activity>

<sim:Activity Id="Mustererkennung" Type="JSDL" Name="JSDL">
    <sim:JSDL id="Mustererkennung">
        <jsdl:JobDescription xmlns:jsdl="http://schemas.ggf.org/jsdl/2005/11/jsdl">
            <jsdl:JobIdentification>
                <jsdl:JobName>Mustererkennung</jsdl:JobName>
            </jsdl:JobIdentification>
            <jsdl:Application>
                <jsdl:ApplicationName>Mustererkennung</jsdl:ApplicationName>
                <jsdl1:POSIXApplication xmlns:jsdl1="http://schemas.ggf.org/jsdl
                    /2005/11/jsdl-posix">
                    <jsdl1:Environment name="INPUT_1">Tokenisierer_Tokenstream</
                        jsdl1:Environment>
                    <jsdl1:Environment name="INPUT_2">
                        Entitaetenerkennung_Entitaetenreferenzindex</jsdl1:Environment>
                </jsdl1:POSIXApplication>
            </jsdl:Application>
            <jsdl:DataStaging>
                <jsdl:FileName>Tokenisierer_Tokenstream</jsdl:FileName>
                <jsdl:CreationFlag>overwrite</jsdl:CreationFlag>
                <jsdl:Source>
                    <jsdl:URI>c9m:${WORKFLOW_ID}/Tokenisierer/Tokenstream</jsdl:URI>
                </jsdl:Source>
            </jsdl:DataStaging>
            <jsdl:DataStaging>
                <jsdl:FileName>Entitaetenerkennung_Entitaetenreferenzindex</
                    jsdl:FileName>
                <jsdl:CreationFlag>overwrite</jsdl:CreationFlag>
                <jsdl:Source>
                    <jsdl:URI>c9m:${WORKFLOW_ID}/Entitaetenerkennung/
                        Entitaetenreferenzindex</jsdl:URI>
                </jsdl:Source>
            </jsdl:DataStaging>
            <jsdl:DataStaging>
                <jsdl:FileName>Musterindex</jsdl:FileName>
                <jsdl:CreationFlag>overwrite</jsdl:CreationFlag>
                <jsdl:Target>
                    <jsdl:URI>c9m:${WORKFLOW_ID}/Mustererkennung/Musterindex</jsdl:URI>
                </jsdl:Target>
            </jsdl:DataStaging>
    </sim:JSDL>
</sim:Activity>

<sim:Activity Id="Reasoning" Type="JSDL" Name="JSDL">
    <sim:JSDL id="Reasoning">
        <jsdl:JobDescription xmlns:jsdl="http://schemas.ggf.org/jsdl/2005/11/jsdl">
            <jsdl:JobIdentification>
                <jsdl:JobName>Reasoning</jsdl:JobName>
            </jsdl:JobIdentification>
            <jsdl:Application>
                <jsdl:ApplicationName>Reasoning</jsdl:ApplicationName>
                <jsdl1:POSIXApplication xmlns:jsdl1="http://schemas.ggf.org/jsdl
                    /2005/11/jsdl-posix">
                    <jsdl1:Environment name="INPUT_1">Mustererkennung_Musterindex</
                        jsdl1:Environment>
                    <jsdl1:Environment name="INPUT_2">
                        Entitaetenerkennung_Disambiguierungen</jsdl1:Environment>
                </jsdl1:POSIXApplication>
            </jsdl:Application>
            <jsdl:DataStaging>
                <jsdl:FileName>Mustererkennung_Musterindex</jsdl:FileName>
                <jsdl:CreationFlag>overwrite</jsdl:CreationFlag>
                <jsdl:Source>
                    <jsdl:URI>c9m:${WORKFLOW_ID}/Mustererkennung/Musterindex</jsdl:URI>
                </jsdl:Source>
            </jsdl:DataStaging>
            <jsdl:DataStaging>
                <jsdl:FileName>Entitaetenerkennung_Disambiguierungen</jsdl:FileName>
                <jsdl:CreationFlag>overwrite</jsdl:CreationFlag>
                <jsdl:Source>
                    <jsdl:URI>c9m:${WORKFLOW_ID}/Entitaetenerkennung/Disambiguierungen</
                        jsdl:URI>
                </jsdl:Source>
            </jsdl:DataStaging>
            <jsdl:DataStaging>
                <jsdl:FileName>Akzeptierte Fakten</jsdl:FileName>
                <jsdl:CreationFlag>overwrite</jsdl:CreationFlag>
                <jsdl:Target>
                    <jsdl:URI>c9m:${WORKFLOW_ID}/Reasoning/Akzeptierte Fakten</jsdl:URI>
                </jsdl:Target>
            </jsdl:DataStaging>
        </jsdl:JobDescription>
```

```
        </sim:JSDL>
    </sim:Activity>

    <sim:Transition Id="df2cd092-0735-4806-b53b-68f1a0dd8710" From="Start" To="Tokenisierer"
        />
    <sim:Transition Id="b947d1ac-3a98-4e81-9744-f6a1e9284f92" From="Tokenisierer" To="
        Entitaetenerkennung"/>
    <sim:Transition Id="3134a15e-e55d-411e-a208-83c448bb7964" From="Entitaetenerkennung" To=
        "Mustererkennung"/>
    <sim:Transition Id="4cd282e9-0c21-40da-b064-ebb9f747796a" From="Mustererkennung" To="
        Reasoning"/>

</sim:Workflow>
```

2.2.2 Wissensextraktion als GWES Workflow

Listing ?? zeigt den gesamten GWES Workflow für das Wissensextraktionsbeispiel in XML und Abbildung ?? zeigt den Kontrollfluss des Wissensextraktionsbeispiels und deren Komponenten in grafischer Form.

Listing 2.2: XML Serialisierung des GWES Workflows für Wissensextraktion

```
<?xml version="1.0" encoding="UTF-8"?>
<workflow xmlns="http://www.gridworkflow.org/gworkflowdl"
          xmlns:xsd="http://www.w3.org/2001/XMLSchema"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://www.gridworkflow.org/gworkflowdl http://www.
              gridworkflow.org/kwfgrid/src/xsd/gworkflowdl_2_1.xsd
                    http://www.gridworkflow.org/gworkflowdl/operationclass http://
    www.gridworkflow.org/kwfgrid/src/xsd/gworkflowdl_operationclass_2_0.xsd"
          ID="No_ID">

  <description>Ein GWES Modell des Workflows der Extraktionskomponenten</description>
  <property name="occurrence.sequence" />

  <place ID="Quelldokument">
    <token ID="d1">
      <data>
        <value xmlns="" xsi:type="xsd:string">Albert Einstein's Geburtsort ist Ulm</value>
      </data>
    </token>
  </place>

  <place ID="Tokenstream" />
  <place ID="TokenizationDone" />
  <place ID="Entitaetenreferenzindex" />
  <place ID="Entitaetendisambiguierungen" />
  <place ID="EntitaetenerkennungDone" />
  <place ID="Musterindex" />
  <place ID="MustererkennungDone" />
  <place ID="ErkannteFakten" />

  <transition ID="Tokenizer">
    <description>Tokenizes texts</description>
    <property name="icon.url">http://www.gridworkflow.org/gworkflowdl/images/exec.png</
        property>
    <inputPlace placeID="Quelldokument" edgeExpression="Text" />
    <outputPlace placeID="Tokenstream" edgeExpression="*" />
    <outputPlace placeID="TokenizationDone" />
    <condition>string-length($value) &gt; 0</condition>
    <condition>$params/@xsi:type = "xsd:string"</condition>
  </transition>

  <transition ID="Entitaetenerkennung">
    <description>Findet Entitaetenreferenzen im Tokenstream</description>
    <property name="icon.url">http://www.gridworkflow.org/gworkflowdl/images/exec.png</
        property>
    <inputPlace placeID="Tokenstream" edgeExpression="in" />
    <inputPlace placeID="TokenizationDone" edgeExpression="true" />
    <outputPlace placeID="Entitaetenreferenzindex" edgeExpression="*" />
    <outputPlace placeID="Entitaetendisambiguierungen" edgeExpression="*" />
    <outputPlace placeID="EntitaetenerkennungDone" />
```

```
␣␣</transition>

␣␣<transition␣ID="Mustererkennung">
␣␣␣␣<description>Findet␣Textmuster␣im␣Tokenstream</description>
␣␣␣␣<property␣name="icon.url">http://www.gridworkflow.org/gworkflowdl/images/exec.png</
    property>
␣␣␣␣<inputPlace␣placeID="Tokenstream"␣edgeExpression="in"␣/>
␣␣␣␣<inputPlace␣placeID="Entitaetenreferenzindex"␣edgeExpression="in"␣/>
␣␣␣␣<inputPlace␣placeID="TokenizationDone"␣edgeExpression="true"␣/>
␣␣␣␣<inputPlace␣placeID="EntitaetenerkennungDone"␣edgeExpression="true"␣/>
␣␣␣␣<outputPlace␣placeID="Musterindex"␣edgeExpression="*"␣/>
␣␣␣␣<outputPlace␣placeID="MustererkennungDone"␣/>
␣␣</transition>

␣␣<transition␣ID="Reasoning">
␣␣␣␣<description>Generiert␣neue␣Fakten␣und␣ueberprueft␣diese␣auf␣Konsistenz</description>
␣␣␣␣<property␣name="icon.url">http://www.gridworkflow.org/gworkflowdl/images/exec.png</
    property>
␣␣␣␣<inputPlace␣placeID="Entitaetenreferenzindex"␣edgeExpression="in"␣/>
␣␣␣␣<inputPlace␣placeID="Musterindex"␣edgeExpression="in"␣/>
␣␣␣␣<inputPlace␣placeID="Entitaetendisambiguierungen"␣edgeExpression="in"␣/>
␣␣␣␣<inputPlace␣placeID="MustererkennungDone"␣edgeExpression="true"␣/>
␣␣␣␣<inputPlace␣placeID="EntitaetenerkennungDone"␣edgeExpression="true"␣/>
␣␣␣␣<outputPlace␣placeID="ErkannteFakten"␣edgeExpression="*"␣/>
␣␣</transition>

</workflow>
```

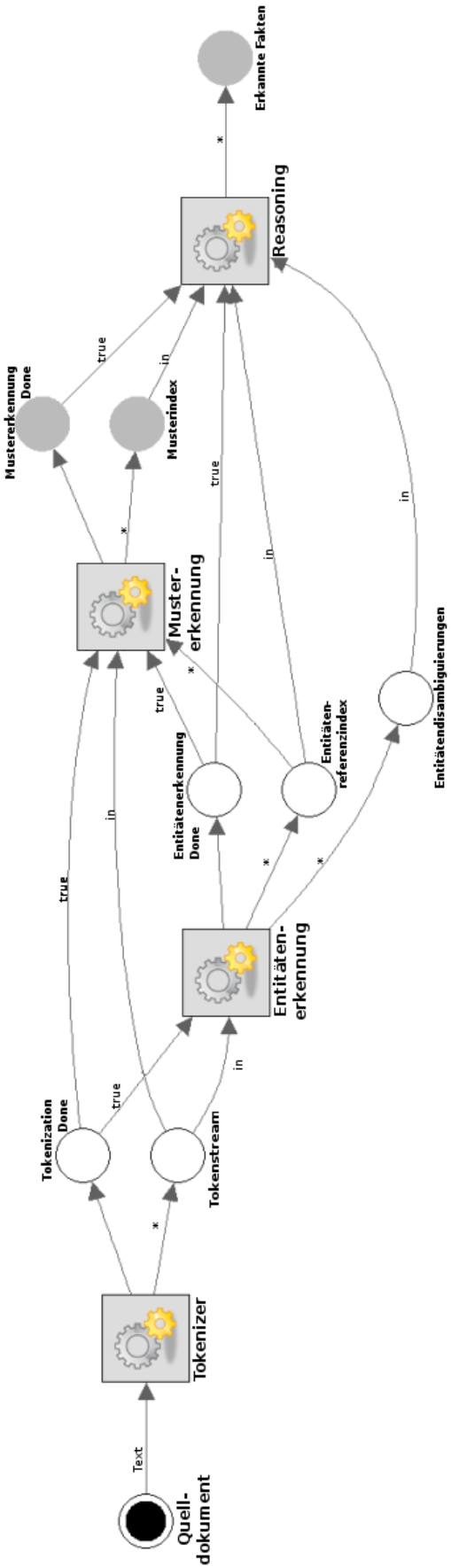


Abbildung 2.3: Visualisierung des GWES Workflows

Kapitel 3

Komposition von semantischen Workflows

In Kapitel 2 haben wir unter Anderem gesehen, wie ein Workflow semantisch modelliert werden kann. Ein wesentlicher Vorteil von semantisch beschriebenen Workflows ist, dass automatische Verfahren zum Schlussfolgern über Workflow-Eigenschaften entwickelt werden können. In diesem Kapitel präsentieren wir einen Ansatz zur automatischen Komposition von semantischen Workflows aus semantisch beschriebenen Diensten, die die Rolle der Komponenten des Workflows übernehmen.

Für eine gegebene Anfrage, die die Eigenschaften eines gewünschten Workflows beschreibt, berechnet der Kompositionsalgorithmus den Kontrollfluss und Datenfluss zwischen den relevanten Diensten, so dass der Workflow die in der Anfrage beschriebenen Eigenschaften besitzt. Da die Dienste im allgemeinen von unabhängigen Akteuren angeboten werden und auch nicht von Benutzern geändert werden können, kann ein auf Diensten basierter Workflow lediglich aus Koordination von Dienstaufrufen mit richtigen Parametern bestehen.

Im Abschnitt3.2 zeigen wir wie ein solcher koordinierender Prozess strukturiert und mit unserer semantischen Workflowmodellierungssprache *suprimePDL* modelliert werden kann. Da das manuelle Modellieren von insbesondere komplexen Workflows aufwändig und Fehler anfällig sein kann, präsentieren wir im Abschnitt 3.3 ein Algorithmus zur automatischen Komposition von semantischen Workflows.

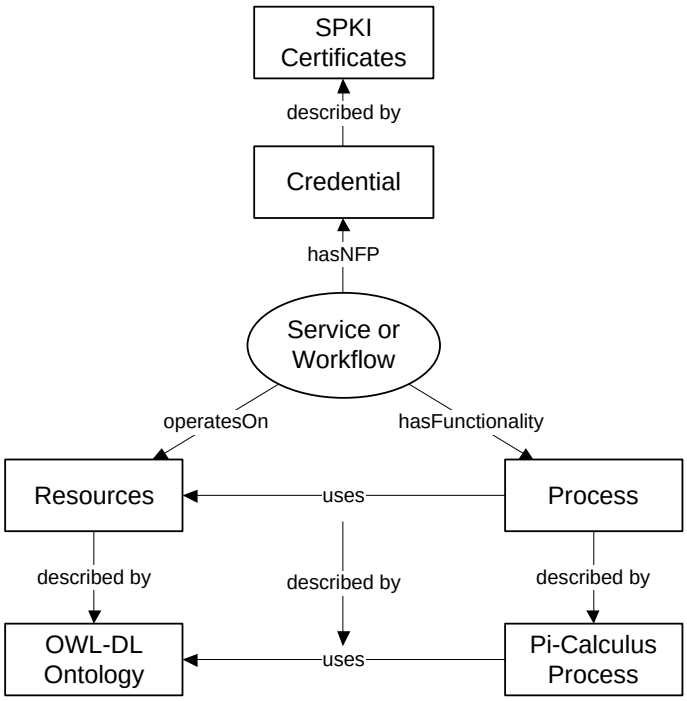


Abbildung 3.1: Abstraktes Modell von Diensten und Workflows

3.1 Semantische Modellierung von Workflows

In diesem Abschnitt präsentieren wir einen Überblick über unseren Ansatz für die semantische Modellierung von Workflows. Im Abschnitt 3.1 präsentieren wir eine Zusammenfassung der Sprache *suprimePDL*, die in [1] ausführlich erlärt wurde. Im Abschnitt **??** zeigen wir wie die semantisch beschriebenen Workflows in WisNetGrid und D-Grid verwendet werden können, um den Mehrwert von semantischen Beschreibungen für D-Grid Community Projekte erzielen zu können.

In diesem Abschnitt präsentieren wir einen Überblick über unsere semantische Workflowsmodellierungssprache. Abbildung 3.1 zeigt auf, dass eine semantische Workflowsbeschreibung die dynamischen Aspekte, die statischen Aspekte und nachweisbare Eigenschaften eines Dienstes erfasst, um sowohl über die funktionalen als auch über die nicht-funktionalen Eigenschaften eines Workflows automatisch Schlussfolgerungen ziehen zu können.

Eine semantische Workflowbeschreibung enthält die folgenden Parameter

1. URI der Dienstbeschreibung: Die URI wird vom WisNetGrid-System bei der Registrierung der Dienstbeschreibung vergeben und an den Benutzer zurückgegeben. Sie ist nicht Teil der vom Dienstmodellierer veränderbaren Parameter.

2. Name des Dienstes: Von dem Dienstmodellierer kann ein String eingegeben werden, der den Dienst bezeichnet. Der Name dient mehr dem Zweck der Beschreibung eines Dienstes als der Identifikation, da der Name nicht eindeutig ist, d.h. es kann mehrere Dienste geben, die den gleichen Namen tragen. Zur Identifikation dient ausschließlich die URI. Dementsprechend ist der Name auch zur Bearbeitung durch Benutzer mit den entsprechenden Zugriffsrechten freigegeben.

3. Beschreibungstext / Freitext: Der Beschreibungstext dient der natürlichsprachlichen Beschreibung des Dienstes. Er kann alle Informationen enthalten, die für andere Benutzer verständlich wiedergegeben werden sollen. Er unterliegt hierbei keinerlei Beschränkungen, kann aber außer durch die Suchkomponente bei einer expliziten Suche in den Freitextannotationen der Dienstbeschreibung nicht weiter maschinell verarbeitet werden. Durch den Beschreibungstext erfüllen wir die von unseren Use-Case Communitys gestellte Anforderung 16.

4. Schlüsselwörter: Unter dem Parameter Schlüsselwörter können Stichpunkte gespeichert werden, die den Dienst weiter beschreiben. Diese Wörter müssen explizit in den Ontologien in WisNetGrid enthalten sein, d.h. die Menge der Schlüsselwörter einer Dienstbeschreibung ist eine Menge von URIs von Ontologiebegriffen. Eine String-Repräsentation, die mit dem Ontologiebegriff einer URI verknüpft ist, kann angezeigt werden. Hierdurch wird z.B. ermöglicht, dass, falls ein Begriff in verschiedenen Sprachen in der Ontologie gespeichert ist, nur der Begriff für den Benutzer angezeigt wird, der in seiner Sprache ist. Durch die ausschließliche Verwendung von Ontologie-Begriffen erlauben die Schlüsselwörter eine maschinell-verarbeitbare Beschreibung des Dienstes. Diese Verarbeitung kann auch das Reasoning und ähnliche Funktionen durch die Ontologiedienste ermöglichen.

   Diese hier vorgestellte Art der Verwendung von Schlüsselwörtern sorgt für die Erfüllung der Anforderungen 2, 3, 5 und 18 durch die Beschreibungssprache (siehe dazu auch Bericht zu den Anforderungen der semantischen Dienstbeschreibungssprache [2]).

5. Metadaten: Metadaten erlauben weitere Spezifikation bzgl. der Eigenschaften der Dienstbeschreibung. Durch sie sind z.B. Autor und Erstel-

lungsdatum der Beschreibung referenzierbar. Auch diese können bei der Suche nach Diensten verwendet werden. Dienstbeschreibungen können Metadaten zu den folgenden Elementen enthalten:

- *modificationDate* Das Datum der letzten Veränderung der Dienstbeschreibung wird gespeichert.
- *changeLog* Das changelog stellt eine Dokumentation zu den Änderungen an der Dienstbeschreibung dar.
- *creationDate* Das Datum der Erstellung der Dienstbeschreibung wird festgehalten.
- *language* Die Sprache der Dienstbeschreibungen, die z.B. in den Freitexten benutzt wurde, wird gespeichert.
- *domain* Die Domäne des Dienstes, der in der Beschreibung dargestellt wird, wird durch dieses Element repräsentiert.
- *contributors* Die UserIDs der Benutzer, die Änderungen an der Beschreibung vornehmen, werden gespeichert.

6. Zertifikate: Die Erfassung aller nicht-funktionalen Eigenschaften und der Bewertung der Dienste erfolgt durch Zertifikate: Jeder Dienstmodellierer besitzt einen *private key* und einen *public key*, mit dem er Dienstbeschreibungen verschlüsseln bzw. signieren kann. Durch die eindeutige und vertrauenswürdige Signatur mit dem private key können auch Eigenschaften eines Dienstes signiert werden. Wir nennen dies ein Zertifikat. Es wird so sichergestellt, dass das Zertifikat vom Dienstmodellierer selbst stammt, da nur er seinen private key kennt und mit diesem verschlüsseln kann. Die Verschlüsselung bzw. Signatur kann von allen überprüft werden, indem sein public key zur Entschlüsselung benutzt wird. Ein Zertifikat kann entweder für alle Benutzer von WisNetGrid, für eine Gruppe von Benutzern oder einzelne Benutzer einsehbar sein, indem der jeweilige public key zur Verschlüsselung verwendet wird.[1]

   Durch dieses Verfahren sind die Eigenschaften und Bewertungen eines Dienstes vertrauenswürdig und der Benutzer kann sich für eine bestimmte Menge an Zertifikaten entscheiden, die er bei der Betrachtung von Diensten berücksichtigen will.

7. Prozessbeschreibung: Falls sich die Beschreibung auf eine Dienstekomposition und nicht auf einen atomaren Dienst bezieht, können die ato-

[1] Für mehr Informationen zu Verschlüsselungs- und Zertifizierungsverfahren mit public und private keys siehe [?].

maren Dienste und ihre Zusammenhänge beschrieben werden. Kompositionen werden durch Verkettung bzw. Verknüpfung von Dienstbeschreibungs-URIs beschrieben. Eine Komposition/ein Prozess P kann dann in Anlehnung an den π-Kalkül[2] [6, 7, 8, 9] wie folgt beschrieben werden:

$$\begin{aligned} P \quad ::= \quad & \mathbf{0} \mid \mathsf{y}[\mathsf{v_1}, ..., \mathsf{v_m}].P \mid \mathsf{y}\langle \mathsf{z_1}, ..., \mathsf{z_m}\rangle.P \mid \Delta.P \mid \\ & \omega?P \mid P_1||P_2 \mid P_1 + P_2 \mid @A\{\mathsf{y_1}, ..., \mathsf{y_n}\} \end{aligned}$$

Der Prozess **0**bezeichnet ein Prozess, der nichts tut und wird zum Terminierung eines Prozessausdrücks verwendet. Der Eingabeprozess $\mathsf{y}[\mathsf{v_1}, ..., \mathsf{v_m}].P$ empfängt über den Kanal y n Eingabewerte, bindet sie an die Variablen $v_1, v_2, ...v_m$ und verhält sich dann wie der Prozess P. Der Ausgabeprozess $\mathsf{y}\langle \mathsf{z_1}, ..., \mathsf{z_m}\rangle.P$ liefert über den Kanal y die Ausgabewerte $z_1, ..., z_m$ und verhält sich dann wie der Prozess P. Der Prozess $\Delta.P$ führt erst die in Δ beschriebenen Änderungen in der beschriebenen Reihenfolge und verhält sich dann wie der Prozess P. Der bedingte Prozess $\omega?P$ verhält sich wie P wenn die Bedingung ω wahr ist, sonst wie **0**. $P_1||P_2$ stellt eine parallele Komposition von den Prozessen P_1 und P_2 und $P_1 + P_2$ eine Auswahl zwischen den Prozessen P_1 und P_2 dar. $@A\{\mathsf{y_1}, ..., \mathsf{y_n}\}$ stellt den Aufruf eines mit A benannten Prozesses mit den Parametern $y_1, ..., y_n$. Einem Prozess kann ein Name oder Bezeichner zugeordnet werden, indem dem Namen ein Prozessausdrück als seine Definition zugewiesen wird, z.B. $A(x_1, ..., x_n) =_{def} P$.

Zwischen den Prozessbeschreibungen und den in WisNetGrid hinterlegten Ontologien besteht eine direkte Verbindung: Die oben erwähnten Ein- und Ausgabeparameter sind Ontologiebegriffe. Hierzu müssen über die Ontologieverwaltung so genannte Domänen-Ontologien der Communitys verwaltet werden. In diesen Ontologien können die Begriffe für mögliche Werte von z.B. den Ein- und Ausgabeparametern aufgenommen und zueinander in Beziehung gesetzt werden. Durch eine Ontologie die diese Information enthält, ist eine semantische Weiterverarbeitung dieser Informationen z.B. für die Komposition möglich (vgl. [10]).

[2]Die Prozessalgebra π-Kalkül erlaubt die Beschreibung von Prozessen, in denen sich die Kommunikation zwischen den Prozesskanälen dynamisch ändern kann. Hierbei wird insbesondere der Datenverkehr innerhalb eines Prozesses genau so beschrieben wie der zwischen zwei unterschiedlichen Prozessen. Dies ist vor allem bei dem hier vorgestellten Ansatz von Nutzen, in dem die Beschreibung eines atomaren Prozesses exakt so wie die Beschreibung einer Komposition erfolgt.

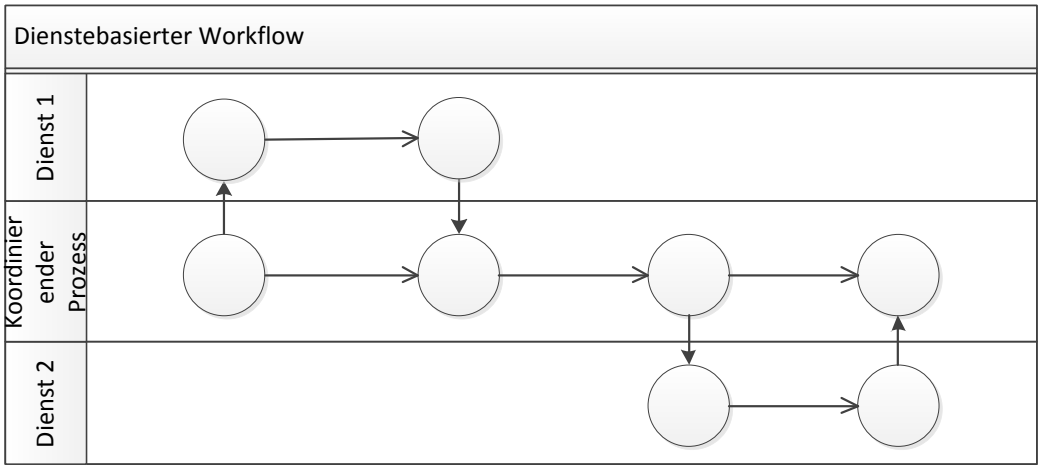


Abbildung 3.2: Synthesized solution template with a filter derived from a constraint.

3.2 Struktur eines koordinierenden Prozesses

Abbildung 3.2 zeigt ein Beispiel eines koordinierenden Prozesses, der zwei Dienste in einer Sequenz aufruft. Im allgemeinen, muss ein koordinierender Prozess mindestens folgende Funktionalität unterstützen können:

- ein koordinierender Prozess muss Dienste je nach Bedarf sequentiell, parallel oder bedingt aufrufen können. Wenn kein Dienst allein aber mehrere Dienste zusammen alle benötigten Werte bereitstellen können, ohne dass ein Dienst von einem anderem abhängt, bedarf es ~~der~~ einer parallelen Schaltung der Dienste. Ist ein Dienst jedoch abhängig von einem anderen Dienst, z.B. wenn die Eingabewerte eines Dienstes erst nach dem Aufruf eines anderen Dienstes vorliegen, müssen die Dienste sequentiell geschaltet werden.

- ein koordinierender Prozess muss den Datenfluss mit dem Benutzer, und den zwischen verschiedenen Diensten, unterstützen können. Wenn mehrere Dienste denselben Eingabewert vom Benutzer benötigen, soll der Benutzer nur einmal zur Eingabe aufgefordert werden. Der koordinierende Prozess soll die Ausgaben der Dienste als Zwischenwerte speichern, so dass sie jeder Zeit bei Bedarf als Eingabe für andere Dienste verwendet werden können.

- ein koordinierender Prozess soll Funktionen zum Filtern und Aggregieren der Werte unterstützen. Wenn die Komponentendienste eines Workflows jeweils eine Teilmenge der gesamten Ergebnismenge liefern, bedarf es in manchen Fällen die Ergebnismenge zu filtern, z.B. das Maximum oder Minimum aller Werte zu berechnen oder zu aggregieren, z.B. den Durchschnitt zu berechnen.

Im Allgemeinen, besteht ein koordinierender Prozess aus mehreren parallel geschalteten Threads. Jeder solcher Threads kommuniziert mit höchstens einem Dienst. Die Dienste laufen parallel zu dem koordinierenden Prozess. Das heißt für einen koordinierenden Prozess C und Komponentdienste $D_1, \ldots, D_n$ bezeichnet $C \parallel D_1 \parallel \ldots \parallel D_n$ den gesamten Workflow. Da der koordinierende Prozess auch ein Prozess ist, dessen Ausdrückmächtigkeit von unserer semantischen Prozessmodellierungssprache *suprimePDL* gedeckt werden kann, können koordinierende Prozess wie Dienste ebenfalls mit *suprimePDL* semantisch beschrieben werden.

3.3 Automatische Komposition von Workflows

Algorithm 1 composeWorkflows

Require: Desired properties $(\mathcal{I}, \mathcal{O}, \Gamma)$ and a set of workflows $\mathcal{S}$

1: **if** $\Gamma = \emptyset$ **then**
2: Stop!
3: select one $\gamma \in \Gamma$
4: let $\mathcal{S}' \subseteq \mathcal{S}$ denote those workflows that do not satisfy γ
5: **if** $\mathcal{S}' \neq \emptyset$ **then**
6: $\mathcal{P}_{match} \leftarrow match(\gamma)$
7: addMatchingServicesToWorkflows
8: **else**
9: remove γ from Γ
10: $composeWorkflows(\mathcal{S}, (\mathcal{I}, \mathcal{O}, \Gamma))$

Für eine gegebene Anfrage berechnet der Kompositionsalgorithmus eine Menge von Workflows, so dass jeder Workflow in der Ergebnismenge die in der anfrage spezifizierten Eigenschaften hat. Die Anfrage, die entweder direkt von Benutzer oder von einer anderen Komponente spezifiziert werden kann, beschreibt gewünschte Eigenschaften der gesuchten Workflows als ein Tupel $(\mathcal{I}, \mathcal{O}, \Gamma)$, wobei $\mathcal{I}$ eine Menge der gewünschten Eingaben, $\mathcal{O}$ eine Menge der

gewünschten Ausgaben und Γ eine Menge von Restriktionen und Relationen zwischen Ein- und Ausgaben darstellt.

Algorithmus 1 beschreibt grob die Vorgehensweise zur automatischen Komposition von Workflows. Wir nehmen an, dass wir eine Methode *match* zur Verfügung stellen, um für einen Term $\gamma \in \Gamma$ die Dienste zu finden, die die mit γ spezifizierte Eigenschaft haben. Der Algorithmus 1 ist ein Plan-Algorithmus [5] (Space Planning)und nimmt eine Anfrage und eine Menge der partiell komponierten Workflows entgegen. Der Algorithmus wählt dann zufällig eine Eigenschaft γ aus der Menge Γ und ruft die Methode *match* mit γ als Parameter auf. Jeder Dienst, der die Eigenschaft γ besitzt und somit in der Ergebnismenge der Methode *match* liegt, stellt eine potentielle Komponente für alle bereits komponierten Workflows dar. Mit ~~der~~ Algorithmus 2 wird jedem partiell komponierten Workflow jeder Dienst aus der Ergebnismenge hinzugefügt. Dies ergibt eine neue Menge der partiell komponierten Workflows, so dass jeder Workflow in dieser Menge die Eigenschaft γ erfüllt. Die Eigenschaft γ kann dann aus der Menge der zu erfüllenden Eigenschaften entfernt werden, so dass diese Menge immer kleiner wird, außer wenn die Vorbedingungen eines Komponentendienstes in die Menge hinzugefügt werden. Nach diesem Schritt ruft sich der Algorithmus 1 rekursiv mit der neuen Menge der zu erfüllenden Eigenschaften und der neuen Menge der partiell komponierten Workflows auf. Dies geschieht so lange bis die Menge der zu erfüllenden Eigenschaften leer ist. Am Anfang wird der Algorithmus 1 mit Parametern Γ und $\emptyset$ (keine bereits komponierten Workflows) aufgerufen. Wenn der Algorithmus 1 terminiert, sind alle möglichen Workflows berechnet worden, so dass jeder Workflow die in Γ spezifizierten Eigenschaften besitzt. Das Endergebnis bekommt man, indem aus der Menge der komponierten Workflows nur die ausgewählt werden, die die Eingaben $\mathcal{I}$ und die Ausgaben $\mathcal{O}$ haben. Diese Auswahl kann mit Hilfe von der Methode *match* zur Modellüberprüfung getroffen werden.

Nachdem die groben Schritte für die automatische Komposition von Workflows feststehen, gehen wir jetzt auf einen wichtigen Schritt zum Hinzufügen eines Komponentdienstes etwas genauer ein. Der Algorithmus 2 bekommt eine Menge von bereits komponierten partiellen Workflows $\mathcal{S}$ und eine Menge von Diensten $\mathcal{P}_{match}$. Der Algorithmus 2 berechnet dann für den Workflow $S \in \mathcal{S}$ und jeden Dienst $P \in \mathcal{P}_{match}$ einen neuen Workflow, indem der Dienst P mit dem koordinierenden Prozess C des Workflows S in parallel geschaltet wird. Als Interaktionspartner mit dem Dienst P wird ein neuer Thread in C parallel zu bereits existierenden Threads hinzufügt. Dieser Thread ist zuständig für das Bereitstellen von Eingaben an, sowie Empfangen von Aus-

Algorithm 2 addMatchingServicesToWorkflows

Require: Set of workflows $\mathcal{S}$ and a set of matching services $\mathcal{P}_{match}$

for all $P \in \mathcal{P}_{match}$ **do**
  for all $S \in \mathcal{S}'$ **do**
    copy S to S'
    let C denotes the controlling process of S'
    add $P \parallel C$
    adjust C
    add ϕ_P, the preconditions of P, to Γ
  remove S from $\mathcal{S}'$
if $\mathcal{S}'$ has changed **then**
  remove γ from Γ
  composeWorkflows$(\mathcal{S}', (\mathcal{I}, \mathcal{O}, \Gamma))$
else
  Failure!

gaben von, Dienst P. Also, das beobachtbare Verhalten von diesem Thread ist grob gesehen das inverse Verhalten vom Dienst P.

Falls ein Dienst P zwar manche gewünschten Werte als Ausgaben bereitstellt, aber die von dem Dienst benötigten Eingaben nicht vom Benutzer eingegeben werden sollen, d.h. nicht in der Menge $\mathcal{I}$liegen, versucht der Algorithmus atomare Dienste zu finden und so zu aneinander zu reihen, dass die Sequenz als Ausgabe die vom Dienst P benötigten Eingaben bereitstellen kann. Dies ist besonders sinnvoll in den Fällen, wenn ein Benutzer lokal bei ihm vorhandene Information, z.B. in einer Datenbank, automatisch integrieren möchte. Wir gehen davon aus, dass in solchen Fällen es entsprechende atomare Web-Dienste zur Datenbankabfrage vorliegen. Ein weiterer sehr sinnvoller Einsatz von atomaren Web-Diensten ist die Bereitstellung von einfachen Hilfsoperationen, z.B. für die Konvertierung von Datenformaten oder einfache mathematische Berechnungen. Die automatische Verkettung von atomaren Web-Diensten ist in den letzten Jahren intensiv untersucht worden, z.B. in [4, 11] und nicht unser Hauptfokus in dieser Arbeit.

3.3.1 Beispiel

In diesem Abschnitt wird gezeigt, wie mittels der Workflow-Komposition das Beispiel der Modellierung eines Textanalyse- und Textextraktions-Workflows auch automatisch erstellt werden kann. Wie bereits oben beschrieben, ist es

das Ziel semi-automatisch Workflow-Modelle zu erstellen, die den gegebenen Anforderungen genügen. Dadurch wird der manuelle Workflow-Modellierungsaufwand, wie in den vorhergehenden Abschnitten zur Beschreibung dieses Beispiels in Abschnitt 2.2 gezeigt wurde, verkleinert und die Flexibilität vergrößert.

Eine Kompositionsanfrage für die Komposition eines Textanalyse-Werkzeugs (Workflows) spezifiziert (1) gewünschte Eingabe-Informationen $\mathcal{I}$, (2) erwartete Ausgaben $\mathcal{O}$ sowie (3) die Wertebereiche der Ein- und Ausgaben und die Beziehungen zwischen diesen (bezeichnet mit Γ). In unserem Beispiel möchte die Nutzerin Maria einen gegebenen Text analysieren lassen. Dazu sollen Entitäten erkannt werden und die Beziehungen zwischen den extrahierten Entitäten sollen die gewünschten und zu extrahierenden Fakten darstellen. Im Unterschied zu einer Workflow-Beschreibung sind technische Details der Faktenextraktion, z.B. den Aufbau von Indexen, nicht in einer Anfrage erforderlich.

Für jede einzelne Aufgabe gibt es mehrere Dienste, die fast dieselben Eingaben benötigen. Zudem, gibt es logische Abhängigkeiten zwischen den Eingaben von Diensten für unterschiedliche Aufgaben. Ohne Unterstützung durch die Workflow-Komposition muss Maria dieselben Eingaben mehrfach manuell eingeben und die Ausführung von Diensten selbst koordinieren. Die WisNetGrid Komponenten für Komposition und Ausführung von Workflows helfen ihr effizienter und effektiver ihre Aufgabe zu erledigen.

Anstatt die Dienste manuell zu suchen und manuell zu komponieren wird Maria eine Anfrage definieren und an die Kompositionskomponente schicken. Die gewünschten Eingaben der Anfrage könnten z.B. wie folgt sein:

$$\mathcal{I} = \{\mathsf{text}\}$$

Der komponierte Workflow soll die gewünschten Ausgaben $\mathcal{O}$ bereitstellen. Diese könnten z.B. der Tokenstrom, die Menge der Entitäten, Disambiguierungen sowie die Menge der gelernten und relevanten Fakten sein. Sie beschreibt die gewünschten Ausgaben mit

$$\mathcal{O} = \{\mathsf{token}, \mathsf{entitaeten}, \mathsf{disambiguierungen}, \mathsf{fakten}\}.$$

Mit Γ beschreibt Maria die gewünschten Wertebereiche der Ein- und Ausgaben, sowie die gewünschten Beziehungen zwischen diesen. Zum Beispiel wie

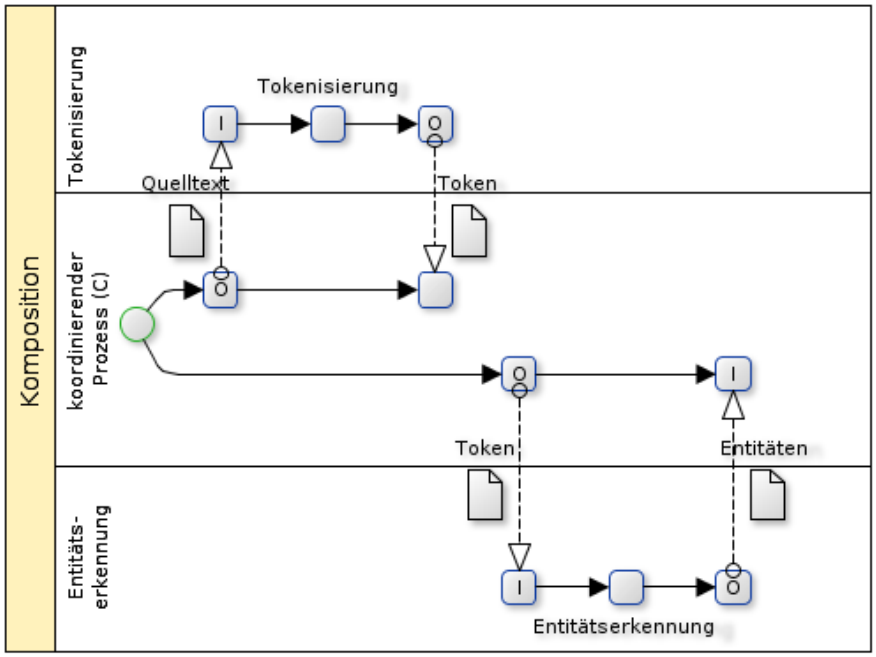


Abbildung 3.3: Ausschnitt eines automatisches erstellten Workflows zur Textanalyse.

folgt,

$$\begin{aligned}\Gamma = \{&\mathsf{ex{:}Quelldatei}(\mathsf{text}), \mathsf{ex{:}Tokenstream}(\mathsf{token}), \mathsf{ex{:}Entitaeten}(\mathsf{entitaeten}),\\ &\mathsf{ex{:}Disambiguierungen}(\mathsf{disambiguierungen}), \mathsf{ex{:}Fakten}(\mathsf{fakten}),\\ &\mathsf{ex{:}enthaeltToken}(\mathsf{text}, \mathsf{token}), \mathsf{ex{:}enthaeltEntitaeten}(\mathsf{token}, \mathsf{entitaeten}),\\ &\mathsf{ex{:}beschreibt}(\mathsf{entitaeten}, \mathsf{fakten}),\\ &\mathsf{ex{:}enthaelt}(\mathsf{fakten}, \mathsf{fakt}), \mathsf{ex{:}Fakt} \sqcap \exists \mathsf{ex{:}relevanz}.\ \geq_{0.95} (\mathsf{fakt})\}\end{aligned}$$

Ein Term in einer solchen Formel kann binär oder unär sein. $\mathsf{ex{:}enthaeltToken}(\mathsf{text}, \mathsf{token})$ besagt, dass der als Eingabe-Parameter bereitgestellte Quelltext einen Strom von Token enthält, welche danach zur Weiterverarbeitung bis hin zur Faktenextraktion zu nutzen sind.

Die Ausgaben von den Diensten können gefiltert und aggregiert werden, wenn ein Wert mit einer Konstante verglichen wird. Der Ausdruck $\geq_{0.95}$ schränkt z.B. die extrahierten Fakten so ein, dass deren Relevanz mit mindestens 95% angegeben sein soll. Die Komposition wird so konstruiert, dass, wie in Abbildung 3.3 gezeigt, ein koordinierender Prozess C die Ausführung von und den Datenfluss zwischen den Komponentendiensten koordiniert. Das

Verhaltensmuster eines Threads des koordinierenden Prozesses C ist das Inverse von dem damit verbundenen Komponentendienst. In der Abbildung wurden 2 Threads für die 2 Komponentendienste (Tokenisierung und Entitätserkennung) erzeugt. Beide Dienste werden sequentiell ausgeführt, da der vom ersten Dienst erzeugte Tokenstrom für den Aufruf des zweiten Dienstes benötigt wird. Die Ausfuehrung und die Kommunikation mit jedem Dienst wird durch verschiedene Threads verwaltet. Dennoch ist das Wissen aller Threads global im Sinne des kontrollierenden Prozesses C, womit die Ausgaben eines Dienstes zur Weiterverarbeitung an einen anderen Dienst weiter gegeben werden können.

Zusätzlich zu der Generierung des inversen Verhaltensmusters stellt der Kompositionsalgorithmus auch sicher, dass der gesamte Workflow die in der Anfrage beschriebenen Anforderungen Γ erfüllt. Zum Beispiel, nachdem die Fakten aus den Entitäten extrahiert wurden, wird eine lokale (bzgl. C) Filteroperation $\geq_{0.95}$ an der Stelle hinzugefügt, so dass nur die Fakten weitergeleitet werden, deren Relevanz genügend groß ist. Der koordinierende Prozess stellt auch sicher, dass die Eingaben, in diesem Beispiel ist dies nur der Quelltext text, die von Diensten verlangt werden, nur einmal vom Nutzer eingegeben werden müssen. Der koordinierende Prozess leitet diese dann an die entsprechenden Dienste weiter.

3.4 Transformation von suprimePDL zur ausführbaren Sprache GworkflowDL/GWES

Eingabe $\mathsf{c}[\mathbf{x}].P$

Eine Eingabeaktivität akzeptiert Eingabeparameter vom Kanal c und bindet diese an die Variablen $\mathbf{x}$ und verhält sich danach wie P. Diese Aktivität wird durch die zwei Stellen (places) p und q und eine Aktivität (transition) *eingabe* in GworkflowDL modelliert (vgl. Abbildung 3.4). Die *parameter* der ersten Flussbeziehung bezeichnen die Eingabeparameter, die nach der *eingabe* an die Variablen $\mathbf{x}$ gebunden wurden.

Der zu verwendende Kommunikationskanal c bestimmte zwischen welchen Akteuren auf welche Art kommuniziert werden kann. Die Sprache GworkflowDL verwendet dazu das 'operation' Element innerhalb der 'transition' Beschreibung. Die zu verwendende Operation ist durch eine Klassifizierung, z.B.

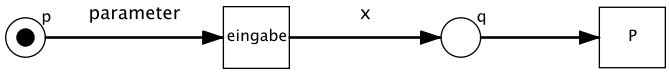


Abbildung 3.4: Eingabeaktivität in GworkflowDL/GWES.

'urn:suprime:EingabeOperation', sowie eventuellen Kandidaten beschrieben. Eine solche Operation implementiert die Funktionalität mittels verschiedenen Technologien und Standards zu kommunizieren.

Listing 3.1: GworkflowDL Eingabe-Transition

```
...
<transition ID="eingabe">
    <inputPlace placeID="p" edgeExpression="parameter" />
    <outputPlace placeID="q" edgeExpression="x" />
    <operation>
        <oc:operationClass xmlns:oc="http://www.gridworkflow.org/gworkflowdl/operationclass"
              name="urn:suprime:EingabeOperation">
            <oc:operationCandidate type="soap" operationName="eingabe" resourceName="http://
                example.org/IO?wsdl" selected="true" />
        </oc:operationClass>
    </operation>
    ...
</transition>
...
```

Ausgabe $\mathsf{c}\langle\mathbf{y}\rangle.P$

Analog zur Eingabeaktivität wird wie in Abbildung 3.5 die Ausgabeaktivität in GworkflowDL modelliert. Dabei werden die Werte von $\mathbf{y}$ über einen Kanal c ausgegeben.

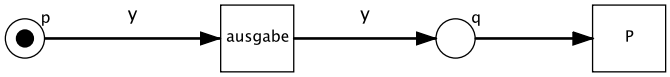


Abbildung 3.5: Ausgabeaktivität in GworkflowDL/GWES.

Lokale Operation $\Delta.P$

Eine lokale Operation berechnet die Änderungen Δ und wird in GworkflowDL durch eine Aktivität L, zwei Stellen p und q sowie zwei Flussbeziehungen modelliert (vgl. Abbildung 3.6).

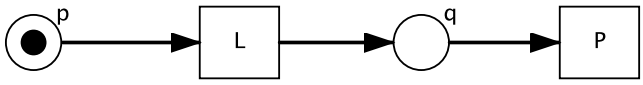


Abbildung 3.6: Lokale Operation in GworkflowDL/GWES.

Wie in Listing 3.1 für die Eingabeaktivität gezeigt wurde, enthält die Beschreibung vom XML Element 'operation' eine Methode zum Aufruf der lokalen Operation L.

Bedingung $\omega?P$

Ein bedingter Prozess verhält sich wie P wenn die Bedingung ω gilt, ansonsten wie $\mathbf{0}$. Eine Bedingung ω wird auf den Parametern *params* in einer leeren Transition wie in Abbildung 3.7 evaluiert. Listing 3.2 zeigt die zugehörige Syntax eines Beispiels, in dem überprüft wird, ob der Wert einer Variablen *name* nicht leer ist.

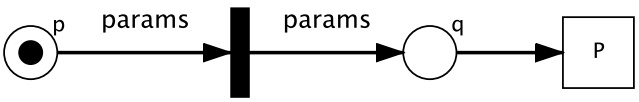


Abbildung 3.7: Bedingung in GworkflowDL/GWES.

Listing 3.2: GworkflowDL Bedingung

```
...
<transition ID="condition_name_provided">
    <description>Name eingegeben</description>
    <inputPlace placeID="p" edgeExpression="params" />
    <outputPlace placeID="q" edgeExpression="*" />
    <condition>string-length($name) &gt; 0</condition>
    <condition>$params/@xsi:type = "xsd:string"</condition>
    ....
</transition>
...
```

Komposition $\prod_{1 \leq i \leq n} P_i$

Bei einer Komposition von n Komponenten P_i werden die Prozesse P_i parallel ausgeführt. Die Abbildung auf GworkflowDL ist in Abbildung 3.8 gezeigt.

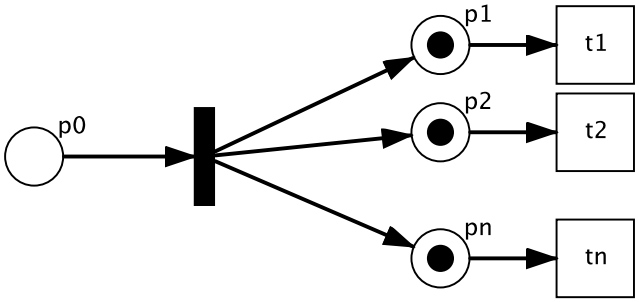


Abbildung 3.8: Komposition in GworkflowDL/GWES.

Auswahl $\sum_{1 \leq i \leq n} P_i$

Die Auswahl verhält sich wie genau einer der n alternativen Prozesse P_i. An der Stelle $p0$ in Abb. 3.9 existiert daher genau ein Token, der dann an eine mögliche Fortsetzung des Prozesses weitergeleitet wird.

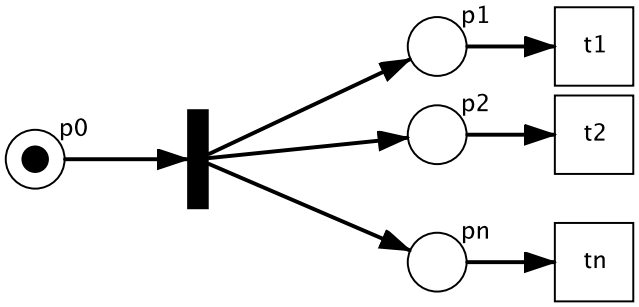


Abbildung 3.9: Auswahl in GworkflowDL/GWES.

Aufruf $@A\{\mathbf{x}\}$

Beim Aufruf eines Agenten A mit den Argumenten $\mathbf{x}$ wird unter Weitergabe der Argumente der Agent durch die entsprechende Transition A ausgerufen (vgl. Abb. 3.10).

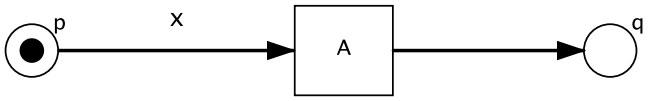


Abbildung 3.10: Aufruf eines Agentens in GworkflowDL/GWES.

Null 0

Null beschreibt eine Aktivität ohne Effekt und kann zur Terminierung benutzt werden. Hierbei werden keine Transitionen ausgeführt. Daher muss die aktuelle Stelle im übersetzten GworkflowDL Modell nicht verändert werden.

Kapitel 4

Zusammenfassung und Ausblick

In diesem Dokument haben wir einen Vergleich der Sprachen und Werkzeugen zur Modellierung und Ausführung von Workflows im Grid präsentiert. Wir haben neben einem Überblick über gängige Workflowsprachen noch mit Hilfe eines Beispiels einer Wissensextraktion, wie sie in dem TextGrid-Projekt Verwendung finden könnte, gezeigt, wie dieses Szenario mit GWorkflowDL und UNICORE 6 modelliert und ausgeführt werden kann. Die manuelle Erstellung von dienstbasierten Workflows kann aufgrund von deren komplexer Struktur und großen Zahl sehr aufwändig werden. Dieses Problem wird in WisNetGrid mit einer semantischen Workflowsprache adressiert. Sie erlaubt aufgrund ihrer formalen Semantik die Entwicklung von automatischen Techniken für z.B. das Suchen von Diensten und Komponieren von Workflows. Wir haben in diesem Dokument neben der semantischen Workflowsprache auch einen Ansatz zu automatischen Komposition von Workflows präsentiert. Die automatisch komponierten semantischen Workflows sind jedoch nicht direkt ausführbar, weil keine Ausführungsengines für solche Workflows existiert. Diese Lücke füllen wir durch eine Transformation von semantischen Workflows in eine Workflowsprache, für die im Grid Ausführungsengines verfügbar sind.

Nachdem jetzt die konzeptionelle, theoretische Grundlage für die automatische Komposition und Transformation von semantischen Workflows in ausführbare Workflows vorliegen, gilt als nächstes die Implementierung der beiden Ansätze und deren Bereitstellung als WisNetGrid-Dienste. Ferner ist die Integration der Dienste mit anderen WisNetGrid-Komponenten geplant, so dass deren Mehrwert im WisNetGrid Portal direkt sichtbar wird.

Literaturverzeichnis

[1] Sudhir Agarwal, Patrick Harms, Rene Jäkel, and Carolin Michels. Wissensnetzwerke im Grid: Semantische Beschreibungssprache für Web-Dienste - Bericht D3.2.2 zu Arbeitspaket 3.2. Technical report, Karlsruhe Institute of Technology (KIT), Juni 2010.

[2] Sudhir Agarwal, Patrick Harms, Carolin Michels, Silke Molch, and Eva Radermacher. D 3.2.1 Anforderungen an die semantische Beschreibungssprache für Web-Dienste, Dezember 2009.

[3] Sudhir Agarwal, Katja Hose, Steffen Metzger, Carolin Michels, and Ralf Schenkel. Wissensnetzwerke im Grid: Architektur des Gesamtsystems für Wissensextraktionsdienste und ihrer Schnittstellen. Technical report, Max-Planck-Institut für Informatik, Juni 2010.

[4] Freddy Lécué. Optimizing QoS-Aware Semantic Web Service Composition. In *ISWC '09: Proceedings of the 8th International Semantic Web Conference*, pages 375–391, Berlin, Heidelberg, 2009. Springer-Verlag.

[5] David A. McAllester and David Rosenblitt. Systematic nonlinear planning. In *AAAI*, pages 634–639, 1991.

[6] Robert Milner. *Communicating and Mobile Systems: the π-Calculus.* The Press Syndicate of the University of Cambridge, 2000.

[7] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, I. *Information and Computation*, 100(1):1 – 40, 1992.

[8] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, II. *Information and Computation*, 100(1):41 – 77, 1992.

[9] Davide Sangiorgi and David Walker. *The pi-calculus: a Theory of Mobile Processes.* Cambridge Universtity Press, 2001.

[10] E. Sirin, B. Parsia, and J. Hendler. Composition-driven Filtering and Selection of Semantic Web Services. In *AAAI Spring Symposium on Semantic Web Services*, 2004.

[11] Shirin Sohrabi and Sheila A. McIlraith. Optimizing Web Service Composition while Enforcing Regulations. In *Proceedings of the 8th International Semantic Web Conference (ISWC09)*, pages 601–617, 2009.