Bachelorarbeit - Informatik

Verifikation von Service-Kompositionen mit Prolog

Philipp Heinisch

Matrikelnummer: 7000281

15.11.2016

Verifikation von Service-Kompositionen mit Prolog

Betreuer: Frau Julia Krämer
Erstgutachter: Frau Prof. Dr. Heike Wehrheim*
Zweitgutachter: Herr Prof. Dr. Gerd Szwillus

Ich möchte mich für die Betreuung und Unterstützung von Julia Krämer und Prof. Dr. Heike Wehrheim bedanken. Ebenfalls möchte ich auch Cedric Richter für seine Arbeit und die Einführung in die Entwicklungsumgebung SeSAME danken, sowie dafür, dass er den SSA-Transformator zur Verfügung gestellt hat.

*AG-Wehrheim: Fachgr. der modellbasierten Softwareentwicklung mit formalen Methoden

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Unterschrift der Verfassers:

Abstract Ziel dieser Arbeit war es, innerhalb von SeSAME, einer Entwicklungsumgebung für Service-Kompositionen, ein Plugin zu erstellen, welches mittels von Java ein Prologprogramm aus einer Service-Komposition erstellt. Dieses Prologprogramm beweist formal auf Grundlage der Hornklausellogik, ob aus jeder validen Eingabe in die Service-Komposition die Nachbedingung dieser gehalten wird und somit als korrekt verifiziert ist. Um eine korrekte Übersetzung in Prolog zu programmieren, habe ich mich zuerst allgemein mit dem heutigen Forschungsstand der formalen Verifikation auseinandergesetzt und eine Übersetzungsroutine entwickelt. Als Ergebnis habe ich auch die Herausforderungen aufgrund von aufkommenden Spezialfällen aufgeführt und Lösungsideen gegeben. Ein weiteres Ziel dieser Arbeit war die Evaluation von Prolog als Verifikationswerkzeug für Service-Kompositionen in SeSAME. Dazu habe ich mit dem SMT-Solver-Ansatz von Walther und Wehrheim [17] verglichen.

Inhaltsverzeichnis

1 Einleitung

Informatik ist die Wissenschaft der systematischen und automatischen Datenverarbeitung. Zur Verarbeitung der Daten werden sich teilweise wiederholende Handlungsroutinen eingesetzt. Diese Routinen werden in der Informatik auch Algorithmen genannt. Man stelle sich beispielsweise eine Bildbearbeitungssoftware vor. Diese kann man je auf ein Bild anwenden und verschiedene Algorithmen (Berechnungen) sorgen dafür, dass nach Anwendung des Bildbearbeitungsprogrammes ein wenig Speicher verbrauchendes mit einem Filter verbessertes Bild entsteht.
Nun wird Computersoftware immer komplexer. In einer fortschreitend digitalisierten und vernetzten Welt müssen immer komplexere Probleme mit verschiedensten Algorithmen gelöst werden [11]. So kauft man beispielsweise heute mit einem modernen Automobil 60 bis 70 Steuerelemente (Diebstahlwarnanlage, Navigationsgerät, Motorsteuerung) [11]. Immer größere Arbeitsgruppen, in denen Einzelne keinen allumfassenden detaillierten Überblick über das ganze Produkt mehr haben, prägen die moderne IT-Welt [11]. Überall befindet sich Software, sei es im Haushalt, in der Medizin, Mobilbereich, Wasserkraftsystemen oder in Atomkraftwerken [11]. Bei all dem ist Softwarequalität wichtig und eine Eigenschaft der Qualität ist die Wiederverwendbarkeit [11]. Besonders bei den immer komplexer werdenden Systemen werden Grundalgorithmen wiederholend benötigt und Entwicklungskosten wären sehr viel höher, wenn Programmierer ständig alles von Grund auf implementieren würden. Deshalb entstand in der Forschung das viel diskutierte Konzept [4] der serviceorientierten Architekturen, abgekürzt SOA [18]. Diese Innovation basiert auf Services, gekapselten Programmen, deren Schnittstellen bekannt sind. Unter SOA verstehen wir nun das Konzept des Entwerfens von Applikationen auf Basis von interoperablen Services [12]. So ist es leichter, Software wieder zu verwenden und große Softwaresysteme (teils automatisch [14]) zu konstruieren. Die Verkettung existierender Services bietet neue Wege, die komplexen Probleme der heutigen Zeit zu lösen. Eine solche Verkettung nennen wir Service-Komposition [17].
Beispielsweise können wir die Bildbearbeitungssoftware schnell mit einer Service-Komposition lösen. Services betrachten wir als Blackbox. Bekannt ist jedoch die Signatur der Eingänge und Ausgänge sowie eine Vorbedingung und eine Nachbedingung. Die Nachbedingung (betrifft den Ein- und Ausgang) gilt dann, wenn die Vorbedingung (betrifft den Eingang) erfüllt ist [17]. Wir nehmen nun an, dass zwei Services existieren- jeweils für die zwei Aktionen, die das Programm vollzieht: die eine skaliert das Bild herunter, wenn es zu groß ist. Dafür wird gesorgt, dass das Bild nur noch wenig Speicherplatz verbraucht. Dann steht noch die Aufgabe des Verbesserns bevor: ein Filter soll angewandt werden. Dazu existiert ein weiterer Service. Beide in Kombination erreichen, dass nach Anwendung des Programmes das Bild sowohl speichergünstig als auch sehenswert ist. Der Quelltext der Services muss nicht veröffentlicht werden, nur die Schnittstelle.

Nun wäre es aber ärgerlich, wenn die Software zur Bildbearbeitung nicht ordnungsgemäß wäre, wenn diese also ewig ohne Ergebnis laufen würde (nicht terminiert) oder speicherintensive und/oder verfälschte Bilder ausgegeben werden. Somit wäre die Anforderung, die man von der Software erwartet, nicht erfüllt und es liegt ein Fehler vor. Besonders in einer digitalisierten Welt ist die Gewährleistung eines sicheren und korrekten Ablaufes notwendig. Sicherheitskritische Bereiche, die immer mehr von Computerprogrammen durchdrungen werden, müssen ordnungsgemäß bedient werden [11]! Denn verschiedene Beispiele zeigen, dass Software, deren Anforderungen nicht sicher gestellt wurden, fatale Folgen haben kann. Neben ärgerlichen Programmabstürzen gab es beispielsweise schon mehrere Rückrufaktionen wegen eines Softwarefehlers in der Automobilbranche, im Konkreten 2015 bei Fiat Chrysler wegen einer Sicherheitslücke. Bei modernen Oberklassewagen ist mittlerweile ein Versagen der Bordelektronik der häufigste Pannengrund [11]. Auch Tote hat es durch fehlerhafte Software bereits gegeben: im Jahr 2000 verrechnete sich ein Bestrahlungsgerät systematisch in der Dosierung[1].
Auch Service-Kompositionen müssen valide sein, um beispielsweise genau diese eben benannten Fehler zu vermeiden. Während in der Praxis SOA an einigen Stellen mit teilweise unbefriedigendem Ergebnis angewandt wird, hat SOA ein großes Potential mit positivem Trend. Dabei drängt sich auch die Frage der Sicherheit und Korrektheit noch stärker auf [4].

Um die Korrektheit und Sicherheit von Software sicher zu stellen, haben sich in der Informatik die Softwaretests, die statische Codeanalyse und die Softwareverifikation durchgesetzt [11]. Auch, wenn es zahlreiche Ansätze im Bereich der Softwaretests und statischen Codeanalyse gibt, so stoßen diese Methoden schnell an ihre Grenzen. So zeigen Tests nur das Vorhandensein von Fehlern, nicht aber deren Abwesenheit. Programme werden sehr schnell zu komplex für Tests, die alle Fälle abdecken [11]. Erschwerend kommt im Bereich des Testens und der statischen Codeanalyse von Service-Kompositionen hinzu, dass in den meisten Fällen nur die Schnittstelle der wichtigen Komponenten (Services) bekannt ist, nicht jedoch der genaue Algorithmus oder der Code. Man kann nicht von einer Beispieldatenbank als Testparameter ausgehen. Ziel der Arbeit ist es also, nun auch in dem neueren Gebiet der SOA eine zuverlässige Möglichkeit der Verifikation zu haben. In dieser Arbeit werde ich für die Erfüllung des Zieles ein Tool vorstellen, welches formal verifiziert. Dabei nehme ich an, dass die einzelnen verwendeten Services hinsichtlich ihrer Schnittstellenbeschreibung valide sind. Diese Annahme wird häufig im Bereich der Service-Kompositionen getroffen wie beispielsweise in [17]. Auch, wenn diese Voraussetzung theoretisch den Anwendungsbereich des in dieser Arbeit vorgestellten Ansatzes einschränkt, so stellen in der Praxis Software-Verifizierungs- und Zertifizierungsverfahren eine fundierte Aussage zu der Korrektheit von Service-Kompositionen mit meinem Ansatz sicher.

Die Verifikation von Service-Kompositionen ist in der Forschung ein wichtiges The-

[1] `https://jaxenter.de/top-10-der-software-katastrophen-181` (02.11.2016)

ma. So ist es nicht verwunderlich, dass bereits einige Ansätze existieren. Von Walther und Wehrheim wurde beispielsweise ein Verifizierer entwickelt, der mit vorverifizierten Templates arbeitet. Dieser kann jedoch nur eine Teilmenge aller Service-Kompositionen verifizieren [18]. Ebenfalls haben die beiden Foscher ein Tool zur Service-Komposition-Verifikation unter Benutzung des SMT-Solvers Z3 von Microsoft entwickelt [17]. Nun möchte ich die Sammlung der Ansätze mit meiner Implementierung, die ich in dieser Arbeit vorstelle, sinnvoll ergänzen. Denn bisher wurden noch keine Verifizierungsansätze für Service-Kompositionen entwickelt, die auf Horn-Klauseln[2] basieren. Über Horn-Klausel gibt es bereits viele wissenschaftliche Arbeiten, und so wurden in den letzten Jahren vermehrt Modelle direkt in Horn-Klauseln übersetzt [5]. Dies liegt mitunter auch daran, dass allgemein die Prädikatenlogik erster Ordnung gut geeignet ist, um Wissen zu modellieren [16]- Ansätze mit Horn-Klauseln sollten insofern gefördert werden [5]. Algorithmen, die Horn-Klauseln lösen, gibt es viele, beispielsweise Duality, HSF, SeaHorn und μZ [5]. Für C-Programme wurden bereits erfolgreich effiziente Verifizierer entwickelt wie beispielsweise HSF(C) [8]. Ziel ist es, einen auf Hornklauseln basierenden Ansatz zur Verifikation von Service-Kompositionen zu entwickeln.
Um diesen Ansatz umzusetzen, werde ich das Tool Prologvalidator erstellen, dass die Service-Komposition mit dem Wissen der Domäne und ihren Spezifikationen als Eingabe erhält und prüft, ob die Nachbedingung bei allen der Vorbedingung entsprechenden Eingabe erfüllt wird. Dazu wird die Service-Komposition in ein Prologprogramm übersetzt werden. Dieses erhält eine zur Lösung der Verifikationsaufgabe passende Anfrage. Daraufhin wird Prolog antworten- diese Antwort wird nach einer Interpretation als Lösung der Verifikationsaufgabe ausgegeben. Prolog ist eine etablierte logische Programmiersprache, die auf der Hornklauselllogik basiert [16]. Diese Sprache kann und wird aufgrund ihrer Entwicklung und Forschungsbekanntheit flexibel genutzt. Daraus können wir schlussfolgern, dass auch in Zukunft Prolog genutzt wird und somit auch in Zukunft gut wartbar sein wird. Ich untersuche mit dieser Arbeit, ob für Service-Kompositionen die Umwandlung in ein Prologprogramm empfehlenswert ist. Gezeigt wurde bereits, dass sich imperative Programme gut in ein Prologprogramm umwandeln lassen [5]. Service-Kompositionen sind imperative Programme mit Zusatz einer Vor- und Nachbedingung. Dieser Zusatz ist ebenfalls gut vereinbar mit der Umwandlung in ein Prologprogramm, da Vor- und Nachbedingungen logisch formuliert sind und Prolog eine logische Programmiersprache ist [16]. Selbiges gilt auch für die einzelnen aufgerufenen Services, die wir ebenfalls nicht im Detail kennen, während jedoch auch hier die Vor- und Nachbedingung bekannt sind. Auch nur diese Information wird später relevant sein, sodass wir einen Aufruf eines Services imperativ formulieren können: wenn die Vorbedingung erfüllt ist, gilt die Nachbedingung. Dies kommt der imperativen Kontrollstruktur der Verzweigung nahe.
Wie der Z3-Verifikationsansatz von Walther und Wehrheim wird auch mein Tool in die Entwicklungsumgebung SeSAME eingebunden sein [17]. SeSAME bedeutet „Service Specification, Analysis, and Matching Environment“ und ist eine Umgebung zur Modellierung und Verifizierung von Service-Kompositionen. Die Umgebung stellt eine high-

[2]Eine Definition und Erklärung zu Horn-Klauseln ist in Kapitel 2.2.1 zu finden

level Service-Kompositions-Spezifikationssprache zur Verfügung. Daneben bietet diese einen Pool existierender Services inklusive einer Filter- und Suchfunktion an, mit der man leicht eine Service-Komposition erstellen kann [2]. Ziel ist es, diese Entwicklungsumgebung innerhalb der Sicherheitskomponete zu ergänzen mit meinem effizienten Ansatz, die erstellten Services mit Prolog zu verifizieren. Somit wird ein Anreiz gesetzt, in der Praxis vermehrt Service-Kompositionen zu fördern und somit dem Komplexitätsproblem heutiger Software eine Lösung aufzuzeigen.

Struktur dieser Arbeit In dieser Arbeit möchte ich zunächst in das Thema einführen und Hintergründe klären (Kapitel 2). Dabei möchte ich Begriffe und Sprachen definieren, die ich im späteren Verlauf verwenden werde. Zentral ist die Einführung in Service-Kompositionen (Kapitel 2.1) und der Programmiersprache Prolog (Kapitel 2.2). Bevor ich dann in Kapitel 4 zur theoretischen Übersetzung von Service-Kompositionen zur Verifizierung in Prolog komme, sorge ich in Kapitel 3 für ein allgemeines Verständnis für Verifikation. Um eine Grundbasis für die Übersetzung in Prolog schaffen, werfe ich einen Blick in den aktuellen Forschungsstand um das Thema. Des Weiteren gehe ich noch gesonderte auf Schleifen ein (Kapitel 3.4), die auch in Service-Kompositionen auftreten können. In Kapitel 5 gehe ich auf die Übersetzung im praktischen Gebrauch mit dem zu dieser Arbeit programmierten Tool ein. In Kapitel 5.3 wird ein konkretes Beispiel übersetzt und verifiziert werden, was uns zur Frage führt, wie dieses Tool im Vergleich zum SMT-Ansatz abschneidet, den wir näher in Kapitel 3.2 kennen lernen werden. Diese Evaluation erfolgt im 6. Kapitel. Im Abschluss der Arbeit fasse ich zusammen, ziehe ein Fazit und gebe Ideen für eine zukünftige Weiterarbeit (Kapitel 7).

2 Einführung und Hintergründe

In diesem Kapitel lege ich die Grundlagen für die kommenden Kapitel. Ich werde auf die Grundidee und die Bestandteile von Service-Kompositionen und von der logischen Programmiersprache Prolog eingehen. Dazu werde ich Begriffe erklären und definieren. Um leichter zu verstehen, was mit welchem Punkt gemeint ist, werde ich das Beispiel aus der Einleitung aufgreifen und für diese und die kommenden Kapitel zur Illustration verwenden. Dieses Beispiel möchte ich nun im Detail vorstellen:
Stellen wir uns vor, es gibt ein Unternehmen, welches sich zur Aufgabe gemacht hat, gute Bildbearbeitungssoftware zu programmieren und diese Produkte dann auszuliefern. Das neueste Bildbearbeitungsprogramm dieses Unternehmens bietet Speicherminimierung und Bildverbesserung in einem Klick an. Das Unternehmen möchte nun seine Kunden davon überzeugen, dass dieses Programm zuverlässig läuft und garantiert beschwerdefrei bleiben wird. Um diesem Versprechen gerecht zu werden, engagiert es nun ein weiteres Unternehmen, welches formal verifizieren soll, dass das Bildbearbeitungsprogramm hält, was es verspricht.

▷ **Definition:** Eine formale Verifikation ist ein Verfahren, welches die korrekte Funktion eines Systems mathematisch nachweist. Die angewandten mathematischen Methoden schließen auch das Gebiet der Aussagen- und Prädikatenlogik mit ein.

Somit ist klar, dass diese Aufgabe nicht mit Ausprobieren und Testen zu bestehen ist, denn es gibt unendliche viele Bilder.

Prädikatenlogik Da wir später auf auf Prädikatenlogik zu sprechen kommen, hier vorab die Definition dazu.

Definition: Sei P ein beliebiger prädikatenlogischer Ausdruck erster Stufe mit reduziertem Umfang, so definieren wir diesen syntaktisch nach folgender Grammatik in Backus-Naur-Form:
$P \ni \varphi_1, \varphi_2, ..., \varphi_n ::= true \mid false \mid c \mid x \mid f(\varphi_1, \varphi_2, ..., \varphi_n) \mid \neg(\varphi_1) \mid (\varphi_1 \wedge \varphi_2) \mid (\varphi_1 \vee \varphi_2) \mid (\varphi_1 \rightarrow \varphi_2) \mid (\varphi_2 = \varphi_2)$, wobei c eine Konstante, x eine Variable und f eine Funktion oder Prädikat mit n Parametern ist. [9]
Wir werden später eine Programmiersprache kennen lernen, die prädikatenlogische Formeln integriert. Dort weicht die Schreibweise wie folgt ab:

- $\varphi_1 \wedge \varphi_2$ wird geschrieben als φ_1 `and` φ_2
- $\varphi_1 \vee \varphi_2$ wird geschrieben als φ_1 `or` φ_2
- $\neg(\varphi_1)$ wird geschrieben als `neg` φ_1

↪ *Bemerkung:* Die Prädikatenlogik erster Stufe mit vollem Umfang erweitert die Menge an gültigen Formeln um den Gebrauch der Quantoren (All- und Existenzquantor), weiteren arithmetischen Operatoren wie + und / und der Äquivalenz. [9]

SSA-Form Bevor ich nun in die Service-Kompositionen einführe, gibt es noch einen davon unabhängigen Begriff, den ich später verwenden werde.

Definition: Wir ordnen genau dann einem Code die Eigenschaft der SSA-Form (single static assignment) zu, wenn sicher gestellt ist, dass jeder Variable des Codes nur einmal ein Wert zugewiesen wird

2.1 Einführung in Service-Kompositionen

Domäne und Ontologien Das beauftragte Unternehmen möchte nun diese formale Verifikation absolvieren. Doch bevor man sich mathematische Methoden überlegt, ist es wichtig, sich Wissen der Domäne anzueignen. Was für ein System mit welchen Bestandteilen liegt vor? Nehmen wir uns das Beispiel vor, so liegt uns die Bildbearbeitungssoftware vor, die (der Veranschaulichung halber) nur eine Funktion kennt: zu große Bilder herunter skalieren und einen Filter auf das eingegebene Bild anwenden. Wir betrachten also Bilder, die durch die Bildbearbeitungssoftware modifiziert werden- erwarten diese Modifizierung- auf jeden Fall ist sicher, dass am Ende des Programmes wieder ein Bild heraus kommt. Spezifiziert man nun das Wissen, fasst alle relevanten Komponenten auf, gelangt man zur Ontologie.

Konzepte	`Boolean` (true oder false), `Image` (repräsentiert ein Bild)
Rollen	`isVariantOf(original:Image, modified:Image) -> Boolean, filterApplied(image:Image) -> Boolean, lessThan5MB(image:Image) -> Boolean`

Tabelle 2.1: Ontologie in Textform (Tabelle) zum Beispiel

▷ **Definition:** Eine Ontologie bildet formalisiertes Wissen zu einer spezifischen Domäne ab. Dieses Wissen teilt sich in Konzepten (auch Klassen oder Typen genannt) und Rollen (Relationen zwischen Konzepten bzw. Eigenschaften eines Konzeptes) auf. Eine Ontologie hat somit mindestens ein Konzept und beliebig viele Rollen. [17]

Um eine Ontologie anhand der Programmiersprache Java zu verstehen: eine Klasse in Java kann man mit einem Konzept vergleichen. Klassen in Java bildet man mit dem Interesse, dass man verschiedene Instanzen haben kann, die in ihrer Struktur (Funktionen, Attribute, Zweck) gleich sind, so, dass man gegebenfalls die Funktionalität für mehrere Objekte nicht doppelt programmieren muss. So ist auch die Bildung von Konzepten zu verstehen [17]. Zugleich haben aber genau diese Klassen und Instanzen Funktionen, Eigenschaften. Ruft man in Java auf einer Instanz der Klasse String die Funktion `toLowerCase()` auf, so erhält man wiederum eine String-Instanz zurück. In dem Falle steht also die Klasse String in Relation zu sich selbst. Ähnlich ist es mit den Rollen (Relationen) auch bei der Ontologie der Service-Komposition zu verstehen. Ontologien kann man grafisch gestalten, aber auch in Textform, auch als prädikatenlogische Formel- formalisieren tun wir die Ontologie unter Benutzung der „Web Ontology Language“(OWL) [17].
Im Falle des Beispiels findet das beauftragte Unternehmen Konzepte und Rollen wie in Tabelle 2.1 aufgelistet.

↪ *Bemerkung:* Neben `Boolean` gibt es noch weitere Konzepte, die sich auf primitive Datentypen beziehen. So gibt es beispielsweise noch das Konzept `Int` (Integer), das Konzept von ganzzahligen Zahlen. Solche primitiven Konzepte bedürfen im späteren Verlauf einer Sonderbehandlung. In der Ontologie fallen diese dadurch auf, dass diese selbst keine Rollen beinhalten, sondern nur durch nicht primitive Konzepte in Relation zu anderen Konzepten stehen.

↪ *Bemerkung:* Ontologien können ebenfalls Regeln beinhalten. Dies sind Behauptungen, die bindend sind für das, was mit dieser Ontologie erstellt wird. Ebenfalls können Rollen mit Eigenschaften belegt werden. Da Rollen in der Prädikatenlogik auch als Funktionen beziehungsweise Prädikate (wenn sie auf ein `Boolean` abbilden) angesehen werden können, kann man diese Eigenschaften auch als Relationseigenschaften betiteln. Folgende Eigenschaften sind aktivierbar: Reflexivität, Irreflexivität, Symmetrie, Antisymmetrie, Asymmetrie, Transitivität, Funktionalität (auch unter 'alternativ' bekannt) [17].

Für meinen Ansatz lasse eine Auswahl von den Eigenschaften und somit ein Teilbereich der Ontologieregeln zu unter der Annahme, dass Eigenschaften nur für einstellige Funktionen oder zweistellige Prädikate aktiviert werden. Dabei wird bei einer Funktion der Eingabeparameter mit der Ausgabe in Relation gesetzt und bei einem Prädikat die beiden Eingabeparameter. Das Prädikat $isVariantOf$ ist beispielsweise symmetrisch, somit gilt $isVariantOf(y, x) = true$, wenn $isVariantOf(x, y) = true$ ist.

Services Das beauftragte Unternehmen kennt nun die Domäne und hat einen Überblick über vorkommende Konzepte und Rollen. Dies ist Basis für das für den Sachverhalt, den wir uns nun anschauen werden- der weder als Rolle noch als Konzept namentlich in der Ontologie zu finden sein wird [17]:

▷ **Definition:** Ein Service ist ein Programm, in dem die Funktionalität gekapselt und deren Schnittstelle publiziert ist [12]. Es liegt minimal folgendes Wissen über den Service vor [17]:

- Servicesignatur
  - Name des Services
  - Anzahl und Konzeptbestimmung (auf Grundlage der Ontologie) der Eingänge
  - Anzahl und Konzeptbestimmung (auf Grundlage der Ontologie) der Ausgänge
- Bedingungen des Services
  - Vorbedingung in einer prädikatenlogischen Formel erster Stufe
  - Nachbedingung in einer prädikatenlogischen Formel erster Stufe

↪ *Bemerkung:* Das Wort Vorbedingung in diesem Kontext ist in der Literatur auch als Prekondition bekannt, das Wort Nachbedingung auch als Postkondition oder Effekt. [17]

▷ **Definition:** Es gilt, dass die Nachbedingung eines Services nur anzunehmen sei, wenn die Vorbedingung sicher gestellt werden kann. Ist die Vorbedingung nicht erfüllt, kann keine Aussage über den Ausgang des Services gemacht werden. [13]

↪ *Bemerkung:* Services können im Allgemeinen auch ihre die Werte ihrer Eingangsvariablen modifizieren. In dieser Arbeit nehmen wir jedoch an, dass alle genutzten Services dies nicht tun. Wir nehmen also an, dass, wenn vor Aufruf des Services die Vorbedingung gilt, so gilt diese auch nach Ausführung der Services. Services können so als Implikation aufgenommen werden: `Vorbedingung` → `Nachbedingung` [13]

↪ *Bemerkung:* Wir nehmen an, dass Services hinsichtlich ihrer Signatur und Be-

dingung valide sind. Wir können jedoch nicht davon ausgehen, dass die Nachbedingung den Ausgang im Detail beschreibt. So wäre als Nachbedingung `true` legitim, nur leider bietet dies dem Benutzer keine Mehrinformation über das, was der Services tut und was dieser zurück gibt. Als Serviceanbieter bietet es sich jedoch an, eine möglichst klare Nachbedingung zu publizieren.
Schlussendlich sei angemerkt: Um Services zur Verfügung zu stellen, reicht es, die Schnittstelle mit der Ausführungsdatei zu publizieren. Es ist nicht notwendig, den Quellcode zur Verfügung zu stellen noch Algorithmen oder Abläufe des Services zu veröffentlichen. So können Services als Blackbox-Funktionen angesehen werden [17].
Services betrachten wir als Codestücke, die uns in der Regel nicht als direkter Quelltext offen liegen, sondern nur deren Schnittstelle bekannt ist. Diese entscheidenden Codestücke, die zum verifizieren notwendig sind, machen eine Service-Komposition zu dem, was es ist: eine Komposition von Services.
Im vorliegenden Beispiel gefällt dem Auftraggeber diese Freiheit sehr. Dieser stellt dem beauftragten Unternehmen das Programm zum Verbessern des Bildes mit Hilfe eines Filters als Service zur Verfügung. Folgendes Wissen zu diesem Service sei gegeben:

- Servicesignatur
  - Name: `applyFilter`
  - Eingänge: `in1: Image` (Variable `in1` vom Typ `Image`)
  - Ausgänge: `out1: Image`
- Bedingungen des Services
  - Vorbedingung: `lessThan5MB(in1)` (prädikatenlogische Formel zu der Aussage: „Das Bild ist kleiner als 5MB“. Dieses einschränkende Prädikat macht Sinn, weil ansonsten der Filterprozess zu lange laufen würde, so begründet das Softwareunternehmen diese Entscheidung)
  - Nachbedingung: `isVariantOf(out1, in1) and filterApplied(out1) and lessThan5MB(out1)` (das Bild wurde nicht verfälscht, besitzt nun den aufbessernden Filter und ist kleiner als 5MB groß)

Der Service bildet somit auf die Implikation $lessThan5MB(in1) \rightarrow isVariantOf(out1, in1) \wedge filterApplied(out1) \wedge lessThan5MB(out1)$ ab.
Des Weiteren bekommt das beauftragte Unternehmen den zweiten Service (`scaleDown(in1: Image) -> out1: Image`), der, geben ein großes Bild, dieses auf ein Maß herunter skaliert, so, dass das Bild nur noch weniger als 5MB Speicherplatz verbraucht. Diese Skalierung verändert jedoch nicht das, was das Bild darstellt.

Service-Kompositionen Diese Services getrennt voneinander helfen jedoch nur bedingt, wenn das beauftragte Unternehmen das Gesamtprodukt verifizieren soll. Abhilfe schafft hierbei das bestimmte kombinieren dieser beiden Services. Mit diesem Ansatz sind wir bei dem Begriff der Service-Komposition.

▷ **Definition:** Eine Service-Komposition ist ein imperatives Programm, welches wiederum Services aufruft und auf einer gezielten Verkettung von diesen basiert [17]. Unter Service-orientierter Architektur, abgekürzt SOA, versteht man das Konzept des Entwerfens von Applikationen auf Basis von interoperablen Services. [12]

↪ *Bemerkung:* Eine Service-Komposition von außen betrachtet ist wiederum ein Service mit Signatur, Vor- und Nachbedingung. Eine Service-Komposition kann so in einer anderen Service-Komposition, welche gegebenfalls ein noch komplexeres Problem löst, wiederverwendet werden. [17]
Für das Erstellen von Service-Kompositionen kann eine Entwicklungsumgebung helfen. Es gibt eine speziell für Service-Kompositionen entwickelte Entwicklungsumgebung names SeSAME [2]. SeSAME („Service Specification, Analysis, and Matching Environment“) ist eine Eclpise-basierte Tool-Suite, die beim Modellieren von Service-Kompositionen hilft und dabei eine umfassende Service-Spezifkations-Sprache anbietet. Mit der Sprache von SeSAME zum Modellieren einer Service-Komposition und deren Kontrollfluss [2] gehe ich auch in dieser Arbeit um. Diese Sprache ist syntaktisch nach Krämer und Wehrheim [13] wie folgt definiert:

▷ **Definition:** $\mathsf{D} = (\mathrm{T}_D, \mathrm{F}_D)$ sei unser Domänewissen, wobei T_D die in der Ontologie definierten Konzepte und F_D die in der Ontologie definierten Rollen sind, bestehend aus Funktionen und Prädikaten. Sei SC eine Service-Komposition, so ist diese nach folgender Grammatik in Backus-Naur-Form beschrieben:
SC $\ni S_1, S_2 ::= Skip \mid S_1; S_2 \mid Tu := S(v_1, v_2, ..., v_n) \mid \mathsf{while} B \mathsf{do} S_1 \mathsf{od} \mid \mathsf{if} B \mathsf{then} S_1 \mathsf{else} S_2 \mathsf{fi}$, wobei $T \in \mathrm{T}_D$, B ein prädikatenlogischer Ausdruck erster Stufe, der nach `true` oder `false` ausgewertet werden kann und S n Eingänge und genau eine Ausgangsvarible besitzt. Elemente aus F_D können in prädikatenlogischen Formeln benutzt oder bei S direkt aufgerufen werden.

Die Semantik ist vergleichbar mit der vieler Programmiersprachen wie Java, so steht `Skip` für eine leere Anweisung, `while` für eine Schleife und `if` für eine Verzweigung des Kontrollflusses [13]. Wir nehmen an, dass Schleifen immer mit einer validen Invariante einprogrammiert werden.
So programmiert das Team des beauftragten Unternehmens die Service-Komposition für die Verifizierung der Bildbearbeitungssoftware (Zeilen, die mit `//` anfangen, gelten als Kommentar, Kontrollfluss grafisch in Abbildung 2.1):

Kontrollfluss der Service-Kompositon zur Bildverarbeitungssoftware

```
1 if neg lessThan5MB(input) then
2   // es liegt ein grosses Bild vor
3   Image output := scaleDown(input);
4   output := applyFilter(output);
5 else
6   // es liegt ein kleines Bild vor
```

```
7   Image output := applyFilter(input);
8 fi
```

↪ *Bemerkung:* So wie ein Service hat auch die Service-Komposition eine Vor- und Nachbedingung, formuliert in Prädikatenlogik erster Stufe. Eine Service-Komposition ist dann valide, wenn für jede Eingabe, die die Vorbedingung erfüllt, auch die Nachbedingung erfüllt ist. Es gilt: (Service Komposition ist valide) ↔ (Vorbedingung → Nachbedingung) [18]

↪ *Bemerkung:* Die Vor- und Nachbedingung einer Service-Komposition wird zusammen in der Literatur auch als die „Anforderungen an die Service-Komposition“ bezeichnet [17]
Für unser Beispiel legen wir, um die Aufgabe zu erfüllen, folgende Vor- und Nachbedingung fest (Definition 2):

- Vorbedingung (*Assume*): `true` (alle Eingaben des Konzeptes `Image` sind erlaubt- also alle Bilder- die Vorbedingung ist immer wahr)
- Nachbedingung (*Assert*): `isVariantOf(i, o) and filterApplied(o) and lessThan5MB(o)` (das ausgebende Bild ist ähnlich dem eingegeben, jedoch mit Filter und kleiner als 5MB)

Eine Grafik über den Kontrollfluss unseres Beispieles ist in Abbildung 2.1 zu finden. Hier wird anschaulich deutlich, wie die Ontologie, Services, Kontrollfluss und die Vor- und Nachbedingung der Service-Komposition zusammen wirken.

2.2 Einführung in Prolog

In dieser Arbeit stelle ich einen Ansatz zur Verifikation von Service-Kompositionen mit Prolog vor. Da also letztlich ein Prologprogramm über die Richtigkeit einer Service-Komposition entscheiden wird, ist es wichtig, die Philosophie und Bestandteile dieser

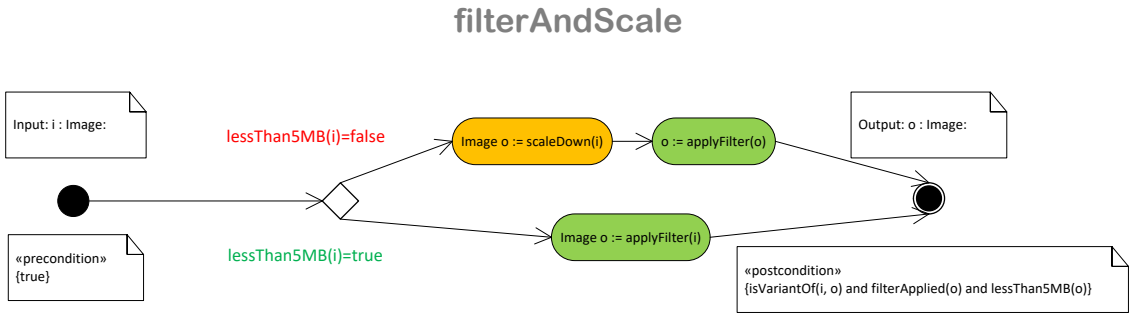


Abbildung 2.1: Kontrollfluss der Beispiel-Service-Komposition 2.1 grafisch

Programmiersprache überblicken zu können.
Für die Aussagen in diesem Unterkapitel 2.2 verweise ich als Quelle auf das Buch vom Stefan Schmitgen „Prolog: Grundlagen und Anwendungen“ [16]

2.2.1 Grundidee von Prolog

Prolog (PROgramming in LOGic) ist eine logische Programmiersprache und wurde 1972 von Philippe Roussel in der Universität von Marseille entwickelt. Prolog unterscheidet sich deutlich von objektorientierten Programmiersprachen wie Java. Als erstes besonderes Merkmal von Prolog sei die Hornklauselllogik genannt, auf der diese Sprache basiert.

▷ **Definition:** Eine Horn-Formel ist eine logische Formel mit der Einschränkung, dass diese folgende Bedingung erfüllen muss:

- die Formel ist in konjunktiver Normalform (die Formel besteht also in der obersten Ebene nur aus Konjunktionen (UND-Verknüpfungen), man spricht auch von einer Konjunktion von Disjunktionstermen)
- in jedem Disjunktionsterm (auf 2. Ebene) tritt höchstens ein Literal nicht negiert auf. Diese Disjunktionsterme nennen wir Hornklauseln.

Beispiel:

- $(a \wedge b) \vee b$ ist keine Horn-Formel, da dieser Term nicht in konjunktiver Normalform ist. Hier befindet sich eine Disjunktion auf der ersten Ebene, eine Konjunktion $(a \wedge b)$ auf zweiter Ebene
- $(a \vee b \vee \neg c) \wedge ... \wedge (...)$ ist ebenfalls keine Horn-Formel, da es auf zweiter Ebene eine Disjunktion gibt, die mehr als ein positives Literal besitzt (a und b)
- $(\neg a \vee b \vee \neg c) \wedge (a)$ ist eine Horn-Formel

Teilt man eine Horn-Formel auf oberster Ebene auf, so erhält man ein Set von Hornklauseln, die für die Erfüllung der gesamten Formel wahr sein müssen.

▷ Satz: $L \vee \neg L_1 \vee \neg L_2 \vee ... \vee \neg L_n$ ist logisch äquivalent zu $L_1 \wedge L_2 \wedge ... \wedge L_n \rightarrow L$. Wir schreiben dies als $L \leftarrow L_1 \wedge L_2 \wedge ... \wedge L_n$

Beweis: $L \vee \neg L_1 \vee \neg L_2 \vee ... \vee \neg L_n \Leftrightarrow \neg(L_1 \wedge L_2 \wedge ...L_n) \vee L \Leftrightarrow L_1 \wedge L_2 \wedge ...L_n \rightarrow L$
Es ergeben sich genau 3 Arten, die wir später im Prologprogramm wie folgt benennen werden:

- **Regel**: Es gibt genau ein positives Literal und mindestens ein negiertes Literal. Das positive Literal bezeichnen wir als Konklusion und die negativen Literale als Prämisse
- **Fakt**: Es gibt genau ein positives Literal, aber kein negiertes Literal

- **Anfrage**: Es gibt kein positives Literal

Ein zweites besonderes Merkmal ist die speziell immer gleiche Weise, wie ein Ergebnis auf eine Anfrage berechnet wird. Dieses besteht entweder aus der Antwort false/true oder einem Set von mit Werten belegten Variablen.
Diese Berechnung geschieht durch die SLD-Resolution mit Hilfe der Unifizierung und durch Anwendung eines Backtracking-Algorithmus zustande. Um dies im Überblick zu erklären, möchte ich kurz auf die Unifizierung eingehen: Wenn eine Anfrage gestellt wird, versucht Prolog, den Term der Anfrage mit Hilfe der programmierten Regeln auf Grundlage der definierten Faktenbasis zu vereinheitlichen. Konkret bedeutet dies für Variablen, dass diese an Ausdrücke gebunden - also unifiziert werden. Dies ist nur im Ansatz mit einer Zuweisung aus Programmiersprachen wie Java zu vergleichen, da es prinzipiell für Prolog nicht wichtig ist, an welcher Stelle des Programms die Variable einen Wert 'zugewiesen' bekommt. Nun probiert Prolog verschiedene Variablenbelegungen aus, um auf ein valides Ergebnis zu kommen. Dieses Ausprobieren hat Struktur und läuft unter dem Backtracking-Algorithmus. Findet Prolog durch die schrittweise Unifizierung (Ersetzung der Teilziele durch Fakten oder Regelrümpfe) eine Lösung, so antwortet Prolog, falls keine Variablen gebunden wurden, mit true, andernfalls mit den Variablen mit ihrem zugeordneten Wert. Im Detail ist dieser komplexe Gesamtalgorithmus in etlichen Literaturquellen nachzulesen. Um mit diesem groben Überblick etwas das Prinzip hinter Prolog etwas verständlich zu machen, folgt nach Erläuterung der Grundbestandteile zur Programmierung dieser Sprache ein kleines Beispiel am Ende von Kapitels 2.2.2.

2.2.2 Bestandteile der Programmiersprache

Ein Hinweis vorab: in dieser Arbeit benutze ich die weit verbreitete freie Entwicklungsumgebung „SWI Prolog“, welche kompatibel ist zum ISO Prolog-Standard [19]. Manche in diesem Unterkapitel aufgeführten Operationen/ Möglichkeiten existieren möglicherweise nicht in anderen Entwicklungsumgebungen oder nur in abgewandelter Form!
Um erste Grundlagen zu setzen, möchte ich mit der Grundstruktur beginnen. Prolog basiert auf der Hornklausellogik, somit bestimmt auch diese die Grundstruktur. Ausgehend von einem Implikationsterm $L \leftarrow L_1 \wedge L_2 \wedge ... \wedge L_n$, logisch umgeformt aus einer Hornklausel, ersetzt man das $\wedge$ durch das Zeichen , und $\leftarrow$ durch $:-$ und somit ist bereits eine korrekt programmierte Zeile Prolog entstanden. Wir haben zwischen Regeln, Fakten und Anfragen unterschieden- diese Unterscheidung kennt auch Prolog.

- **Anfrage**: in Prolog ?- [Prämisse]. . Die Anfrage wird Prolog mithilfe der Fakten und Regeln versuchen, erfolgreich zu unifizieren.

- **Fakt**: in Prolog [Konklusion] :- ., oder in Kurzform [Konklusion].. [Konklusion] kann für eine Konstante oder ein selbstdefiniertes Prädikat mit den dazugehörigen Paramtern stehen. Eine Konklusion ist immer wahr. Prolog wird bei der Unifikation, sofern semantische Übereinstimmung gegeben ist, ausprobieren, den passenden Teil aus der zu bearbeitenden Formel mit diesem Fakt zu unifizieren.

- **Regel**: in Prolog: [Konklusion] :- [Prämisse].. Beim Suchen einer Lösung wird Prolog bei der Unifikation, wenn möglich, die Konklusion in der zu bearbeitenden Formel mit der Prämisse ersetzen. Abstrakt (und der Verständnis halber) gesehen kann man Regeln auch als eine Art Funktion sehen. Wenn man eine Regel durch die Nennung ihres Konklusionnamens und der richtigen Anzahl von Parametern aufruft, für die als Ausgangsvariablen angesehenen Variablen ungebundene Variablen angibt, so kann man (durch Bindung dieser Ausgangsvariablen) die Rückgabe der Regel empfangen und im späteren Verlauf auswerten.
  Bespiel: stellt man die Anfrage test(Input, Output). und es existiert die Regel test(Input, Output) :- Input=Output., so versucht Prolog, den Eingang mit dem Ausgang zu unifizieren.

  ↪ *Bemerkung:* Zu einer Konklusion kann man mehrere Prämissen definieren. Durch den Backtracking-Algorithmus wird Prolog eine Prämisse nach der anderen zur Lösungsfindung ausprobieren. Mehrere Prämissen zu einer Konklusion haben ist logisch äquivalent zu einer Regel mit der Konklusion, gefolgt von allen Prämissen, getrennt durch ;, welches Prolog als ein $\vee$-Zeichen interpretiert

  ↪ *Bemerkung:* Mit Prolog lässt sich leicht Rekursion betreiben, indem beispielsweise die Prämisse die Konklusion mit enthält. Um ein Deadlock zu vermeiden, ist es jedoch unabdingbar, die Konklusion in der Prämisse mit einer vorangestellten Abbruchbedingung zu verknüpfen.

Nun möchte ich einige Operatoren, die der Programmierer benutzen kann, auflisten. Sei dazu a, b und c ein beliebiges Literal in Prolog:

- a==b: Auf Anfrage prüft Prolog, ob es eine Unifizierung gibt, in der a gleich b ist. Dabei werden a und b bis auf die mögliche Bindung der Variablen nicht ausgewertet. So ist 3==3 wahr, (1+2)==(2+1) jedoch falsch.

- a=:=b: wie a==b, nur, dass hier auch zusätzlich a und b arithmetisch ausgewertet wird. So ist (1+2)=:=(2+1) wahr.

- a=b: Prolog unifiziert a mit b. Sind a und b jeweils keine ungebundenen Variablen, so antwortet Prolog mit false, wenn nicht a==b gilt.

- a is b: Wie a=b, nur, dass hier vor der Unifizierung a und b noch arithmetisch ausgewertet werden

- (a)->b;c: Wenn a wahr ist, führt Prolog mit b fort und behandelt nicht c, wenn nicht, dann wird b übersprungen und c behandelt. Vergleichbar mit `if a then b else c fi`

- []: definiert eine leere Liste. Um eine nichtleere Liste zu definieren, schreibt man die Elemente, mit Komma getrennt, in die eckigen Klammern. [Head|Tail] wird die

Variable Head mit dem ersten Element der Liste binden, Tail mit der Liste, aus der das erste Element entfernt ist

- not(a): wahr, wenn a nicht erfolgreich unifiziert werden kann
- fail: gleichzusetzen mit false
- Des weiteren kann Prolog alle arithmetischen Zeichen interpretieren

Weiterhin gilt in Prolog die Regel: Zeichenketten, die mit kleinen Buchstaben oder Zahlenzeichen beginnen, sieht Prolog als Konstanten, Zeichenketten, die mit großen Buchstaben beginnen, als Variablen, die zunächst generell ungebunden (ohne Wert und Typspezifizierung) sind. SWI-Prolog hat noch eine große Reihe von vordefinierten Prädikaten (Regeln, die der Programmierer nicht selbst implementieren muss) [19]. Auch mein Prologvalidator-Programm wird eine Auswahl dieser vordefinierten Prädikate benutzen.

Nun sind die Grundlagen für die Programmierung gelegt. Um nun einen Eindruck vom Auswertungsalgorithmus bezüglich Prolog zu bekommen, folgt nun das Beispiel:
Erinnern wir uns an das Beispiel der gesamten Bachelorarbeit. Reduzieren wir der Einfachheit halber die Service-Komposition `filterAndScale` auf eine Signatur mit null Eingängen und einem Ausgang. Betrachten wir als `LESSTHAN5MB` als statische Eigenschaft. Diese kann entweder wahr oder falsch sein. Wir wollen nun verifizieren, ob es eine Galerie geben kann, die genau diese Eigenschaft für sich als wahr beantworten kann. Somit ergeben sich zwei Regeln und eine Anfrage:

Prologbeispiel zur Überprüfung der Eigenschaft LESSTHAN5MB

```
1 %Fakten
2 filterAndScale(false).
3 filterAndScale(O) :- O=true.
4
5 %Anfrage
6 ?- filterAndScale(LESSTHAN5MB), LESSTHAN5MB==true.
```

Berechnungsweg von Prolog:

1. Prolog liest vom *toplevel*, sprich von der Anfrage filterAndScale(LESSTHAN5MB) und versucht dies zu unifizieren- die ungebundene Variable LESSTHAN5MB zu binden.
2. Prolog findet einen passenden Fakt filterAndScale(false) und unifiziert und bindet, die Anfrage lautet nun false==true
3. false==true ist falsch, dies ist keine positive Antwort, Prolog sucht nach einer anderen Lösung mittels des Backtracking-Algorithmus und geht wieder in diesem Fall zum Anfang

4. Prolog liest filterAndScale(LESSTHAN5MB) und unifiziert nun mit der Regel filterAndScale(O) :- O=true.. Hier wird O an true gebunden und Prolog kann die Anfrage kürzen: true==true.

5. true==true ist wahr, Prolog hat nun eine Lösung gefunden, indem er O erfolgreich an true binden konnte. Die Ausgabe wird entsprechend O=true lauten und somit ist bewiesen, dass es eine Galerie geben kann, für die diese Eigenschaft zutrifft.

3 Verwandte Arbeiten und Grundlagen der Verifikation

Bevor wir uns die Verifikation von Service-Kompositionen mit Prolog anschauen, ist es interessant zu erfahren, welche Forschungsergebnisse bereits vorlagen, bevor diese Arbeit entstand (Kapitel 3.1). Ebenfalls werden in dieser Arbeit wichtige Grundkonzepte zum Vorgehen einer Verifikation genannt (Kapitel 3.3)

3.1 Diskussion der relevanten Literatur zur Verifikation von Service-Kompositionen

Verifikation ist ein wichtiger und bedeutender Teil der Softwareentwicklung. So ist es nicht verwunderlich, dass schon früh im Zeitalter der Computerprogrammierung sich die Forschung Gedanken zur Verifikation und des Debugging (das Lokalisieren von Fehlern, um sie zu beheben) gemacht hat. Im Bereich der Standardsoftwaresysteme gibt es bereits viele Ansätze zur Verifikation und dem Debugging, im Bereich der Service-Kompositionen weniger [13]. Aber auch in diesem Bereich gibt es zumindest im Bereich der Verifikation schon einige Ansätze. Eine Auswahl davon möchte ich in diesem Kapitel diskutieren.
Als ersten Ansatz möchte ich einen Pi-Kalkül-basierten Absatz vorstellen. Dieser Ansatz automatisiert den entscheidenden Prozess der Verifikation von Service-Kompositionen mit Hilfe der algebraischen Pi-Kalkül-Theorie von Robin Milner [15]. Hierbei werden Services als Prozesse verstanden, die kommunizieren- beispielsweise verarbeitet der eine Service den Ausgang des anderen. Dieser Ansatz kann gut zum Herausfiltern von passenden Services aus einer großen Servicemenge benutzt werden [15]. Als zweiten Ansatz möchte ich den der templatebasierten Konstruktion zur Verifikation von Service-Kompositionen vorstellen [18]: hierbei wird die zu verifizierende Service-Komposition mit Hilfe von vor-verifizierten Templates („Zusammensetzungsmuster“) abgebildet. Dazu hat jedes Template, welches als spezifische Workflow-Beschreibung mit Serviceplatzhaltern gegeben ist, eine abstrakte Ontologie, worauf eine spezifische Ontologie abgebildet werden kann. Wird nun ein solches Template mit einer spezifischen Domain und dazu möglichen Services belegt, werden die Serviceplatzhalter mit den konkreten Services ersetzt sowie die abstrakte Ontologie mit der spezifischen Ontologie. Werden dabei keine Restriktionen, die ein Template beinhalten kann, gebrochen, ist gezeigt, dass die spezifische Service-Komposition im abstrakten Template valide ist [18].

3.2 Vorstellung des SMT-Ansatzes

Als dritten für diese Bachelorarbeit zu Grunde liegenden Ansatz zur Verifikation von Service-Kompositionen möchte ich den wissensbasierten SMT-Ansatz von Walther und Wehrheim präsentieren [17]. Dieser Ansatz nutzt Z3 von Microsoft [7], einen SMT-Solver „satisfiability modulo theories solver“ [17]. Dieser SMT-Solver löst logische Formeln, die nach smt-lib-Standard formuliert sind [3]. Dazu erhält dieser zur Verifikation unter dem Wissen der Domäne (optional mit Regeln) den Kontrollfluss der Service-Komposition, die Schnittstellen der benutzten Services und die Vor- und Nachbedingung der Service-Komposition. Dazu werden nach smt-lib-Standard Konzepte der Domäne in Typen definiert, Rollen der Domäne werden als uninterpretierte Funktionen aufgefasst und Regeln der Ontologie (somit der Domäne) in Behauptungen über die uninterpretierten Funktionen aufgefasst. Der Kontrollfluss wird in eine logische Formel $\phi_{SC}(i, o)$ übersetzt, sowie die Vorbedingung $pre_{SC}(i)$ und Nachbedingung $post_{SC}(i, o)$ der Service-Komposition. Mit erfolgreicher Überprüfung aller Schleifeninvarianten inklusive der Sicherstellung der Terminierung dieser ist die Service-Komposition mit der Formel $\forall i, o : pre_{SC}(i) \wedge \phi_{SC}(i, o) \rightarrow post_{SC}(i, o)$ verifiziert [17].
Dieser SMT-Verfikations-Ansatz ist eingebettet in die Eclipse-basierte Toolsuite SeSAME. Da auch mein Ansatz in SeSAME eingebettet sein wird und diese Toolsuite somit sinnvoll ergänzen wird, möchte ich nun kurz auf diese Entwicklungsumgebung eingehen [2]: SeSAME (Service Specification, Analysis, and Matching Environment) wird der Entwicklung gerecht, dass immer mehr Softwarekomponenten in Form von Services gehandelt und zur Verfügung stehen (zum Beispiel über den Google APIs Explorer[1]). Diese können sinnvoll kombiniert komplexe Aufgaben bewältigen. Dabei vereint SeSAME die Funktionalität, Service-Komposition zu modellieren, existierende Services gezielt zu durchsuchen (mittels eines konfigurierbaren Matching-Prozesses) und seine Service-Komposition auf seine Korrektheit und Qualität zu überprüfen. Zwar stellen auch andere Tools diese Funktionen bereit, jedoch nur einzeln und nicht vereint wie bei SeSAME. Um Service-Kompositionen zu modellieren, stellt SeSAME eine Modellierungsumgebung mit eigener Sprache (B3) zur Verfügung, auf dessen Grundlage das in dieser Arbeit entwickelte Prologvalidator-Tool aufbauen wird. Diese Sprache bezüglich der Beschreibung des Kontrollflusses einer Service-Komposition wurde bereits in Kapitel 2.1 beschrieben.

3.3 Hinführung der Verifikation mit Prolog: Vorstellung bisheriger Horn-Klausel-Ansätze

Hornklauseln stellen eine gute Möglichkeit bereit, automatisch Programme formal zu verifizieren, mit der Frage, ob eine bestimmte Formel erfüllbar ist oder nicht [5]. In unserem Beispiel muss sich das beauftragte Unternehmen abstrakt formuliert eine Formel errechnen, die das zu verifizierende Programm modelliert mit einer verletzten Nachbe-

[1] `https://developers.google.com/apis-explorer`

dingung unter erfüllter Vorbedingung, wo nun die Frage gestellt wird, ob diese Formel denn erfüllbar sei.
Um die beiden nachfolgenden Ansätze sowie später meinen Ansatz besser verstehen zu können, möchte ich hier das Prinzip der logischen Programmierung mit Beschränkungen einführen. Da Prolog logische Programmierung voraussetzt [16], sind an dieser Stelle das Bemerkenswerte die Beschränkungen.

▷ **Definition:** Die beschränkte logische Programmierung (*Constraint Logic Programming*, auch mit CLP abgekürzt) ist ein Programmiermethode mit der Erweiterung der beschränkten Hornklauseln. Dabei wird der logischen Formel eine logische Behauptung vorangestellt. Nur, wenn diese erfüllt ist, gilt der Rest der Formel. [5]

Die Programmierung mit CLP in Prolog ergänzt vorangestellt einen Fakt, eine Regel oder Anfrage mit einer Behauptung, die mit dem Rest konjugiert wird. Ist diese Behauptung nicht wahr, so bricht Prolog die Behandlung des Restes des Termes ab, da der Gesamtausdruck nicht mehr wahr werden kann. Somit ist CLP mächtig und flexibel, um Programmsyntax, Semantik oder Regeln für viele verschiedene Programmeigenschaften zu beschreiben [1].
In Prolog wird mit CLP einem Fakt, einer Regel beziehungsweise einer Anfrage eine Beschränkung auferlegt. Im Beispiel dieser Bachelorarbeit würde dies für den Service `applyFilter` (der als Vorbedingung hat, dass das Bild kleiner als 5 MB groß ist) im prologähnlichen Pseudocode bedeuten: applyFilter(Input,Output) :- Input_size<5MB, {Nachbedingung des applyFilter-Services}. Hier ist Input_size<5MB die Beschränkung.

Konkrete Ansätze zur Verifikation mit Horn-Klauseln Ein ganz konkretes Verifizierungsprogramm, welches C-Programme auf Sicherheits- und Lebendigkeitseigenschaften untersuchen kann, ist HSF(C)[2] [8]. Dabei definieren Sicherheitseigenschaften solche Zustände, die nie erreicht werden dürfen. Lebendigkeitseigenschaften beschreiben hingegen eine Terminierung: ein definierter Zustand wird zu einem beliebigen, nicht unendlich entfernten Zeitpunkt auf jeden Fall erreicht werden. HSF(C) verifiziert mit einer Abstraktion von Prädikaten, die ein Bestandteil der Prädikatenlogik sind. Dabei bedient sich dieses Tool des CEGAR-Algorithmus Dieser Algorithmus überprüft zunächst eine grobe Abstraktion des C-Programmes und verfeinert diese dann schrittweise. Wird eine Verletzung (Gegenbeispiel) der gegebenen Bedingungen (Anforderungen) gefunden, prüft der Algorithmus, ob diese Verletzung aufgrund der groben ungenauen Abstraktionsstufe zu Stande gekommen ist oder wirklich echt ist. Ist die Echtheit bestätigt, wird die Invalidität dem Benutzer gemeldet, ansonsten wird mit einem Gegenbeweis gezeigt, dass diese Verletzung auf der Abstraktionsebene nicht vorkommen kann [8]. Dabei benutzt HSF(C) Horn-Klauseln anstatt sonstiger Coding-Prozeduren und iteriert über eine Menge von Prädikaten. Diese Vorgehensweise kann stellenweise an den Backtracking-Algorithmus erinnern. HSF(C) ist bereits erfolgreich in Prolog implementiert und beinhaltet ein C-Frontend, einen Programm-zu-Hornklausel-Übersetzer und einen Hornklauselöser, der

[2] `https://www7.in.tum.de/tools/hsf/`

durch Prolog weniger komplex ausfällt [8].
Ein weiteres breit einsetzbares Verifikationstool für C-Programme mittels Hornklauseln ist VeriMap[3] [1].

3.4 Umgang der bisherigen Ansätze mit Schleifen

Schleifen sind ein mögliches Konstrukt, welches im Kontrollfluss von SeSAME verwendet werden kann [13]. Schleifen können als das komplexeste Konstrukt in seiner Behandlung angesehen werden. Dies ist begründet in folgenden Umständen:

- Schleifen unterteilen sich in zwei Teile: die Schleifenbedingung (welche eine Abfrage darstellt) und dem Schleifenrumpf (welche eine Ausführung darstellt)
- Im Gegensatz zu anderen Kontrollstrukturen wird der Schleifenrumpf nicht null (gegebenfalls der Rumpf einer Verzweigung) oder einmal ausgeführt, sondern die Ausführungsanzahl beträgt je nach Schleifenbedingung und Zustand null bis unendlich viel mal. Dies führt zu unterschiedlichen Herausforderungen
  - Es muss der Fall beachtet werden, dass ein Schleifenrumpf niemals ausgeführt wird
  - Es muss der Fall beachtet werden, dass der Schleifenrumpf endlich viele Male ausgeführt wird: hierbei muss beachtet werden, dass die gleichen Variablen mehrmals Werte zugewiesen bekommen, wird die Schleife zwei- oder mehrmals ausgeführt
  - Es muss der Fall beachtet werden, dass der Schleifenrumpf unendlich oft ausgeführt wird: es erfolgt keine Terminierung, somit ist das Programm fehlerhaft, obwohl zeitgleich ohne Berücksichtigung der Terminierungsüberprüfung der Fall eintreten kann, dass die Service-Komposition als formal korrekt verifiziert werden kann, da aus der Vorbedingung die Nachbedingung errechnet werden kann [17]

Von sofern möchte ich vor der Übersetzung in Prolog auf zwei verschiedene Arten aus der Literatur im Umgang mit Schleifen eingehen, um eine begründete Grundsatzentscheidung für die spätere Übersetzung zu treffen.

3.4.1 Verifikation mittels Bounded Model Checking

Bounded Model Checking bedeutet „Beschränkte Modellprüfung“. Hier wird das zu Verifizierende modelliert und dann zur Überprüfung expandiert, jedoch nur bis zu einer bestimmten oberen Schranke. Dies bedeutet, dass das zu Verifizierende ausgeführt wird inklusive der Abarbeitung der Schleifen, jedoch ohne konkrete Eingabevariablen. Die Idee ist, ein Gegenbeispiel zu suchen innerhalb einer Expansionslänge k (eine Schleife wird maximal k mal durchlaufen). Falls innerhalb dieser Expansion kein Gegenbeispiel

[3] `http://map.uniroma2.it/VeriMAP`

gefunden wurde, nimmt man Korrektheit an [6].
Für Schleifen ist ein Vorgehen mit BMC folgendes:

1. lege (dynamisch oder statisch) eine Obergrenze $n \in \mathbb{N}$ fest
2. wandle den `while B do C od` in eine Verzweigung `if B then C else Skip fi`[4] um
3. Wiederhole diese Verzweigungsanweisung n mal
4. Am Ende dieser n Verzweigungen nehme an, dass `neg B` gilt (die Schleifenbedingung gilt nicht mehr) [6]

↪ *Bemerkung:* Dieses Vorgehen macht das Entfalten der Schleife auf viele einzelne identische Verzweigungen deutlich. Von daher findet man in der Literatur für BMC auf den Begriff „unfolding“[1]
Vorteile der Verifizierung mittels BMC:

- der Service-Kompositions-Programmierer hat keinen Mehraufwand, keine der Aufgaben oder Schritte der Verifikation wird auf ihn ausgelagert
- ist die Obergrenze hoch genug angesetzt und es werden tatsächlich alle möglichen Schleifendurchläufe durch das Entfalten abgebildet, so wird durch das BMC eine präzise Lösung ohne Informationsverlust errechnet

Nachteile der Verifizierung mittels BMC:

- Während der Fall, dass der Schleifenrumpf keinmal oder nur wenige Male ausgeführt wird, präzise von BMC behandelt wird, so kann das Abschneiden weiterer Schleifendurchläufe über die Grenze hinaus zu einem falschen Verifikationsergebnis führen. Ebenfalls wäre das Verifikationsergebnis mit BMC falsch, wenn es einen Fall gibt, in der die Schleife nicht terminiert. Mit zusätzlichen Workarounds (beispielsweise mit einer vorherigen Prüfung auf Terminierung) kann man diesem Problemfall jedoch stückweise entgegentreten
- Wird das n höher als die tatsächlich möglichen Schleifendurchläufe gewählt, produziert man unnötigen Rechenaufwand, da die überfälligen Verzweigungen ebenfalls vom Verifikationstool behandelt werden müssen
- Bei einem $n > 1$ werden Werte von Variablen im Schleifenrumpf überschrieben. Dadurch geht die SSA-Form verloren, wenn diese denn zuvor bestand

Ein Verifikationstool, welches erfolgreich BMC implementiert, ist VeriMAP [1].

[4] Der Code entstammt von einem Kontrollfluss einer Service-Komposition, siehe hierzu Kapitel 2.1

3.4.2 Verifikation mittels Schleifeninvariante

Die Verifikation von Schleifen mittels Schleifeninvariante bedient sich der Annahme, dass der Programmierer jede Schleife mit einer validen Schleifeninvariante ausstattet.

▷ **Definition:** Eine Schleifeninvariante definieren wir als einen prädikatenlogischen Ausdruck, der innerhalb einer Schleife in einer Service-Komposition in SeSAME mit dem Schlüsselwort `@Invariant:` definiert wird. Dieser Ausdruck muss am Anfang und am Ende einer Ausführung des Schleifenrumpfes immer wahr sein, unabhängig von der Anzahl bisheriger Schleifenläufe [17]

↪ *Bemerkung:* Die Schleifeninvariante ist ein beliebtes Mittel innerhalb der Verifikation.
Somit lassen sich alle Fälle, in denen die Schleife ein bis endlich mal oft ausgeführt wird, auf eine Formel herunter brechen. Für den Fall, dass der Schleifenrumpf nie ausgeführt wird, nehmen wir die negierte Schleifenbedingung an. Dies ist offensichtlich, da diese in einem solchen Fall nie erfüllt war, und somit deren Negation erfüllt gewesen ist.
Jedoch muss bei dieser Art der Verifizierung noch der Fall der Nichtterminierung beachtet werden, um eine totale Korrektheit nachzuweisen. Um Terminierung garantieren zu können, muss nachgewiesen werden, dass die Schleifenbedingung von einem Zustand abhängt, der vom Schleifenrumpf beeinflusst wird. Wird beispielsweise in der Schleifenbedingung eine Variablenwertober- oder Untergrenze festgelegt (beispielsweise durch den Ausdruck `var<x`), so muss nachgewiesen werden, dass sich mit jedem Schleifendurchlauf die Variable echt der Grenze nähert [17].
Vorteile der Verifizierung mittels Schleifeninvariante:

- Man benutzt einen prädikatenlogischen Ausdruck, der das Verhalten der gesamten Schleife beschreibt: die Schleifeninvariante
- Bei der Annahme, dass Schleifen immer terminieren: leicht in eine logische Sprache zu implementieren

Nachteile der Verifizierung mittels Schleifeninvariante:

- Der Service-Kompositions-Programmierer muss selbst die Schleifeninvariante ohne allgemeine Algorithmenhilfe finden [17]. Bei nicht trivialen Schleifen kann der Prozess des Probierens und Findens einer präzisen Schleifeninvariante sehr lange dauern- der Mehraufwand wird auf den Service-Kompositions-Programmierer ausgelagert
- Grundsätzlich kann eine Schleifeninvariante mit logischer Programmierung validiert werden [17]. Wird dies jedoch nicht gemacht, kann das Verifikationsergebnis durch gegebenfalls fehlerhaft vom Service-Kompositions-Programmierer aufgestellte Invariante verfälscht werden
- definiert der Service-Kompositions-Programmierer die Schleifeninvariante unpräzise beziehungsweise nichts aussagend (wie beispielsweise `@Invariant: true`), so

wird auch das Ergebnis der Verifikation unpräzise. Durch fehlende Annahmen fehlen Informationen für nachfolgende Beweise. Somit kann beispielsweise eine Nachbedingung, die mit präziser Schleifeninvariante herleitbar gewesen wäre, mit unpräziser Schleifeninvariante nicht nachgewiesen werden

Ein Verifikationsansatz, der mit einer Schleifeninvariante Schleifen bearbeitet, ist der SMT-Ansatz von Walther und Wehrheim [17].

Grundsatzentscheidung für meinen Ansatz der Verifikation mit Prolog Auch ich werde mich für meinen Ansatz für die Verifikation mit Schleifeninvarianten aufgrund der Vorteile und mangelnder guter Alternativen entscheiden.

4 Übersetzung einer Service-Komposition in Prolog

Mit den gelegten Grundlagen in Kapitel 2 und Grundkenntnissen über Verifikation und deren aktuellen Forschungsstand (Kapitel 3) komme ich nun zur konkreten Übersetzung von einer Service-Komposition in ein Prologprogramm kommen. Über das entstandene Prologprogramm wird mittels einer ebenfalls generierten Anfrage die Service-Komposition verifiziert werden kann. Dabei setze ich wie der SMT-Solver von Walther und Wehrheim als Eingang eine Ontologie, die Service-Komposition als B3-Kontrollfluss, die Vor- und Nachbedingung der Service-Komposition sowie die Spezifizierungen der vorkommenden Services als gegeben voraus [17]. Neben dem SMT-Solver wurde diese Übersetzung auch von der Arbeit von Bjørner et al. inspiriert [5].
Als Annahmen über den Eingang lege ich fest:

- Annahmen über den Kontrollfluss der Service-Komposition an sich:
  - jede Schleife ist mit einer Schleifeninvariante ausgestattet und terminiert
  - es gibt keine Iterationsschleifen über deklarierte Mengen
  - Services haben genau eine Ausgangsvariable
  - Services modifizieren ihre Eingangsvariablen nicht (Wenn vor der Ausführung eines beliebigen Services die Vorbedingung gültig ist, so ist sie es auch nach der Ausführung.)
  - die prädikatenlogischen Terme der Nachbedingungen von Services müssen erfüllbar sein. So ist eine Nachbedingung `false` oder `x and neg x` nicht erlaubt
- Annahmen über verarbeitete prädikatenlogische Formeln:
  - es sind nur prädikatenlogische Formeln mit reduziertem Umfang zugelassen (wie in Kapitel 2 definiert)
  - folgende Tautologien dürfen nicht verwendet werden: wenn Variable x dem Boolean-Konzept zugeordnet ist: `x or neg x`. Diese Tautologie würde mit meiner Übersetzung als falsch ausgewertet werden, wenn x nicht eindeutig einen Wert zugeordnet bekommt

Neben diesen Annahmen gibt es noch eine weitere Annahme, die den Kontrollfluss der Service-Komposition in SSA-Form verlangt. Diese Annahme können wir jedoch selbst aus jedem beliebigen Kontrollfluss erschaffen. Dazu benutze ich den internen

SSA-Transformer von SeSAME. Um die Funktionsweise des SSA-Transformers nachzuvollziehen, folgt ein kurzes Beispiel. Dabei handelt es sich um einen Auszug aus der Service-Komposition des Beispieles der Bachelorarbeit:

Auszug des Kontrollflusses der Service-Kompositon zur Bildverarbeitungssoftware

```
1 Image o := scaleDown(i);
2 o := applyFilter(o);
```

Wie wir hier sehen, wird die Variable o nicht nur einmal, sondern mehrfach definiert. Dies stellt ein Bruch mit der SSA-Form da. Um diesen Konflikt nun zu lösen, wenden wir das Prinzip der Indexierung an. Jede Deklaration zählt dabei den Index um eins nach oben. Angewandt auf den Auszug bedeutet dies:

Auszug des Kontrollflusses der Service-Kompositon zur Bildverarbeitungssoftware in SSA-Form

```
1 Image o1 := scaleDown(i);
2 Image o2 := applyFilter(o1);
```

4.1 Mögliche Anfragen und Interpretation des Ergebnisses

Die Service-Komposition ist ein komplexes Konstrukt aus einer Signatur, die Vor- und Nachbedingung beinhaltet, der Information aus der Ontologie, Informationen über eingebundene Services und der Beschreibung des Service-Komposition-Programmes. Letzteres besteht aus hintereinander geschalteten Kommandos, die unter anderem Aufrufe von Services beinhalten können.
Um mit Prolog verifizieren zu können, muss man auf einer passenden Datenbasis eine *Anfrage* stellen. Diese entscheidende Anfrage wird von Prolog beantwortet. Damit eine Antwort entsteht, die zur Verifizierung der Service-Komposition verwendbar ist, ist das Stellen der richtigen Anfrage entscheidend.
Dabei unterscheide ich zwischen zwei Grundfragen an die Verifizierung:

1. Der Benutzer möchte wissen, ob *eine* valide Eingabe zu seiner Service-Komposition *existiert*: $[\![pre]\!], [\![sc]\!], [\![post]\!]$[1]. Dabei entspricht $[\![pre]\!]$ der übersetzten Vorbedingung der Service-Komposition. $[\![sc]\!]$ ist die übersetzte Service-Komposition an sich- ihre Ausführung wird zur besseren Übersichtlichkeit in eine eigene Regel ausgelagert. $[\![post]\!]$ ist die übersetzte Nachbedingung der Service-Komposition. In natürlicher Sprache formuliert würde die Prologanfrage lauten: gilt die Vorbedingung, die Service-Komposition und die Nachbedingung? Oder anders formuliert: gibt es eine Unifikation (kann man die Variablen in einer zulässigen Form binden), so dass die sowohl Vorbedingung erfüllt ist, als auch sicher die Nachbedingung nach Durchführung der Service-Komposition? Dabei wird die Vorbedingung als gegeben

[1] Für einen beliebigen Ausdruck a gilt: $[\![a]\!]$ stellt a in Semantikklammern dar und weist auf eine semantische Funktion $[\![a]\!] : \mathcal{P} \to \{wahr, falsch\}$ hin

(Variablenbindung) angesehen und die Nachbedingung als gefragt (Variablenabfrage)[2]. Gibt Prolog eine valide Variablenbindung zurück (nicht false.), so ist die Gültigkeit der Service-Komposition für die Existenz einer validen Eingabe gezeigt, ansonsten ist die Service-Komposition widerlegt.

2. Der Benutzer möchte wissen, ob *alle* validen Eingaben zu seiner Service-Komposition zu einer erfüllten Nachbedingung führen. Dies ist gleich dem Fall des Existenzbeweises, nur, dass wir das vordefinierte Prädikat forall benutzen. Als ersten Parameter geben wir die Vorbedingung ein sowie den Kontrollfluss der Service-Komposition. Als zweiten Parameter benutzen wir die Nachbedingung. Das Prädikat forall ist nun so definiert, dass es genau dann `true` zurück gibt, wenn für alle möglichen Variablenbindungen innerhalb des ersten Parameters der zweite Parameter mit Erfolg geprüft werden kann [19]. In anderen Worten: für alle Eingänge, die der Vorbedingung genügen und von der Service-Komposition abgearbeitet werden, gilt sicher die Nachbedingung. Dies ist genau das, was wir verifizieren wollen.

4.2 Grundsätzliche Übersetzungsanweisung

Bei der Verifikation mit Prolog ist das Übersetzen der eingegebenen Information nach Prolog notwendig. Dabei unterscheiden wir zwei Sorten von Ebenen:

- auf der oberen Ebene steht das B3-Model der Service-Komposition in der Entwicklungsumgebung SeSAME inklusive Ontologie. Auf dieses Modell hin kann beispielsweise der Kontrollfluss der Service-Komposition abgefragt werden kann

- In den darunterliegenden spezifischeren Ebenen können, zum Beispiel in einer Vor- und Nachbedingung eines Services, prädikatenlogische Formeln erster Stufe definiert sein

Auch, wenn es zwischen den beiden Konzepten Gemeinsamkeiten gibt, die ich in der Implementierung berücksichtigen werde, so gibt es doch auch größere Unterschiede. Die Service-Komposition beschreibt die Oberstruktur und Vorgehensweise des Programmes und enthält wiederum der Prädikatenlogik ferne Konzepte wie Schleifen. Die Details und Bedingungen werden in einer prädikatenlogischen Formel beschrieben, beispielsweise, was vor einem Service-Aufruf gelten muss und gegeben dieser Bedingung nach diesem Aufruf gilt.
Im späteren Prozess werde ich manche Kontrollstrukturen der Service-Komposition direkt auf die Übersetzung eines prädikatenlogischen Terms übertragen. Daher werde ich zunächst auf die Übersetzung prädikatenlogischer Formeln eingehen. Dabei werde ich Details und Sonderbehandlungen in Kapitel 4.3 auslagern. Somit ist zuerst eine Übersicht gegeben, um danach die Sonderfälle besser verstehen zu können.

[2]Eine nicht angenommene Vorbedingung würde keinen Aussagenwert liefern, da man aus einer negierten Aussage in der Logik alle Aussagen folgern kann

4.2.1 Grundsätzliche Übersetzungsanweisung einer logischen Formel

In diesem Kapitel wird eine Übersetzungsroutine für prädikatenlogische Formeln mit reduziertem Umfang beschrieben. Dieser Begriff wurde in Kapitel 2 definiert und zeichnet sich durch eine reduzierte syntaktische Mächtigkeit aus. Dies vereinfacht die Übersetzungsroutine. In Kapitel 4.3.3 werden noch zusätzlich Ansätze zur Übersetzung derjenigen Konstrukte behandelt, die durch den reduzierten Umfang ausgeschlossen sind.
Wenn wir nun eine prädikatenlogische Formel übersetzen, so übersetzen wir von außen nach innen. Bei einer Formel, die nicht aus einem Atom besteht, übersetzen wir somit zuerst einen binären oder unären Ausdruck.

Binäre prädikatenlogische Ausdrücke

▷ a und b seien jeweils prädikatenlogische Formeln. Eine **Konjunktion** ⟦a⟧ `and` ⟦b⟧ wird mit ⟦a⟧, ⟦b⟧ übersetzt

Nehmen wir uns von unserem Beispiel der Verifizierung des Bildbearbeitungsprogrammes die Nachbedingung des Services `scaleDown` vor:
`lessThan5MB(out1) and isVariantOf(out1, in1)`.
So lautet die Übersetzung in Prolog davon
⟦$lessThan5MB(out1)$⟧, ⟦$isVariantOf(out1, in1)$⟧.
Diese Übersetzung ist trivial, weil das Kontrollprädikat genau als Konjunktion ausgeschrieben ist [19].

▷ a und b seien eine prädikatenlogische Formel. Eine **Disjunktion** ⟦a⟧ `or` ⟦b⟧ wird mit ⟦a⟧; ⟦b⟧ übersetzt

Nehmen wir uns von unserem Beispiel der Verifizierung des Bildbearbeitungsprogrammes die modifizierte Nachbedingung des Services `scaleDown` vor:
`lessThan5MB(out1) or isVariantOf(out1, in1)`.
So lautet die Übersetzung in Prolog davon
⟦$lessThan5MB(out1)$⟧; ⟦$isVariantOf(out1, in1)$⟧.
Diese Übersetzung ist trivial, weil das Kontrollprädikat genau als Disjunktion ausgeschrieben ist [19]. Dabei unifiziert Prolog zunächst den linken Ausdruck mit der Anfrage. Falls dies nicht zum Erfolg führt, geht Prolog mittels Backtracking wieder zu der Ebene des Kontrollprädikates zurück und unifiziert den rechten Ausdruck mit der Anfrage. Eine Alternative der Übersetzung wäre, wenn diese Disjunktion bei einer Generierung einer Prologregel vorkommt, die Prologregel zu duplizieren. In der einen Regel würde dann der linken Teil behandelt und in der anderen der rechten Teil. Beide Prologregeln müssen dann bis zur vollständigen Abarbeitung des prädikatenlogischen Ausdrucks weiter aufgebaut werden. Da dies von der Programmierung aufwändiger als das Setzen eines ;-Zeichens ist, entscheide ich mich gegen diese Alternative.

▷ a und b seien eine prädikatenlogische Formel. Eine **Implikation** ⟦a⟧ `=>`⟦b⟧ wird mit als `neg` ⟦a⟧ `or` ⟦b⟧ übersetzt

Nehmen wir als Beispiel die Formel $true \rightarrow false$: `true => false`. So lautet die Übersetzung in Prolog davon false; false, was als `false` ausgewertet werden wird. Dies ist korrekt, denn $true \rightarrow false$ kann niemals wahr werden.

In Prolog gibt es kein direkt gleichbedeutendes Kontrollprädikat für die Implikation. Zwar gibt es :Condition -> :Action, jedoch gibt Prolog im Falle von einer nichterfüllten Bedingung ein `false` zurück, was der Definition der Implikation widerspricht. Ein Workaround wäre :Condition -> :Action ; true [19]. Obwohl es diese Möglichkeit von Prolog gibt, habe ich mich, um bei einem schlanken Übersetzungskonzept zu bleiben, mich für den Zwischenschritt von der Formelumwandlung mit dem Implikationsgesetz entschieden [9].

▷ a und b seien eine prädikatenlogische Formel. Ein **Vergleich** $[\![a]\!]$ = $[\![b]\!]$ wird im Allgemeinen mit $[\![a]\!] = [\![b]\!]$ übersetzt

Der Vergleich ist ein nicht trivialer Fall und wird in Kapitel 4.3 dieser noch einmal aufgenommen.
Nehmen wir als Beispiel die Formel $a = true$: $[\![a]\!]$=`true`. So lautet die Übersetzung in Prolog $[\![a]\!]$=true
Der Vergleich drückt aus, dass der linke Term den selben Wert hat wie der rechte, sprich, dass der linke Teil mit dem rechten ersetzt werden kann. Genau das ist der Vergleichs- und Unifizierungsoperator = in Prolog: er unifiziert den linken mit dem rechten Ausdruck, macht diese wertgleich (wenn möglich, d. h. wenn Variablen auf beiden Seiten schon gebunden sind, oder, wenn atomare nicht unifizierbare Ausdrücke deklariert sind, muss Prolog gegebenfalls und richtigerweise ein `false` zurück geben. Das muss Prolog auch bei der Kontradiktion $true = false$- eine generell falsche Aussage) [19].

Unäre prädikatenlogische Ausdrücke

Sei a eine prädikatenlogische Formel. Eine **Negation** `neg a` schaltet das Flag *isNegated* um, welches im weiteren Verlauf der Übersetzung von a über eine Modifizierung von a auf *false* gebracht wird. Effektiv findet bei vollständiger Ausführung der Übersetzung eine Umformung in die Negationsnormalform statt. Dabei finden unter anderem folgende universal geltende logischer Gesetzte Anwendung [9]:
Sind a und b prädikatenlogische Ausdrücke, so gilt:

▷
- $\neg(a \wedge b) \Leftrightarrow \neg a \vee \neg b$
- $\neg(a \vee b) \Leftrightarrow \neg a \wedge \neg b$
- $\neg\neg a \Leftrightarrow a$
- $\neg a = b \Leftrightarrow a \neq b$
- $\neg true \Leftrightarrow false$, $\neg false \Leftrightarrow true$

In unserem Beispiel gibt es als Verzweigungsbedingung den Ausdruck `neg lessThan5MB(i)`. Setzen wir `lessThan5MB(i)` dem Ausdruck a gleich, so lautet die Verzweigungsbedingung $\neg a \Leftrightarrow \neg(a = true)$. Zur Negationseliminierung ziehen wir die Negation in diesen

Ausdruck hinein, somit behandeln wir im nächsten Schritt $a = \neg true \Leftrightarrow a = false$.
Die Bedeutung der Negation hängt von a ab, also dem, was negiert wird. Von daher wird die Negation durch die Transformation in a umgesetzt. In Prolog gibt es aber auch das Kontrollprädikat \+ beziehungsweise not. Dieses gibt `true` zurück, wenn der davon betreffende Ausdruck nicht nachgewiesen werden kann [19]. Im ersten Moment erscheint dies die Lösung für die Übersetzung einer Negation zu sein, jedoch scheitert dieses Kontrollprädikat, wenn der betreffende Ausdruck nicht nachgewiesen werden soll, sondern Variablen gebunden werden sollen. So gibt der Ausdruck not(X=zuweisung) `false` zurück, wenn X noch ungebunden ist- X wird nicht unifiziert. Dies ist in einigen Fällen ein unerwünschtes Verhalten, weswegen ich mich gegen dieses Kontrollprädikat entschieden habe.

Atomare prädikatenlogische Ausdrücke

▷ Sei a eine **Konstante**. `a` wird ohne Veränderung mit a übersetzt

Zunächst kennen wir von SeSAME nur die Konstanten `true` und `false`. Hinzu kommen Zahlen, welche auch direkt ohne Veränderung zu übersetzen sind. Man könnte nun um selbstdefinierte weitere Konstanten erweitern. Dafür ist in Prologvalidator ebenfalls eine Routine vorgesehen. Diese gibt im Allgemeinen auch eine direkte Übersetzung aus (mit dem Unterschied, dass Groß- zu Kleinbuchstaben umgewandelt werden). Jedoch möchte ich mich in diesem Kapitel auf die zwei vordefinierten Konstanten beschränken.
Diese Übersetzung ist trivial, weil Prolog syntaktisch und semantisch gleiche Kontrollprädikate zu den zwei Konstanten kennt: true - dieser Ausdruck ist immer wahr- und false (Alternativschreibweise: fail) - dieser Ausdruck ist immer falsch [19].

▷ Sei a eine **Variable**. `a` wird großgeschrieben ohne sonstige Veränderungen mit A übersetzt

Einzeln stehende Variablen in einer prädikatenlogischen Formel werden von SeSAME als ein boolescher Vergleich gewertet. Somit muss dies die Übersetzung gleich tun. Eine einzelne Variable `a` wird als `a=true` übersetzt und `neg a` als `a=false`.
Diese Übersetzung ist trivial, da Prolog alle Zeichenketten, die mit einem Großbuchstaben beginnen, als Variablen wertet. Die semantische Bedeutung einer Variable in Prädikatenlogik und der in Prolog unterscheidet sich nicht [19].

▷ Sei s ein **Service**, $i_1, ..., i_n$ die Eingänge und o der Ausgang. `s(`$[\![i_1]\!]$`, ..., `$[\![i_n]\!]$`)`, $[\![o]\!]$ wird übersetzt mit I1 = $[\![i_1]\!]$, ..., In = $[\![i_n]\!]$, s(I1, ..., In, O) übersetzt

Jeder Service s ist in einer eigenen Prologregel s(I1, ..., In, O) :- $[\![Vorbedingung]\!]$ -> $[\![Nachbedingung]\!]$; true beschrieben. Dies ist semantisch gleichzusetzen mit $Vorbedingung \rightarrow Nachbedingung$- denn nur, wenn ich die Vorbedingung garantieren kann, kann ich gegebenfalls auch etwas über den Ausgang sagen [17]. Im anderen Falle kann ich keine Aussage treffen, die Ausgangsvariable bleibt ungebunden und der Service-Aufruf wird in eine einzelne immer wahre virtuelle Maschineninstruktion umgewandelt [19].
Somit ist letztlich die Übersetzung eines Service-Aufrufes trivial. Da die Eingänge beliebige prädikatenlogische Formeln sein können, so lange es keine Typkonflikte gibt, lagere ich die Bearbeitung dieser aus dem Aufruf der Prologregel für den Service aus und binde

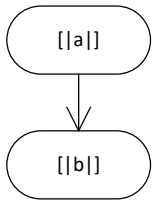
das Ergebnis jeden Parameters vorerst an eine Zwischenvariable.

Die Übersetzung einer Funktion oder Prädikates erfolgt aufgrund der Komplexität in Kapitel 4.3.2.

4.2.2 Grundsätzliche Übersetzungsanweisung einer Service-Kompositions-Beschreibung

Der Kontrollfluss einer Service-Komposition ist prinzipiell simpler aufgebaut als die Datenstruktur einer prädikatenlogischen Formel. Die fünf Bausteine, die innerhalb eines Kontrollflusses existieren können, sind nun mit Übersetzung und Erklärung aufgeführt.

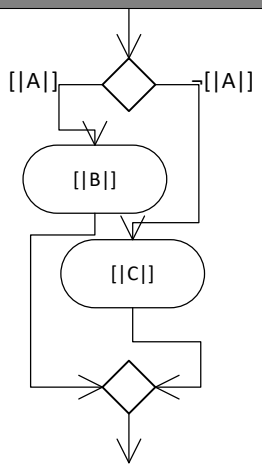
Sequenz



Anweisung: $[\![A]\!]$; $[\![B]\!]$ | In Prolog: $[\![A]\!]$, $[\![B]\!]$

Der ganze Kontrollfluss der service-Komposition wird schrittweise abgearbeitet. Dabei gilt jedes Sequenzglied in Abhängigkeit mit den vorher ausgeführten Sequenzgliedern. Genau das wird durch eine Konjunktion in Prolog simuliert: alle Glieder werden geprüft und in Abhängigkeit der Umgebung ausgeführt.

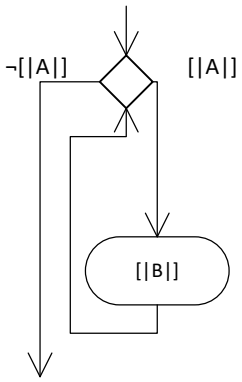
Verzweigung



Anweisung: `if` $[\![A]\!]$ `then` $[\![B]\!]$ `else` $[\![C]\!]$ `fi` | In Prolog: $([\![A]\!], [\![B]\!]);([\![\neg A]\!], [\![C]\!])$

Hinter einer Verzweigung steckt die prädikatenlogische Formel $([\![A]\!] \wedge [\![B]\!]) \vee (\neg[\![A]\!] \wedge [\![C]\!])$- wenn die Bedingung wahr ist, und nur dann, gilt in der Ausführung der Rumpf des if-Teiles. Und wenn die Bedingung nicht wahr ist, und nur genau dann, gilt in der Ausführung der Rumpf des else-Teiles. Genau dies simuliert Prolog exakt, da erst die Bedingung validiert wird und Prolog nur bei einer erfolgreichen Unifizierung auch den Rumpf des if-Teiles behandeln wird- analog in der Argumentation dazu else-Teil. Durch das Verwenden des Operators $=$, sollte $[\![A]\!]$ Variablen beinhalten, wird auch automatisch der Fall behandelt, dass zum Zeitpunkt des Eintretens in der Schleife aufgrund mangelnder Faktenlage nicht entschieden werden kann, ob die Bedingung gilt oder nicht: Prolog validiert einmal unter der Annahme der Bedingung und einmal unter Annahme der negierten Bedingung. Kann die Bedingung jedoch validiert werden, so sind vorkommende Variablen in der Bedingung gebunden und Prolog wird mit einem `false` den nicht erfüllbaren Verzweigungspfad abbrechen, da in dem Falle (sei X eine Variable in der Bedingung) X=<wert> nicht unifiziert werden kann.

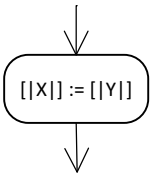
Schleife



Anweisung: `while` $[\![A]\!]$ `do` $[\![B]\!]$ `od` (mit der gegebenen Schleifeninvariante $[\![I]\!]$) | In Prolog: $[\![I]\!], [\![\neg A]\!]$

Schleifeninvarianten gelten vor, während (am Anfang und am Ende des Schleifenrumpfes) und nach der Schleife. Da diese das Verhalten einer Schleife beschreiben, brauchen wir uns somit den Schleifenrumpf als solchen nicht anzusehen. Somit genügt das Setzen des Wissens aus der Schleifeninvariante für die Übersetzung der Schleife. Zusätzlich, nach der Schleife (egal, ob sie gar nicht ausgeführt wurde oder so oft, bis die Schleifenbedingung nicht mehr wahr ist), wissen wir, dass die Negation der Schleifenbedingung nun gilt. Jedoch liegt hierin ein zu lösendes Problem: wenn in der Schleifenbedingung Variablen abgefragt werden, deren Werte wir konkret kennen, die also bereits gebunden sind, werden wir beim Setzen der Negation versuchen, gebundenen Variablen einem Wert zuzuweisen, was nicht im Sinne von Prolog ist und mit `false` antworten wird. Ein Vorschlag zur Umgehung des Problems ist, die Nachbehandlung der Schleifen in den Kontrollfluss auszulagern und dann auf diesen die SSA-Transformation anzuwenden.

Zuweisung



Anweisung: $[\![X]\!]$:= $[\![Y]\!]$ | In Prolog: $[\![X]\!] = [\![Y]\!]$*

*Diese Übersetzung ist nur im Allgemeinen anwendbar, jedoch nicht in bestimmten Spezialfällen (Kapitel 4.3.1). Im Grunde genommen wird eine Zuweisung wie eine Gleichung einer prädikatenlogischen Formel gehandhabt (Kapitel 4.2.1). Eine besondere Form der Zuweisung ist der Serviceaufruf. In diesem Falle wird die Ausgangsvariable der Prologregel, die den Service beschreibt, mit der zugewiesen Variable unifiziert.

Skip (leerer Befehl)

Anweisung: `Skip` | In Prolog: `true`

Ein `Skip` ist ein leerer Befehl, immer wahr ohne Seiteneffekte. Dies gleicht dem `true`-Kontrollprädikat in Prolog [19], die Übersetzung ist somit trivial.

4.3 Die Notwendigkeit zweier Übersetzungsprinzipien

Gehen wir nun zurück zu unserem Beispiel: ein Student entwickelte einen Service inklusive dazugehörigem Objekteigenschaftshandling, der überprüft, ob es sich bei der Eingabe der Bildbearbeitungssoftware um ein Bild handelt, also vom Konzept *Image* ist. Dazu spezifiziert er den Service folgendermaßen:

- Service *isImage*
  - Eingänge `i:?` (unbekanntes Objekt)
  - Ausgänge: `o:Boolean`
  - Vorbedingung: `i=iImage`
  - Nachbedingung: `o` (wenn die Vorbedingung erfüllt ist, dass `i` gleich der Konstante `iImage` ist, dann gebe `o=true` zurück)

▷ Die Übersetzungsanweisung zur Erstellung einer Prologregel zu einem Service s mit den Eingängen $i_1, i_2, ..., i_n$ und dem Ausgang o sowie der Vorbedingung $[\![pre]\!]$ und der Nachbedingung $[\![post]\!]$ lautet: $s(I_1, I_2, ..., I_n, O)$:- $(([\![pre]\!])->[\![post]\!])$; true.

Bildet man nun daraus mit den Übersetzungsansweisungen die Prologregel, erhält man isImage(I,O) :- ((I=iImage)-> O=true); true. Für den Fall, dass i=iImage ist, bindet die Regel die Ausgangsvariable erfolgreich an true. Für den Fall, dass I an einen anderen Wert gebunden ist, wird die Nachbedingung nicht ausgefüllt- dies ist ebenfalls wie erwartet. Doch für den Fall, dass I ungebunden ist, also kein Wissen vorliegt (und wir somit annehmen, dass es sich um kein Objekt des Konzeptes *Image* handelt) wird I an iImage gebunden und O an true- dies ist ein ungewolltes Verhalten.
Dies liegt an der grundsätzlichen Unterscheidung von Prolog, ob Terme direkt unifiziert werden sollen (und man das Ergebnis abfängt, ob die Unifikation erfolgreich gewesen ist), wie es beim Operand = der Fall ist. Oder ob man lediglich Wissen abfragen möchte, man beispielsweise Ausdrücke vergleichen möchte wie es beim Operand == der Fall ist.
So wollen wir in unserem Beispiel nicht I in der Vorbedingung unifizieren (binden), sondern nur abfragen. Richtig übersetzt müsste also die Regel zur Service-Komposition in unserem Studentenbeispiel lauten: isImage(I,O) :- ((I==iImage)-> O=true); true.
Was aber nun auffällt, ist, dass wir zwei Übersetzungen für die gleiche Klasse (in diesem Fall der Klasse der Gleichungen) haben: einmal wird mit dem Operand == übersetzt und einmal mit =. Dies führt zu der schlussendlichen Folgerung, dass wir zwei Übersetzungsalgorithmen benötigen. Den ersten benötigen wir für den *set*-Fall. Hier füge ich Wissen hinzu und betreibe aktiv Unifikation. Sollte das wegen Variablengebundenheit scheitern, bekomme ich false als Antwort. Den zweiten Algorithmus benötigen wir für den *get*-Fall. Hier frage ich mein gesammeltes Wissen ab. In diesem Fall wird mein Wissen nicht erweitert. In vielen Übersetzungsmethoden unterscheiden sich diese Fälle nicht. Die gewichtigsten Unterschiede treten bei Gleichungen und Vergleichen (4.3.1) auf. Alle weiteren wichtigen Punkte der Übersetzung sind in diesem Kapitel aufgelistet.

Anwendungsfälle im allgemeinem Kontext Ob ein *get* oder *set*-Fall vorliegt, muss vom Übersetzer individuell entschieden werden. Dabei lässt der jeweilige Rahmenkontext Rückschlüsse auf die Bestimmung des Falles zu, siehe dazu Tabelle 4.1.

get-Fall	*set*-Fall
Nachbedingung der kompletten Service-Komposition	Vorbedingung der kompletten Service-Komposition
Vorbedingung eines Services (bei Service-Aufrufen)	Nachbedingung eines Services (bei Service-Aufrufen)
Zuweisung: der Term, der zugewiesen werden soll	Zuweisung: Variable, die einen Wert zugewiesen bekommt
Parameter einer Funktion/ Service-Aufruf	alle sonstigen Kontrollflussanweisungen

Tabelle 4.1: Entscheidungstabelle, ob ein *set*- oder *get*-Fall vorliegt

4.3.1 Gleichungen und Vergleiche übersetzen

In diesem Kapitel behandle ich Gleichnisse und Vergleiche, die sich in der Prädikatenlogik mit dem =-Operand (aber auch bei einer geordneten Menge mit >=, >, <, <=) auszeichnen und im Kontrollfluss der Service-Komposition als Zuweisung zeigen.
Dabei wird bereits am Wortlaut der Unterschied zwischen dem *get*- und *set*-Fall deutlich. Im *get*-Fall spreche ich von einem Vergleich mit der Frage, ob dieser Vergleich wahr oder falsch ist. Im *set*-Fall von einer Zuweisung/ Unifikation- unser Wissen wird erweitert.
Des Weiteren hängt die Übersetzung von dem Konzept der beteiligten Variablen und Zuweisungen ab. Durch die Typprüfung von SeSAME kann ich die nicht einschränkende Annahme treffen, dass die Konzepte des linken und rechten gesamten Ausdruckes der Gleichung übereinstimmen.

Konstanten (außer Booleans) und Strings

Der einfachste Fall ist, wenn ein direkter Vergleich stattfinden kann. Dies ist bei Konstanten und Zeichenketten, sogenannten *Strings*, der Fall. Um Wissen aus Negationen abbilden zu können (beispielsweise $\neg(i = iImage) \Leftrightarrow i \neq iImage$), definiere ich, dass eine solche Negation als eine normale Gleichung behandelt wird mit dem Unterschied, dass bei einer Variablenbindung ein neg/ vor dem zugeordneten Wert vorgeschaltet wird. Sei nun $[\![a]\!] = [\![b]\!]$ der zu übersetzende Term.

***get*-Fall** Im *get*-Fall lautet die Übersetzung zur Überprüfung: $[\![a]\!]==[\![b]\!]$.

***set*-Fall** Im *set*-Fall kann direkt übersetzt werden: $[\![a]\!]=[\![b]\!]$. Da wir eine SSA-Form voraussetzen, und somit voraussetzen können, dass eine zu definierende Variable ungebunden ist, wird die Unifikation immer erfolgreich sein.

Booleans

Booleans sind Konstanten aus der Menge {`true`, `false`}. Die Schwierigkeit besteht hierin, dass ein komplexer Term einen booleschen Rückgabewert haben kann, somit also

erst ausgewertet werden muss, bevor die Behandlung der Gleichung erfolgt. So würde ohne Vorbehandlung der Prologbefehl (true,true)==true `false` zurückgeben, obwohl ausgewertet dieser Vergleich hätte wahr sein müssen. Die Rückgabe von Prolog kommt jedoch von daher zu Stande, dass Prolog die Terme vor der Prüfung auf Gleichheit nicht auswertet, sondern nur (ausgenommen die Klammern) Zeichen für Zeichen vergleicht. Von daher ist folgende Vorbehandlung für jeweils den linken und rechten Ausdruck vorgesehen: $[\![a]\!]$->VARLINKS=true; VARLINKS=false, analog dazu für den rechten Term. Nun entsteht jedoch ein weiteres Problem, wenn der Berechnungsalgorithmus von Prolog auf eine ungebundene Variable stößt. Dies kann sein, wenn wir den Wert für eine Variable nicht kennen. Jedoch weiß Prolog in diesem Fall nicht, was er tun soll, da eine ungebundene Variable weder an sich wahr, noch falsch ist und wirft einen Fehler (`ERROR: '<meta-call>'/1: Arguments are not sufficiently instantiated`). Um diesen abzufangen, bietet Prolog das catch-Prädikat an, welches das `try... catch ...`-Konstrukt aus vielen bekannten Programmiersprachen wie Java simuliert. Da wir nun in einem solchen Fehlerfall nichts über den Wert des Termes sagen können, binden wir in dem Fall VARLINKS beziehungsweise VARRECHTS an trueORfalse_<variablenID>. Nur im Falle eines Vergleiches zweier Variablen können wir mit diesem Hilfskonstrukt im Falle einer Wissenslücke den Wert der Gleichung positiv bestimmen.

***get*-Fall** Im *get*-Fall lautet die Übersetzung zur Überprüfung final: catch ($[\![a]\!]$) -> VARLINKS=true; VARLINKS=false, _, VARLINKS = trueORfalse_<variablenID>), catch ($[\![b]\!]$)->VARRECHTS = true; VARRECHTS = false, _, VARRECHTS = trueORfalse_<variablenID>), VARLINKS == VARRECHTS.

***set*-Fall** Im *set*-Fall lautet die Übersetzung zur Unifikation final: catch ($[\![a]\!]$) -> VARLINKS=true; VARLINKS=false, _, VARLINKS = trueORfalse_<variablenID>), catch ($[\![b]\!]$)->VARRECHTS = true; VARRECHTS = false, _, VARRECHTS = trueORfalse_<variablenID>), VARLINKS = VARRECHTS.
↪ *Bemerkung:* Um nicht selbst eine nicht erfolgreiche Unifikation durch den Versuch einer Überschreibung eines Wertes einer Variable zu erzeugen, sind im Tool VARLINKS und VARRECHTS so kodiert und indexiert, dass jede Variable nur maximal einmal unifiziert wird.

Integers

Die Unterstützung für ganzzahlige Zahlen (Integers) bietet SeSAME ebenfalls an. Doch Zahlen in und mit Prolog sind nicht trivial. Im folgenden möchte ich aufzeigen, dass Prolog für die Verifikation mit Zahlen keine zu favorisierende Programmiersprache ist. Des Weiteren möchte ich eine Idee geben, wie man diese Probleme mit viel Programmieraufwand doch lösen kann:

Zahlen schalten den Gebrauch von arithmetischen Ausdrücken frei. Alle arithmetischen Operatoren, die SeSAME kennt, kennt auch Prolog, so, dass diese direkt übersetzt werden können. Jedoch wertet Prolog mit den bisher gebrauchten Vergleichs- und Unifikationsprädikaten arithmetische Ausdrücke nicht aus. So wird bei X=1+1 die Variable nicht an die Zahl 2, sondern an die drei Zeichen 1+1 gebunden. Die Auswertung arithmetischer Ausdrücke muss mit is (für Unifikation) beziehungsweise =:= und >, >=, =<, < für Vergleiche angefordert werden. Somit muss bei der Übersetzung entschieden werden, ob es sich um einen arithmetischen Ausdruck handelt oder nicht. Denn wendet man die arithmetischen Operatoren von Prolog auf nichtarithmetische Ausdrücke an, wirft Prolog einen Fehler (beispielsweise `ERROR: is/2: Arithmetic: ‘const/0’ is not a function`). Wenn man diese Unterscheidung nicht bereits bei der Übersetzung treffen möchte, kann man dieses Verhalten von Prolog mit catch($[\![a]\!]$ is $[\![b]\!]$, _, $[\![a]\!]$ = $[\![b]\!]$) entschärfen (Übersetzung für $[\![a]\!]$ = $[\![b]\!]$).

Zahlen stellen eine geordnete Menge dar. In der Prädikatenlogik schaltet das die Operatoren $>, \geq, \leq, <$ frei. Das Problem: wenn man nun solches Wissen abbilden möchte (*set*-Fall), so geht dies nicht auf direktem Wege, da Prolog nur konkrete Werte an eine Variable binden kann, nicht aber Wertebereiche. Der *set*-Fall hingegen ist einfach, denn hier bietet auch Prolog die selben Operatoren >, >=, =<, < an.

- Eine Lösung zum *set*-Problem ist das hierzu vordefinierte Prädikat between(+Low, +High, ?Value), welches eine Untergrenze und eine Obergrenze sowie eine Variable erwartet. Ist diese ungebunden, so wird der gesamte Zahlenbereich an die Variable gebunden. Andernfalls wird überprüft, ob der Wert der Variable in dem aufgespannten Zahlenbereich liegt [19]. Das Problem an dieser Lösung ist die Laufzeit. Muss man das Wissen mit einer offenen Ober- oder Untergrenze modellieren, so hat man einen extrem großen Zahlenbereich (im Beispiel $X < 0$, wenn es sich um ein 32-bit-Integer handelt: $\frac{2^{32}}{2} - 1 = 2^{31} - 1 = 2.147.483.647$ verschiedene Möglichkeiten, welchen Wert x tatsächlich haben kann). Da anzunehmen ist, dass hinter dem vordefinierten Prädikat eine Disjunktion steht, bei der in jedem Glied X an einen möglichen Wert gebunden wird, gibt es 2.147.483.647 Unifikationen, die Prolog im schlechtesten Fall alle durchprobieren muss. Dass dies von außen gesehen zu einem Zustand ohne kurz- oder mittelfristige Rückmeldung führt, ist relativ offensichtlich. Diese Lösung ist also nur praktikabel, wenn man als Datentyp nicht Integer, sondern beispielsweise Short wählt (16Bit $= 2^{16} = 65.535$ verschiedene Zahlen). Mit dem Datentyp Short hätte in unserem Beispiel $X < 0$ Prolog nur noch $\frac{2^{16}}{2} - 1 = 2^{15} - 1 = 32.767$ verschiedene Möglichkeiten zu überprüfen. Dies führt an einem modernen PC mit durchschnittlichen handelsüblichen Prozessor zu keiner spürbaren Verzögerung. Erst ab 2^{20} Möglichkeiten wird Worst-Case eine Berechnungszeit für den Benutzer spürbar (jedoch noch unter einer Sekunde), ab etwa 2^{24} Möglichkeiten steigert sich die Wartezeit auf mehrere Sekunden. Bei etwa 2^{28} Möglichkeiten ist im Worst-Case die Schmerzgrenze eines handelsüblichen PCs von über einer halben Minute Berechnungszeit erreicht.
  Die dazugehörige Lösung zum *get*-Fall ist nun hier leider nicht mehr trivial, der

Grund ist: mit $X < 0$ im *set*-Fall sagen wir aus: es existiert ein Wert aus der Menge $\{GLOBAL\_MIN\_VALUE, -1\}$, der X zugeordnet ist $(\exists \in \{GLOBAL\_MIN\_VALUE, -1\} : X = x)$. Welchen genau, wissen wir nicht. Mit dem vordefinierten Prädikat between sagen wir aber aus: X wird nacheinander an alle Werte aus dem angegebenen Bereich gebunden [19]- somit ist dies gleichzusetzen mit $\forall \in \{GLOBAL\_MIN\_VALUE, -1\} : X = x$. So würde nach dem *set* von $X < 0$ die Abfrage, ob X denn -999 ist, mit true beantwortet werden, obwohl an dieser Stelle ein false die richtige Antwort sein müsste, weil wir nicht garantieren können, dass $X = -999$ gilt. An dieser Stelle muss noch nach einer Lösung geforscht werden. Versuche mit einem negierten Allquantor not(forall(...)) [19], der somit die Bedeutung eines Existenzquantoren trägt, schlugen in der Forschung im speziellen fehl.

- Eine andere Lösung zum *set*-Problem ist das Aufsplitten aller Prologvariablen in zwei Ausprägungen: {VAR}_min und {VAR}_max. Man unifiziert also nicht mehr eine Variable mit einem Wert, sondern mit deren Unter- und Obergrenze. So würde beispielsweise $X < 0 \Leftrightarrow X \leq -1$) mit X_max is -1 übersetzt werden. Die Untergrenze lassen wir hierbei offen, da diese nicht explizit in diesem Beispiel definiert wurde. Im Falle einer Gleichheit wird so übersetzt, dass $\{VAR\}_min = \{VAR\}_max = \{$zugewiesenerWert$\}$ gilt
  Im *get*-Fall werden die gesetzten Variablen abgefragt:
  - Abfrage `X >` $[\![wert]\!]$: var({VAR}_max), X > {VAR}_min (eine Obergrenze für die Variable wurde nicht gesetzt (gibt es also nicht) und die Untergrenze ist unter der Abgefragten)
  - Abfrage `X >=` $[\![wert]\!]$: var({VAR}_max), X >= {VAR}_min (eine Obergrenze für die Variable wurde nicht gesetzt (gibt es also nicht) und die Untergrenze ist unter oder gleich der Abgefragten)
  - Abfrage `X =` $[\![wert]\!]$: X =:= {VAR}_max, X =:= {VAR}_min (die Obergrenze gleicht der Untergrenze (die Variable hat einen ganz spezifischen Wert) und dieser spezifische Wert gleicht dem Abgefragten)
  - Abfrage `X <=` $[\![wert]\!]$: X =< {VAR}_max, var({VAR}_min) (eine Untergrenze für die Variable wurde nicht gesetzt (gibt es also nicht) und die Obergrenze ist über oder gleich der Abgefragten)
  - Abfrage `X <` $[\![wert]\!]$: X < {VAR}_max, var({VAR}_min) (eine Untergrenze für die Variable wurde nicht gesetzt (gibt es also nicht) und die Obergrenze ist unter der Abgefragten)

  Eine Negation im *set*-Fall wie $x \neq 0$ könnte man in Prolog beispielsweise mit einer Bindung der Untergrenzenvariable an die Konstante *neg* und der Bindung der Obergrenzenvariable an den negierten Wert übersetzen. Auch andere Übersetzungsmuster wären denkbar, wichtig ist nur, dass man diese konsistent hält, vor allen Dingen zwischen dem *set* und *get*-Fall.

- den vorherigen Ansatz könnte man auch anlog statt der zwei Variablen für Unter- und Obergrenze einer Service-Kompositions-Kontrollflussvariablen mit einer Liste in Prolog gestalten. So würde eine beliebige Service-Kompositions-Kontrollflussvariable `X` mit X=[{Untergrenze}, {Obergrenze}] gehandhabt werden. Bei Konstanten und Strings wäre das erste Element der Liste gleich dem zweiten

Die größte Herausforderung mit Zahlen ist der Umgang mit Unbekannten. Das Problem: wenn eine Variable in einem arithmetischen Ausdruck in Prolog vorkommt, und diese Variable noch nicht gebunden ist, wirft Prolog den Fehler `ERROR: [Operand]/2: Arguments are not sufficiently instantiated` (Ausnahme: bei is, wenn der rechte Ausdruck eine ungebundene Variable ist, wird diese ohne Fehler mit dem linken Ausdruck unifiziert, sofern dort keine ungebundenen Variablen sind). Dieses Verhalten ist jedoch sehr problematisch, da wir in Service-Kompositionen häufig auf Variablen stoßen, denen wir keinen festen Wert zuordnen können. Leider führt in diesem Moment auch das Fangen der Fehlermeldung und umleiten in nicht arithmetische Operatoren zu keinem Erfolg. Folgendes Gegenbeispiel sei gegeben:

Maximum: Gegenbeispiel- verdeutlicht die Problematik des Gebrauches mit Zahlen

```
1 if int1 < int2 then
2   int max = 0+int2
3 else
4   int max = int1+0
```

Als Eingang werden beliebige zwei Zahlen erwartet, als Ausgang das Maximum der beiden Zahlen. Offensichtlich ist dazu die Service-Komposition *Maximum* valide. Eine Prologübersetzung schlägt hier allerdings ohne Erfolg, da int1 und int2 zwei Unbekannte sind. Wenn nun in Zeile 2 max gesetzt wird, wirft Prolog eine Fehlermeldung und max bekommt die Zeichenkette 0+int2 zugewiesen- unausgewertet ist diese Zeichenkette nicht der Wert des Maximums der beiden Variablen. Um das Problem zu lösen, könnte man in verschiedene Richtungen gehen

- Ein möglicher Lösungsweg ist zu überprüfen, ob alle auftauchenden Variablen gebunden sind, bevor arithmetische Ausdrücke benutzt werden. Wenn nicht, bindet man diese an Variablen eindeutigen numerischen Hashwert (var(@Term) [19] um zu überprüfen, ob Variablen nun ungebunden sind). So würde die Anweisung `max = 0+int2` mit INT2 is {hash}, MAX is 0+INT2 übersetzt werden. Problem an dieser Methode ist jedoch, dass man zum einen einen Hashwert außerhalb des Bereiches wählen muss, in dem Werte auch natürlich vorkommen können und zum anderen nimmt man sich so offene Entscheidungen vorweg. So wie bei Zeile 1 des Maximum-Beispieles: INT1 und INT2 müssten zu dieser Abfrage ihren Hashwert zugewiesen bekommen, damit Prolog den Ausdruck INT1 < INT2 auswerten kann. Durch die Hashbindung ist jedoch nun determiniert, ob dieser Ausdruck wahr oder falsch ist, obwohl wir dies nicht wissen können und so den if- als auch den else-Teil der Verzweigung betrachten müssten. Leider ist im Gegensatz zum Unifikationsprädikat = eine nachträgliche Variablenbindung ohne Fehler bei den arithmetischen

Operatoren in Prolog nicht möglich. So wirft X=:=0; X is 0 als auch X=:=0; X = 0 einen Fehler, während X==0; X=0 erfolgreich unifiziert, da durch die Disjunktion beide Ausdrücke ausgewertet werden, 0 erfolgreich an X gebunden wird und X somit gleich 0 ist. Um nun doch noch eine Lösung für das Problem mit diesem Lösungsansatz zu geben, müsste bei der Übersetzung entschieden werden, ob mit der Hashbelegung gegebenfalls der Kontrollfluss manipuliert werden könnte und wenn ja, müssten noch durch geschickte Ersetzung bestimmter Terme Fehlerquellen eliminiert werden. Dies ist jedoch sehr komplex und gleicht dann eher einer Verifizierung vom Übersetzungstool als nur der Übersetzung selbst.

- Ein zweiter Lösungsgedanke wäre, die arithmetischen Prädikate (jedoch mit den gewünschten Eigenschaften) in Prolog nachzubauen. So könnte das is-Prädikat durch eine Regel ersetzt werden, die die Zuweisung durchscannt, auswertbare Teile in einer Konstante zusammenfasst und Variablen hinter diese Konstante schiebt. Dieser Lösungsweg ist ebenfalls sehr komplex und aufwändig zu implementieren.

Fazit: Insgesamt kann man sehen, dass mit viel Aufwand das Verifizieren von Arithmetik mit Prolog prinzipiell möglich ist, auch, wenn manche Probleme erst im Detail oder Spezialfällen auffallen. Jedoch ist dieser Bereich definitiv keine Stärke von Prolog und es wäre gut denkbar, dass es hier andere Sprachen gibt, die leichtere Lösungen in der Arithmetik anbieten.

4.3.2 Prädikate und Funktionen übersetzen

Prädikate sind Sonderformen von Funktionen, die ihre Eingabe auf `true` oder `false` abbilden. Aus diesem Grund schließe ich, wenn ich über Funktionen schreibe, im Folgenden Prädikate mit ein.
Während die einfache Variablendeklaration aufgrund der Unifikation von Prolog eingängig ist, gestalten sich Funktionen als schwieriger. Es wäre falsch, einfach den Funktionsnamen mit einem Wert gleich zu setzen, da im Allgemeinen Funktionen gegebene Eingabeparameter auf verschiedene Werte abbilden. Diese Eingabeparameter muss ich somit berücksichtigen. Aus diesem Grund habe ich mich für Listen in Prolog entschieden, um Funktionen zu übersetzen. Dabei gibt es eine Liste für alle bekannten Funktionsdeklarationen. Jedes Listenelement beschreibt eine Funktionsdeklaration $f(X1, ..., Xn) = Y$ und ist wiederum eine Liste, die wie folgt implementiert ist: $[f, X1, ..., Xn, Y]$.

***get*-Fall** Wird eine Funktion abgefragt, wird die Liste durchsucht. Ein selbst implementierter Prologalgorithmus sucht dann mittels Backtracking nach einem passenden Eintrag. Verläuft die Suche erfolglos, existiert zwar der Rückgabewert der Funktion, ist jedoch nicht bekannt und somit antwortet Prolog mit `false`.
Dieser selbst implementierte Algorithmus mit Prologregeln ohne Zahlenunterstützung ist wie folgt programmiert:

Prologregeln zur Validierung der Funktionsliste (ohne Hilfmethoden abgebildet)

```
1  %Aufruf des Algorithmus ueber ph_isFunctionSet bei Funktionen
       ohne Eigenschaften mit
2  % - der Liste aller Funktionen
3  % - eine Liste der angefragten Funktion (moechte man den
       Rueckgabewert (letztes Element der Liste) der Funktion nicht
        mit abgleichen, so schreibt man hierfuer '_')
4  % - eine Variable fuer den Rueckgabewert einer bereits
       bekannten passenden Funktion
5  %-------------------------------------------------------------%
6  %Durchsuchen der Liste mit allen Funktionen
7  ph_isFunctionSet([FLelement|FUNCTIONLISTrest], FUNCTION, OUT)
       :- ph_isValidFunction(FLelement, FUNCTION, INNEROUT)->
8    (OUT=INNEROUT;true);
9    ph_isFunctionSet(FUNCTIONLISTrest, FUNCTION, OUT).
10 ph_isFunctionSet([], _, _) :- false.
11 %Abgleichen eines Listenelementes (Funktion - FistH/FistT) mit
       der Abfrage (FsollH/FsollT) [man bemerke das
       Negationshandling mit 'neg/']
12 ph_isValidFunction([FistELEMENT|FistREST], [FsollELEMENT|
       FsollREST], OUT) :-
13 ((not(length(FistREST, 0)), term_string(FistELEMENT,FistHString
       ), split_string(FistHString, "/", "", FistHStringList),
       member("neg", FistHStringList)) ->
14   ((term_string(FsollELEMENT,FsollHString), split_string(
         FsollHString, "/", "", FsollHStringList), member("neg",
         FsollHStringList)) -> FistELEMENT==FsollELEMENT;
         FistELEMENT\=neg/FsollELEMENT);
15   ((length(FistREST,0), length(FsollREST,0)) ->
16     (FistELEMENT=FsollELEMENT, OUT=FistELEMENT);
17     (FistELEMENT==FsollELEMENT))),
18 ph_isValidFunction(FistREST, FsollREST, OUT).
19 %Anzahl der Parameter muessen ebenfalls uebereinstimmen!
20 ph_isValidFunction([],Fsoll, _) :- Fsoll==[] -> true; false.
21 ph_isValidFunction(Fist,[], _) :- Fist==[] -> true; false.
22 %-------------------------------------------------------------%
23 %Aufruf bei Funktionseigenschaften- siehe  Definitionen
24 % (length_3: Funktion, length_4: Praedikat)
25 ph_isFunctionSet_reflexive(FUNCLIST, FUNC, OUT) :- nth0(1, FUNC
       , P_left), nth0(2, FUNC, P_right), (last(FUNC, LAST), ((
       length(FUNC, 4), P_left\=P_right) -> ph_isFunctionSet(
       FUNCLIST, FUNC, OUT);
26   (var(LAST) -> ((P_left==P_right)-> OUT=true; OUT=false);
```

```
27    (LAST==false -> (P_left\=P_right, OUT=false);
28    (P_left==P_right, OUT=true))))).
29
30  ph_isFunctionSet_irreflexive(FUNCLIST, FUNC, OUT) :- nth0(1,
      FUNC, P_left), nth0(2, FUNC, P_right), last(FUNC, LAST),
31    (var(LAST) -> ((P_left==P_right) -> OUT=false;
        ph_isFunctionSet(FUNCLIST, FUNC, OUT));
32    (LAST==false -> (P_left==P_right, OUT=false);
33    (P_left\=P_right, ph_isFunctionSet(FUNCLIST, FUNC, OUT)))).
34
35  ph_isFunctionSet_symmetric(FUNCLIST, FUNC, OUT) :-
      ph_isFunctionSet(FUNCLIST, FUNC, OUT);
36  (swap_nth0_and_tth0_element(FUNC, 1, 2, SWAP), ph_isFunctionSet
      (FUNCLIST, SWAP, OUT)).
37
38  ph_isFunctionSet_asymmetric(FUNCLIST, FUNC, OUT) :- nth0(1,
      FUNC, P_left), nth0(2, FUNC, P_right),
      swap_nth0_and_tth0_element(FUNC, 1, 2, SWAP), last(FUNC,
      LAST), (var(LAST)->AskForOutput=true;true),
39    (ph_isFunctionSet(FUNCLIST, FUNC, OUT)->
40      (AskForOutput==true; LAST==false; not((((length(FUNC,3))->
          ph_isFunctionSet(FUNCLIST, SWAP, OUT); (LAST==true,
          ph_isFunctionSet(FUNCLIST, SWAP, OUT)))))));
41    ((AskForOutput==true;LAST==false) -> (OUT=false,(P_left==
        P_right;(swap_return_ifpredicate(SWAP,PSWAP),
        ph_isFunctionSet(FUNCLIST, PSWAP, _))))))).
42
43  ph_isFunctionSet_antisymmetric(FUNCLIST, FUNC, OUT) :- nth0(1,
      FUNC, P_left), nth0(2, FUNC, P_right),
      swap_nth0_and_tth0_element(FUNC, 1, 2, SWAP), last(FUNC,
      LAST), (var(LAST)->AskForOutput=true;true),
44    (ph_isFunctionSet(FUNCLIST, FUNC, OUT)->
45      (AskForOutput==true; LAST==false; not((P_left\=P_right,((
          length(FUNC,3))-> ph_isFunctionSet(FUNCLIST, SWAP, OUT)
          ;(LAST==true, ph_isFunctionSet(FUNCLIST, SWAP, OUT)))))))
          ;
46    (((AskForOutput==true;LAST==false), P_left\=P_right) -> (
        swap_return_ifpredicate(SWAP,PSWAP), ph_isFunctionSet(
        FUNCLIST, PSWAP, _), OUT=false))).
```

Diese Regeln decken alle Fälle ab. So ist es auch möglich, den Ausgangs eines Prädikates zu erfragen, wenn man die Parameter kennt und dieser Ausgang definiert wurde oder durch Prädikateneigenschaften hergeleitet werden kann. Bei Funktionen allgemein muss die Herleitung über Funktionseigenschaften noch ausgearbeitet werden.

***set*-Fall** Funktionen haben in der Prädikatenlogik die definierte Eigenschaft, dass sie rechtseindeutig sind. Das heißt, bei gleichen Parametern dürfen nicht unterschiedliche Ausgaben von der Funktion zurückgeliefert werden. Diese Eigenschaft muss ich beispielsweise bei der Verifizierung einer Verzweigungsbedingung in meiner Übersetzung ebenfalls sicher stellen. Deshalb muss ich zuerst prüfen, ob das, was gesetzt werden soll, mit den Namen und Parametern schon in der Funktionsliste verzeichnet ist. Wenn ja, muss ich den Ausgang damit unifizieren: ist dieser identisch zu dem, was gesetzt werden soll, handelt es sich einfach nur um redundantes Wissen und die dieser Fall kann bei einem true verbleiben. Ist dem nicht so und es liegt bei dem, was gesetzt werden soll somit ein Bruch mit der Rechtseindeutigkeit vor, muss Prolog dies mit einem false auswerten und das Setzen abbrechen.
Ist die Funktion mit den Parametern, die gesetzt werden soll, nicht in der Funktionsliste und Wissensbasis vorhanden, kann ich allgemein von Rechtseindeutigkeit ausgehen und die Funktionsliste ergänzen.

4.3.3 Weitere Spezialbehandlungen

Übersetzung von Mengen

Prolog kennt die Datenstruktur der Liste und eine große Funktionsbibliothek (bsp.: library(lists): List Manipulation) um diese Datenstruktur herum [19]. Hier übersteigt der Funktionsumfang von Prolog abermals den von SeSAME. Darüber hinaus können Operationen auf Mengen direkt mit einer ausgewählten Funktion in Prolog übersetzt werden. Prinzipiell sind Listen in Prolog aufgrund ihrer internen Implementierung nicht Mengen im eigentlichen Sinne, da Listen Duplikate mit Auswirkung beinhalten können. Während in Prolog $[a]$ und $[a, a]$ zwei nicht gleiche Listen sind, wären die dazugehörige Menge $\{a\}$ und $\{a, a\}$ gleich. Die zweite Menge hätte nur redundante Informationen. Dieses Problem lässt sich jedoch mit den vordefinierten Funktionen is_set und list_to_set bzw. mit einer eigens programmierten Regel lösen.

Übersetzung von Quantoren

In der Prädikatenlogik mit reduziertem Umfang habe ich die Quantoren außen vor gelassen. Ein Existenzquantor ($\exists$) und ein Allquantor ($\forall$) sind jedoch in der allgemein gebräuchlichen Prädikatenlogik definiert [9], so auch bei SeSAME. Diese können auch in der Vor- und Nachbedingung einer Service-Komposition mit einprogrammiert werden. Ein Übersetzungsansatz ist in Tabelle 4.2 definiert (gilt nur, wenn Zahlen nicht unterstützt werden).

	Vorbedingung der Service-Komposition (*set*-Fall)	Nachbedingung der Service-Komposition (*get*-Fall)
Existenzquantor	setze direkt nach Aufruf der Service-Kompositions-Regel vor der Nachbedingung in den Bedingungsparameter der forall-Anweisung an Prolog eine Disjunktion aller Variablen der Menge, die die Ungebundenheit jeder Variable prüft und unter dieser Voraussetzung die Aussage des Existenzquantors dieser zuweist. Das Verfahren von Prolog unter dem forall-Befehl wird jede Möglichkeit validieren, dass eine bestimmte Variable vom Existenzquantor betroffen ist	Alle Variablen der Menge werden in der Nachbedingung in einer Disjunktion auf die beschriebene Eigenschaft hin untersucht. Erfüllt ein Element die Eigenschaft, ist die gesamte Disjunktion erfüllt [9]
Allquantor	setze direkt nach Aufruf der Service-Kompositions-Regel vor der Nachbedingung in den Bedingungsparameter der forall-Anweisung an Prolog eine Konjunktion aller ungebundenen Variablen der Menge auf den beschriebenen Wert. Prolog wird diese Variablenbindung im Nachhinein einbeziehen	Alle Variablen der Menge werden in einer Konjunktion auf die beschriebene Eigenschaft hin untersucht. Erfüllt ein Element die Eigenschaft nicht, ist die gesamte Konjunktion falsch [9]

Tabelle 4.2: Übersetzungansatz für Quantoren in Vor- und Nachbedingung der Service-Komposition

5 Vorstellung des Prologvalidator-Tools

All das in Kapitel 4 erfasste wurde in einem selbst entwickelten Tool namens Prologvalidator implementiert. Ich werde zunächst in Kapitel 5.1 auf die Architektur des Tools eingehen, in Kapitel 5.2 auf die Benutzung und schlussendlich werde ich in Kapitel 5.3 auf das Beispiel der Bachelorarbeit zurück kommen: hier wird das Bildbearbeitungsprogramm über die bereits programmierte Service-Komposition mittels Prolog verifiziert werden!

5.1 Architektur

Mein Tool Prologvalidator ist nach dem Schema der hierarchischen 3-Schichtenarchitektur aufgebaut. Diese ist eine bewährte Softwarearchitektur, bei der Komponenten einer Schicht nur auf die darunterliegenden Schichten zugreifen dürfen [10].In meinen Fall limitiere ich die Zugriffsmöglichkeit auf die nächst darunterliegende Schicht, am das System. Die drei Schichten der Architektur möchte ich nun nachfolgend an meinem Tool erklären möchte:

- Die Daten[haltungs]schicht verwaltet die gespeicherten Daten einer Applikation [10]. In meinem Falle werden alle erzeugten Prologcodeteile in dieser Ebene gesammelt und strukturiert. Bei der Ausführung von Prolog können diese über die Klasse `RuleCollector` durch einen einzigen Aufruf abgefragt werden. Dabei ist zu erwähnen, dass ich bei der Implementierung den Begriff der Regel (siehe Klasse `Rule`) allgemein auch für Anfragen (besitzen nur einen `RuleBody`) und Fakten (besitzen nur einen `RuleHead`) angewandt habe. Der Regelrumpf wird durch die Klasse `RuleBody` abgebildet. Diese referenziert einen `RuleBodyPart`. Hierhin werden in eine Baumstruktur rekursiv die Daten für einen Prologrumpf von der Controller-Ebene abgelegt. Neben den Speichern verwaltet diese Ebene auch die Daten, dies schließt mit ein:
  - das auf Abfrage Zusammenfügen der Regelteile, so, dass valider Prologcode entsteht
  - das Bereitstellen eines globalen Zählers für ID-Nummern-Gebrauch (dies ist beispielsweise wichtig, um erzeugte Variablen in ihrer Deklaration in Prolog einzigartig zu halten, indem eine ID-Nummer abgehangen wird)- siehe dazu die Klasse `IDGetter` mit `Item`

- Die Logikschicht, auch Geschäftslogikschicht genannt, verarbeitet die Daten (Eingänge) [10]. In unserem Fall geschieht ein Ereignis in der Eingangs-Präsentationsschicht, wenn eine Verifikation mit Prolog angefragt wird. Der `Controller` nimmt die Service-Kompositions-Eingabe und zerlegt diese in die Ontologie, den Kontrollfluss der Service-Komposition, die Vor- und Nachbedingung der Service-Komposition und die Spezifikationen der Services. Alles leitet der `Controller` innerhalb seiner Schicht an die jeweiligen Verarbeiter weiter. So bekommt der `ServiceRuleBuilder` stückweise die Spezifikationen der Services. Den Kontrollfluss erhält der `ServiceCompositionRuleBuilder`, und der `QueryBuilder` empfängt für die Erstellung der entscheidenden Anfrage an Prolog die Vor- und Nachbedingung der Service-Komposition. `QueryBuilder` wird im weiteren Verlauf über die Methode *validate()* einen SWI-Prolog-Prozess erzeugen und ihm alle erzeugten Regeln und die Anfrage geben sowie die Antwort von Prolog darauf hin auswerten. Wenn der Prozess Fehler erzeugt beziehungsweise zu lange für die Berechnung einer Antwort braucht, sind Abfangungs- und Schutzmechanismen eingebaut. Zwei entscheidende Klassen (inklusive ihrer Hilfsklassen) sind noch nicht erklärt in der Controller-Ebene:
  - `TermToProlog`: diese Klasse nimmt einen beliebigen prädikatenlogischen Term mit beschränktem Umfang und wandelt diesen rekursiv in Prologregelteile um, die in der Model-Ebene gespeichert werden. Dabei benutzt `TermToProlog` rekursive Unterroutinen, wenn beispielsweise `TermToProlog` auf ein Objekt mit Parametern stößt- diese Parameter werden in die Unterroutine gegeben, die den Wert der Parameter wiederum Variablen zuweist, die dann wiederum die Hauptroutine nutzt.
  - `FlowToProlog`: diese Klasse nimmt einen beliebigen Kontrollfluss und wandelt diesen dann sequentiell (Codeblock nach Codeblock) sowie rekursiv (Verschachtelungen (Verzweigungen, Schleifen)) in Prologregelteile um, die in der Model-Ebene gespeichert werden. Trifft `FlowToProlog` dabei auf einen prädikatenlogischen Term, lässt sich dies die Klasse von `TermToProlog` übersetzen

- Die Präsentationsschicht präsentiert die Benutzerschnittstelle [10]. Im Falle des Tools fällt diese Schicht sehr dünn aus, da, abgesehen von der Konsole, Prologvalidator nicht direkt an eine GUI, also einer grafischen Benutzeroberfläche, angeschlossen ist. Jedoch empfängt die `In`-Klasse auf dieser Ebene die Verifizierungsanfrage und leitet diese an die Controller-Ebene weiter. Ebenfalls erhält `In` ein Log-Objekt. Dieses wird die Klasse am Ende der Ausführung nutzen, um eine Loggingdatei zu erstellen.

Die Trennung von Datenhaltung (Datenschicht) und Datengenerierung (Logikschicht) halte ich für die Übersicht sehr zuträglich. Deshalb habe ich mich für die 3-Schichten-Architektur entschieden.
Die Abbildung 5.1 gibt einen Überblick über die Architektur. Die Datenlinien repräsentieren wichtige Datenflüsse. Dieser Überblick soll lediglich eine Idee von der Struktur

und dem Verarbeitungsfluss des Tools geben. Er beinhaltet jedoch weder alle Klassen noch Methoden der Datenflüsse.

5.2 Einbettung und Anwendung des Tools

Dieses Tool ist in der Entwicklungsumgebung SeSAME als Plugin eingebettet und als solches über SeSAME zu benutzen. So kann später die Prologverifikation über das Kontextmenü auf einem B3-Quelltext der Service-Komposition in einer SeSAME-Eclipse-Instanz einfach gestartet werden [2]. Ohne Javaprogrammierung ist diese Verifikation jedoch durch Einschränkungen von SeSAME limitiert. Folgende Limitierungen für Service-Kompositionen gelten nicht für die Übersetzungsmöglichkeit mit meinem Prologvalidator, sind aber nach aktuellem Stand von SeSAME gesetzt:

1. Beschränkung: Der Gleichungsoperator `=` ist nur für arithmetische Gleichungen zugelassen. Der Zuweisungsoperator `:=` ist zusätzlich für das Abfangen der Rückgabewerte von Service-Aufrufen freigegeben. Um boolesche Variablen zu definieren, genügt für das Wahr-Setzen das bloße Nennen der Variable und für das Falsch-Setzen `neg <variablenname>`. Analog dazu kann der Rückgabewert von Prädikaten beschrieben werden.

2. Beschränkung: es sind keine Strings erlaubt

3. Beschränkung: es sind keine Funktionen erlaubt (außer die Spezialform einer Funktion mit dem booleschen Rückgabewert: das Prädikat)

4. Beschränkung: Service-Aufrufe dürfen nur in Verbindung mit einer Gleichung/Zuweisung beschrieben werden, damit der genau eine Ausgang abgefangen werden kann. Services ohne Ausgang sind nicht erlaubt.

Des Weiteren sind folgende Beschränkungen durch den Prologvalidator vorhanden:

1. Beschränkung: es sind keine arithmetischen Ausdrücke erlaubt

2. Beschränkung: es sind nur prädikatenlogische Formeln mit reduziertem Umfang erlaubt

3. Beschränkung: Ontologieregeln werden nur beschränkt verarbeitet (nur Eigenschaften über Funktionen/ Prädikate, davon maximal eine pro Funktionen/ Prädikat außer die transitive und totale Eigenschaft)

4. Beschränkung: selbst definierte Mengen sind nicht erlaubt

Des Weiteren muss SWI-Prolog lokal auf den Rechner im Systemprogrammordner installiert sein. SWI-Prolog kann man sich kostenlos unter `http://www.swi-prolog.org/Download.html` herunterladen [19].

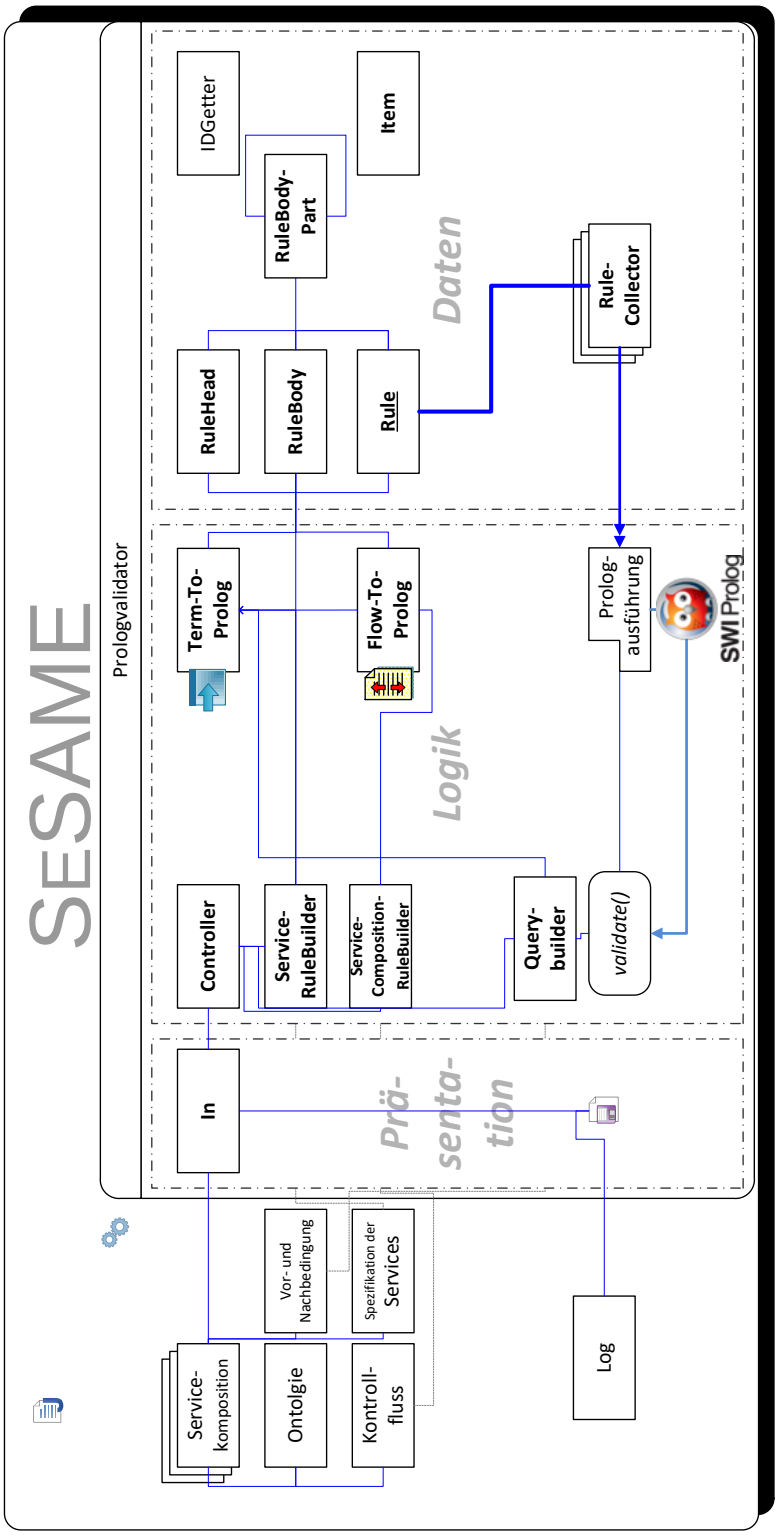


Abbildung 5.1: Architektur und Einbettung des Tools der Bachelorarbeit: Prologvalidator

Eine weitere Möglichkeit, mit Prologvalidator zu verifizieren, ist das direkte Interagieren mit dem zugehörigen Plugin `de.upb.crc901.sse.analyze.exporter.prolog` in Java unter Benutzung der `export`-Schnittstelle (implementiert `de.upb.crc901.sse.analyze.datamodel.IDatamodelRoot`). Der Methode muss eine Loggingklasse sowie ein Datenmodell gegeben werden. Unabhängig, welche Loggingklasse im Konkreten gewählt wird, wird am Ende der Ausführung von Prologvalidator eine Loggingdatei im *temp*-Verzeichnis angelegt.

5.3 Veranschaulichung am Beispiel

Nun folgt die Verifikation des konkreten Fallbeispiels der Bachelorarbeit: Verifikation der Bildbearbeitungssoftware. Ziel ist es dabei, einen Einblick in den Verarbeitungsprozesses meines Tools zu geben. Es sei vorweg gesagt, dass Variablen zur Verständlichkeit teils konsistent in diesem Beispiel umbenannt sind und bei einer vergleichenden Ausführung mit Prologvalidator in dieser Hinsicht Unterschiede auftreten können. Ferner sei angemerkt, dass dieses Kapitel nicht erklärt, warum diese Ausgaben so gestaltet sind, da dies in der Theorie bereits in Kapitel 4 und entsprechend der Softwarearchitektur in Kapitel 5.2 getan wurde.
Folgende Aufgaben, repräsentiert an dem Bildbearbeitungsbeispiel, übernimmt der Prologvalidator:

1. **Servicekomposition mit Hilfe von SeSAME in SSA-Form bringen**
   Das Ergebnis hiervon ist:

Kontrollfluss der Service-Kompositon zur Bildverarbeitungssoftware

```
1  Load Ontology "./src/imageFilter.owl";
2
3  Service Definitions:
4  service scaleDown { SDIN:Image; SDOUT:Image; neg
       lessThan5MB(SDIN); lessThan5MB(SDOUT) and isVariantOf
       (SDOUT, SDIN) };
5  service applyFilter { AFIN:Image; AFOUT:Image;
       lessThan5MB(AFIN); isVariantOf(AFOUT, AFIN) and
       filterApplied(AFOUT) and lessThan5MB(AFOUT)};
6
7  Service Composition:
8  Name: filterAndScale;
9  Input: SCIN : Image;
10 Output: SCOUT : Image;
11 //Vorbedingung
12 Assume true;
13
14 //Kontrollfluss:
15 if neg lessThan5MB(SCIN) then
```

```
16      Image SCOUT0 := scaleDown(SCIN);
17      SCOUT := applyFilter(SCOUT0);
18    else
19      Image SCOUT := applyFilter(SCIN);
20    fi
21
22    //Nachbedingung
23    Assert filterApplied(SCOUT) and lessThan5MB(SCOUT)
```

2. **Anfrage an Prolog zur Verifikation (aus Vor- und Nachbedingung der Service-Komposition) bilden**
   In unserem Beispiel wird hierbei auf die Funktionsliste zurückgegriffen, die generell als globale Variable registriert wird. Diese muss initial erst einmal gesetzt werden, dies geschieht am Anfang der Abfrage. Der erzeugte Prologcode ist: ?-forall((b_setval(functionlist, []), filterandscale(SCIN, SCOUT)), ((b_getval(functionlist, F)), (I11=SCOUT, O1=true, ph_isFunctionSet(F,[filterapplied, I11, O1], _)), (b_getval(functionlist, F), I21=SCOUT, O2=true, ph_isFunctionSet(F, [lessthan5mb, I21, O2], _)))).[1]

3. **Prologregel zur Service-Komposition erstellen**
   Die Regel bildet auf dem Kontrollfluss der Service-Komposition ab. In unserem Beispiel lautet die Regel: filterandscale(SCIN, SCOUT) :- true->(b_getval(functionlist, BACKSAVEfunctionlist), (b_setval(functionlist, BACKSAVEfunctionlist), (b_getval(functionlist, F0), I11=SCIN, O11=false, (ph_isFunctionSet(F0, [lessthan5mb, I11, _], O100) -> ((O100==O11) -> true;false); (append(F0, [[lessthan5mb, I11, O11]], Fset0), b_setval(functionlist, Fset0)))), (SDIN=SCIN, scaledown(SDIN, Osd), Osd= SCOUT0, AFIN=SCOUT0), (applyfilter(AFIN, Oaf), Oaf=SCOUT))); (b_setval(functionlist, BACKSAVEfunctionlist),b_getval(functionlist, F1), (b_getval(functionlist, F1), I21=SCIN, O21=true, (ph_isFunctionSet(F1, [lessthan5mb, I21, _], O101) -> ((O101==O21) -> true;false); (append(F1, [[lessthan5mb, I21, O21]], Fset1), b_setval(functionlist, Fset1)))), (AFIN=SCIN, applyfilter(AFIN, Oaf), Oaf=SCOUT)); false.

4. **Prologregeln zu den einzelnen Services erstellen**
   Dazu geht Prologvalidator jeweils einmal die Spezifikation der Services durch:

   - Regel für Service `scaledown`: scaledown(SDIN, SDOUT) :- (b_getval(functionlist, F), I01=SDIN, O01=false, ph_isFunctionSet(F, [lessthan5mb, I01, O01], _))->((b_getval(functionlist, F0), I11=SDOUT, O11=true, (ph_isFunctionSet(F0, [lessthan5mb, I11, _], O100)->((O100==O11)->true;false); (append(F0, [[lessthan5mb, I11, O11]], Fset0), b_setval(functionlist, Fset0)))), (b_getval(functionlist, F1), I21=SDOUT, I22=SDIN, O21=true,

[1] Die Überprüfung von `isVariantOf` erfolgt an dieser Stelle nicht, da dieses Prädikat auch transitiv ist- die Übersetzung dieser Eigenschaft würde jedoch an dieser Stelle den Rahmen sprengen

(ph_isFunctionSet(F1, [isvariantof, I21, I22, _], O101)->((O101==O21)->true;false);
(append(F1, [[isvariantof, I21, I22, O21]], Fset1), b_setval(functionlist, Fset1))));
true.

- Regel für Service `applyfilter`: applyfilter(AFIN, AFOUT) :- (b_getval(functionlist, F), I01=AFIN, ph_isFunctionSet
(F, [lessthan5mb, I01, true], _)) -> ((b_getval(functionlist, F0), I11=AFOUT, I12=AFIN, O11=true, (ph_isFunctionSet(F0, [isvariantof, I11, I12, _], O100)->((O100==O11)->true;false);(append(F0, [[isvariantof, I11, I12, O11]], Fset0), b_setval(functionlist, Fset0)))), (b_getval(functionlist, F1), I21= AFOUT, O21=true, (ph_isFunctionSet(F1, [filterapplied, I21, _], O101) -> ((O101==O21) -> true;false); (append(F1, [[filterapplied, I21, O21]], Fset1), b_setval(functionlist, Fset1)))), (b_getval(functionlist, F2), I31=AFOUT, O31=true,
(ph_isFunctionSet(F2, [lessthan5mb, I31, _], O102) -> ((O102==O31) -> true;false); (append(F2, [[lessthan5mb, I31, O31]], Fset2), b_setval(functionlist, Fset2))));
true.

5. **Prologprozess starten/ instruieren**
   Dem Prozess werden alle Regeln gegeben (einschließlich die Regeln für die Durchsuchung der Funktionswissensbasis (Kapitel 4.3.2)) und danach wird die Anfrage gestellt. In unserem Beispiel erhalten wir die Antwort: `true`.

6. **Ergebnis interpretieren**
   Es wird eine hochsprachliche Ausgabe erzeugt, die die Service-Komposition als valide in unserem Beispiel ausweist. Es gibt demnach keinen Eingang, für den die Vorbedingung und unter Ausführung der Service-Komposition die Nachbedingung nicht gilt!

6 Evaluation des Tools und Vergleich mit SMT-Solver

Mein Tool wurde als Plugin in SeSAME entwickelt und ist demnach nicht unabhängig von SeSAME. Da es jedoch im Laufe der Bearbeitung immer wieder zu Problemen und Einschränkungen in SeSAME kam, kann ohne großen Zeiteinsatz keine große vollständige Evaluation erfolgen, die auch auf die Features eingeht, die der Prologvalidator unterstützt, die allerdings SeSAME nicht unterstützt.
Das Ergebnis der Evaluation zeigt, dass Prolog an sich eine gute Alternative zu anderen Ansätzen und Programmiersprachen darstellt. Ist erst einmal das Prologprogramm von meinem Prologvalidator erstellt, so ist die Laufzeit von Prolog extrem gering. So ist beispielsweise das Ausführen der Anfrage des Bildbearbeitungsbeispiels im bereits übersetzten Prologprogramm selbst im Millisekundenbereich nicht messbar. Die Gesamtausführungszeit des Prologvalidators besteht jedoch nicht nur aus der Anfrage in Prolog, sondern auch aus:

- Lesen der Service-Komposition und der Ontologie und Übersetzen in ein B3-Datenmodell (geschieht durch SeSAME außerhalb des Prologvalidators)
- Transformieren des B3-Datenmodells in eines mit SSA (geschieht durch eine Funktion von SeSAME)
- Übersetzung des B3-Datenmodells in eine Prologanfrage und Prologregeln
- Lokalisieren der Prologausführungsdatei, daraus starten eines Prologprozesses und Abgreifen von deren Ein- und Ausgabeströmen
- Auswerten der Antwort von Prolog
- Anlegen und beschreiben von Dateien (für das Prologprogramm/ Logdatei)

In der Gesamtheit dieser Punkte gibt es zur Zeit noch Probleme. Bei zu umfangreichen Aktivitäten auf den Ausgabestrom des Prologprozesses ist es eine Herausforderung, diesen Ausgabestrom noch in geordnetem Maße auszulesen. Obwohl viel Arbeit in die Bandung der Ein- und Ausgabeströme investiert wurde, zeigt sich, dass ein externer Prozess mit Java nicht so gut zusammenarbeiten kann, wie erhofft. Des Weiteren kam es bei komplexen Eingaben in Prolog immer wieder vorkam, dass Prolog an sich schnell die Anfrage abgearbeitet hat, diese Ausgabe aber nicht vom Ausgabestrom von Java abgegriffen werden konnte. Hintergrund: Java muss selbst entscheiden, wann es das Warten auf eine Ausgabe von Prolog bei zu großer Zeitverzögerung abbricht und somit

in letzter Konsequenz der Abbruch des Prologprozesses anordnet. Des Weiteren kommt es durch das Suchen der ausführbaren Prologdatei (in einem Microsoftbetriebssystem: `swipl.exe`) zu merkbaren Verzögerungen. Überraschend war bei den Systemtests die Laufzeit der SSA-Operation. Bei einem tief verschachtelten Kontrollfluss ist genau diese Operation neben den Fehlern des Ausgabestroms, die sich jedoch durch das manuelle Interagieren mit Prolog und der vom Prologvalidator im temporärer Verzeichnis abgelegten Datei mit dem Prologprogramm umgehen lassen, der Limitierer hinsichtlich der maximal sinnvollen Komplexität, die eine Service-Komposition für den Prologvalidator haben sollte. Nehmen wird dazu den Kontrollfluss folgender Service-Komposition:

Kontrollfluss für den Laufzeittest (Rekusionsstufe 1)

```
1 if input then
2   Skip;
3 else
4   Skip;
5 fi
```

Der komplette Durchlauf des Prologvalidators braucht hier unter Verwendung eines normalen Laptops (Intel Pentium CPU 8960 2 x 2.2GHz, 4GB RAM) nur unter einer Sekunde, wobei der Großteil der Zeit die Suche nach der ausführbaren Prologdatei läuft. Eliminiert man diese durch das Setzen des Pfades zu Prolog, ist man bei deutlich unter einer Sekunde, wobei dann der größte Zeitanteil für die SSA-Transformation benötigt wird (188ms). Ersetzt man nun manuell in jeder Rekursionsstufe jedes `Skip;` durch die Verzweigung des Kontrollflusses für den Laufzeittest (Rekursionsstufe eins), so wächst die Zahl der Verzweigungen pro Rekursionsstufe exponentiell. Jedoch wächst die Laufzeit der SSA-Transformation schneller als die Exponentialkurve. Ab einer Rekursionsstufe von vier gibt es den Fehler, dass der Ausgabestrom von Prolog nicht mehr korrekt ausgelesen werden kann, bei einer Rekursionsstufe von fünf dauert die SSA-Transformation bereits über zehn Minuten bei einer fast vollständigen Auslastung des Prozessors. Prolog selbst jedoch ist in der Verarbeitung sogar bei hoher Rekursionsstufe sehr schnell, deutlich unter der Laufzeit des SMT-Solver-Tools von Walther und Wehrheim [17]. Dabei muss Prolog im schlechtesten Fall doppelt so viel berechnen, als es eigentlich müsste, da bisher die Regel für die gesamte Service-Komposition fälschlicherweise zweimal vom Prologvalidator in das Prologprogramm geschrieben wird. Dies ändert aber nichts an der Richtigkeit des Ergebnisses mit Prologvalidator. Ein Graph zu den Laufzeiten bei unterschiedlichen Rekursionsstufen ist in Abbildung 6.1 zu finden.

Theoretischer Vergleich mit SMT-Solver Im Vergleich zum SMT-Solver von Walther und Wehrheim ist der Unterschied des Entwicklungsaufwandes inklusive der Unterstützung der „German Research Foundation“ zu merken [17]. Der SMT-Solver kennt Konzepte für Verifikationen von Service-Kompositionen, die Komponenten unterstützen, bei denen der Prologvalidator beschränkt ist. So verifiziert der SMT-Solver Mengen, Zahlen und berücksichtigt Ontologieregeln [17]. Während jedoch beim SMT-Solver-Ansatz noch der Code für den SMT-Solver manuell hergeleitet werden muss [Stand 2013] [17],

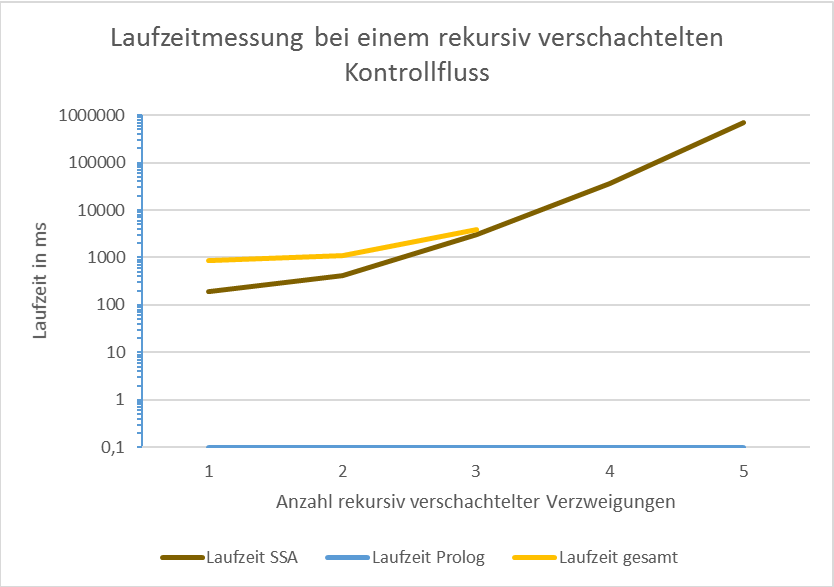


Abbildung 6.1: Laufzeittest-Statistik (Intel Pentium CPU 8960 2 x 2.2GHz, 4GB RAM)

geschieht dies im Falle von Prologcode automatisch mit dem Prologvalidator. Ansonsten ähneln sich beide Ansätze in ihrem Umgang mit dem Kontrollfluss einer Service-Komposition, zum Beispiel der Umgang mit Schleifen. Beide Ansätze benutzen den Weg über die Schleifeninvariante [17].

7 Zusammenfassung und Fazit

In dieser Arbeit habe ich die Verifikation von Service-Kompositionen mit Prolog behandelt. Dazu habe ich zuerst die Entwicklungsumgebung SeSAME kennen gelernt inklusive den Bestandteilen einer Service-Komposition: Ontologie, Kontrollfluss, die Spezifikationen verwendeter Services sowie die Vor- und Nachbedingung der Service-Komposition an sich. Bei der Verifikation wird unter der Annahme der Validität der einzelnen Services gezeigt, dass für jede Eingabe, die die Vorbedingung erfüllt, gilt, auch die Nachbedingung gilt [17]. Als Basis für die Verifikation habe ich die logische auf Hornklauseln basierte Programmiersprache Prolog betrachtet [16]. Um aber mit Prolog verifizieren zu können, muss erst ein Prologprogramm geschaffen werden. Mit der Vorbereitung von Kapitel 3 habe ich in Kapitel 4 eine Übersetzungsroutine entwickelt, mit der ein B3-Datenmodell einer Service-Komposition von SeSAME in ein Prologprogramm mit passender Anfrage an Prolog übersetzt werden kann. Da diese Übersetzung jedoch im Detail nicht trivial ist, musste ich mich mit mehreren Spezialfällen bei der Übersetzung beschäftigen (Kapitel 4.3). Dabei musste ich zusätzlich unterscheiden zwischen der Übersetzung von prädikatenlogischen Formeln und der des Kontrollflusses der Service-Komposition. Des Weiteren musste ich teils abweichende Übersetzungen für Fakten, also das Setzen/ Wissen von Informationen einerseits Abfragen von Informationen andererseits entwickeln. Besonders bei Gleichungen ist diese Unterscheidung bei Prolog entscheidend (Kapitel 4.3.1). Auf Grundlage dieser Übersetzungsroutine habe ich ein Tool namens Prologvalidator in Java als Plugin für SeSAME entwickelt und in Kapitel 5 vorgestellt.

7.1 Fazit

Grundsätzlich ist Prolog eine Sprache mit einer sehr kurzen Bearbeitungszeit für unsere Anwendung (siehe Kapitel 6). Der Backtracking-Algorithmus von Prolog erreicht eine effiziente Untersuchung möglicher Variablenbelegungen. Bedingung hierfür ist, dass man nicht selbst eine endliche Menge zur Modellierung eines Bereiches oder bei Unwissenheit eines konkreten Wertes zur Unifizierung einprogrammiert. Dies wäre beim vordefinierten Prädikat `between` unter SWI-Prolog der Fall [19]. Dabei besteht die Möglichkeit, dass andere Entwicklungsumgebungen/ Compiler und Interpreter für Prolog wie „YAP“ noch bessere Geschwindigkeiten erreichen. Aufgrund der Popularität, der Reife durch das Alter (seit 1987) und vordefinierten Prädikate, die teilweise sehr die Programmierung erleichterten, habe ich mich jedoch in dieser Arbeit für SWI-Prolog entschieden [19]. Des Weiteren passt im Gegensatz zu einer imperativen Programmierung die logische Programmierung sehr zum vorliegenden Problem, da wir bei der Verifikation von Service-Kompositionen wissen, was gilt und was anzunehmen ist [16].

Im Laufe der Arbeit mit Prolog ergaben sich jedoch mehrere nicht triviale Fälle, denen ich mit Spezialbehandlungen begegnen musste. Eine Schwäche von der Verifikation mit Prolog ist, wie diese Sprache mit Zahlen umgeht. Unbekannte (ungebundene Variablen) können in Prolog nicht in arithmetischen Ausdrücken verarbeitet werden. Auch nach längerer Forschung wurde keine Lösung gefunden, wie das Problem ohne erheblichen Arbeitseinsatz zu lösen wäre. Diesen Sachverhalt habe ich in Kapitel 4.3.1 beschrieben. Als nächstes musste berücksichtigt werden, dass Prolog Ausdrücke nicht mehr auswertet, wenn diese linker oder rechter Teil einer Gleichung sind (Ausnahme hierzu bilden die arithmetischen Operatoren wie is und =:=) oder als Parameter gesetzt werden (Ausnahme hierzu bilden einige vordefinierte Prädikate [19]). Das ist im Gegensatz zu manchen imperativen Programmiersprachen wie Java anders, d so stellt diese Einschränkung eine weitere Herausforderung dar. Im Allgemeinen muss also ein Ausdruck vor der Verwendung erst mittels einer Hilfsvariablen ausgewertet werden, ehe er verwendet werden kann (in meinem Fall: bei booleschen Ausdrücken und benutzten Funktionen/ Prädikaten der Ontologie der Service-Komposition). Bei einer rekursiven Übersetzungsroutine (die durch das B3-Datenmodell stark begünstigt wird) verlangt dies nach zusätzlichem Aufwand. Auch der Umstand, dass in Prolog keine Variablenwerte überschrieben werden, und somit die Service-Komposition in SSA-Form vorliegen muss, verlangt nach einer weiteren Programmiereinheit und Berechnungszeit. Dieses Feature ist implementiert.
Insgesamt ist zu sagen, dass Prolog für die Verifikation von Service-Kompositionen nur mit sehr viel Aufwand zu empfehlen ist. Die Suche nach einer Alternative zu Prolog kann sich also, was die Handhabung für den Entwickler anbelangt, durchaus lohnen.

7.2 Zukünftige Arbeit

An dem Tool Prologvalidator ist eine Weiterentwicklung, wenn man es denn benutzen möchte, durchaus ratsam. Zum Einen, damit eine größere Menge von Service-Kompositionen verifiziert werden kann (einige Vorschläge zu bestimmten Problemen sind in diesem Papier bereits gemacht). Hier sehe ich besonders die Einbindung von Zahlen und das Einbinden der Möglichkeit, Ontologieregeln zu verarbeiten.
Zum Anderen, damit der Prologvalidator optimiert wird. Hier sehe ich zu allererst die Entfernung des wiederholten Übersetzens der Service-Komposition. Auch wäre in Abstimmung mit SeSAME eine ausführliche Produkttestphase ratsam- besonders auch dann, wenn in Zukunft Beschränkungen von SeSAME-Seite aufgehoben werden.
Des Weiteren können Schleifen von Prologvalidator noch nicht übersetzt werden. Ein Lösungsansatz ist ein modifizierter Kontrollfluss der Service-Komposition. Hierzu bietet SeSAME jedoch aktuell keine öffentlich deklarierten Möglichkeiten. Quantoren werden ebenfalls noch nicht unterstützt. Dies ist aber nicht entscheidend relevant, da die Unterstützung von selbst definierten Mengen ebenfalls noch nicht vorhanden ist und somit ein großes Einsatzfeld der Quantoren fehlt.

Literaturverzeichnis

[1] E. D. Angelis, F. Fioravanti, A. Pettorossi, & M. Proietti. *Tools and Algorithms for the Construction and Analysis of Systems*, chap. VeriMAP: A Tool for Verifying Programs through Transformations, pp. 568–574. Springer Berlin Heidelberg (2014).

[2] S. Arifulina, S. Walther, M. Becker, & M. C. Platenius. *SeSAME: Modeling and Analyzing High-Quality Service Compositions*. University of Paderborn.

[3] C. Barrett, P. Fontaine, & C. Tinelli. 'The SMT-LIB Standard: Version 2.5'. Tech. rep., Department of Computer Science, The University of Iowa (2015). Available at `www.SMT-LIB.org`.

[4] A. Becker, T. Widjaja, & P. Buxmann. 'Nutzenpotenziale und herausforderungen des einsatzes von serviceorientierten architekturen'.

[5] N. Bjørner, A. Gurfinkel, K. McMillan, & A. Rybalchenko. 'Horn clause solvers for program verification'. In *Fields of Logic and Computation II*. Lev D. Beklemishev, Andreas Blass, Nachum Dershowitz, Bernd Finkbeiner, Wolfram Schulte (2015).

[6] E. Clarke, D. Kroening, & F. Lerda. 'A tool for checking ansi-c programs'.

[7] L. de Moura & N. Bjørner. *Tools and Algorithms for the Construction and Analysis of Systems*, chap. Z3: An efficient SMT solver, pp. 337–340. Springer (2008). 4th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary.

[8] S. Grebenshchikov, A. Gupta, N. P. Lopes, C. Popeea, & A. Rybalchenko. 'Hsf(c): A software verifier based on horn clauses — (competition contribution)'. In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, vol. 7214 of *Lecture Notes in Computer Science*, pp. 549–551. Springer International Publishing (2012).

[9] P. Hartmann. *Mathematik für Informatiker*, chap. Logik, pp. 28–49. Vieweg+Teubner Verlag (2012).

[10] W. Hasselbring. 'Software-architektur'. *Informatik-Spektrum* **29**:48–52 (2006).

[11] D. W. Hoffmann. *Software-Qualität*, vol. 2. Springer Verlag (2013).

[12] B. Humm. 'Was ist eigentlich ein Service?' (2008). Fachgruppenbericht `pi.informatik.uni-siegen.de/gi/stt/28_4/01_Fachgruppenberichte`.

[13] J. Krämer & H. Wehrheim. 'A formal approach to error localization and correction in service compositions'. Paderborn University – Computer Science, Paderborn, Germany.

[14] F. Mohr, A. Jungmann, & H. K. Büning. 'Automated online service composition'. In *Proceedings of the 12th IEEE International Conference on Services Computing (SCC)* (2015).

[15] Y. Peng, L. Ye, Z. Zheng, J. Xiang, J. Gao, J. Ai, Z. Lu, Y. Jin, & X. Jiang (eds.). *Automatic Service Composition Verification Based on Pi-Calculus*. IEEE (2009).

[16] S. Schmitgen. *Prolog: Grundlagen und Anwendungen*. B. G. Teuber Stuttgart 1986 (2013). Mitwirkende Personen: Hans Kleine Büning.

[17] S. Walther & H. Wehrheim. 'Knowledge-based verification of service compositions – an smt approach'. *International Conference on Engineering of Complex Computer Systems* (2013).

[18] S. Walther & H. Wehrheim. *Formal Aspects of Component Software*, chap. Verified Service Compositions by Template-Based Construction, pp. 31–48 (2015).

[19] J. Wielemaker, L. D. Koninck, T. Fruehwirth, M. Triska, & M. Uneson. *SWI Prolog Reference Manual 7.1*. Books on Demand (2014). `http://www.swi-prolog.org`.