Dissertation

Kontraktbasierte Modellierung, Implementierung und Suche von Komponenten in serviceorientierten Architekturen

Schriftliche Arbeit zur Erlangung des Grades

"Doktor der Naturwissenschaften"

Fakultät für Elektrotechnik, Informatik und Mathematik

der Universität Paderborn

vorgelegt von

Dipl.-Inform. Marc Lohmann

Paderborn, Juni 2006

Danksagung

Die hier vorliegende Arbeit wäre nicht ohne die Mitwirkung von zahlreichen netten, kompetenten und mir wohlgesinnten Menschen entstanden. An dieser Stelle möchte ich mich ganz herzlich bei den Personen bedanken, die an dem erfolgreichen Abschluss meiner Promotion mitgewirkt haben.

Ich danke meinem Doktorvater und Chef Gregor Engels dafür, dass er mir diese Arbeit ermöglicht hat. Seine konstruktive Kritik hat mir sehr geholfen, diese Arbeit abzurunden und abzuschließen. Ich möchte mich bei ihm aber auch für das entgegengebrachte Vertrauen und die Förderung meiner persönlichen Entwicklung in den letzten fünf Jahren bedanken.

Weiterhin geht mein Dank an die Mitglieder meiner Promotionskommission Wilhelm Schäfer, Odej Kao, Reiko Heckel und Mathias Gehrke für die Auseinandersetzung mit meiner Arbeit sowie die sehr spannende Verteidigung. Wilhelm Schäfer danke ich insbesondere für seine Bereitschaft ein Gutachten zu meiner Arbeit zu erstellen. Ein besonderer Dank geht auch an Reiko Heckel. In zahlreichen Diskussionen hat er mir bei der grundlegenden Themenfindung geholfen.

Allen Mitgliedern des Fachgebietes Datenbank- und Informationssysteme danke ich für das angenehme und gemeinschaftliche Arbeitsklima. Hier geht ein besonderer Dank an meinen ehemaligen Bürokollegen Jan Hendrik Hausmann und an Stefan Sauer. Beide haben mich bei der Ausarbeitung der zentralen Ideen und Publikationen zu meiner Arbeit unterstützt und mit ihrem Arbeitseifer und Arbeitseinsatz infiziert.

Ein ganz lieber Dank geht auch an meine Frau Sandra und meine Tochter Anna-Malin. Sandra hat mir immer den notwendigen Rückhalt gegeben und mir auch in den stressigsten Phasen die notwendigen Freiräume geschaffen. Anna-Malin danke ich dafür, dass sie mich auf ihre kindliche Art immer wieder motiviert und zu guten Leistungen anspornt.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die stetig zunehmende Globalisierung und Internationalisierung und die sich damit in immer kürzeren Zyklen wandelnden Marktanforderungen stellen die Unternehmen vor große Herausforderungen. Nur wenn ein Unternehmen in der Lage ist, seine Unternehmensstrukturen und Geschäftsprozesse schnell und kostengünstig an neue Marktanforderungen anzupassen, kann es am Markt bestehen.

Die Konsequenzen der gestiegenen Marktanforderungen sind unter anderem von der IT zu tragen. Die Geschäftsprozesse oder genauer gesagt die einzelnen Aktivitäten eines Geschäftsprozesses bestimmen die Funktionalitäten, die den Fachabteilungen eines Unternehmens von der IT zur Verfügung gestellt werden müssen. Zur Umsetzung der benötigten Funktionalitäten müssen existierende und neu zu entwickelnde IT-Systeme innerhalb und außerhalb eines Unternehmens flexibel integriert werden (Abbildung 1.1).

Frühere Integrationsprojekte haben sich auf einzelne Fachabteilungen oder ein einzelnes Unternehmen beschränkt. Abteilungs- oder unternehmensübergreifende Aktivitäten in einem Geschäftsprozess wurden manuell durchgeführt und nicht von der IT unterstützt [ACKM04]. Ein wesentlicher Aspekt für den Erfolg dieser Integrationsprojekte war, dass nur eine begrenzte Anzahl bekannter Systeme komponiert wurde. Der Systemkontext und die Aufgaben der verschiedenen Systeme waren den Entwicklern bekannt.

Eine weitergehende Unterstützung der Geschäftsprozesse eines Unternehmens erfordert eine abteilungs- oder unternehmensübergreifende Integration verschiedener IT-Systeme. Aufgrund der hohen Vernetzung der Rechner in einem Unternehmen und der globalen Vernetzung über das Internet sind die hierfür notwendigen technischen Grundlagen bereits vorhanden. Jedoch

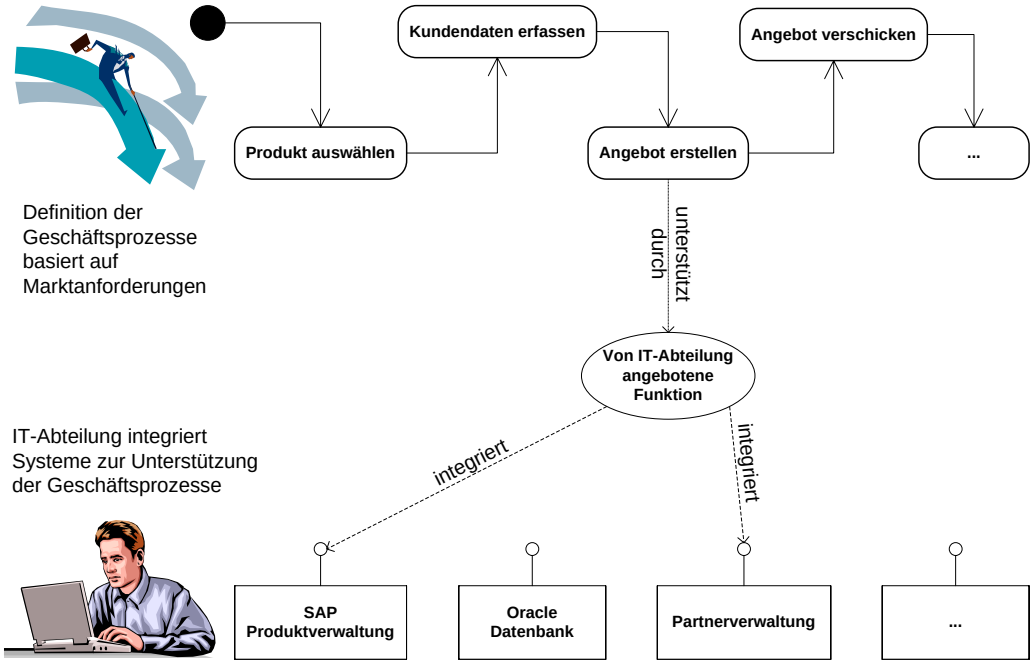


Abbildung 1.1: Integration existierender Systeme zur Unterstützung der Geschäftsprozesse

geht der wesentliche Erfolgsfaktor früherer Integrationsprojekte — die überschaubaren Systemlandschaften — verloren. Die IT-Abteilungen stehen nun vor dem Problem, dass sie auch Systeme integrieren müssen, die unabhängig voneinander von unterschiedlichen Personengruppen in unterschiedlichen Unternehmen entwickelt wurden. Die Aufgaben der verschiedenen Systeme und deren Kontext sind den Entwicklern zu Beginn eines Integrationsprojektes nicht bekannt.

Serviceorientierte Architekturen (SOA) bieten ein großes Potential zur Linderung dieser Integrationsproblematik. In einer SOA werden bestehende und neu zu entwickelnde Systeme über Services gekoppelt. Die Services können sowohl unternehmensintern als auch unternehmensübergreifend miteinander gekoppelt werden. Ein *Service* entspricht einer Operation oder einem Objekt mit einer stabilen und veröffentlichten Schnittstelle, die von einem Client aufgerufen werden kann.

An einer SOA sind drei zentrale Entitäten beteiligt (vgl. Abbildung 1.2). Ein *Service Provider* bietet einen Service für potentielle Nutzer. Dazu hinterlegt der Service Provider eine *Servicebeschreibung* bei einem Discovery Service. Ein *Discovery Service* verwaltet veröffentlichte Servicebeschreibungen und

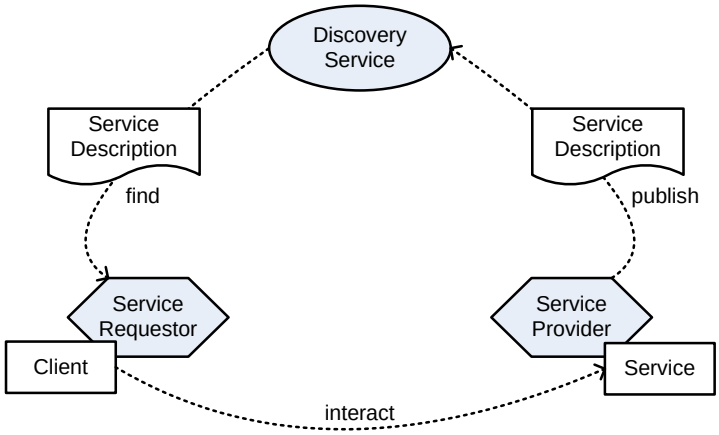


Abbildung 1.2: Serviceorientierte Architekturen

kann Informationen über Services an Service Requestoren weiter geben. Ein *Service Requestor* durchsucht einen Discovery Service nach einem benötigten Service. Nachdem ein Service Requestor einen geeigneten Dienst ausgewählt hat, kann er die Informationen aus den öffentlichen Servicebeschreibungen verwenden, um den Service in eine Anwendung zu integrieren.

Die öffentlichen Servicebeschreibungen stellen in einer serviceorientierten Architektur also ein zentrales Element dar, um einen existierenden Service in einen Client zu integrieren. Häufig wird im Kontext einer SOA auch von einer *losen Kopplung* gesprochen, um die Bedeutung der öffentlichen Servicebeschreibungen hervorzuheben. Hiermit soll ausgedrückt werden, dass die Integration der Services ausschließlich auf Basis der öffentlichen Servicebeschreibungen erfolgt. Weitere Informationen über einen Service stehen aufgrund der unabhängigen Definition, Entwicklung und Verwaltung der verschiedenen Services in unterschiedlichen Unternehmen eher selten zur Verfügung.

Neben dieser grundlegenden Architektur spielt der Softwareentwickler, insbesondere auf der Seite des Service Providers und Service Requestors, eine entscheidende Rolle (siehe Abbildung 1.3). Auf der Seite des Service Providers erstellt ein Softwareentwickler zum einen die Servicebeschreibung, die an den Discovery Service geschickt wird. Zum anderen implementiert ein Softwareentwickler den Service, der die öffentliche Servicebeschreibung realisiert. Die Implementierung eines Service erfolgt heutzutage noch überwiegend manuell und wird nur wenig durch eine automatisierte Codegenerierung unterstützt.

Auf der Seite des Service Requestors ist ein Softwareentwickler für die Implementierung einer Client-Anwendung verantwortlich. Wenn der Software-

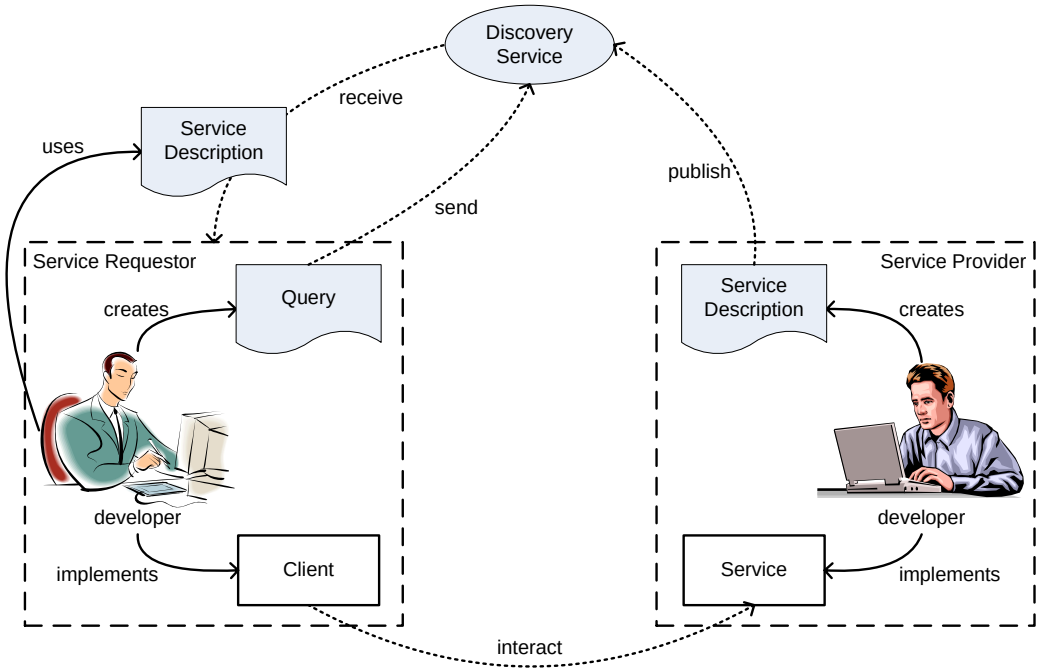


Abbildung 1.3: Rolle der Entwickler in einer SOA

entwickler eine Funktionalität benötigt, die nicht von dem Client selbständig berechnet werden kann oder soll, muss ein Service in den Client eingebunden werden. Zur Bestimmung eines geeigneten Service erstellt der Softwareentwickler eine Suchanfrage und schickt diese Suchanfrage an den Discovery Service. Der Discovery Service sendet daraufhin eine Menge von Servicebeschreibungen zurück, die auf die Suchanfrage passen. Aus dieser Menge von Servicebeschreibungen wählt letztendlich der Softwareentwickler einen geeigneten Service aus, um diesen in den Client einzubinden.

Der Erfolg serviceorientierter Architekturen ist somit eng gekoppelt an die Inhalte und den Einsatz der öffentlichen Servicebeschreibungen:

- Die Services in einer SOA müssen sowohl syntaktisch als auch semantisch beschrieben werden [SH04, ACKM04, BHM+04]. Mit den *syntaktischen Servicebeschreibungen* werden die Datentypen, Signaturen und Transportprotokolle beschrieben, die ein Client verwenden muss, um einen Service aufzurufen. In früheren Integrationsprojekten waren diese syntaktischen Beschreibungen eines Systems aufgrund des begrenzten und bekannten Systemkontextes ausreichend.

  In einer SOA fehlen einem Softwareentwickler jedoch die Informationen

über den Systemkontext eines Service aus welchen er Rückschlüsse auf die Semantik eines Service ziehen kann. Aus diesem Grund kommt den semantischen Servicebeschreibungen in einer SOA eine höhere Bedeutung zu. Mit den *semantischen Servicebeschreibungen* wird das Verhalten eines Service beschrieben. Insbesondere wird festgehalten, wie ein Service auf verschiedene Eingaben reagiert.

- Die öffentlichen Servicebeschreibungen stellen in einer SOA das zentrale Kommunikationsmittel zwischen Service Provider und Requestor dar. Service Provider und Requestor müssen also über Abteilungs- und Unternehmensgrenzen hinweg auf Basis der öffentlichen Servicebeschreibungen ein gemeinsames Verständnis über die Leistungen und Eigenschaften eines Service erlangen. Zu berücksichtigen ist, dass sowohl auf der Seite des Service Providers bei der Erstellung der Servicebeschreibung als auch auf der Seite des Service Requestors bei der Verwendung einer Servicebeschreibung ein Softwareentwickler stark involviert ist. Die Servicebeschreibungen müssen also auch für einen Menschen intuitiv verständlich sein. Hierzu sind angemessene und intuitive Sprachmittel notwendig, welche gleichzeitig von den Details des Systemkontextes, wie z.B. den eingesetzten Implementierungstechnologien, abstrahieren.

- Ein Service kann im Kontext einer SOA von hunderten oder tausenden Service Requestoren aufgerufen werden. Ein Service Requestor verwendet einen Service gemäß seiner öffentlichen Servicebeschreibungen. Verhält ein Service sich nicht gemäß seiner öffentlichen Servicebeschreibung, so kann dies — allein aufgrund des hohen Grades der Wiederverwendung — beträchtliche Probleme und Kosten verursachen. Daher ist in einer SOA sicher zu stellen, dass ein Service funktional korrekt gegenüber seiner Servicebeschreibung ist.

- Wenn man weiterhin betrachtet, dass allein die Anzahl der unternehmensinternen Services leicht hohe dreistellige Werte annimmt, wird deutlich, dass geeignete Verfahren zur Erstellung von Servicebeschreibungen und deren Verwaltung notwendig werden. So berichtet zum Beispiel die Credit Suisse vom Aufbau einer SOA mit weit über 700 Services [Hag03].

  Wenn die öffentlichen Servicebeschreibungen das zentrale Kommunikationsmittel zwischen Service Provider und Requestor darstellen, so muss auch eine effiziente Suche nach einem Service auf Basis der öffentlichen Servicebeschreibungen erfolgen. Ein zu beachtender Punkt an dieser Stelle ist, dass ein Service Requestor weiß, welche Funktio-

nalität benötigt wird, d.h. die Semantik eines gesuchten Service ist bekannt. Die syntaktische Servicebeschreibung ist bei der Suche nach einem Service allerdings nicht notwendigerweise bekannt. Sie werden erst benötigt, wenn ein Service in einen Client eingebunden werden soll [LRE+06]. Primär muss in serviceorientierten Architekturen also eine semantische Suche unterstützt werden.

1.1 Aktuelle Web-Service-Technologien für serviceorientierte Architekturen

Aktuelle Web-Service-Technologien sind ausgerichtet auf die Realisierung serviceorientierter Architekturen. Der Begriff Web Service wird heute mit teilweise unterschiedlichen Bedeutungen verwendet. Existierende Definitionen sind entweder sehr generisch und zu allgemeingültig, oder sie sind zu spezifisch und restriktiv. Zur Konkretisierung des Begriffs Web Service verwenden wir eine Definition des World Wide Web Consortium (W3C):

> „A Web service is a software system identified by a URI, whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web service in a manner prescribed by its definition, using XML based messages conveyed by Internet protocols [AGB+04].“

Die Definition des W3C betont, dass ein Web Service ein Softwaresystem ist, dessen öffentlichen Schnittstellen mit XML-basierten Standards definiert und beschrieben werden. Die Definitionen können gefunden und in Softwaresystemen eingesetzt werden, um XML-basierte Nachrichten mit dem Web Service über das Internet auszutauschen.

Damit gibt die Definition eines Web Service grob den Aufbau einer serviceorientierten Architektur wieder. In das Zentrum der Definition rückt jedoch die Verwendung von XML-basierten Standards. Die Verwendung standardisierter Technologien für den Austausch von Informationen zwischen den einzelnen Entitäten einer SOA ist ein wichtiger Schlüssel zur Erleichterung der Integration von Anwendungen. Durch die Vorgabe und Verwendung von Standards wird die Heterogenität der Technologien in einer SOA reduziert.

Im Zuge der Standardisierung von Web-Service-Technologien sind drei grundlegende Standards zu nennen: SOAP, WSDL und UDDI[1]. SOAP ist ein XML-basiertes Datenformat zum Austausch von Nachrichten. Der Aufruf eines Web Service und die Antwort des Web Service kann in ein SOAP-Dokument [Mit03, GHM+03a, GHM+03b] kodiert werden. Die Schnittstellen (Methodennamen und Signaturen) eines Web Service werden mit der Web Service Description Language (WSDL) [BL05, CMRW05, CHL+05] beschrieben. Eine WSDL-Datei enthält jedoch nur syntaktische Beschreibungen eines Web Service. Diese Schnittstellenspezifikation kann zusammen mit einigen Schlagworten zur Beschreibung des Web Service bei einem zentralen UDDI-Server (Discovery Service) veröffentlicht werden [CHRR04, UDD01b, UDD01a]. Ein Service Requestor kann mit Hilfe der Schlagworte nach einem geeigneten Web Service suchen. Eine semantische, automatisierte Suche ist jedoch nicht möglich.

Aktuelle Web-Service-Technologien stellen also lediglich eine minimale Infrastruktur zur Realisierung serviceorientierter Architekturen dar:

- Die semantische Beschreibung von Web Services wird bisher nicht oder nur sehr unzureichend unterstützt [AFM+05].
- Die verwendeten XML-basierten Standards zur Beschreibung von Web Services erlauben eine automatisierte Verarbeitung. Sie stellen jedoch kein für Menschen geeignetes Kommunikationsmittel dar.
- Die bisherigen Standardisierungsbemühungen konzentrieren sich auf die Definition von Austauschformaten für die Beschreibung von Web Services. Es fehlen Vorgehensweisen, um zu überprüfen, ob eine Implementierung eines Web Service konsistent zu seinen öffentlichen Servicebeschreibungen ist.
- Aufgrund der fehlenden Standards zur Beschreibung der Semantik von Web Services, ermöglichen aktuelle Web-Service-Technologien auch keine geeignete semantische Suche nach einem Web Service.

1.2 Aufgabenstellung

In dieser Arbeit wird eine modellbasierte Vorgehensweise beschrieben, die eine Entwicklung von Softwarekomponenten für den Einsatz in einer serviceorientierten Architektur unterstützt. Hierbei berücksichtigen wir die zentrale

[1] Eine detaillierte Diskussion dieser Standards findet sich in Abschnitt 6.1.

Rolle der Softwareentwickler und der öffentlichen Servicebeschreibungen in einer serviceorientierten Architektur.

Insbesondere untersuchen wir in dieser Arbeit die Erstellung und Verwendung semantischer Servicebeschreibungen in einer serviceorientierten Architektur. Zur Unterstützung der Softwareentwickler müssen die semantischen Servicebeschreibungen intuitiv verständlich sein. Die Erstellung der öffentlichen Servicebeschreibungen ist in heutige modellbasierte Softwareentwicklungsprozesse zu integrieren, um sicherstellen zu können, dass ein Service sich gemäß seiner semantischen Servicebeschreibungen verhält. Weiterhin sollen diese Servicebeschreibungen für die Verwaltung der Services, d.h. für eine effiziente Suche, einsetzbar sein.

In Kapitel 2 geben wir einen Überblick über die grundlegenden Konzepte unserer Arbeit zur Lösung der angesprochenen Probleme sowie einen Überblick über den Aufbau dieser Arbeit.

Kapitel 2

Kontraktbasierte Modellierung, Implementierung und Suche in serviceorientierten Architekturen

Öffentliche Servicebeschreibungen und Softwareentwickler nehmen — wie bereits zuvor geschildert — in einer serviceorientierten Architektur eine zentrale Rolle ein. Auf der Provider-Seite sind die Softwareentwickler für die Erstellung der Servicebeschreibungen und die Implementierung der dazugehörigen Services verantwortlich. Auf der Requestor-Seite sind die Softwareentwickler für die Suche nach einem geeigneten Service, der in einen Client eingebunden werden soll, verantwortlich. In dieser Arbeit konzentrieren wir uns insbesondere auf die semantischen Servicebeschreibungen und deren Verwendung durch den Softwareentwickler in einer serviceorientierten Architektur.

Abbildung 2.1 gibt einen Überblick über die Probleme in einer serviceorientierten Architektur, die wir in dieser Arbeit näher betrachten. Zur Beschreibung der Semantik eines Service ist eine Notation, die für einen Softwareentwickler sowohl auf Provider- als auch auf Requestor-Seite intuitiv verständlich ist, notwendig. Ein Service Provider muss sicherstellen können, dass ein angebotener Service sich korrekt gegenüber seinen öffentlichen Servicebeschreibungen verhält. Hierbei ist zu berücksichtigen, dass die Implementierung eines Softwaresystems in einer serviceorientierten Architektur in der Regel manuell erfolgt. Die öffentlichen Servicebeschreibungen müssen zudem für die Verwaltung der Services in einer SOA geeignet sein. Hierzu muss ein Discovery Service eine Servicebeschreibung mit einer Suchanfrage vergleichen können. Dabei ist zu berücksichtigen, dass ein Softwareentwickler auf Requestor-Seite geeignete Möglichkeiten zur Erstellung einer Suchanfra-

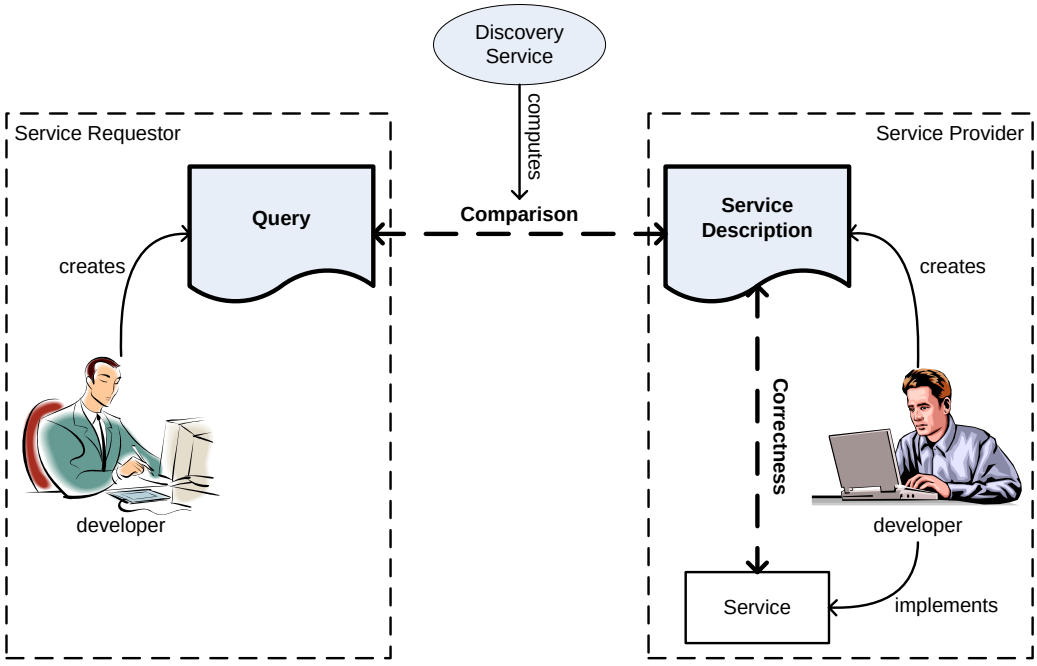


Abbildung 2.1: Probleme in einer serviceorientierten Architektur

ge benötigt.

Zur Lösung der oben dargestellten Probleme in einer serviceorientierten Architektur integrieren wir in dieser Arbeit drei neue, innovative Konzepte. Zur Spezifikation der Semantik von Services führen wir *visuelle Kontrakte* ein. Die Verwendung der visuellen Kontrakte in einem objektorientierten Softwareentwicklungsprozess ermöglicht ein *Model-Driven Monitoring* zur Überprüfung der Korrektheit eines Service. Der Einsatz der visuellen Kontrakte zur Beschreibung einer Suchanfrage erlaubt den Vergleich einer Suchanfrage mit einer Servicebeschreibung auf Basis unseres *Model-Driven Matchings*.

In diesem Kapitel geben wir einen Überblick über diese drei grundlegenden Konzepte unserer Arbeit und deren Integration mit Hilfe eines objektorientierten Softwareentwicklungsprozesses. Da wir mit den visuellen Kontrakten Design-by-Contract-Konzepte von der Implementierungsebene auf die Modellebene heben, erläutern wir zuvor die Grundlagen von Design by Contract und dessen Einsatz in einem Softwareentwicklungsprozess. Weiterhin motivieren wir die Bedeutung von Modellen in einer serviceorientierten Architektur zur Erstellung von Web Services. Zum Abschluss des Kapitels geben wir einen Überblick über den Aufbau der Arbeit sowie bibliographische Hinweise über die bereits erfolgten Veröffentlichungen der Ergebnisse dieser Arbeit.

2.1 Design by Contract

In den letzten Jahren hat sich Design by Contract (DbC) [Mey92a, Mey97] als eine Technologie zur Entwicklung zuverlässiger Software weitgehend etabliert. Die Design-by-Contract-Technologie integriert verschiedene Konzepte aus drei grundlegenden Forschungsfeldern der Informatik: Objektorientierung [NDM70][1], abstrakte Datentypen [Gut77, GTW78] sowie der Theorie der Programmverifikation und systematischen Konstruktion von Programmen [Flo67, Hoa69, Dij76].

Die grundlegende Idee von Design by Contract basiert auf der Verwendung von Assertions zur Spezifikation der Semantik einer Klasse. Die Semantik der Operationen einer Klasse wird mit Kontrakten spezifiziert. Ein Kontrakt besteht aus einer Vor- und einer Nachbedingung. Die Vorbedingung beschreibt die Voraussetzungen für den erfolgreichen Aufruf einer Operation, die Nachbedingung beschreibt die Effekte des Aufrufs einer Operation. Diese semantischen Spezifikationen beschreiben, wie eine Provider-Klasse — ein *Anbieter* von Diensten bzw. Operationen — mit möglichen Client-Klassen (*Clients*, welche die Operationen des Anbieters aufrufen) zusammenarbeiten kann. Ein Softwaresystem wird dabei als ein Netzwerk von miteinander kommunizierenden Clients und Anbietern betrachtet, dessen Austausch von Anfragen und Services durch dezentralisierte Kontrakte detailliert beschrieben sind. Bei der Verwendung von Design by Contract wird also jedes Programm, welches Dienste für potentielle Clients anbietet, als Kontraktor angesehen. Der Schlüssel zur Entwicklung zuverlässiger Software ist dabei nicht nur die Nachbedingung, die spezifiziert, was der Kontraktor macht, sondern auch die Vorbedingung, die spezifiziert, was der Client zu berücksichtigen hat.

Design by Contract hat damit viel gemeinsam mit Verträgen zwischen Organisationen oder Personen. Wenn zum Beispiel eine Person in Paderborn ein wichtiges Paket an eine bestimmte Adresse am anderen Ende der Stadt liefern muss, dann bleibt die Möglichkeit das Paket selbst zu liefern oder einen Paketdienst zu beauftragen[2]. Für eine schnelle Auslieferung macht der Paketdienst (siehe Abbildung 2.2) jedoch gewisse Einschränkungen, die zu beachten sind. Wenn sich die Parteien auf einen Vertrag einigen, so werden beide vor Unannehmlichkeiten geschützt. Der Client weiss, was er tun muss, damit sein Paket ankommt. Der Anbieter spezifiziert für ihn akzeptable Bedingungen. Die Verpflichtungen des Einen werden zum Nutzen des Anderen.

[1] Simula war die erste objektorientierte Sprache, welche die Konzepte Vererbung und dynamisches Binden eingeführt hat.

[2] Das Beispiel ist angelehnt an das Beispiel aus [Mey92a]

Partei	Verpflichtungen	Nutzen
Client	Das Paket darf maximal ein Gewicht von 35 kg und 50 x 50 x 50 cm groß sein.	Das Paket wird innerhalb von 30 Minuten oder weniger innerhalb Paderborn ausgeliefert.
Anbieter	Das Paket wird innerhalb von 30 Minuten oder weniger innerhalb Paderborn ausgeliefert.	Es besteht nicht die Gefahr, dass ein Paket zu gross oder zu schwer für die eingesetzten Fahrzeuge ist.

Abbildung 2.2: Kontrakt zur Beförderung von Paketen

In Abbildung 2.2 stellt die Verpflichtung des Clients, nur Pakete mit einer maximalen Größe und einem maximalen Gewicht aufzugeben einen Nutzen für den Anbieter dar, da dieser seine Fahrzeuge geeignet beladen kann.

Ein weiteres wichtiges Design-by-Contract-Konzept ist die Überwachung der Kontrakte zur Laufzeit. Hierzu werden die Kontrakte typischerweise mit einer Erweiterung einer Programmiersprache spezifiziert und in den Programmcode eingebettet. Ein Compiler kann dann den Programmcode einschließlich der eingebetteten Assertions in ausführbaren Code übersetzen. Dieser ausführbare Code kann überprüfen, ob die Vor- oder Nachbedingung eines Kontraktes verletzt sind, wenn ein Client eine Operation aufruft. Jede Verletzung eines Kontraktes kann somit während der Ausführung eines Programmes bemerkt werden. Basierend auf den Kontrakten, wird somit auch ein konsistenter Mechanismus zur Behandlung von Fehlern ermöglicht. Wenn eine Assertion zur Laufzeit überwacht wird, können Verletzungen eines Kontraktes erkannt und entsprechende Ausnahmebehandlungen (*Exceptions*) ausgeführt werden. Dezentralisierte Handler zur Verwaltung der Exceptions können dann als genereller Mechanismus definiert und implementiert werden, um auftretende Fehler zu verwalten.

Es bleibt festzuhalten, dass die Verwendung von Softwarekontrakten in der professionellen Softwareentwicklung ein wichtiger Schritt hin zur Entwicklung von zuverlässiger (korrekter und robuster) Software ist. Diese Qualitäten werden im Kontext von wiederverwendbarer Software immer wichtiger, insbesondere wenn das Verhalten viele Applikationen beeinflusst. Im Kontext serviceorientierter Architekturen kommt noch hinzu, dass die Anzahl und Art der Clients, die auf die Funktionalität eines Service zugreifen, nicht bekannt ist. Insbesondere Programme, die Funktionalitäten als Services anbieten, müssen also zuverlässig sein.

Design by Contract bietet einen systematischen Ansatz zur Entwicklung von korrektem objektorientiertem Code. Wenn bei der Entwicklung einer Anwendung Softwarekontrakte nach der Design-By-Contract-Methodik erstellt werden sollen, so müssen den Klassen, genauer gesagt den Operationen einer Klasse, zusätzliche Spezifikationen hinzugefügt werden. Das heißt die Kontrakte werden während der Implementierung spezifiziert. In heutigen Softwareentwicklungsprozessen erfolgt die Spezifikation einer Anwendung jedoch bereits in der Analyse oder dem Design und nicht erst während der Implementierung.

Die Möglichkeit, Softwarekontrakte bereits in der objektorientierten Analyse oder dem objektorientierten Design — also sobald erste Operationen und deren Verhalten identifiziert werden — zur Spezifikation der Semantik einer Anwendung zu verwenden, erlaubt eine saubere Trennung von Spezifikation und Implementierung. Für die durchgängige Verwendung von Design-by-Contract-Methodiken im Softwareentwicklungsprozess ist es notwendig, dass die Spezifikationen (in der Analyse oder dem Design) und die Implementierung einer Anwendung auf derselben konzeptionellen Sicht einer Problemdomäne basieren, d.h. sie müssen eine gemeinsame Menge von Klassen und Assoziationen besitzen. Bei objektorientierten Softwareentwicklungsprozessen kann diese Notwendigkeit als gegeben vorausgesetzt werden. Im folgenden Abschnitt gehen wir noch detaillierter auf dieses notwendige Prinzip im Zusammenhang mit Softwarekontrakten ein.

2.2 Objektorientierte Softwareentwicklungsprozesse und Design by Contract

Softwareentwicklungsprozesse basieren auf der grundlegenden Idee, dass bei der Softwareentwicklung verschiedene Abstraktionsebenen zur Beschreibung der zu entwickelnden Anwendung durchlaufen werden. In den späteren Phasen eines Softwareentwicklungsprozesses wird die Beschreibung der Anwendung bis hin zur Implementierung immer konkreter. Eine zu entwickelnde Anwendung wird also sukzessiv in den späteren Phasen immer weiter detailliert.

Objektorientierte Softwareentwicklungsprozesse erlauben zudem eine reibungslose Transition der Benutzeranforderungen über die Analyse und das Design in die Implementierung einer Anwendung. Der Grund dafür liegt in der Objektorientierung der Softwareentwicklungsprozesse: Der zentrale Abstrakti-

onsmechanismus der Objektorientierung — die Klasse — kann in verschiedenen Phasen eines Softwareentwicklungsprozesses verwendet werden.

Zum einen können mit Klassen Elemente der realen Welt, wie z.B. Krankenhäuser oder Hochregallager beschrieben werden. Zum anderen können mit Klassen Objekte beschrieben werden, die in einer Implementierung zur Darstellung von Adressen oder Aufträgen benötigt werden. Auch wenn sich die semantische Interpretation der gewählten Abstraktionsebene einer Klasse in den beiden dargestellten Fällen ändert, so ändert sich die generelle Aussage nicht. In beiden Fällen werden statische Informationen mit Klassen und ihren Attributen und Assoziationen beschrieben. Das Verhalten wird mit Operationen dargestellt.

Wenn ein zentrales Paradigma in einem Softwareentwicklungsprozess von der Anforderungsanalyse bis zur Implementierung, Installation und Wartung beibehalten werden kann, so hat dies verschiedene Vorteile. Die Kommunikation zwischen den Projektbeteiligten, denen in den Phasen eines Softwareentwicklungsprozesses unterschiedliche Rollen zukommen, wird enorm vereinfacht, da alle das grundlegende Konzept der Klasse verstehen. Künstliche Barrieren zwischen Designern und Programmierern lösen sich auf und machen Platz für eine gesamtheitliche Sicht auf das System. Noch wichtiger ist, dass ein Prozess aufgrund der wohl definierten, nahtlosen Übergange zwischen den Phasen leichter nachvollziehbar wird. Klassen und Assoziationen, die in der Analyse und im Design zur Beschreibung eines Systems eingeführt wurden, werden auch in der Implementierung vorhanden sein [WN94]. Die Entwicklung initialer Anforderungen kann über das Design bis hin zur Implementierung verfolgt werden (vgl. auch existierende Prozesse wie z.B. [Kru99]).

Softwarekontrakte beziehungsweise die Assertions in den Softwarekontrakten sind Aussagen über den Zustand eines Softwaresystems vor und nach dem Aufruf einer Operation. Die primitiven Elemente in den Assertions sind lediglich Anfragen zur Bestimmung des Zustands eines Objektes oder Konstanten. Die primitiven Elemente können durch unterschiedliche Operatoren kombiniert werden, um logische Ausdrücke zu bilden. Da in einer Assertion nur die Zustände existierender Objekte abgefragt werden, ist die Semantik eines Kontraktes spezifiziert durch die Objekte bzw. die Klassen der Objekte, die das Softwaresystem ausmachen und als Anfrage in die Assertion eingebunden sind. Somit ist festzuhalten, dass die Spezifikation eines objektorientierten Systems unter der Verwendung von Design by Contract aus einer Menge von Klassen besteht, deren Operationen mit Bezug auf die Spezifikation anderer Klassen des Systems spezifiziert werden. Die Klasse als zentraler Abstraktionsmechanismus der Objektorientierung ist also auch ein elemen-

tarer Bestandteil der Spezifikation eines Kontraktes.

Die Objekte und Links bzw. Klasssen und Assoziationen, die während der Spezifikation der Anwendung in der Design-Phase verwendet werden, sind auch Teil der späteren Implementierung. Damit sind auch die Objekte und Links, die während der Analyse oder des Designs in den Assertions verwendet werden, Teil der späteren Implementierung. Diese Beobachtung ist wichtig, da ansonsten die Operationen der Implementierung mit einer Menge von Objekten spezifiziert werden, die nicht Teil des Systems sind. In der Regel werden einem System in späteren Entwicklungsphasen lediglich weitere Klassen hinzugefügt, vornehmlich aus zwei Gründen: Die initiale Spezifikation war unvollständig und oft bedingt die Wahl der Implementierung weitere Klassen. Eine wichtige Beobachtung und Schlussfolgerung ist, dass eine objektorientierte Implementierung und seine Spezifikation dieselbe Basismenge an Klassen und Assoziationen enthalten. Die grundlegende Annahme dabei ist, dass eine gute Spezifikation alle Klassen, Attribute und Assoziationen enthält, um die aus der Anforderungsdefinition hervorgehenden notwendigen Zustände des Systems zu beschreiben. Der Rest der Implementierung besteht aus neuen Klassen, zusätzlichen Attributen, neuen Operationen auf existierenden Klassen oder aus allgemeinen Support-Klassen [WN94].

Design by Contract erlaubt die Annotation von Operationen mit Kontrakten während der Implementierung. Um eine Überwachung der Kontrakte zur Laufzeit zu ermöglichen, werden die Kontrakte in einer Erweiterung der im Softwareentwicklungsprozess verwendeten Programmiersprache definiert. Wie oben erläutert, ist ein durchgängiger Einsatz von Design-by-Contract-Methodiken in einem Softwareentwicklungsprozess möglich. Die Klassen, Attribute und Assoziationen, die in den Assertions zur Spezifikation des Verhaltens von den Operationen in der Analyse oder dem Design verwendet werden, können auch in der Implementierung verwendet werden. Die Assertions aus dem Design müssen lediglich in die Implementierung integriert werden. Damit kann auch in einem Softwareentwicklungsprozess, der durchgängig Design by Contract verwendet, der große Vorteil von Design by Contract, die Überwachung der Assertions zur Laufzeit, beibehalten werden.

Die Formulierung der Assertions in einer Programmiersprache, um diese ausführbar zu machen, ist aber auch gleichzeitig ein Nachteil von Design by Contract. Ein Softwareentwickler, der einen Client implementiert, muss die Beziehung zwischen seinem Client und den Anbietern von Diensten verstehen. Wenn die Dienste in unterschiedlichen Programmiersprachen entwickelt wurden, muss er die verschiedenen Programmiersprachen kennen, um einen Dienst aus seinem Client korrekt aufrufen zu können. In modernen Softwa-

reentwicklungsprozessen hat sich zudem der Einsatz von Modellen bewährt. Sie bieten eine geeignete Abstraktion von den Details der zur Implementierung verwendeten Technologien. Insbesondere statische Spezifikationen in Form von UML 2.0 Klassendiagrammen haben sich in der Analyse und dem Design durchgesetzt. In einem modellbasierten Softwareentwicklungsprozess sollten auch die Verhaltensspezifikationen von Operationen mit Kontrakten auf der gleichen Grundlage erfolgen. Zum Einsatz der Design-by-Contract-Konzepte in der Analyse oder dem Design werden somit geeignete Modelle benötigt. Die Verwendung von Modellen erlaubt auch eine intuitive Verwendung der Design-by-Contract-Konzepte in serviceorientierten Architekturen. Die Vorteile des Einsatzes von Modellen in einer serviceorientierten Architektur, insbesondere im Kontext von Web Services, erläutern wir im folgenden Abschnitt.

2.3 Modelle für Web Services

Web Services sind keine Standalone-Technologie. Web Services sind eine standardisierte Schnittstellen- oder Middlewaretechnologie, um die Funktionalitäten eines komplexen Softwaresystems über das Internet entweder zu veröffentlichen oder aufzurufen. Die Verwendung der Web-Service-Technologien ist normalerweise weder das Hauptziel bei der Entwicklung eines Softwaresystems — es sei denn, das Ziel der Entwicklung ist eine Technologiedemonstration — noch sind Web-Service-Technologien die wichtigste oder einzige verwendete Technologie in einem Softwareentwicklungsprozess. Daher stehen Softwareentwickler bei der Entwicklung von Web Services der Frage gegenüber, wie die Web-Service-Technologien mit dem Rest des Systems integriert werden können. Ein Service Provider muss hierfür entscheiden, welche Schnittstellen eines zu entwickelnden Systems als Web Services veröffentlicht werden sollen [JZC04].

Wenn entschieden wurde, welche Schnittstellen als Web Service bereit gestellt werden sollen, müssen geeignete Beschreibungen der Schnittstellen erstellt und veröffentlicht werden. Die Signaturen der Schnittstellen, bestehend aus Operationen und den Datentypen der Parameter, werden heutzutage mit einem WSDL-Dokument (Web Service Description Language) [BL05, CMRW05, CHL+05] veröffentlicht. Wenn der angebotene Service eine komplexere Struktur besitzt und eventuell aus einer Sequenz von Operationsaufrufen besteht, so wird die Verwendung einer geeigneten Prozessbeschreibungssprache wie zum Beispiel BPEL (Business Process Execution

Language for Web Services) [ACD+03] notwendig. Zusätzlich wird ein Layer, der die SOAP-basierte [Mit03, GHM+03a, GHM+03b] Kommunikation zwischen Client und Service in Methodenaufrufe übersetzt, benötigt und die Servicebeschreibungen müssen auf einem UDDI-Server [CHRR04, UDD01b, UDD01a] veröffentlicht werden. Wenn weiterhin neben der syntaktischen Beschreibung mit WSDL eine semantische Beschreibung eines Web Service veröffentlicht werden soll, so kann heutzutage beispielsweise ein Dialekt der Sprachen DAML (DARPA Agent Markup Language) [CvHH+01] oder OWL (Web Ontology Language) [SWLM04, MvH04, BvHH+04] verwendet werden (vgl. auch Abschnitt 6.1).

Bei der Entwicklung von Web Services müssen somit verschiedene Beschreibungen in unterschiedlichen Sprachen erstellt werden. Die unterschiedlichen Beschreibungen enthalten dabei sogar zum Teil redundante Informationen. So werden zum Beispiel die als Web Service veröffentlichten Signaturen einer Schnittstelle in verschiedenen Web-Service-Spezifikationen (WSDL, OWL-S) und auch in der zur Implementierung des Web Service verwendeten Programmiersprache beschrieben.

Ohne eine geeignete Unterstützung für die Planung, die Implementierung, das Testen und die Dokumentation des Web-Service-Bestandteils eines Systems ist es fragwürdig, ob der Web-Service-Teil zuverlässig und konsistent mit dem Rest des Systems ist. Insbesondere ist es fragwürdig, ob der Web-Service-Teil zuverlässig und konsistent zum restlichen System bleibt, wenn das System z.B. an neue Anforderungen angespasst wird.

Der Einsatz von Modellen bei der Entwicklung von Web Services und deren Beschreibungen sind ein erheblicher Schritt zur Lösung dieser Probleme. Modelle bieten die Möglichkeit von den detaillierten Problemen der Implementierungstechnologien zu abstrahieren, wodurch auch ein Entwickler die Möglichkeit bekommt sich mehr auf die abstrakteren Ziele in einem Softwareentwicklungsprojekt zu konzentrieren. In den letzten Jahren haben sich insbesondere die Diagramme der Unified Modeling Language (UML) [OMG04] als ein wichtiger Bestandteil moderner Softwareentwicklungsprozesse etabliert. Mit dem Aufkommen der Model Driven Architecture (MDA) [OMG01a, RSN03] bekommen Modelle bei der Entwicklung von Softwaresystemen einen noch höheren Stellenwert, da Artefakte der Implementierung automatisch aus den Modellen generiert werden sollen, um Zeit zu sparen und um sicher zu stellen, dass die Implementierung konsistent zu ihrer modellbasierten Spezifikation ist.

Eine weitergehende Unterstützung der Entwicklung von Web Services mit

Modellen hat verschiedene Vorteile (vgl. auch [HHL05, HHL04]):

- Modelle sind unabhängig von einer Zielsprache. Somit ist es möglich, verschiedene Spezifikationen aus denselben Modellen zu generieren. Weiterhin können die Spezifikationen aktuell gehalten werden. Im Kontext serviceorientierter Architekturen bedeutet das, das aus den Modellen, die zur Spezifikation eines Systems verwendet werden, sowohl Codefragmente zur Implementierung des Systems als auch WSDL-Dokumente und semantische Servicebeschreibungen generiert werden können.

  Wenn sowohl die Implementierung als auch die veröffentlichten Beschreibungen eines Web Service aus einer Quelle generiert werden, so kann die Konsistenz zwischen dem Web-Service-Teil der Anwendung und der restlichen Anwendung sichergestellt werden. Bei einer Änderung der Anwendung (auf Modell- bzw. Spezifikationsebene) können gleichzeitig zur Sicherung der Konsistenz die WSDL-Dokumente und semantischen Beschreibungen neu generiert werden.

  Weiterhin existieren zur Zeit verschiedene Sprachen zur Repräsentation der Semantik von Web Services, wie z.B. OWL [MvH04, SWLM04] oder WSMO [BFM02, FB02, RKL+05]. Unklar ist, ob sich einer dieser beiden Sprachen oder eventuell eine andere Semantic-Web-Sprache in Zukunft als Industriestandard etablieren wird. Dies stellt einen weiteren Grund für die Verwendung einer Vorgehensweise dar, die unabhängig vom verwendeten Austauschformat ist [AFM+05, SVSM03].

- Modelle sind visuell. Wenn strukturelle Information wie z.B. Ontologien mit visuellen Modellen dargestellt werden, können sie wesentlich intuitiver verstanden werden als z.B. seitenweise XML-Code (vgl. Abschnitt 6.1). Im Bereich des Software Engineering sind die Vorteile visueller Modelle bereits seit langem bekannt und werden dort intensiv für das Design von Systemen genutzt.

  Im Bereich der Web Services und des Semantic Webs wird diese Idee bereits teilweise genutzt, um Ontologien mit UML-Klassendiagrammen darzustellen und diese dann in maschinenauswertbare Semantic-Web-Sprachen zur Repräsentation von Ontologien wie z.B. DAML+OIL oder OWL zu übersetzen [BKK+02, BKK+01, FFJ+02, Cra01, CP99, FSS03].

- Heutzutage werden Modelle immer wichtiger, was sich auch in der Vielzahl an Werkzeugen wiederspiegelt, um Modelle im Softwareentwicklungsprozess einsetzen zu können. Wenn die Entwicklung von Web Ser-

vices und deren Beschreibungen wie auch der Rest des Systems auf modellbasierten Standards aufbaut, so können existierende Informationen und Werkzeuge wieder verwendet werden. Damit kann die Entwicklung von Web Services in modellbasierte Softwareentwicklungsprozesse integriert werden.

Aus diesen Gründen haben wir zur Beschreibung der Semantik von Web Services (bzw. den einzelnen Operationen eines Web Service) und Suchanfragen eine Technik (visuelle Kontrakte) gewählt, die intuitiv einsetzbar und kompatibel mit der UML ist und in einen Softwareentwicklungsprozess eingebettet werden kann.

2.4 Visuelle Kontrakte

Design by Contract erlaubt die präzise Spezifikation des Verhaltens einer Operation mit Kontrakten. Ein Kontrakt besteht aus einer Vor- und einer Nachbedingung. Existierende DbC-Ansätze verwenden hauptsächlich textuelle Repräsentationen von Kontrakten, welche in Form von Assertions zur Annotation des Quellcodes verwendet werden. Damit sind diese Kontrakte für den Einsatz in Softwareentwicklungsprozessen und in serviceorientierten Architekturen nur bedingt geeignet. Die implementierungsnahen Kontrakte sind nicht intuitiv verständlich und unterstützen somit nicht die Kommunikation zwischen verschiedenen Softwareentwicklern.

Die UML (Unified Modeling Language) [OMG04] als objektorientierte Modellierungssprache war in den letzten Jahren sehr erfolgreich und ist ein etablierter Bestandteil modellbasierter Softwareentwicklungsprozesse. Sie konzentriert sich auf Möglichkeiten zur Beschreibung der statischen Struktur von Softwaresystemen, sowie deren Abhängigkeiten während der Laufzeit. Graphische Notationen zur Beschreibung von Vor- und Nachbedingungen oder genauer gesagt zur Beschreibung der Änderung von Objektstrukturen (die einen Systemzustand beschreiben) bei der Ausführung einer Operation werden von der UML nicht zur Verfügung gestellt. In der UML wird die Spezifikation von Vor- und Nachbedingungen lediglich durch die textuelle Object Constraint Language (OCL) [OMG03b] unterstützt. Mit der OCL können die erlaubten Instanzen eines Modells weiter eingeschränkt werden. Jedoch sind Spezifikationen in der OCL textuell und aufgrund ihrer Komplexität nur sehr beschwerlich zu erstellen, insbesondere für Softwareentwickler, welche typischerweise nicht mit OCL arbeiten.

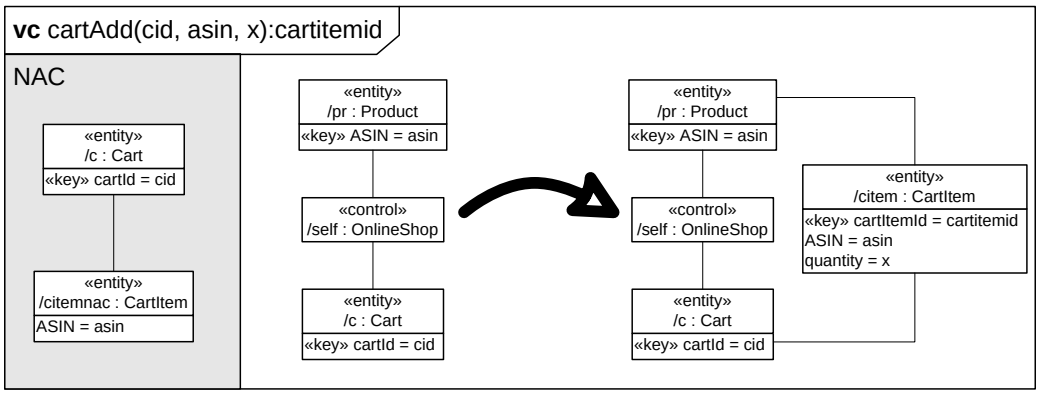


Abbildung 2.3: Visueller Kontrakt zur Beschreibung der Vor- und Nachbedingung einer Operation

Diese Lücke soll mit unseren visuellen Kontrakten geschlossen werden, um den bereits existierenden Ansatz zu vervollständigen. Die grundlegende Idee unserer visuellen Kontrakte ist die Formulierung von Zustandsänderungen über den Klassen eines Klassendiagramms. Während hierfür verschiedene Technologien existieren (z.B. logische Ausdrücke wie sie üblicherweise im klassischen Design-by-Contract-Ansatz verwendet werden), verwenden wir Composite-Structure-Diagramme (Kompositionsstrukturdiagramme) aus der UML 2.0 Spezifikation, um die Vor- und Nachbedingungen einer Operation zu beschreiben. Ein visueller Kontrakt beschreibt das Verhalten einer Operation über die Veränderung der Objektstrukturen durch Vor- und Nachbedingungen, welche mit einem Paar von UML-Composite-Structure-Diagrammen dargestellt werden. Sowohl die Vor- als auch die Nachbedingung eines visuellen Kontraktes sind getypt über einem Klassendiagramm. Mit der Verwendung von UML-Diagrammen verwenden wir eine intuitive, modellbasierte und visuelle Notation, die Softwareentwicklern vertraut ist. Abbildung 2.3 zeigt ein Beispiel für einen visuellen Kontrakt. Im Detail werden die visuellen Kontrakte in Kapitel 3 eingeführt.

Die visuellen Kontrakte ermöglichen eine integrierte Beschreibung von Transformationen über die Datenzustände eines Systems und reaktivem Verhalten. Ein visueller Kontrakt beschreibt somit eine Transformationsregel über die internen Objektstrukturen einer Anwendung. Die Transformationsregel kann ausgelöst werden durch den Aufruf einer Operation, die dem visuellen Kontrakt zugeordnet ist. Mit den visuellen Kontrakten werden die Hauptmängel der UML 2.0 Kommunikationsdiagramme, die nur den Kontrollfluss von Methoden modellieren können, beseitigt. Eine Modellierung der dynamischen Veränderungen von Objektstrukturen, die den Zustand eines Systems aus-

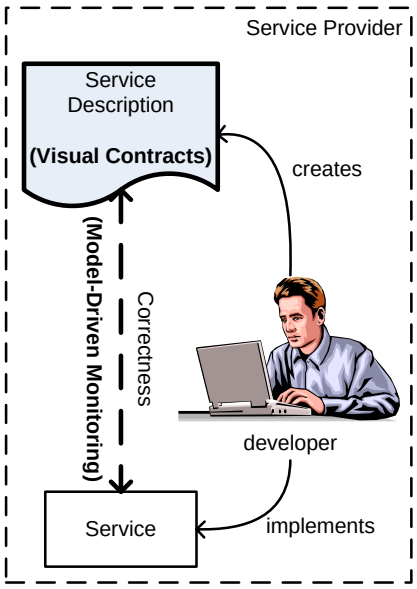


Abbildung 2.4: Model-Driven Monitoring mit visuellen Kontrakten

machen, ist mit Kommunikationsdiagrammen nicht möglich.

2.5 Model-Driven Monitoring

Klassische Model-Driven-Development-Ansätze konzentrieren sich auf die automatische Generierung von vollständigem, funktionalem Code aus Modellen und Model-Driven-Testing-Ansätze konzentrieren sich auf die automatische Ableitung von Testfällen und Testorakeln aus Modellen. Neben diesen existierenden modellbasierten Ansätzen ist Model-Driven Monitoring eine neue, innovative Strategie zur Unterstützung der modellbasierten Softwareentwicklung, die wir in diesem Abschnitt erläutern.

In heutigen Softwareentwicklungsprozessen erfolgt die Implementierung eines Softwaresystems häufig noch manuell. Eine Konsequenz ist, dass nicht garantiert werden kann, dass sich eine manuell erstellte Implementierung korrekt gegenüber ihrer Spezifikation, die mit Modellen (häufig mit UML-Modellen) festgehalten wird, verhält. Mit dem Model-Driven Monitoring verfolgen wir einen Ansatz, der es erlaubt aus Modellen, insbesondere aus visuellen Kontrakten, Codeannotationen (Assertions) zu generieren, welche zur Laufzeit eine manuell erstellte Implementierung überwachen. Wird eine Spezifikation zur Laufzeit verletzt, so wird dies von den generierten Assertions erkannt

und es kann eine entsprechende Ausnahmebehandlung durchgeführt werden. Da aus Sicht des Service Providers eine öffentliche Servicebeschreibung mit visuellen Kontrakten auch eine modellbasierte Anforderung bzw. Spezifikation darstellt, kann mit dem Model-Driven Monitoring sichergestellt werden, dass sich ein manuell implementierter Service gemäß seiner öffentlichen Servicebeschreibung verhält (vgl. Abbildung 2.4).

Abbildung 2.5 zeigt ausgehend von der Designebene beispielhaft die Einbettung unserer visuellen Kontrakte in einen modellbasierten Softwareentwicklungsprozess. In der Design-Phase wird von einem Softwareentwickler ein Modell des zu entwickelnden Systems erstellt. Das Modell besteht aus einem Klassendiagramm, welches durch visuelle Kontrakte ergänzt wird. Das Klassendiagramm beschreibt die statischen Aspekte des Systems. Die visuellen Kontrakte spezifizieren das Verhalten der Operationen.

Beim Übergang vom Design zur Implementierung im Softwareentwicklungsprozess wird in unserem Ansatz aus den Modellen Code generiert. Der generierte Code besteht aus zwei Teilen. Erstens werden aus den Klassen des Design-Klassendiagramms Java-Klassen generiert. Die Java-Klassen enthalten die spezifizierten Attribute und Methoden (jedoch keine Methodenrümpfe) sowie weitergehende Attribute und vollständige Methoden zur Verwaltung der Assoziationen.

Zweitens werden zur Annotation der Operationen aus den visuellen Kontrakten JML-Assertions generiert. Die Java Modeling Language (JML) [Iow, BCC+05, LBR05, LC04] erlaubt die Spezifikation von Java-Methoden mit Vor- und Nachbedingungen. Dazu wird der Java-Code mit JML-Assertions annotiert. Die Assertion-Sprache von JML basiert auf Java-Ausdrücken und folgt dem Beispiel der Design-by-Contract-Realisierung in Eiffel [Mey92b].

Mit den aus den visuellen Kontrakten generierten JML-Annotationen wird es möglich zu überprüfen, ob der manuell implementierte Code konsistent zu den Design-Modellen ist, d.h. konsistent zur Spezifikation ist. Der JML-Code zur Überprüfung der Konsistenz zwischen Design-Modellen und Implementierung muss sich zur Laufzeit transparent gegenüber dem restlichen Code der Anwendung verhalten. Es ist sicherzustellen, dass die Assertions keine Seiteneffekte haben, d.h. die Assertions selber dürfen den Systemzustand nicht beeinflussen. Dies stellen wir mit der gewählten Codegenerierung, die wir in Kapitel 4 im Detail beschreiben, sicher.

Die generierten Klassengerüste kann ein Programmierer in der Phase der Implementierung verwenden und den fehlenden Code zur Implementierung des Verhaltens der Anwendung integrieren, um eine vollständig lauffähige

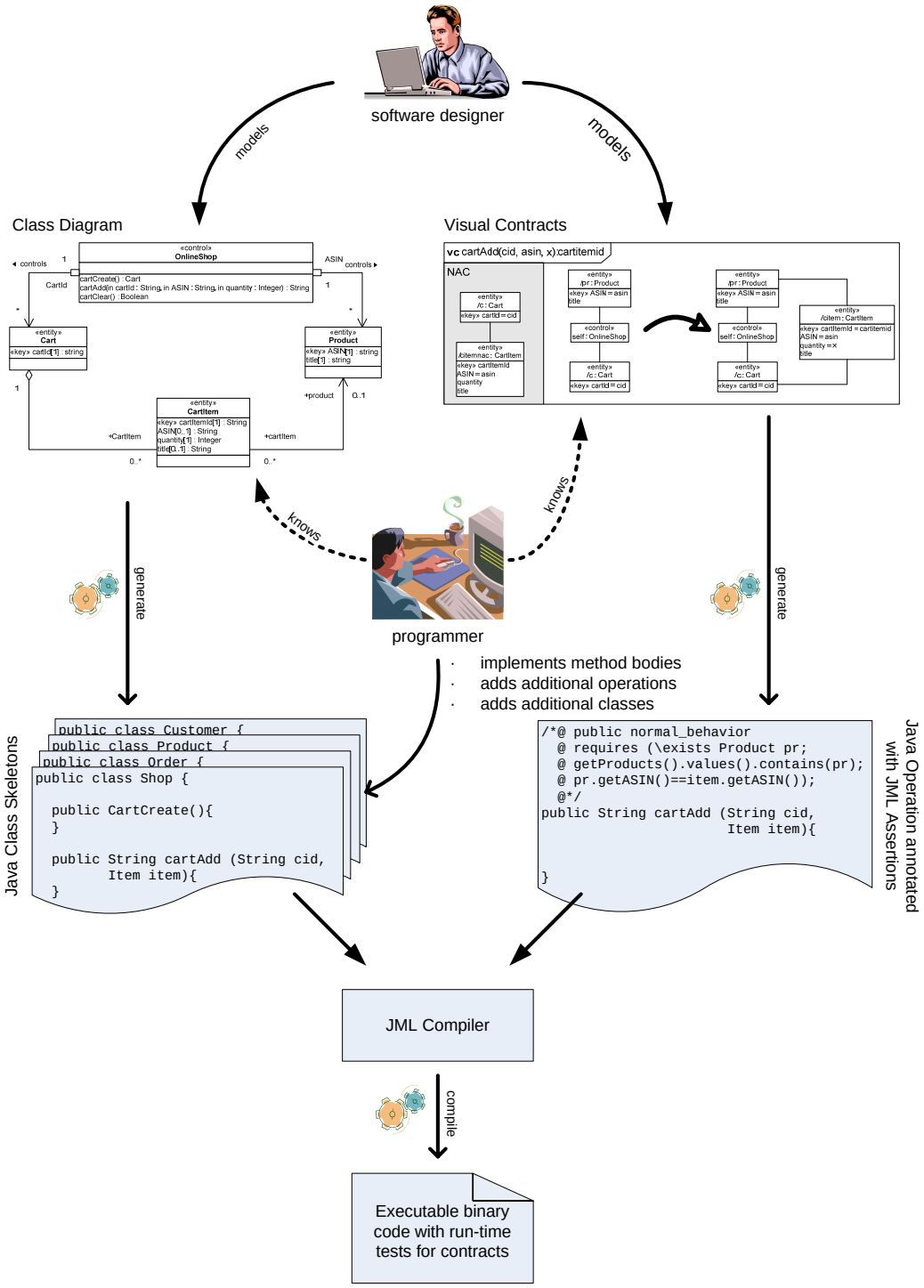


Abbildung 2.5: Model-Driven Monitoring im Softwareentwicklungsprozess

Anwendung zu erzeugen. Die Aufgaben für den Programmierer ergeben sich aus den Design-Modellen des Systems. Insbesondere muss der Programmierer bei der Implementierung der Methodenrümpfe die visuellen Kontrakte als Referenz für das Verhalten der Operationen verwenden. Weiterhin kann der Programmierer neue Operationen zu existierenden Klassen hinzufügen oder auch neue Klassen erzeugen. Allerdings darf der Programmierer in unserem Ansatz nicht die JML-Annotationen, d.h. die Kontrakte auf der Ebene der Implementierung, selbst verändern. Nur so kann sichergestellt werden, dass die JML-Kontrakte konsistent zu den visuellen Kontrakten bleiben. Weiterhin darf der Programmierer die automatisch generierten Attribute und Operationen nicht verändern. Diese Einschränkungen sind mit den Einschränkungen verschiedener Arbeiten aus dem Bereich des Model-Driven Development vergleichbar (siehe Abschnitt 3.1).

Wenn neue Anforderungen im Laufe des Softwareentwicklungsprozesses an das zu entwickelnde System gestellt werden, die neue Funktionalitäten in Form neuer Operationen erfordern, so ist diese neue Funktionalität zuerst mit Hilfe der visuellen Kontrakte zu spezifizieren bevor der Programmierer diese implementieren kann. Eine inkrementelle Codegenerierung ermöglicht jedoch die Integration der vorherigen manuellen Implementierungsarbeiten in den neu generierten Code. Wenn visuelle Kontrakte auf diese Weise verwendet werden, so ähnelt unsere Vorgehensweise agilen Entwicklungsansätzen oder Extreme-Programming-Ansätzen, wo ein Entwickler zuerst eine Menge von Testfällen zu erzeugen hat, bevor er den entsprechenden Code implementiert.

Die Integrität der visuellen Kontrakte auf der Ebene der Implementierung kann unterstützt werden indem die Java-Klassen und die JML-Annotationen in unterschiedlichen Dateien verwaltet werden, was von den existierenden JML-Implementierungen auch unterstützt wird. Dabei kann gleichzeitig dem Programmierer der Zugriff auf die Datei mit den JML-Assertions verboten werden. Die Programmierer müssen nicht notwendigerweise die JML-Annotationen sehen, da diese aufgrund ihrer Komplexität nicht intuitiv sind. Auf der Implementierungsebene haben JML-Konstrukte das gleiche Problem wie OCL-Constraints auf der Modellebene sobald komplexere Bedingungen ausgedrückt werden sollen. Die Programmierer sollen daher die visuellen Kontrakte als Ausgangspunkt für ihre Programmieraufgaben verwenden, da diese intuitiver als die textuellen Beschreibungen sind.

Der oben beschriebene Zusammenhang zwischen den visuellen Kontrakten auf Modellebene, den JML-Annotationen auf der Implementierungsebene und den Freiheiten, die einem Programmierer in unserem Ansatz zugestanden werden, wird unterstützt durch die Formalisierung unserer visuellen Kontrak-

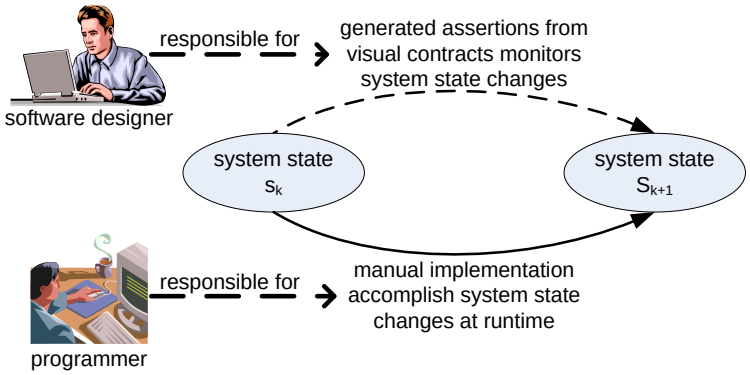


Abbildung 2.6: Verhalten einer mit visuellen Kontrakten entwickelten Anwendung zur Laufzeit

te mit dem Double-Pullback (DPB) Ansatz (vgl. Abschnitt 3.4). Ein visueller Kontrakt spezifiziert, welche Änderungen des Systemzustandes bei dem Aufruf einer Operation mindestens ausgeführt werden müssen. Weitere Veränderungen des Systemzustandes, die z.B. entstehen wenn ein Programmierer zusätzliche Objekte oder Attribute, die nicht Teil des visuellen Kontraktes sind, ändert, sind erlaubt. Die Klassen über welche die zusätzlich veränderten Objekte getypt sind oder die zusätzlich veränderten Attribute müssen nicht Teil des Design-Klassendiagramms sein. Mit unserer losen Interpretation der visuellen Kontrakte werden dem Programmierer die notwendigen Freiheiten zur Optimierung seiner Implementierung gegeben. Er kann zusätzliche Effekte bei der Ausführung einer Operation implementieren. Andererseits können zusätzliche negative Vor- oder negative Nachbedingungen in unseren visuellen Kontrakten verwendet werden, um die Freiheiten des Programmierers weiter einzuschränken. Aufgund der gewählten Semantik für unsere visuellen Kontrakte kommen wir den in Abschnitt 2.2 beschriebenen Anforderungen für einen durchgängigen Einsatz der Design-by-Contract-Konzepte in einem Softwareentwicklungsprozess nach.

Wenn ein Programmierer das Verhalten implementiert hat, kann der JML-Compiler verwendet werden, um ausführbaren Java-Bytecode zu erzeugen. Dieser Java-Bytecode enthält die Implementierung der Methodenrümpfe des Programmierers als auch Code, um die JML-Assertions zur Laufzeit zu überprüfen. Der Code zum Überprüfen der JML-Assertions und damit der Code zur Überprüfung der visuellen Kontrakte wird automatisch von dem JML-Compiler erzeugt. Dies führt zu einem Laufzeitverhalten des Systems wie in Abbildung 2.6 dargestellt. Die manuelle Implementierung des Programmierers ist für die Transformation des Systemzustandes verantwortlich. Die aus

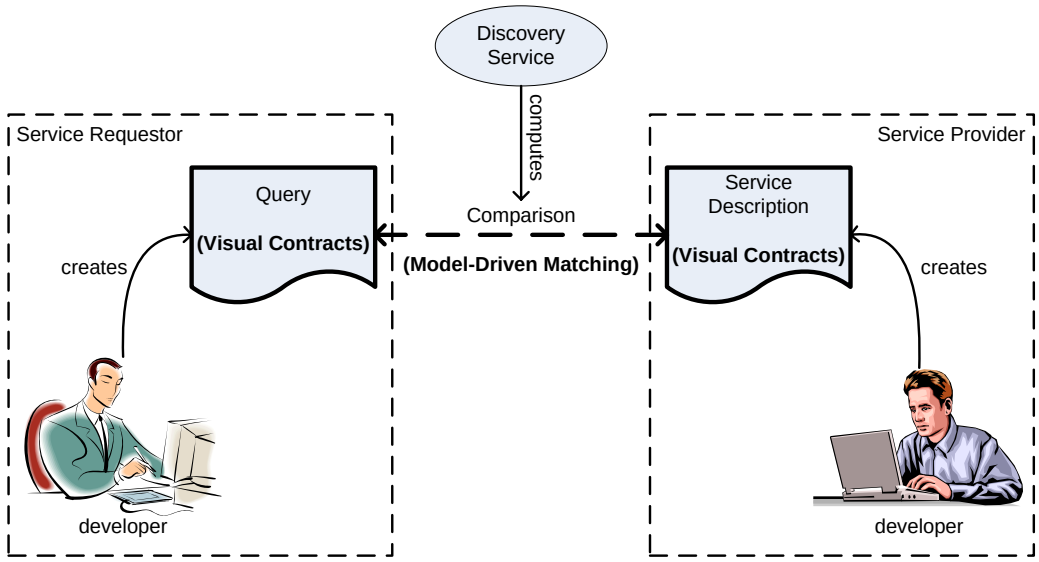


Abbildung 2.7: Model-Driven Matching

den JML-Assertions generierten Tests überprüfen die Vor- und Nachbedingungen während der Ausführung des Systems, d.h. sie überprüfen, ob das manuell implementierte Verhalten einer Operation den JML-Spezifikationen entspricht. Somit überprüfen wir, ob der manuell implementierte Code den visuellen Kontrakten aus dem Design entspricht, da die JML-Annotationen aus den visuellen Kontrakten generiert wurden. Das bedeutet, wir unterstützen die Überwachung (Monitoring) der Implementierung indem wir unsere visuellen Kontrakte nach JML übersetzen.

2.6 Model-Driven Matching

Model-Driven Matching ist eine neue Strategie zur Unterstützung einer semantischen, automatisierten Suche eines Service Requestors nach einem Service. Wir verwenden visuelle Kontrakte zur Beschreibung der Semantik eines Service und einer Suchanfrage (vgl. Abbildung 2.7). Das Matching-Konzept legt fest, wie eine Servicebeschreibung mit einer Suchanfrage zu vergleichen ist, um zu bestimmen, ob ein Service die Anforderungen eines Service Requestors erfüllt. Mit der Verwendung von Modellen berücksichtigen wir insbesondere die Rolle des Softwareentwicklers bei der Suche nach einem geeigneten Service und ermöglichen auf der Provider-Seite eine Einbettung der Erstellung der semantischen Servicebeschreibungen in einen Softwareentwick-

lungsprozess. Eine Transformation der Modelle in eine existierende Sprache zur Repräsentation der Semantik von Web Services, ermöglicht eine automatisierte Berechung unseres Matching-Konzeptes (z.B. von einem Discovery Service). Weiterhin kommen wir mit dieser Transformation der Forderung des World Wide Web Consortiums nach, die öffentlichen Schnittstellen eines Web Service mit XML-basierten Standards zu beschreiben.

Im Kontext serviceorientierter Architekturen muss ein Matching-Konzept verschiedene Anforderungen erfüllen. Die Anforderungen ergeben sich aus einer wirtschaftlichen, technologischen und einer aus einem Softwareentwicklungsprozess getriebenen Sichtweise [HHL05, HHL04]. Wirtschaftlich gesehen können mit serviceorientierten Architekturen B2B- und B2C-Lösungen für den Handel auf einer globalen Ebene über das Web realisiert werden. Einfache Services wie, Authentifizierung, Bezahlungen oder Versand können von verschiedenen Service Providern angeboten werden und zu neuen, attraktiven Produktangeboten für Kunden kombiniert werden.

Einer der zentralen Charakteristiken des Electronic Commerce im Web ist seine enorme Dynamik. Der Grund ist, dass der Konkurrenzdruck im Web außerordentlich stark ist und dass es sich um einen internationalen Markt handelt, dessen Größe nicht einschätzbar ist [Pre98, ELW03, ELW06]. Als Ergebnis ergibt sich eine extreme Notwendigkeit, anderen auf dem Markt mit innovativen Produkten bzw. Services zuvorzukommen, um sich einen Marktanteil zu sichern [RPHB02]. Die Möglichkeit aufgrund des impliziten Auslieferungsmechanismus des Webs einen Service sofort zu veröffentlichen verstärkt diese Notwendigkeit noch. Eine Technologie zur semantischen Beschreibung von Web Services muss diese Dynamik insbesondere bei der Verwaltung existierender Services unterstützen. Ein Service Provider muss adhoc eine geeignete Beschreibung für einen neuen, innovativen Web Service erstellen und veröffentlichen können.

Ein globaler Markt in dem sich die Services und damit auch die Beschreibung der Services ständig weiter entwickeln erfordert auch ein flexibles Matching-Konzept zum Vergleich von Servicebeschreibungen und Suchanfragen. Ein flexibles Matching bestimmt einen geeigneten Service für eine Suchanfrage, so dass beide Parteien — Service Provider und Service Requestor — zufrieden sind. Das Matching darf dabei jedoch nicht auf eine einfache Identitätsprüfung beschränkt werden, da ansonsten ein Service Requestor seine Anfrage der Entwicklung der angebotenen Services anpassen muss. Weiterhin gehen wir davon aus, dass ein Service Provider seine Services sehr genau (wenn auch nicht vollständig) beschreibt, während ein Service Requestor eine Suchanfrage etwas allgemeiner formuliert, um eine größere Auswahl an in Wettbewerb

miteinander stehender Services zu finden. Auf jeden Fall muss das Matching alle zu einer Suchanfrage passenden Services finden.

Technologisch gesehen entstehen bei diesem Szenario verschiedene Probleme, die typisch für verteilte Systeme sind. Die allgemein anerkannte Lösung, um die Kommunikation zwischen zwei verteilten Komponenten zu unterstützen, ist die Verwendung von Standards bzw. Standardprotokollen, an welche sich die an der Kommunikation beteiligten Komponenten halten müssen. Das bedeutet, dass vom technischen Standpunkt aus gesehen, es am sinnvollsten ist, wenn sich alle Parteien auf einen Standard einigen und diesen für einen möglichst langen Zeitraum einhalten. Dieser Aspekt der Beständigkeit steht jedoch im Widerspruch zu den wirtschaftlichen Anforderungen. Ein geeigneter Ansatz für die Suche nach Web Services muss einen geeigneten Mittelweg finden. Er muss auf einem Standard basieren, aber dennoch flexibel genug sein, um verschiedene Services beschreiben zu können.

Mit dem Model-Driven Matching kombinieren wir die visuellen Kontrakte mit Ontologien. Eine Ontologie beschreibt eine standardisierte Begriffswelt, die sowohl dem Service Provider als auch dem Service Requestor bekannt ist. In unserem Ansatz verwenden wir Klassendiagramme bestehend aus Klassen und Assoziationen zur Beschreibung einer Ontologie[3]. Die geforderte Flexibilität erreichen wir durch den Einsatz der visuellen Kontrakte zur Beschreibung der Semantik eines Service oder einer Suchanfrage. Zur Berechnung eines Matchings müssen der visuelle Kontrakt der Servicebeschreibung und der Suchanfrage über derselben Ontologie getypt sein (vgl. Abbildung 2.8).

Ein weiterer Aspekt, welcher zu berücksichtigen ist, ist die Entwicklung eines Service. Auf der einen Seite müssen die Technologien zum Aufbau einer serviceorientierten Architektur und damit auch die zur Beschreibung eines Service oder einer Suchanfrage verwendeten Technologien unabhängig von der zur Implementierung eines Service oder Clients verwendeten Plattform sein. Auf der anderen Seite ist zu beachten, dass die semantische Beschreibung eines Service keine eigenständige Einheit ist, die unabhängig von den Methodiken, die zur Entwicklung eines Service eingesetzt werden, verwendet werden kann. Das bedeutet insbesondere, dass die Entwickler eines Service verstehen müssen, wie die Beschreibung eines Service mit seiner Implementierung zusammenhängt. Hierzu sind die Vorgehensweisen zur Erstellung der Servicebeschreibungen in heutige Softwareentwicklungsprozesse zu integrieren, um eine einfache und konsistente Verwendung von Technologien zum Aufbau serviceorientierter Architekturen zu ermöglichen. Ohne eine vernünf-

[3]Eine detaillierte Disskussion über die Bedeutung von Ontologien und die Eignung von Klassendiagrammen zur Beschreibung von Ontologien findet sich in Abschnitt 6.2.1

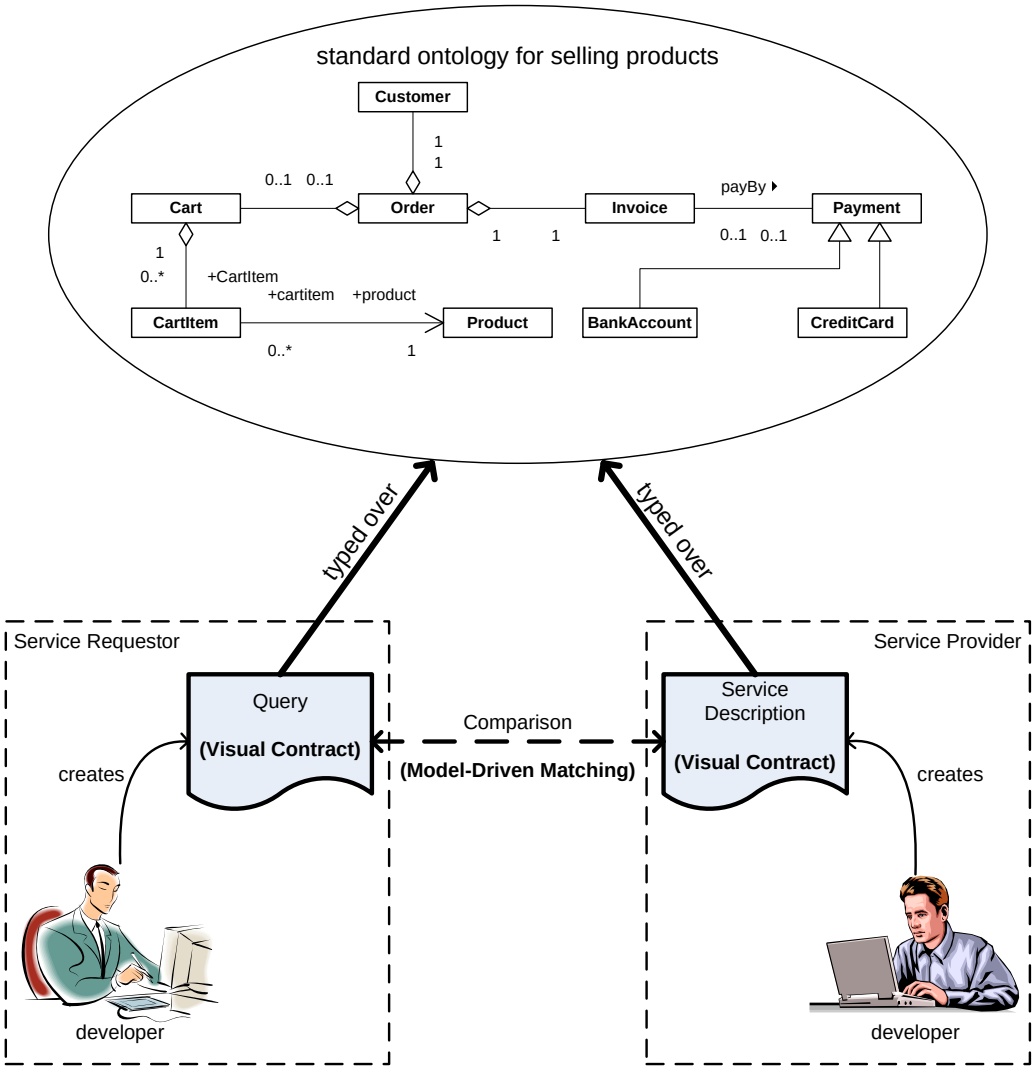


Abbildung 2.8: Model-Driven Matching mit visuellen Kontrakten und Ontologien

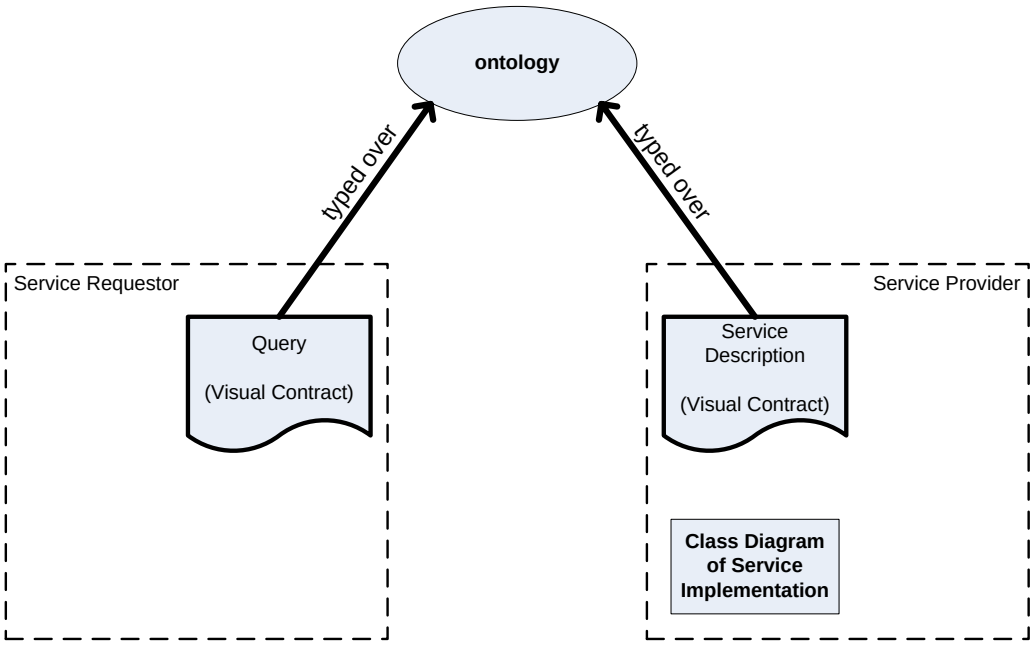


Abbildung 2.9: Ontologien und Klassendiagramm des Service Provider in einer SOA

tige Integration, werden die öffentlichen Servicebeschreibungen schnell inkosistent zur Implementierung des Systems (vgl. auch Abschnitt 2.3), womit die öffentlichen Servicebeschreibungen und damit auch das Matching bedeutungslos werden.

Unser Formalismus zur Beschreibung der Semantik eines Service (visuelle Kontrakte) berücksichtigt jedoch bereits Möglichkeiten zur Erstellung dieser Beschreibungen, da wir eine Notation verwenden, die sich an aktuelle Notationen im Bereich des Software Engineering anlehnt. Im Folgenden skizzieren wir, wie die Erstellung der semantischen Servicebeschreibungen in einen modellbasierten Softwareentwicklungsprozess eingebettet werden kann. Hierzu müssen wir den Fokus von den Modellen zur Beschreibung der Semantik eines Service oder Suchanfrage auf die Modelle zur Implementierung eines Service verschieben. Damit besteht das Szenario aus zwei verschiedenen Vokabularen, die mit Klassendiagrammen dargestellt werden (Abbildung 2.9).

Auf der einen Seite ist für das oben beschriebene Matching-Konzept ein gemeinsames Datenmodell zwischen Service Provider und Service Requestor notwendig. Eine standardisierte Ontologie stellt ein gemeinsames Datenmodell dar, das es erlaubt semantische Servicebeschreibungen und Suchanfragen miteinander in Beziehung zu setzen.

Auf der anderen Seite existiert auf Provider-Seite ein Klassendiagramm, das die Implementierung beschreibt. Das Klassendiagramm der Implementierung muss nicht notwendigerweise identisch sein mit der Ontologie. Im Folgenden betrachten wir, wie ein Service Provider semantische Beschreibungen für seinen Web Service erstellen kann, wenn er eine Anwendung neu entwickelt.

Wir gehen von der Annahme aus, dass ein Service Provider beim Start der Entwicklung einer neuen Anwendung das Klassendiagramm der standardisierten Ontologie seiner Anwendungsdomäne als Klassendiagramm für seine Implementierung übernimmt. Die Klassen der Ontologie werden somit in der Analyse und Design-Phase eines Softwareentwicklungsprozesses zu Entity-Klassen, welche die Implementierung des zu entwickelnden Systems beschreiben. Damit stellt der Service Provider sicher, dass er die vorgegebenen Fachkonzepte einer Domäne in seiner Implementierung verwendet, was eine Basis für wieder verwendbare, fachliche Softwarekomponenten darstellt.

Diese Vorgehensweise wird unterstützt durch die Entwicklung domänenspezifischer Standards, die es Softwareherstellern und Domänenexperten erlauben, zu einem Standard konforme Softwaresysteme auf einer objektorientierten Ebene zu realisieren. Prominente Beispiele aus der Versicherungsbranche sind zum Beispiel das objektorientierte, fachliche Referenzmodell der VAA (Versicherungs Anwendungs Architektur) [Ges01] oder der US-amerikanischen Organisation ACORD [ACO06]. Ein Ziel dieser Projekte ist die Entwicklung eines unternehmensübergreifenden objektorientierten, fachlichen Referenzmodells, welches erlaubt die Fachlichkeit eines Unternehmens strukturiert zu beschreiben. Damit soll eine Basis für wieder verwendbare fachliche Softwarekomponenten geschaffen werden.

Danach kann die Anwendung gemäß der in Abschnitt 2.5 beschriebenen modellbasierten Vorgehensweise entwickelt werden. Die einzige Einschränkung ist, dass in der Analyse oder Design-Phase die durch die Ontologie vorgegebene Strukturierung beibehalten wird, d.h. die Klassen und Assoziationen aus der Ontologie nicht gelöscht werden. Ein Softwareentwickler kann der Spezifikation der Anwendung jedoch neue Klassen, Attribute, Operationen und Assoziationen hinzufügen. Damit wird sichergestellt, dass das Klassendiagramm der Ontologie ein Subgraph des Design-Klassendiagramms der Implementierung ist (vgl. auch Abschnitt 2.2).

Während des Designs der Anwendung, kann ein Softwareentwickler das Verhalten der Anwendung spezifizieren. Hierzu kann er, wie in Abschnitt 2.5 beschrieben, das Verhalten einzelner Operationen mit visuellen Kontrakten spezifizieren. Insbesondere muss ein Softwareentwickler das Verhalten

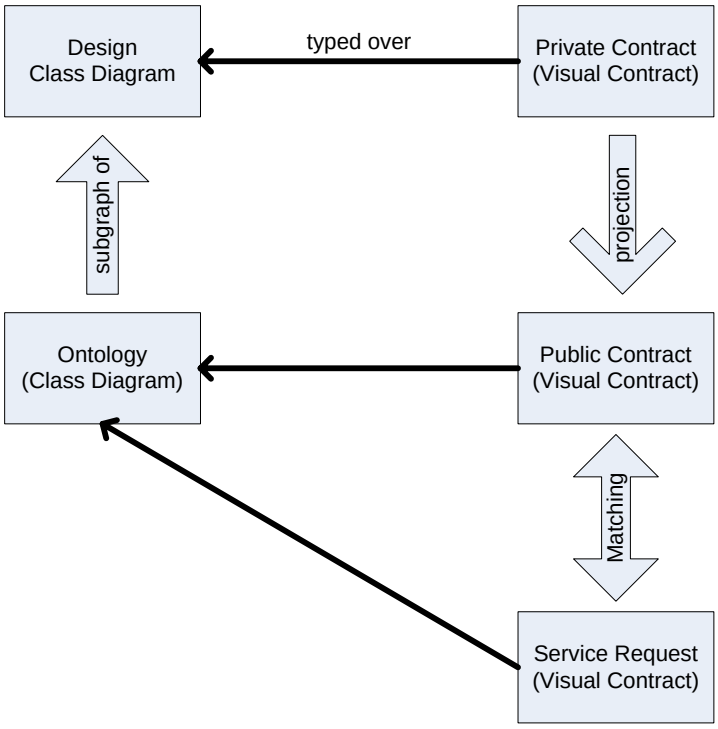


Abbildung 2.10: Bestimmung öffentlicher Servicebeschreibungen aus privaten Kontrakten

der Operationen, die als Service veröffentlicht werden sollen, mit visuellen Kontrakten spezifizieren. Diese visuellen Kontrakte sind über dem Design-Klassendiagramm getypt und werden als *private Kontrakte* (*private contracts*) bezeichnet.

Sollen einzelne Operationen des Systems als Service veröffentlicht werden, so müssen für diese semantische, öffentliche Servicebeschreibungen erstellt werden. Gemäß unserem Model-Driven-Matching-Konzept müssen diese visuellen Kontrakte über der Ontologie getypt sein. Die visuellen Kontrakte der öffentlichen Servicebeschreibungen bezeichnen wir als *öffentliche Kontrakte* (*public contracts*).

Zur Erstellung der öffentlichen Kontrakte können die privaten Kontrakte, die während des Design im Softwareentwicklungsprozess erstellt wurden, wieder verwendet werden. Die Ontologie stellt einen Teilgraph des Klassendiagramms aus dem Design dar. Zur Bestimmung der öffentlichen Kontrakte müssen alle Elemente, Attribute und Links, die nicht Teil der Ontologie sind, aus den Vor- und Nachbedingungen der privaten Kontrakte entfernt werden.

Das Ergebnis dieser Projektion sind öffentliche Kontrakte, die über der Ontologie getypt sind und für unser Model-Driven Matching verwendet werden können (vgl. Abbildung 2.10).

Mit dem hier geschilderten Vorgehen kann die Erstellung der öffentlichen Servicebeschreibungen für einen Service in einen modellbasierten Softwareentwicklungsprozess integriert werden. Mit den so erstellten öffentlichen Servicebeschreibungen kann mit unserem Matching-Konzept überprüft werden, ob ein Service den Anforderungen eines Service Requestors genügt. Weiterhin wird aufgrund des Einsatzes des Model-Driven-Monitoring-Konzeptes sichergestellt, dass die Implementierung eines Service sich korrekt gegenüber seiner öffentlichen Servicebeschreibung verhält.

2.7 Fazit

In diesem Kapitel haben wir unseren Ansatz zur kontraktbasierten Modellierung, Implementierung und Suche in serviceorientierten Architekturen überblicksweise beschrieben. Der Ansatz erlaubt eine modellbasierte, semantische Beschreibung und die Überprüfung der Korrektheit von Services sowie einen Vergleich von Servicebeschreibungen (vgl. Abbildung 2.11).

Zur Beschreibung der Semantik von Services werden Kontrakte auf Modellebene eingesetzt. Die Vor- und Nachbedingungen dieser visuellen Kontrakte werden mit UML-Composite-Structure-Diagrammen beschrieben, die über einem Klassendiagramm getypt sind. Mit der Verwendung von UML-Diagrammen benutzen wir eine Notation, die Softwareentwicklern vertraut ist.

Die Einbettung der visuellen Kontrakte in einen modellbasierten Softwareentwicklungsprozess und die Generierung von Runtime-Assertions aus den visuellen Kontrakten ermöglicht ein Model-Driven Monitoring: Die Überwachung der Korrektheit einer manuell erstellten Implementierung eines Service zur Laufzeit.

Weiterhin unterstützen wir die Suche eines Service Requestors nach geeigneten Services in einer serviceorientierten Architektur mit unserem Model-Driven Matching. Hierzu ermöglichen wir auch einem Service Requestor die Verwendung der visuellen Kontrakte zur Erstellung einer Suchanfrage und definieren einen automatisierbaren Kompatibilitätsbegriff zum Vergleich einer öffentlichen Servicebeschreibung mit einer Suchanfrage. Da unser Model-Driven Matching voraussetzt, dass die Servicebeschreibung und die Suchan-

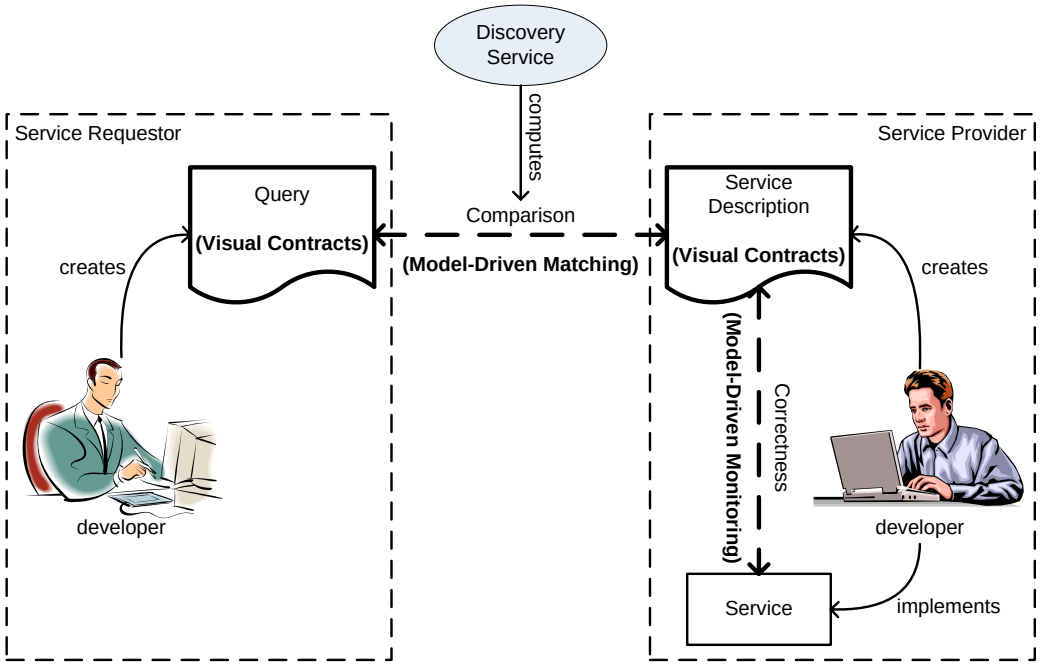


Abbildung 2.11: Visuelle Kontrakte zur Unterstützung der Modellierung, Implementierung und Suche in einer SOA

frage über demselben Klassendiagramm getypt sind, haben wir eine Vorgehensweise für die Erzeugung öffentlicher Servicebeschreibungen, die gegenüber der Ontologie einer Domäne getypt sind, eingeführt. Diese Vorgehensweise basiert auf einer Einbettung der Ontologie in den Softwareentwicklungsprozess zur Entwicklung eines Service.

Durch die Verwendung dieses Ansatzes, der sowohl die Korrektheit einer manuell erstellten Implementierung als auch den semantischen Vergleich von Suchanfragen und Servicebeschreibungen unterstützt, kann die Qualität serviceorientierter (oder komponentenbasierter) Anwendungen verbessert werden. In den folgenden Kapiteln dieser Arbeit detaillieren wir die visuellen Kontrakte, das Model-Driven Monitoring und das Model-Driven Matching.

2.8 Aufbau der Arbeit

Die vorliegende Arbeit gliedert sich wie in Abbildung 2.12 gezeigt in drei Teile und ist auf neun Kapitel aufgeteilt:

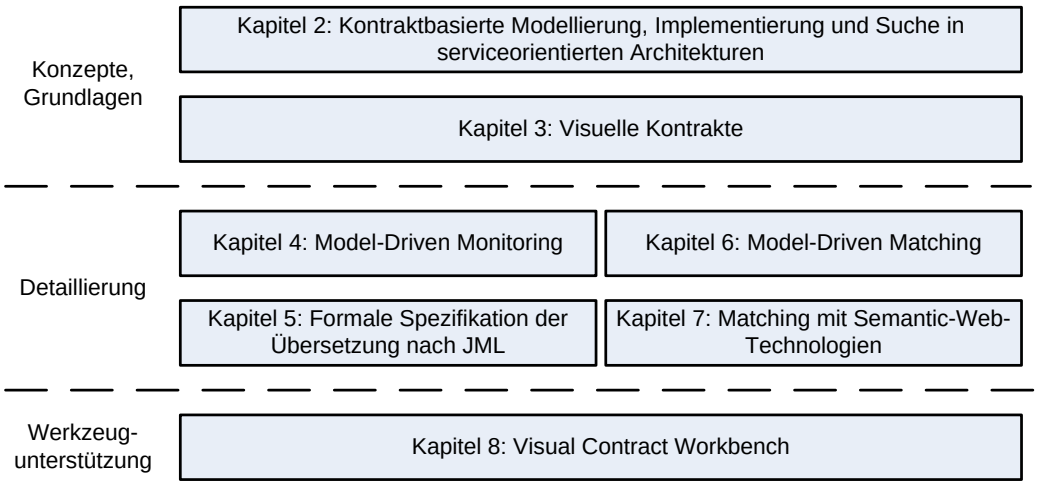


Abbildung 2.12: Überblick über die Kapitel der Arbeit

In diesem Kapitel (Kapitel 2), *Kontraktbasierte Modellierung, Implementierung und Suche in serviceorientierten Architekturen*, haben wir einen Gesamtüberblick über unser Vorgehen gegeben.

In Kapitel 3, *Visuelle Kontrakte*, definieren wir eine Sprache zur semantischen Beschreibung von Services. Ein visueller Kontrakt besteht aus einem Paar von UML-Composite-Structure-Diagrammen zur Beschreibung der Vor- und Nachbedingungen eines Services. Die Sprachdefinition wird festgehalten mit einem Metamodell, das an dem Metamodell der UML 2.0 ausgerichtet ist.

In Kapitel 4, *Model-Driven Monitoring*, beschreiben wir eine Vorgehensweise, die es erlaubt zu prüfen, ob sich eine manuell erstellte Implementierung eines Service oder einer Operation gemäß der gegebenen Spezifikation mit einem visuellem Kontrakt verhält. Hierzu beschreiben wir eine Übersetzung der visuellen Kontrakte in die Java Modeling Language (JML), einer Design-by-Contract-Erweiterung für Java.

In Kapitel 5, *Formale Spezifikation der Übersetzung nach JML*, geben wir eine formale Spezifikation der Übersetzung visueller Kontrakte nach JML an. Die Entwicklung einer formalen und genauen Spezifikation ist der Schlüssel zu einer automatisierten Übersetzung.

In Kapitel 6, *Model-Driven Matching*, zeigen wir, wie die visuellen Kontrakte zur Verwaltung von Services eingesetzt werden können. Hierzu beschreiben wir als erstes, wie ein visueller Kontrakt von einem Service Requestor zur Beschreibung seiner Anforderungen eingesetzt werden kann. Danach definieren wir ein Matching-Konzept, das es erlaubt die Anforderungen eines Service

Requestors mit einer Servicebeschreibung zu vergleichen.

In Kapitel 7, *Matching mit Semantic-Web-Technologien*, zeigen wir, wie unser Matching-Konzept mit Hilfe von Semantic-Web-Sprachen umgesetzt werden kann. Hiermit kommen wir der Anforderung des World Wide Web Consortiums nach, dass die öffentlichen Schnittstellen eines Web Service mit XML-basierten Sprachen zu beschreiben sind.

In Kapitel 8, *Visual Contract Workbench*, beschreiben wir die Implementierung eines Editors, der die Erstellung von visuellen Kontrakten erlaubt. Der Editor unterstützt die in dieser Arbeit vorgeschlagene Methodik. Zum einen erlaubt er eine automatisierte Generierung von JML-Kontrakten, um die Korrektheit einer manuellen Implementierung zu überprüfen. Zum anderen erlaubt der Editor eine automatisierte Übersetzung der visuellen Kontrakte in eine Semantic-Web-Sprache und ermöglicht damit die Suche nach Services auf Basis der in Kapitel 7 erläuterten Umsetzung des Matching-Konzepts.

Zum Abschluss der Arbeit erfolgt in Kapitel 9 eine Zusammenfassung und Diskussion der erreichten Ziele.

2.9 Bibliographische Hinweise

Die Ergebnisse dieser Arbeit sind zum Teil bereits in diversen Publikationen veröffentlicht worden. Die grundlegende Idee, Web Services oder Web-Anwendungen mit Graphtransformationsregeln zu beschreiben, ist in [HL03a] beschrieben worden. In diesem Artikel ist eine Modellierungsmethode entwickelt worden, die ausnutzt, dass es sich bei Web Services beziehungsweise Web-Anwendungen um reaktive Informationssysteme handelt. Zur Integration von Reaktivität und Zustandstransformation sind *graphische Reaktionsregeln* entwickelt worden. Graphische Reaktionsregeln sind Transformationsregeln, die über einen Request ausgelöst werden. Die Tragfähigkeit dieses Ansatzes wurde in einer Bachelorarbeit überprüft [Bau04].

Danach wurde der Ansatz der graphischen Reaktionsregeln hin zu den in dieser Arbeit verwendeten visuellen Kontrakten verfeinert. Dabei wurden die zwei in dieser Arbeit verfolgten Techniken zur Erhöhung der Qualität einer serviceorientierten Architektur auf Basis semantischer Beschreibungen mit visuellen Kontrakten zunächst in unterschiedlichen Publikationen weiter detailliert.

Die Übersetzung visueller Kontrakte nach JML einschließlich der Anwendung

dieses Ansatzes in einem Softwareentwicklungsprozess ist zum ersten Mal in [LSE05] beschrieben worden. Eine ausführliche Beschreibung der Übersetzung findet sich in [HL06, ELSH06].

Das in dieser Arbeit vorgestellte Matching-Konzept zum Vergleich einer Servicebeschreibung mit einer Suchanfrage ist zum ersten Mal in [HHL03] beschrieben worden. Der Ansatz ist in weiteren Veröffentlichungen detailliert worden [HHL04, HHL05, ELS05a]. Während wir in dieser Arbeit eine mengentheoretische Definition des Matchings angeben, ist in einer weiteren Doktorarbeit in der Arbeitsgruppe von Prof. Dr. Engels eine Formalisierung dieses Matching-Konzeptes auf der Basis von Graphtransitionen, die durch den Double-Pullback-Ansatz definiert sind, erarbeitet worden [HCL04, HC05, Che06]. Eine prototypische Implementierung des Matching-Konzeptes auf Basis existierender Semantic-Web-Technologien ist in einer Diplomarbeit entwickelt worden [Sev03].

In [HL04] haben wir zum ersten Mal veröffentlicht, wie die Qualität einer serviceorientierten Architektur unter Berücksichtigung folgender zwei Aspekte erhöht werden kann: zum einen muss sichergestellt werden, dass die Implementierung eines Service seiner (veröffentlichten) Spezifikation entspricht und zum anderen ist ein Matching notwendig, um den Bedarf eines Requestors mit dem (spezifizierten) Angebot eines Providers zu vergleichen. Die finale Konzeption unseres Ansatzes und damit die Integration der zuvor in unterschiedlichen Publikationen veröffentlichten Ergebnisse ist in [HHL05, ELS05b] veröffentlicht worden.

Eine erste Version eines prototypischen, graphischen Editors zur Modellierung visueller Kontrakte ist in einer Studienarbeit entwickelt worden [Wag05]. Dieser Editor ist im weiteren Verlauf dieser Arbeit weiter ausgearbeitet worden. Unter anderem wurde er um Komponenten zur Generierung von JML-Annotationen aus visuellen Kontrakten als auch Komponenten zur Generierung von Semantic-Web-Repräsentationen der visuellen Kontrakte ergänzt. Eine Beschreibung des Editors und seiner Architektur ist in [LES06] veröffentlicht worden.

Aufbauend auf diesen zentralen Veröffentlichungen der Ergebnisse unserer Arbeit, haben wir in [HL03b, HL04] beschrieben, wie visuelle Kontrakte zum Testen von Web Services eingesetzt werden können. Eine ausführliche Evaluation unseres Ansatzes erfolgte in Kooperation mit der sd&m AG. Die Ergebnisse dieser Evaluation finden sich in [EGJ+06, LRE+06].

Kapitel 3

Visuelle Kontrakte

In diesem Kapitel führen wir visuelle Kontrakte ein. Mit den visuellen Kontrakten erlauben wir den Einsatz von Design-by-Contract-Methodiken in einem modellbasierten Softwareentwicklungsprozess. Nach einer Beschreibung verwandter Arbeiten führen wir unsere visuellen Kontrakte mit einem Fallbeispiel über einen Online Shop ein. Zur Definition der Syntax der visuellen Kontrakte erläutern wir eine geeignete UML-Metamodellerweiterung. Die Semantik der visuellen Kontrakte erläutern wir auf der Basis von Graphtransformationen.

3.1 Verwandte Arbeiten

Der UML-Standard [OMG03a] enthält seit der Version 1.1 eine textuelle, formale Sprache zur Annotation von UML-Modellen: Die Object Constraint Language (OCL) [OMG03b]. Im Gegensatz zu den grafischen Modellierungssprachen der UML, handelt es sich bei OCL um eine textuelle Sprache. Die Konsequenz ist, dass OCL-Ausdrücke eine vollständig andere Notation als die Diagramme der UML besitzen. Die Object Constraint Language unterstützt die Definition von präzisen Constraints auf den Instanzen von UML-Modellen, welche nicht mit den UML Diagrammen selbst ausgedrückt werden können. Im UML Standard wird OCL verwendet, um präzise Wohlgeformtheitsregeln auf der Metamodellebene zu spezifizieren. Die Arten der Constraints, die mit OCL ausgedrückt werden können, enthalten Invarianten für Klassen und Typen als auch Vor- und Nachbedingungen für Operationen. Auch wenn es verschiedene Ansätze gibt, welche die OCL nutzen, um daraus ausführbaren Code zu erstellen, so mangelt es der OCL immer noch an einer

einfach zu benutzenden Repräsentation. Infolgedessen ist OCL selbst in Prozessen oder Unternehmen, die UML intensiv verwenden, von eingeschränktem Nutzen. Die Komplexität der OCL und die Schwierigkeiten der Integration einer reinen textbasierten Sprache wie OCL mit Diagrammen sind wichtige Gründe für diese Situation.

Die Business Object Notation (BON) [Ner92, WN94, WD98] ist eine Methode und grafische Notation für die objektorientierte Analyse und das Design. Die Notation ist im weitesten Sinne vergleichbar mit der UML [PO99]. Design by Contract ist ein zentrales Konzept in BON. Es wird verwendet, um das Verhalten der Elemente einer Klasse oder der Interaktion von Klassen zu spezifizieren. Im Gegensatz zur UML ist ein Kontrakt Bestandteil der Notation einer Klasse, d.h. Klasse und Kontrakt werden nicht separat dargestellt. Jedoch verwendet auch der BON-Ansatz nur eine textuelle Notation, basierend auf der First-Order-Prädikatenlogik, zur Beschreibung der Kontrakte. Dadurch ergeben sich im BON-Ansatz die gleichen Nachteile wie bei UML und OCL.

In anderen Ansätzen wird ein visuelles Gegenstück zur OCL entwickelt. Diese Ansätze basieren auf einer Visualisierung mit einer mengentheoretischen Semantik. Kent et. al. haben einen visuellen Formalismus zur Definition von Constraints in objektorientierten Modellen entwickelt. Als erstes haben sie Constraint-Diagramme [Ken97b, Ken97a, KG98] entwickelt. Constraint-Diagramme basieren auf Venn-Diagrammen [HD96], um die mengentheoretische Semantik von OCL-Constraints darstellen zu können. Später haben Howse et. al. aufbauend auf diesem Ansatz Spider-Diagramme [KH99, GHK02] entwickelt, um automatische Reasoning-Mechanismen auf den Diagrammen zu ermöglichen.

Visual OCL [BKPPT00, BKPPT01, BKPPT02, KTW02a, KTW02b] ermöglicht eine graphische Repräsentation der OCL. Der Ansatz basiert auf einem angepassten OCL-Metamodell [RG99] und erweitert Kollaborationsdiagramme der UML 1.4 [OMG01b][1] zur visuellen Beschreibung von Navigationsausdrücken der OCL. Die Visualisierung basiert auf einer Übersetzung der OCL-Constraints in Graphregeln und Transformation Units, um eine präzise Semantik für OCL-Constraints definieren zu können.

[1] Ab UML 2.0 heißen die Kollaborationsdiagramme Kommunikationsdiagramme (Communication Diagrams). Neben der Namensänderungen hat sich auch das entsprechende Metamodell und ihre Ausdrucksmächtigkeit grundlegend geändert. Ab der UML 2.0 stellen die Kommunikationsdiagramme nur noch eine alternative Repräsentation von UML-Sequenzdiagrammen dar. Sie sind damit nicht zur Beschreibung von Navigationsausdrücken geeignet.

In den beiden zuletzt genannten Ansätzen werden textuelle logische Ausdrücke in die Diagramme eingebunden, was zu einer hybriden Notation von OCL-Constraints führt. Weiterhin unterscheiden sich die Diagramme stark von den weitläufig in Softwareentwicklungsprozessen eingesetzten Diagrammen der UML, d.h. die Softwareentwickler müssen neben der OCL noch eine weitere neue visuelle Sprache lernen. Damit sind auch diese Ansätze nur von eingeschränktem Nutzen. Die Ansätze besitzen weiterhin aufgrund der Verwendung von OCL eine hohe Komplexität und sind aufgrund der verwendeten, neuen Diagrammarten nur schwer in heutige UML-basierte Softwareentwicklungsprozesse zu integrieren.

Wir suchen daher eine intuitive, visuelle Notation, die auf den bekannten Notationen der UML 2.0 beruht, um die Einarbeitungszeit eines Softwareentwicklers möglichst gering zu halten. Mit der Notation soll es möglich sein, das Verhalten von typischen Operationen oder Services in einer serviceorientierten Architektur zu beschreiben. Eine Turing-Vollständigkeit ist jedoch nicht gefordert, da wir aus unseren Modellen keinen vollständig, ausführbaren Code generieren wollen. Wir wollen lediglich die Überwachung der Korrektheit einer manuellen Implementierung und eine semantische Suche ermöglichen.

3.2 Modellierung von Kontrakten

In diesem Abschnitt werden die visuellen Kontrakte zur Spezifikation von Operationen anhand eines Fallbeispiels eingeführt. Dabei werden die verschiedenen Aspekte und Sprachkonstrukte der visuellen Kontrakte beschrieben. Als Fallbeispiel dient ein Online Shop, der die Verwaltung von Warenkörben zulässt. Das Fallbeispiel haben wir in Anlehnung an die Operationen der Amazon Web Services [Ama05b] zur Verwaltung von Warenkörben entwickelt.

3.2.1 Online Shop

In dieser Arbeit wird ein Online Shop, der seinen Warenkatalog über das Internet zur Verfügung stellt, als durchgehendes Fallbeispiel verwendet. In den dargestellten Beispielen konzentrieren wir uns auf die Operationen des Online Shops zur Verwaltung eines Einkaufswagens bzw. Warenkorbs (*Shopping Cart*) für einen Kunden. Warenkörbe werden nahezu von allen Online

Shops im Internet, wie z.B. Amazon[2], Otto[3] oder Fleurop[4], zur Verwaltung von Bestellungen verwendet.

Bei der Gestaltung unseres Fallbeispiels haben wir uns an den angebotenen Daten und Diensten des Online Shops Amazon orientiert, die nicht nur über die bekannte Weboberfläche sondern auch als Web Service zur Verfügung gestellt werden. Insbesondere die von Amazon angebotenen Web Services bieten einen detaillierten Einblick in die öffentlichen Schnittstellen und die über diese Schnittstellen kommunizierten Datenstrukturen. Aufbauend auf diesen Informationen kann ein realistisches Fallbeispiel konstruiert werden. Im Folgenden werden die Amazon Web Services näher erläutert.

Die Amazon Web Services (AWS) bieten Softwareentwicklern direkten Zugriff auf die Amazon-Technologieplattformen und die Produktdaten von Amazon. Die als Web Service angebotene Funktionalität reicht von der Suche nach Informationen zu einem bestimmten Produkt über die Verwaltung von Warenkörben bis hin zum Abfragen des Status einer Transaktion. Die Amazon Web Services können über REST (Representational State Transfer, auch „XML über HTTP“ oder „XML/HTTP“ genannt) [Fie00] oder auch über SOAP [GHM+03a, GHM+03b] angesprochen werden.

Die Funktionen zum Verwalten von Warenkörben sind Teil der Amazon E-Commerce Services (ECS) [Ama05a]. Der ECS-Warenkorb ist ein Mechanismus zur Verwaltung von Warenposten, die ein Kunde kaufen möchte. Die maschinenlesbaren Spezifikationen stehen als getrennter Download in den Formaten WSDL [CMRW05, CHL+05] und XML Schema [TBMM04, BPM04] zur Verfügung. Anwendungen, die einem Kunden den Einkauf von Produkten bei Amazon ermöglichen, funktionieren immer nach dem gleichen Prinzip. Die Anwendung stellt dem Kunden eine Auswahl von Produkten, die bei Amazon bestellt werden können, zur Verfügung. Danach wird ein Warenkorb bei Amazon erstellt, welcher von der Anwendung verwaltet werden kann. Diesem Warenkorb können beliebig Produkte hinzugefügt werden. Zur Abwicklung der Bezahlung und Beendigung der Bestellung wird der Kunde dann auf die Webseiten von Amazon weitergeleitet[5].

[2] http://www.amazon.com/

[3] http://www.otto.de

[4] http://www.fleurop.com/

[5] Einen Überblick über verschiedene Anwendungen findet man auf den Amazon Web Services Webseiten. Eine sehr interessante Anwendung, welche auch gleichzeitig zeigt wie flexibel die Amazon Schnittstellen sind, ist von der Firma Hive Group (`http://www.hivegroup.com/amazon4.html`). Hier kann ein Anwender mit einer innovativen visuellen Benutzeroberfläche die Spezifikationen der bei Amazon angebotenen Produkte vergleichen.

In unserem Online-Shop-Beispiel haben wir von verschiedenen Details abstrahiert. So können mit dem ECS z.B. zwei Arten von Warenkörben verwaltet werden. Zum einen kann ein Warenkorb erstellt werden, der später an Amazon zur Durchführung der Bestellung übergeben wird. Zum anderen kann parallel ein zweiter Warenkorb verwaltet werden, der es erlaubt, Warenposten für eine spätere Sitzung zusammenzustellen. In unseren Beispielen werden wir nur die Verwaltung von einem einzelnen Warenkorb betrachten.

Auch Funktionen oder Mechanismen zur Gewährung der Sicherheit werden nicht in unser Beispiel übernommen. So generiert der Amazon Web Service z.B. neben einer ID für einen Warenkorb auch gleichzeitig einen Token mit einer zeitlich begrenzten Lebensdauer (*HMAC*). Beide Informationen müssen bei jedem Aufruf der Operationen für den Zugriff auf einen bestehenden Warenkorb mit angegeben werden. Wenn eine vergebene HMAC eine gewisse Zeit nicht mehr verwendet wurde, so kann nicht mehr auf den zugehörigen Warenkorb zugegriffen werden. Damit soll sichergestellt werden, dass ein Warenkorb nicht längere Zeit nur auf der Clientseite verwaltet wird, d.h. nach Möglichkeit sollen die Clients den lokalen und den Warenkorb bei Amazon synchron halten.

Insgesamt besteht unser Online Shop aus den folgenden vier Operationen:

CartCreate: Mit dieser Operation wird ein *Remote Shopping Cart* erstellt. Wenn ein Warenkorb erstellt wird, so wird eine *CartID* erstellt, die dem Client mitgeteilt wird, damit dieser auf den neu erstellten Warenkorb zugreifen kann.

CartAdd: Mit dieser Operation können einem bestehenden Warenkorb weitere verfügbare Warenposten hinzugefügt werden.

CartOrder: Mit dieser Operation kann ein Kunde einen existierenden Warenkorb bestellen.

CartClear: Mit dieser Operation werden alle Warenposten aus einem bestehendem Warenkorb entfernt.

Die Detaillierungen in den folgenden zwei Abschnitten sind zum Teil aus der WSDL-Datei der ECS abgeleitet worden. Zum Teil mussten wir die Informationen ergänzen, um ein vollständiges Beispiel für einen Online Shop zu erhalten. Wie bereits weiter oben beschrieben, enthält eine WSDL-Datei die Beschreibung einer öffentlichen Schnittstelle. Die Schnittstelle besteht aus Operationen inklusive deren Signaturen. Die in den Signaturen verwendeten Datenmodelle werden in einer WSDL-Datei mit einem XML Schema beschrieben. Aus diesen Informationen konnten einige Klassen für unser Fall-

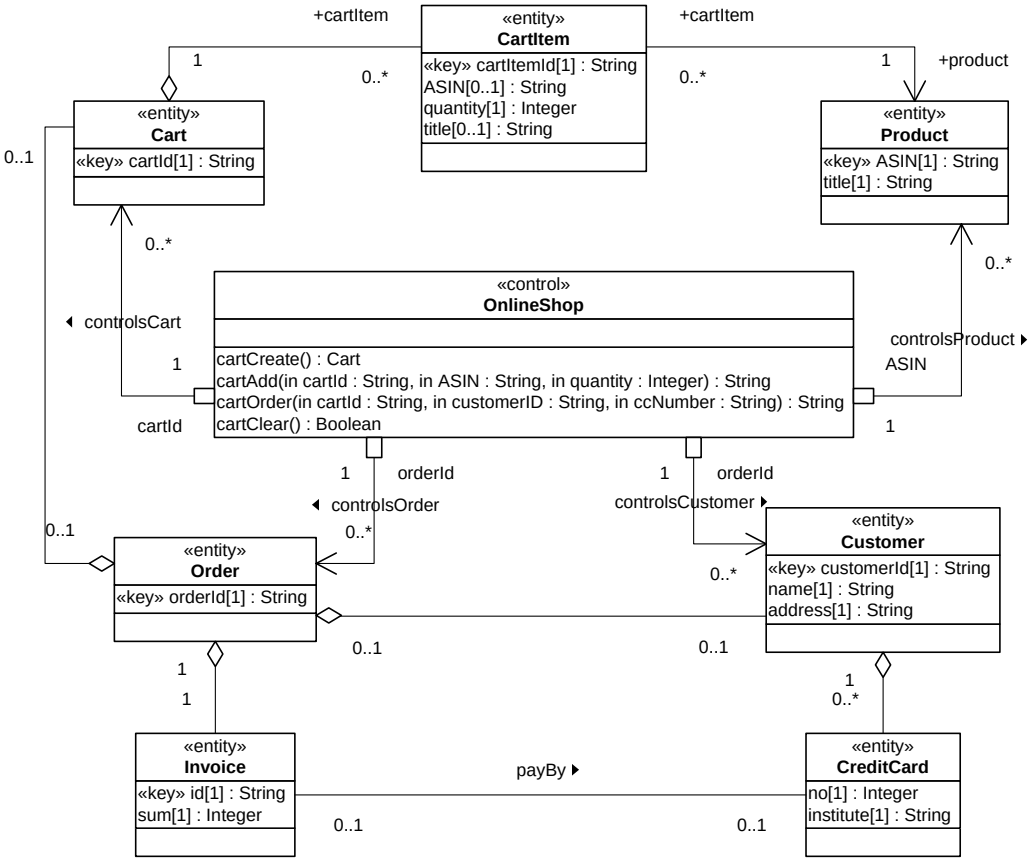


Abbildung 3.1: Klassendiagramm Online Shop

beispiel abgeleitet werden. Die visuellen Kontrakte haben wir zum Teil aus der von Amazon bereitgestellten Prosa-Dokumentation ihrer Schnittstellen erstellt. Auch hier mussten wir Ergänzungen vornehmen, um ein vollständiges Beispiel zu erhalten.

3.2.2 Design-Klassendiagramm des Online Shops

Wir verwenden UML-Klassendiagramme für die Darstellung von statischen Aspekten in unseren Design-Modellen. Abbildung 3.1 zeigt ein Klassendiagramm unseres Online Shops.

Die Stereotypen `control` und `entity` werden gemäß den Vorgaben im Unified Process [JBR99, Kru99] verwendet. Jeder dieser Stereotypen weist einer Klasse eine bestimmte Rolle in der Implementierung zu. Instanzen von Control-Klassen kapseln die Logik einer Anwendung und koordinieren an-

dere Objekte. Entity-Klassen werden zur Modellierung von langlebigen oder persistenten Informationen verwendet. Mit dem Stereotyp `key` können die Attribute einer Klasse ausgezeichnet werden (für Details und die Definition eines entsprechenden UML-Profils siehe [Amb03]). Ein Schlüsselattribut hat für eine Menge von Objekten desselben Typs einen eindeutigen Inhalt.

Die Klasse `OnlineShop` ist mit verschiedenen Entity-Klassen des Systems über qualifizierende Assoziationen verbunden. Die zur Qualifizierung ausgewählte Attributmenge (z.B. `ASIN` an Assoziation `controlsProduct`) am Assoziationsende mit dem Rechteck ist eine Teilmenge der Attribute der Klasse am anderen Ende der Assoziation. In Kombination mit den Schlüsselattributen kann so über eine qualifizierende Assoziation eindeutig auf ein spezifisches Objekt zugegriffen werden.

Die Klassen in dem Klassendiagramm aus Abbildung 3.1 selbst haben folgende Aufgaben. Die Klasse `OnlineShop` ist eine Kontrollklasse. Sie enthält die Operationen, die ein Client zur Verwaltung eines Warenkorbs benötigt. Die Klasse `Cart` stellt einen Warenkorb dar, der eine Menge von `CartItems` (Warenposten) enthält. Ein `Cart` kann eindeutig über seine `cartId` identifiziert werden. Ein `CartItem` referenziert ein Produkt, welches von Amazon angeboten wird. Weiterhin speichert ein `CartItem` die bestellte Anzahl eines Produktes, den Titel, sowie die ASIN eines Produktes. Die Klasse `Product` enthält auch die ASIN und den Titel als Attribute. Über die ASIN (Amazon Standard Identification Number) kann ein Produkt eindeutig identifiziert werden. Zu einem existierenden Warenkorb kann für einen Kunden (`Customer`) eine Bestellung (`Order`) erstellt werden. Die Bestellung und der Kunde können über einen Schlüssel eindeutig identifiziert werden. Weiterhin gehört zu einem Auftrag eine Rechnung (`Invoice`), die mit einer Kreditkarte, die einem Kunden zugeordnet ist, bezahlt werden kann.

3.2.3 Visuelle Kontrakte

In unserem Ansatz werden Klassendiagramme auf der Designebene ergänzt durch visuelle Kontrakte, um das Verhalten einer Operation zu spezifizieren. Dafür wird es notwendig den Fokus der Modellierung auf einzelne Operationen zu verschieben. Für eine intuitive, visuelle Beschreibung des Verhaltens einer Operation unterstützen unsere visuellen Kontrakte eine funktionale Sicht, die statische und dynamische Aspekte integriert. Ein visueller Kontrakt beschreibt die Effekte der Ausführung einer Operation auf den Zustand des modellierten Systems. Wie bei der objektorientierten Modellierung

üblich kann der Zustand eines Systems mit Objektdiagrammen festgehalten werden.

Im Folgenden detaillieren wir die Operationen unseres Online Shops mit visuellen Kontrakten, um die Verwendung der visuellen Kontrakte zu erklären. Abbildung 3.2 zeigt einen visuellen Kontrakt zur Beschreibung des Verhaltens der Operation `cartCreate`. Diese Operation erzeugt einen neuen Warenkorb. Ein visueller Kontrakt ist eingebettet in einen *Frame*, welcher aus einem Titel (*heading*) und einem Arbeitsbereich (*content area*) besteht. Frames sind ein Konzept der UML 2.0 Notationen für Diagramme (vgl. Anhang A in [OMG04]). Sie erlauben die Definition eines wieder verwendbaren Kontexts für Diagramme der UML. Der Titel ist ein String, welcher in ein Rechteck mit einer abgeschnittenen Ecke in den oberen linken Bereich eines Frames eingebettet ist. Das Schlüsselwort `vc` weist auf den Typ des Diagramms hin (in diesem Fall auf einen visuellen Kontrakt). Dem Schlüsselwort folgt der Operationsname, welcher durch den visuellen Kontrakt spezifiziert wird. Dem Operationsnamen kann wiederum eine Parameterliste und ein Rückgabewert, wie sie im Klassendiagramm für die Operation spezifiziert wurden, folgen. Die Parameterliste ist eine sortierte Menge von Variablen oder konkreten Werten. Der Rückgabewert ist ebenso eine Variable oder ein konkreter Wert. Die Variablen aus der Parameterliste und dem Rückgabewert können in dem visuellen Kontrakt im Arbeitsbereich des Frames wieder verwendet werden. Mit einem Frame kann ein visueller Kontrakt somit einer Operation aus dem Klassendiagramm eindeutig zugewiesen werden. Eventuell ist dem Operationsnamen im Titel des Frames ein Klassenname voranzustellen, um die Zuweisung eindeutig zu machen.

Ein visueller Kontrakt in dem Arbeitsbereich eines Frames besteht im Wesentlichen aus einem Paar von Kompositionsstrukturdiagrammen (engl. *composite structure diagram*). Jedes der Diagramme ist getypt über dem Design-Klassendiagramm. Die einfache, intuitive Interpretation eines visuellen Kontraktes ist, dass alle Modellelemente, welche auf der linken Seite eines visuellen Kontraktes vorhanden sind (Repräsentation der Vorbedingung) vor der Ausführung der Operation im System vorhanden sein müssen und alle Modellelemente, welche auf der rechten Seite eines Kontraktes (Repräsentation der Nachbedingung) vorhanden sind, müssen nach der Ausführung der Operation im System vorhanden sein. Durch einen Vergleich der linken und rechten Seite eines visuellen Kontraktes ergibt sich, dass alle Modellelemente, welche nur auf der linken Seite vorhanden sind, beim Ausführen der Operation gelöscht werden. Alle Objekte, welche nur auf der rechten Seite eines visuellen Kontraktes vorhanden sind, werden erzeugt. Modellelemente, die

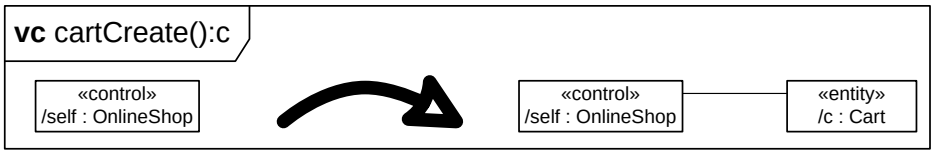


Abbildung 3.2: Verhalten der Methode `cartCreate`

auf beiden Seiten eines visuellen Kontraktes vorhanden sind, werden beibehalten, jedoch können sich die Inhalte der Attribute zwischen der Vor- und Nachbedingung unterscheiden.

Falls mehrere Modellelemente desselben Typs in einem visuellen Kontrakt verwendet werden, so müssen diese mit einem Rollennamen versehen werden, welcher vom Typ durch einen Doppelpunkt getrennt wird. Ansonsten können auch unbenannte Modellelemente verwendet werden.

Negative Vor- oder Nachbedingungen werden innerhalb des Frames mit einem grauen Rechteck hinterlegt. Wenn sich ein graues Rechteck links von der Vorbedingung befindet, so spezifiziert dieses Elementstrukturen, welche vor der Ausführung der Operation nicht erlaubt sind (negative Vorbedingung, vgl. Abbildung 3.3). Wenn sich ein graues Rechteck rechts von der Nachbedingung befindet, so spezifiziert es Elementstrukturen, die nach der Ausführung der Operation nicht erlaubt sind (negative Nachbedingung).

Ist ein visueller Kontrakt einer Operation zugeordnet, so müssen alle Elemente auf der linken oder rechten Seite eines visuellen Kontraktes von dem Element `self` — das Element, welches die in einem visuellen Kontrakt spezifizierte Operation ausführt — aus erreichbar sein. Nur so kann sichergestellt werden, dass zur Laufzeit die Vor- und Nachbedingungen auch überprüft werden können. Andere Werkzeuge, die Kontrakte bzw. Graphtransformationen unterstützen, wie z.B. PROGRES [Zün96] oder AGG [Tae04] machen derartige Einschränkungen nicht. Jedoch sind in diesem Fall die Navigations- und Kontrollstrukturen, die bei einer Codegenerierung aus einem Klassendiagramm berechnet und für die Bestimmung eines Matchings zur Laufzeit verwendet werden können (vg. Abschnitt 4.2), nicht ausreichend. Weiterhin wird in einem objektorientierten Programm eine Operation von einem Objekt ausgeführt, d.h. ein Element `self` ist in einem objektorientierten Programm immer gegeben.

Abbildung 3.2 zeigt einen visuellen Kontrakt zur näheren Spezifikation des Verhaltens der Operation `cartCreate`, die einen neuen Warenkorb (`Cart`) erzeugt. Gemäß dem visuellen Kontrakt kann die Methode immer ausgeführt

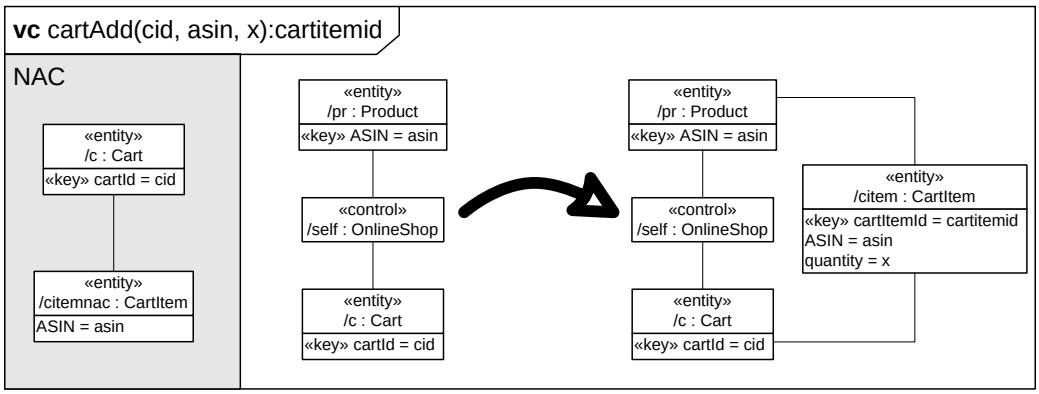


Abbildung 3.3: Verhalten der Methode `cartAdd`

werden, da als Vorbedingung lediglich das Element `self` angegeben ist. Als Effekt erzeugt die Operation ein neues Element vom Typ `Cart` und einen Link zwischen dem Element `self` und dem neuen Element vom Typ `Cart`. Die Variable `c` wird sowohl im Titel des Frames als auch als Rollenbezeichner des neu erzeugten Elementes `Cart` verwendet, d.h. das neu erzeugte Element ist auch der Rückgabeparameter der Operation `cartCreate`.

Abbildung 3.3 zeigt einen komplexeren visuellen Kontrakt, welcher die Operation `cartAdd` beschreibt. Diese Operation fügt einem existierenden Warenkorb einen weiteren Warenposten in Form eines `CartItem` hinzu. Im Gegensatz zu dem visuellen Kontrakt aus Abbildung 3.2 sind die Variablen in der Parameterliste und der Rückgabewert keine Referenzen auf Rollen. Stattdessen werden mit diesen Variablen die Inhalte der Attribute von verschiedenen Elementen des visuellen Kontraktes festgelegt. Für eine erfolgreiche Ausführung der Operation `cartAdd` muss das Kontextelement `self` ein Element vom Typ `Product`, welches ein Attribut `ASIN` mit dem Inhalt `asin` besitzt und ein Element vom Typ `Cart`, welches ein Attribut `cartID` mit dem Inhalt `cid` besitzt, kennen. Die Variablen werden zur Laufzeit durch den Aufruf der Operation durch einen Client an konkrete Werte gebunden.

Der visuelle Kontrakt aus Abbildung 3.3 enthält als Teil der Vorbedingung weiterhin eine negative Vorbedingung (*negative Anwendungsbedingung — NAC*). In dieser negativen Vorbedingung müssen entweder alle Elemente vom Kontextelement `self` aus erreichbar sein oder die Elemente müssen von einem Element aus erreichbar sein, welches Teil der Vorbedingung ist und in der Vorbedingung von `self` aus erreichbar ist. Die negative Vorbedingung aus Abbildung 3.3 besagt, dass das bestellte Produkt (bestimmt durch die `ASIN`) nicht bereits zuvor in den Warenkorb eingefügt worden sein darf. Wenn die

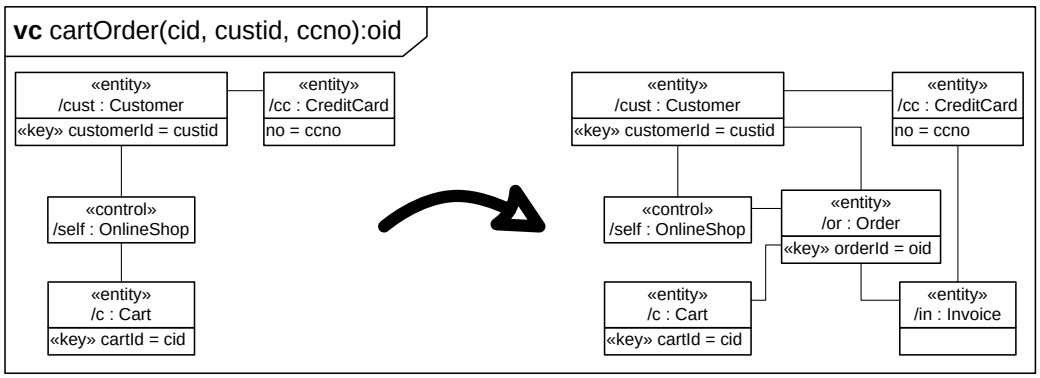


Abbildung 3.4: Verhalten der Methode `cartOrder`

Operation erfolgreich ausgeführt wird, so wird dem Element `Cart` ein neues `CartItem` hinzugefügt, welches das entsprechende Produkt referenziert.

Abbildung 3.4 zeigt wiederum einen komplexeren visuellen Kontrakt, welcher die Operation `cartOrder` detailliert. Diese Operation erstellt zu einem existierenden Warenkorb einen Auftrag (`Order`) für einen Kunden (`Customer`) des Online Shops. Der Kunde muss zur Erstellung eines Auftrages bereits Kreditkarteninformationen hinterlegt haben. Zur Bezahlung eines aktuellen Auftrages wird eine Kreditkarte über ihre Nummer (Attribut `no` des Elementes `cc:CreditCard`) ausgewählt. Neben dem Auftrag wird eine Rechnung erstellt, die über einen Link mit der ausgewählten Kreditkarte verknüpft ist. Weiterhin verlangt dieser visuelle Kontrakt die Löschung des Links zwischen `c:Cart` und `self`. Damit wird verhindert, dass nach der Ausführung der Operation `cartOrder` der ausgewählte `Cart` verändert werden kann, da nun die Vorbedingungen der hier beschriebenen Operationen zur Modifikation dieses `Cart`s nicht mehr gültig sein können. Sowohl in Abbildung 3.3 als auch in Abbildung 3.4 werden in einigen Elementen (z.B. `CreditCard` oder `Product`) nur die Attribute eingeblendet, die zur Beschreibung des Verhaltens einer Operation mit einem visuellen Kontrakt notwendig sind.

Abbildung 3.5 zeigt wie mit visuellen Kontrakten Mengen von Elementen, deren Kardinalitäten zur Entwicklungszeit nicht bekannt sind, modelliert werden können. Der visuelle Kontrakt spezifiziert die Operation `cartClear`, die alle Warenposten aus einem existierenden Warenkorb entfernt. In der Vorbedingung enthält der Link zwischen den Elementen `Cart` und `CartItem` ein Multiplizitätssymbol. Mit diesem Link wird spezifiziert, dass die Operation ausgeführt wird, wenn es eine Menge (die auch leer sein kann) von Elementen des Typs `CartItem` gibt, die von `Cart` aus erreichbar sind. Nach der Ausführung der Operation sind alle `CartItem`-Elemente aus der Vorbedingung, die

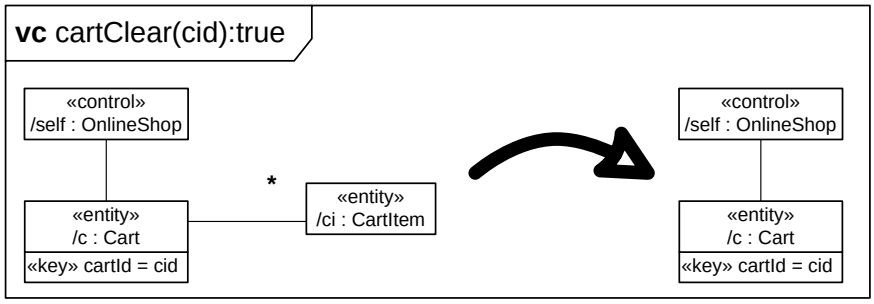


Abbildung 3.5: Verhalten der Methode `cartClear`

über einen Link mit `Cart` verbunden waren, einschließlich der entsprechenden Links gelöscht.

3.3 Metamodell für visuelle Kontrakte

In diesem Abschnitt geben wir eine präzise Syntaxdefinition für unsere visuellen Kontrakte an. Damit legen wir fest aus welchen Bestandteilen ein visueller Kontrakt besteht und wie diese kombiniert werden dürfen. Für visuelle Modellierungssprachen wie der UML setzt sich die Syntax in der Regel aus der konkreten und der abstrakten Syntax zusammen. In der konkreten Syntax wird die Notation der Sprache festgelegt. Die konkrete Syntax der visuellen Kontrakte haben wir im vorherigen Abschnitt anhand von Beispielen erläutert. Demgegenüber spiegelt die abstrakte Syntax die Struktur der Sprache wieder. In der UML-Spezifikation wird ein Metamodell zur Beschreibung der abstrakten Syntax verwendet. Das Metamodell selbst wird mit einem Klassendiagramm beschrieben. Die erlaubten Modelle einer Sprache sind Instanzen dieses Metamodells.

Auch wir verwenden zur Definition der abstrakten Syntax der visuellen Kontrakte ein Metamodell. Gleichzeitig kann dieses Metamodell als Vorlage bei der Entwicklung eines Werkzeuges für unseren Ansatz dienen, da existierende Frameworks zur Erstellung visueller Editoren, wie z.B. das Eclipse Modeling Framework [Ecl06b], mit Metamodellen arbeiten. Unser Metamodell stellt eine Erweiterung des UML 2.0 Metamodells dar. Im Folgenden werden als erstes die Pakete und die wichtigsten Klassen des UML 2.0 Metamodells, die für die Erweiterungen benötigt werden, vorgestellt. Bei den in unserem Ansatz verwendeten Klassendiagrammen handelt es sich um standardisierte UML 2.0 Klassendiagramme. Daher gehen wir nur auf die Metamodellerwei-

terungen für die visuellen Kontrakte ein. Für nähere Informationen zu dem Metamodell für Klassendiagramme verweisen wir hier lediglich auf die UML 2.0 Spezifikation [OMG04].

3.3.1 Verwendete UML 2.0 Pakete und Klassen

Das Metamodell für visuelle Kontrakte baut im wesentlichen auf den Elementen aus den Paketen `Kernel`, `InternalStructures`, `Collaborations` und `StructuredActivities` der UML 2.0 Spezifikation auf.

Das Paket `Kernel` enthält die grundlegenden UML-Modellierungskonzepte, einschließlich Klassen, Assoziationen und Paketen. Dieses Paket ist der zentrale Bestandteil der UML. Große Teile dieses Paketes sind aus der UML 2.0 Infrastructure [OMG03a] wieder verwendet worden, da hier ähnliche Konzepte verwendet werden.

In den Paketen `InternalStructures` und `Collaborations` werden die benötigten Elemente zur Definition des Metamodells für visuelle Kontrakte definiert. Diese Pakete sind Teil des Kapitels „Composite Structures“ der UML 2.0 Superstructure Spezifikation [OMG04]. *Composite-Structure-Diagramme* der UML 2.0 umfassen die aus der UML 1.4 [OMG01b] bekannten Composite-Context-Diagramme, Notationen für strukturelle Pattern, und Collaboration Roles. Mit Composite-Structure-Diagrammen können Ausschnitte einer Systemkonfiguration, die zur Erreichung eines bestimmten Ziels notwendig sind, beschrieben werden. Zur Beschreibung der Systemkonfiguration werden Rollen, die miteinander über Links interagieren können, verwendet. Zur Laufzeit können Instanzen diese Rollen einnehmen, um das in dem Composite-Structure-Diagramm bestimmte Ziel zu erreichen.

Das Paket `InternalStructures` bietet Mechanismen zur Strukturierung von verbundenen Elementen, welche bei der Initialisierung eines gruppierenden `Classifier` erzeugt werden. Eine derartige Strukturierung repräsentiert eine Dekomposition eines Classifiers und wird auch als seine interne Struktur bezeichnet.

In einem objektorientierten System kooperieren typischerweise Objekte miteinander, um das Verhalten des Systems zu realisieren. Das Verhalten kann durch eine Menge von zusammenarbeitenden Instanzen (typisiert durch einen `Classifier`), die durch das Senden von Signalen oder den Aufruf von Methoden miteinander kommunizieren, dargestellt werden. Jedoch ist es zur Erleichterung des Verständnisses der Modelle wichtig, dass nur die zur Erreichung eines Ziels notwendigen Aspekte eines `Classifier` einschließlich

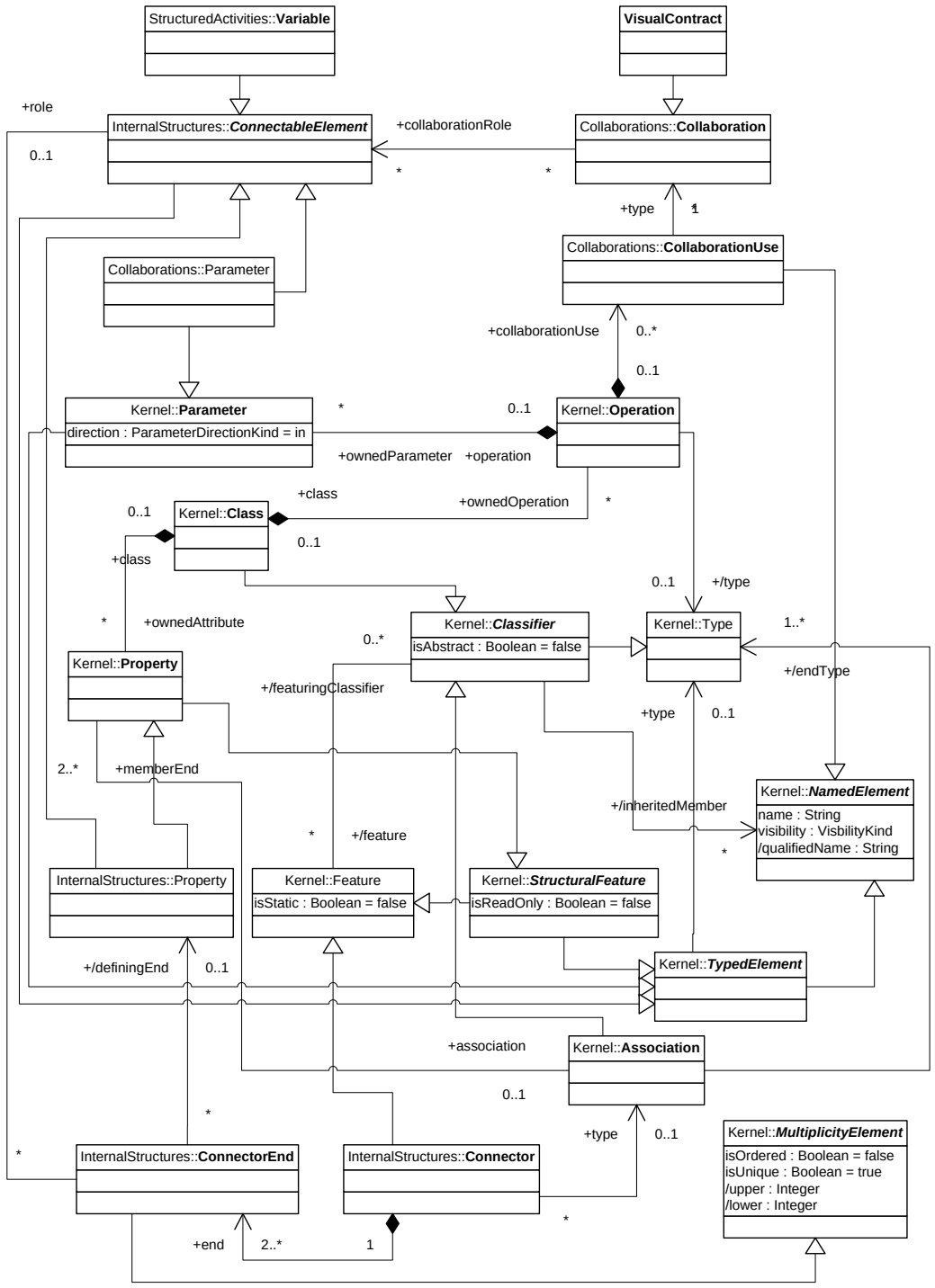


Abbildung 3.6: Relevante Teile des UML 2.0 Metamodells für visuelle Kontrakte

seiner Interaktionen beschrieben werden. Mit Hilfe Paketes `Collaborations` wird es möglich nur die relevanten Aspekte einer Kooperation zu beschreiben. Dazu werden Rollen angegeben, welche die an der Kooperation beteiligten Instanzen zur Laufzeit einnehmen. Mit `Collaborations` kann also beschrieben werden, wie die Elemente eines Modells kooperieren, um ein bestimmtes Verhalten zu realisieren. Bei den beteiligten Elementen kann es sich um Klassen und Objekte, Assoziationen und Links, Attribute und Operationen, Schnittstellen, Use Cases, Komponenten oder Nodes handeln. Das Verhalten kann ein Use Case, eine Operation, eine Menge von Operationen oder ein genereller Mechanismus des Systems sein. Mit anderen Worten, die in dem UML 2.0 Paket `Collaborations` beschriebenen Konzepte können auf verschiedenen Abstraktionsebenen eingesetzt werden.

Abbildung 3.6 zeigt die Metamodellklassen der UML 2.0 auf denen das Metamodell für visuelle Kontrakte aufbaut. Die Metamodellklassen stammen aus den Paketen `Kernel`, `InternalStructures`, `Collaborations` und `StructuredActivities`. Zur Verdeutlichung der Paketstruktur enthält eine Klasse als Bezeichner den Paketnamen, gefolgt von zwei Doppelpunkten und dem Namen der Klasse.

Eine `Collaboration` (Kollaboration) definiert eine Menge von Rollen, die zur Erfüllung einer bestimmten Aufgabe benötigt werden. Die Rollen einer Kollaboration werden von Instanzen eingenommen, die miteinander kommunizieren. Die Beziehungen, die zur Erfüllung einer bestimmten Aufgabe relevant sind, werden durch Konnektoren zwischen den Rollen dargestellt. Die Rollen einer Kollaboration definieren die Verwendung (Bedeutung) einer Instanz in dieser Kollaboration, während der Typ einer Rolle (festgelegt durch einen `Classifier`) die benötigten Charakteristiken (`Feature`) einer Instanz festlegt. Eine Instanz kann zur Laufzeit nur eine bestimmte Rolle einnehmen, wenn sie die durch den Classifier definierten Charakteristiken besitzt. Weiterhin müssen zur Laufzeit auch die Kommunikationspfade existieren, welche durch Konnektoren (`Connector`) zwischen den Rollen spezifiziert werden. Eine Kollaboration kann nicht direkt instanziiert werden. Stattdessen beschreibt sie eine Kooperation, die von Instanzen, welche die Rollen der Kollaboration zur Laufzeit einnehmen, erfüllt werden kann.

Die Rollen einer Kollaboration werden mit `ConnectableElement`s dargestellt. Eine Rolle wird von einer `Collaboration` referenziert. Gemäß der UML 2.0 Spezifikation erbt `ConnectableElement` von `TypedElement`. Ein `TypedElement` besitzt einen `Type`. Eine `Class` ist ein `Classifier`, welcher ein `Type` ist. Somit kann ein `ConnectableElement` in einer Kollaboration Rollen definieren, die über Klassen getypt sind.

`ConnectableElement`s können über einem `Connector` miteinander verknüpft werden. Ein `Connector` spezifiziert einen Kommunikationspfad, der die Kommunikation zwischen zwei oder mehr Instanzen ermöglicht. Ein Konnektor ist die Instanz einer Assoziation (im UML-Metamodell ist der Typ eines `Connector`s eine `Association`). Im Gegensatz zu Assoziationen, die Links zwischen beliebigen Instanzen der zugehörigen `Classifier` spezifizieren, beschreiben Konnektoren Verbindungen zwischen Instanzen, die spezifische in einer Kollaboration definierte Rollen einnehmen. Ein Konnektor kann zwei oder mehr Rollen referenzieren, wobei jede Rolle auch eine Menge von Instanzen repräsentieren kann. Dies wird in dem UML-Metamodell mit der Klasse `ConnectorEnd` möglich. `ConnectorEnd` erbt von der Klasse `MultiplicityElement`, wodurch wir den Rollen in unseren visuellen Kontrakten auch Multiplizitäten zuordnen können. Wir schränken die in visuellen Kontrakten erlaubten Multiplizitäten jedoch auf die praktisch relevanten Multiplizitäten `1` und `*` ein.

Eine `Collaboration` kann durch eine `CollaborationUse` an eine `Operation` oder `Classifier` gebunden werden. Damit beschreibt eine Kollaboration, wie ein `Classifier` oder eine Operation durch eine Menge von zusammenarbeitenden Instanzen realisiert werden kann. Wenn eine Kollaboration die Implementierung einer einzelnen Operation beschreibt, so enthält sie die Klassen, Assoziationen, Konnektoren und Rollen, welche von Objekten zur Laufzeit zur Realisierung des beschriebenen Verhaltens eingenommen werden müssen.

Das Paket `Collaborations` spezialisiert weiterhin die UML 2.0 Metamodellklassen zur Repräsentation von Parametern. Damit können die Parameter einer Operation im Rahmen einer `Collaboration` als Rollen wieder verwendet werden. Weiterhin sind auch Variablen (`Variable` erbt von `ConnectableElement`) Rollen und können somit wie andere Elemente in einer Kollaboration verwendet werden.

3.3.2 Erweiterungen des UML 2.0 Metamodells

Die Abbildungen 3.7 bis 3.10 zeigen das Metamodell für unsere visuellen Kontrakte. Die Metamodellklassen für visuelle Kontrakte haben wir dem Paket `VisualContracts` zugeordnet.

Abbildung 3.7 zeigt die Metamodellklassen für die Elemente, die in der Vor- oder Nachbedingung eines visuellen Kontraktes verwendet werden können. Alle neu definierten Elemente erben von der abstrakten UML-Klasse `ConnectableElement`. Die Elemente eines visuellen Kontraktes definieren so-

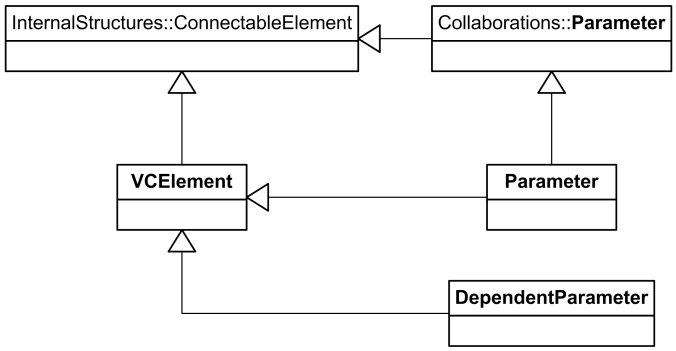


Abbildung 3.7: Elemente von visuellen Kontrakten (Paket VisualContracts)

mit Rollen, deren Typ eine Klasse sein kann. Die Rollen können zur Laufzeit von Instanzen eingenommen werden.

Abbildung 3.8 zeigt das Metamodell zur Strukturierung eines visuellen Kontraktes. Ein visueller Kontrakt spezialisiert eine `Collaboration` und kann somit einer Operation zugewiesen werden (über `CollaborationUse`). Visuelle Kontrakte bestehen aus Vor- und Nachbedingungen. Beide bestehen aus einer Menge von `VCElement`en, die über Konnektoren — da ein `VCElement` eine Spezialisierung von `ConnectableElement` ist — miteinander verbunden sind. Um zu unterscheiden, ob eine Rolle oder ein Konnektor zu einer Vor- oder Nachbedingung gehört, komponiert ein visueller Kontrakt (`VisualContract`) die Klassen `Precondition` und `Postcondition`. Beide Klassen sind Spezialisierungen der UML-Metamodellklasse `Collaboration`, d.h. sie können Rollen und Konnektoren definieren.

Ein visueller Kontrakt darf sowohl negative Vor- als auch negative Nachbedingungen enthalten. Diese negativen Bedingungen beschreiben nicht erlaubte Strukturen vor bzw. nach der Ausführung einer Operation und stellen eine Erweiterung der Vor- bzw. Nachbedingung dar. Im Metamodell werden diese negativen Bedingungen durch jeweils eine Assoziation von der Klasse `Precondition` bzw. `Postcondition` zur Klasse `NegativeCondition` berücksichtigt. Auch die Klasse `NegativeCondition` spezialisiert die Klasse `Collaboration` und kann somit Rollen und Konnektoren definieren.

Eine Assoziation zwischen der Klasse `VCElement` und einer der drei Klassen `Precondition`, `Postcondition` und `NegativeCondition` definiert eine Teilmenge der `ConnectableElements` (Klasse `VCElement` erbt von der Klasse `ConnectableElement`) der gesamten Kollaboration bzw. des gesamten visuellen Kontraktes. Die Vereinigung aller drei Submengen enthält alle Rollen, die an der Kollaboration beteiligt sind.

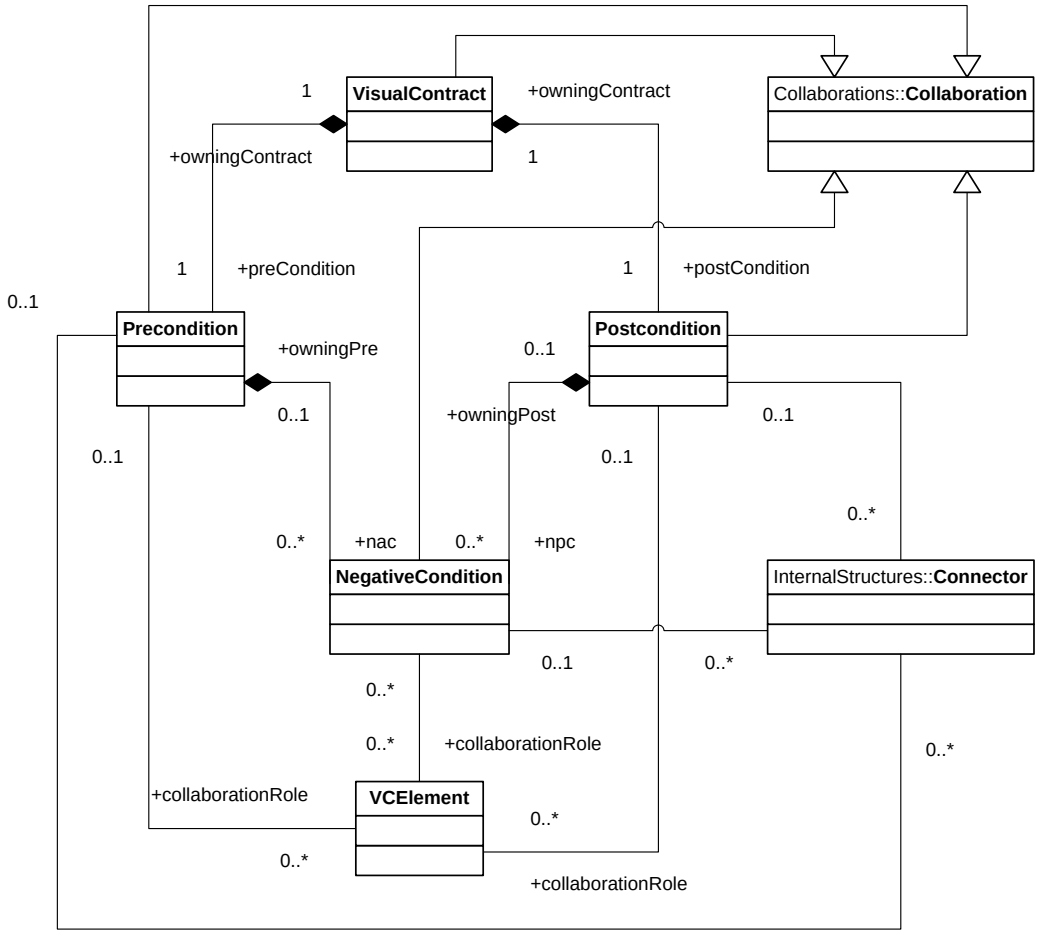


Abbildung 3.8: Metamodell für visuelle Kontrakte (Paket VisualContracts)

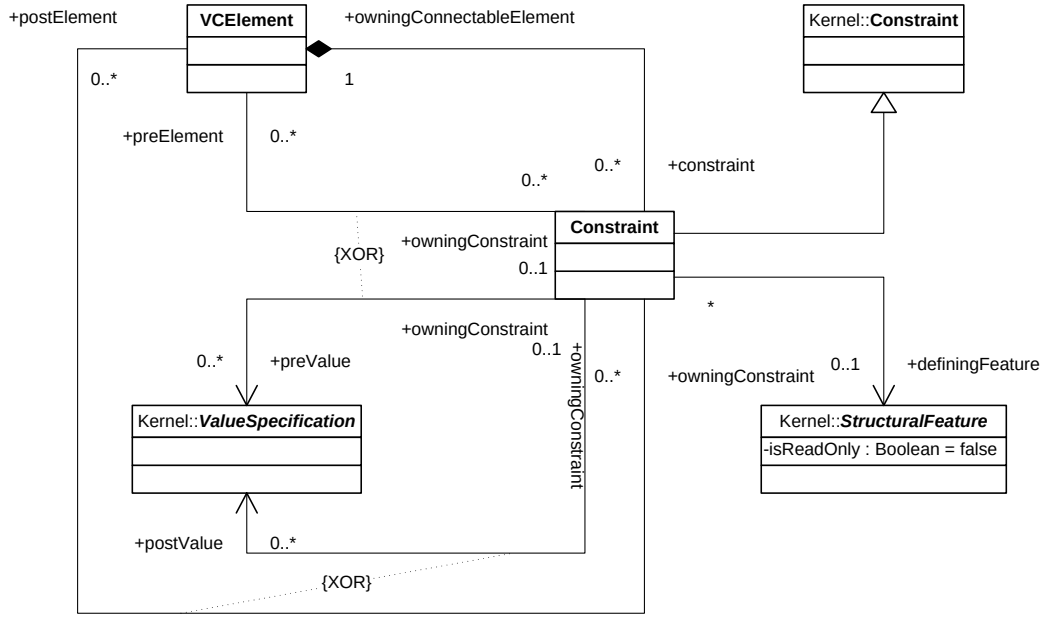


Abbildung 3.9: Metamodell für visuelle Kontrakte — Attributinhalte für positve Vor- und Nachbedingungen (Paket VisualContracts)

In den visuellen Kontrakten können auch Attributinhalte, die eine Instanz besitzen muss, um eine bestimmte Rolle einzunehmen, spezifiziert werden. Bei den Attributinhalten kann es sich um einfache Daten, Aufrufparameter der Operation oder auch Variablen handeln. Die Aufrufparameter und Variablen werden zur Laufzeit beim Aufruf der Operation durch einen Client an konkrete Werte gebunden. Mit der Spezifikation der Attributinhalte werden einer Rolle neben der Typdefinition also zusätzliche Constraints zugewiesen.

Gemäß dem UML 2.0 Metamodell können die Inhalte der Attribute, die der Typ einer Rolle definiert, nicht weiter eingeschränkt werden. Diese weiteren Einschränkungen einer Rolle können lediglich über die Metamodellklasse `Kernel::Constraint`, die gemäß dem UML-Metamodell jedem Element zugewiesen werden kann, realisiert werden. Der Constraint selbst kann mit der Object Constraint Language beschrieben werden.

Existierende Frameworks zur Erzeugung visueller Editoren auf Basis von Metamodellen, wie z.B. das Eclipse Modeling Framework [Ecl06b], unterstützen die Object Constraint Language jedoch nur sehr unzureichend. Um die Entwicklung eines Werkzeuges zu vereinfachen, haben wir uns daher entschieden, keine OCL-Constraints zur Beschreibung der Inhalte von Attributen zu verwenden. Stattdessen haben wir dem Metamodell für visu-

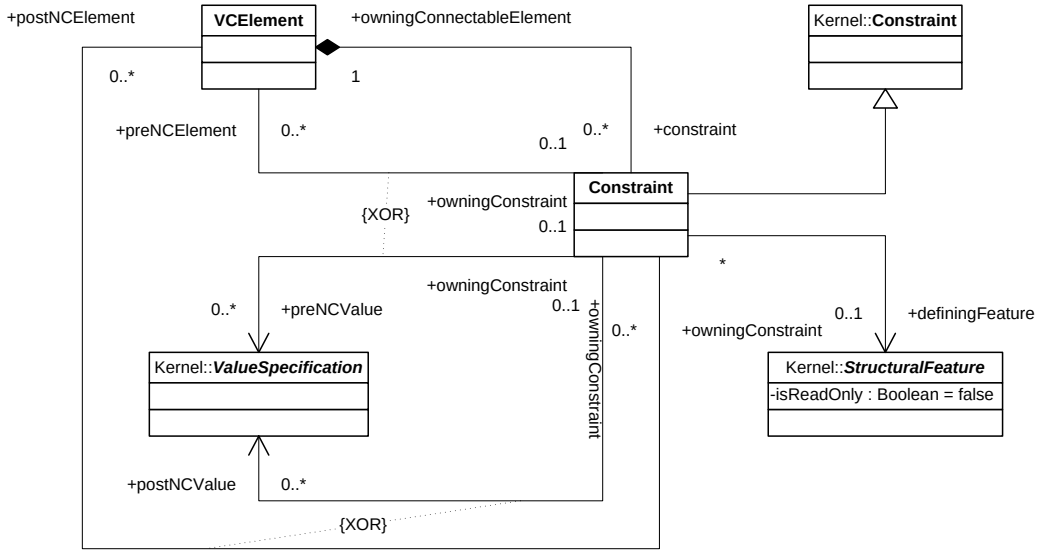


Abbildung 3.10: Metamodell für visuelle Kontrakte - — Attributinhalte für negative Vor- und Nachbedingungen (Paket VisualContracts)

elle Kontrakte die zusätzliche Klasse `Constraint` (als Spezialisierung der Klasse `Kernel::Constraint`) hinzugefügt. Die Klasse `Constraint` ist einem `VCElement` zugeordnet. Sie referenziert die `StructuralFeature` des Classifers über den die Rolle getypt ist und gleichzeitig entweder einen elementaren Datentypen oder ein weiteres `VCElement`, d.h. eine weitere Rolle, einen Parameter oder eine Variable. Damit für eine Rolle in der Vor- und Nachbedingung sowie in den negativen Vor- und Nachbedingungen unterschiedliche Inhalte definiert werden können, referenziert ein `Constraint` die Inhalte in Abhängigkeit vom Einsatz der Rolle in einer Vor-, Nachbedingung oder negativen Vor-, Nachbedingung (vgl. Abbildung 3.9 und 3.10).

Anhang A enthält eine detailliertere, strukturierte Beschreibung der hier eingeführten Metamodellklassen des Paketes `VisualContracts`. Die Beschreibung erfolgt in einem standardisierten Format, das dem Format der UML-Spezifikation zur Beschreibung von Metamodellklassen ähnelt.

3.4 Semantik von visuellen Kontrakten

Nach der Beschreibung der Syntax der visuellen Kontrakte erläutern wir im Folgenden die Semantik der visuellen Kontrakte auf der Basis von Graphtransformationen. Damit kann die Semantik der visuellen Kontrakte wohl-

definiert werden, da die Semantik von Graphtransformationen wohldefiniert ist. So stellen wir sicher, dass die visuellen Kontrakte immer auf die gleiche Weise interpretiert werden können.

Graphtransformationen bilden eine etablierte theoretische Grundlage, die nun seit mehr als 30 Jahren erforscht wird. Basierend auf den fundamentalen Konzepten von Pfaltz [PR69], Montanari [Mon70], Schneider und Ehrig [EPS73] sowie anderen haben sich verschiedene Formalisierungen entwickelt. Ein Überblick über die verschiedenen Varianten von Graphtransformationen und ihre Anwendung findet sich in den Büchern „Handbook of Graph Grammars“ [Roz97, EEK99, EMK99]. Sämtliche Formalisierungen von Graphtransformationen basieren jedoch auf demselben einfachen Kern: Eine Graphtransformation produziert aus einem gegebenen Graphen — bestehend aus Knoten und Kanten — einen veränderten Graphen. Die Veränderung des Graphen während einer Transformation wird mit einer Graphtransformationsregel beschrieben. Die Unterschiede zwischen den verschiedenen Ansätzen ergeben sich zum Beispiel aus der Definition der den Graphtransformationen unterliegenden Graphen oder der Art, wie eine Graphtransformationsregel auf einen Graphen angewendet wird.

Im Folgenden führen wir getypte, attributierte Graphen und getypte attributierte Graphtransformationen zur Formalisierung von Klassendiagrammen und den darüber getypten visuellen Kontrakten ein. Die grundlegende Idee hierbei ist, dass ein UML-Klassendiagramm als ein attributierter Typgraph und ein visueller Kontrakt als eine getypte, attributierte Graphtransformation betrachtet werden kann. Zur Erhöhung der Ausdrucksfähigkeit unserer visuellen Kontrakte erlauben wir negative Vor- bzw. Nachbedingungen und auch Mengen von Objekten. Auch diese Konzepte sind in der Theorie der Graphtransformationen wohlbekannt und werden in diesem Kapitel beschrieben.

Die klassische Interpretation von Graphtransformationen basiert auf dem Double-Pushout-Ansatz (DPO) [EPS73, CMR$^+$97]. Sie besagt, dass die Transformationen eines Graphen vollständig mit einer Graphtransformationsregel spezifiziert wird. Bei der Ausführung einer Graphtransformationsregel darf nichts ausser den explizit spezifizierten Effekten verändert werden.

Unsere visuellen Kontrakte benötigen jedoch eine losere Interpretation, da ein visueller Kontrakt eine unvollständige Beschreibung des Verhaltens einer Operation ist und damit auch nicht mehr ausführbar ist. Stattdessen ist ein visueller Kontrakt als eine Spezifikation anzusehen. Ein Programmierer muss sich bei der Umsetzung der Spezifikation, d.h. bei der Implementierung

des zu entwickelnden Systems, an diesem visuellen Kontrakt orientieren. Ein visueller Kontrakt gibt die Transformationen vor, die eine Implementierung mindestens ausführen muss. Das heißt, die Implementierung einer Operation, die mit einem visuellen Kontrakt detailliert wurde, darf neben den spezifizierten Transformationen, noch weitere Änderungen des Systemzustandes verursachen (vgl. auch Abbildung 2.6). Zu diesem Zweck kann eine losere Interpretation von Graphtransformationen in Form von *Graphtransitionen*, die auf der Theorie des Double-Pullback-Ansatzes (DPB) [Hec98, HEWC01] beruht, verwendet werden. Dieser Ansatz definiert Graphtransitionen als eine Verallgemeinerung der DPO-Transformationen und erlaubt Veränderungen im Systemzustand, die nicht explizit in den Graphtransformationsregeln beschrieben worden sind.

3.4.1 Getypte, attributierte Graphen

Graphtransformationen transformieren Graphen. Zur Beschreibung von Graphen wiederholen wir kurz die Definition von Graphen und Graphmorphismen. Zu beachten ist, dass die einfache Definition eines Graphen — G besteht aus einer Menge von Knoten V und einer Menge von Kanten E mit $E \in V \times V$ — nicht die Definition von verschiedenen Kanten zwischen zwei identischen Knoten erlaubt, wie es in Klassendiagrammen oder auch Composite-Structure-Diagrammen möglich ist. Daher verwenden wird die folgende Definition von Graphen:

Definition 3.1 (Graph) *Ein* gerichteter und unmarkierter Graph *ist ein Tupel* $G = \langle G_V, G_E, src_G, tar_G \rangle$ *mit einer Knotenmenge* G_V*, einer Kantenmenge* G_E *und Abbildungen* $src_G : G_E \to G_V$ *und* $tar_G : G_E \to G_V$*, die jeder Kante einen Start- und Zielknoten zuordnen.*

Alle Kanten in diesen Graphen verbinden zwei Knoten und sind gerichtet. Ungerichtete Graphen oder Hypergraphen — Graphen, in denen Kanten mehr als zwei Knoten miteinander verbinden — werden hier nicht betrachtet.

Getypte Graphen Im Software Engineering wird zwischen Typ- und Instanzgraphen unterschieden. Beispiele hierfür aus dem Bereich der UML sind Klassen- und Objektdiagramme bzw. Composite-Structure-Diagramme (Kompositionsstrukturdiagramme) oder auch die Definition der UML selbst (hier wird der Instanziierungsmechanismus auf den 4 Ebenen des Metamodells verwendet). Auch in unseren visuellen Kontrakten unterscheiden wir zwischen Typ- und Instanzgraphen. Ein Klassendiagramm ist ein Typgraph,

der zur Laufzeit erlaubte Strukturen vorgibt. Die Vor- und Nachbedingung eines visuellen Kontraktes sind getypt über dem Klassendiagramm.

In der Theorie der Graphtransformationen ist dieses objektorientierte Konzept mit den *getypten Graphen* [CMR96, HCEL96] eingeführt worden. Die Typisierung eines Graphen wird durch eine strukturverträgliche Abbildung in einen Typgraphen beschrieben, dessen Knoten und Kanten Typen für den Ausgangsgraphen bilden. Um den Begriff des Typgraphen präzise einführen zu können, wird die Typisierungsbeziehung mit einem *Graphmorphismus* beschrieben.

Definition 3.2 (Graphmorphismus) *G, H seien zwei Graphen. Ein* Graphmorphismus *$f : G \to H$ ist ein Paar von Abbildungen $\langle f_V : G_V \to H_V; f_E : G_E \to H_E \rangle$, die die Ausgangs- und Zielknoten von Kanten bewahren, d.h. $src_H \circ f_E = f_V \circ src_G$ und $tar_H \circ f_E = f_V \circ tar_G$.*

Definition 3.3 (Getypter Graph) *Ein* getypter Graph *ist ein Graph G, dessen Knoten und Kanten durch einen Graphmorphismus $type : G \to TG$ Typen aus einem* Typgraphen *TG zugeordnet werden [CMR96]. Für ein Element $x \in G$ heißt $type(x) = t$ sein* Typ*, und x wird oft in der Form $x : t \in G$ notiert.*

*Das Paar $\langle G, type \rangle$ heißt TG-*getypter Graph.

*Ein Morphismus $g : G \to G'$ heißt TG-*getypter Graphmorphismus *zwischen den TG-getypten Graphen $\langle G, type \rangle$ und $\langle G', type' \rangle$, falls $type' \circ g = type$ gilt.*

Attributierte Graphen Ein weiteres wichtiges objektorientiertes Konzept, das wir in unseren visuellen Kontrakten verwenden, ist das der Attribute. Klassen können Attribute definieren, d.h. einen Attributnamen und einen Typen für das Attribut. In der Instanz einer Klasse wird ein Attribut an einen Wert gebunden. In unseren visuellen Kontrakten werden den Attributen einer Rolle Werte zugewiesen, um die möglichen Instanzen, die zur Laufzeit die beschriebene Rolle einnehmen können, weiter einzuschränken.

In der Theorie der Graphtransformationen sind attributierte Graphen in [CMR+97, LKW93] eingeführt worden. Graphen werden als attributiert bezeichnet, falls deren Knoten oder Kanten mit Elementen eines abstrakten Datentyps (z.B. einem String oder Integer) verknüpft (koloriert) sind. Abstrakte Datentypen integrieren Daten und Operationen. Genauer gesagt werden einem Graphen algebraische Signaturen zugewiesen. Über diesen Signaturen wird eine Algebra definiert.

Im Folgenden betrachten wir eine vereinfachte Form von attributierten Graphen, die es erlaubt den Knoten eines Graphen Attributinhalte zuzuweisen. Die Zuweisung von Werten zu Kanten wird nicht unterstützt, da dies in unserem Ansatz nicht verwendet wird. Als erstes führen wir Signaturen und eine darauf definierte Σ-Algebra ein. Danach erweitern wir die Graphdefinition, um Graphen mit Inhalten aus einer Algebra zu verknüpfen.

Definition 3.4 (Signatur) *Eine* Signatur Σ *ist ein Tupel* $\langle S, OP\rangle$ *bestehend aus einer Menge* S *von* Typsymbolen *(*Sorten*), und einer Familie von Mengen von Operationssymbolen* $OP = (OP_{w,s})_{w\in S^*, s\in S}$*, die durch ihre Argumente und ihren Ergebnistyp indiziert sind. Statt* $op \in OP_{s_1,\ldots,s_n,s}$ *kann auch* $op : s_1 \ldots s_n \to s$ *geschrieben werden.*

Definition 3.5 (Σ-Algebra) *Eine* Algebra A *einer Signatur* Σ *(auch* Σ-Algebra *genannt) ist ein Tupel* $A = \langle (A_s)_{s\in S}, (op^A)_{op\in OP}\rangle$ *bestehend aus einer Familie* $(A_s)_{s\in S}$ *von Trägermengen für die Typsymbole in* S *und einer Familie von Operationen* $op^A : A_{s_1} \times \ldots \times A_{s_n} \to A_s$ *für jedes Operationssymbol* $op : s_1 \ldots s_n \to s$ *in* Σ*. Wir bezeichnen mit* $|A|$ *die disjunkte Vereinigung der Trägermengen* D_s *von* A *für alle* $s \in S$*, d.h.* $|A| = \biguplus_{s\in S} D_s$*.*

Ein attributierter Graph ist ein gerichteter und unmarkierter Graph, der aus einem einfachen Graphen und einem Graphen zur Repräsentation von Attributinhalten besteht. Gemäß [CMR+97] werden Attribute als Verknüpfungen zwischen Knoten und Attributwerten angesehen. Wenn ein Knoten ein Attribut mit dem Inhalt v besitzt, so ist dieser Knoten über eine Kante mit der Identität des Attributnamens mit einem Knoten verbunden, der diesen Attributwert darstellt. Formal ergibt sich folgende Definition für attributierte Graphen.

Definition 3.6 (Attributierter Graph) *Ein* attributierter Graph AG *über einer Signatur* $\Sigma = \langle S, OP\rangle$ *ist ein Tupel* $\langle G, A\rangle$ *bestehend aus einem Graphen* $G = \langle G_V, G_E, src_G, tar_G\rangle$ *und einer* Σ*-Algebra* A*, für die gilt:* $|A| \subseteq G_V$ *und* $\forall e \in G_E : src_G(e) \notin |A|$*.*

Laut dieser Definition ist ein attributierter Graph ein gerichteter, unmarkierter Graph, der aus einem einfachen Graphen G und einem Graphen zur Repräsentation der Attributinhalte besteht. Datenwerte werden nach dieser Definition durch *Datenknoten* $d \in |A|$ des Graphen repräsentiert. Die Datenknoten sind zu unterscheiden von den *Objektknoten* $v \in G_V \setminus |A|$. Die Kanten zwischen Objektknoten werden *Links* genannt. Die Kanten, die einen Objektknoten mit einem Datenknoten — Kanten $e \in G_E$ mit $scr_G(e) \in G_V \setminus |A|$ und $tar_G(e) \in |A|$ — verbinden werden *Attributkanten* genannt. Die Knoten aus $|A|$ dürfen keine ausgehenden Kanten haben[ETH+03].

In anderen Ansätzen, wie z.B. in [LKW93] werden spezielle Träger für Datenwerte (*Attribute Carrier*) eingeführt, um Graphelemente mit Datenwerten aus einer Algebra zu verknüpfen. Bei diesen Ansätzen müssen die algebraischen Strukturen auch in weitergehende Definitionen integriert werden. Morphismen und Single-Pushouts sind z.B. in [LKW93] redefiniert worden. In dem hier beschriebenen Ansatz ist die Verbindung eines Knoten mit einem Datenwert über eine Kante innerhalb des Graphen geregelt. Dies vereinfacht die Definition von weitergehenden Merkmalen und Charakteristiken, da attributierte Graphen in diesem Fall als ein spezieller Fall von einfachen Graphen angesehen werden können, wenn die obigen Constraints berücksichtigt werden. Bei dieser Definition von attributierten Graphen muss der Formalismus der Graphtransformationen also nicht angepasst werden. Jedoch können in dem oben definierten Ansatz den Kanten keine Datenwerte zugewiesen werden, während in [LKW93] auch Kanten Attribute besitzen können.

Getypte, attributierte Graphen Zur Typisierung von attributierten Graphen ist der Begriff des Morphismus zu erweitern.

Definition 3.7 (Σ-Homomorphismus) *Ein* Σ-Homomorphismus $f_A : A_1 \rightarrow A_2$ *zwischen zwei* Σ*-Algebren* A_1 *und* A_2 *ist eine Familie von Abbildungen* $f_A = (f_s)_{s \in S}$*, die verträglich mit den Operationen der beiden Algebren ist.*

Definition 3.8 (Attributierter Graphmorphismus) *Ein* attributierter Graphmorphismus $f : (G_1, A_1) \rightarrow (G_2, A_2)$ *ist ein Paar* $f = (f_G, f_A)$ *aus einem Graphmorphismus* $f_G = (f_V, f_E) : G_1 \rightarrow G_2$ *und einem* Σ*-Homomorphismus* $f_A : A_1 \rightarrow A_2$*, so dass* $|f_a| \subseteq f_V$ *mit* $|f_A| = \cup_{s \in S} f_s$ *und* $A_{1_s} = f_v^{-1}(A_{2_s})$ *für alle* $s \in S$*.*

Die erste Bedingung besagt, dass die Abbildung der Datenknoten in der Abbildung der Objektknoten enthalten ist. Die zweite Bedingung fordert, dass Attribute aus A_1 nach A_2 abgebildet werden und dass andere Knoten aus V_1 nicht nach A_2 abgebildet werden. Somit werden Objektknoten auf Objektknoten und Datenknoten auf Datenknoten abgebildet.

Definition 3.9 (Getypte, attributierte Graphen) *Ein* attributierter Typgraph *ist ein attributierter Graph* $ATG = \langle TG, Z \rangle$ *über* Σ*, wobei* Z *die finale* Σ*-Algebra* $Z = (D, (op^D)_{op \in OP}$ *mit* $D_s = \{s\}$ *für alle* $s \in S$ *und* $op^D : \{s_1\} \times \ldots \times \{s_n\} \rightarrow \{s\}$ *für jedes Operationssymbol* $op : s_1 \ldots s_n \rightarrow s$ *in* Σ *ist.*

Ein attributierter Instanzgraph $\langle AG, ag \rangle$ *über ATG ist ein attributierter Graph AG über derselben Signatur mit einem attributierten Graphmorphismus* ag :

$AG \rightarrow ATG$.

Ein Morphismus $h : \langle AG_1, ag_1 \rangle \rightarrow \langle AG_2, ag_2 \rangle$ *über getypte, attributierte Graphen ist ein attributierter Graphmorphismus, der die Typen beibehält, d.h.* $ag_2 \circ h = ag_1$.

Gemäß dieser Definition repräsentieren die Elemente aus Z die Sorten der Signatur, die in TG zur Typisierung der Datenknoten eingefügt werden. Im allgemeinen repräsentieren die Knoten und Kanten, die in TG definiert werden, Knoten- und Kantentypen. Die Attribute in ATG sind die Attributdeklarationen.

Die Instanzgraphen enthalten normalerweise eine unendliche Menge von Datenknoten. Wenn z.B. der Datentyp $\mathbb{N}$ der natürlichen Zahlen verwendet wird, so muss für jedes $n \in \mathbb{N}$ ein separater Knoten im Instanzgraphen eingefügt werden. Da jedoch während der Graphtransformationen die Datentypknoten nicht verändert werden, ist es nicht notwendig die unendliche Menge der Datenknoten zu repräsentieren (vgl. auch Abbildung 3.11).

Visuelle Kontrakte und getypte, attributierte Graphen Nach der Einführung der Theorie der getypten, attributierten Graphen beschreiben wir nun, wie diese Theorie zur Formalisierung der visuellen Kontrakte verwendet werden kann. Damit wird die Grundlage zur Formalisierung der visuellen Kontrakte mit Graphtransitionen beschrieben.

In unserem Ansatz ist eine Vor- bzw. Nachbedingung eines visuellen Kontraktes über einem UML-Klassendiagramm getypt. Ein UML-Klassendiagramm kann im Prinzip als attributierter Typgraph interpretiert werden. Ein Klassendiagramm besteht aus Klassen, Attributen und Assoziationen. Zur Vereinfachung gehen wir davon aus, dass es sich bei unseren Klassendiagrammen um flache Klassendiagramme handelt. Vererbungsbeziehungen sind also aus dem Klassendiagramm entfernt worden. Auf der Implementierungsebene können Vererbungsbeziehungen auf verschiedene Weisen simuliert werden: Zum Beispiel ist es möglich Status-Flags einzuführen oder die Kontrolle kann an eine entsprechende versteckte Klasse delegiert werden (vgl. auch [Fow96])[6].

[6]In anderen radikaleren Ansätzen wird sogar vorgeschlagen vollständig auf Vererbung zu verzichten. Stattdessen sollen nur noch Schnittstellen verwendet werden, die nicht erweitert werden dürfen. Lediglich eine Klasse kann mehrere Schnittstellen implementieren [WS96]. Ein Hinweis, dass derartige Design Prinzipien in der modernen Softwareentwicklung einsetzbar sind, ist Microsoft's Component Object Model (COM) [Box97]. In COM werden auf der Spezifikationsebene keine (Vererbungs-)Hierarchien erlaubt. Ein Objekt kann lediglich eine beliebige Anzahl von Schnittstellen implementieren.

Ansätze zur Integration von Vererbungskonzepten in die Theorie der Graphtransformationen finden sich in [EPT04, BEL+03, BELT04].

Geht man von einem flachen Klassendiagramm aus, so ist der Zusammenhang zwischen einem UML-Klassendiagramm, visuellen Kontrakten und getypten, attributierten Graphen offensichtlich:

- Eine Klasse ist ein Knoten in einem Typgraphen.
- Ein Attribut im Klassendiagramm bekommt eine Attributkante zu einem Typsymbol (einer Sorte).
- Assoziationen im Klassendiagramm sind Links im Typgraphen.
- Eine Rolle in einer Vor- oder Nachbedingung ist ein Knoten im Instanzgraphen.
- Ein Attribut einer Rolle, dem ein Wert zugewiesen ist, bekommt eine Attributkante zu einem Datenknoten im Instanzgraphen.
- Konnektoren in einer Vor- oder Nachbedingung sind Links im Instanzgraphen.

In einer formalen graphischen Darstellung werden Objektknoten mit Rechtecken und Datenknoten mit Elipsen dargestellt. Links verlaufen zwischen zwei Objektknoten, Attributlinks verlaufen zwischen einem Objekt- und einem Datenknoten.

Abbildung 3.11 zeigt den Zusammenhang zwischen attributierten, getypten Graphen und einer UML Collaboration zur Repräsentation einer Vor- oder Nachbedingung in einem visuellen Kontrakt. Auf der linken Seite findet sich oben ein Klassendiagramm (Ausschnitt des Klassendiagramms aus Abbildung 3.1) und unten ein entsprechender Typgraph. Die Inhalte des Attributes `name` können die Elemente der Trägermenge `String` sein, d.h. dem Attribut `name` wird der Typ `String` zugewiesen. Ähnlich verhält es sich bei dem Attribut `quantity`. Diesem Attribut wird der Typ `Integer` zugewiesen. Zu beachten ist, dass es sich bei den Beschriftungen der Knoten und Kanten um die Identitäten der Knoten und Kanten handelt, da wir nicht mit gelabelten Graphen arbeiten. Auf der rechten Seite der Abbildung sind Instanzen beschrieben. Oben ist eine UML Collaboration und unten ein attributierter, getypter Graph beschrieben. Der getypte Graph ist auf die Datenknoten, die mit einem Objektknoten verbunden sind, reduziert. Zu beachten ist, dass der attributierte Graphmorphismus durch die Beschriftung der Knoten im formalen Instanzgraphen gegeben ist. Zum Beispiel wird der Knoten $c : Cart$ im Instanzgraphen auf den Knoten $Cart$ im Typgraphen abgebildet.

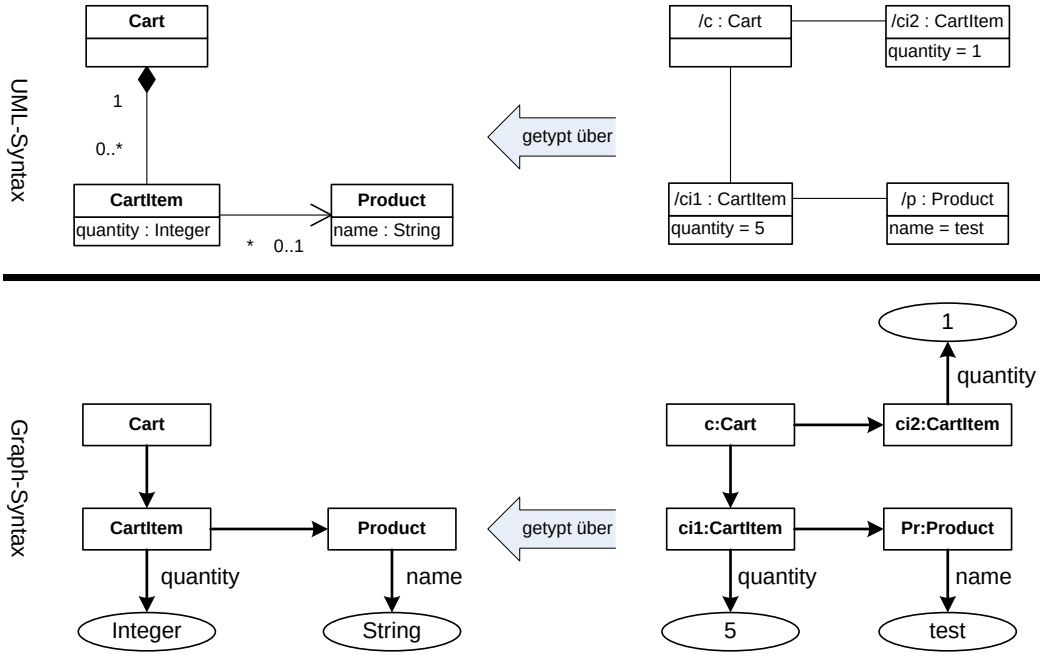


Abbildung 3.11: UML-Klassendiagramm und UML Collaboration vs. Attributierter Typgraph und Instanzgraph

3.4.2 Graphtransitionen

Die Struktur eines Graphen kann mit einer *Graphtransformation* verändert werden. Die generelle Idee der Theorie der Graphtransformationen ist, dass die Veränderung eines Graphen mit *Graphtransformationsregeln* (auch *Produktionen* genannt), die von der konkreten Beschreibung der zu transformierenden Graphen durch die Verwendung von Pattern abstrahieren, beschrieben werden können. Eine Graphtransformationsregel besteht aus einem Paar von Graphen, der linken und rechten Seite. Informell gesprochen wird eine Graphtransformationsregel auf einem Hostgraphen ausgeführt, indem zuerst ein *Match* der linken Seite der Regel gefunden wird. Match bedeutet, dass die auf der linken Seite der Graphtransformationsregel beschriebenen Strukturen im Hostgraphen vorhanden sind. Danach wird der Match durch die rechte Seite der Graphtransformationsregel ersetzt. Für detailliertere Informationen zu den Grundlagen von Graphtransformationen möchten wir den Leser auf [CMR+97] verweisen.

Graphtransformationen Das Konzept der Graphtransformationen kann auch für die Transformation von attributierten Graphen verwendet werden,

um z.B. die Transformation von Systemzuständen, die mit Collaborations der UML (vgl. auch Abschnitt 3.3 sowie Abbildung 3.6) dargestellt werden, zu beschreiben. Dies führt zu dem Konzept von attributierten Graph Grammatiken und attributierten Graphtransformationen. Attributierte Graph Grammatiken haben ihren Ursprung in attributierten Grammatiken, welche von Knuth [Knu68] zur Spezifikation von so genannten statischen semantischen Aspekten von Programmiersprachen eingeführt wurden (für einen Überblick siehe auch [Paa95]). Das bekannte Konzept der Double-Pushout-Graphtransformationen [CMR+97] auf Graphen ist in [HKT02a] auf getypte, attributierte Graphen erweitert worden. Der Double-Pushout-Ansatz [CMR+97] für Graphtransformationen basiert auf der Existenz des Double-Pushout-Diagramms, um die Klebebedingungen (gluing condition) [CMR+97] sicherzustellen. Im Folgenden verwenden wir jedoch eine mengentheoretische Definition der Graphtransformationen.

Definition 3.10 (Graphtransformationsregel, Graphtransformation) *Eine* Graphtransformationsregel $p : L \rightarrow R$ *besteht aus einem Paar von Graphen* L, R, *die über einem Typgraphen* TG *getypt sind und eine wohldefinierte Vereinigungsmenge* $L \cup R$ *definieren.*

Wohldefiniert bedeutet in diesem Fall, dass die Kanten, die sowohl in L *als auch* R *verwendet werden, in beiden Graphen mit denselben Knoten verbunden sind und dass Knoten mit demselben Namen auch denselben Typ besitzen müssen, etc.*

Die linke Seite einer Graphtransformationsregel L *repräsentiert die Vorbedingung, während die rechte Seite* R *die Nachbedingung einer Regel repräsentiert.*

Gegeben seien zwei Graphen G, H. *Eine* Graphtransformation $G \overset{p(o)}{\Rightarrow} H$ *ist beschrieben durch einen attributierten, getypten Teilgraphisomorphismus* $o : L \cup R \rightarrow G \cup H$ *(*occurence *genannt) mit:*

- $o(L) \subseteq G$, *d.h. die linke Seite einer Regel kann in den Vorzustand eingebettet werden.*
- $o(R) \subseteq H$, *d.h. die rechte Seite der Regel kann in den Nachzustand eingebettet werden.*
- $o(L \setminus R) = G \setminus H$, *d.h. es wird genau der Teil von* G *gelöscht auf den die Elemente von* L *abgebildet werden.*
- $o(R \setminus L) = H \setminus G$, *d.h. es wird genau der Teil von* H *hinzugefügt auf den die Elemente von* R *abgebildet werden.*

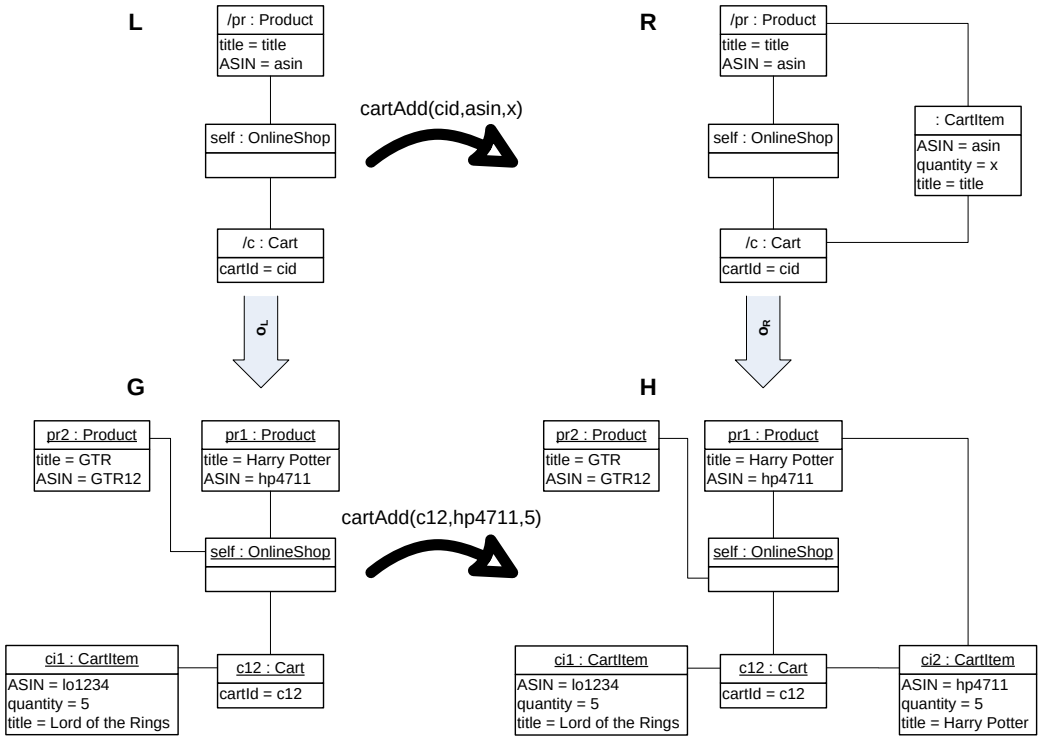


Abbildung 3.12: Anwendung der Regel `cartAdd` (DPO)

Der DPO-Ansatz stellt eine Einschränkung der Anwendbarkeit einer Transformationsregel in einem gegebenen Zustand dar: eine Regel, die eine Instanz löscht, darf nicht angewendet werden, falls dadurch eine Kante ohne Quell- oder Zielknoten erhalten bleibt. Diese *dangling condition* ist das konservative Verfahren, um sicher zu stellen, dass die resultierende Struktur wieder ein wohldefinierter Graph ist, d.h. dass die Abhängigkeit zwischen Kanten und ihren Quell- und Zielknoten bewahrt wird. Es werden nur die explizit in der Regel spezifizierten Teile des Kontextes gelöscht (siehe auch [EPS73]). Damit eignet sich der DPO-Ansatz für die Beschreibung von Verhalten in Form von Zustandsänderungen eines Systems, ohne unbeabsichtigte Seiteneffekte nach sich zu ziehen.

Abbildung 3.12 zeigt die Anwendung einer Graphtransformationsregel auf den Zustand eines Systems. Der Zustand des Systems wird vor und nach Ausführung der Graphtransformationsregel mit einem Objektdiagramm beschrieben. Die Ausführung einer attributierten Graphtransformation erfolgt in drei Schritten

1. Der Teilgraph o_L der linken Seite der Graphtransformationsregel muss

in dem gegebenen Graphen G gefunden werden. Dabei werden den Variablen aus der Graphtransformationsregel, die in L verwendet werden, die Werte aus den entsprechenden Attributen in G zugewiesen. In dem Beispiel wird z.B. der Variablen `title` der Wert `Harry Potter` zugewiesen.

2. Alle Knoten, Kanten und Attribute (Attributlinks), die zu $L \setminus R$ gehören, werden entfernt. Dabei ist sicherzustellen, dass der verbliebene Graph $D := G \setminus o(L \setminus R)$ weiterhin ein legaler Graph ist, d.h. es dürfen aufgrund des Löschens von Knoten keine Dangling Edges existieren.

3. Der Graph D muss mit dem Graphen $R \setminus L$ verbunden werden, um den Ergebnisgraphen H zu erhalten. Dazu gehört die Generierung neuer Attributlinks zu Datenknoten durch die Auswertung der Variablen, denen in der Vorbedingung beim Matching Inhalte zugewiesen wurden.

In Abschnitt 2.2 haben wir bereits beschrieben, dass Kontrakte bzw. visuelle Kontrakte zur Spezifikation von Operationen während der objektorientierten Analyse oder des objektorientierten Designs im Softwareentwicklungsprozess eingesetzt werden können. In den späteren Phasen der Entwicklung werden dem System weitere Klassen hinzugefügt, weil die initiale Spezifikation entweder unvollständig oder die gewählte Implementierung weitere Klassen benötigt. Eine wichtige Beobachtung war, dass die Spezifikation des objektorientierten Systems und seine Implementierung dieselbe Basismenge an Klassen und Assoziationen besitzen. Die grundlegende Annahme dabei ist, dass eine gute Spezifikation alle Klassen, Attribute und Assoziationen enthält, um die aus der Anforderungsdefinition hervorgehenden notwendigen Zustände des Systems zu beschreiben. Dadurch ergibt sich jedoch, dass die visuellen Kontrakte das Verhalten einer Operation nicht exakt beschreiben. Sie beschreiben lediglich, wie der Systemzustand mindestens zu verändern ist. Wenn eine Operation zur Laufzeit ausgeführt wird, kann es jedoch passieren, dass zusätzliche Effekte auftreten, z.B. wenn eine weitere Klasse zur Unterstützung der Implementierung initialisiert oder ein weiterer Attributinhalt berechnet wird.

Der hier beschriebene Double-Pushout-Ansatz geht davon aus, dass die Produktionen das System und sein Verhalten in Form von Zustandsänderungen vollständig beschreiben. Eine Graphtransformationsregel beschreibt die Veränderungen des Systemzustandes in Form von Objektveränderungen. Der in der Graphtransformationsregel nicht beschriebene Teil des Systemzustandes wird als Kontext interpretiert und darf sich nicht verändern. Aus diesem Grund ist der Double-Pushout Ansatz ungeeignet zur Formalisierung der Se-

mantik der visuellen Kontrakte.

Graphtransitionen Der Double-Pullback-Ansatz (DPB) definiert gegenüber dem Double-Pushout Ansatz eine losere Interpretation einer Graphtransformation und führt den Begriff der *Graphtransition* ein. Eine Graphtransformationsregel beschreibt immer noch die Veränderungen einer Vorbedingung für die ein erfolgreiches Matching gefunden wurde, jedoch können während der Ausführung der Graphtransition weitere Veränderungen des Systemzustandes erfolgen. Eine Graphtransition stellt lediglich sicher, dass Elemente gemäß einer Graphtransformationsregel gelöscht, erzeugt oder beibehalten werden. Jedoch kann der Systemzustand durch das Hinzufügen oder Löschen von weiteren Elementen zusätzlich verändert werden. Beim DPB-Ansatz werden somit die impliziten Frame Conditions des DPO-Ansatzes (alle Elemente eines Graphen, welche nicht zum Match gehören, bleiben unverändert) nicht verwendet. Formal werden Graphtransitionen mit der Ersetzung der direkten Ableitung aus einem Double-Pushout-Diagramm mit einem Double-Pullback erklärt [HEWC01]. Im Folgenden verwenden wir jedoch eine mengentheoretische Definition der Graphtransitionen.

Definition 3.11 (Graphtransformationsregel, Graphtransition) *Eine* Graphtransformationsregel $p : L \to R$ *besteht aus einem Paar von Graphen* L, R, *die über einem Typgraphen* TG *getypt sind und eine wohldefinierte Vereinigungsmenge* $L \cup R$ *definieren.*

Wohldefiniert bedeutet in diesem Fall, dass die Kanten, die sowohl in L *als auch* R *verwendet werden, in beiden Graphen mit denselben Knoten verbunden sind und dass Knoten mit demselben Namen auch denselben Typ besitzen müssen, etc.*

Gegeben seien zwei Graphen G, H. *Eine* Graphtransition $G \overset{p(o)}{\rightsquigarrow} H$ *ist beschrieben durch einen attributierten, getypten Teilgraphisomorphismus* $o : L \cup R \to G \cup H$ *(*occurence *genannt) mit:*

- $o(L) \subseteq G$, *d.h. die linke Seite einer Regel kann in den Vorzustand eingebettet werden.*
- $o(R) \subseteq H$, *d.h. die rechte Seite der Regel kann in den Nachzustand eingebettet werden.*
- $o(L \setminus R) \subseteq G \setminus H$, *d.h. es wird mindestens der Teil von* G *gelöscht auf den die Elemente von* L *abgebildet werden.*

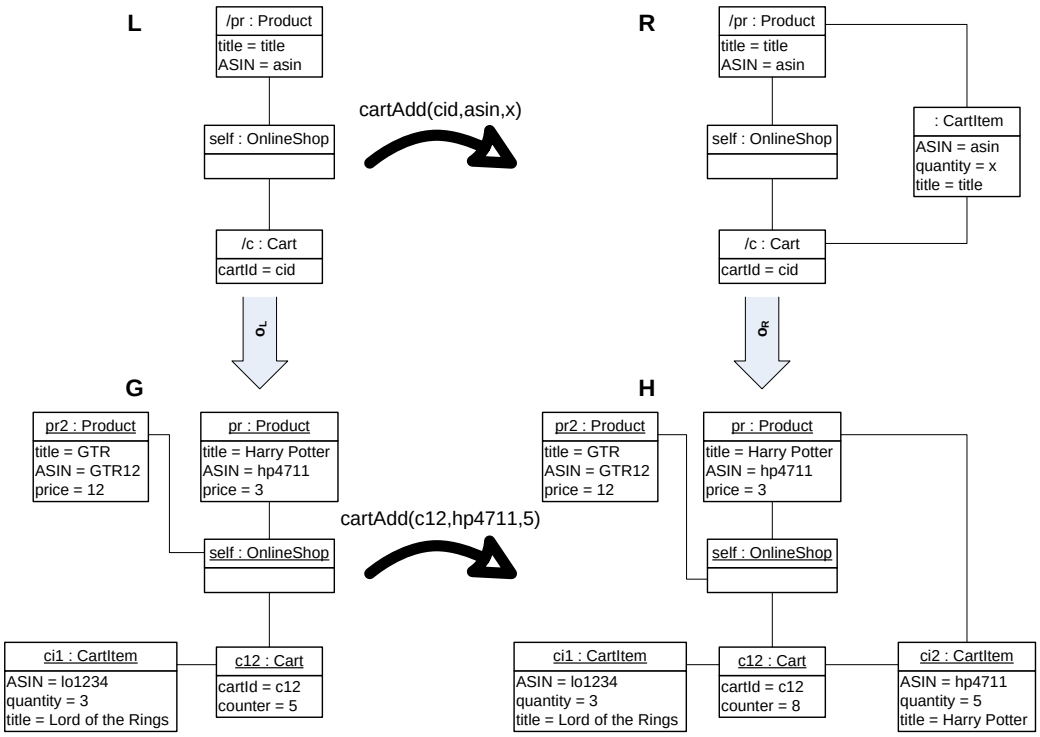


Abbildung 3.13: Anwendung der Regel `cartAdd` (DPB)

- *$o(R \setminus L) \subseteq H \setminus G$, d.h. es wird mindestens der Teil von H hinzugefügt auf den die Elemente von R abgebildet werden.*

Graphtransitionen stellen ein geeignetes Mittel zur formalen Beschreibung der Semantik der visuellen Kontrakte dar, wenn davon ausgegangen wird, dass die Spezifikation eines objektorientierten Systems und seine Implementierung dieselbe Basismenge an Klassen und Assoziationen besitzen. Ein visueller Kontrakt beschreibt, welche minimalen Veränderungen des Systemzustandes bei der Ausführung einer Operation erreicht werden müssen. Zusätzliche Effekte können z.B. auftreten, wenn eine manuelle Implementierung einer Operation einem Attribut, das nicht in dem visuellen Kontrakt berückschichtigt wurde, einen Wert zuweist oder wenn Instanzen von Klassen erzeugt werden, die zur Unterstützung der Implementierung benötigt werden, aber nicht Teil der Spezifikation sind usw. Diese zusätzlichen Effekte sind gemäß der Semantik von Graphtransitionen erlaubt.

Abbildung 3.13 zeigt ein Beispiel für die Anwendung einer Graphtransition auf einen Systemzustand. Die Graphtransformationsregel zur Beschreibung der Operation `cartAdd` hat sich nicht gegenüber der Graphtransformations-

regel aus Abbildung 3.12 geändert. Jedoch wird bei der Ausführung einer Implementierug dieser Graphtransformationsregel bzw. des visuellen Kontraktes als zusätzlicher Effekt die Anzahl aller bestellten Waren in dem Warenkorb gezählt (Attribut `counter` in Objekt `Cart`). Diese Interpretation der Graphtransformationsregeln wird durch die lose Semantik von Graphtransitionen unterstützt.

3.4.3 Negative Bedingungen

Die Elemente in den Graphtransformationsregeln beschreiben, welche Strukturen in einem Hostgraphen gegeben sein müssen. Zur Erhöhung der Mächtigkeit unserer visuellen Kontrakte erlauben wir auch die Definition von Strukturen, die nicht im Systemzustand vorkommen dürfen (negative Vor- und Nachbedingungen). Ein Beispiel ist der visuelle Kontrakt aus Abbildung 3.3. In der Theorie der Graphtransformationen sind *negative Anwendungsbedingungen* (*Negative Application Condition — NAC*) ein bekanntes Konzept zur Beschreibung von nicht erlaubten Strukturen [HHT96]. Eine NAC ist eine Erweiterung der linken oder rechten Seite einer Regel. Die Semantik einer NAC ist, dass die in einer NAC definierte Struktur nicht im Kontext der *Occurence* einer Regel (bestimmt durch Teilgraphisomorphismus) vorhanden sein darf. Negative Anwendungsbedingungen werden von den meisten Werkzeugen für Graphtransformationen unterstützt.

Die Semantik einer negativen Anwendungsbedingung beschreiben wir im Folgenden an einer beispielhaften Anwendung des visuellen Kontraktes aus Abbildung 3.3. Die Funktion `cartAdd` wird mit den Variablenbelegungen `cid=cid1`, `asin=as1` und `x=3` aufgerufen. Wird die Operation `cartAdd` auf den oberen Systemzustand aus Abbildung 3.14 aufgerufen, so ist die Vorbedingung erfüllt. Der Systemzustand enthält nicht die in der Vorbedingung definierte Struktur. Wird dieselbe Operation auf dem unteren Systemzustand aufgerufen, so ist die Vorbedingung nicht erfüllt, da `c1:Cart` mit einem `CartItem` mit dem Inhalt `as1` für das Attribut `asin` verbunden ist. Wenn die negative Anwendungsbedingung aus dem visuellen Kontrakt entfernt wird, ist die Vorbedingung wieder erfüllt.

3.4.4 Allquantifizierung

In Graphtransformationsregeln wird durch ein Element auf der linken Seite ausgedrückt, dass ein Match im Hostgraphen gefunden werden muss, d.h.

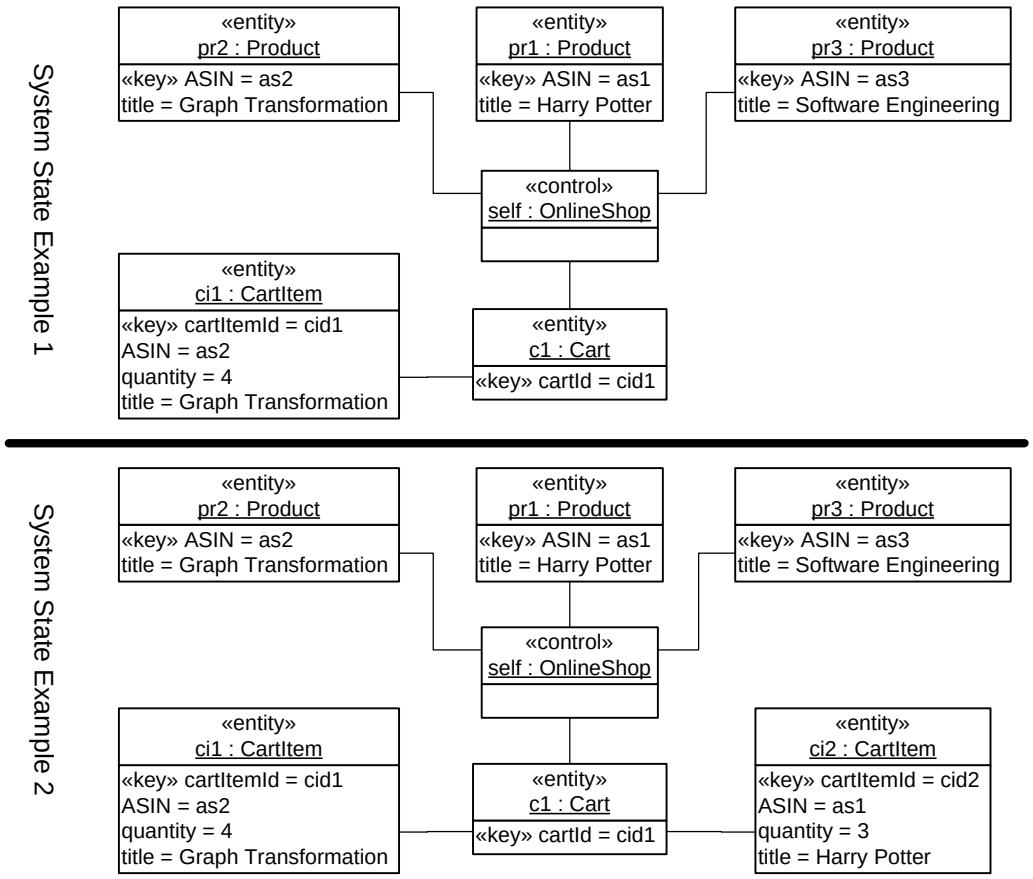


Abbildung 3.14: Systemzustände

in der Regel handelt es sich um Existenzquantifikationen. In verschiedenen Situationen wird jedoch auch eine Allquantifizierung benötigt. Ein Beispiel stellt die Operation `cartClear` aus Abbildung 3.5 dar. Beim Aufruf dieser Operation soll der Inhalt eines Warenkorbs gelöscht werden. Da der Warenkorb mehrere Elemente enthalten kann, muss die Graphtransformationsregel auf alle Elemente des Warenkorbs matchen und sicherstellen, dass diese gelöscht werden. Elemente mit einer Allquantifizierung auf der linken Seite einer Graphtransformationsregel matchen auf alle Elemente im Hostgraphen, die auf die Beschreibung in der Regel zutreffen.

Verschiedene Ansätze aus der Theorie der Graphtransformationen beschäftigen sich mit der Allquantifizierung von Elementen. Parallele Graphtransformationen [Tae96, LETE04] sind ein formaler Ansatz zur Integration von Allquantifizierungen in Graphtransformationen. Ein anderer Ansatz wird in PROGRESS verfolgt. Hier wird der Matchingbegriff für die Allquantifizierung angepasst [Sch91].

In den verschiedenen Ansätzen ist die Anwendung von Allquantoren in Graphtransformationen normalerweise eingeschränkt. Oft dürfen Allquantoren nur mit einem Existenzquantorelement verbunden werden. Als Semantik wird dann ein vollständiger Match über den Kontext des Existenzquantorelements angenommen. Weiterhin ist in den Ansätzen die Manipulation von allquantifizierten Elementen ein Problem. Während beim Löschen einfach alle Elemente gelöscht werden, ist die Erzeugung einer Kante zu einem allquantifizierten Element jedoch schon schwieriger zu lösen. Es ist festzulegen, ob ein Link zu einem Element der Menge oder zu allen Elementen der Menge erzeugt werden soll. Noch komplizierter wird die Erzeugung von Links zwischen zwei allquantifizierten Elementen. Hier ist zu klären, ob ein Link oder das komplette kartesische Produkt der Elemente mit Links verbunden werden soll. Aktuell existieren keine theoretischen Grundlagen zu Graphtransformationen, die beschreiben, wie diese Problematik zu lösen ist.

Die Verwendung von Allquantifizierungen in visuellen Kontrakten ist also nicht trivial, vor allem wenn eine inituitive und eingängige Interpretation der Kontrakte erwünscht ist. Zur Vermeidung dieser Probleme dürfen in visuellen Kontrakten keine allquantifizierten Rollen in der Nachbedingung angegeben werden, wenn diese nicht bereits in der Vorbedingung angegeben wurden. Allquantifizierte Rollen werden also entweder beibehalten oder gelöscht. Weiterhin dürfen allquantifizierte Rollen nur mit einer weiteren Rolle verbunden sein. Allquantifizierte Rollen stellen in einem visuellen Kontrakt also immer ein „Blatt“ dar.

3.5 Fazit

In diesem Kapitel ist die Syntax und Semantik der visuellen Kontrakte entwickelt worden. Mit den visuellen Kontrakten heben wir Design-by-Contract-Ideen, die üblicherweise auf der Implementierungsebene verwendet werden, auf die Modellebene. Visuelle Kontrakte können als Technik zur Spezifikation von Systemen, insbesondere von Operationen eingesetzt werden. Sie erlauben die Spezifikation von Zustandstransformationen mit Vor- und Nachbedingungen. Die Vor- und Nachbedingungen werden mit UML-Composite-Structure-Diagrammen beschrieben. Mit der Verwendung von UML-Diagrammen bauen unsere visuellen Kontrakte auf einem wohl bekannten Standard, der in heutigen Softwareentwicklungsprozessen eingesetzt wird, auf. Unsere visuellen Kontrakte sind somit für Softwareentwickler leicht verständlich und können in heutigen modellbasierten Softwareentwicklungsprozessen eingesetzt werden.

Die Syntax der visuellen Kontrakte wurde mit einer Erweiterung des UML 2.0 Metamodells definiert. Der semantische Bereich der Graphtransformationen hat sich als geeignet für die Definition der Semantik der visuellen Kontrakte erwiesen. Da mit den visuellen Kontrakten nur minimale Anforderungen an eine mögliche Implementierung gestellt werden, verwenden wir die lose Semantik der Graphtransitionen aus dem Double-Pullback-Ansatz.

Kapitel 4

Model-Driven Monitoring

Model-Driven Monitoring ist eine neue Strategie zur Unterstützung der modellgetriebenen (model-driven) Softwareentwicklung. Das Konzept der modellgetriebenen Softwareentwicklung ist in den letzten Jahren auf akademischer und industrieller Seite vielfach diskutiert worden. Das von der Object Management Group (OMG) eingeführte Konzept der Model-Driven Architecture [MF05, OMG01a, RSN03, SG00] hat diese Diskussion noch verstärkt. Vielfach dient in den Ansätzen der modellgetriebenen Softwareentwicklung ein Modell als Quelle für eine automatische Codegenerierung, die als Ausgabe den vollständigen, ausführbaren Code der modellierten Anwendung erzeugt.

Die Generierung von vollständigem, ausführbarem Code ist jedoch nicht immer realistisch und wirtschaftlich. Eine vollautomatisierte und damit standardisierte Abbildung von Modellen auf eine Implementierung kann zu unflexibel sein, insbesondere, wenn projektspezifische oder unternehmensspezifische Technologien, Bibliotheken, Pattern oder Code Conventions verwendet werden sollen. Auch Effizienzanforderungen können in diesem Zusammenhang eine Hürde sein. Weiterhin benötigt die Generierung von ausführbarem Code vollständige Modelle auf einer niedrigen Abstraktionsebene. Die Produktion derartiger Modelle ist in der Regel sehr kostenintensiv, da insbesondere Verhaltensbeschreibungen sehr feingranulare Modelle erfordern. Aufgrund dieser detaillierten Beschreibungen geht die in Modellen für ein besseres Verständnis gewünschte Abstraktion häufig verloren.

Bei unserem Ansatz des Model-Driven Monitoring ist es nicht das Ziel, aus Modellen vollständigen, ausführbaren Code zu generieren. Stattdessen stellen die Modelle in unserem Ansatz eine Spezifikation dar, die ein Softwareentwickler bei der manuellen Implementierung eines Systems zu berücksichtigen

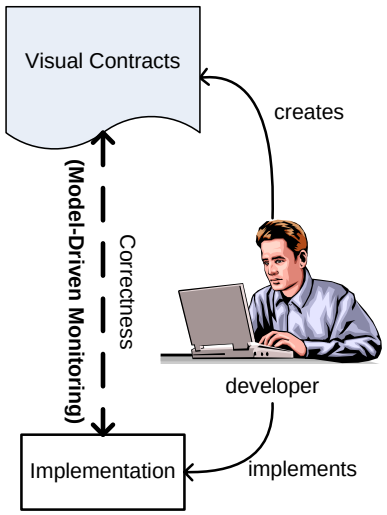


Abbildung 4.1: Model-Driven Monitoring mit visuellen Kontrakten

hat. Dadurch erhält ein Softwareentwickler auch die Möglichkeit zu bestimmen, wie detailliert die Modelle einer Anwendung sein sollen. Weiterhin können wir mit dem Model-Driven Monitoring sicherstellen, dass die manuelle Implementierung eines Systems korrekt gegenüber der Spezifikation — in unserem Fall korrekt gegenüber den visuellen Kontrakten — ist (vgl. Abbildung 4.1). Hierzu generieren wir aus den Modellen bzw. unseren visuellen Kontrakten Assertions, die zur Laufzeit das Verhalten der manuell erstellten Implementierung überwachen.

In diesem Kapitel beschreiben wir die Übersetzung der visuellen Kontrakte in die Java Modeling Language, einer Design-by-Contract-Erweiterung für Java. Die Basis ist eine Übersetzung von UML-Klassendiagrammen nach Java. Zum Abschluss des Kapitel skizzieren wir, wie unsere Vorgehensweise das Laufzeitverhalten einer Anwendung ändert und zum Testen von Anwendungen verwendet werden kann. Im Folgenden Abschnitt diskutieren wir jedoch zunächst verwandte Arbeiten.

4.1 Verwandte Arbeiten

Die Java Modeling Language (JML) [Iow, BCC+05, LBR05, LC04] ist eine Sprache zur Spezifikation von Java-Klassen und Schnittstellen. In der Literatur sind JML und vergleichbare Sprachen unter dem Begriff *Behavior Interface Specification Language* (*BISL*) bekannt. Mit JML können Invari-

anten für Klassen und Schnittstellen sowie Vor- und Nachbedingungen für Methoden spezifiziert werden. Dazu wird der Java-Code mit *Assertions* annotiert. Die Assertion-Sprache von JML basiert auf Java-Ausdrücken und folgt dem Beispiel der Design-by-Contract-Realisierung in Eiffel [Mey92b]. JML ist jedoch ausdrucksstärker als Eiffel, da verschiedene weitergehende Konstrukte wie Existenz- und Allquantifizierung unterstützt werden. JML basiert auf modellbasierten Grundlagen wie VDM (Vienna Development Method) [Jon90] und Z [Spi92], d.h. JML-Spezifikationen können abstrakt im Sinne von mathematischen Modellen wie Mengen und Relationen definiert werden. Jedoch werden die mathematischen Modelle in JML-Spezifikationen als Java-Klassen definiert, so dass der Anwender nicht noch eine andere Sprache wie z.B. Larch [GHG+93] lernen muss. Das Hauptargument für die Verwendung Java-basierter Notationen in JML ist, dass ein Anwender dieselbe Sprache zur Programmierung und Spezifikation verwenden kann. Aber, auch wenn die Notation ähnlich zu Java ist, so ist der Aufbau der Ausdrücke zur Spezifikation von Vor- oder Nachbedingungen doch sehr komplex und nicht intuitiv nachvollziehbar. Neben den Möglichkeiten zur Spezifikation von Vor- und Nachbedingungen existieren weiterhin eine Reihe von Werkzeugen zur Unterstützung von JML (ein Überblick ist in [BCC+05] zu finden). Dazu gehört unter anderem ein Compiler, der aus den JML-Annotationen Code erzeugt, um die Spezifikationen zur Laufzeit zu überprüfen [CL02a].

Hamie [Ham02, Ham04] schlägt ein Mapping von UML-Klassendiagrammen (auf Designebene), welche mit OCL annotiert sind, in mit JML annotierte Java-Klassen vor. In diesem Ansatz wird OCL verwendet, um die gewünschten Effekte einer Operation in Form von Vor- und Nachbedingungen zu beschreiben. Weiterhin erlaubt Hamie die Definition von Invarianten auf Klassen mit OCL-Constraints. Die unterschiedlichen OCL-Constraints werden dann in JML-Annotationen übersetzt. Bei der Übersetzung der JML-Konstrukte wird — ähnlich wie in unserem Ansatz — eine standardisierte Übersetzung der Klassendiagramme nach Java vorausgesetzt.

Das Dresden OCL Toolkit [HDF00, Dre, HDF02] ist in Java entwickelt worden und besteht aus mehreren Modulen. Ein OCL-Compiler [Fin00] erlaubt das Parsen, die Typüberprüfung und Normalisierung von OCL-Constraints. Ein Code-Generator übersetzt OCL-Constraints in Java-Code-Fragmente, die überprüfen können, ob die OCL-Constraints erfüllt sind. Als Eingabe bekommt der Java-Code-Generator UML/XMI-Dateien mit Klassendiagrammen und eine zusätzliche Textdatei mit OCL-Constraints. Intern übersetzt der Compiler die OCL-Constraints in eine normalisierte Subsprache der OCL, welche die Verwendung von mehrfach verwendeten Iteratoren, iterierende Ei-

genschaften ohne Deklarationen, die mehrfache Verwendung eines Variablennamen und die Verwendung des Standardkontextes für die Navigation vermeidet. Der Java-Code-Generator verwendet zur Generierung der Java-Code-Fragmente, die zur Überprüfung der OCL-Constraints verwendet werden können, einen regelbasierten Ansatz. Für die Implementierung der generierten Methodenrümpfe wird auf eine spezielle OCL Library [Fin99] zurückgegriffen. Diese Library enthält zum Beispiel Implementierungen von Standarddatentypen zur Verwaltung von OCL-Kollektionen. Zu beachten ist, dass die generierten Codefragmente nicht für den menschlichen Leser gedacht sind, sondern lediglich für die automatisierte Überprüfung der OCL-Constraints. Für die Abfrage des Systemzustandes, der durch die aktuelle Objektkonfiguration des Systems bestimmt wird, wurde eine Schnittstelle definiert, die auf dem Factory Design Pattern basiert. Mit dieser Schnittstelle wird es möglich, den generierten Assertion Code an verschiedene Java-Repräsentationen der mit OCL annotierten UML-Konstrukte anzupassen. Zur Verwendung der generierten Java-Code-Fragmente ist ein zusätzliches Tool für die Java-Code-Instrumentierung [Wie00, HDF02] entwickelt worden.

Auch wenn die beiden oben genannten Ansätze Code zur Überprüfung der OCL-Constraints generieren können, sind sie dennoch schwer in heutige modellbasierte Softwareentwicklungsprozesse zu integrieren. Der Grund ist, dass eine einfache visuelle Repräsentation der Constraints fehlt. Der Einsatz von OCL ist selbst in Unternehmen, die UML-Diagramme einsetzen, nur von sehr eingeschränktem Nutzen. Wichtige Gründe hierfür sind die Komplexität von OCL und die Schwierigkeiten der Integration einer reinen textuellen Sprache wie OCL in Diagramme (vgl. auch Abschnitt 3.1).

Die Fujaba Tool Suite [Sof] ist ein Open Source Case Tool, welches an der Universität Paderborn entwickelt wurde. Fujaba verwendet UML als visuelle Programmiersprache zur Generierung von Java Source Code. Die statischen Strukturen einer Anwendung werden mit UML-Klassendiagrammen dargestellt. Für die visuelle Beschreibung des Verhaltens einer Anwendung wurden Story-Diagramme eingeführt, die eine regelbasierte Spezifikation des Systemverhaltens ermöglichen. Story-Diagramme betten erweiterte UML-Kollaborationsdiagramme (Kollaborationsdiagramme der UML 1.5) in UML-Aktivitätendiagramme ein, wodurch eine prozedurale Spezifikation von Verhalten möglich wird. Kollaborationsdiagramme werden in diesem Fall als Notation für Graphtransformationsregeln verwendet. Im Gegensatz dazu sind Spezifikationen mit visuellen Kontrakten eher deklarativ. Weiterhin unterscheidet sich die Vorgehensweise zur Verwendung der Modelle in Fujaba von unserem Model-Driven-Monitoring-Ansatz grundsätzlich. Die Verhaltensmo-

delle in Fujaba repräsentieren Methodenrümpfe. Aus diesen Modellen kann vollständiger, ausführbarer Code generiert werden. Beim Model-Driven Monitoring wird aus den Verhaltensmodellen (visuellen Kontrakten) kein ausführbarer Code generiert. Stattdessen erzeugen wir Assertions, welche die manuell erstellte Implementierung eines Programmierers zur Laufzeit überwachen können.

Ähnlich wie in unserem Ansatz gehen die hier dargestellten verwandten Arbeiten davon aus, dass die Modelle und die Implementierung einer Anwendung auf derselben konzeptionellen Sicht der Problemdomäne basieren, d.h. sie verwenden eine gemeinsame Menge von Klassen und Assoziationen[1] (vgl. auch Abschnitt 2.2). Wie gezeigt, existieren bereits verschiedene Ansätze, die Code zur Überprüfung von OCL-Constraints erzeugen. Jedoch bieten diese Ansätze keine intuitive, visuelle Notation und sind somit nur schwer in heutige modellbasierte Softwareentwicklungsprozesse zu integrieren. Im Gegensatz zu diesen Ansätzen bieten wir mit unseren visuellen Kontrakten eine intuitive, visuelle Notation, die einfach in heutige Softwareentwicklungsprozesse integriert werden kann. Weiterhin können wir aus unseren visuellen Kontrakten Assertions generieren, die zur Laufzeit eine manuelle Implementierung überwachen.

4.2 Übersetzung von UML-Klassendiagrammen nach Java

Für die Übersetzung von UML-Klassendiagrammen in objektorientierte Programmiersprachen, wie z.B. Java oder C++, existieren verschiedene gut bekannte Techniken. Diese sind in unterschiedlichen Publikationen [JBR99, EHSW99a, EHSW99b, FNTZ98, GdCL03] veröffentlicht worden und werden in angepasster Form in fast allen UML Case Tools, wie z.B. Fujaba [Sof], Rational Rose [IBM] oder Together [Bor], zur Codegenerierung genutzt.

Die unterschiedlichen Ansätze basieren auf dergleichen grundlegenden Idee: Klassendefinitionen oder Datentypdefinitionen werden in entsprechende Klassen der Zielsprache übersetzt und Assoziationen werden in Referenzen zwischen den beteiligten Klassen und Datentypen übersetzt. Zur Erweiterung dieser grundlegenden Idee können Kardinalitäten von Assoziationen, Sichtbarkeiten von Attributen und Rollen an Assoziationsenden berücksichtigt

[1] Aufgrund der automatischen Codegenerierung ist das z.B. bei Fujaba selbstverständlich.

werden. Weiterhin muss festgelegt werden, wie spezielle UML-Konzepte auf der Ebene der Programmiersprache gehandhabt werden, wenn eine direkte Übersetzung nicht möglich ist, da der Programmiersprache entsprechende Konzepte fehlen. Ein Beispiel hierfür ist z.B. das Konzept der Mehrfachvererbung, welches in UML-Klassendiagrammen bekannt ist, in Java aber nicht.

Die Übersetzung der visuellen Kontrakte nach JML ist abhängig von der Übersetzung der UML-Klassendiagramme nach Java. Daher werden im Folgenden die wichtigsten Eigenschaften der Übersetzung von Klassendiagrammen nach Java kurz zusammengefasst:

- Jede Klasse aus einem gegebenen UML-Klassendiagramm wird direkt in eine entsprechende Java-Klasse übersetzt.

- Alle `private`- oder `protected`-Attribute einer UML-Klasse werden in `private`- oder `protected`-Attribute der entsprechenden UML-Klasse übersetzt. Dabei werden die entsprechenden Datentypen berücksichtigt. Gemäß den Java Coding Style Guides [Red00] werden `public`-Attribute in `private`-Java-Attribute übersetzt, welche über entsprechende `get`- und `set`-Methoden erreichbar sind. Damit wird auch die Kapselung einer Klasse sichergestellt. Ein direkter Zugriff auf die Attribute einer Klasse wird verhindert. Die Namen der primitiven Datentypen der UML (`Integer`, `String`, ...) müssen bei der Übersetzung der Klassendiagramme nach Java entsprechend der Java-Syntax angepasst werden (z.B. wird aus `Integer int`). Attribute mit einer Multiplizität größer als eins werden auf ein Referenzattribut abgebildet, welches einen Java-Containertyp zugewiesen bekommt.

- Die Operationen in einem UML-Klassendiagramm werden in Methodendeklarationen übersetzt und in die entsprechende Java-Klasse eingefügt. Auch hier müssen bei der Übersetzung syntaktische Modifikationen vorgenommen werden, damit die Methodendeklaration der Java Syntax entspricht.

- Vererbungsbeziehungen sind bei der Übersetzung von Klassendiagrammen nach Java auch zu berücksichtigen. Eine Klasse `B`, welche eine direkte Subklasse einer Klasse `A` ist, wird zu einer Java-Klasse `B`, welche von der Java-Klasse `A` erbt.

  Die Übersetzung der Mehrfachvererbung von UML nach Java wird hier nicht weiter betrachtet, da aktuelle objektorientierte Programmiersprachen keine Mehrfachvererbung erlauben und wir uns hier auf die Übersetzung von visuellen Kontrakten nach JML konzentrieren. Das heißt

wir gehen im Rahmen dieser Arbeit davon aus, dass auch auf der Modellebene nur Einfachvererbung verwendet wird. Hinweise zum Auflösen bzw. zur Simulation der Mehrfachvererbung in einem objektorientierten System befinden sich bereits in Abschnitt 3.4.1.

- Objektorientierte Programmiersprachen, so auch Java, bieten keine direkte Unterstützung der Syntax oder Semantik von UML-Assoziationen [Rum87]. Assoziationen müssen durch eine geeignete Kombination von Klassen, Attributen und Methoden implementiert werden (vgl. z.B. [Rum87, Nob96, Nob97, Amb01]). Am einfachsten können Assoziationen in objektorientierten Programmiersprachen implementiert werden, wenn ein Attribut zur Speicherung des Links und Accessor-Methoden zur Manipulation des Links angeboten werden. In anderen Ansätzen werden jedoch auch Java-Schnittstellen zur Implementierung von Assoziationen [HBR00] verwendet. Wir haben den ersten, bekannteren Ansatz zur Implementierung von Assoziationen gewählt.

  Die Übersetzung von Assoziationen nach Java ist abhängig von den Kardinalitäten an den Assoziationsenden. Bei einer zu-1 Assoziation zwischen zwei Klassen wird der entsprechenden Java-Klasse ein Klassenattribut mit dem Namen der Assoziation oder dem Rollennamen am Assoziationsende der referenzierten Klasse hinzugefügt. Als Typ bekommt dieses Attribut die referenzierte Klasse. Die Assoziation zwischen der Klasse `CartItem` und `Product` aus Abbildung 3.1 wird zum Beispiel auf Seiten der Klasse `CartItem` in ein `private`-Klassenattribut mit dem Namen `product` vom Typ `Product` übersetzt. Weiterhin werden die entsprechenden Zugriffsmethoden der Java-Klasse hinzugefügt. Bei bidirektionalen Assoziationen wird beiden Klassen ein Attribut zur Verwaltung der Referenz zwischen den Klassen hinzugefügt.

  Bei der Übersetzung von Assoziationen mit einer Kardinalität größer als eins, wird ein Referenzattribut in die Klasse eingefügt, dem als Typ eine Implementierung der Standard-Java-Schnittstelle `Collection` zugewiesen wird. Eine `Collection` repräsentiert eine Gruppe von Objekten. Insbesondere verwenden wir die Klasse `TreeSet`, die eine Implementierung der Subklasse `Set` von `Collection` darstellt. Der Datentyp `Set` hat den Vorteil, dass er im Gegensatz zu einfachen Kollektionen keine doppelten Elemente enthält.

  Zur Realisierung von qualifizierenden Assoziationen verwenden wir die Java-Klasse `HashMap`, welche eine Implementierung der Standard-Java-Schnittstelle `Map` darstellt. In einer `Map` kann kein Schlüssel mehrfach

```
1 public class CartItem {
2
3     private String cartItemID;
4     private String ASIN;
5     ...
6
7     private Product product;
8     ...
9
10    public String getCartItemID() {...}
11    public void setCartItemID(String cartItemID) {...}
12
13    public String getASIN() {...}
14    public void setASIN(String asin) {...}
15
16    public Product getProduct() {...}
17    public void setProduct(Product product) {...}
18
19    ...
20 }
```

Listing 4.1: Ausschnitt aus der Java-Klasse `CartItem` des OnlineShop-Beispiels

verwendet werden und jeder Schlüssel kann nur auf einen Wert abgebildet werden.

Zur Verwaltung der Container-Klassen generieren wir zusätzliche Zugriffsmethoden, welche das Hinzufügen oder Löschen von Elementen erlauben. Zum Beispiel werden der Klasse `OnlineShop` die Zugriffsmethoden `addProduct` und `removeProduct` hinzugefügt. Um zu überprüfen, ob ein Objekt bekannt ist, werden zusätzliche Methoden wie z.B. `hasProduct` generiert. Bei der Verwendung von qualifizierenden Attributen, kann auf die referenzierten Objekte mittels Schlüsseln zugegriffen werden. Hierfür werden den Klassen entsprechende Methoden wie z.B. `getProductByASIN` hinzugefügt.

Zur Sicherung der Konsistenz von Referenzen zwischen zwei Klassen, welche über eine Assoziation miteinander verbunden sind, verwenden wir verschiedene Algorithmen zur Implementierung der Zugriffsmethoden. Für Details sei der Leser auf [GdCL03, FNTZ98, Rum87, Nob96, Nob97] verwiesen.

Listing 4.1 zeigt beispielhaft die Übersetzung der UML-Klasse `CartItem` aus Abbildung 3.1. Die UML-Klasse enthält 4 Attribute sowie Assoziationen. In Listing 4.1 sind die Attribute `cartItemID` und `ASIN` gezeigt, welche einen eindeutigen Inhalt zur Referenzierung des Warenpostens und eines Produktes besitzen. Weiterhin zeigt das Beispiel die Übersetzung der Assoziation zwischen `CartItem` und `Product`. Der generierte Code in den Rümpfen der Methoden zur Verwaltung der Attribute und Assoziationen ist in dem Beispiel nicht gezeigt. Für den Zugriff auf die Attribute sind öffentliche `get`- und `set`-Methoden generiert worden.

Festzuhalten ist, dass die automatisch generierten Implementierungen der Methoden zum Abfragen von Attributen oder Assoziationen — nicht die Methoden zum Ändern von Attributen oder Assoziationen — keine Seiteneffekte verursachen. Die Methoden verändern den Zustand eines Systems nicht und können somit bei der Übersetzung der visuellen Kontrakte nach JML verwendet werden.

4.3 Übersetzung visueller Kontrakte nach JML

In diesem Abschnitt wird die Übersetzung der in Kapitel 3 eingeführten visuellen Kontrakte nach JML beispielhaft erläutert. Ein wesentliches Problem ist, dass für eine Vor- oder Nachbedingung ein Match auf einem Host- bzw. Systemgraphen (vgl. Abschnitt 3.4.2) zu finden ist. Ein Systemgraph beschreibt den Zustand eines Systems. Er besteht aus Objekten und Links zwischen den Objekten. Zur Laufzeit müssen also die Rollen in der Vor- oder Nachbedingung eines visuellen Kontraktes auf die Laufzeitobjekte, die den Zustand des Systems ausmachen, abgebildet werden. Die Laufzeitobjekte müssen die in den visuellen Kontrakten spezifizierten Eigenschaften besitzen, d.h. sie müssen die spezifizierten Rollen einnehmen können. Die Berechnung eines Matchings entspricht dem Problem der Teilgraphensuche.

Bei der Übersetzung der visuellen Kontrakte nach JML setzen wir voraus, dass die Klassendiagramme gemäß Kapitel 4.2 nach Java übersetzt worden sind. Alle Operationen, deren Verhalten mit einem visuellen Kontrakt spezifiziert worden sind, werden im Java-Code mit einer JML-Spezifikation, welche die Vor- und Nachbedingungen des visuellen Kontraktes wiedergibt, annotiert.

Die Bestimmung eines Match kann in unseren Ansatz auf eine Teilgraphensuche, die wir im Folgenden zunächst erläutern, zurückgeführt werden. Danach

beschreiben wir die grundlegende Struktur einer JML-Annotation und stellen eine Abbildung der visuellen Kontrakte nach JML vor.

4.3.1 Teilgraphensuche

Bei der Teilgraphensuche muss eine isomorphe Abbildung einer Vor- oder Nachbedingung in den Systemgraphen, der den Systemzustand mit Objekten und Links repräsentiert, gefunden werden. Dazu müssen alle Rollen einer Vor- oder Nachbedingung im Systemgraphen gefunden werden. Erfüllt eine Menge von Objekten im Systemgraphen, die in einer Vor- oder Nachbedingung spezifizierten Rollen, so werden die Objekte an die entsprechenden Rollen gebunden. Die Teilgraphensuche gehört zu den NP-vollständigen Problemen [Meh84]. In der Theorie der Graphtransformationen ist dieses Problem unter dem Begriff *Subgraph Isomorphism Problem* bekannt.

Generell spannen die Algorithmen für die Teilgraphensuche einen Suchbaum auf, der mittels eines Backtracking-Mechanismus durchlaufen wird. Auch wir verwenden diese Vorgehensweise, um die Rollen einer Vor-/Nachbedingung an Laufzeitobjekte zu binden. Die Knoten in einem Suchbaum repräsentieren einen Schritt bei der Teilgraphensuche. In jeder Ebene des Suchbaums wird eine Rolle an ein Laufzeitobjekt des Systems gebunden. Die Wurzel unseres Suchbaums stellt das Element `self` dar, welches in jeder Vor- oder Nachbedingung in unseren visuellen Kontrakten vorhanden sein muss. Das grundlegende Vorgehen zur Berechnung eines Match zeigt der Pseudocode-Algorithmus in Listing 4.2 überblicksweise.

Der Algorithmus bekommt als Eingabe eine UML-`Collaboration` (Kollaboration), die eine Vor- oder Nachbedingung in einem visuellen Kontrakt repräsentiert. Eine Kollaboration definiert eine Menge von Rollen und Beziehungen zwischen den Rollen (vgl. auch Abschnitt 3.3 sowie Abbildung 3.6). Für diese Kollaboration ist ein Match zu berechnen, d.h. in einem Systemgraphen sind eine Menge von Objekten zu suchen, welche die Rollen der Kollaboration einnehmen können. Hierzu unterscheidet der Algorithmus zwischen gebundenen Objekten des Systems (Menge `B` im Algorithmus) und ungebundenen Objekten des Systems. Die Menge `B` enthält alle Laufzeitobjekte, die einen Teil der gesuchten Kollaboration ausmachen und bereits gefunden wurden. Ein Laufzeitobjekt aus der Menge `B` nimmt also eine Rolle der Kollaboration ein. Weiterhin unterscheidet der Algorithmus zwischen den disjunkten Mengen `P` und `U`. Beide enthalten Elemente aus der dem Algorithmus übergebenen Kollaboration `rulepart`. Die Menge `P` enthält Elemente der übergebenen Kollaboration `rulepart` zu denen bereits Laufzeitobjekte, die eine Rolle aus

```
1 findMatch(rulepart : Collaboration)
2 // rulepart is a pre-, post-condition or a negative
3 // pre- or post-condition of a visual contract.
4
5 if !(this fulfillsRoleOf rulepart.self)
6         return NoMatchFound;
7
8 Set P := rulepart.self;
9 // set of processed elements of a collaboration.
10 // initialized with element self of visual contract.
11 Set B := this;
12 // set of bound runtime objects.
13 // initialized with runtime object this.
14 Set U := rulepart/rulepart.self;
15 // set of unprocessed elements of a collaboration.
16 Relate(rulepart.self, this);
17
18 while (U != empty) {
19   Set Temp = P;
20   while (Temp != empty) {
21     Element el = Temp.removeNext();
22     Set linkedElements = el.getLinkedElements();
23     while (linkedElements != empty) {
24       Element testElement = linkedElements.removeNext();
25       if !(testElement isElementOf P) {
26         Object o = Relate(el);
27         Set linkedObjects = o.getLinkedObjects();
28         while (linkedObjects != empty) {
29           Object testObject = linkedObjects.removeNext();
30           if (testObject fulfillsRoleOf testElement) {
31             B.add(testObject);
32             U.remove(testElement);
33             P.add(testElement);
34             Relate(testElement,testObject);
35           }
36         }
37       }
38     }
39   }
40 }
```

Listing 4.2: Algorithmus (Pseudocode) zur Berechnung eines Match für eine Kollaboration

der Kollaboration einnehmen können, gefunden wurden. Das heißt die Mengen `P` und `B` sind gleich groß. Die Elemente sind über eine Relation `Relate` miteinander in Beziehung gesetzt. Mit der Relation wird festgehalten, welche Rolle ein Laufzeitobjekt einnimmt. Die Menge `U` enthält alle Rollen einer Kollaboration zu denen noch kein Laufzeitobjekt gefunden wurde. Die Menge `P` wird mit dem Element `self` des visuellen Kontraktes initialisiert. Die Menge `B` wird mit dem Objekt `this` initialisiert, d.h. sie wird mit dem Objekt initialisiert, welches die aktuelle Operation ausführt. Weiterhin werden bei der Initialisierung `self` und `this` miteinander in Relation gesetzt. Die Menge `U` enthält alle Elemente des visuellen Kontraktes außer dem Element `self`.

In jeder Iteration wählt der Algorithmus ein Element aus der Menge `U` aus, welches über einen `Connector` mit einem Element aus der Menge `P` verbunden ist (Zeilen 19-24). Dann wird ein Laufzeitobjekt gesucht, welches die Rolle des ausgewählten Elementes einnehmen kann (Zeilen 25-30). Dabei wird überprüft, ob das Laufzeitobjekt die entsprechenden Links und Attributinhalte besitzt (Operation `fulfillsRoleOf`, Zeile 30). Wenn diese Bedingungen erfüllt sind, wird das ausgewählte Element aus der Menge `U` entnommen und in die Menge `P` eingefügt. Das gefundene Objekt wird in die Menge der gebundenen Objekte `B` eingefügt und in Relation zu dem bearbeiteten Element gesetzt.

Unser Algorithmus zur Berechnung eines Match arbeitet mit ähnlichen Matching Strategien wie Progres oder Fujaba (vgl. [Zün96, FNT98, FNTZ98]). Die Effizienz des Algorithmus zum Auffinden einer Kollaboration ist abhängig von der Anzahl der Iterationen, die benötigt werden, um zu entscheiden, ob eine Kollaboration im aktuellen Systemzustand gefunden werden kann oder nicht. Die Anzahl der Iterationen ist dabei abhängig von der Reihenfolge in welcher für die Elemente der Kollaboration überprüft wird, ob ein entsprechendes Laufzeitobjekt die Rolle einnehmen kann. Unser Algorithmus abstrahiert von einer Festlegung der Reihenfolge, d.h. es ist nicht deterministisch, welches Element als nächstes bei der Berechnung eines Matching überprüft wird.

Verschiedene Heuristiken zur Optimierung besagen, dass ein Algorithmus versuchen soll, ein Missmatch so früh wie möglich zu finden (*Force-Failure-Prinzip*, vgl. auch [Smi97]). Dies kann z.B. erreicht werden, indem Elemente, welche durch eine one-to-one oder durch eine qualifizierende Assoziationen aus der Menge der bereits bearbeiteten Elemente erreicht werden, zuerst getestet werden [Zün96]. In anderen Ansätzen wird die Worst-Case Execution Time zur Berechnung eines Match optimiert, indem die Reihenfolge, in wel-

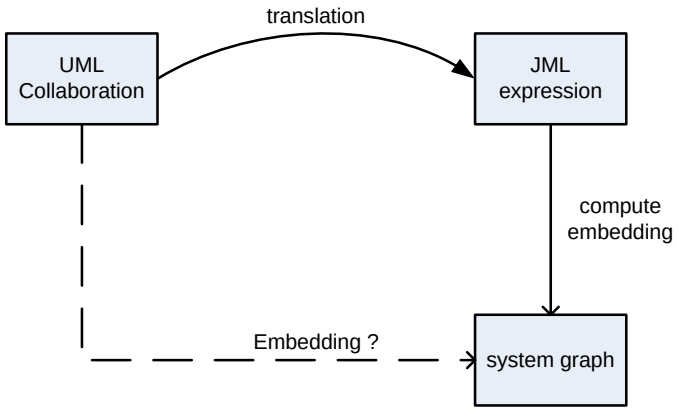


Abbildung 4.2: Übersetzung von UML `Collaboration`s in einen JML-Ausdruck zur Berechnung eines Match

cher die Elemente einer Kollaboration abgearbeitet werden, laufzeiteffizienter gestaltet werden [BGST05]. Diese Strategien sind in einer effizienten Implementierung unseres Ansatzes zu berücksichtigen, sie beeinflussen jedoch nicht die Korrektheit des dargestellten Algorithmus oder unserer Vorgehensweise.

Im Folgenden zeigen wir, wie ein visueller Kontrakt in einen JML-Ausdruck zur Annotation einer Operation übersetzt werden kann. Ein Problem dabei ist die Berechnung einer Einbettung einer Vor- oder Nachbedingung in einem Systemgraphen. Die Vor- oder Nachbedingung eines visuellen Kontraktes ist durch eine UML `Collaboration` gegeben. Wir zeigen, wie aus einer `Collaboration` ein spezifischer JML-Ausdruck zur Berechnung einer Einbettung bestimmt werden kann. Der JML-Ausdruck setzt Backtracking-Mechanismen ein und versucht dabei alle Rollen einer Kollaboration an Laufzeitobjekte zu binden, um eine Einbettung einer gegebenen `Collaboration` in den Systemgraphen zu berechnen (vgl. Abbildung 4.2).

4.3.2 Struktur von JML-Annotationen

Listing 4.3 zeigt den grundlegenden Aufbau einer Java-Methode, die mit einer JML-Spezifikation annotiert ist. Die Spezifikation besteht aus zwei Teilen, einem statischen und einem dynamischen Teil. Die Java-Deklaration der Methode selbst ist die statische Spezifikation der Schnittstelle. Das Verhalten der Methode wird mit einer Vor- und Nachbedingung in der JML-Annotation beschrieben. Die Verhaltensbeschreibung (JML-Annotation) der Methode ist eingebettet in die Java-Kommentare vor der Spezifikation der Methodensig-

```
 1 public class A implements {
 2
 3   ...
 4
 5   /*@ public normal_behavior
 6     @ requires JML-PRE;
 7     @ ensures JML-POST;
 8     @*/
 9   public Tr m(T1 v1, ... Tn vn) {...}
10
11   ...
12
13 }
```

Listing 4.3: Template zur Spezifikation von Vor- und Nachbedingungen mit JML

natur (in Listing 4.3 sind das die Zeilen 5-8 vor der Deklaration der Methode `m`). Ein normaler Java-Compiler berücksichtigt die JML-Spezifikationen somit nicht. Die JML-Spezifikation kann sich sowohl in einer separaten Datei als auch in den Java-Dateien selbst befinden.

Das Schlüsselwort `public normal_behavior` setzt die grundlegende Bedeutung der Spezifikation fest. Die Spezifikation ist für öffentliche Methoden („public“). Weiterhin darf die Implementierung der Operation (d.h. der von einem Programmierer entwickelte Methodenrumpf) keine Runtime Exceptions werfen, wenn der Client die Vorbedingung erfüllt [LBR05]. Grundlegend können in einer `public`-Spezifikation nur Variablen und Operationen verwendet werden, welche auch `public` sind[2]. Jedoch können sowohl Variablen als auch Operationen mit zusätzlichen JML-Constraints annotiert werden, um unabhängig von der Java-Deklaration festzulegen, ob diese im Rahmen der JML-Spezifikationen als `private`, `protected` oder `public` interpretiert werden sollen. Hierdurch wird es möglich die Attribute und Assoziationen unabhängig von ihrer im Klassendiagramm definierten Sichtbarkeit in einem JML-Ausdruck zu verwenden. Das heißt, wenn z.B. in einem visuellen Kontrakt einer öffentlichen Operation eine `private`-Variable verwendet wird, so kann diese auch in dem entsprechenden JML-Ausdruck zur Laufzeit überprüft werden.

[2]In einer `protected`-Spezifikation können Variablen und Operationen verwendet werden, die `public` oder `protected` sind. In einer `private`-Spezifikation können sämtliche Variablen und Operationen der Klasse verwendet werden.

Die Vorbedingung einer JML-Spezifikation folgt dem Schlüsselwort `requires`. Nachbedingungen folgen dem Schlüsselwort `ensures`. Beide Schlüsselwörter können in einer JML-Spezifikation mehrfach verwendet werden. In diesem Fall müssen alle spezifizierten Vor- bzw. Nachbedingungen erfüllt sein. Die Ausdrücke, welche den Schlüsselwörtern `requires` bzw. `ensures` folgen, stellen boolesche Ausdrücke dar, die mit einer erweiterten Notation für Java-Ausdrücke beschrieben werden. Die Vorbedingung macht Aussagen über den Zustand des Systems in Kombination mit den Parametern eines Methodenaufrufs. Wenn die Vorbedingung erfüllt ist, muss die Methode das System in einem Zustand zurücklassen, welcher die Nachbedingung erfüllt.

Beim Einsatz von Vererbung kann es vorkommen, dass eine Methode von einer Subklasse überschrieben wird. In diesem Fall muss die Methode, welcher ein neuer Kontrakt zugewiesen wurde, sowohl den Kontrakt der übergeordneten Klasse als auch den zusätzlichen Kontrakt erfüllen, d.h. die Methode erbt den Kontrakt von der Superklasse. In JML wird die Vererbung von Kontrakten durch das Schlüsselwort `also` ausgedrückt werden.

4.3.3 Übersetzung visueller Kontrakte nach JML: Intuitiver Ansatz

Bei einer Übersetzung der visuellen Kontrakte nach JML müssen `JML-PRE` und `JML-POST` (siehe Listing 4.3) korrekte Interpretationen der Vor- und Nachbedingung (einschließlich der negativen Vor- und Nachbedingungen) eines visuellen Kontraktes sein. Das bedeutet, dass eine JML-Vorbedingung (Nachbedingung) die in der Vorbedingung (Nachbedingung) des visuellen Kontraktes spezifizierte Kollaboration in der Menge von Objekten, welche den aktuellen Systemzustand ausmachen, finden muss.

Um das Auftreten einer Kollaboration zu finden, startet ein JML-Ausdruck die Berechnung eines Match bei dem aktiven Objekt `this` (Element `self` im visuellen Kontrakt), welches die aktuelle Methode ausführt. Wenn eine JML-Vorbedingung (Nachbedingung) erfolgreich eine Kollaboration findet wird `true` zurückgegeben, ansonsten `false` (vgl. auch Abschnitt 4.3.1). Im Folgenden wird der Aufbau der JML-Annotationen zum Auffinden einer Kollaboration erläutert.

Bei der Übersetzung nach JML führt der in Abschnitt 4.3.1 dargestellte Algorithmus zu ineinander geschachtelten „Schleifen“ für jede Rolle der Kollaboration, die gefunden werden soll. In jeder Schachtelung wird eine bereits gefundene Teillösung um ein weiteres Objekt erweitert. Der Startpunkt der

```
1 /*@ public normal_behavior
2   @ requires (
3   @    (\exists Cart c;
4   @     this.getCarts().values().contains(c);
5   @     c.getCartId().equals(cid)));
6   @*/
7 public String cartAdd (String cid, String ASIN,
8                         int quantity);
```

Listing 4.4: Erste Schachtelung der JML-Spezifikation für Vorbedingung

Suche nach der Kollaboration ist das Objekt `this`, welches die aktuelle Operation ausführt. Listing 4.4 zeigt die erste Schachtelung der JML-Konstrukte für den visuellen Kontrakt der Operation `cartAdd` aus Abbildung 3.3.

In der ersten Schachtelung werden die ausgehenden Links des aktiven Objekts `this:OnlineShop` überprüft. Als erstes wird getestet, ob das Objekt `this` ein Objekt vom Typ `Cart`, welches den Attributinhalt `cid` für das Attribut `cartId` hat, kennt. Der Wert der Variablen `cid` wird durch den Übergabeparameter `cid` bestimmt, d.h. die Variable wird zur Laufzeit beim Aufruf der Operation initialisiert.

Für die Suche nach einem Objekt eines bestimmten Typs können die Existenz- (`exists`) und Allquantoren (`forall`) von JML eingesetzt werden, die eine Erweiterung der von Java bekannten logischen Ausdrücke darstellen. Die generelle Syntax dieser quantifizierenden Ausdrücke ist `(\forall T x; r; p)` bzw. `(\exists T x; r; p)`. Der Ausdruck `(\forall T x; r; p)` ist wahr, wenn alle Objekte `x` vom Type `T`, welche `r` erfüllen auch `p` erfüllen. Der Ausdruck `(\exists T x; r; p)` ist wahr, wenn es mindestens ein Objekt `x` vom Typ `T` gibt, welches `r` und `p` erfüllt.

Mit Hilfe der `exists`-Ausdrücke kann der JML-Ausdruck in Listing 4.4 erzeugt werden. Dem Schlüsselwort `requires` folgt ein `exists`-Quantor, der erfüllt ist, wenn es mindestens ein Objekt vom Typ `Cart` gibt, welches vom Objekt `this:OnlineShop` aus erreichbar ist und den Attributinhalt `cid` für das Attribut `cartItem` hat. Mit `this.getCarts().values().contains(c)` wird überprüft, ob das Objekt `this` das Objekt `c` kennt. Hierzu gibt die Operation `getCarts()` eine `HashMap` zurück. Auf der `HashMap` kann die Operation `values` aufgerufen werden, die eine `Collection` zurückgibt. Diese kennt eine Operation `contains`, die `true` zurückgibt, falls das Element in der `Collection` enthalten ist. Wird ein entsprechendes Objekt gefunden, so wird es gemäß des Algorithmus aus Abschnitt 4.3.1 zur Menge `B` der gefun-

denen Objekte hinzugefügt oder anders gesprochen: ein gefundenes Objekt wird an die Rolle bzw. Variable `c` gebunden.

Listing 4.5 zeigt den vollständigen JML-Ausdruck zur Überprüfung des visuellen Kontraktes der Operation `cartAdd` aus Abbildung 3.3. Im Wesentlichen besteht die Vor- oder die Nachbedingung des JML-Ausdrucks aus ineinander geschachtelten Existenzquantoren. Jeder JML-Quantor eines JML-Ausdrucks bindet eine Rolle einer Vor- oder Nachbedingung an ein Laufzeitobjekt. Ist ein Laufzeitobjekt gebunden worden, so muss die nächste Rolle an ein Laufzeitobjekt gebunden werden. Dies erreichen wir durch eine Schachtelung der JML-Quantoren. Gleichzeitig wird durch die Schachtelung der Quantoren die Umsetzung des in Abschnitt 4.3.1 beschriebenen Backtrackings zur Bestimmung eines vollständigen Match für eine Vor- oder Nachbedingung umgesetzt, da jeder Existenzquantor eine Menge von Objekten in Abhängigkeit des übergeordneten Quantors durchläuft. Im Folgenden beschreiben wir die Vorbedingung des JML-Ausdrucks in Listing 4.4 im Detail.

In den Zeilen 6-8 wird getestet, ob das Objekt `this` ein Objekt vom Typ `Product` mit dem Attributinhalt `ASIN` (siehe Übergabeparameter der Operation) für das Attribut `ASIN` kennt. Auch diese Überprüfung erfolgt wiederum mit einem `exists`-Quantor, welcher in den zuvor beschriebenen quantifizierenden Ausdruck für die Suche nach einem Objekt vom Typ `Cart` geschachtelt wird (siehe `&&` am Ende der Zeile 5).

Die Zeilen 3-8 testen die positive Vorbedingung. In den Zeilen 9-12 findet die Überprüfung der negativen Vorbedingung des visuellen Kontraktes aus Abbildung 3.3 statt. In den Zeilen 10-12 wird getestet, ob das zuvor gefundene Objekt `c` ein Objekt vom Typ `CartItem` kennt, welches den Attributinhalt `ASIN` für das Attribut `ASIN` besitzt. Der `exists`-Ausdruck wird mit einer Negation versehen (Zeile 9). Das heißt es wird überprüft, ob nicht schon ein Warenposten für das bestellte Produkt besteht. Die gesamte negative Anwendungsbedingung wird in den letzten `exists`-Quantor der positiven Vorbedingung geschachtelt. Bei einer nicht korrekten Schachtelung der Quantoren, könnte innerhalb der negativen Vorbedingung nicht auf das zuvor gefundene Objekt `c` zurückgegriffen werden, da in einem JML-Quantor definierte Variablen nicht außerhalb von diesem bekannt sind.

Der Aufbau der Nachbedingung in Listing 4.5 ist im Prinzip identisch zur Vorbedingung. Es muss lediglich die Existenz eines weiteren Objektes (vom Typ `CartItem`) getestet werden. Weiterhin muss keine negative Nachbedingung überprüft werden. In der Nachbedingung sind zwei Besonderheiten zu beachten. Mit dem JML-Ausdruck `\result` in Zeile 28 kann auf den Rück-

```
1 /*@ public normal_behavior
2   @ requires (
3   @   (\exists Cart c;
4   @     this.getCarts().values().contains(c);
5   @     c.getCartId().equals(cid) &&
6   @     (\exists Product pr;
7   @       this.getProducts().values().contains(pr);
8   @       pr.getASIN().equals(ASIN) &&
9   @       !(
10  @         (\exists CartItem citemnac;
11  @           c.getCartItems().contains(citemnac);
12  @           citemnac.getASIN().equals(ASIN)
13  @ ))))));
14  @
15  @ ensures (
16  @   (\exists Cart c;
17  @     this.getCarts().values().contains(c);
18  @     c.getCartId().equals(cid) &&
19  @     (\exists Product pr;
20  @       this.getProducts().values().contains(pr);
21  @       pr.getASIN().equals(ASIN) &&
22  @       (\exists CartItem citem;
23  @         c.getCartItems().contains(citem);
24  @         citem.getASIN().equals(ASIN) &&
25  @         citem.getQuantity() == quantity &&
26  @         citem.getProduct() == pr &&
27  @         \fresh(citem) &&
28  @         citem.getCartItemId.equals(\result)
29  @ ))));
30  @*/
31 public String cartAdd (String cid, String ASIN,
32                         int quantity);
```

Listing 4.5: Vollständiger JML-Ausdruck zur Überprüfung des visuellen Kontraktes der Operation `cartAdd` aus Abbildung 3.3

```
1 public class OnlineShop {
2
3     private HashMap products;
4     private HashMap carts;
5
6     /*@ public normal_behavior
7       @
8       @ ensures (
9       @   (\exists Object c;
10      @     getCarts().values().contains(c);
11      @     \fresh(c)
12      @    && c == \result
13      @ ));
14      @*/
15    public Cart cartCreate();
16
17    public /*@ pure @*/ HashMap getProducts();
18
19    public /*@ pure @*/ HashMap getCarts();
20
21    ...
22 }
```

Listing 4.6: Ausschnitt aus der JML-Datei zur Spezifikation der visuellen Kontrakte

gabewert der Operation zugegriffen werden. Mit dem JML-Ausdruck `\fresh` (Zeile 27) kann überprüft werden, ob ein Objekt neu angelegt wurde.

Listing 4.6 zeigt einen Ausschnitt der JML-Datei zur Spezifikation des Verhaltens der Operationen der Klasse `OnlineShop`. Neben den Spezifikationen für die Operation `cartCreate`, deren Verhalten mit einem visuellen Kontrakt spezifiziert ist (Abbildung 3.2) enthält die Datei JML-Spezifikationen für weitere Methoden, die nicht mit einem visuellen Kontrakt beschrieben wurden. Insbesondere müssen die Java-Methoden, die innerhalb der JML-Spezifikationen (Operationen `getProducts` und `getCarts` in den Zeilen 15 und 17) verwendet werden, als `pure` gekennzeichnet werden. Damit kann einem JML-Compiler oder einem anderem JML-Werkzeug mitgeteilt werden, dass die Methoden keine Seiteneffekte verursachen. Sind die in einer JML-Assertion verwendeten Operationen nicht als `pure` gekennzeichnet, so werden die Assertions von einem JML-Werkzeug nicht akzeptiert. Allerdings überprüft der JML-Compiler nicht, ob eine als `pure` gekennzeichnete Operation

tatsächlich keine Seiteneffekte auf den Systemzustand hat, d.h. der JML-Compiler geht von einer korrekten Kennzeichnung der Operationen aus. In den JML-Assertions zur Repräsentation eines visuellen Kontraktes verwenden wir jedoch nur automatische generierte Methoden. Für diese Methoden stellen wir mit unserer Codegenerierung sicher, dass sie keine Seiteneffekte haben.

4.3.4 Detaillierung der intuitiven Übersetzung

Der JML-Kontrakt aus Listing 4.5 interpretiert die Vor- und die Nachbedingung des visuellen Kontraktes aus Abbildung 3.3 als komplexe boolesche Ausdrücke. Dabei werden in dem JML-Kontrakt nicht die Zusammenhänge zwischen der linken und rechten Regelseite eines visuellen Kontraktes betrachtet. Jede Seite wird als ein individueller Ausdruck interpretiert.

Die individuelle Auswertung der linken und rechten Seite eines Kontraktes entspricht jedoch nicht der Semantik visueller Kontrakte (vgl. Abschnitt 3.4). In einem visuellen Kontrakt bestimmen Variablen für Attributinhalte oder Rollennamen identische Inhalte bzw. identische Objekte in den Vor- und Nachbedingungen. Nur so können Beziehungen zwischen der Vor- und Nachbedingung eines visuellen Kontraktes hergestellt werden. Die mit den JML-Ausdrücken umgesetzte Teilgraphensuche muss also auch die Abhängigkeiten zwischen den Vor- und Nachbedingungen berücksichtigen. Dieses Problem tritt in anderen Ansätzen für Graphtransformationen (z.B. in AGG [Tae04] oder Fujaba [Sof]) nicht in dieser Form auf. In diesen Ansätzen muss nur ein Match für die Vorbedingung bestimmt werden. Dieser Match wird dann mit der Nachbedingung ersetzt (vgl. auch Abschnitt 3.4.2), wodurch implizit auch eine Beziehung zwischen der linken und rechten Seite einer Graphtransformation hergestellt wird.

Mit dem alleinigen Einsatz von JML-Existenz- und Allquantoren kann die korrekte Interpretation jedoch nicht sichergestellt werden. Mit den Quantoren kann nicht ausgedrückt werden, dass ein Objekt, welches in der Vorbedingung gefunden wurde, auch in der Nachbedingung an dieselbe Variable gebunden wird. Für die JML-Annotation aus Listing 4.5 bedeutet das zum Beispiel: ein Objekt, welches in der Vorbedingung an die Variable `pr` gebunden wird, muss nicht identisch sein mit einem Objekt, welches in der Nachbedingung an die Variable `pr` gebunden wird. Ein Binding, welches im Rahmen von quantifizierenden Ausdrücken in einer Vorbedingung dynamisch gefunden wird, kann in der Nachbedingung nicht wieder verwendet werden. Für dieses Problem sind unterschiedliche Lösungen denbkar. Drei mögliche

Lösungen wollen wir im Folgenden kurz beschreiben . Das Ziel ist es, in einer Assertion weitestgehend nur JML-Konstrukte zu verwenden. Nur so kann auf existierende JML-Tools, die zum Teil sogar eine statische Codeanalyse erlauben, zurückgegriffen werden. Weiterhin sollen die generierten JML-Assertions soweit wie möglich manuell auswertbar sein. Hierzu ist es hilfreich, wenn eine JML-Assertion in sich geschlossen ist, d.h. die Auswertung einer Assertion sollte nicht auf mehreren Stellen im generierten Quellcode verteilt sein.

Verwendung von Schlüsseln zur eindeutigen Identifikation von Objekten

Eine mögliche Lösung ist die Kennzeichnung ausgewählter Attribute einer Klasse als Schlüssel. Die Instanzen der Klassen sind dann eindeutig über ihre Schlüsselattribute identifizierbar. In dem Klassendiagramm unseres Onlineshops (siehe Abbildung 3.1) haben wir zum Beispiel für die Klassen `Product`, `Cart` und `CartItem` Schlüsselattribute bestimmt. Wenn die zu suchenden Objekte in den Existenzquantoren einer JML-Spezifikation über ihre Schlüsseleigenschaft identifiziert werden, wie das auch in dem Beispiel in Listing 4.5 der Fall ist, so können die Objekte eindeutig bestimmt werden und obige Diskussion wird obsolet. Die Suche über die Schlüsselattribute stellt sicher, dass in den Vor- und Nachbedingungen identische Objekte gefunden werden. Der Zusammenhang zwischen der Vor- und Nachbedingung eines visuellen Kontraktes wird damit auch auf der Ebene der Implementierung sichergestellt.

Verwendung von JML-Modellmethoden

Wenn die in einem visuellen Kontrakt beschriebenen Objekte jedoch nicht eindeutig über einen Schlüssel identifiziert werden können, so hat das zur Konsequenz, dass mehr als ein Subgraph mit Objekten, welche den Rollen in dem visuellen Kontrakt entsprechen, gefunden werden kann. Abbildung 4.3 zeigt einen möglichen Ausschnitt eines Systemzustandes des in Kapitel 3 eingeführten Onlineshops, wenn keine eindeutigen Schlüssel zur Identifizierung der Objekte vergeben werden. In diesem Fall kann die ASIN zweier Produkte identisch sein. Beim Aufruf der Operation `cartCreate` mit der `ASIN QX1234` und der `cid GH27` kann die Vorbedingung — ohne die negative Anwendungsbedingung näher zu betrachten — auf zwei Arten in den Instanzgraphen aus Abbildung 4.3 eingebettet werden : $o_{L1} = \{\{self, this\}, \{pr, pr1\}, \{c, cart\}\}$ und $o_{L2} = \{\{self, this\}, \{pr, pr2\}, \{c, cart\}\}$. In diesem Fall ist nicht sicher-

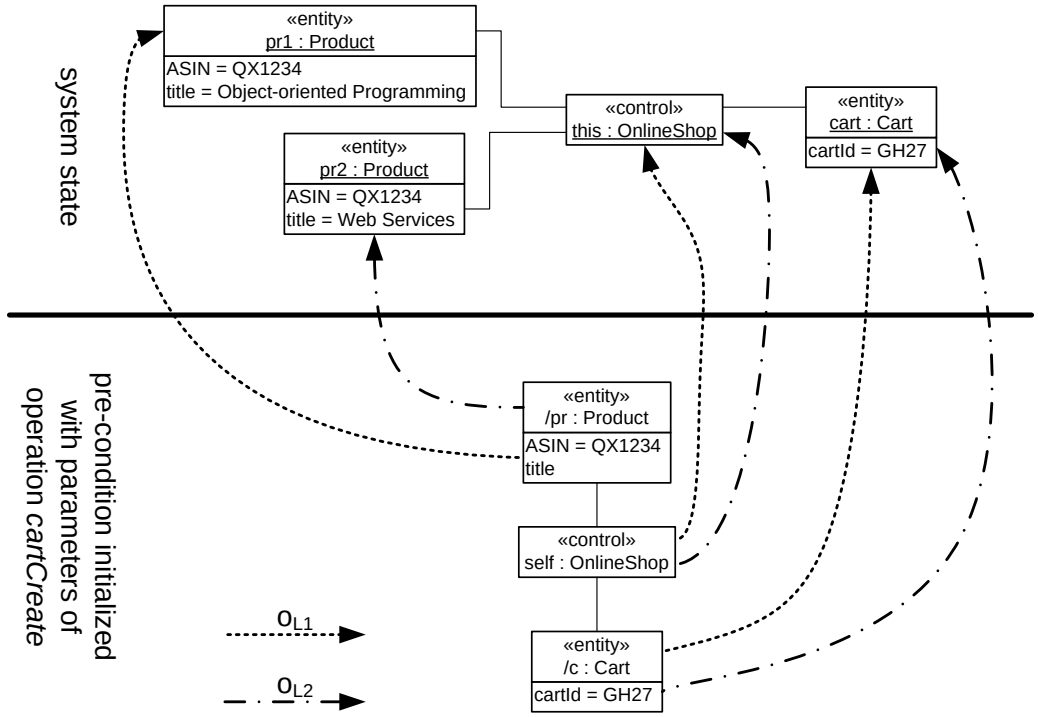


Abbildung 4.3: Systemzustand mit zwei möglichen Einbettungen

```
1 /*@ public normal_behavior
2   @ old Cart c = findCartByCartID(cid);
3   @ old Product pr = findProductByASIN(ASIN);
4   @
5   @ requires c != null;
6   @ requires pr != null;
7   @
8   @ ensures c != null;
9   @ ensures pr != null;
10  @ ensures \not_modified(pr);
11  @ ensures (
12  @   (\exists CartItem citem;
13  @     c.getCartItems().values().contains(citem);
14  @     citem.getASIN().equals(ASIN) &&
15  @     citem.getQuantity() == quantity &&
16  @     \fresh(citem) &&
17  @     citem.getCartItemId.equals(\result)
18  @ ));
19  @*/
20 public String cartAdd (String cid, String ASIN,
21                         int quantity);
```

Listing 4.7: Bindung von Objekten an Variablen mit JML `old`

gestellt, dass die JML-Annotation sowohl in der Vorbedingung als auch in der Nachbedingung dasselbe Binding berechnet.

Eine mögliche Lösung, um sicherzustellen, dass sowohl in der Vor- als auch in der Nachbedingung dasselbe Binding verwendet wird, ist die Verwendung des JML-Konstruktes `old`. Damit können Objekte oder Inhalte an Variablen gebunden werden. Listing 4.7 (Zeilen 2 und 3) zeigt, wie der Befehl `old` in einer JML-Assertion verwendet werden kann. Wie dargestellt kann mit dem `old`-Befehl einer Variablen ein Wert zugewiesen werden. Der Ausdruck wird vor der Ausführung der Operation und auch vor der Auswertung der Vorbedingung ausgewertet. Im Folgenden erläutern wir das Beispiel aus Listing 4.7 im Detail.

Zum Binden der Variablen an Objekte, welche die in einem visuellen Kontrakt spezifizierten Rollen einnehmen, müssen spezifische Methoden aufgerufen werden, die zum Beispiel auf Basis der Attributinhalte nach den Objekten suchen. Die aus den Klassendiagrammen generierten Accessor-Methoden sind hierfür nicht ausreichend. Es müssen also zusätzliche Methoden gene-

```
1 /*@
2   @ public Product findProductByASIN(String ASIN) {
3   @   Iterator it = getProducts().values().iterator();
4   @   while (it.hasNext()) {
5   @     Product pr = (Product)it.next();
6   @     if (pr.getASIN().equals(ASIN))
7   @       return pr;
8   @   }
9   @     return null;
10  @   }
11  @*/
```

Listing 4.8: Modellmethode `findProductByASIN` der Klasse `OnlineShop`

riert werden, die in den Kontrakten für die Suche nach Objekten eingesetzt werden können. Da diese Methoden sehr spezifisch sind, sollten sie dem Programmierer nach Möglichkeit nicht zugänglich sein, d.h. die Methoden sollten nicht Teil des normalen Java-Codes sein.

Hierfür bietet JML *Modellmethoden* (*model methods*) an. Modellmethoden sind normale Java-Methoden, die innerhalb von Java-Kommentaren definiert werden, so dass diese nur innerhalb einer JML-Spezifikation verwendet werden können. Sie sind nicht Teil des regulären Java-Codes. Mit Hilfe dieser JML-Modellmethoden kann die volle Ausdrucksfähigkeit von Java innerhalb von JML-Assertions verwendet werden. Zur Verwendung dieser Vorgehensweise bei einer automatischen Codegenerierung können eine Menge von Modellmethoden-Templates spezifiziert werden, die bei der JML-Codegenerierung geeignet instanziiert werden müssen. Diese Modellmethoden müssen es ermöglichen, Objekte mit bestimmten Eigenschaften zu finden, z.B. mit bestimmten Attributinhalten. Ein Beispiel für eine derartige Modellmethode ist die Methode `findProductByASIN` aus Listing 4.8. Diese Methode ermöglicht die Suche nach Produkten mit einer bestimmten `ASIN`. Auch diese Modellmethoden können zusätzlich mit JML-Kontrakten annotiert werden.

Listing 4.7 (Zeilen 2 und 3) zeigt den Aufruf der Modellmethoden. Das Ergebnis wird an die Variable `c` bzw. an die Variable `pr` gebunden. Nachdem wir mittels des `old`-Ausdrucks ein Binding erzeugt haben, welches sowohl in der Vor- als auch in der Nachbedingung verwendet werden kann, kann in der Vorbedingung getestet werden, ob ein Objekt mit den gegebenen Charakteristiken gefunden wurde. Hierbei muss in der Vorbedingung lediglich noch überprüft werden, ob die Variablen von den Modellmethoden belegt wurden

```
1 /*@ public normal_behavior
2   @
3   @ old Vector embeddings =
4   @             this.findEmbeddings(cid, ASIN, quantity);
5   @
6   @ requires embeddings != null;
7   @
8   @ ensures embeddings != null;
9   @ ensures this.testEmbedding(embeddings) == true;
10  @*/
11 public String cartAdd (String cid, String ASIN,
12                         int quantity);
```

Listing 4.9: Verwaltung von nichtdeterministischen Einbettungen

(siehe Zeile 5 und 6 in Listing 4.7), da die Modellmethoden die Variablen gemäß den Vorgaben aus der Vorbedingung eines visuellen Kontraktes belegen. In der Nachbedingung müssen die Links, Attribute und neu erzeugten Objekte gemäß den Vorgaben der Nachbedingung überprüft werden. Der JML-Ausdruck `not_modified` (Zeile 10) legt fest, dass die Werte der Objekte in der Vorbedingung dieselben sein müssen wie in der Nachbedingung.

Allerdings ist diese Lösung noch nicht ausreichend. Es wird zwar sichergestellt, dass die Vor- und Nachbedingung dasselbe Binding benutzen, jedoch kann es aufgrund der zwei möglichen Einbettungen der Vorbedingungen in den aktuellen Systemzustand zu Inkonsistenzen zwischen den JML-Spezifikationen und der manuellen Implementierung kommen. Auf der einen Seite ist es möglich, dass während der Auswertung der Vorbedingung aus Listing 4.7 die Einbettung o_{L1} in den Variablen `c` und `pr` gespeichert wird (vgl. Abbildung 4.3). Dann wird gemäß dem visuellen Kontrakt aus Abbildung 3.3 angenommen, dass dem Objekt `cart` ein Objekt vom Typ `CartItem` hinzugefügt wird, welches wiederum mit dem gebundenen Produkt `pr1` verlinkt wird. Andererseits kann die manuell erstellte Implementierung die Einbettung o_{L2} (siehe auch Abbildung 4.3) auswählen und ein Objekt vom Typ `CartItem` erzeugen, welches mit dem Produkt `pr2` verbunden wird. In diesem Fall wird die Auswertung der Nachbedingung einen Fehler in der Implementierung anzeigen, obwohl diese eigentlich korrekt ist. Das Problem ist, dass die manuelle Implementierung zur Laufzeit bei ihrem Start eine andere Einbettung ausgewählt hat als die JML-Vorbedingung.

Dieses Problem kann behoben werden, indem vor der Ausführung der Operation alle möglichen Einbettungen der Vorbedingung des visuellen Kontraktes

gespeichert werden. Zu diesem Zweck können auch wiederum entsprechende Modellmethoden generiert werden, die alle möglichen Einbettungen finden und diese in einem Array, das mittels des JML-Konstruktes `old` an eine Variable gebunden wird, speichern. Listing 4.9 zeigt ein mögliches Beispiel. Alle gefundenen Einbettungen werden in der Variablen `embeddings` gespeichert (Zeile 3). Die Methode `findEmbeddings` (Zeile 4) sucht die entsprechende Einbettung. Die Realisierung dieser Methode kann ähnlich zu den Graph-Pattern-Matching-Algorithmen in Fujaba oder Progress erfolgen. Nach der Ausführung der Operation, muss auch für die Nachbedingung überprüft werden, ob eine entsprechende Einbettung existiert, welche die Nachbedingung des visuellen Kontraktes erfüllt. Dies kann auch durch eine Modellmethode erreicht werden, welche als Eingabe die in der Vorbedingung berechneten Einbettungen bekommt.

Die Verwendung von Modellmethoden zum Auffinden einer Vor- oder Nachbedingung eines visuellen Kontraktes in einem Systemgraphen bietet zwar eine hohe Flexibilität, jedoch werden bei dieser Lösung kaum existierende JML-Konstrukte verwendet. Sämtliche Funktionalität zur Berechnung eines Match liegt in den generierten JML-Modellmethoden. Ein Nachteil dieser Lösung ist, dass die verschiedenen Werkzeuge, welche für mit JML annotiertem Code existieren, kaum genutzt werden können, da die in den Modellmethoden generierten Ausdrücke einfach zu komplex sind. Weiterhin ist die Auswertung der generierten Assertions auf eine Vielzahl verschiedener Stellen im Quellcode verteilt, so dass ein Softwareentwickler kaum in der Lage ist, die Assertions zu verstehen. So enthält z.B. die JML-Assertion aus Listing 4.9 keine Informationen darüber, was in der Vor- oder Nachbedingung getestet wird. Um zu bestimmen, wie die Assertions funktionieren, müsste ein Softwareentwickler noch die in der Assertion aufgerufenen Methoden analysieren. Bei einer Verwendung dieser Vorgehensweise könnte theoretisch sogar vollständig auf die Verwendung der Sprache JML verzichtet werden. Stattdessen können existierende Ansätze der aspektorientierten Programmierung verwendet werden, um automatisch generierten Java-Code zur Überprüfung der Vor- und Nachbedingungen in den manuell implementierten Code zu weben [MW99]. Dies würde jedoch auch die Komplexität eines CASE-Tools zur Unterstützung unserer Methodik erhöhen. Wir konzentrieren uns daher auf eine weitere Variante zur Überprüfung der visuellen Kontrakte, die vollständig auf JML-Assertions beruht, um die Verwendung existierender JML-Tools zu ermöglichen.

```
1 /*@ public normal_behavior
2   @ requires (
3   @   (\exists Cart c;
4   @     this.getCarts().values().contains(c);
5   @     c.getCartId().equals(cid) &&
6   @     (\exists Product pr;
7   @       this.getProducts().values().contains(pr);
8   @       pr.getASIN().equals(ASIN) &&
9   @       !(
10  @         (\exists CartItem citemnac;
11  @           c.getCartItems().contains(citemnac);
12  @           citemnac.getASIN().equals(ASIN)
13  @ ))))));
14  @
15  @ ensures (
16  @   (\exists Cart c;
17  @     \old(this.getCarts().values()).contains(c);
18  @     \old(c.getCartID()).equals(cid) &&
19  @     c.getCartID().equals(cid) &&
20  @     (\exists Product pr;
21  @       \old(getProducts().values()).contains(pr);
22  @       \old(pr.getASIN()).equals(ASIN) &&
23  @       pr.getASIN().equals(ASIN)&&
24  @       !(
25  @         (\exists CartItem citemnac;
26  @           \old(c.getCartItems()).contains(citemnac);
27  @           \old(citemnac.getASIN()).equals(ASIN)
28  @         )) &&
29  @       (\exists CartItem citem;
30  @         c.getCartItems().contains(citem);
31  @         citem.getASIN().equals(ASIN) &&
32  @         citem.getQuantity() == quantity &&
33  @         citem.getProduct() == pr &&
34  @         \fresh(citem) &&
35  @         citem.getCartItemId.equals(\result)
36  @ ))));
37 public String cartAdd (String cid, String ASIN,
38                         int quantity);
```

Listing 4.10: JML-Ausdruck mit `old` in Nachbedingung

JML-basierte Lösung

Zur Realisierung einer JML-basierten Lösung kann das JML-Konstrukt `old` auch in der Nachbedingung einer JML-Assertion verwendet werden, um Objekte bzw. Werte des Systemzustandes vor der Ausführung der Operation zu referenzieren. Damit können in einer Nachbedingung sowohl Werte des Systemzustandes vor als auch nach der Ausführung einer Operation ausgewertet werden. Die Notation in diesem Fall ist `\old(E)`, wobei es sich bei `E` um einen Java-Ausdruck handelt. Die Auswertung von `\old(E)` in einer Nachbedingung liefert das gleiche Ergebnis wie die Auswertung des Ausdrucks `E` in der Vorbedingung einer Operation. Zeile 17 in Listing 4.10 zeigt ein Beispiel. Der Ausdruck `\old(this.getCarts().values()).contains(c)` besagt, dass der Rückgabewert der Methode `this.getCarts().values()` vor dem Start der Operation als Referenzwert in der JML-Nachbedingung verwendet wird. Der Rest des Ausdrucks bezieht sich auf Inhalte, die nach der Ausführung der Operation gültig sind. Die dargestellte Auswertung des obigen Ausdrucks ist notwendig, da in dem Beispiel in der Nachbedingung eine Assoziation der Vorbedingung des visuellen Kontraktes aus Abbildung 3.3 überprüft werden soll. Im Folgenden erklären wir kurz die Umsetzung des visuellen Kontraktes aus Abbildung 3.3 unter Verwendung des `old`-Konstruktes anhand Listing 4.10.

Die Vorbedingung des Kontraktes aus Listing 4.10 ist identisch mit dem Kontrakt aus Listing 4.5. Das `requires`-Statement wird lediglich verwendet, um zu überprüfen, ob die durch den visuellen Kontrakt gegebene Vorbedingung erfüllt ist. Die Vorbedingung dient nicht dazu, irgendwelche Bindings dauerhaft zu speichern.

Die Nachbedingung greift dann mit dem `old`-Kommando auf den Systemzustand vor der Ausführung der Operation zurück. Die mit `old` gefundenen Objekte werden gebunden. Damit kann der Zustand dieser Objekte sowohl vor als auch nach der Ausführung der Operation überprüft werden. Dies führt dazu, dass in der Nachbedingung die meisten Attribute und Assoziationen zweimal überprüft werden: einmal für den Zustand vor der Ausführung der Operation, und einmal nach der Ausführung der Operation, um sicherzustellen, dass sich die Inhalte gemäß den Vorgaben im visuellen Kontrakt geändert haben. Bei der reinen JML-basierten Lösung wird also nahezu der vollständige visuelle Kontrakt in die JML-Nachbedingung kodiert.

Auch wenn wir nichtdeterministische Einbettungen von Vor- oder Nachbedingungen in einen Systemzustand handhaben können, sind wir der Meinung, dass der Softwareentwickler derartige Probleme mit einem detailliertem Klas-

```
 1 /*@ public normal_behavior
 2   @ requires (
 3   @   (\exists Cart c;
 4   @     this.getCarts().values().contains(c);
 5   @     c.getCartID().equals(cid) &&
 6   @     !c.getCartItems().values().isEmpty()
 7   @ ));
 8   @
 9   @ ensures (
10   @   (\exists Cart c;
11   @     this.getCarts().values().contains(c);
12   @     c.getCartID().equals(cid) &&
13   @     c.getCartItems().values().isEmpty()
14   @ ));
15   @*/
16 public boolean cartClear(String cid);
```

Listing 4.11: Löschen von Objekten

sendiagramm (Einführung von Schlüsseln) oder mit der Verwendung weiterer Rollen in einer Vor- oder Nachbedingung, um ein Match weiter einzuschränken, vermeiden sollte. Ansonsten verhält sich die Anwendung nichtdeterministisch gegenüber ihrer Spezifikation, was zu unerwarteten und schwer zu findenden Fehlern führen kann.

Weiterhin kann auch bei der hier beschriebenen JML-basierten Lösung die Berechnung einer Einbettung mit verschiedenen bekannten Verfahren verbessert werden. Durch eine geeignete Schachtelung der JML-Quantoren kann die Reihenfolge in welcher die Elemente einer Kollaboration abgearbeitet werden beeinflusst werden. Dadurch kann z.B. versucht werden ein Missmatch so früh wie möglich zu finden [Smi97] oder es kann die Worst-Case Execution Time zur Berechnung einer Einbettung optimiert werden [BGST05].

Löschen von Objekten

Ein weiteres bei der Generierung von Assertions zur berücksichtigendes Problem ist das Löschen von Elementen. Der visuelle Kontrakt aus Abbildung 3.5 beschreibt, wie ein Warenkorb gelöscht werden kann: alle Objekte vom Typ `CartItem`, die von einem über das Schlüsselattribut `cartId` identifizierten `Cart` aus erreichbar sind, müssen gelöscht werden. Wie wir im Folgenden sehen werden, erhöht beim Löschen von Elementen in visuellen Kontrakten ein

Allquantor die Komplexität des zu erzeugenden JML-Ausdrucks nicht wesentlich. Listing 4.11 zeigt die Übersetzung nach JML. Bei der Übersetzung gehen wir wieder davon aus, dass die Entity-Objekte in einem System über ihre Schlüsselattribute eindeutig identifiziert werden können. In der Vorbedingung wird überprüft, ob das aktive Objekt `this` ein Objekt vom Typ `Cart` mit dem Inhalt `cid` für das Attribut `cartId` kennt (Zeilen 3-4). Danach wird überprüft, ob das gefundene Objekt vom Typ `Cart` ein Objekt vom Typ `CartItem` referenziert (Zeile 5), d.h. es wird überprüft ob der Warenkorb leer ist oder nicht. Wenn der Warenkorb nicht leer ist, ist die Vorbedingung erfüllt. Die Vorbedingung stellt also lediglich sicher, dass ein Multiobjekt existiert, d.h. es wird nur sichergestellt, dass ein Matching existiert. Die Nachbedingung ist bis auf Zeile 13 identisch. Im Gegensatz zur Vorbedingung ist die Nachbedingung erfüllt, wenn das gefundene Objekt vom Typ `Cart` kein Objekt vom Typ `CartItem` referenziert.

In dem JML-Kontrakt wird also nicht geprüft, ob ein Objekt gelöscht wird. Es wird lediglich geprüft, ob die im visuellen Kontrakt verwendeten Referenzen auf die zu löschenden Objekte in der Nachbedingung nicht mehr existieren. Soll in einem JML-Kontrakt überprüft werden, ob ein Objekt während der Ausführung einer Operation gelöscht wurde, so müssten zum einen in den JML-Kontrakten zur Annotation von Operationen wieder zu großen Teilen Modellmethoden verwendet werden. Dies würde zu den bereits weiter oben genannten Nachteilen führen. Zum anderen benötigt die Überprüfung, ob ein Objekt gelöscht wurde, zusätzliche Linkstrukturen neben denen, die im Design-Klassendiagramm vorgegeben werden. Die Verwaltung derartiger Strukturen kann aufgrund der erlaubten manuellen Programmierung zu Inkonsistenzen führen (z.B. würde der Java Garbage Collector nicht mehr funktionieren). Aus diesen Gründen haben wir uns beim Löschen von Objekten für die einfachere JML-basierte Variante entschieden und überprüfen lediglich, ob die im visuellen Kontrakt verwendeten Referenzen auf das Objekt gelöscht werden.

4.4 Einsatz der visuellen Kontrakte in der Implementierung

Zur Realisierung unseres Model-Driven Monitoring haben wir in diesem Kapitel gezeigt, wie Klassendiagramme und visuelle Kontrakte in Java-Klassen und JML-Assertions übersetzt werden können. Auf der Ebene der Implementierung können existierende JML-Werkzeuge verwendet werden. Ein wichti-

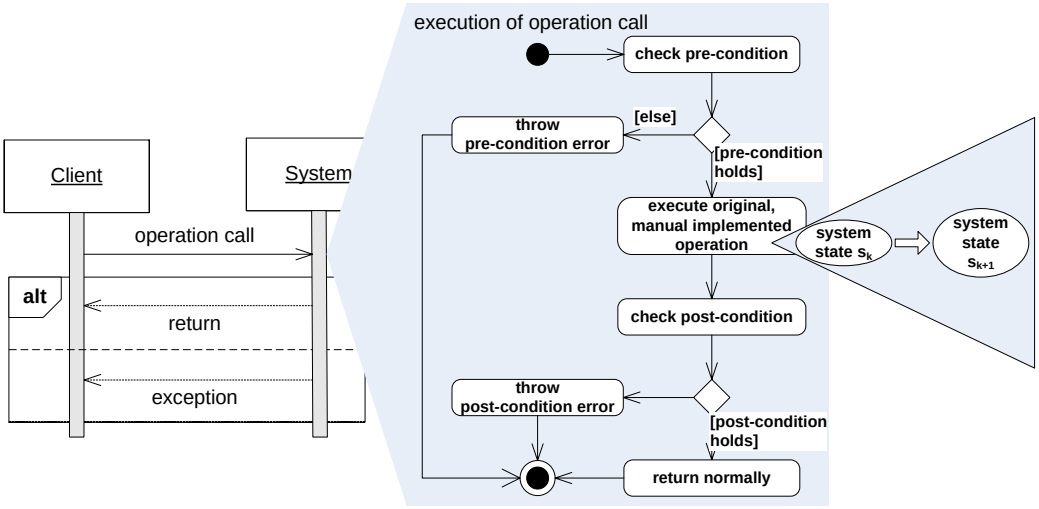


Abbildung 4.4: Laufzeitverhalten einer mit JML annotierten Operation

ges Werkzeug ist der JML-Compiler, der einen Standard-Java-Compiler erweitert. Der JML-Compiler erzeugt aus den JML-Annotationen Methoden zur Überprüfung der Vor- und Nachbedingungen. Die manuelle Implementierung einer Operation wird von dem JML-Compiler durch eine automatisch generierte Wrapper-Methode ersetzt. Die originale, manuelle Implementierung bekommt einen neuen Namen und die Sichtbarkeit der Operation wird auf `private` gesetzt. Die Wrapper-Methode delegiert die Methodenaufrufe eines Clients an die originale, manuelle Implementierung der Operation und ruft die entsprechenden Methoden zur Überprüfung der Assertions auf.

Dies führt für eine Operation, die mit einem JML-Kontrakt annotiert ist, zu dem in Abbildung 4.4 gezeigten Laufzeitverhalten. Ruft eine Client eine Operation auf, so wird als erstes eine vom JML-Compiler generierte Methode zur Überprüfung der Vorbedingung aufgerufen. Ist die Vorbedingung verletzt, so wird eine Exception geworfen. Ist die Vorbedingung nicht verletzt, so wird die originale, manuelle Implementierung der Operation aufgerufen, wodurch der Zustand des Systems geändert wird. Nach der Ausführung der originalen, manuellen Implementierung der Operation wird eine Methode zur Überprüfung der Nachbedingung aufgerufen. Ist die Nachbedingung nicht erfüllt, so wird wiederum eine Exception geworfen.

Wird während der Überprüfung der Vorbedingung eine Exception geworfen, so hat der Client beim Aufruf der Operation die durch den Kontrakt vorgegebenen Anforderungen mißachtet. Dies ist ein Fehler auf der Seite des Clients. Eine Exception, die bei der Überprüfung der Nachbedingung geworfen wird,

```
1 try {
2   shop.cartAdd(cartId, ASIN, quantity);
3 } catch (JMLPreconditionError e) {
4         System.out.println("Violation of precondition "
5                                 + e.getMessage());
6 } catch (JMLPostconditionError e) {
7   System.out.println("Violation of postcondition "
8                                 + e.getMessage());
9 } catch (Error e) {
10  System.out.println("Unidentified error!"
11                                + e.getMessage();
12 }
```

Listing 4.12: Verwaltung von Exceptions

weist daraufhin, dass die manuelle Implementierung einer Operation nicht in der Lage war, den Kontrakt einzuhalten. In diesem Fall enthält die manuelle Implementierung der Operation einen Fehler, während der Client nicht für den Fehler verantwortlich ist.

Wird der Kontrakt zur Laufzeit nicht verletzt, so bemerkt ein Client die Überprüfung der Vor- und Nachbedingungen nicht. Der Systemzustand wird nur von der originalen, manuellen Implementierung der Operation geändert. Die generierten Java-Fragmente und JML-Ausdrücke haben keinen Einfluss auf den Zustand des Systems. Das heisst, bis auf den Zeit- und Speicherplatzbedarf der zusätzlich generierten Assertions, verhält sich eine korrekte manuelle Implementierung so, als ob keine Assertions vorhanden wären.

Mit den generierten Assertions können wir also die Korrektheit einer manuellen Implementierung zur Laufzeit überwachen. Sollen die Vorteile unseres Model-Driven-Monitoring-Ansatzes weitergehend ausgenutzt werden, so muss ein Softwaresystem in der Lage sein, auf Exceptions, die bei der Überprüfung einer Vor- oder Nachbedingung geworfen werden, geeignet zu reagieren. Hierzu enthalten die JML Tools auch eine JML-Library, die von einem Programmierer genutzt werden kann. Bestandteil dieser Library sind unter anderem die Klassen `JMLPreconditionError` und `JMLPostconditionError` zum Abfangen der JML-Exceptions. Listing 4.12 zeigt wie die Exceptions in einer Java-Implementierung abgefangen werden können. Die Operation in Zeile 2 zu Beginn des `try-catch`-Blockes ist eine Operation, die während der Design-Phase mit einem visuellen Kontrakt annotiert wurde. Ein Programmierer kann die Java-Mechanismen zum Abfangen von Exceptions nutzen, um auf Fehler bei der Überprüfung der Vor- und Nachbedingungen geeignet

```
1 Violation of precondition by method OnlineShop.cartAdd
2 regarding specifications at
3 File "de\upb\dbis\amazonmini\OnlineShop.refines-java",
4 line 34, character 18 when
5   'cid' is Cart_1
6   'ASIN' is Product_3
7   'quantity' is 3
8   'this' is de.upb.dbis.amazonmini.OnlineShop@1d520c4
```

Listing 4.13: Beispiel für eine geworfene Exception

zu reagieren. In Listing 4.12 werden lediglich detaillierte Fehlermeldungen ausgegeben. Listing 4.13 zeigt die generierte Ausgabe bei einer verletzten Vorbedingung.

Ergänzend kann ein Programmierer zu Testzwecken die Software JMLUnit [BCC+03, CL02b] verwenden. JMLUnit kombiniert JML mit dem populären Testwerkzeug JUnit [BG, BG98]. Die grundlegende Idee von JMLUnit ist, dass die Spezifikation einer Operation mit Vor- und Nachbedingungen als ein Testorakel angesehen werden kann [AH00, PP98] und dass mit einem Assertion Checker zur Laufzeit überprüft werden kann, ob ein Testfall erfolgreich ist. Das JMLUnit Tool kann den Programmierer bei der Verwaltung von Testfällen signifikant entlasten. Es erstellt automatisiert eine Menge von Klassen für die Verwaltung von Testfällen für Java-Klassen und Schnittstellen. Das generelle Problem — die Erzeugung geeigneter Testfälle — bleibt jedoch noch erhalten. Jedoch werden diese Probleme bereits in verschiedenen Arbeiten intensiv diskutiert. Zum Beispiel finden sich in [Bin00, BL01, BL02] Ansätze, wie aus Statecharts oder Sequenzdiagrammen Testfälle abgeleitet werden können. Erste Ansätze zum Einsatz visueller Kontrakte für das Testen, insbesondere für das Testen von Web Services, haben wir bereits in [HL04, HL03b] vorgestellt.

4.5 Fazit

In diesem Kapitel haben wir gezeigt, wie wir aus den Modellen, die in der Design-Phase eines Softwareentwicklungsprozesses erstellt werden, Code generieren. Die statischen Modelle des Systems (Klassendiagramme) werden in Klassentemplates in der Zielsprache Java übersetzt. Die visuellen Kontrakte werden in die Java Modeling Language (JML) übersetzt. JML ist eine

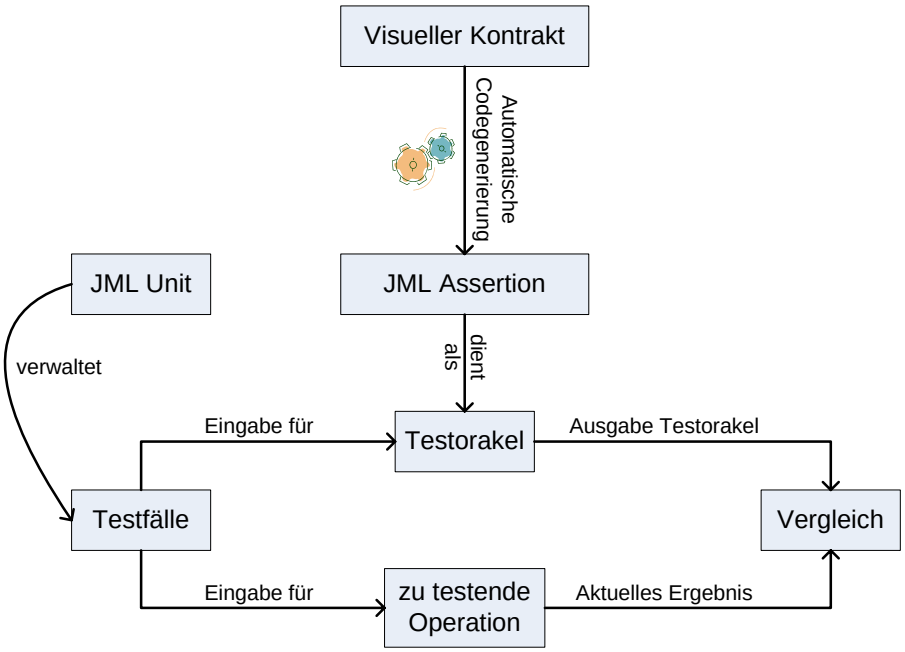


Abbildung 4.5: Einsatz von Assertions als Testorakel

Design-by-Contract-Erweiterung für Java. Dann verwenden wir den JML-Compiler zur Übersetzung der JML-Assertions in ausführbaren Bytecode. In den compilierten Bytecode sind Bestandteile integriert, welche die Vor- und Nachbedingungen eines visuellen Kontraktes zur Laufzeit überprüfen. Damit kann die Konsistenz der manuell erstellten Implementierung mit den visuellen Kontrakten zur Laufzeit überprüft werden, da die automatisch generierten Assertions überwachen, ob sich das System gemäß seiner Spezifikation verhält.

Mit der Übersetzung unserer Design-Modelle nach Java und JML haben wir gezeigt, dass unser Ansatz des Model-Driven Monitoring anwendbar ist. Eine Übersetzung unserer visuellen Kontrakte in andere Design-by-Contract-Sprachen, wie zum Beispiel Microsofts Spec# [BLS04] — einer Design-by-Contract-Erweiterung für Microsofts C# [HWG03] — ist denkbar.

Kapitel 5

Formale Spezifikation der Transformation nach JML

In diesem Kapitel beschreiben wir eine formale, operationalisierbare Spezifikation der Übersetzung der visuellen Kontrakte nach JML. Hierzu wurde bereits im vorherigen Kapitel anhand von Beispielen erläutert, wie eine JML-Assertion auf der Ebene der Implementierung einen visuellen Kontrakt wiedergeben kann. Die Entwicklung einer formalen, operationalisierbaren Modelltransformation ist der Schlüssel zur Erstellung einer Implementierung für die automatisierte Übersetzung von visuellen Kontrakten nach JML.

Bevor wir die Transformation formal beschreiben, formulieren wir zunächst die Anforderungen an eine Transformation und diskutieren verschiedene Ansätze zur Realisierung von Modelltransformationen.

5.1 Formale Spezifikationen

In dieser Arbeit übersetzen wir unter anderem Modelle bestehend aus UML-Klassendiagrammen und visuellen Kontrakten in Java-Klassengerüste, die mit JML-Spezifikationen annotiert sind. Mit dieser Übersetzung soll sichergestellt werden, dass die später manuell hinzugefügte Implementierung konsistent zur Spezifikation ist. Grundsätzlich gibt es zahlreiche Mittel zur Beschreibung dieser Abbildung. In Kapitel 4 haben wir anhand von Beispielen den Zusammenhang zwischen visuellen Kontrakten und JML-Assertions zur Repräsentation von visuellen Kontrakten erläutert.

Aus dieser beispielhaften Beschreibung der Abbildung der Modelle nach Java

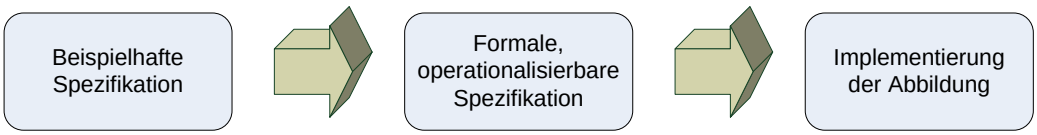


Abbildung 5.1: Ablauf zur Bestimmung der Implementierung der Transformation

und JML kann aus verschiedenen Gründen nicht unmittelbar eine Implementierung der Abbildung abgeleitet werden. Erstens abstrahiert die gegegebene Beschreibung der Abbildung von den Details zur Verwaltung der Modelle. Das den Modellen zugrunde liegende Metamodell wird nicht berücksichtigt. Zweitens gibt die beispielhafte Beschreibung lediglich den Start und den Zielpunkt einer Modelltransformation für ein konkretes Modell vor und abstrahiert dabei von den Details, die beschreiben, wie die Transformation erfolgt. Es wird nicht dargelegt, wie das Ergebnis erreicht wird, bzw. wie die Transformation ausgeführt wird. Es wird von der Reihenfolge der Generierung der einzelnen Bestandteile der Zielsprache abstrahiert.

In diesem Kapitel geben wir eine formale Beschreibung der Transformation der Klassendiagramme und visuellen Kontrakte nach Java und JML an. Diese formale Spezifikation stellt explizit dar, wie die Modelltransformation von einem Start zu einem Zielpunkt erfolgt. Aufgrund der höheren Genauigkeit der formalen Spezifikation kann diese als Vorlage für die Implementierung einer automatisierten Übersetzung von visuellen Kontrakten nach JML verwendet werden. Abbildung 5.1 gibt einen Überblick über das Vorgehen zur Bestimmung einer geeigneten Implementierung für eine Transformation. Die Übersetzung zwischen den einzelnen Ebenen erfolgt manuell. Auch die Übersetzung der formalen Spezifikation in eine Implementierung der Abbildung erfolgt in diese Arbeit manuell. Das Ziel der hier gegebenen formalen Spezifikation ist es, die Transformation genau, intuitiv und unabhängig von einem bestimmten Framework zu beschreiben, so dass aus der Spezifikation möglichst einfach eine Implementierung (für eventuell unterschiedliche Plattformen) abgeleitet werden kann. Wir nutzen die formale Spezifikation also, um die Abbildung unserer visuellen Kontrakte nach JML möglichst genau zu beschreiben. Im Folgenden beschreiben wir die Anforderungen an eine geeignete formale Spezifikation genauer.

Ein Ansatz für eine formale Spezifikation einer Modelltransformation muss für unsere Zwecke verschiedene Anforderungen erfüllen. Die Spezifikation muss in einer Form gegeben sein, dass eine Implementierung aus ihr abgeleitet werden kann. Die Möglichkeit einer automatisierten Übersetzung un-

serer Modelle ist entscheidend für die Realisierung des Model-Driven Monitoring auf Basis des in Kapitel 2.5 skizzierten Softwareentwicklungsprozesses. Ohne eine automatisierte Übersetzung nach JML ist die Überwachung der Korrektheit einer manuell erstellten Implementierung gegenüber den visuellen Kontrakten nicht möglich. Weiterhin muss die Spezifikation *genau* sein. Die Spezifikation darf nicht von den Details zur Verwaltung der Modelle (den Metamodellen) abstrahieren. Mit der Spezifikation muss festgelegt werden können in welcher *Reihenfolge* das Ziel generiert wird, um die Pfadausdrücke bzw. Navigationen auf den Objekten einer Anwendung mit den generierten JML-Ausdrücken korrekt zu ermöglichen. Ein Ansatz für eine formale Spezifikation einer Modelltransformation muss weiterhin *flexibel* im Hinblick auf eine leichte Anpassbarkeit der Transformationen sein. Der generierte JML-Code ist abhängig von dem aus den Klassendiagrammen generierten Code. Aus diesem Grund muss die Spezifikation zur Beschreibung der JML-Codegenerierung flexibel an die Vorgaben der Codegenerierung aus den Klassendiagrammen anpassbar sein. Da die Entwicklung der Transformationen einen großen Aufwand bedeutet, sollte ein entsprechender Ansatz auch die *Wiederverwendung* von Modelltransformationen unterstützen. In unserem Ansatz sollte zum Beispiel die Spezifikation der Transformation an verschiedene Zielsprachen anpassbar sein. Dabei kann es sich um andere Design-by-Contract-Sprachen für Java, wie z.B. JContract [Par02] oder Jass [BFMW01], sowie um Design-by-Contract-Erweiterungen für andere Programmiersprachen, wie z.B. Spec# [BLS04], handeln.

5.2 Existierende Ansätze

Im Folgenden skizzieren wir verschiedene Ansätze zur formalen Spezifikation von Modelltransformationen. Wir unterscheiden zwischen Ansätzen, die aus Modellen Code generieren und Ansätzen, die aus Modellen wiederum Modelle erzeugen. Einem Modell liegt ein Metamodell zugrunde, während es sich bei Code um einfachen Text handelt, dem eine bestimmte Grammatik zugrunde liegt. Generell werden mit Ansätzen, die Code aus Modellen erzeugen, lediglich Textartefakte erzeugt, d.h. es wird nicht überprüft, ob syntaktisch korrekter Code generiert wird (vgl. auch [CH03]).

Eine allgemeinere Sichtweise ist, dass die Ansätze für die Transformation von Modellen in Modellen auch zur Generierung von Code verwendet werden können. Bei dieser allgemeineren Sichtweise wird jedoch ein Metamodell für die Zielprogrammiersprache benötigt (wie z.B. in [LG03]). Metamodelle

für Programmiersprachen stehen generell jedoch nicht zur Verfügung. Ein Beispiel ist das (unvollständige) Java-Metamodell der Object Management Group, das im Rahmen der EDOC (Enterprise Distributed Object Computing) Spezifikation veröffentlicht [Obj04] wurde. Für die uns bekannten Design-by-Contract-Erweiterungen für Java oder C# stehen keine Metamodelle zur Verfügung. Daher betrachten wir im Folgenden Code als einfachen Text.

5.2.1 Transformation von Modellen in Code

Bei den Ansätzen zur Generierung von Code aus Modellen handelt es sich in der Regel um sehr technische Ansätze, die durch ein entsprechendes Framework unterstützt werden. Diese technischen Frameworks können in zwei Kategorien unterteilt werden. Frameworks, die eine Traversierung des Modells unterstützen und Frameworks, die eine templatebasierte Codegenerierung unterstützen.

Traversierung der Quellmodelle

Eine Möglichkeit aus existierenden Modellen Code zu generieren ist eine vollständige Implementierung eines Transformationsalgorithmus in einer Programmiersprache wie z.B. Java. In der Regel durchsucht (traversiert) ein Transformationsalgorithmus die interne Repräsentation eines Modells in einer bestimmten Reihenfolge, die es erlaubt den benötigten Code zu erzeugen. Oft wird der Code dabei sequentiell in einen Textstream geschrieben.

Die manuelle Implementierung eines Transformationsalgorithmus kann mit entsprechenden Frameworks unterstützt werden. Ein Beispiel ist das objektorientierte Open-Source Framework Jamda [Boo03]. Jamda oder ähnliche Frameworks bieten eine Menge von Klassen zur Repräsentation von UML-Modellen, sowie eine API zur Manipulation von Modellen. Zur Verwendung dieser Frameworks implementiert ein Entwickler eine Reihe von Callback-Operationen, deren Signatur durch eine API vorgegeben ist. Diese Callback-Operationen verarbeiten an sie übergebene Modellinstanzen und müssen in dem Framework registiert werden. Das Framework ruft dann diese Operationen bei dem Durchsuchen des Modells zur Codegenerierung in einer zuvor bestimmten Reihenfolge auf. Das Vorgehen zur Erzeugung von Code mit Callback-Operationen ist vergleichbar mit den bekannten SAX-Parsern [Meg], die auch Callback-Operationen zur Bearbeitung von XML-

Dokumenten definieren. Häufig bieten diese Frameworks allerdings keine Unterstützung für beliebige Metamodelle, was deren Verwendung einschränkt.

Obwohl die vollständige Implementierung eines Transformationsalgorithmus einem Entwickler weitgehende praktische Möglichkeiten bietet, um die Codegenerierung zu steuern, sind diese Ansätze in der Regel nicht gut anwendbar. Um die Abbildung des Modells in den Code zu verstehen, muss ein Entwickler sich auf untersuchte oder gegebene Beispiele verlassen oder er muss die Sourcen für den Transformationsalgorithmus untersuchen. Häufig besteht jedoch kein sichtbarer Zusammenhang zwischen den Beispielen für die Übersetzung und den Transformationsalgorithmen. Eine Anpassung der Codegenerierung ist häufig nur sehr schwer oder gar nicht möglich.

Templatebasierte Ansätze

Templatebasierte Ansätze sind die am häufigsten eingesetzte Form zur Umsetzung einer Codegenerierung aus Modellen. Dies zeigt sich auch an der großen Zahl von Frameworks zur Unterstützung der templatebasierten Codegenerierung, wie zum Beispiel JET [Pop04a, Pop04b], b+m Generator Framework [b+m], FUUT-je [vEB00], Codagen Architect [Cod], AndroMDA [And], ArcStyler [Int], OptimalJ [Com, MF05] oder XDE [IBM].

Ein Framework zur Unterstützung templatebasierter Ansätze durchsucht ein Modell und ruft abhängig von den gefundenen Strukturen ein Template auf. Ein Template besteht normalerweise aus dem zu generierenden Zielcode für eine Menge von Modellelementen. Der Zielcode ist angereichert mit Metainformation bzw. Metacode, um auf Informationen aus dem Quellmodell zugreifen zu können, die in den Zielcode eingefügt werden und um die Codegenerierung mit entsprechenden Kontrollstrukturen zu steuern. In der Regel erfolgt die Codegenerierung bei den templatebasierten Ansätzen iterativ. Kontrollstrukturen werden in templatebasierten Ansätzen realisiert indem ein Template aus einem anderem Template heraus aufgerufen werden kann. In der Regel basieren Templates auf Textpattern, die ungetypt sind [CH03]. Somit können mit templatebasierten Ansätzen Textpattern für beliebige Zielsprachen, d.h. z.B. auch JML-Ausdrücke generiert, werden.

Im Vergleich zur manuellen Implementierung mit einem Transformationsalgorithmus wird bei einem templatebasiertem Ansatz der Zusammenhang zwischen den Templates und dem aus den Templates generiertem Code sehr gut deutlich. Der Inhalt eines Templates entspricht nahezu dem generiertem Code. Dadurch erlauben templatebasierte Ansätze eine sehr einfache und ite-

rative Entwicklung, die aus einer deskriptiven Beschreibung einer Transformation abgleitet werden kann. Eine einfache Anpassung des zu generierenden Code wird aufgrund der Strukturen der Templates auch möglich. Allerdings ist in diesen Ansätzen der Zusammenhang zwischen den Quellmodellen und den Templates nicht gut erkennbar, da in der Regel die Quellmodelle nur indirekt über Methodenaufrufe, die in den Templates eingebettet sind, referenziert werden. Damit wird eine Anpassung der Templates an Änderungen im Metamodell erschwert.

5.2.2 Transformation von Modellen in Modelle

Verfahren zur Transformation von Quellmodellen in Zielmodelle lassen sich auch in mehrere Kategorien einteilen. Bei den meisten Ansätzen können Quell- und Zielmodell Instanzen desselben oder unterschiedlicher Metamodelle sein [CH03].

Traversierung der Quellmodelle

Ein Möglichkeit zur Transformation eines Quell- in ein Zielmodell, ähnlich wie bei den Ansätzen zur Generierung von Code, ist die vollständige Implementierung eines Transformationsalgorithmus in einer Programmiersprache wie z.B. Java. Bei der Transformation in ein Zielmodell arbeitet der Algorithmus üblicherweise auf einer internen Repräsentation der Modelle. Quell- und Zielmodell werden in einer bestimmten Reihenfolge durchsucht bzw. erzeugt.

Die manuelle Implementierung kann unterstützt werden durch Frameworks (wie z.B. Jamda [Boo03]), die Klassen zur internen Repräsentation der Quell- und Zielmodelle bereitstellen sowie eine API zu deren Manipulation. In der Regel bieten diese Frameworks jedoch nur sehr geringe weitergehende Möglichkeiten zur Organisation der Transformation. Zum Beispiel müssen die Entwickler die Transformationsregeln und deren Ausführungsreihenfolgen immer noch fast vollständig selbst implementieren. Dazu müssen auch in diesem Ansatz häufig Callback-Operationen, die von dem Framework vorgegeben werden, implementiert werden (vgl. auch Abschnitt 5.2.1). Da die Frameworks in der Regel nur Strukturen zur Verwaltung einer internen Repräsentation der Modelle anbieten, können diese nicht für beliebige Metamodelle angewendet werden.

Obwohl ein Entwickler bei diesem Ansatz weitgehende Möglichkeiten hat, um die Generierung eines Zielmodells zu steuern, ist dieser Ansatz in der

Regel — ähnlich wie bei der Codegenerierung durch eine Traversierung der Modelle — nicht gut anwendbar. Der Zusammenhang zwischen dem Algorithmus sowie einer Transformation ist nicht direkt ersichtlich, wodurch eine Anpassung des Algorithmus erschwert wird. Ist eine Anpassung notwendig, so muss der Entwickler auch in diesem Fall versuchen, sich anhand von Beispielen die Funktionsweise der Implementierung zu erarbeiten. Häufig ist jedoch der Zusammenhang zwischen den Beispielen und der Implementierung nicht gut ersichtlich.

Relationale Ansätze

Relationale Ansätze beruhen auf mathematischen Relationen und ermöglichen eine deklarative Definition von Beziehungen zwischen Modellen. Eine Relation besteht aus einer Menge von Paaren. Dabei wird durch die Relation bestimmt, wie die Paare konstruiert werden dürfen. Ein Paar besteht aus zwei Modellelementen (aus unterschiedlichen Modellen). Die Bedeutung eines Paares kann mit einem Constraint detailliert werden.

Ein Beispiel für einen modellbasierten relationalen Ansatz ist [AK02]. In diesem Ansatz werden mathematische Relationen in ein objektorientiertes Modell kodiert. In [HK03, Hau03] wird dieser Ansatz um eine konkrete Syntax sowie ein Metamodell zur Spezifikation von Paaren und Relationen ergänzt. Relationale Ansätze sind jedoch häufig nicht operationalisierbar. So geben diese Ansätze z.B. häufig keine Ausführungsreihenfolge für die Generierung eines Zielmodells vor, weshalb sie nur schwer als Vorlage für die Implementierung einer Transformation verwendet werden können. Weiterhin lassen sich mit relationalen Ansätzen nur schwer textuelle Zielmodelle integrieren, wenn kein geeignetes Metamodell zur Verfügung steht. Da jedoch nicht immer ein Metamodell für eine Zielsprache vorausgesetzt werden kann und wir nach einem Ansatz suchen, der auch eine einfache Anpassung an verschiedene Zielsprachen (wie z.B. C#, Spec#) ermöglicht, sind relationale Ansätze eher weniger geeignet.

Graphtransformationen

In den letzen Jahren wurden verschiedene Ansätze zur Spezifikation von operationalen Modelltransformationen entwickelt, die auf der Theorie von Graphtransformationen [EPS73] basieren. Insbesondere arbeiten diese Ansätze auf getypten, attributierten Graphen [AEH+96, AEH+99] (vgl. auch

Kapitel 3.4). Mit diesen Graphen können UML-ähnliche Modelle repräsentiert werden. Die generelle Idee dieser Ansätze ist, die Modelltransformationen mit einer Menge von Graphtransformationsregeln, die über dem UML-Metamodell getypt sind, zu spezifizieren. Beispiele für Ansätze, die Graphtransformationen zur Transformation von Modellen einsetzen, sind VIATRA [CHM+02], ATOM [LV02], GreAT [Agr03] oder UMLX [Wil03]. Andere Ansätze verwenden Graphtransformationen für die Spezifikation einer operationalen Semantik für UML-Modelle [EHHS00, HHS04, HHS03, HHS01, HKS01]

Wie bereits zuvor beschrieben, bestehen die linke und die rechte Seite einer Graphtransformationsregel aus Graphpattern. Das Pattern auf der linken Seite einer Graphtransformationsregel wird im Modell, das transformiert werden soll, gesucht. Danach wird das im Modell gefundene Pattern durch das Pattern auf der rechten Seite der Graphtransformationsregel ersetzt. In vielen Ansätzen werden die Graphtransformationsregeln in zusätzliche Kontrollstrukturen eingebettet, um die Ausführungsreihenfolge der Graphtransformationen bestimmen zu können. Mit diesen Kontrollstrukturen können zum Beispiel deterministische, nicht-deterministische Selektionen oder Iterationen festgelegt werden (vgl. auch Abschnitt 5.3.2).

Graphtransformationen erlauben eine operationale Beschreibung der Modelltransformationen. Die Verwendung von Graphpattern und das den Graphtransformationsregeln zugrundeliegende Matching von Graphpattern unterscheidet diese Ansätze von relationalen Modelltransformationen. Weiterhin sind Modelltransformationen auf Basis von Graphtransformationen aufgrund der Charakteristiken von Graphtransformationssystemen operationalisierbar und flexibel. Ein weiterer Vorteil von Graphtransformationen ist, dass der Zusammenhang zwischen einer in einer Graphtransformationsregel beschriebenen Transformation und den Quell- bzw. Zielmodellen gut erkennbar ist. Ein Nachteil ist jedoch, dass mit Graphtransformationen nur schwer textuelle Zielmodelle, wie z.B. Java-Code, erzeugt werden können, wenn kein geeignetes Metamodell zur Verfügung steht. Da jedoch nicht immer ein Metamodell für eine Zielsprache vorausgesetzt werden kann und wir nach einem Ansatz suchen, der auch eine einfache Anpassung an andere Zielsprachen (wie z.B. C#, Spec#) ermöglicht, sind Graphtransformationen nur bedingt geeignet.

Weitere Ansätze

Neben den oben bereits skizzierten Ansätzen, gibt es noch weitere, die sich nicht so einfach einer der obigen Kategorien zuordnen lassen. Hier wollen wir noch auf zwei weitere eher technisch orientierte Ansätze eingehen.

Ein Beispiel für die Verwendung von XSLT (XSL Transformations) [Cla99] zur Spezifikation von Modelltransformationen findet sich in [DHO01, Mar04]. In diesem Fall werden die Modelltransformationen auf Basis der Extensible Markup Language (XML) [BPSM+04] beschrieben. Diese können jedoch nicht in aktuellen Case Tools verwendet werden. Deshalb gibt es wiederum Ansätze, die auf XMI [OMG05] basieren und insbesondere die *XMI-difference* Konzepte ausnutzen, um Modelltransformationen zu beschreiben [Wag02, KH02].

Bei beiden Vorgehensweisen werden verschiedene Regeln definiert, die mehrere Male angewendet werden können, um eine Modelltransformation auszuführen. Jedoch müssen in diesen Ansätzen implementationsspezifische Aspekte berücksichtigt werden. Weiterhin erlauben sie keine flexible, visuelle Spezifikation der Modelltransformationen.

5.2.3 Hybride Ansätze: Kombination von Graphtransformationen mit templatebasierten Ansätzen

Die obige Diskussion hat gezeigt, dass sich die Ansätze zur Generierung von Code bzw. Modellen aus existierenden Modellen zum Teil stark unterscheiden. Zur Generierung von Code sind templatebasierte Ansätze sehr gut geeignet, da ein Template sehr gut darstellt, wie der zu generierende Code aussieht, wodurch deren Anwendung vereinfacht wird. Jedoch wird der Zusammenhang eines Templates zu einem Modellelement nur sehr unzureichend deutlich, da innerhalb eines Templates lediglich logische Ausdrücke zum Zugriff auf das Quellmodell verwendet werden. Eine visuelle Repräsentation fehlt. Graphtransformationen erlauben hingegen eine einfache Repräsentation der Modelle auf Basis der Metamodelle. Die Metamodelle sind zur Verwendung von Graphtransformationen allerdings auch notwendig, wodurch die Generierung von Code mit Graphtransformationen wiederum erschwert wird.

In [EHSW99a, EHSW99b] sind *Metaregeln* zur Generierung von ausführbarem Java-Code aus UML-Modellen eingeführt worden. Diese Metaregeln vereinen templatebasierte Ansätze mit Graphtransformationen. Die in [Küs04] eingeführten *Compound Rules* zur Beschreibung von Modelltransformationen erweitern und formalisieren den Ansatz der Metaregeln. Der Ansatz erlaubt die Abbildung von UML-Modellen in einen semantischen Bereich [HKT02b]. Er erweitert und spezialisiert verschiedene existierende Ansätze, wie z.B. Pair Grammars von Pratt [Pra71] und Triple Graph Grammatiken von Schürr [Sch94]. Bei den Compound Rules handelt es sich generell um Graphtrans-

formationen, aber sie sind auf die Abbildung von UML-Modellen in eine semantische Domäne ausgerichtet. Eine semantische Domäne wird bei diesen Regeln in einem textbasiertem Format beschrieben. Jede der Compound Rules besteht aus zwei Transformationsregeln, einer Quell- und Zieltransformationsregel. Compound Rules können in diesem Ansatz zu *Transformation Units* zusammengefasst werden, welche zusätzlich spezifizieren, in welcher Reihenfolge die Regeln angewendet werden sollen.

Da der Ansatz von [Küs04], ähnlich wie templatebasierte Ansätze zur Generierung von Code, keine syntaktischen Anforderungen an die in einem textbasierten Format dargestellte semantische Domäne stellt, kann dieser Ansatz auch zur Generierung von Java- und JML-Code verwendet werden. Im Folgenden verwenden wir Compound Rules zur Spezifikation der Transformation von visuellen Kontrakten nach JML, da die Spezifikation unabhängig von den Implementierungsdetails eines Frameworks erfolgen soll.

Der Ansatz bietet verschiedene Vorteile. Aufgrund der Charakteristiken von Graphtransformationen, sind die Regeln genau und operationalisierbar, d.h. aus ihnen kann eine Implementierung abgeleitet werden. Eine weitere Eigenschaft, die dieser Ansatz von Graphtransformationen übernimmt, ist eine hohe Flexibilität im Hinblick auf eine leichte Anpassbarkeit der Regeln. Eine hohe Wiederverwendung der Regeln erlauben die Compound Rules auf zwei Arten. Zum einen muss bei der Anpassung der Transformation an eine neue Zielsprache fast nur der textuelle, templatebasierte Teil angefasst werden. Da die unterschiedlichen Design-by-Contract-Sprachen strukturell sehr ähnlich aufgebaut sind, wird so eine hohe Wiederverwendung möglich. Zum anderen muss bei einer Änderung des Metamodells für die Quellmodelle nur der graphische Teil der Regeln angefasst werden, während die Templates nahezu unverändert bleiben. Mit der Verwendung von Transformation Units kann zudem sichergestellt werden, dass der Zielcode in einer bestimmten Reihenfolge erzeugt wird.

5.3 Spezifikation der Transformation nach JML

Dieser Abschnitt erläutert die Spezifikation der Transformation von UML-Klassendiagrammen und visuellen Kontrakten nach JML. Bei der Transformation gehen wir stets von syntaktisch korrekten UML-Modellen aus. Wir formulieren die Transformation durch eine Menge von Compound Rules, deren Anwendung UML-Klassendiagramme und visuelle Kontrakte nach JML abbildet. Bei der Spezifikation der Transformation der visuellen Kontrakte

nach JML haben wir den intuitiven Ansatz aus Kapitel 4 gewählt, der es uns erlaubt, die Vor- und Nachbedingungen unabhängig voneinander zu betrachten.

Im Folgenden beschreiben wir als erstes das Konzept der Compound Rules anhand eines Beispiels. Danach erläutern wir das Konzept der Transformation Units anhand einer konkreten Transformation Unit aus unserer Übersetzung. Zum Abschluss dieses Abschnitts erläutern wir den Ablauf einer Transformation nach JML. In Anhang B finden sich die zur Transformation notwendigen Compound Rules sowie eine Dokumentation der notwendigen Transformation Units.

5.3.1 Modelltransformationen mit Compound Rules

Die grundlegende Idee des Ansatzes in [Küs04] ist, dass eine Modelltransformation zur Übersetzung eines Modells aus einer Quell- in eine Zielsprache mit zwei synchronisierten Modelltransformationen auf der Quell- und Zielsprache beschrieben werden kann. Die Modelltransformationen werden mit einer Menge von Compound Rules spezifiziert. Eine Compound Rule besteht aus zwei Teilen: Einer Transformationsregel auf der Quellsprache (*Quellregel*) und einer Transformationsregel auf der Zielsprache (*Zielregel*). Oftmals handelt es sich bei den Transformationsregeln auf der Quellsprache um identische Transformationen, die das Quellmodell unverändert lassen. Auch die von uns verwendeten Compound Rules zur Übersetzung visueller Kontrakte nach JML lassen das Quellmodell unverändert. Es werden lediglich zusätzliche Statusinformationen hinzugefügt, um bereits bearbeitete Elemente zu markieren.

Die in [EHSW99a, EHSW99b, HKT02b, HKT02a] verwendeten Compound Rules (Metaregeln) zur Transformation von UML-Modellen in einen semantischen Bereich bestehen aus mehreren Teilen. Zum einen besteht eine Regel aus einem UML-Teil, der die Quellregel darstellt und einem textuellem Teil für den semantischen Bereich, der die Zielregel darstellt. Die Quellregel zeigt die zu übersetzenden Modellelemente als Instanzen des UML-Metamodells in der abstrakten UML-Syntax an. Die Zielregel wird textuell als Produktionsregel dargestellt. Auf beiden Seiten der Zielregel können insbesondere die Namen der Modellelemente aus der Instanz des UML-Metamodells verwendet werden, so dass diese Namen in die Zielsprache übernommen werden können. Außerdem können in den Regeln für die Zielsprache neue Nichtterminale, also zu ersetzende Muster, eingeführt werden.

	$r_S : L_S ::= R_S$	$r_T : L_T ::= R_T$

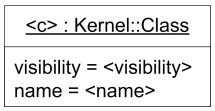
tr_1		ϵ ::=
		`//@ refine <name>.java`
		
		`<visibility> class <name> {`
		
		`#Attributes(<c>)#`
		
		`#Methods(<c>)#`
		
		`}`

Abbildung 5.2: Transformationsregel für visuelle Kontrakte

Die in dieser Arbeit verwendeten Compound Rules $r : (r_s, r_t)$ bestehen aus zwei Teilen, einem UML-Teil und einem JML-Teil. Abbildung 5.2 zeigt eine Compound Rule tr_1 zur Übersetzung von visuellen Kontrakten nach JML. Sowohl r_s als auch r_t können als Graphtransformationsregeln interpretiert werden [CMR+97].

Generell spezifiziert die Quellregel $r_s : L_S ::= R_S$ die Transformation auf dem Quellmodell, die Zielregel $r_t : L_T ::= R_T$ spezifiziert die Transformation auf dem Zielmodell. Die Transformation auf dem Quellmodell wird in abstrakter Syntax mit einem Paar von Instanzen des UML-Metamodells für Klassendiagramme und der in Abschnitt 3.3 vorgestellten Erweiterung für visuelle Kontrakte beschrieben. Wenn die Quellregel keine Veränderung auf dem Zielmodell durchführt ($L_S = R_S$), stellen wir nur die linke Seite einer Regel dar (vgl. Abbildung 5.2). L_T in der Zielregel aus Abbildung 5.2 enthält ein Epsilon (d.h. die Regel hat eine leere Vorbedingung) und einen JML-Teil R_T (rechte Seite), der bei Anwendung der Regel erzeugt wird.

Quell- und Zielregeln werden über gemeinsam benutzte Variablen miteinander verknüpft. Die Variablen werden mit spitzen Klammern (`<variable>`) dargestellt. In der obigen Regel ist zum Beispiel der Rollenname `<c>` des Objekts vom Typ `Class` eine Variable, die in beiden Regeln verwendet wird. Die Variablen werden durch konkrete Werte ersetzt, sobald im abzubildenden UML-Modell eine mögliche Regelanwendung gefunden wird. Die in unterschiedlichen Compound Rules verwendeten, gleichnamigen Variablen sind voneinander unabhängig, d.h. der Gültigkeitsbereich der Variablen wird durch die Compound Rules definiert. In den Transformationsregeln können Nichtterminale eingesetzt werden (vergleiche [EHSW99a, EHSW99b, HKT02b]). Diese sind durch Buchstabenketten, die mit dem Symbol # umrandet sind, gekennzeichnet. Nichtterminale ermöglichen eine inkrementelle Erzeugung des Zielmodells.

Die Anwendung einer Übersetzung von einer Quell- in eine Zielsprache mit einer Compound Rule, unter der Annahme, dass $X = x_1, ..., x_n$ die Menge der Variablen von L_S ist, funktioniert wie folgt (vgl. [Küs04]):

1. Im Quellmodell wird das Auftauchen der linken Seite L_S der Quellregel gesucht. Das Auftreten einer linken Seite L_S im Quellmodell wird im Folgenden als *Source Match* bezeichnet.
2. Wenn ein Source Match gefunden wird, werden den Variablen konkrete Werte zugewiesen. Dies führt zu einer Variablenbelegung X_I.
3. Die linke Seite L_T der Zielregel wird mit den Variablen instanziert, was wir mit $L_T(X_I)$ kennzeichnen. Informell gesprochen, weist die Variableninstanziierung jeder Variablen einen Wert zu.
4. Im Zielmodell wird das Auftauchen der instanziierten linken Seite der Zieltransformation gesucht (*Target Match*).
5. Die rechte Seite R_T der Zieltransformation wird mit den Werten der Variablen instanziiert.
6. Das Auftreten der instanziierten linken Seite wird durch die instanziierte rechte Seite R_T der Zieltransformationsregel ersetzt.
7. Die rechte Seite der Quellregel R_S wird mit der Variablenbelegung X_I instanziert, was mit $R_S(X_I)$ gekennzeichnet wird.
8. Das Auftreten der linken Seite wird durch die instanziierte rechte Seite der Quelltransformationsregel im Quellmodell ersetzt.

Abbildung 5.2 zeigt eine einfache Transformationsregel für visuelle Kontrakte. Diese Transformationsregel erzeugt eine initiale JML-Datei zu einer Klasse aus einem Design-Klassendiagramm. Während der Transformation wird diese initiale JML-Datei mit weiteren JML-Annotationen angereichert. Im folgenden Beispiel wenden wir diese Regel auf die Klasse `OnlineShop` aus Abbildung 3.1 an.

Mit der Bestimmung des Source-Match ergibt sich eine Variablenbelegung $\{visibility = public, name = OnlineShop\}$. In diesem Beispiel enthält die linke Seite der Zieltransformation keine Variablen, die instanziiert werden müssen. Somit kann in dem Beispiel direkt zum nächsten Schritt übergegangen werden. Das Symbol (ϵ) auf der linken Seite der Zieltransformation kennzeichnet eine leere JML-Datei. Dadurch wird erreicht, dass die Zieltransformation nur einmalig und nur zu Beginn des Übersetzungsprozesses matcht.

Nach der Instanziierung der rechten Seite der Zieltransformation kann im

Zielmodell die linke Seite mit der rechten Seite der Zieltransformation ersetzt werden. Konventionsmäßig entspricht die Ersetzung bei der Verwendung eines ϵ einer simplen Hinzufügung des Ausdrucks. Hierbei sind die entsprechenden Variablenbelegungen zu beachten. Weiterhin werden die Nichtterminale `Attributes(<c>)` und `Methods(<c>)` erzeugt. Die Variable `<c>` entspricht einem Objektidentifier einer Instanz der Klasse `Class`. Diese Variable wird verwendet, um auch in den nachfolgenden Schritten der Codegenerierung bestimmen zu können, zu welcher Klasse der aktuell generierte JML-Code gehört. Durch die Verwendung derart strukturierter Nichtterminale werden die Matches der Vorbedingung der Zieltransformation eingeschränkt, und somit sichergestellt, dass weiterer Code an der korrekten Stelle in der JML-Spezifikation eingefügt wird. Abbildung 5.5 (Schritt 1) zeigt den generierten JML-Code.

5.3.2 Kontrollstrukturen

Im Folgenden beschreiben wir die in [Küs04] eingeführten Kontrollstrukturen, um die Anwendungsreihenfolgen der Compound Rules explizit beeinflussen zu können. Die implizite Anwendungsreihenfolge, die sich aus den linken Regelseiten der Compound Rules ergibt, ist für eine korrekte Übersetzung der Modelle nicht ausreichend. Zusätzlich zu den jeweiligen linken Seiten einer Regel gibt der hier verwendete Formalismus vor, dass eine Regel nur angewendet werden kann, wenn sie im jeweiligen Kontext noch nicht eingesetzt wurde.

Gemäß [Küs04] können die Compound Rules mit Hilfe von Transformation Units [Kus00] gruppiert werden, um einer Modelltransformation zusätzliche explizite Kontrollstrukturen hinzuzufügen. Eine Transformation Unit besteht aus einer Menge von Compound Rules, die mit Kontrollstrukturen annotiert werden. Durch die Organisation der Compound Rules mit Regelmengen ergibt sich ein turingvollständiger Ansatz [HP01]. Die Regelmengen werden dann in Sequenzen von Regelmengen organisiert, wobei jede Regelmenge als eigenständige Ebene betrachtet wird. Innerhalb einer Regelmenge können die Regeln nichtdeterministisch angewendet werden. Eine Transformation Unit besteht somit aus einer Menge von Compound Regeln in Kombination mit einem Kontrollausdruck, welcher die Organisation der Regeln in den Regelmengen organisiert und bestimmt, ob eine Regel nur einmal oder so oft wie möglich angewendet werden soll.

Syntaktisch werden die Ebenen der Regeln mit Sequenzen von Regelmengen spezifiziert. So spezifiziert unter der Annahme das p_1, p_2, p_3 drei Regeln dar-

stellen der Ausdruck $\{p_1, p_2\}, p_3$ zwei Ebenen. Die erste Ebene besteht aus p_1, p_2 und die zweite Ebene besteht aus p_3. Der Kontrollausdruck besagt, dass die Regel der zweiten Ebene erst angewendet werden kann, wenn alle Regeln aus der ersten Ebene angewendet wurden. Mit einer Annotation wird festgelegt, ob die Regelmenge nur einmal oder so oft wie möglich ausgeführt werden soll. Der Ausdruck $p \downarrow$ besagt zum Beispiel, dass die Regel so oft wie möglich angwendet werden soll. In [Küs04] ist die Syntax von Kontrollausdrücken wie folgt definiert.

Definition 5.1 (Syntax von Kontrollausdrücken [Küs04]) *Sei P eine Menge von Regelnamen. Dann wird ein Kontrollausdruck zur Steuerung der Regelanwendung folgendermaßen definiert:*

$Exp ::= Seq \mid Set \mid Exp \downarrow \mid name$

$Seq ::= \langle Exp, ..., Exp \rangle$

$Set ::= \{Exp, ..., Exp\}$

mit $name \in P$.

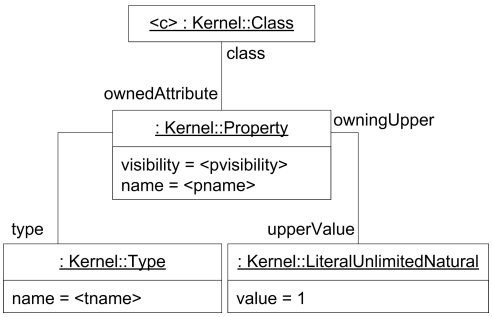
Abbildung 5.3 zeigt drei Compound Rules zur Generierung von JML-Code für die `get`-Operationen, die zur Verwaltung der Attribute und Assoziationen aus dem Klassendiagramm verwendet werden.

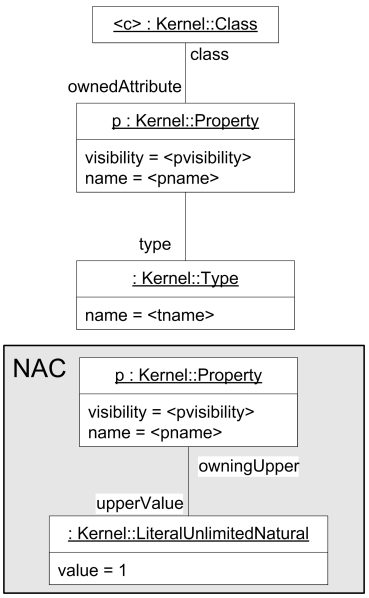
Die Regel tr_2 erzeugt für jedes Attribut mit der Kardinalität 1 eine entsprechende `get`-Operation. Diese Operation bekommt die Sichtbarkeit des Attributes aus dem Klassendiagramm zugewiesen. Der Rückgabewert entspricht dem Typ des Attributes. Der Operationenname setzt sich aus dem Begriff „get“ und dem Namen des Attributes zusammen. Damit entspricht die generierte Signatur der Operation den Vorgaben aus Abschnitt in 4.2. Die Signatur wird in der JML-Spezifikation zusätzlich mit dem JML-Schlüsselwort `pure` (eingeschachtelt in Java-Kommentare) annotiert.

Der Grund für die Annotation der automatisch generierten `get`-Operationen ist, dass diese in den aus den visuellen Kontrakten automatisch generierten JML-Kontrakten verwendet werden, um auf die Inhalte von Attributen zugreifen zu können. Da JML-Ausdrücke jedoch grundsätzlich keine Seiteneffekte haben dürfen, können Operationen in den JML-Annotationen nur verwendet werden, wenn diese keine Seiteneffekte auf den Systemzustand haben. Die Information, dass eine Operation keine Seiteneffekte hat, wird dem JML-Compiler mit dem Schlüsselwort `pure` mitgeteilt.

Die Regel tr_3 erzeugt die Spezifikation für Attribute, die eine Kardinalität größer als eins haben. In diesem Fall wird von den `get`-Operationen ein Ar-

	$r_S : L_S ::= R_S$	$r_T : L_T ::= R_T$

tr_2		`#Attributes(<c>)# ::=`
		`<pvisibility> /*@ pure @*/`
		`  <tname> get<pname>();`
		`#Attributes(<c>)#`

tr_3		`#Attributes(<c>)# ::=`
		`<pvisibility> /*@ pure @*/`
		`  <tname>[] get<pname>();`
		`#Attributes(<c>)#`

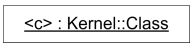
tr_4		`#Attributes(<c>)# ::=` ϵ

Abbildung 5.3: Transformationsregeln zur Generierung der Spezifikation für von `get`-Operationen

ray (dem Datentyp werden zwei eckige Klammern „[]“ angehängt) zurückgegeben. Mit der negativen Anwendungsbedingung wird sichergestellt, dass diese Transformationsregel nur ausgeführt wird, wenn die Kardinalität der Operation ungleich eins ist.

In beiden Regeln erzeugt die Zieltransformation wiederum ein Nichtterminal `Attributes(<c>)`, so dass in folgenden Modelltransformationen weitere JML-Spezifikationen für Operationen eingefügt werden können. Die Regel tr_4 entfernt dieses Nichtterminal. Die Regel darf allerdings erst ausgeführt werden, wenn alle Attribute einer Klasse bearbeitet wurden. Andernfalls werden die fehlenden Attribute nicht mehr bearbeitet, da die Regeln tr_2 und tr_3 nicht mehr matchen.

Damit die Regel tr_4 wirklich erst ausgeführt wird, wenn alle Attribute einer Klasse bearbeitet wurden, ist der Kontrollausdruck $\langle \{tr_2, tr_3\} \downarrow, tr_4 \rangle$ notwendig. Mit diesem Kontrollausdruck wird sichergestellt das die Regeln tr_2 und tr_3 solange in einer nichtdeterministischen Reihenfolge ausgeführt werden, bis sie nicht mehr angewendet werden können. Erst danach wird die Transformationsregel tr_4 einmal ausgeführt und somit das Nichtterminal `Attributes(<c>)` aus dem JML-Code gelöscht.

5.4 Arten der auszuführenden Transformationsregeln

In diesem Kapitel haben wir bisher einen Ansatz zur Beschreibung einer formalen Modelltransformation ausgewählt und beschrieben. In Kapitel 4 haben wir die Übersetzung der visuellen Kontrakte nach JML anhand von Beispielen beschrieben. Zur Detaillierung dieser beispielhaften Beschreibung haben wir eine formale Beschreibung der Transformation von visuellen Kontrakten nach JML ausgearbeitet.

Die Transformation wird in mehreren Schritten ausgeführt. Im Folgenden beschreiben wir den Ablauf der Transformation einer Vorbedingung eines visuellen Kontraktes in eine JML-Vorbedingung. Die Compound Rules zur Übersetzung einer Vorbedingung eines visuellen Kontraktes nach JML finden sich in Anhang B. Die Übersetzung einer Nachbedingung eines visuellen Kontraktes in eine JML-Spezifikation mit Compound Rules funktioniert ähnlich, da wir hier die in Kapitel 4 beschriebene intuitive Übersetzung, in welcher die JML-Vor- und Nachbedingungen nahezu identisch strukturiert sind, gewählt haben.

1. Für jede Klasse der Anwendung muss eine initiale, leere JML-Datei erzeugt werden, die im Verlauf der Transformation mit weiterem Inhalt gefüllt wird (Regel tr_1).

2. Die get-Operationen für den Zugriff auf die Attribute einer Klasse innerhalb eines visuellen Kontraktes müssen als `pure` gekennzeichnet werden. Operationen, die in JML-Ausdrücken verwendet werden, dürfen keine Seiteneffekte auf den Systemzustand haben. Die Information, dass eine Operation keine Seiteneffekte hat, muss dem JML-Compiler mit Hilfe des Schlüsselwortes `pure` mitgeteilt werden (Regeln tr_2, $\ldots tr_4$, vgl. Abschnitt 5.3.2).

3. Eine JML-Spezifikation besteht aus zwei Teilen, einem statischen und einem dynamischen Teil (vgl. auch Abschnitt 4.3.2). Die Standard-Java-Deklaration einer Operation selbst stellt den statischen Teil der Spezifikation dar. Das Verhalten der Operation wird mit Vor- und Nachbedingungen beschrieben. Die Verhaltensbeschreibung, d.h. die JML-Annotation einer Operation, ist eingebettet in Java-Kommentare vor der Spezifikation der Operationssignatur (vgl. auch Listing 4.3). Es sind Transformationsregeln zu erzeugen, die eine Operationssignatur sowie einen Rahmen für die JML-Annotation einer Operation mit Vor- und Nachbedingungen schaffen.

   (a) Als erstes muss eine Operationssignatur ohne die Aufrufparameter und ein Rahmen für die JML-Annotationen erzeugt werden (Regeln op_1, $\ldots$, op_4, Anhang B).

   (b) Eine Operation kann beliebig viele Parameter besitzen, daher können die Parameter nicht in einem Schritt mit dem Rest der Operationssignatur erzeugt werden. Bei der Erzeugung der Parameterliste ist der Name und Typ eines Parameters zu berücksichtigen (Regeln pa_1, $\ldots$, pa_4, Anhang B).

4. Die JML-Spezifikation einer Vorbedingung eines visuellen Kontraktes muss eine Teilgraphensuche in einem Systemgraphen durchführen. Der Systemgraph ist eine Repräsentation des Systemzustandes mit Objekten und Links. Dazu müssen alle Rollen einer Vorbedingung im Systemgraphen gefunden werden.

   (a) Die Teilgraphensuche beginnt beim Element `self`. Vor dem Start der Suche müssen die Attribute des Elements `self` aus dem visuellen Kontrakt überprüft werden (Regeln $prsv_1$ und $prsv_2$, Anhang B).

(b) Die zu generierenden JML-Ausdrücke müssen eine Teilgraphensuche, wie in Abschnitt 4.3 beschrieben, realisieren. Hierzu müssen die JML-Quantoren geeignet ineinander geschachtelt werden. Ein Laufzeitobjekt erfüllt eine Rolle in einem visuellen Kontrakt, wenn die Inhalte seiner Attribute den vorgegebenen Inhalten im visuellen Kontrakt entsprechen und wenn das Laufzeitobjekt mit weiteren Laufzeitobjekten gemäß den Vorgaben im visuellen Kontrakt verbunden ist. Bevor JML-Code zur Überprüfung der Attribute mit unseren Transformationsregeln erzeugt wird, wird Code erzeugt, um die Links zwischen den Laufzeitobjekten zu überprüfen. Das heißt, als erstes wird eine korrekte Schachtelung der notwendigen JML-Quantoren erzeugt.

Bei den Transformationsregeln ist zu unterscheiden zwischen den Regeln, die den JML-Code zum Aufbau der geschachtelten JML-Quantoren startend vom Element `self` (Regeln $prsr_1, \ldots, prsr_6$, Anhang B) erzeugen und Regeln, die den JML-Code zum Aufbau der geschachtelten JML-Quantoren ausgehend von den restlichen Objekten weiter erzeugen. Beim weiteren Aufbau des Suchbaums ist zu unterscheiden zwischen Regeln, die einen noch nicht besuchten Knoten in den Suchbaum einfügen (Regeln $prer_1, \ldots, prer_6$) und Regeln, die eine noch nicht überprüfte Kante zu einem bereits besuchten Knoten prüfen (Regeln $prer_7$ und $prer_8$).

Nach dem Abschluss der Suche sind eventuell noch überflüssige Nichtterminale zu löschen (Regeln $prsr_7, prsr_8, prer_9, prer_{10}$).

(c) Nach dem Aufbau des Suchbaums muss JML-Code zur Überprüfung der Inhalte der Attribute einer gebundenen Rolle generiert werden (Regeln $pre_1, \ldots, pre_5$, $pat_1, \ldots, pat_6$).

(d) In einem visuellen Kontrakt können Variablen verwendet werden, um die Inhalte unterschiedlicher Attribute miteinander zu vergleichen. In der JML-Spezifikation müssen die Variblen mit konkreten Zugriffsmethoden ersetzt werden, um auf die Inhalte von Attributen, die den Inhalt der Variablen vorgeben, zugreifen zu können (Regeln $pv_1, \ldots, pv_3$).

Im Folgenden erläutern wir die hier dargestellten Transformationsregeln kurz an einem Beispiel. Abbildung 5.4 zeigt einen einfachen visuellen Kontrakt zur Detaillierung der Operation `cartAdd`. Wir erläutern die Transformation der Vorbedingung dieses visuellen Kontraktes nach JML. Dazu zeigen wir, was die von uns erstellten Compound Rules auf dem Zielmodell (JML-Code)

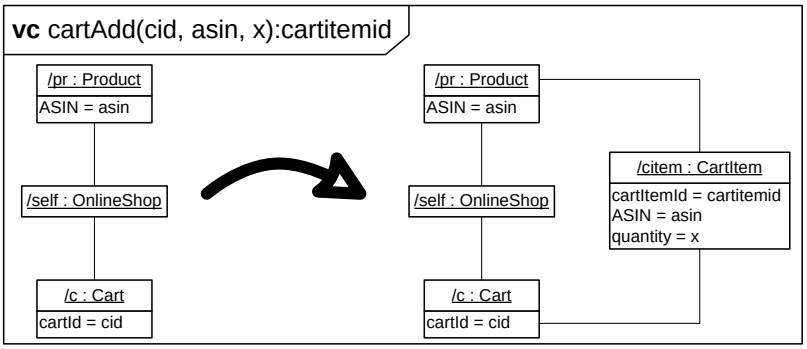


Abbildung 5.4: Beispiel für einen visuellen Kontrakt

jeweils am Ende der oben genannten Schritte erzeugt haben (siehe Abbildung 5.5 und 5.6). Auf das Quellmodell gehen wir in unseren Erläuterungen nicht detailliert ein, da dieses aufgrund des zugrundeliegenden Metamodells sehr groß wird (vgl. auch Compound Rules in Anhang B).

Die Operation `cartAdd` gehört zur Klasse `OnlineShop`. Im ersten Schritt erzeugen unsere Regeln eine initiale, leere JML-Datei mit zwei Nichtterminalen (vgl. Schritt 1 in Abbildung 5.5). Die Nichtterminale enthalten beide die Variable `<exclass>`. Über Variablen wird das Zielmodell mit dem Quellmodell verknüpft. Die Variable entspricht dem Objektidentifier eines Objektes zur Repräsentation der Klasse `OnlineShop` gemäß dem UML 2.0 Metamodell. Das Objekt ist vom Typ `Class` (vgl. auch Abbildung 5.2).

In Schritt 2 wird das Nichtterminal `Attributes(<exclass>)` lediglich gelöscht, da die Klasse `OnlineShop` keine Atttribute enthält.

In Schritt 3a wird das Nichtterminal für Operationen der Klasse `OnlineShop`, die mit visuellen Kontrakten detailliert wurden, durch entsprechende Signaturen für Java-Methoden und Rahmen für die Vor- und Nachbedingungen ersetzt. Sowohl die Signaturen als auch die Vor- und Nachbedingungen werden im Schritt 3a noch nicht vollständig erzeugt. Schritt 3a in Abbildung 5.4 zeigt die bereits erzeugte Signatur und JML-Constraints für die Operation `cartAdd`. Dem Zielmodell werden während der Ausführung drei Nichtterminale hinzugefügt. Die Variable `<exop>` wird zur Verknüpfung des Zielmodells mit der Operation `cartAdd` im Quellmodell benötigt.

In Schritt 3b wird das Nichtterminal `Argsstart(<exop>)` durch die Aufrufparameter der Operation `cartAdd` ersetzt. Damit ist die Signatur der Methode im Zielmodell vollständig.

Der Schritt 4a entfällt in unserem Beispiel, da das Element `self` in dem

Schritt 1	`//@ refine "OnlineShop.java";`
	
	`public class OnlineShop {`
	
	`#Attributes(<exclass>)#`
	
	`#Methods(<exclass>)#`
	
	`}`

Schritt 2	`//@ refine "OnlineShop.java";`
	
	`public class OnlineShop {`
	
	`#Methods(<exclass>)#`
	
	`}`

Schritt 3a	`//@ refine "OnlineShop.java";`
	
	`public class OnlineShop {`
	
	`/*@ public normal_behavior`
	`  @ #Pre(<exop>)#`
	`  @ #Post(<exop>)#`
	`public String cartAdd(#Argstart(<exop>)#);`
	
	`...`
	`}`

Schritt 3b	`//@ refine "OnlineShop.java";`
	
	`public class OnlineShop {`
	
	`/*@ public normal_behavior`
	`  @ #Pre(<exop>)#`
	`  @ #Post(<exop>)#`
	`public String cartAdd(`
	`   String cid , String ASIN, int quantity);`
	
	`...`
	`}`

Abbildung 5.5: Beispieltransformation — Schritt 1 - 3b

Schritt 4b

```
//@ refine "OnlineShop.java";

public class OnlineShop {

/*@ public normal_behavior
  @ requires (\exists Cart c;
      this.getCart().values().contains (c);
      #PreCheck(<exCart>)# &&
      (\exists Product pr;
        this.getProduct().values().contains(pr);
        #PreCheck(<exproduct>)# &&
        true ) && true);
  @ #Post(exop)#
public String cartAdd(
   String cid , String ASIN, int quantity);

...
}
```

Schritt 4c

```
//@ refine "OnlineShop.java";

public class OnlineShop {

/*@ public normal_behavior
  @ requires (\exists Cart c;
      this.getCart().values().contains (c);
      c.getCartId().equals(cid) &&
      (\exists Product pr;
        this.getProduct().values().contains(pr);
        pr.getASIN().equals(ASIN) &&
        true ) && true);
  @ #Post(<exop>)#
public String cartAdd(
   String cid , String ASIN, int quantity);

...
}
```

Abbildung 5.6: Beispieltransformation — Schritt 4b, 4c

visuellen Kontrakt aus Abbildung 5.4 keine Attribute enhält.

In Schritt 4b (siehe Abbildung 5.6) werden die `exists`-Quantoren geschachtelt. Hierbei ist zu beachten, dass tatsächlich nur die Quantoren geschachtelt werden, d.h. die Rollen aus der Vorbedingung im visuellen Kontrakt werden der JML-Vorbedingung hinzugefügt. Die Attribute der Rollen werden noch nicht überprüft. In Schritt 4b werden nur bereits die entsprechende Nichtterminale `PreCheck(<excart>)` bzw. `PreCheck(<exproduct>)` als Platzhalter eingefügt. Die beiden Variablen `excart` und `exproduct` entsprechen den Objektidentifiern der Objekte zur Repräsentation der Rolle `c:Cart` bzw. `pr:Product` gemäß dem Metamodell für visuelle Kontrakte. Zu beachten ist, dass während der Ausführung der Transformationen in Schritt 4b zweimal `true` eingefügt wird. Hier weicht der mit den Compound Rules generierte JML-Code leicht von dem JML-Code aus Abschnitt 4.3 ab. Der Grund ist, dass während der Ausführung von Schritt 4b auch temporär erzeugte Nichtterminale gelöscht werden müssen. Um dennoch eine korrekte Verundung beizubehalten, werden einige Nichtterminale nach `true` abgeleitet. Die Semantik des JML-Ausdrucks ändert sich damit jedoch nicht. Die Generierung des Ausdrucks `true` stellt eine Vereinfachung unserer Regelmenge dar. Alternativ können auch Compound Rules erstellt werden, die keine Ableitung von Nichtterminalen nach `true` durchführen. In diesem Fall würde sich die Anzahl der Compound Rules aus Schritt 4b jedoch nahezu verdoppeln.

In Schritt 4c werden die Nichtterminale `PreCheck(<excart>)` beziehungsweise `PreCheck(<exproduct>)` durch JML-Code zur Überprüfung der Attributinhalte der Rollen ersetzt. Der Schritt 4d entfällt in unserem Beispiel.

5.5 Fazit

In diesem Kapitel haben wir eine formale Beschreibung der Transformation von UML-Klassendiagrammen und visuellen Kontrakten nach JML beschrieben. Hierzu haben wir als erstes Anforderungen an eine formale Spezifikation formuliert und verschiedene Ansätze zur Spezifikation von Modelltransformationen betrachtet. Der ausgewählte Ansatz kombiniert Graphtransformationen und templatebasierte Ansätze zu sogenannten Compound Rules.

Danach haben wir die Übersetzung der visuellen Kontrakte in eine JML-Spezifikation mit Compound Rules detailliert beschrieben. Diese detaillierte, formale Beschreibung kann als Grundlage zur Erstellung einer manuellen Implementierung der Transformation visueller Kontrakte nach JML verwendet

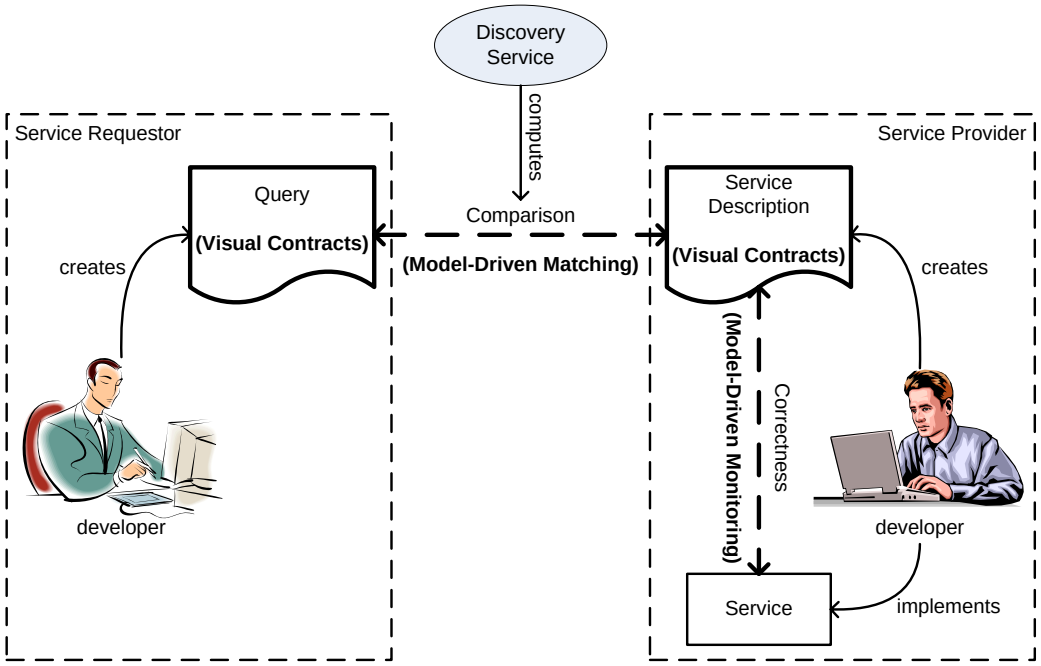


Abbildung 5.7: Visuelle Kontrakte zur Unterstützung der Modellierung, Implementierung und Suche in einer SOA

werden.

Mit der hier beschriebenen Transformation können wir unser Model-Driven Monitoring realisieren. Eine Implementierung der Transformation erlaubt eine automatische Übersetzung der visuellen Kontrakte nach JML, um die Korrektheit einer manuellen Implementierung zu überwachen. Das Model-Driven Monitoring kann also von einem Service Provider verwendet werden, um sicherzustellen, dass eine manuell erstellte Implementierung korrekt gegenüber einer (veröffentlichten) Spezifikation (Servicebeschreibung) ist. Im Folgenden zeigen wir, wie wir visuelle Kontrakte zur Beschreibung einer Suchanfrage einsetzen können. Darauf aufbauend erlaubt unser Model-Driven Matching einen Vergleich einer Suchanfrage mit einer Servicebeschreibung, um eine Suche nach einem Service auf Basis der Servicebeschreibung zu ermöglichen (vgl. Abbildung 5.7).

Kapitel 6

Model-Driven Matching

Die Vision serviceorientierter Architekturen ist, dass ein Softwareentwickler beim Bedarf einer bestimmten Funktionalität, die nicht selbständig realisiert werden kann oder soll, eine entsprechende Suchanfrage an ein zentrales Verzeichnis (*Discovery Service*) sendet. Das Ergebnis dieser Suchanfrage ist eine Liste von Servicebeschreibungen von potentiellen Anbietern dieser Funktionalität.

Ein Beispiel ist eine eLearning-Anwendung, die für eine Universität entwickelt wird. Dieses System kann seinen Anwendern die Funktionalität anbieten, Bücher, die für einen bestimmten Kurs empfohlen werden, zu bestellen. Die Abwicklung eines Geschäftsprozesses zur Bestellung eines Buches liegt jedoch außerhalb des Zuständigkeitsbereichs der eLearning-Anwendung. Wenn diese Funktion dennoch angeboten werden soll, muss ein Service eines Buchhändlers wie zum Beispiel Amazon (siehe auch Kapitel 3.2.1) aufgerufen werden, wenn ein Anwender ein Buch bestellt.

Ein wichtiger zu beachtender Punkt an dieser Stelle ist, dass ein Service Requestor weiß, welcher Service benötigt wird, aber nicht notwendigerweise weiß, wie der Service eines Providers aufgerufen wird. Die *Semantik* eines gesuchten Service ist somit bekannt, aber nicht seine *Syntax* [LRE+06]. Die Syntax eines Service benötigt der Service Requestor erst, wenn er sich für einen konkreten Service entschieden hat und diesen in seinen Client einbinden möchte.

In diesem Kapitel zeigen wir, wie die Semantik eines Service und einer Suchanfrage modellbasiert mit visuellen Kontrakten und Ontologien beschrieben werden kann. Zum Vergleich einer Suchanfrage mit einer Servicebeschreibung führen wir ein flexibles, modellbasiertes Matching-Konzept ein (vgl. Abbil-

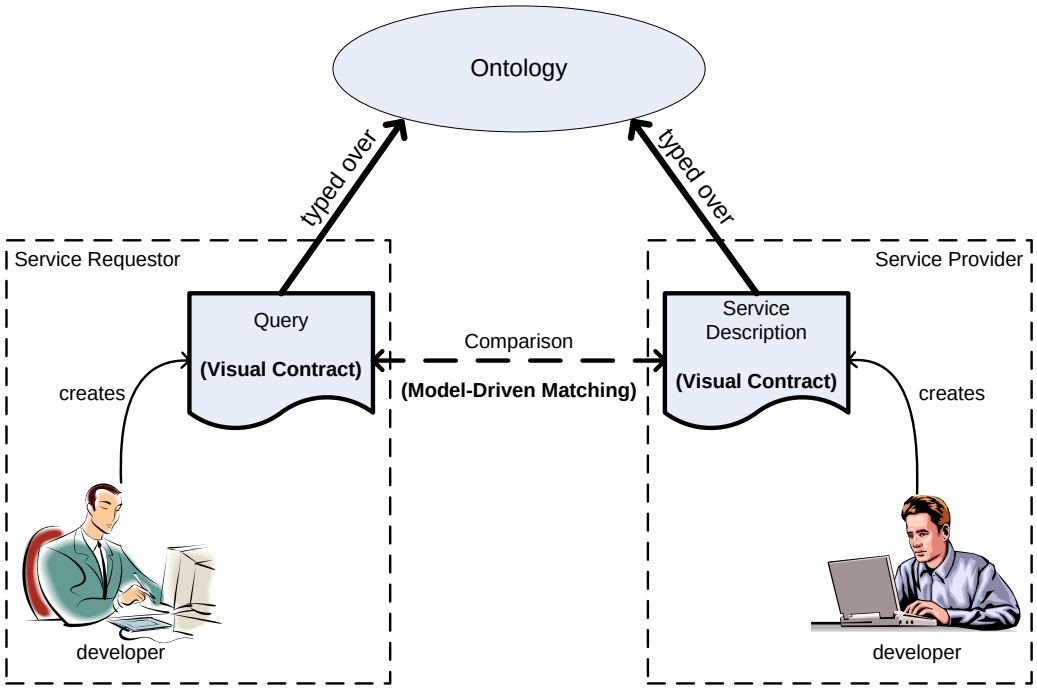


Abbildung 6.1: Model-Driven Matching

dung 6.1). Im folgenden Abschnitt diskutieren wir jedoch zunächst verwandte Arbeiten.

6.1 Verwandte Arbeiten

Zentral für die Suche nach einem geeignetem Service sind die Servicebeschreibungen. Eine Servicebeschreibung wird von einem Service Provider bei einem Discovery Service hinterlegt. Der Discovery Service kann die Servicebeschreibungen mit einer Suchanfrage vergleichen, um ein Matching zu berechnen. Informationen, die nicht Teil der hinterlegten Servicebeschreibungen sind, können nicht für ein automatisiertes Matching verwendet werden. Daher konzentrieren wir uns bei der Analyse der in diesem Abschnitt beschriebenen verwandten Arbeiten auf die Möglichkeiten, die eine Servicebeschreibung zur Realisierung einer semantischen Suche bietet. Wir untersuchen existierende Web-Service-Standards sowie wissenschaftliche Ansätze.

6.1.1 Web-Service-Standards

Die Definition einer Schnittstelle ist die technische Beschreibung der öffentlichen Schnittstellen eines Web Service. Mit ihr werden die möglichen Anfragen (Requests) und Antworten (Response) eines Web Service beschrieben. Die Web Services Description Language (WSDL — gesprochen „whistle“) [BL05, CHL+05, CMRW05] des World Wide Web Consortiums (W3C) definiert einen plattform-, programmiersprachen- und protokollunabhängigen XML-Standard zur Beschreibung der Schnittstellen eines Web Service. Es werden im Wesentlichen die einzelnen Operationen eines Web Service sowie die Parameter und Rückgabewerte dieser Operationen, die von außen zugänglich sind, definiert.

Diese Art der Beschreibung von Schnittstellen wird vom W3C und auch in diversen anderen Publikationen als „documentation of the mechanics of the message exchange“ [BHRT03, BHM+04] bezeichnet. Diese „mechanischen Informationen“ (Signaturen und Adressen eines Web Service) müssen für den Aufruf (Binding) eines Web Service bekannt sein. Eine automatische Suche eines Web Service basierend auf WSDL ist jedoch nicht möglich. Mit einem WSDL-Dokument wird nur die Syntax eines Web-Service-Aufrufs beschrieben. Aussagen über die Semantik eines Web Service werden mit einem WSDL-Dokument nicht gemacht. Natürlich kann ein Entwickler eventuell anhand des Namens und der Signatur einer Operation, wie zum Beispiel `orderBook(isbn:String)`, ihre Bedeutung abschätzen. Derart explizite Operationsnamen sind jedoch technisch nicht erforderlich und helfen bei einer automatisierten Suche auch nur sehr eingeschränkt.

Universal Description, Discovery and Integration (UDDI) [Bel02, UDD01a, UDD01b] ist eine industrielle Initiative zur Standardisierung eines Verzeichnisdienstes zur Veröffentlichung von Servicebeschreibungen. Hierzu wird in dem Standard das Datenmodell für ein Repository von Service Providern und Services sowie eine API zum Zugriff auf dieses Repository festgelegt. Das Datenmodell besteht aus drei Elementen. Die Business-Entities umfassen die weißen und gelben Seiten des Verzeichnisdienstes. Sie enthalten Informationen über den Service Provider (zum Beispiel den Unternehmensnamen) oder die Kategorie eines Service. Die Business-Services sind ein Teil der „green pages“ und stellen nichttechnische Informationen über einen Service, wie z.B. den betrieblichen Nutzen, bereit. Die Binding-Templates sind ein weiterer Teil der „green pages“. Sie enthalten die technischen Web-Service-Beschreibungen, welche für die Einbindung eines Web Service in einen Client notwendig sind. Eine genaue Beschreibung der Datenstrukturen findet sich

zum Beispiel in [Her04].

Eine WSDL-Datei kann auf einem UDDI-Server abgelegt und mit zusätzlichen Informationen in Form von erklärendem Text oder Schlüsselwörtern annotiert werden [CER02]. Erklärender Text und Schlüsselwörter sind eine Möglichkeit, um die Semantik eines Web Service zu beschreiben. Jedoch wird bei diesen semantischen Annotationen die Suche nur sehr unzureichend unterstützt. Das Ergebnis einer Suchanfrage ist bei der Verwendung von Schlüsselwörtern sehr stark davon abhängig, ob der richtige Suchbegriff verwendet wird und nicht irgendein Synonym für ein bestimmtes Schlüsselwort.

Heutzutage werden diese semantischen Probleme durch eine intensive Integration des Softwareentwicklers in den Suchprozess gelöst. Wenn ein Service Requestor einen Web Service benötigt, so durchsucht der Entwickler einen UDDI-Server von Hand. Dazu verwendet er die auf dem Server vorgegebene Kategorisierung oder auf Schlüsselwörtern basierende Suchmechanismen und vor allem die menschliche Intuition. Für die genauere Auswahl eines Web Service muss der Entwickler die detaillierten textuellen Beschreibungen lesen. Werkzeuge wie z.B. WebSphere Studio [Flu03] integrieren die Suche auf einem UDDI-Server in eine integrierte Entwicklungsumgebung (IDE). Wenn der Entwickler einen Web Service ausgewählt hat, so lädt WebSphere Studio die WSDL-Datei vom UDDI-Server und erstellt ein Proxy, welches zum Aufruf des Service von einem Client verwendet werden kann.

Um das Problem der unterschiedlichen synonymen Schlüsselwörter zu lösen, ist es notwendig, dass Service Requestor und Service Provider dasselbe Vokabular für die Beschreibung einer Suchanfrage und eines Service verwenden. In diesem Kontext spielen Ontologien eine wichtige Rolle. Gemäß [FB02] definiert eine Ontologie ein Vokabular, welches allen Beteiligten in einer Domäne bekannt ist:

> „Ontologies are formal and consensual specifications that provide a shared and common understanding of a domain, an understanding that can be communicated across people and organization systems.“

Die Web Ontology Language (OWL) [MvH04, SWLM04, BvHH+04] oder DAML+OIL [CvHH+01] sind XML- [BPSM+04] bzw. genauer gesagt RDF-basierte [MM04, Bec04, KJC04] industrielle Standards zur Beschreibung von Ontologien. In diesen Standards geht es darum, die Terme einer Domäne und deren Beziehungen formal so zu beschreiben, dass auch Softwaresysteme ihre Bedeutung verarbeiten („verstehen“) können. Ein Problem dieser Sprachen ist, dass sie sich auf die Beschreibung statischer Informationen konzentrieren.

Das Verhalten eines Service kann mit diesen Sprachen nur beschrieben werden, wenn das Verhalten als Term in der Ontologie standardisiert worden ist. Dies widerspricht der Anforderung nach einer flexiblen Beschreibungsform (vgl. auch Abschnitt 2.6).

Die Sprache OWL-S (Semantic Markup for Web Services) [MBH+04] ist eine Ontologie-basierte (OWL-basierte) Auszeichnungssprache zur Beschreibung der Aufgaben und Eigenschaften eines Web Service. Die Ziele von OWL-S sind es das Finden, den Aufruf, die Komposition und das Monitoring von Web Services zu automatisieren. Die Beschreibung eines Web Service mit OWL-S unterscheidet drei unterschiedliche Informationskategorien: Informationen über den Service Provider (z.B. einfache Kontaktinformationen), funktionale Beschreibungen und weitere Informationen zur Spezifikation der Charakteristiken eines Web Service (hauptsächlich festgehalten mit Referenzen auf existierende Taxonomien). Die interessanteste Kategorie sind die funktionalen Beschreibungen. Die funktionalen Beschreibungen enthalten Ein- und Ausgabeparameter, sowie zusätzliche Vorbedingungen und Effekte. Die Vorbedingungen werden in der OWL-S Spezifikation auch als „externe Bedingungen“ beschrieben, die vor dem Aufruf eines Service wahr sein müssen. Unter den Effekten werden Seiteneffekte bei der Ausführung eines Service verstanden. Solange jedoch die Semantik eines Web Service nur mit Vorbedingungen und Effekten in Form von Termen, die in einer Ontologie vorgegeben sind (ohne Zusammenhang zwischen Vorbedingung und Effekten), beschrieben werden kann, fehlt auch OWL-S die notwendige Flexibilität zur Beschreibung verschiedener Web Services.

6.1.2 Wissenschaftliche Ansätze

In diversen wissenschaftlichen Publikationen finden sich unterschiedliche Ansätze zur Beschreibung von Services beziehungsweise Web Services, um eine semantische Suche zu ermöglichen.

In den beiden Ansätzen [MSZ01] und [DCLK02] wird die Semantik eines Service mit einer Referenz auf einen Term in einer Ontologie beschrieben. Dieser Term beschreibt die Funktionalität eines Service. Der Ansatz [MSZ01] basiert auf der Darpa Agent Markup Language (DARPA)[1][CvHH+01], während [DCLK02] DAML-S (DAML Services)[2] [Mar03, Coa04] verwendet und in eine UDDI-Registry integriert. Beide Ansätze erlauben die Definition von

[1] Vorgänger von OWL
[2] Vorgänger von OWL-S

speziellen Termen in einer Ontologie, um Services zu beschreiben.

Die Terme in einer Ontologie in [MSZ01] sind sehr spezifisch, sie enthalten zum Teil sogar spezifische Produktinformationen. So findet sich in der in dem Artikel verwendeten Ontologie z.B. ein Term `buyLufthansaTicket`, um Services aus dem Geschäftsbereich Flugreisen beschreiben zu können. In dem Artikel werden dann komplexe Strategien aufgezeigt, die beschreiben, wie diese detaillierten Informationen für die Suche nach spezifischen Services genutzt werden können. In [DCLK02] werden abstrakte Service-Typen, wie z.B. `CarRentalService`, eingeführt, die zur semantischen Beschreibung von konkreten Services verwendet werden können.

In beiden Ansätzen wird es schwer, die Semantik von neuen, innovativen Services, die z.B. das Bieten auf billige Tickets ermöglichen, zu beschreiben. Hierzu muss die Ontologie um einen entsprechenden Term erweitert werden, der zur Beschreibung eines Service genutzt werden kann. Die kontinuierliche Integration komplexer, neuer Terme in eine Ontologie widerspricht jedoch der dynamischen Natur des Webs, dem Umfeld für Web Services. So muss zum Beispiel in [MSZ01] die Ontologie bereits angepasst werden, wenn eine neue Handelsmarke eingeführt oder eine existierende Handelsmarke gelöscht (z.B. durch Umbenennung) wird. Änderungen in einer Ontologie sind jedoch grundsätzlich zwischen den beteiligten Partnern abzustimmen, was sehr zeitaufwendig ist. Änderungen in einer Ontologie erfordern auch häufig eine Anpassung der Anwendungen, welche diese Ontologie verwenden. Aufgrund der notwendigen Absprachen und aufwendigen Prozesse zum Anpassen einer Ontologie, enthält eine Ontologie oftmals nur sehr allgemeine Begriffe, um eine hohe Anwendbarkeit sicherzustellen. Zur Lösung dieses Problems wird zum Beispiel in [DOH+01] eine Klassifikation der Services basierend auf ihrem Typ oder der Industriebranche vorgeschlagen. Diese Kategorisierung ist jedoch offensichtlich nicht ausreichend detailliert, um spezifische Services unterscheiden zu können.

Wenn zur Beschreibung eines Service nur ein einzelner Term verwendet wird, so wird auch deren operationale Natur mißachtet. Bei der Ausführung eines Service erwartet ein Service Requestor bestimmte Veränderungen, das heißt der Zustand eines Systems oder die reale Welt wird auf eine für den Service Requestor signifikante Art geändert. Es wird zum Beispiel ein Auftrag erzeugt, eine Zahlung durchgeführt oder ein Termin festgelegt. Servicebeschreibungen sollten diese operationale Natur von Services berücksichtigen. Hierfür ist die Verwendung von Vor- und Nachbedingungen — bekannt aus Design by Contract Ansätzen [Mey92a, Mey97] (vgl. auch Abschnitt 2.1) — eine geeignete Möglichkeit.

In [PKPS02, SVSM03] werden Vor- und Nachbedingungen zur Beschreibung von Services verwendet. In beiden Ansätzen referenzieren die Vor- und Nachbedingungen Terme aus einer spezifischen Ontologie. Während in [PKPS02] ein Matching nur für einzelne Terme in Vor- und Nachbedingung gezeigt wird, werden in [SVSM03] verschiedene Terme kombiniert (z.B. `CardCharged-TicketBooked-ReadyforPickup`), um eine Vor- oder Nachbedingung zu beschreiben. Damit wird es möglich, unterschiedliche, aber ähnliche Services (z.B. Buchen eines Flugtickets, das Kunden geschickt wird versus Buchen eines Tickets, das am Schalter abgeholt wird) zu beschreiben ohne der Ontologie für jeden Service einen neuen Begriff hinzufügen zu müssen.

Obwohl diese einfache Kombination unterschiedlicher Terme bereits in die richtige Richtung zeigt, denken wir, dass dieses Konzept noch weiter entwickelt werden sollte. Eine Servicebeschreibung sollte komplexe Strukturen über den Termen einer Ontologie zur semantischen Beschreibung eines Service enthalten können. Der Vorteil dieses Ansatzes ist, dass die Ontologie selbst nur sehr allgemeine und grundlegende Begriffe, die stabil sind gegenüber möglichen Veränderungen, einer Domäne enthalten muss. Zur Beschreibung eines Service können die Terme dann individuell kombiniert werden, um eine spezifische Semantik zu beschreiben. Damit wird die notwendige Stabilität in einer Ontologie und zugleich eine hohe Flexiblität zur Beschreibung verschiedener Services erreicht (vgl. Anforderungen in Abschnitt 2.6).

Ein Nachteil aller hier beschriebenen Ansätze ist, dass sie zur Beschreibung eines Service eine XML-Kodierung verwenden, die sehr schnell sehr lang und sehr komplex werden kann. Die semantischen Servicebeschreibungen sind damit für einen Softwareentwickler, der immer noch eine zentrale Rolle in einer serviceorientierten Architektur einnimmt, nicht intuitiv verständlich. Weiterhin kann die (manuelle) Erstellung der Servicebeschreibungen nur sehr schwer in heutige modellbasierte Softwareentwicklungsprozesse integriert werden.

In den folgenden Abschnitten beschreiben wir einen neuen, modellbasierten Ansatz basierend auf unseren visuellen Kontrakten zur Beschreibung der Semantik von Services und Suchanfragen sowie ein entsprechendes Matching-Konzept. Der Ansatz ist visuell und erlaubt eine flexible und detaillierte Beschreibung unterschiedlicher Services.

6.2 Semantische Beschreibung von Services und Suchanfragen

In diesem Abschnitt beschreiben wir einen visuellen Ansatz zur Beschreibung der Semantik von Services und Suchanfragen. Dazu kombinieren wir visuelle Kontrakte mit standardisierten Ontologien.

6.2.1 Ontologien

Bei einer automatisierten semantischen Suche eines Service Requestors nach einem geeignetem Service ist ein Vergleich der Servicebeschreibungen mit einer Suchanfrage notwendig. Dies setzt voraus, dass sich sowohl Service Requestor als auch Service Provider zu einer gemeinsamen Begriffswelt bekennen. Diese gemeinsame Begriffswelt wird als Ontologie bezeichnet. Eine Ontologie wird spezifisch für eine Anwendungsdomäne definiert. Die Erstellung und Verwendung von akzeptierten Ontologien ist gemäß Berners-Lee et. al. [BLHL01] eine der Schlüsseltechnologien für das Semantische Web (*Semantic Web*) und die semantische Informationssuche.

Im Folgenden beschäftigen wir uns mit einer geeigneten Darstellungsform von Ontologien und nicht mit Verfahren zur Erstellung von Ontologien. Der Aufbau einer vertrauenswürdigen Ontolgie ist eher eine philosophische und sozialökonomische Herausforderung. Auf Fragen zur Erstellung geeigneter Ontologien wird zum Beispiel in [GW02] eingegangen.

In der Informatik gibt es mindestens zwei Forschungsbereiche, die sich mit der Gestaltung von Ontologien beschäftigen. Im Bereich der künstlichen Intelligenz (KI) werden Ontologien als Teil der Forschung zur Wissensrepräsentation verwendet. Die KI untersucht semantische Netze und Regelsysteme auf der Basis von Prädikatenlogik und entwickelt dazu verschiedene Beschreibungsformalismen, zu denen auch Beschreibungssprachen für Ontologien gehören. Jedoch haben sich diese Ansätze in der Praxis noch nicht durchgesetzt. Nur in ganz wenigen Fällen werden in „echten“ KI-Anwendungen Ontologien dazu verwendet, um aufgrund von Basisdaten weiter reichende Inferenzen (Beweisketten) zu konstruieren, die dann als Entscheidungshilfe dienen können [Beh03, GDD05, GDD04].

Weiterhin gibt es verschiedene Ansätze, die zur Gestaltung von Ontologien Software-Engineering-Techniken verwenden. Insbesondere wird die Unified Modeling Language (UML) zur Repräsentation von Ontologien verwendet,

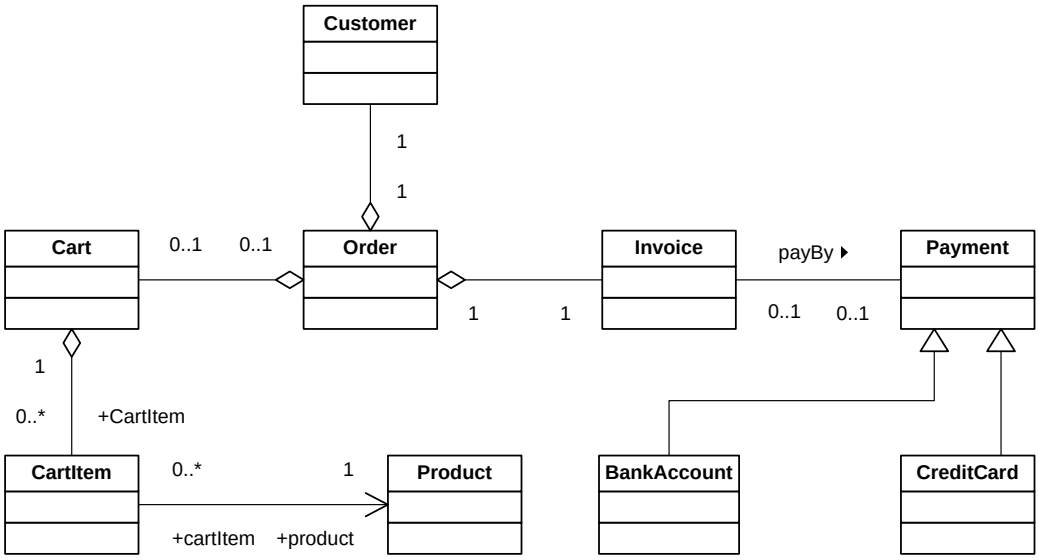


Abbildung 6.2: Ontologie zum Verkaufen von Produkten

da die UML ein weit verbreiteter Standard ist, der den Entwicklern aus der Praxis bekannt ist und bei der Entwicklung von Softwaresystemen eingesetzt wird [GDD05, GDD04]. Eine Ontologie ist in diesem Fall üblicherweise eine hierarchisch geordnete Struktur von Klassen und Unterklassen und Assoziationen (siehe z.B. [FFJ+02, DGD05, Cra01, CP99]).

Über das Maß und die Methode der Systematisierung von Ontologien gehen in der Praxis die Auffassungen jedoch auseinander. Viele Ontologien sind eher strukturierte Glossare (*controlled vocabulary*). Andere Ontologien sind hierarchisch in Form von Taxonomien aufgebaut. Die Art der Hierarchisierung ist jedoch sehr unterschiedlich und geht bis zu formal definierten Vererbungshierachien in objektorientierten Modellen [Beh03]. Auch die Verwendung von Attributen in Ontologien wird unterschiedlich gehandhabt. Zum Beispiel spielen gemäß [WVV+01] Attribute in Ontologien und insbesondere bei der Anwendung von Ontologien eher eine untergeordnete Rolle.

In unserem Ansatz verwenden wir UML-Klassendiagramme mit Klassen, Assoziationen sowie Spezialisierungen zur Darstellung einer Ontologie. Attribute und Operationen verwenden wir in unseren Ontologien nicht, um den Vergleich zweier visueller Kontrakte zu vereinfachen (vgl. Abschnitt 6.3). Eine Erweiterung der Ontologien um Attribute hätte jedoch keinen wesentlichen Einfluss auf unser Gesamtkonzept zur Modellierung, Implementierung und Suche von Services in serviceorientierten Architekturen. Weiterhin sind wir der Meinung, dass die Beschränkung der Ontologien keine wesentliche Ein-

schränkung ist, da mit unserem Model-Driven-Matching-Konzept lediglich *potentiell* geeignete Services gefunden werden sollen. Die finale Auswahl eines Service liegt immer noch beim Softwareentwickler (vgl. auch Abschnitt 6.5).

Abbildung 6.2 zeigt eine einfache Ontologie für das Verkaufen von Produkten. Ein Auftrag (`Order`) zur Bestellung einer Menge von Produkten referenziert den Kunden des Auftrages (`Customer`) und einen Warenkorb (`Cart`), der verschiedene Einträge (`CartItem`) enthalten kann. Weiterhin referenziert ein `CartItem` ein Produkt. Ein Auftrag kann entweder mit Kreditkarte `CreditCard`) oder mit einer Lastschrift (`BankAccount`) bezahlt werden. Die Spezialisierung zeigt, dass die Klassen `CreditCard` und `BankAccount` Spezialisierungen des grundlegenden Konzeptes `Payment` sind.

6.2.2 Kombination von Ontologien und visuellen Kontrakten

Eine geeignete Technologie für die Verwaltung von Services in einer serviceorientierten Architektur sollte — gemäß den Anforderungen aus Abschnitt 2.6 — flexibel sein. Diese Anforderung ist jedoch konfliktär zu der Tatsache, dass für einen Vergleich von Servicebeschreibungen und Suchanfragen Service Provider und Requestor eine gemeinsame Sprache verwenden müssen. Ontologien sind standardisierte Kollektionen von Konzepten und können damit eine derartige gemeinsame Sprache darstellen (vgl. Abschnitt 6.2.1).

Wie bereits zuvor geschildert, können Ontologien jedoch nicht einfach angepasst werden, um flexibel die Semantik neuer Services zu beschreiben. Ontologien sind das Ergebnis eines (internationalen) Übereinkommens zwischen verschiedenen Parteien. Bei der Erstellung oder Veränderung einer Ontologie kann es sich um einen langwierigen Prozess handeln, der nicht mit der Einigung auf eine neue Ontologie abgeschlossen ist, da auch die Anwendungen, die eine bestimmte Ontologie verwenden, angepasst werden müssen. Dies behindert die Dynamik, die den Märkten aufgrund der stetig zunehmenden Globalisierung heutzutage zweifellos unterliegt.

Eine flexible Beschreibung der Semantik von Services und Suchanfragen wird möglich, wenn zur Beschreibung Ausdrücke über den Begriffen einer Ontologie gebildet werden können. In Abschnitt 6.1 wurden bereits verschiedene Ansätze, die zum Teil zwar formal jedoch nicht flexibel genug zur Beschreibung verschiedener Services sind, diskutiert. Weiterhin bieten diese Ansätze keine geeignete Integration in heutige modellbasierte Softwareentwicklungsprozesse und sie berücksichtigen auch nicht die Rolle des Softwareentwick-

lers bei der Erstellung von Servicebeschreibungen. Während eine formale, präzise Beschreibung notwendig ist, um ein Matching von Servicebeschreibungen und Suchanfragen zu ermöglichen, wird ein Formalismus gesucht, der im Einklang mit einer visuellen Darstellung der Ontologien und heutigen Software-Engineering-Techniken ist und auch durch einen Entwickler intuitiv einsetzbar ist.

Visuelle Kontrakte sind eine geeignete Möglichkeit, um die Semantik von Services und Suchanfragen zu beschreiben. Sie besitzen eine Formalisierung auf der Basis von Graphtransformationen und eine visuelle Darstellung, die der UML sehr ähnlich ist. Wir schlagen also zur Beschreibung der Semantik von Services und Suchanfragen eine Vorgehensweise vor, die auf Ontologien basiert und den Mechanismus von visuellen Kontrakten verwendet.

Bevor wir auf das Matching eingehen, wollen wir erst die Struktur der visuellen Kontrakte zur Beschreibung der Semantik eines Service und einer Suchanfrage erläutern. Bei den visuellen Kontrakten für das Suchen und Finden eines Service handelt es sich um die in Kapitel 3 eingeführten visuellen Kontrakte. Für unser Model-Driven-Matching-Konzept machen wir bezüglich der Komplexität der visuellen Kontrakte jedoch verschiedene Einschränkungen.

Gemäß den Vorgaben in Kapitel 3 ist ein visueller Kontrakt eingebettet in den Arbeitsbereich eines Frame und besteht aus verschiedenen Kompositionsstrukturdiagrammen. Jedes dieser Diagramme ist getypt über einem Klassendiagramm. Zur Realisierung unseres Model-Driven-Matching-Konzeptes setzen wir voraus, dass es sich bei dem Klassendiagramm um eine Ontologie, die sowohl vom Service Provider als auch vom Service Requestor verwendet wird, handelt. Bei den visuellen Kontrakten für das Suchen und Finden eines Service erlauben wir keine negativen Vor- oder Nachbedingungen sowie keine Parameterlisten und Rückgabewerte[3]. Der visuelle Kontrakt kann jedoch einen Namen bekommen, der als Titel des Frames dargestellt wird (siehe Abbildung 6.3). Mit dem Verzicht auf die Verwendung von Operationssignaturen kommen wir dem Service Requestor entgegen. Bei der Suche nach einem Service kennt der Service Requestor die Semantik eines gesuchten Service (z.B. Bestellung ausführen), er hat jedoch nicht notwendigerweise Anforderungen an die Syntax eines gesuchten Service (vgl. Seite 135 und

[3]Gemäß des UML-Metamodells aus Abbildung 3.6 wird in diesem Fall ein visueller Kontrakt, der eine `Collaboration` ist, nicht über eine Klasse `CollaborationUse` an eine Operation gebunden. Die Möglichkeit ist gegeben, da die Assoziation zwischen den Metamodellklassen `CollaborationUse` und `Operation` aufgrund der gegebenen Kardinalitäten optional ist.

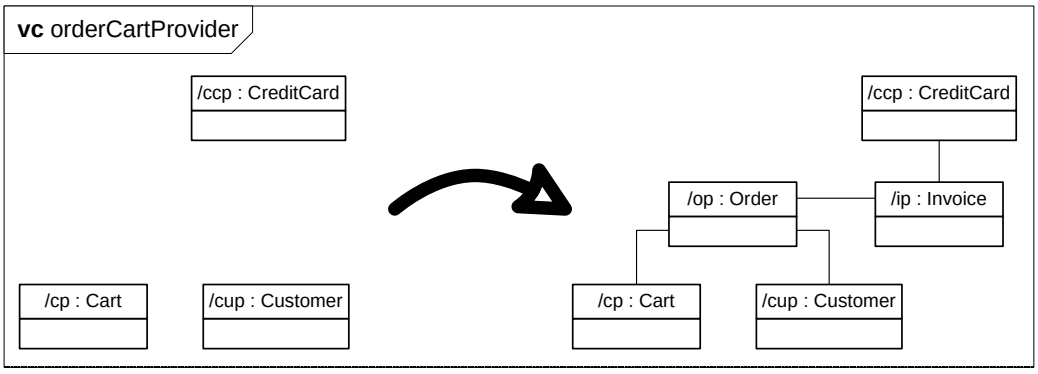


Abbildung 6.3: Provider-Kontrakt für eine Operation zur Bestellung eines Einkaufswagens

Abschnitt 6.3.2). Mit dem Verzicht auf negative Vor- und Nachbedingen vereinfachen wir zudem die Formulierung der Regeln und damit auch die Suche nach einem Service für den Service Requestor. Der Service Requestor kann damit leichter eine Menge potentiell geeigneter Services finden. Da die den visuellen Kontrakten zugrundeliegenden Ontologien keine Attribute besitzen, können auch in den visuellen Kontrakten keine Attribute verwendet werden. Als Semantik behalten wir die lose Interpretation von Graphtransformationsregeln in Form von Graphtransitionen — basierend auf der Theorie des Double-Pullback-Ansatzes [Hec98, HEWC01] — bei.

Gemäß diesen Einschränkungen kann ein Service für die Bestellung eines Einkaufswagens, der mit Kreditkarte bezahlt werden kann, durch den in Abbildung 6.3 dargestellten visuellen Kontrakt beschrieben werden. Die linke und rechte Seite des visuellen Kontraktes sind getypt über der Ontologie aus Abbildung 6.2. Der visuelle Kontrakt besagt, dass der Service als Eingabe Daten über den zu bestellenden Warenkorb, den Auftraggeber der Bestellung sowie Kreditkarteninformationen (Elemente auf der linken Seite des visuellen Kontraktes) benötigt. Das Ergebnis der Ausführung dieses Service ist, dass eine Bestellung und eine Rechnung für den bestellten Warenkorb erzeugt werden. Die Rechnung wird mit der Kreditkarte bezahlt. Die lose Semantik unserer visuellen Kontrakte (auf der Basis von Graphtransitionen) erlaubt eine unvollständige Beschreibung des Service. Das heißt, bei der Ausführung des Service können zusätzliche Effekte auftreten, die in dem visuellen Kontrakt dargestellten Effekte sind jedoch garantiert.

Mit der Kombination von Ontologien und visuellen Kontrakten wird es einfach möglich, verschiedene Arten von Services für eine standardisierte Menge von Begriffen zu beschreiben. So kann zum Beispiel für die Ontologie aus

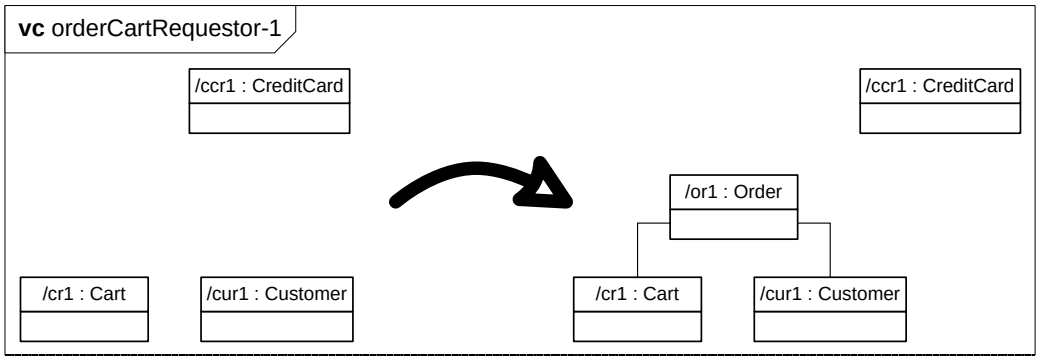


Abbildung 6.4: Requestor-Kontrakt — Beispiel 1

Abbildung 6.2 ein visueller Kontrakt erstellt werden, der beschreibt, wie ein weiteres Produkt zu einem bestehendem Warenkorb hinzugefügt wird oder wie ein Produkt wieder aus einem Warenkorb gelöscht wird (vgl. auch visuelle Kontrakte in Kapitel 3). Da reale Ontologien wesentlich größer sind als das in Abbildung 6.2 gezeigte Beispiel, kann so die Semantik diverser Services mit Hilfe von visuellen Kontrakten flexibel beschrieben werden ohne die gegebene, standardisierte Ontologie in einem langwierigen Prozess ändern zu müssen[4].

Visuelle Kontrakte sollen jedoch nicht nur zur Beschreibung eines angebotenen Service sondern auch zur Formulierung einer Suchanfrage eingesetzt werden können. Die Lesart eines *Provider-Kontraktes* (Servicebeschreibung) unterscheidet sich dabei leicht von der Lesart eines *Requestor-Kontraktes* (Suchanfrage). Aus der Sicht eines Service Providers spezifiziert die linke Seite eines visuellen Kontraktes die Vorbedingung des Service, d.h. eine Situation oder einen Zustand, der für einen Service bereitgestellt werden muss, damit er ausgeführt werden kann. Die rechte Seite des visuellen Kontraktes beschreibt die Nachbedingung, d.h. sie charakterisiert eine Situation oder einen Zustand nach der erfolgreichen Ausführung eines Service.

Dagegen beschreibt die linke Seite eines Requestor-Kontraktes, welche Informationen der Service Requestor an einen Service zu schicken bereit ist. Die rechte Seite beschreibt eine Situation oder einen Zustand, welche ein

[4]In einer mit der sd&m AG durchgeführten Fallstudie zur Evaluation visueller Kontrakte für den Einsatz in serviceorientierten Architekturen [LRE+06, EGJ+06] haben wir z.B. aus dem objektorientierten fachlichen Referenzmodell der VAA (Versicherungs Anwendungs Architektur) [Ges01] eine Ontologie (bzw. ein fachliches Referenzmodell) abgeleitet, das aus ca. 40 Klassen besteht. Mit diesen Klassen konnten wir die Semantik von ca. 45 Services (Aktivitäten), die zum Abschluss eines neuen Versicherungsvertrages notwendig sind, mit visuellen Kontrakten beschreiben.

Service Requestor durch den Aufruf eines Service erreichen möchte. Abbildung 6.4 zeigt einen Requestor-Kontrakt für die Suche nach einem Händler, der zu einem gegebenen `Cart` eine Bestellung erstellen kann. Dieser Kontrakt drückt aus, dass ein Requestor Informationen über ein `cr1:Cart`, eine `ccr1:CreditCard` und einen `cur1:Customer` bereitstellen kann (siehe linke Seite des visuellen Kontraktes). Der Requestor sucht nach einem Service, der eine Bestellung (Element `or1:Order` auf rechter Seite des visuellen Kontraktes inklusive der Konnektoren zu den Elementen `cr1:Cart` und `cur1:Customer`) für die Eingabedaten erstellen kann. Gemäß der losen Semantik unserer visuellen Kontrakte spezifiziert die Suchanfrage eines Service Requestors nur Effekte, die auf jeden Fall benötigt werden. Zusätzliche Effekte werden mit einem visuellen Kontrakt nicht ausgeschlossen.

6.3 Matching von visuellen Kontrakten

Wie in nahezu jedem Softwareentwicklungsprozess hat auch ein Service Requestor bei der Entwicklung eines Clients die Motivation, verschiedene Anforderungen zu erfüllen. Zur Realisierung der Anforderungen muss ein Client bestimmte Funktionalitäten aufweisen. Im Kontext von serviceorientierten Architekturen ist die Vorgehensweise jedoch etwas spezieller (wenn auch nicht neu, wenn man an komponentenbasierte Systeme denkt), da die benötigte Funktionalität nicht vollständig vom Service Requestor implementiert wird. Stattdessen wird bei der Entwicklung eines Clients auf andere Services zurückgegriffen. Dazu muss ein Service Requestor in einer serviceorientierten Architektur nach Services, welche die benötigte Funktionalität zur Verfügung stellen, suchen können. Gleichzeitig muss der Service Requestor wissen, ob er die Anforderungen des Service Providers, um einen Service erfolgreich aufrufen zu können, erfüllen kann.

In Abschnitt 6.2 haben wir gezeigt, wie ein Service Provider die Semantik eines Service beziehungsweise ein Service Requestor die Semantik einer Suchanfrage mit visuellen Kontrakten beschreiben kann. Ein Discovery Service muss in der Lage sein, eine Servicebeschreibung mit einer Suchanfrage zu vergleichen, um entscheiden zu können, ob ein Service die Anforderungen eines Service Requestors erfüllt.

In diesem Abschnitt geben wir eine intuitive Definition, der Bedeutung von „Ein Service Provider erfüllt die Anforderungen eines Service Requestors“ beziehungsweise der „visuelle Kontrakt eines Service Providers (Provider-Kontrakt) *matcht* auf den visuellen Kontrakt einer Suchanfrage (Requestor-

Kontrakt)“. Weiterhin geben wir eine operationale Interpretation dieser Aussage an.

6.3.1 Matching — intuitive Definition

Um automatisiert bestimmen zu können, wann ein Service Provider die Anforderungen eines Service Requestors erfüllt, müssen die Vor- und Nachbedingungen der Provider- und Requestor-Kontrakte miteinander verglichen werden. Eine Voraussetzung für einen geeigneten Vergleichsalgorithmus ist, dass beide Kontrakte über dem gleichen Klassendiagramm, d.h. über der gleichen Ontologie, getypt sind.

Ein Vergleich des Provider-Kontraktes aus Abbildung 6.3 und des Requestor-Kontraktes aus Abbildung 6.4 zeigt zum einen, dass der Service Requestor offensichtlich bereit ist, alle Informationen zu liefern, die vom Service Provider benötigt werden. Gemäß der Vorbedingung des Provider-Kontraktes sind dies Informationen über einen `Cart`, einen `Customer` und eine `CreditCard`. Zum anderen zeigt der Vergleich, dass der Service alle Ergebnisse produziert, die vom Service Requestor erwartet werden. Der Service Requestor erwartet die Erzeugung einer Bestellung (`Order`), die den `Cart` und den `Customer` umfasst. Der Service erzeugt eine Bestellung und zusätzlich eine Rechnung (`Invoice`) für die Bestellung (zahlbar mit Kreditkarte).

Diese Abhängigkeiten können auf einfache Subgraph-Beziehungen zurückgeführt werden (vgl. Abbildung 6.5). Wenn die Vorbedingung eines Provider-Kontraktes ein Subgraph der Vorbedingung eines Requestor-Kontraktes ist, dann stellt der Service Requestor alle zur Ausführung des Service benötigten Informationen zur Verfügung (und eventuell einige zusätzliche Informationen). Wenn die Nachbedingung eines Requestor-Kontraktes ein Subgraph der Nachbedingung eines Provider-Kontraktes ist, dann generiert der Service alle Ergebnisse, die vom Service Requestor erwartet werden und eventuell weitere Informationen. In unserem Beispiel erzeugt der Service zusätzlich eine Rechnung, welche von dem Service Requestor zu begleichen ist.

Jedoch kann bei einem intuitivem Vergleich einer Servicebeschreibung mit einer Suchanfrage ein Service die Anforderungen eines Service Requestors erfüllen, obwohl die oben beschriebenen Subgraph-Relationen nicht gegeben sind. Dies ist insbesondere dann der Fall, wenn (1) der Service Requestor bereit ist, Eingabeinformationen zu liefern, an deren Verarbeitung er keine Bedingung stellt; und (2) wenn vom Service Provider nicht alle angebotenen Informationen benötigt werden, um die vom Service Requestor verlangten

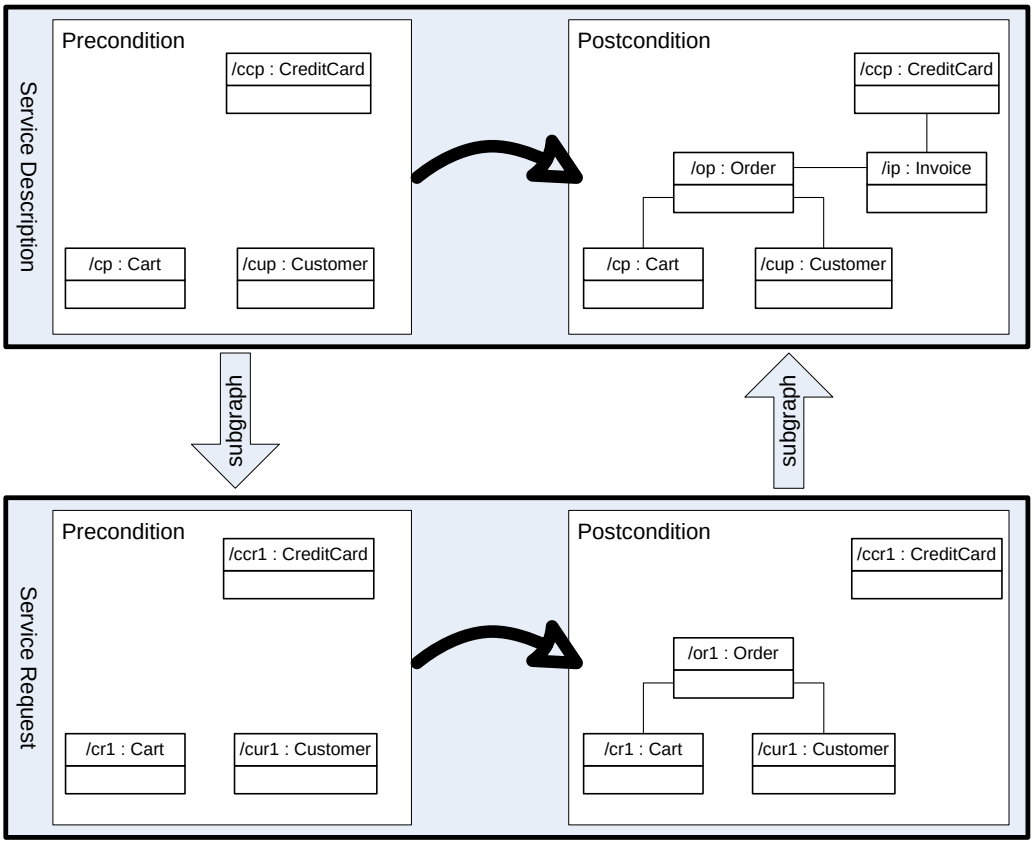


Abbildung 6.5: Subgraph-Relationen zwischen den Vor- und Nachbedingungen — Beispiel 1

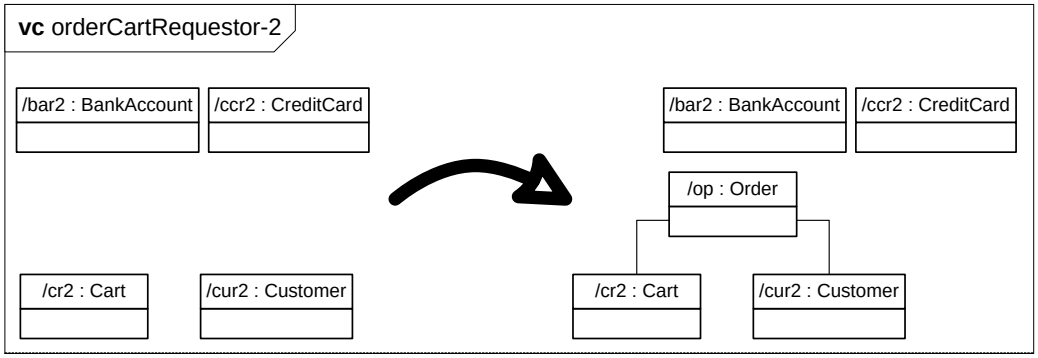


Abbildung 6.6: Requestor-Kontrakt — Beispiel 2

Effekte zu berechnen.

Abbildung 6.6 zeigt einen Requestor-Kontrakt, der gegenüber dem Requestor-Kontrakt aus Abbildung 6.4 nur leicht verändert wurde. Gemäß dem neuen visuellen Kontrakt ist der Service Requestor bereit, zusätzlich Kontoinformationen an den Service Provider zu schicken, d.h. der Requestor ist bereit, die Bestellung per Kreditkarte oder per Lastschrift zu bezahlen. Auch in diesem Fall ist die Vorbedingung des Provider-Kontraktes ein Subgraph der Vorbedingung des Requestor-Kontraktes (beim Requestor-Kontrakt ist lediglich ein weiteres Element hinzugekommen). Jedoch ist die Nachbedingung des Requestor-Kontraktes kein Subgraph der Nachbedingung des Provider-Kontraktes mehr (vgl. Abbildung 6.7), da das Element `BankAccount` nicht Teil der Nachbedingung des Service Providers ist. In einem intuitiven Vergleich erfüllt der Service jedoch durchaus die Anforderungen des Service Requestors. Der Service Requestor stellt dem Service Provider eine Menge von Informationen (`CreditCart`, `BankAccout`) zur Verfügung an deren Verarbeitung er keine Bedingung stellt (außer der Bedingung, dass diese Elemente nicht gelöscht werden dürfen). Betrachtet man lediglich die Elemente der Servicebeschreibung an deren Verarbeitung der Service Requestor Bedingungen stellt (`Order`, `Cart`, `Customer`), so erfüllt der Service die Anforderungen des Service Requestors.

Zur Berechnung eines Match zwischen einem Provider- und einem Requestor-Kontrakt ist der Vergleich der Vor- und Nachbedingungen somit nicht ausreichend. Stattdessen müssen die Effekte der visuellen Kontrakte miteinander verglichen werden. Die Effekte eines visuellen Kontraktes ergeben sich aus dem Vergleich der Vor- und Nachbedingungen. Alle Elemente, die Teil der Nachbedingung sind und nicht verändert werden, sind nicht Teil dieser Effekte. Wenn die Effekte eines visuellen Kontraktes bestimmt wurden, so kann

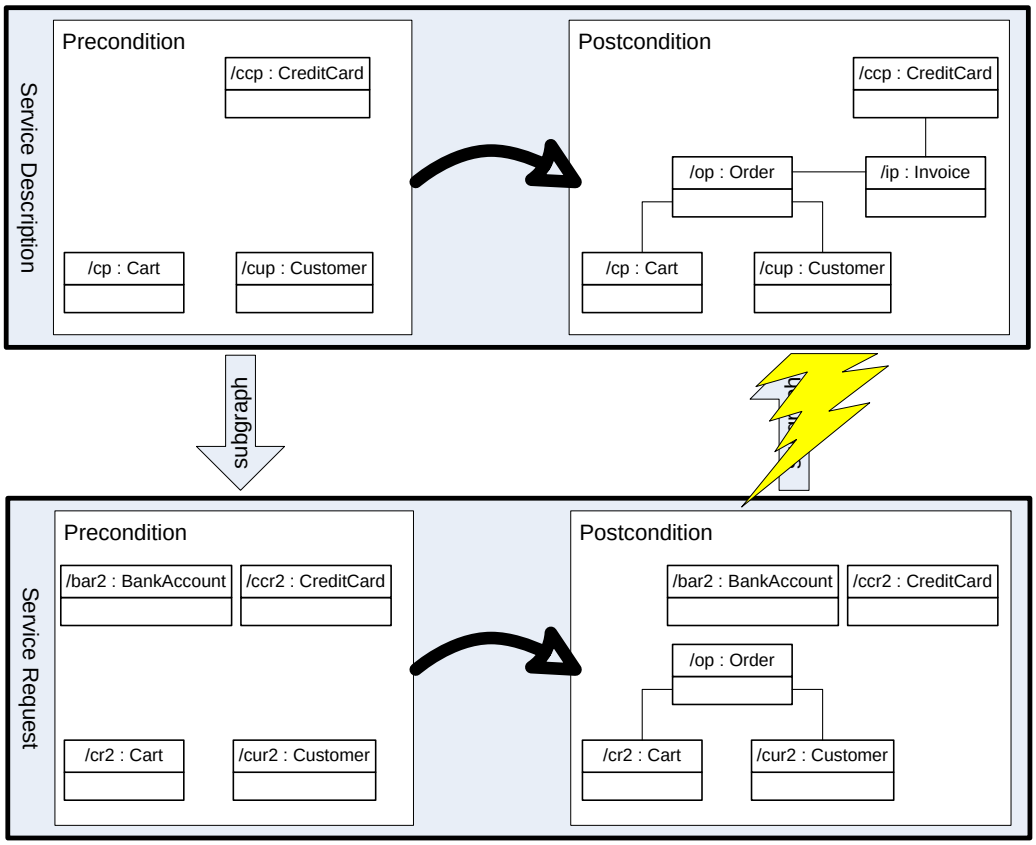


Abbildung 6.7: Subgraph-Relationen zwischen den Vor- und Nachbedingungen — Beispiel 2

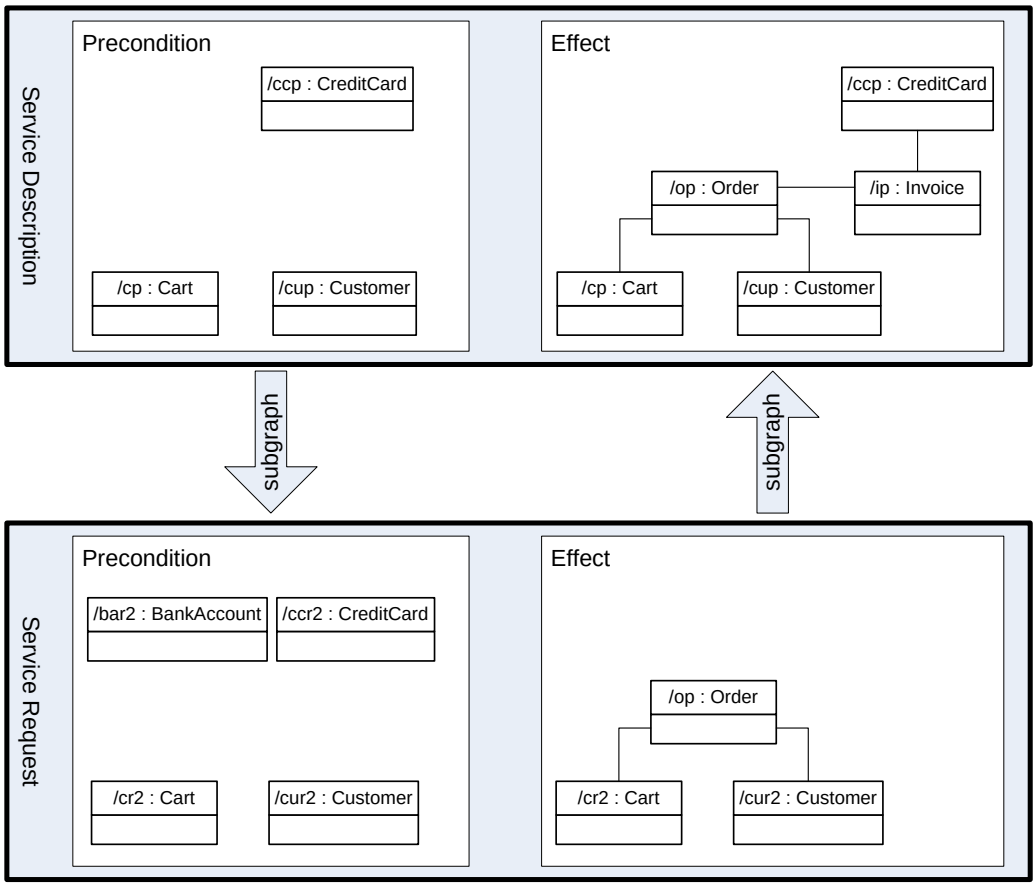


Abbildung 6.8: Subgraph-Relation zwischen Vorbedingungen und Effekten

die Berechnung eines Matchings wiederum auf einfache Subgraph-Relationen zurückgeführt werden, wie wir im Folgenden zeigen.

Die Kompositionsstrukturdiagramme in Abbildung 6.8 zeigen die Vorbedingungen und die Effekte der visuellen Kontrakte aus Abbildung 6.3 und Abbildung 6.6. Das Matching ist erfolgreich, wenn die in Abbildung 6.6 gezeigten Subgraph-Relationen gültig sind. Damit kann ein Service Requestor einen Service aufrufen, wenn die Vorbedingung des Service Requestors stärker oder identisch zur Vorbedingung des Service Providers ist und wenn die Effekte des Service Requestors schwächer oder identisch zu den Effekten des Service Providers sind. Das bedeutet, die Vorbedingung des Service Providers ist erfüllt, wenn die Vorbedingung des Service Requestors erfüllt ist, und die Effekte des Service Requestors sind erfüllt, wenn die Effekte des Service Providers erfüllt sind. Damit verhalten sich aus der Sicht des Requestors die Vorbedingung des Providers kontravariant zur Vorbedingung des Requestors

und die Effekte des Providers kovariant zu den Effekten des Requestors.

Das hier vorgestellte Matching-Konzept ist konform zur Definition der Semantik der visuellen Kontrakte. Ein visueller Kontrakt spezifiziert, welche Transformationen mindestens ausgeführt werden müssen (vgl. Abschnitt 3.4). Mit der Suchanfrage eines Service Requestors wird also spezifiziert, welche Transformationen ein gesuchter Service mindestens ausführen soll. Mit den oben dargestellten Subgraph-Relationen stellen wir sicher, dass die Transformation, die mit dem visuellen Kontrakt eines Service Providers beschrieben wird, mindestens die vom Service Requestor geforderten Änderungen eines „abstrakten Systemzustandes“ bewirkt, der Service Provider kann jedoch zusätzliche Transformationen durchführen.

6.3.2 Operationale Interpretation des Matchings

Das Matching von Spezifikationen beziehungsweise visuellen Kontrakten bietet uns die Möglichkeit, statische Aussagen über die Interoperabilität von Service Providern und Service Requestoren zu machen. Die Annahme dabei ist, dass die für die Berechnung des Matchings verwendeten visuellen Kontrakte tatsächlich die Anforderungen und Zusicherungen (das Verhalten) der Implementierung eines entsprechenden Service auf der Provider-Seite und Clients auf der Requestor-Seite wiedergeben. Abbildung 6.9 zeigt eine operationale Interpretation des in diesem Kapitel eingeführten Matching-Konzeptes. Die Abbildung zeigt den Aufruf eines Service durch einen Requestor sowie die Vorbedingungen und Effekte eines Requestor- bzw. Provider-Kontraktes.

Der Aufruf eines Service durch einen Service Requestor läuft dann wie folgt ab:

1. Bei einem Aufruf eines Service garantiert der Service Requestor, dass die Vorbedingung seines Requestor-Kontraktes erfüllt ist.
2. Bei einem positiven Matching kann der Service Provider voraussetzen, dass die Vorbedingung seines Provider-Kontraktes erfüllt ist, da diese eine Teilmenge der Vorbedingung des Requestor-Kontraktes ist. Somit kann die aufgerufene Operation des Service Providers ausgeführt werden.
3. Nach der Ausführung der Operation garantiert der Service Provider, dass die Effekte seines Provider-Kontraktes erfüllt sind.
4. Der Service Requestor kann annehmen, dass die Effekte des Requestor-Kontraktes erfüllt sind, da sie eine Teilmenge der Effekte des Provider-

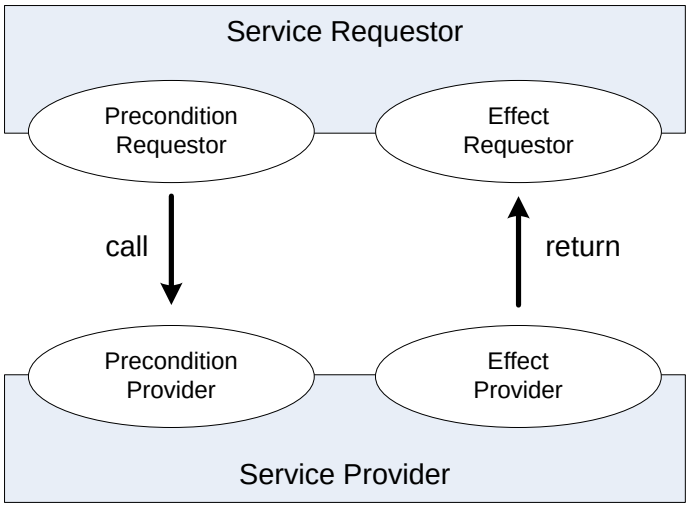


Abbildung 6.9: Operationale Interpretation des Matchings

Kontraktes sind.

Somit wird es theoretisch möglich, dass beim Aufruf eines Service durch einen Service Requestor die Überprüfung der Schritte 3 und 4 nicht mehr notwendig ist, da diese durch ein positives Matching sichergestellt werden. Dies spart nicht nur Zeit, sondern auch den Aufwand für das Abfangen von Fehlern, falls diese Überprüfungen nicht erfolgreich sein sollten.

In einem praktischen Kontext ist dennoch sicher zu stellen, dass die Anwendungen des Service Providers und des Service Requestors korrekt zusammen arbeiten. Hierzu muss zum einen sichergestellt werden, dass die in 1 und 3 dargestellten Garantien tatsächlich eingehalten werden. Zum anderen ist sicher zu stellen, dass die Annahmen aus 2 und 4 tatsächlich ausreichen, um eine korrekte Funktion der Anwendungen des Service Providers und Service Requestors zu ermöglichen. Dies ist jedoch kein Problem des in diesem Kapitel vorgestellten Matching-Konzeptes. Es handelt sicher hierbei um ein (vertikales) Konsistenzproblem zwischen den Spezifikationen (visuellen Kontrakten) des Service Providers bzw. Requestors und ihren Implementierungen. Es ist zu überprüfen, ob sich die Implementierungen des Service Providers und Requestors gemäß ihrer Spezifikation verhalten.

Auf der Provider-Seite kann mit dem in Kapitel 4 eingeführten Konzept des Model-Driven Monitoring sichergestellt werden, dass die Implementierung eines Service sich korrekt gegenüber seiner Spezifikation mit visuellen Kontrakten verhält. Die Verwendung von standardisierten Ontologien bei der Entwicklung eines Service (vgl. Abschnitt 2.6) stellt zudem sicher, dass die

für das Matching verwendeten visuellen Kontrakte gegenüber einer Ontologie getypt sind. Auf der Provider-Seite wird aufgrund des Model-Driven Monitoring somit sichergestellt, dass die Vor- und Nachbedingungen eines visuellen Kontraktes (vor und nach der Ausführung einer Operation) erfüllt sind. Ansonsten wird der Service mit einer Exception abgebrochen.

Auf der Requestor-Seite gestaltet sich das Vorgehen ein wenig anders. Nach der Auswahl eines konkreten Service wird auf der Requestor-Seite auf Basis der Servicebeschreibung, insbesondere auf Basis der syntaktischen Beschreibungen mit z.B. einem WSDL-Dokument, ein Proxy generiert. Die von diesem Proxy zur Verfügung gestellten Operationen stellen die Services eines Service Providers dar. Die in dem Proxy vorhandenen Operationen können von einem Client aufgerufen werden, um einen Service des Service Providers aufzurufen. Damit wird auf Seiten des Service Providers mit Hilfe des Proxies ein korrekter Aufruf eines Service sichergestellt. Die Vor- und Nachbedingungen des Requestor-Kontraktes müssen nicht mehr explizit berücksichtigt werden, sie werden aufgrund des zuvor berechneten Matchings und der Überwachung der Vor- und Nachbedingungen auf der Provider-Seite implizit berücksichtigt. Explizit werden die Requestor-Kontrakte somit nur für die Suche nach einem geeigneten Service benötigt.

Betrachtet man das in diesem Kapitel eingeführte Matching-Konzept aus der Sicht des Testens einer Anwendung, so kann ein Matching der Spezifikationen den Aufwand für das Integrationstesten reduzieren oder die Integrationstests ersetzen. Jedoch kann das Matching der Spezifikationen nicht die Korrektheit der einzelnen Anwendung — auf Provider- und Requestor-Seite — sicherstellen. Hier muss z.B. mittels Unit-Tests überprüft werden, ob die Anwendungen sich gemäß ihrer Spezifikation verhalten [HL04] (vgl. auch Abschnitt 4.4).

6.4 Matching — formale Definition

Im Folgenden beschreiben wir eine mengentheoretische Formalisierung unseres Matching-Konzeptes, um die Servicebeschreibung eines Service Providers mit der Suchanfrage eines Service Requestors vergleichen zu können. Zuvor fassen wir jedoch die Bedeutung eines Provider- und Requestor-Kontraktes noch einmal zusammen und legen hierbei insbesondere die Bedeutung der Effekte mengentehoretisch fest.

Ein Provider-Kontrakt $p : L_p \rightarrow R_p$ beschreibt die Semantik eines Service. Für den erfolgreichen Aufruf eines Service werden alle Elemente und Konnek-

toren aus der Vorbedingung L_p benötigt. Die Effekte, die der Service Provider garantiert, ergeben sich aus dem Vergleich der linken und rechten Seite eines visuellen Kontraktes:

- Die Erzeugung von $R_p \setminus L_p$,
- das Löschen von $L_p \setminus R_p$, und
- die Beibehaltung von $L_p \cap R_p$.

Das bedeutet für den Provider-Kontrakt aus Abbildung 6.3, dass die Elemente `ccp:CreditCard` und `cp:Cart`, `cup:Customer` für den Aufruf des mit dem visuellen Kontrakt beschriebenen Service benötigt werden. Es wird garantiert, dass diese Elemente bei der Ausführung des Service beibehalten werden ($L_p \cap R_p = \{ccp, cp, cup\}$), und dass weiterhin die Elemente `op:Order`, `ip:Invoice` einschließlich vier Links (gehören zur Menge $R_p \setminus L_p$) erzeugt werden.

Ein Requestor-Kontrakt $r : L_r \rightarrow R_r$ ist ein wenig anders zu interpretieren. Ein Requestor-Kontrakt beschreibt die Annahmen, die ein Service Requestor über einen gesuchten Service macht. Die linke Seite des Requestor-Kontraktes L_r spezifiziert die Elemente und Links, dessen Existenz vom Service Requestor beim Aufruf eines Service garantiert werden. In dem Requestor-Kontrakt aus Abbildung 6.4 handelt es sich dabei z.B. um die Elemente `ccr1:CreditCard`, `cr1:Cart`, `cur1:Customer`.

Die Effekte, welche ein Service Requestor durch den Aufruf eines Service erreichen möchte, können beschrieben werden durch den Vergleich der linken und rechten Seite des Requestor-Kontraktes r:

- Elemente und Links in $L_r \setminus R_r$ sollen gelöscht werden,
- Elemente und Links in $R_r \setminus L_r$ sollen neu erzeugt werden,
- während Elemente und Links in der Schnittstelle $L_r \cap R_r$ beibehalten werden sollen, *wenn sie beim Aufruf des Service verwendet werden.*

In dem Requestor-Kontrakt aus Abbildung 6.4 sind keine Löschungen spezifiziert, somit sind alle Elemente der linken Seite gleichzeitig auch Teil der Schnittmenge $L_r \cap R_r$ und sollen somit beibehalten werden. In dem visuellen Kontrakt wird weiterhin die Erzeugung eines Elementes *or1:Order* mit zwei Konnektoren (gehören zur Menge $L_r \setminus R_r$) gefordert.

Das Matching von Provider- und Requestor-Kontrakten vergleicht nun die Garantien, die von beiden Seiten gegeben werden mit den Annahmen der jeweils anderen Seite. Die Vorbedingung des Service Requestors muss alle

Elemente und Links der Vorbedingung des Service Providers enthalten, und die Effekte des Service Providers müssen alle Effekte des Service Requestors enthalten. Hierbei ist zu beachten, dass es sich bei den Namen der Modellelemente in den visuellen Kontrakten um Rollennamen handelt. Wenn die Rollennamen konsistent umbenannt werden, so ändert sich die Bedeutung eines visuellen Kontraktes nicht. In den visuellen Kontrakten aus unseren Beispielen kann man durch einfaches Hinsehen feststellen, dass die Bedingungen für ein erfolgreiches Matching erfüllt sind (vgl. auch Abschnitt 6.3.1). Formal ergibt sich aus diesen einfachen Bedingungen folgende Definition für das Matching von einem Provider- und einem Requestor-Kontrakt:

Definition 6.1 (Matching [HHL04, HHL05]) *Ein Provider-Kontrakt* $p : L_p \rightarrow R_p$ matcht *auf einen Requestor-Kontrakt* $r : L_r \rightarrow R_r$*, wenn eine strukturkompatible, partielle, eins-zu-eins Korrespondenz* $\mathbf{h} \subseteq L_p \cup R_p \times L_r \cup R_r$ *existiert, so dass gilt:*

1. $\forall x \in L_p \; \exists y \in L_r : x \; \mathbf{h} \; y$*, d.h., jedes Element aus* L_p *muss eine Entsprechung in* L_r *haben;*
2. $\forall y \in L_r \setminus R_r \; \exists x \in L_p \setminus R_p : x \; \mathbf{h} \; y$*, d.h. alle Elemente und Links in* L_r*, die von* r *gelöscht werden sollen, müssen über* h *auf ein Element in* R_p *abgebildet werden, welches tatsächlich von* p *gelöscht wird;*
3. $\forall y \in L_r \cap R_r \; \exists x \in L_p \cup R_p : x \; \mathbf{h} \; y \implies x \in L_p \cap R_p$*, d.h. jedes Element aus* L_p*, welches über* h *mit einem Element verbunden ist, welches laut der Spezifikation von* r *beibehalten werden soll, muss tatsächlich von* p *beibehalten werden.*
4. $\forall y \in R_r \setminus L_r \; \exists x \in R_p \setminus L_p : x \; \mathbf{h} \; y$*, d.h. jedes Element aus* R_r*, welches laut* r *erzeugt werden soll, muss via* h *zu einem Element gehören, welches von* p *tatsächlich erzeugt wird.*

Bei der strukturkompatiblen, partiellen, eins-zu-eins Korrespondenz handelt es sich um eine zweistellige „Umbenennungs"-Relation über den Mengen $L_p \cup R_p$ und $L_r \cup R_r$. Wir schreiben $x \; \mathbf{h} \; y$, wenn $x \in L_p \cup R_p$ in Relation zu $y \in L_r \cup R_r$ steht.

Unter einer *strukturkompatiblen* Relation verstehen wir eine Relation bei der Elemente, die in Relation zueinander stehen, einen kompatiblen Typ besitzen und Links, die in Relation zueinander stehen, besitzen Quell- und Zielelemente, die einen kompatiblen Typ haben.

Partiell bedeutet in diesem Zusammenhang, dass nicht alle Elemente und Links aus $L_p \cup R_p$ und $L_r \cup R_r$ Teil der Relation $\mathbf{h}$ sein müssen. Die Relation

h ist also weder linkstotal oder rechtstotal (surjektiv).

Unter *eins-zu-eins* verstehen wir, dass jedes Element oder jeder Link höchstens einmal in der Relation vorkommen darf. Die Relation **h** ist also linkseindeutig (injektiv) und rechtseindeutig.

Die *Umbenennung* ist erforderlich, da die Elemente in den Provider- und Requestor-Kontrakten unterschiedliche Rollennamen (wie auch in unseren Beispielen) haben können, die nicht berücksichtigt werden müssen, solange die gefundenen Strukturen identisch sind.

Im Folgenden wenden wir diese Definition des Matchings auf die in diesem Kapitel vorgestellten visuellen Kontrakte zur Beschreibung von Services und Suchanfragen an.

Abbildung 6.3 zeigt einen Provider-Kontrakt $p : L_p \rightarrow R_p$, Abbildung 6.4 zeigt einen Requestor-Kontrakt $r : L_r \rightarrow R_r$. Gemäß der oben gegebenen formalen Definition muss eine Korrespondenz $\mathbf{h}_1$ die Paare $\{(ccp, ccr1), (cp, cr1), (cup, cur1)\} \subseteq \mathbf{h}_1$ enthalten, um Anforderung 1 zu erfüllen. Die Elemente eines Paares sind in diesem Fall von demselben Typ. Die Anforderung 2 erfordert eine Inklusion der gelöschten Elemente von r in die von p, was in diesem Fall trivial ist, da die Mengen für die beiden visuellen Kontrakte leer sind. Aus demselben Grund sind die Schnittmengen $L_r \cap R_r = L_r$ und $L_p \cap R_p = L_p$ isomorph über $\mathbf{h}_1$, womit Anforderung 3 erfüllt ist. Abschließend erfordert Anforderung 4, dass alle Elemente, die in r erzeugt werden (d.h. das Element *or1:Order* mit den beiden anonymen Links), in Korrespondenz zu Elementen sind, welche in p erzeugt werden. Somit muss die Relation $\mathbf{h}_1$ um das Paar $(op, or1)$ sowie die Links ausgehend von $or1$ und den entsprechenden ausgehenden von op erweitert werden. Damit haben wir eine Korrespondenz zwischen den Mengen $L_p \cup R_p$ und $L_r \cup R_r$, welche die obigen vier Bedingungen erfüllt, gefunden. Der Provider-Kontrakt aus Abbildung 6.3 matcht also auf den Requestor-Kontrakt aus Abbildung 6.4.

Die Situation ist ein wenig komplizierter, wenn man den Provider-Kontrakt $p : L_p \rightarrow R_p$ aus Abbildung 6.3 mit dem Requestor-Kontrakt $r : L_r \rightarrow R_r$ aus Abbildung 6.6 vergleicht. Die Relation $\mathbf{h}_2 = \{(ccp, ccr2), (cp, cr2), (cup, cur2)\}$ erfüllt Anforderung 1. Bei dieser Relation ist zu beachten, dass das Element `btr2:BankAccount` aus der Menge L_r nicht Teil der Relation ist, da diese Information nicht von dem Provider benötigt wird. In dem Requestor-Kontrakt ist dieses Element jedoch enthalten, um eine größere Flexibilität bei der Suche nach einem geeigneten Service zu erreichen, d.h. um mehr Matches zu erhalten. Die Mengen der gelöschten und erzeugten Elemente des Requestor-Kontraktes aus Abbildung 6.6 ist identisch zu den Mengen

der gelöschten und erzeugten Elemente des Requestor-Kontraktes aus Abbildung 6.4. Jedoch ändert sich in dem Requestor-Kontrakt aus Abbildung 6.6 die Menge der Elemente und Links, die beibehalten beziehungsweise nicht geändert werden sollen, wenn sie einem Service übergeben werden. Die Elemente aus der Schnittmenge $L_r \cap R_r = \{bar2, ccr2, cr2, cur2\}$ sollen beibehalten werden. Die Menge der Elemente aus p, welche über $\mathbf{h}_2$ mit diesen in Verbindung stehen ist $\{ccp, cp, cup\}$. Die Elemente aus dieser Menge werden gemäß des Provider-Kontraktes beim Aufruf des Service beibehalten. Damit ist Anforderung 3 erfüllt.

An dieser Stelle wird auch noch einmal der Unterschied zwischen der Anforderung 3 und den Anforderungen 2 und 4 aus Definition 6.1 deutlich. Bei den Anforderungen 2 und 4 *müssen die Elemente eines bestimmten Typs aus r in Relation* zu einem Element aus p stehen. Dagegen müssen bei der Anforderung 3 *Elemente, die in Relation zu einander stehen*, denselben Typ besitzen. Somit sind die Elemente aus $L_r \cap R_r$ (Elemente die gemäß des Requestor-Kontraktes beim Aufruf eines Service beibehalten werden sollen) optional, d.h. sie müssen nicht vom Provider verwendet werden. Wenn sie jedoch verwendet werden, so dürfen sie nicht gelöscht werden.

Im Gegensatz zu der intuitiven Definition des Matchings aus Abschnitt 6.3.1 stellt die hier vorgestellte formale Definition explizit einen Zusammenhang zwischen den Vor- und Nachbedingungen her. Bei der intuitiven Definition des Matchings wird bereits ein erfolgreicher Match gefunden, wenn die Vorbedingung des Requestor-Kontraktes stärker oder identisch zur Vorbedingung des Provider-Kontraktes ist und wenn die Effekte des Requestor-Kontraktes schwächer oder identisch zu den Effekten des Provider-Kontraktes sind. Der Zusammenhang zwischen den Vor- und Nachbedingungen eines visuellen Kontraktes wird bei der intuitiven Definition des Matchings nicht berücksichtigt. Die formale Definition des Matching zieht aufgrund der Relation $\mathbf{h}$, die über der Menge $L_p \cup R_p \times L_r \cup R_r$ definiert ist, die Beziehungen zwischen den Vor- und Nachbedingungen explizit mit ein und stellt damit einen genaueren Matchingbegriff dar.

Die höhere Genauigkeit der formalen Definition verdeutlichen wir kurz an dem Requestor-Kontrakt aus Abbildung 6.10. Der Requestor liefert als Eingabeinformationen Daten über einen existierenden `Cart`, eine `CreditCart` und einen `Customer`. Im Gegensatz zu den Requestor-Kontrakten aus den Abbildungen 6.4 und 6.6 verlangt der Requestor-Kontrakt jedoch die Erzeugung eines neuen Kunden (`cur3b:Customer`), welchem ein Auftrag hinzugefügt wird. Der Kunde (`cur3a:Customer`) aus der Vorbedingung wird in der Nachbedingung nicht verändert.

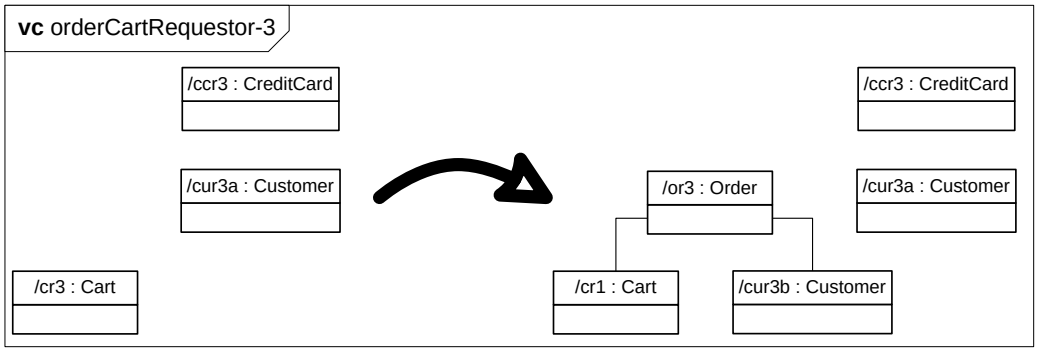


Abbildung 6.10: Requestor-Kontrakt — Beispiel 3

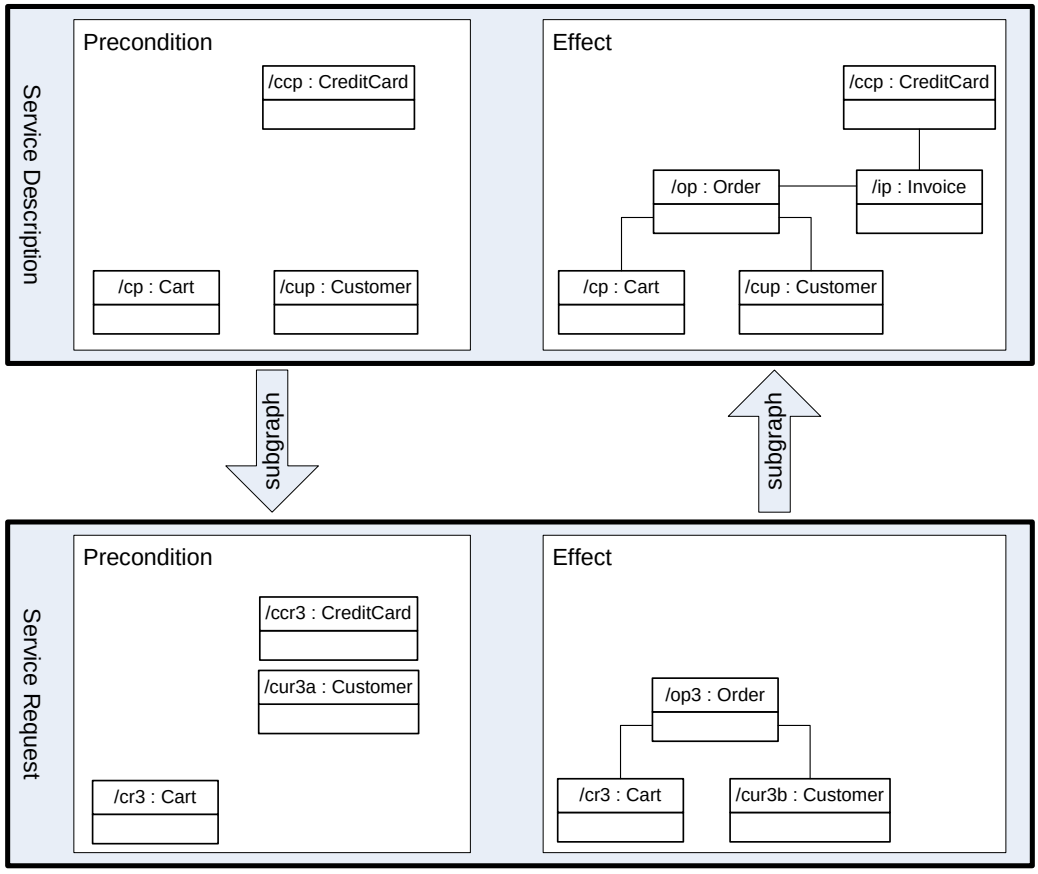


Abbildung 6.11: Subgraph-Relation, Faulse Positive bei intuitivem Matching

Das intuitive Matching dieses Requestor-Kontraktes gegenüber dem Provider-Kontrakt aus Abbildung 6.3 ist in Abbildung 6.11 abgebildet. Die Vorbedingung des Service Providers ist ein Subgraph der Vorbedingung des Service Requestors. Die Effekte des Service Requestors sind ein Subgraph der Effekte des Service Providers. Das in Abschnitt 6.3.1 definierte intuitive Matching liefert in diesem Fall ein positives Matching-Ergebnis. Die formale Definition des Matchings liefert jedoch ein negatives Matching-Ergebnis. Der Grund ist, dass bei dem intuitiven Matching die Vorbedingungen und Effekte unabhängig voneinander verglichen werden, während bei der formalen Definition aufgrund der Korrespondenz h die Beziehung zwischen den Vor- und Nachbedingungen berücksichtigt werden. Für die Kontrakte aus Abbildung 6.3 und Abbildung 6.10 findet sich kein h, dass alle Bedingungen der formalen Definition erfüllt.

6.5 Fazit

In diesem Kapitel haben wir einen neuen modellbasierten Ansatz zur Spezifikation der Semantik von Services und Suchanfragen beschrieben, um eine modellbasierte Suche eines Service Requestors nach geeigneten Services zu ermöglichen. Der Ansatz benutzt eine formale und intuitive visuelle Repräsentation. Wir erlauben ein flexibles Matching indem wir Ontologien mit visuellen Kontrakten kombinieren. Eine Ontologie wird mit einem Klassendiagramm dargestellt und definiert eine Menge von Konzepten und deren Beziehungen. In visuellen Kontrakten werden diese Konzepte verwendet, um die Semantik eines Service oder einer Suchanfrage zu beschreiben. Für das vorgestellte Matching-Konzept haben wir eine mengentheoretische Definition angegeben.

In einer weiteren Doktorarbeit [Che06] in der Arbeitsgruppe von Prof. Dr. Engels ist parallel zu dieser mengentheoretischen Definition des Matchings eine Formalisierung des Matching-Konzeptes auf der Basis von Graphtransformationsregeln beschrieben worden. Hierbei werden Graphtransitionen (ein Formalismus auf dem auch unsere visuellen Kontrakte beruhen, vgl. Kapitel 3.4) verwendet, die durch den Double-Pullback-Ansatz definiert sind (vgl. auch [HCL04, HC05]).

Eine Charakteristik unseres Matching-Konzeptes ist, dass die visuellen Kontrakte auf die Beschreibung struktureller Änderungen, d.h. das Löschen oder Erzeugen von Elementen oder Konnektoren, beschränkt sind. Eine Erweiterung des Matching-Konzeptes auf visuelle Kontrakte mit Attributen, nega-

tiven Anwendungsbedingungen oder logischen Ausdrücken würde zwar die Ausdrucksmächtigkeit einer semantischen Servicebeschreibung oder Suchanfrage erhöhen, jedoch auch ein effizientes Matching verhindern.

Die eingeschränkte Ausdrucksmächtigkeit unseres Matching-Konzeptes stellt aus unserer Sicht jedoch kein Problem dar, da das Matching, welches automatisiert von einem Discovery Service bestimmt wird, nicht der letzte Schritt eines Service Requestors bei der Suche und Bestimmung eines geeigneten Service ist. Mit diesem einfachen Matching sollen lediglich *potentiell* geeignete Services gefunden werden. Daher ist auch die unvollständige Beschreibung einer Servicebeschreibung oder einer Suchanfrage aufgrund der verwendeten losen Semantik unserer visuellen Kontrakte keine Einschränkung. Nach der Bestimmung des Matchings sind zusätzliche Schritte seitens des Service Requestors notwendig, um aus der Menge der Ergebnisse des Discovery Service den am besten geeigneten Service auszuwählen. Dabei muss nur noch eine kleinere Menge von Services untersucht werden. Hierzu können feingranulare, automatisierte Vorgehensweisen oder eine manuelle Auswahl angewendet werden.

Unterstützt wird diese Aussage durch die in Kooperation mit der sd&m AG durchgeführte Evaluation zum Einsatz unserer visuellen Kontrakte in einer serviceorientierten Architektur [EGJ+06, LRE+06]. Das Ergebnis dieser Evaluation ist, dass auf einer fachlichen Ebene (entspricht der Ebene der Ontologien in dem in dieser Arbeit beschriebenen Ansatz) weniger ausdrucksstarke visuelle Kontrakte zur Beschreibung von Services ausreichend sind. Auf der Ebene der Implementierung (vergleichbar mit der in Kapitel 3 und 4 beschriebenen Detaillierung) werden zur Beschreibung konkreter Operationen jedoch detailliertere visuelle Kontrakte benötigt.

Andere mögliche Erweiterungen für unser Matching-Konzept sollen im Rahmen einer Kooperation mit der Arbeitsgruppe von Prof. Dr. Odej Kao weiter verfolgt werden. In der Arbeitsgruppe von Prof. Dr. Kao ist ein Verfahren zur Auffindung von Grid-Ressourcen, welches auf Description Logics (DL) basiert, entwickelt worden [HHK04]. Das Verfahren zeichnet sich insbesondere dadurch aus, dass einer Ontologie zur Laufzeit neue Konzepte hinzugefügt werden können. Mit einem DL-System kann zusätzlich überprüft werden, ob ein Konzept ein anderes umfasst, so dass auch mit neuen Konzepten beschriebene Ressourcen mit bestehenden Suchanfragen gefunden werden können. Ein erster Vergleich [Fre05] unseres Ansatzes mit dem aus der Arbeitsgruppe von Prof. Dr. Kao hat gezeigt, dass unserer Ansatz insbesondere von den erweiterbaren Ontologien profitieren kann, während der Ansatz aus [HHK04] von einer geeigneten, intuitiven Darstellung ähnlich der visuellen Kontrakte

profitieren könnte.

Kapitel 7

Matching mit Semantic-Web-Technologien

In Kapitel 6 haben wir Ontologien und visuelle Kontrakte zur Spezifikation der Semantik von Services und Suchanfragen verwendet, um einem Service Requestor eine modellbasierte Suche nach geeigneten Services zu ermöglichen. Für einen Vergleich von Servicebeschreibungen und Suchanfragen haben wir ein flexibles Matching-Konzept definiert.

Für den praktischen Einsatz unseres modellbasierten Matching-Konzeptes für die Suche nach Web Services, ist eine Umsetzung mit existierenden Web-Service-Technologien beziehungsweise Semantic-Web-Technologien erforderlich. Insbesondere ist die Verwendung eines weitgehend standardisierten Austauschformates zur Repräsentation der Ontologien und visuellen Kontrakte notwendig, welches gleichzeitig auch für eine automatisierte Interpretation geeignet ist, um die Berechnung eines Matching zu ermöglichen.

In diesem Kapitel führen wir als erstes den Begriff des Semantic Web ein und skizzieren die wichtigsten Semantic-Web-Sprachen. Diese Sprachen nutzen wir als Austauschformat zur Repräsentation der Ontologien und visuellen Kontrakte zur Beschreibung der Semantik eines Service und einer Suchanfrage. Weiterhin präsentieren wir einen Algorithmus, der die Semantic-Web-Repräsentationen der Ontologien und visuellen Kontrakte sowie ein existierendes Semantic Web Framework nutzt, um zwei Kontrakte gemäß unserem Matching-Konzept zu vergleichen.

7.1 Semantic Web

Der Begriff des *Semantic Web* existiert bereits seit 1998 [BL98]. Jedoch ist die Bedeutung dieses Begriffes für die weitere Entwicklung des Web erst durch den viel beachteten Artikel von Berners-Lee, Hendler und Lassila [BLHL01] im Scientific American für ein breites Publikum deutlich geworden. In diesem Artikel wird das Semantic Web als eine Erweiterung des herkömmlichen Webs, in der Informationen mit eindeutigen Bedeutungen versehen werden, definiert. Damit soll die Kommunikation zwischen Mensch und Maschine erleichtert werden:

> „The Semantic Web is an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation."

Die Weiterentwicklung basiert gemäß [BLHL01] auf den folgenden drei Stützpfeilern (vgl. auch [Beh03]):

- Geeigneten *Beschreibungssprachen zur semantischen Auszeichnung* von Webseiten oder Web Services, die für eine automatisierte Interpretation durch Software geeignet sind,
- standardisierte Begriffswelten bzw. *Ontologien* auf die sich die Produzenten und Verbraucher semantischer Auszeichnungen einigen und
- *Softwaresystemen* bzw. Softwareagenten, die für einen menschlichen Auftraggeber die semantischen Informationen im Web auswerten.

Zur Unterstützung dieser drei Stützpfeiler gibt es verschiedene Basistechnologien. Die aktiven Komponenten des semantischen Webs bilden Softwaresysteme, die Suchanfragen gemäß einer Ontologie bearbeiten und Ergebnisse bestimmen, die für den Endbenutzer wahrscheinlich von Interesse sind. Die Bedeutung allgemein akzeptierter und verbindlicher Ontologien wurde bereits in Kapitel 6.2.1 diskutiert.

Die Verwendung einer genormten Beschreibungssprache zur semantischen Auszeichnung ist wichtig, damit diese von unterschiedlichen Softwaresystemen verarbeitet werden kann. Die semantischen Auszeichnungssprachen des Semantic Web benutzen die Extensible Markup Language (XML) [BPSM$^+$04] als Trägerformat und das *Resource Description Framework (RDF)* [MM04, Bec04, KJC04, HM04] zum Beschreiben und Auffinden von Web-Inhalten.

Jedoch sind weder XML noch RDF ausreichend mächtig, um Wissensstrukturen wie Ontologien einfach und formal abgesichert zu spezifizieren [Beh03].

Daher wurden mit der *RDF Vocabulary Description Language* [BG04] verschiedene erweiterte Beschreibungssprachen entwickelt, die es erlauben innerhalb von RDF eine Ontologie zu spezifizieren. Die Beschreibungssprache *DARPA Agent Markup Language - Ontology* (*DAML-ONT*) [HM00] wurde im Jahr 2000 als Erweiterung zu RDF und XML zur Darstellung von Ontologien von der Defense Advanced Research Projects Agency (DARPA)[1] aus der Taufe gehoben. Etwa zur gleichen Zeit wurde in dem europäischen Projekt On-To-Knowledge[2] die Sprache *Ontology Inference Layer* (*OIL*) [FHvH+00, HFB+00] zur Darstellung von Ontologien entwickelt. Mit dem Standard *DAML+OIL* (*DARPA Agent Markup Language + Ontology Inference Layer*) [CvHH+01] haben sich die beiden Protagonisten einander angenähert. Der direkte Nachfolger von DAML+OIL, die *Web Ontology Language* (*OWL*) [MvH04, DSB+04, PSHH04], ist vom World Wide Web Consortium standardisiert worden.

Obwohl OWL der aktuelle Hauptvertreter zur Beschreibung von Ontologien ist, beziehen sich die folgenden Beschreibungen auf seinen Vorgänger DAML+OIL. Der Grund ist, dass die Umsetzung des in Kapitel 6 beschriebenen Matching-Konzeptes mit Semantic-Web-Technologien im Rahmen einer Diplomarbeit [Sev03] erfolgt ist. Zum Zeitpunkt der Diplomarbeit war die Verwendung der Sprache DAML+OIL sinnvoll beziehungsweise notwendig, da für OWL die notwendige Werkzeugunterstützung noch nicht vorhanden war. Der Unterschied zwischen DAML+OIL und OWL ist jedoch so gering, dass sich die im Folgenden beschriebenen Vorgehensweisen und Ergebnisse auf OWL übertragen lassen.

7.2 Semantic-Web-Sprachen

In diesem Abschnitt beschreiben wir die Grundlagen der Semantic-Web-Sprachen, die wir für die Definition unserer Austauschformate zur Repräsentation von Ontologien und visuellen Kontrakten im Web verwenden. Wir skizzieren kurz, wie XML als allgemeines Darstellungsformat für RDF-Dokumente verwendet wird und erklären insbesondere, wie RDF, RDF Schema und DAML+OIL aufeinander aufbauen.

Weiterhin wird in diesem Abschnitt eine Anfragesprache für RDF-Dokumente (RDQL) beschrieben, die wir in der Implementierung unseres Matching-Algorithmus verwenden.

[1] `http://www.darpa.mil/`
[2] `http://www.ontoknowledge.org/`

7.2.1 RDF

Das World Wide Web Consortium hat insgesamt sechs Spezifikationen zur Detaillierung des *Resource Description Framework (RDF)* veröffentlicht[3]. In diesem Abschnitt skizzieren wir die zwei grundlegenden Spezifikationen, das RDF-Datenmodell [KJC04] und seine Serialisierung nach XML [Bec04].

Das RDF-Datenmodell besteht aus Entitäten (*entities*), die durch eindeutige Bezeichner (*Unique Identifier* — UID) repräsentiert werden und binären Beziehungen (*statements*). Ein Statement verbindet ein *Subjekt* (Quell-Entität) und ein *Objekt* (Ziel-Entität) über ein Prädikat (*predicate* — teilweise auch als *property* bezeichnet). Subjekte sind alle Dinge, die durch RDF-Ausdrücke beschrieben werden. Prädikate haben in RDF — wie auch in der natürlichen Sprache — die Aufgabe das Subjekt zu erläutern. Objekte beschreiben den Wert eines Prädikats.

Weiterhin wird im RDF Datenmodell zwischen *Ressourcen* und *Literalen* unterschieden. Eine Ressource wird durch einen Uniform Resource Identifier (URI) [BL94, BLFM05], wie z.B. `http://wwwcs.upb.de/ag-engels/topic/SemanticMatching`, repräsentiert. Literale sind einfache Strings. Subjekte und Prädikate sind immer als Ressourcen ausgeprägt, ein Objekt kann entweder als Ressource oder als Literal ausgeprägt sein.

Abbildung 7.1 zeigt einen einfachen RDF-Graphen mit drei verschiedenen Statements. Gemäß den Vorgaben aus den RDF-Spezifikationen werden Ressourcen mit Ovalen und Literale mit Vierecken dargestellt. Kontaktperson für die Ressource `SemanticWeb` (der AG-Engels) ist eine weitere Ressource `person/ml`. Diese Ressource besitzt den Namen `Marc Lohmann` und die Email-Adresse `mlohmann@upb.de`. Listing 7.1 zeigt eine Serialisierung des RDF-Graphen mit der XML-basierten Syntax RDF/XML [Bec04].

7.2.2 RDF Vocabulary Description Language

In RDF-Statements können Aussagen über Ressourcen mit beliebigen Prädikaten und Objekten gemacht werden. Ebenso wie bei XML mit Hilfe von DTDs oder XML Schemas [BPM04, TBMM04, FW04] für konkrete Anwendungsbereiche spezielle Vokabulare definiert werden können, werden entsprechende Mechanismen auch für RDF benötigt, um eine standardisierte Interpretation von RDF-Dokumenten auf Basis eines konkreten Vokabulars zu ermöglichen.

[3] siehe `http://www.w3.org/RDF/`

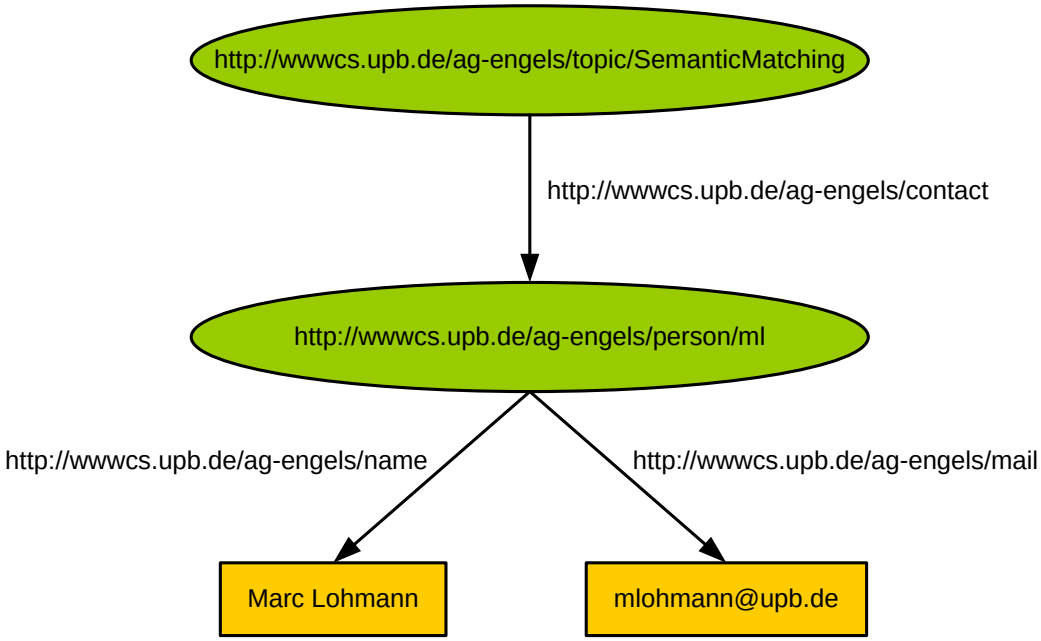


Abbildung 7.1: Beispiel für einen RDF-Graphen

```
1 <?xml version="1.0"?>
2 <rdf:RDF
3   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
4   xmlns:ag-engels="http://wwwcs.upb.de/ag-engels/">
5
6     <rdf:Description
7      rdf:about=
8    "http://wwwcs.upb.de/ag-engels/topic/SemanticMatching">
9         <ag-engels:contact rdf:resource=
10          "http://wwwcs.upb.de/ag-engels/person/ml"/>
11    </rdf:Description>
12
13    <rdf:Description
14      rdf:about=
15      "http://wwwcs.upb.de/ag-engels/person/ml">
16        <ag-engels:name>Marc Lohmann</ag-engels:name>
17            <ag-engels:mail>mlohmann@upb.de</ag-engels:mail>
18    </rdf:Description>
19 </rdf:RDF>
```

Listing 7.1: XML/RDF Serialisierung des RDF-Graphen aus Abbildung 7.1

Mit der *RDF Vocabulary Description Language* (*RDF Schema*) [BG04] können RDF-Vokabulare definiert werden. In einem RDF Schema können die in einem RDF-Statement für einen bestimmten Anwendungsbereich erlaubten Prädikate und Objekte definiert werden, um Klassen von möglichen Ressourcen zu definieren. Im Folgenden geben wir einen Überblick über die wichtigsten RDF-Schema-Konstrukte.

Klassen von Ressourcen können mit folgenden Ausdrücken festgelegt werden:

Class: Legt ein abstraktes Objekt fest und dient in Verbindung mit dem RDF-Statement `rdf:type` zur Erzeugung von Instanzen.

Resource: Jede Entität in einem RDF-Modell ist eine Instanz dieser besonderen Klasse.

Property: Dies ist die Basisklasse zur Definition von Eigenschaften und eine Unterklasse von `Resource`.

Literal: Klasse für Literalwerte, also Zeichenketten etc.

Eigenschaften können mit folgenden Ausdrücken festgelegt werden:

subClassOf: Eine transitive Eigenschaft zur Festlegung von Vererbungshierarchien von Klassen.

subPropertyOf: Eine transitive Eigenschaft zur Festlegung von Vererbungshierarchien von Eigenschaften.

domain: Legt den Anwendungsbereich einer Eigenschaft in Bezug auf eine Klasse fest.

range: Legt den Wertebereich einer Eigenschaft fest.

7.2.3 DAML+OIL

Bisher wurden die zwei Semantic-Web-Sprachen RDF und RDF Schema kurz vorgestellt. RDF ist ein Datenmodell zur Beschreibung von Objekten (Ressourcen) und Beziehungen zwischen diesen Objekten. RDF bietet eine einfache Semantik zur Interpretation des Datenmodells, welches mit XML serialisiert werden kann. Ein RDF Schema ermöglicht die Definition eines RDF-Vokabulars für bestimmte Anwendungsbereiche.

Weder RDF noch RDF Schema sind jedoch ausreichend mächtig genug, um Wissensstrukturen wie Ontologien gut darstellen zu können. RDF Schemas

bieten zwar hilfreiche Konstrukte zur Definition von Wissensstrukturen, jedoch fehlen zum Beispiel Möglichkeiten zur Modellierung von Assoziationen oder Kardinalitäten.

DAML+OIL [CvHH+01] erweitert RDF und RDF Schema um weitere Sprachkonstrukte, um Ontologien wie wir sie in Kapitel 6.2.1 vorgestellt haben gut darstellen zu können. Im Gegensatz zu RDF Schema bietet DAML+OIL zum Beispiel Sprachkonstrukte zur Modellierung von Assoziationen oder Kardinalitäten. Auf die Details der DAML+OIL-Syntax gehen wir hier nicht weiter ein. In Anhang C.1 ist eine DAML+OIL-Ontologie abgebildet.

7.2.4 RDQL

RDF und DAML+OIL sind flexible und erweiterbare Formate zur Erstellung semantischer Informationen. Zur Realisierung des in Kapitel 6 eingeführten Matching-Konzeptes, ist es aber auch notwendig, Informationen aus einem RDF-Graphen zu extrahieren bzw. nach bestimmten Strukturen in einem RDF-Graphen zu suchen.

Zur Formulierung einer Suchanfrage kann die *RDF Data Query Language* (*RDQL*) [Sea04][4] verwendet werden. Verschiedene Implementierungen, wie zum Beispiel das Jena Semantic Web Framework, unterstützen RDQL und ermöglichen eine automatische Auswertung von RDQL-Suchanfragen für einen RDF-Graphen. Im Folgenden gehen wir kurz auf die Struktur von RDQL-Suchanfragen ein. Hierbei soll insbesondere deutlich werden, dass die in einer RDQL-Suchanfrage beschriebene, gesuchte Struktur sich nicht deutlich von einem RDF-Dokument unterscheidet.

Wie bereits oben beschrieben, besteht ein RDF-Graph aus einer Menge von Tripeln. Jedes Tripel besteht aus einem Subjekt, Prädikat und Objekt. Listing 7.2 zeigt eine tripelbasierte Darstellung des RDF-Graphen aus Abbildung 7.1. Insgesamt enthält das Listing drei Tripel (Zeilen 1-3, 4-6 und 7-9).

Die RDQL-Anfragesprache basiert auf dem Matching von Graphpattern, die mit Tripeln definiert werden. Hierzu können in einer RDQL-Suchanfrage Subjekt, Prädikat oder Objekt eines Tripels mit Variablen ersetzt werden. Die

[4]Die RDQL-Spezifikation ist von HP beim W3C als Beitrag zur Standardisierung einer RDF-basierten Anfragesprache eingereicht worden. Aktuell wird der Nachfolger von RDQL, die *SPARQL Query Language for RDF* [PS06], vom W3C standardisiert. Die in der Diplomarbeit [Sev03] entwickelte Implementierung zur Umsetzung unseres Matching-Konzeptes basiert noch auf RDQL. Aktuelle Jena-Versionen unterstützen jedoch bereits SPARQL.

```
1 <http://wwwcs.upb.de/ag-engels/topic/SemanticMatching>
2         <http://wwwcs.upb.de/ag-engels/contact>
3         <http://wwwcs.upb.de/ag-engels/person/ml>
4 <http://wwwcs.upb.de/ag-engels/person/ml>
5         <http://wwwcs.upb.de/ag-engels/name>
6         "Marc Lohmann"
7 <http://wwwcs.upb.de/ag-engels/person/ml>
8         <http://wwwcs.upb.de/ag-engels/mail>
9         "mlohmann@upb.de"
```

Listing 7.2: RDF-Tripel des RDF-Graphen aus Abbildung 7.1

Kombination mehrerer Tripel in einer RDQL-Suchanfrage erlaubt die Spezifikation eines Graphpatterns, das in einem gegebenen RDF-Graphen gesucht werden soll.

Eine RDQL-Suchanfrage bettet RDF-Graphen in SQL-ähnliche Konstrukte ein. Der Aufbau einer RDQL-Anfrage ist der folgende:

SELECT: Mit der SELECT-Klausel werden die Variablen angegeben, deren konkrete Werte bei einer erfolgreichen Suche ausgegeben werden sollen.

FROM: Mit der FROM-Klausel wird das zu durchsuchende RDF-Modell bestimmt (URI, Pfad im Dateisystem etc.).

WHERE: Innerhalb einer WHERE-Klausel werden die zu suchenden Tripel spezifiziert. In den Tripeln können Subjekt, Prädikat oder Objekt durch Variablen ersetzt werden.

AND: Die AND-Klausel enthält logische Ausdrücke, um die möglichen Inhalte der Variablen weiter einzuschränken.

USING: In der USING-Klausel können Abkürzungen spezifiziert werden. Diese Abkürzungen können in der Anfrage verwendet werden, um die Komplexität einer Anfrage zu reduzieren.

Im Folgenden wird die Funktionsweise von RDQL-Suchanfragen am Beispiel der Anfrage aus Listing 7.3 erläutert. In der RDQL-Suchanfrage in Listing 7.3 werden zwei Tripel (Zeile 4-5, Zeile 6) zu einem gesuchten Graphpattern kombiniert. Die Kombination der zwei Tripel erfolgt über die Variable `?contact`. Wird diese RDQL-Suchanfrage auf den RDF-Graphen aus Abbildung 7.1 (für eine Tripel-basierte Darstellung siehe Listing 7.2) angewendet, so wird als Ergebnis der String „Lohmann“ zurückgegeben (siehe Variable `?name` in Zeile 1 und 6).

```
1 SELECT ?name
2 WHERE
3
4  (<http://wwwcs.upb.de/ag-engels/topic/SemanticMatching>,
5      <http://wwwcs.upb.de/ag-engels/contact>, ?contact)
6  (?contact, <http://wwwcs.upb.de/ag-engels/name>, ?name)
```

Listing 7.3: Einfache RDQL-Anfrage

7.3 Repräsentation von Ontologien und Kontrakten mit Semantic-Web-Sprachen

In Kapitel 6.2 ist ein visueller, modellbasierter Ansatz zur semantischen Beschreibung von Services und Suchanfragen mit Ontologien und visuellen Kontrakten eingeführt worden. Für den praktischen Einsatz in einem Web Service Kontext ist die Definition eines Austauschformates notwendig, welches auf existierenden Standards beruht.

Ontologien können mit DAML+OIL-Dokumenten dargestellt werden. Für die Repräsentation von visuellen Kontrakten skizzieren wir die Struktur eines möglichen Austauschformates.

7.3.1 Repräsentation von Ontologien

In dieser Arbeit Ansatz werden UML-Klassendiagramme zur Repräsentation von Ontologien verwendet. Die Eignung von UML-Klassendiagrammen zur Modellierung von Ontologien ist bereits in Kapitel 6.2.1 diskutiert worden. Für den Einsatz unseres Model-Driven-Matching-Konzeptes im Kontext von Web Services, ist neben der visuellen Darstellung eine geeignete Repräsentation der Ontologien basierend auf existierenden, standardisierten Semantic-Web-Sprachen notwendig (vgl. auch Definition Web Services in Abschnitt 1.1). Der Zusammenhang zwischen UML und existierenden Semantic-Web-Sprachen zur Repräsentation von Ontologien ist bereits in diversen Publikationen diskutiert worden [Cra01, BKK+01, BKK+02, FFJ+02, DGD05].

Die Semantic-Web-Sprache DAML+OIL ist eine RDF-Erweiterung, die speziell zur Repräsentation von Ontologien entwickelt wurde. Ein geeignetes Mapping zwischen UML-Klassendiagrammen und DAML+OIL ist z.B. von Baclawski et. al. definiert worden [BKK+01, BKK+02, Wen02]. Bei diesem Mapping werden die diversen Ähnlichkeiten von Klassendiagrammen und

DAML+OIL ausgenutzt. Wir verwenden das Mapping, um eine DAML+OIL-basierte Repräsentation der mit UML-Klassendiagrammen modellierten Ontologien zur erzeugen. In Anhang C.1 ist eine DAML+OIL Repräsentation der Ontologie aus Abbildung 6.2 abgebildet, die gemäß dem Mapping von Baclawski et. al. erzeugt wurde[5].

7.3.2 Repräsentation von visuellen Kontrakten

Die Semantik eines Service oder einer Suchanfrage wird in dem in dieser Arbeit verwendeten modellbasierten Ansatz mit visuellen Kontrakten beschrieben. Die linke und rechte Seite eines visuellen Kontraktes besteht aus UML-Composite-Structure-Diagrammen, die über einer gegebenen Ontologie (dargestellt mit UML-Klassendiagrammen), getypt sind.

In DAML+OIL-Dokumenten können nicht nur Ontologien, sondern auch Instanzen von Ontologien dargestellt werden. DAML+OIL-Dokumente können also zur Repräsentation von UML-Composite-Structure-Diagrammen verwendet werden. Im Kontext der Abbildung von UML-Klassendiagrammen nach DAML+OIL haben Baclawski et. al. auch ein Mapping von UML-Objektdiagrammen nach DAML+OIL definiert [BKK+01, BKK+02, Wen02]. Das Mapping verwenden wir zur Abbildung von UML-Composite-Structure-Diagrammen (die nahezu identisch zu UML-Objektdiagrammen sind) nach DAML+OIL.

Derzeit existieren jedoch keine standardisierten Semantic-Web-Sprachen zur Repräsentation von visuellen Kontrakten beziehungsweise zur Repräsentation von Paaren von Composite-Structure-Diagrammen oder Objektdiagrammen. Daher ist im Kontext einer Diplomarbeit [Sev03] ein RDF-basiertes Dateiformat zur Repräsentation von visuellen Kontrakten entwickelt worden. Dieses Dateiformat erlaubt die Einbettung von Paaren von DAML+OIL-Dokumenten zur Repräsentation von UML-Composite-Structure Diagrammen in ein RDF-Dokument. Das heißt mit diesem Dateiformat kann die linke und rechte Seite eines visuellen Kontraktes in einem RDF-basierten Dateiformat gespeichert werden. Die RDF bzw. DAML+OIL basierte Repräsentation der linken und rechten Seite eines visuellen Kontraktes wird gemäß dem Mapping von Baclawski et. al. erzeugt. Die RDF-basierte Repräsentation des visuellen Kontraktes aus Abbildung 6.3 befindet sich in Anhang C.2[6].

[5]Die Ontologie in Anhang C.1 haben wir automatisiert mit unserer VC Editor Workbench (Kapitel 8) aus einem Klassendiagramm generiert.

[6]Das RDF-Dokument haben wir automatisiert mit unserer VC Editor Workbench aus einem visuellen Kontrakt generiert.

7.4 Matching-Algorithmus

In diesem Abschnitt beschreiben wir den in einer Diplomarbeit [Sev03] implementierten Algorithmus zur Berechnung des in Kapitel 6 vorgestellten Matching-Konzeptes vorgestellt. Der Algorithmus nutzt die RDF-basierte Repräsentation der Ontologien und visuellen Kontrakte zur Berechnung des Matchings aus. Für die Bestimmung der notwendigen Teilgraphbeziehungen wird die RDF-basierte Anfragesprache RDQL und das Semantic Web Toolkit Jena [Hew] verwendet.

Das Aktivitätendiagramm in Abbildung 7.2 skizziert die Vorgehensweise des Matching-Algorithmus aus [Sev03]. Als Eingabe bekommt der Algorithmus drei RDF-Dateien: Einen Requestor- ($r : L_r \rightarrow R_r$), einen Provider-Kontrakt ($p : L_p \rightarrow R_p$) und die Ontologie (o), über welche die Kontrakte getypt sind. Wenn der Provider-Kontrakt auf den Requestor-Kontrakt matcht, so gibt der Algorithmus `true` zurück. Passt der Provider-Kontrakt nicht zu dem Requestor-Kontrakt, so gibt der Algorithmus `false` zurück. Bei der Berechnung des Matchings wird zwischen einer Validierungs- und einer Matchingphase unterschieden.

Der Provider- und der Requestor-Kontrakt sind in dem aus Abschnitt 7.3.2 beschriebenem RDF-basierten Dateiformat gespeichert. Jeder Kontrakt enthält ein Paar von DAML+OIL-Dokumenten zur Repräsentation der Vor- und Nachbedingung. In der Validierungsphase werden als erstes die Vor- und Nachbedingungen der Kontrakte extrahiert, d.h. der Algorithmus extrahiert die vier DAML+OIL-Dokumente L_r, R_r, L_p und R_p aus dem Requestor-Kontrakt r und dem Provider-Kontrakt p.

Dann wird ein DAML-Validator aufgerufen, der überprüft, ob es sich bei den extrahierten Vor- und Nachbedingungen um valide DAML+OIL-Dokumente handelt. Ist eines der DAML+OIL-Dokumente nicht valide, so ist dieses Dokument auch nicht korrekt gegenüber der DAML+OIL-Ontologie getypt. In diesem Fall ist auch der gesamte Kontrakt nicht korrekt über der Ontologie getypt und die Berechnung des Matching wird abgebrochen. Wenn alle vier Dokumente valide sind, so sind auch die zugehörigen Kontrakte korrekt gegenüber der Ontologie getypt und der Algorithmus kann mit der Matching-Phase fortgesetzt werden.

Die Matchingphase beginnt mit dem Vergleich der Vorbedingungen L_r und L_p. Dazu wird aus der Vorbedingung des Providers L_p zunächst eine RDQL-Suchanfrage berechnet. Wie bereits in Abschnitt 7.2.4 gezeigt, beschreibt eine RDQL-Suchanfrage einen gesuchten Graphen mit RDF-Tripeln. Subjekt,

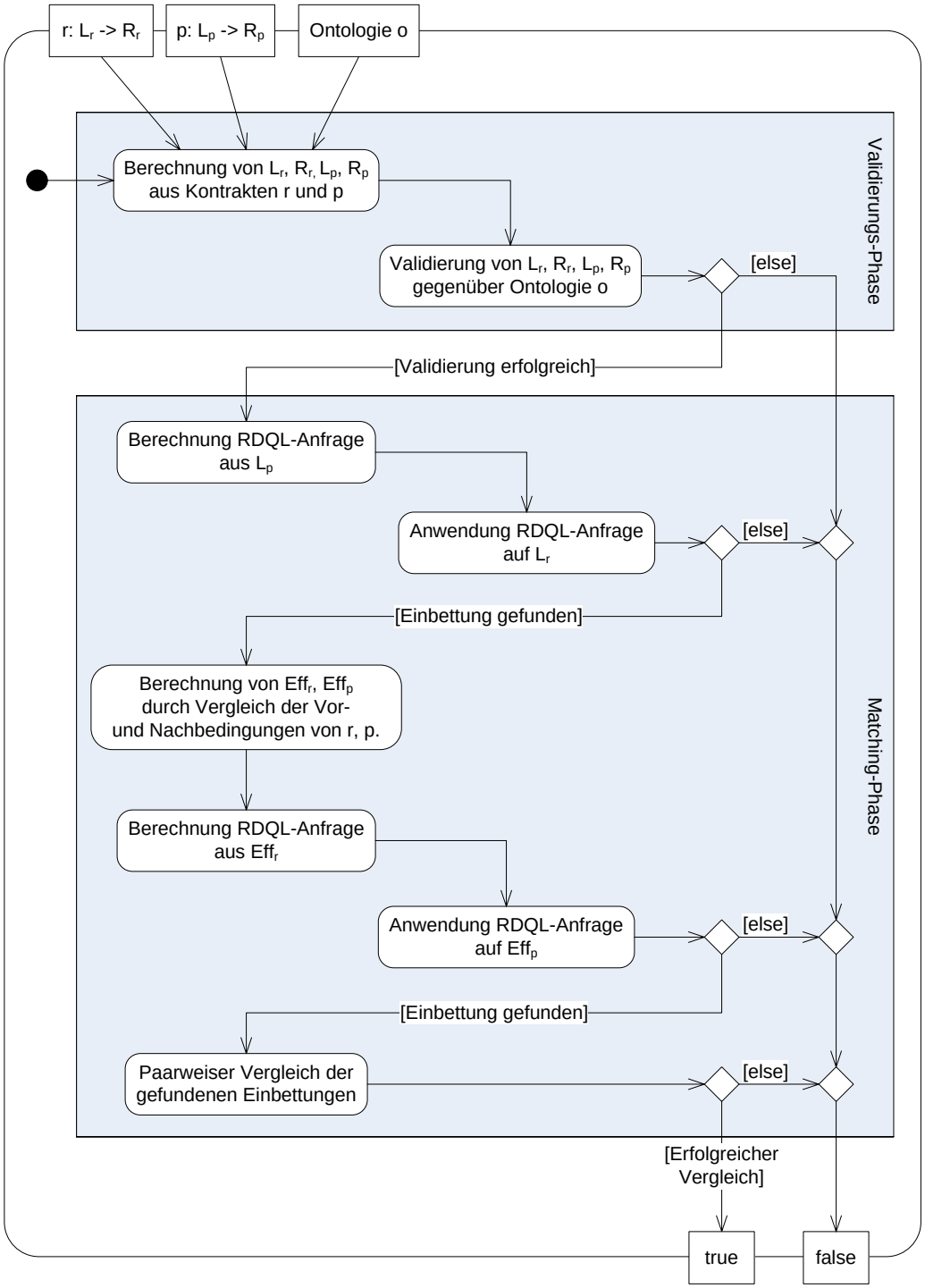


Abbildung 7.2: Matching-Algorithmus

Prädikat oder Objekt eines RDF-Tripels können in einer RDQL-Suchanfrage durch Variablen ersetzt werden. Aus dem RDF-Graphen des Providers L_p kann eine geeignete Suchanfrage generiert werden, indem die Objektnamen in dem RDF-Graphen durch Variablen ersetzt werden. Hierzu ist den Objektnamen lediglich ein Fragezeichen voranzustellen. Wird dieses Vorgehen auf die visuelle Modellebene übertragen, so werden den Rollennamen der `ConnectableElements` in einem Composite-Structure-Diagramm Fragezeichen vorangestellt. Alle so erzeugten Variablen werden dann dem SELECT-Statement der RDQL-Suchanfrage hinzugefügt, um im letzten Schritt der Matchingphase auf die gefundenen Einbettungen zurückgreifen zu können. Die Typinformationen und die Links werden ohne Änderung aus L_p in die RDQL-Suchanfrage übernommen.

Die generierte RDQL-Suchanfrage kann dann auf den RDF-Graphen L_r angewendet werden. Werden für alle in der RDQL-Suchanfrage definierten Variablen (d.h. für alle Objekte aus L_p) konkrete Werte gefunden, so stellt der in der RDQL-Suchanfrage definierte RDF-Graph und damit der RDF-Graph L_p einen Teilgraph von L_r dar. In diesem Fall bietet der Requestor alle vom Provider benötigten Eingaben zur Ausführung des Service an und die Berechnung des Matchings kann fortgesetzt werden. Wenn nicht für alle in der RDQL-Suchanfrage definierten Variablen konkrete Werte gefunden werden, so ist die Vorbedingung des Service Providers kein Subgraph der Vorbedingung des Service Requestors, d.h. der Requestor liefert nicht alle für die Ausführung eines Service notwendigen Informationen. In diesem Fall kann das Matching mit einem negativen Ergebnis abgebrochen werden.

Im nächsten Schritt des Matching-Algorithmus sind die Effekte der Provider-Regel mit den Effekten der Requestor-Regel zu vergleichen. Hierzu müssen als erstes die Effekte durch einen Vergleich der Vor- und Nachbedingungen berechnet werden. Der Algorithmus berechnet zwei neue RDF-Graphen: Eff_p durch Vergleich von L_p und R_p beziehungsweise Eff_r durch einen Vergleich von L_r und R_r. Die Effekte in Abschnitt 6.4 umfassen sowohl die erzeugten Objekte und Links als auch die gelöschten Objekte und Links sowie alle Objekte an den Enden von erzeugten Objekten und Links. Da in einem RDF-Dokument keine gelöschten Elemente bzw. nicht existenten Elemente dargestellt werden können, ist der Algorithmus aus [Sev03] gegenüber unserer formalen Definition des Matchings aus Abschnitt 6.4 eingeschränkt. Es werden nur die „positiven“ Effekte berechnet, d.h. die Mengen Eff_p und Eff_r enthalten nur die in einem Kontrakt erzeugten Objekte und Links.

Nach der Ermittlung der Effekte für p und r, berechnet der Algorithmus aus Eff_r eine RDQL-Suchanfrage. Wie bereits oben beschrieben, kann aus dem

RDF-Graphen Eff_r eine geeignete Suchanfrage generiert werden, indem die Objektnamen in dem RDF-Graphen durch Variablen ersetzt werden, d.h. aus den Rollennamen der `ConnectableElements` werden Variablen. Hierzu ist den Objektnamen lediglich ein Fragezeichen voranzustellen. Auch an dieser Stelle des Algorithmus werden alle so erzeugten Variablen dem SELECT-Statement der RDQL-Suchanfrage hinzugefügt, um im letzten Schritt der Matchingphase auf die gefundenen Einbettungen zurückgreifen zu können. Die Typinformationen und die Links werden ohne Änderung aus Eff_r in die RDQL- Suchanfrage übernommen.

Die generierte RDQL-Suchanfrage kann dann auf den RDF-Graphen Eff_p angewendet werden. Werden für alle in der RDQL-Suchanfrage definierten Variablen (d.h. für alle Objekte aus Eff_r) konkrete Werte gefunden, so stellt der in der RDQL-Suchanfrage definierte RDF-Graph und damit der RDF-Graph Eff_r einen Teilgraph von Eff_p dar. In diesem Fall berechnet der Service Provider alle vom Service Requestor benötigten Ergebnisse und die Berechnung des Matchings kann fortgesetzt werden. Wenn nicht für alle in der RDQL-Suchanfrage definierten Variablen konkrete Werte gefunden werden, so sind die Effekte des Service Requestors kein Subgraph der Effekte des Service Providers, d.h. der Provider liefert nicht alle vom Requestor geforderten Ergebnisse. In diesem Fall wird das Matching mit einem negativen Ergebnis abgebrochen werden.

Wenn mit allen in dem Matching-Algorithmus durchgeführten Suchanfragen erfolgreich ein Subgraph bestimmt werden konnte, so ist die Vorbedingung des Provider-Kontraktes ein Subgraph der Vorbedingung des Requestor-Kontraktes und die Effekte des Requestor-Kontraktes sind ein Subgraph der Effekte des Provider-Kontraktes. Bisher hat der Matching-Algorithmus also das in Abschnitt 6.3.1 definierte intuitive Matching bestimmt. Um auch der formalen Definition des Matchings gerecht zu werden, werden im letzten Schritt der Matching-Phase die gefundenen Subgraph-Beziehungen der Vorbedingungen und Effekte paarweise miteinander verglichen, um zu bestimmen, ob eine Relation h (wie in Abschnitt 6.4 definiert) existiert. Bei einem positiven Ausgang der Überprüfung wurde ein Matching gefunden und der Algorithmus kann beendet werden. Bei einem negativen Ausgang der Prüfung matcht p nicht auf r.

Die Implementierung des hier skizzierten Algorithmus wurde im Rahmen einer Diplomarbeit entwickelt [Sev03]. Die Implementierung wurde mit Java in der Version 1.4 realisiert. Zur Validierung der Vor- und Nachbedingungen der Provider- und Requestor Regeln wird der unter [DAM] verfügbare DAML-Validator verwendet. Die RDQL-Suchanfragen werden mit dem Open-Source

Semantic Web Toolkit Jena in der Version 1.6.1 von Hewlett Packard [Hew] bearbeitet.

7.5 Fazit

In diesem Kapitel haben wir einen Algorithmus vorgestellt, der existierende Semantic-Web-Sprachen als Eingabe sowie Semantic-Web-Werkzeuge verwendet, um das in Kapitel 6 beschriebene Matching-Verfahren zu realisieren. Zuvor haben wir gezeigt, wie DAML+OIL als Austauschformat für Ontologien verwendet werden kann. Weiterhin haben wir ein RDF/XML-basiertes Austauschformat für Kontrakte eingeführt, in welchem die linke und rechte Seite einer Regel als DAML+OIL-Dokumente gespeichert werden. Mit der Repräsenation der Ontologien und unserer visuellen Kontrakte in einem XML-basierten Austauschformat kommen wir der Forderung des World Wide Web Consortiums, dass Web Services mit XML-basierten Sprachen zu beschreiben sind, nach.

Der Matching-Algorithmus extrahiert die DAML+OIL-Dokumente aus dem RDF/XML-basierten Austauschformat zur Berechnung eines Matching. Die Berechnung des in Abschnitt 6.4 eingeführten Matching-Konzeptes erfolgt auf der Basis von Teilgraphbeziehungen. Zur Berechnung der Teilgraphenbeziehungen werden RDQL-Suchanfragen aus dem RDF/XML-basierten Austauschformat berechnet. Das Semantic Web Toolkit Jena von HP kann die RDQL-Suchanfragen auswerten und somit mögliche Grapheinbettungen berechnen.

Kapitel 8

Visual Contract Workbench

Die in dieser Arbeit vorgestellte modellbasierte Vorgehensweise zur Entwicklung von Softwarekomponenten für den Einsatz in serviceorientierten Architekturen basiert auf zwei grundlegend neuen Techniken.

Zum einen haben wir das Konzept des Model-Driven Monitoring eingeführt. Hierbei werden visuelle Kontrakte nach JML übersetzt, um die Korrektheit einer manuellen Implementierung gegenüber einer Spezifikation mit Klassendiagrammen und visuellen Kontrakten zu überprüfen.

Zum anderen haben wir das Konzept des Model-Driven Matchings erläutert. Das hierbei eingeführte Matching-Konzept zum Vergleich zweier visueller Kontrakte erlaubt eine modellbasierte Suche von Services in einer serviceorientierten Architektur. Eine Übersetzung der visuellen Kontrakte nach DAML+OIL erlaubt eine Berechnung des eingeführten Matchings mit aktuellen Semantic Web Technologien.

Aktuelle CASE Tools unterstützen die beschriebenen Vorgehensweisen zum Einsatz visueller Kontrakte nicht. Daher haben wir zur praktischen Erprobung unseres Ansatzes eine prototypische Implementierung einer integrierten Entwicklungsumgebung für visuelle Kontrakte (*Visual Contract Workbench*) entwickelt. Die Visual Contract Workbench erlaubt die Modellierung von Klassendiagrammen und visuellen Kontrakten sowie eine Codegenerierung nach Java bzw. JML und DAML+OIL.

Die Visual Contract Workbench ist als Eclipse Plugin realisiert. Nach einer kurzen Einführung in die verwendeten Eclipse Technologien, beschreiben wir die wichtigsten Funktionen des Prototypen und geben einen kurzen Überblick über seine Architektur.

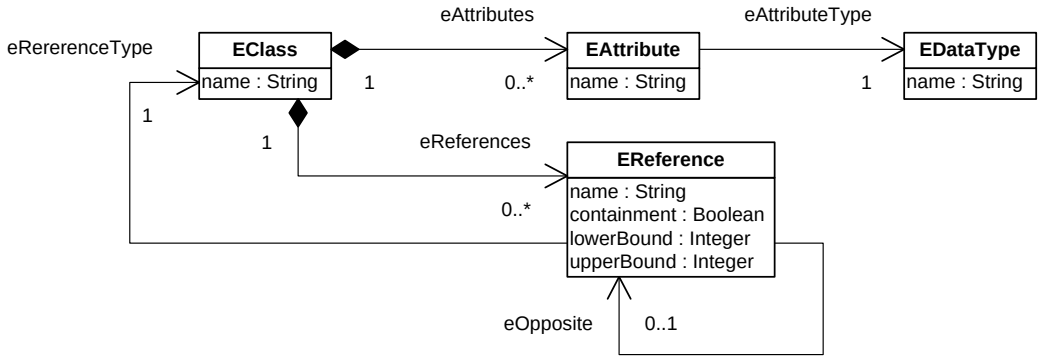


Abbildung 8.1: Ausschnitt aus dem EMF-Ecore-Model

8.1 Eclipse

Die Open-Source Plattform Eclipse hat sich nicht nur als Entwicklungsumgebung für Java sondern auch als erweiterbare Plattform für Entwicklungsumgebungen einen Namen gemacht. Die Visual Contract Workbench ist als Plugin für die Eclipse-IDE entwickelt worden.

Im Folgenden geben wir einen kurzen Überblick über die wichtigsten Eclipse-Technologien, die zur Entwicklung der Visual Contract Workbench verwendet wurden. Dabei konzentrieren wir uns auf eine konzeptionelle Beschreibung der einzelnen Technologien und weniger auf detaillierte technische Eigenschaften, wie z.B. Schnittstellen etc.

8.1.1 Eclipse Modeling Framework

Das *Eclipse Modeling Framework* (*EMF*) [Ecl06b] ist ein Java-Framework und eine Codegenerierungs-Engine, um Anwendungen für Eclipse zu erstellen, die auf strukturierten Modellen basieren.

Die Grundlage hierzu bildet ein vorgegebenes Metamodell, welches in EMF als *Ecore* bezeichnet wird. Das EMF-Ecore-Modell ist bis auf kleinere Unterschiede identisch zu dem Essential-MOF-Modell (EMOF), einer Teilmenge des MOF-Modells (MetaObject Facility Specification) [OMG06]. Abbildung 8.1 zeigt einen Ausschnitt[1] des Ecore-Modells.

[1]Die in der Abbildung dargestellten Klassen besitzen im vollständigen Ecore-Modell noch eine gemeinsame Oberklassse `ENamedElement`, die das Attribut `name` definiert. In dem vereinfachten Modell haben wir dieses Attribut allen Klassen hinzugefügt.

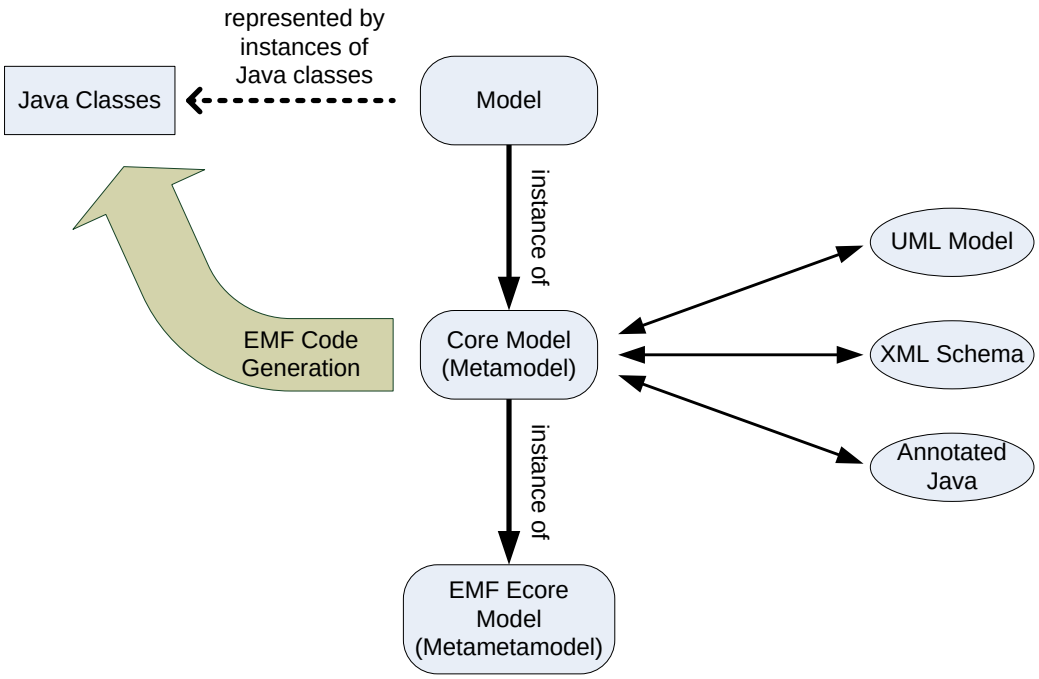


Abbildung 8.2: Zusammenhang zwischen EMF-Ecore-Modell, Core-Modell und Instanzen des Core-Modells

Die Klassen des Ecore-Metamodells können zur Definition neuer (Meta-) Modelle verwendet werden. Instanzen eines Ecore-Modells werden als *Core Model* bezeichnet. Im Moment sieht das EMF-Framework unterschiedliche Möglichkeiten zur Generierung eines Core-Modells vor (vgl. auch Abbildung 8.2). Das EMF-Framework kann ein Core-Modell aus einem XML Schema, einem UML-Modell oder aus annotierten Java-Klassen ableiten. Weiterhin kann das Core-Modell mit entsprechenden Werkzeugen, wie z.B. dem OMONDO-UML-Plug-In, direkt editiert werden.

Aus einem EMF-Core-Modell kann wiederum Java-Code generiert werden. Dieser Java-Code kann in einer Anwendung genutzt werden, um Instanzen des Core-Modells zu verwalten. Abbildung 8.2 fasst die Zusammenhänge zwischen den verschiedenen Modellen noch einmal zusammen.

8.1.2 Graphical Editor Framework

Das *Graphical Editor Framework* (*GEF*) [Ecl06a] ist ein Framework zur Unterstützung der Entwickung grafischer Editoren auf Basis von Modellen. Edi-

toren, die GEF verwenden, liegt das Model-View-Controller-Prinzip zugrunde, um ein Programm mit den drei Einheiten Modell (engl. Model), Präsentation (engl. View) und Programmsteuerung (engl. Controller) aufzubauen.

In GEF kann es sich bei einem Modell um eine beliebige Java-Klasse bzw. ein beliebiges Objekt handeln. Das Modell kennt weder den View noch den Controller. Änderungen im Modell werden allerdings über einen Update-Mechanismus, der Events auslöst, bekannt gegeben. Dazu muss sich zumindest die View als abhängiges Objekt am Modell registrieren, um über Datenänderungen informiert zu werden.

Der View präsentiert ein Modell visuell. Die Programmlogik sollte aus dem View entfernt werden, und er sollte unabhängig vom Modell und Controller entwickelt werden. Die graphische Repräsentation von Modell-Elementen wird in GEF durch Draw2d-Figures realisiert. *Draw2d* ist eine weitere Eclipse-Technologie, die verschiedene grundlegende Funktionalitäten zur Darstellung und Zeichnung von graphischen Elementen bietet. GEF baut auf Draw2d auf und erlaubt z.B. eine hierarchische Strukturierung der Figures und die Verwendung von Layout-Managern zu deren Formatierung.

Die Controller steuern das Zusammenwirken des Modells mit seinen Views. Die Controller werden in GEF mit so genannten `EditPart`s realisiert. Bei einer Änderung des Modells sendet dieses eine Nachricht an den entsprechenden `EditPart`, das daraufhin ein Update des Views initialisiert.

8.1.3 Java Emitter Templates

Das Eclipse-Projekt *Java Emitter Templates* (*JET*) [Ecl06c] ist ein Teilprojekt des Eclipse Modeling Frameworks. Mit JET soll die Entwicklung einer automatischen Codegenerierung vereinfacht werden. Die Ausgabe von JET ist nicht auf ein bestimmtes Format beschränkt. JET ist in der Lage, jede Art von textbasiertem Output (Java, XML, JML usw.) zu generieren.

Wie der Name bereits erkennen lässt, handelt es sich bei JET um eine templatebasierte Technologie. Um JET einsetzen zu können, wird eine Eclipse-Version mit einer kompatiblen EMF-Version benötigt. Die JET-Engine — wie auch EMF — ist nur unter Eclipse lauffähig.

Der Prozess der Codegenerierung unterteilt sich bei JET in zwei Schritte (vgl. Abbildung 8.3). Der Entwickler schreibt zunächst eine Menge von Templates, die ähnlich wie der zu erzeugende Output strukturiert sind (vgl. auch Abschnitt 5.2.1). Diese Templates werden von dem JET-Framework inter-

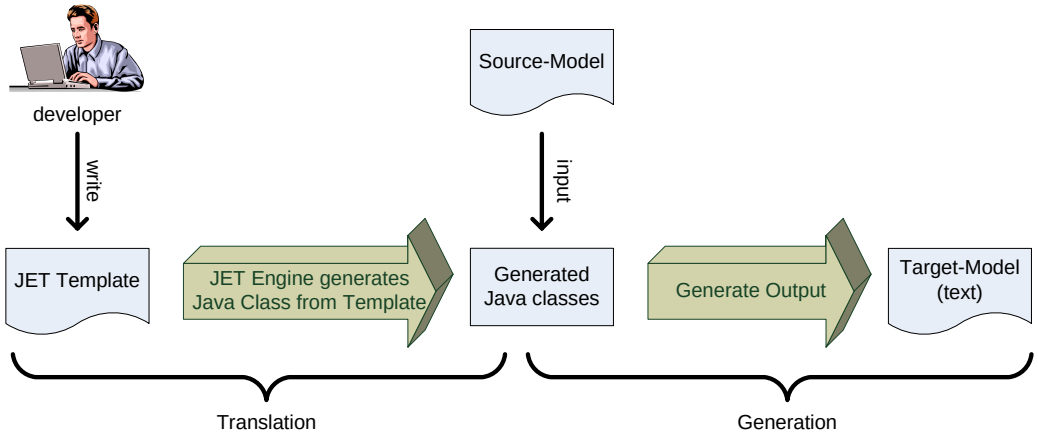


Abbildung 8.3: Funktionsweise der Java Emitter Templates

pretiert und in so genannte Implementierungsklassen übersetzt. Dieser Teilprozess wird als *Translation* bezeichnet.

Die erzeugten Klassen können nun beliebig instanziiert und mit einem passendem Quellmodell aufgerufen werden, um die gewünschte Ausgabe zu generieren. Die Ausgabe erfolgt als String und kann weiterverarbeitet werden. Dieser Teilprozess wird als *Generation* bezeichnet. Hier erfolgt die eigentliche Codegenerierung.

8.2 Produktfunktionen

Zur Realisierung des Model-Driven Monitoring haben wir in Abschnitt 2.5 gezeigt, wie visuelle Kontrakte in einem Softwareentwicklungsprozess eingesetzt werden können. Abhängig von der Phase des Softwareentwicklungsprozesses haben wir für den Softwareentwickler und den Programmierer unterschiedliche Aufgaben identifiziert.

Für das Model-Driven Matching haben wir gezeigt, wie visuelle Kontrakte zur semantischen Beschreibung eines Service und einer Suchanfrage eingesetzt werden können. Die Umsetzung des Matchings kann mit Semantic-Web-Sprachen erfolgen. Jedoch soll ein Domänenexperte die Ontologien, die Servicebeschreibungen und Suchanfragen mit unserer Visual Contract Workbench modellbasiert erstellen können.

Abbildung 8.4 zeigt die wichtigsten Use Cases der Visual Contract Workbench. Wir unterscheiden gemäß den oben gemachten Vorgaben zwischen

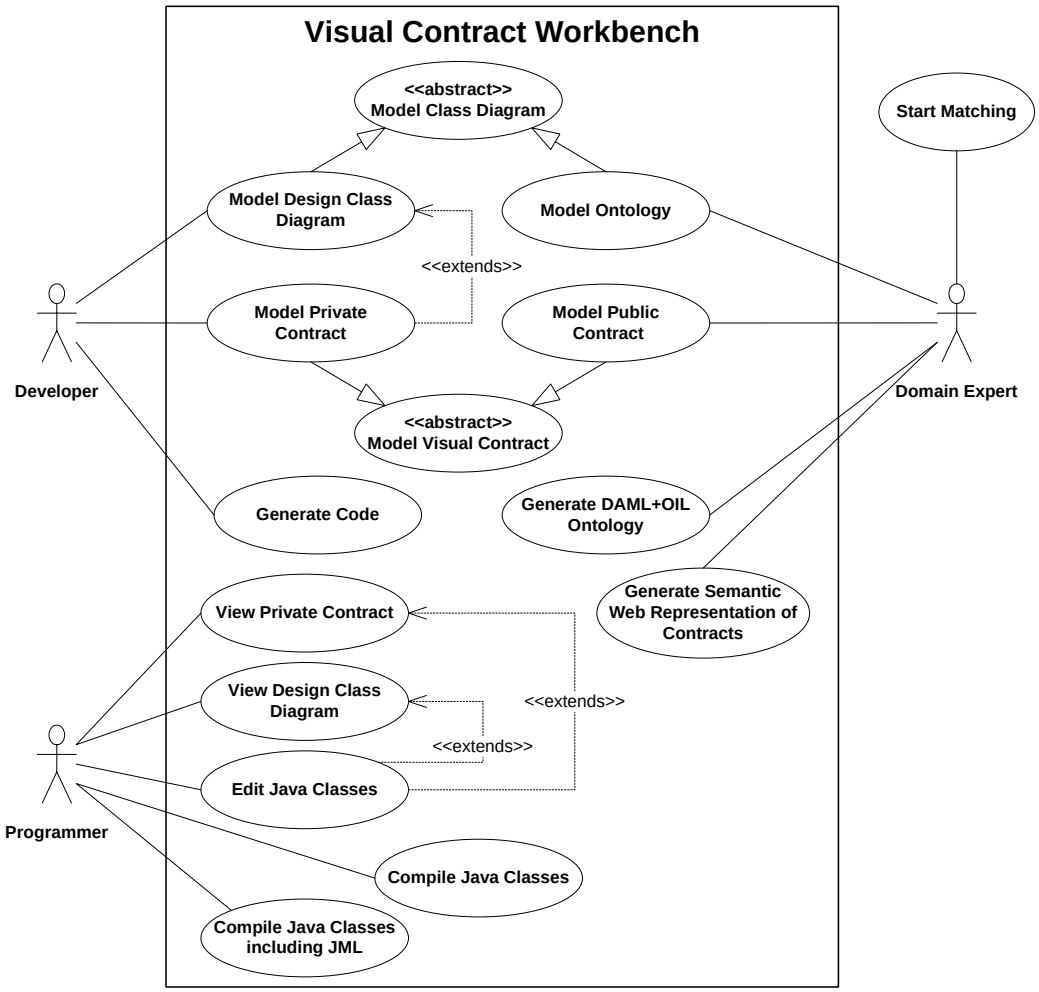


Abbildung 8.4: Basis-Use-Cases der Visual Contract Workbench

einem Softwareentwickler, einem Programmierer und einem Domänenexperten.

8.2.1 Modellierung von Klassendiagrammen

Bei der Produktfunktion *Model Class Diagram* ist zu unterscheiden, ob mit der Visual Contract Workbench ein Design-Klassendiagramm oder eine Ontologie erstellt werden soll.

Modellierung von Design-Klassendiagrammen

Mit der Produktfunktion *Model Design Class Diagram* der Visual Contract Workbench kann ein Softwareentwickler ein statisches Modell eines zu entwickelnden Systems erstellen. Bei dem statischen Modell handelt sich um ein Klassendiagramm, das mit visuellen Kontrakten (Produktfunktion *Model Private Contract*) zu Modellierung von Verhalten ergänzt werden kann.

Abbildung 8.5 zeigt einen Screenshot der Visual Contract Workbench, der die Erstellung von Design-Klassendiagrammen illustriert. Zentral in dem Screenshot ist die Arbeitsfläche zum Editieren von Klassendiagrammen. Der Screenshot zeigt die Umsetzung des Klassendiagramms aus Abbildung 3.1 mit der Visual Contract Workbench. Das Klassendiagramm besteht aus Klassen mit Attributen und Operationen sowie aus Assoziationen, die gerichtet oder ungerichtet sein können.

Links neben der Arbeitsfläche zum Editieren von Klassendiagrammen befindet sich eine Palette. In der Palette können zum einen Werkzeuge zum Selektieren einzelner (*Select*) oder einer Menge von Objekten (*Marquee*) ausgewählt werden. Zum anderen enthält diese Palette eine Menge von UML-Notationselementen (*Class*, *Interface*, *Operation*, etc.), die selektiert werden können, wenn auf der Arbeitsfläche ein neues Element eingefügt werden soll. Die Visual Contract Workbench stellt die korrekte Verwendung der Elemente sicher. Assoziationen können zum Beispiel nur zwischen zwei Klassen bzw. Schnittstellen gezeichnet werden und Attribute oder Operationen werden in den korrekten Bereich einer Klasse eingefügt.

Unter der Arbeitsfläche befindet sich ein *Properties View*. Über diesen View können die Eigenschaften eines im Hauptfenster selektierten UML-Elementes editiert werden. Über den Properties View werden z.B. Klassennamen oder Variablennamen und Typen von Variablen festgelegt. In dem dargestellten Screenshot werden in dem Properties View die Eigenschaften der Assoziation `controlsProduct` angezeigt.

Mit einem Doppelklick auf eine Operation einer Klasse oder über ein kontextsensitives Menu (siehe Abbildung 8.6), das mit der rechten Maustaste aufgerufen wird, kann ein bestehender visueller Kontrakt einer Operation geöffnet werden oder der Operation ein neuer visueller Kontrakt hinzugefügt werden.

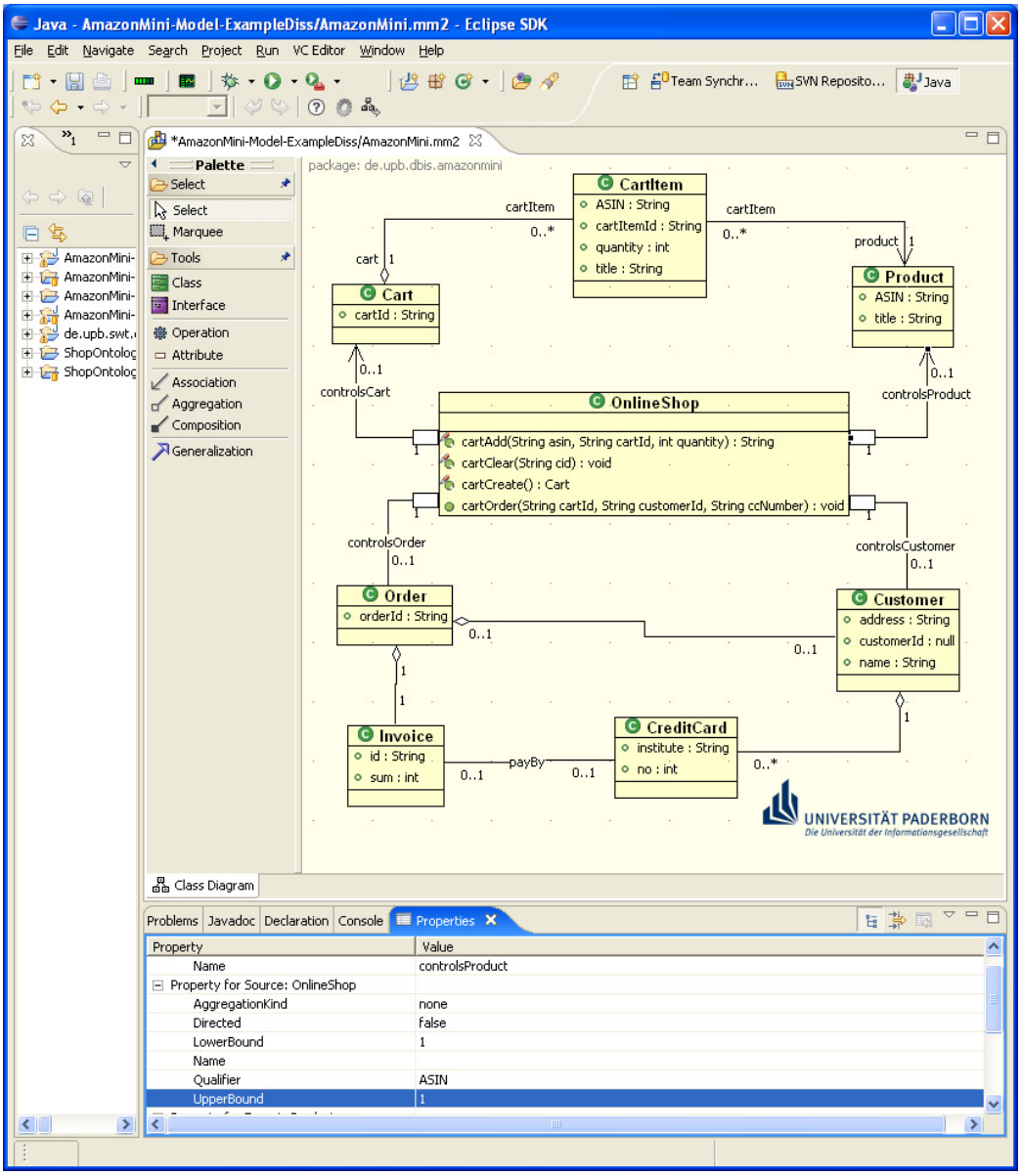


Abbildung 8.5: Design-Klassendiagramm in Visual Contract Workbench

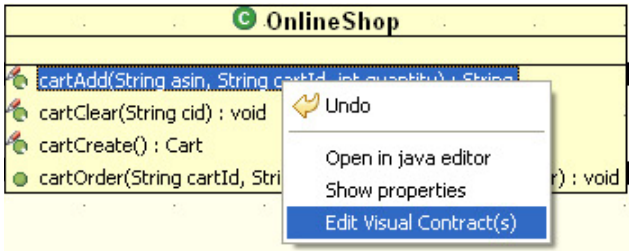


Abbildung 8.6: Kontextmenu zum Editieren der Kontrakte einer Operation

Modellierung von Ontologien

Mit der Produktfunktion *Model Ontology* der Visual Contract Workbench kann ein Domänenexperte Ontologien modellieren (siehe Abbildung 8.7). Zur Modellierung der Ontologien werden Klassendiagramme bestehend aus Klassen, Assoziationen, Aggregationen, Kompositionen und Vererbungen verwendet. Die Modellierung von Schnittstellen, Operationen oder Attributen sind bei der Modellierung von Ontologien nicht erlaubt. Damit kann ein semantischer Kontrakt auch nicht einer Operation zugeordnet werden. Er ist lediglich über eine Ontologie getypt (vgl. auch Erläuterungen zur Produktfunktion *Model Semantic Contract*). Ansonsten verhält sich die Oberfläche der Visual Contract Workbench zur Modellierung von Ontologien wie in der Beschreibung der Produktfunktion *Model Design Class Diagram*.

8.2.2 Modellierung von visuellen Kontrakten

Mit der Produktfunktion *Model Visual Contract* der Visual Contract Workbench können visuelle Kontrakte modelliert werden. Hierbei ist zu unterscheiden, ob die visuellen Kontrakte zur Detaillierung eines Softwaresystems bzw. einer Operation eines Softwaresystems (*Model Private Contract*) oder zur semantischen Beschreibung eines Service bzw. einer Suchanfrage (*Model Public Contract*) verwendet werden sollen (vgl. auch Abschnitt 2.6 und insbesondere die Erläuterungen zu Abbildung 2.10).

Modellierung privater Kontrakte

Mit der Produktfunktion *Model Private Contract* der Visual Contract Workbench kann ein Softwareentwickler das Verhalten einer Operation mit einem

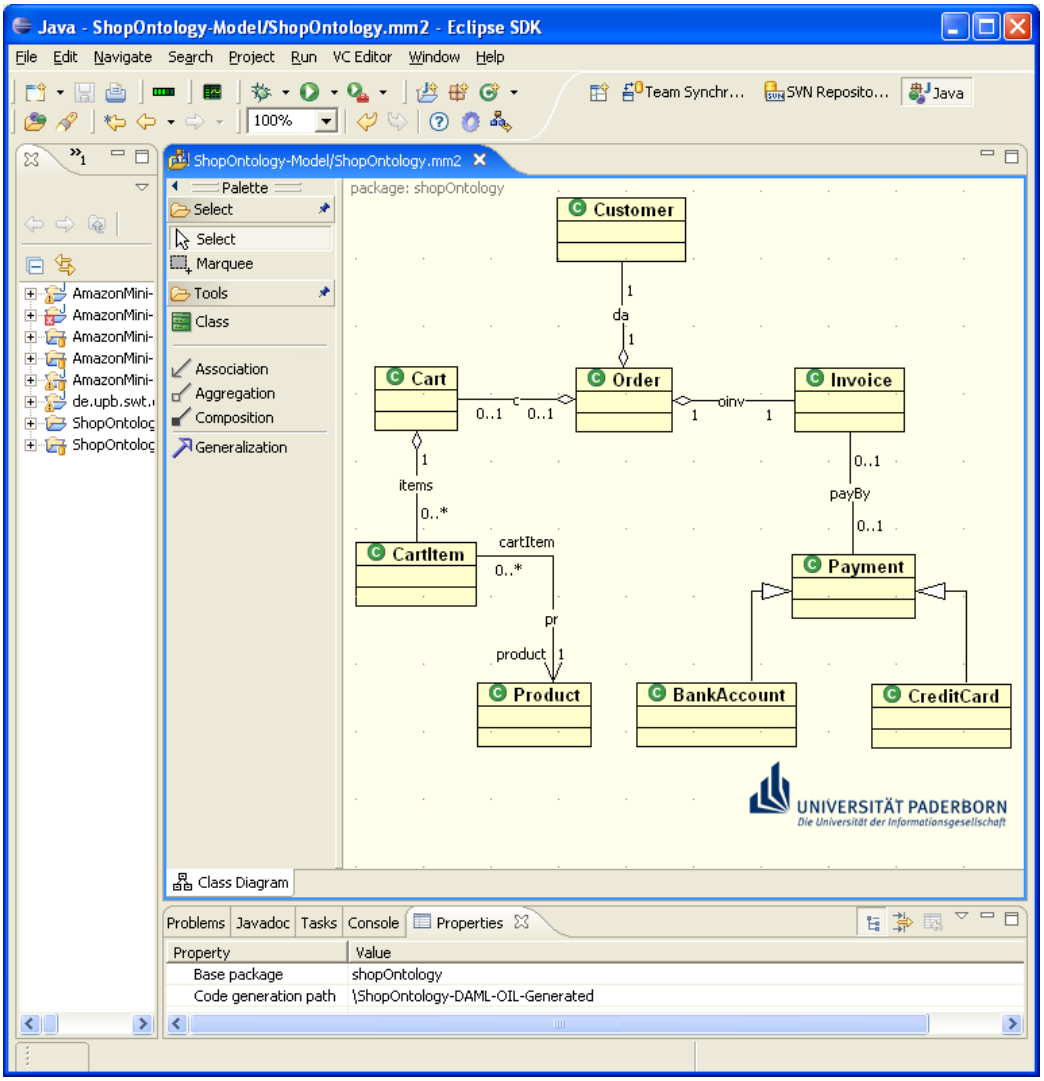


Abbildung 8.7: Klassendiagramm zur Repräsentation einer Ontologie

visuellen Kontrakt detaillieren. Ein visueller Kontrakt beschreibt das Verhalten einer Operation über die Veränderung der Objektstrukturen durch Vor- und Nachbedingungen. Abbildung 8.8 zeigt einen Screenshot zur Illustration der Editiermöglichkeiten von visuellen Kontrakten.

Zentral in dem Screenshot ist die Arbeitsfläche zum Editieren der visuellen Kontrakte. In dem Feld links unten auf der Arbeitsfläche (*LHS*) wird die Vorbedingung eines visuellen Kontraktes modelliert. In dem Feld rechts unten (*RHS*) wird die Nachbedingung eines visuellen Kontraktes modelliert. In beide Felder wird bei der Erstellung eines neuen visuellen Kontraktes automatisch eine Rolle `this`[2] eingefügt. Ausgehend von dieser Rolle müssen alle weiteren Rollen, die in der Vor- oder Nachbedingung verwendet werden, über Konnektoren erreicht werden können.

In dem Feld links oben auf der Arbeitsfläche (*NAC*) kann eine negative Vorbedingung und in dem Feld rechts oben eine (*NPC*) eine negative Nachbedingung modelliert werden.

In alle vier Felder des Hauptarbeitsbereichs können Rollen, die über Konnektoren miteinander verbunden sind, eingefügt werden. Den Rollen können die in dem Klassendiagramm definierten Typen (Klassennamen) zugewiesen werden. Weiterhin können in einer Rolle die im Klassendiagramm definierten Attribute verwendet werden. Dabei ist eine Auswahl der zur Beschreibung der Operation benötigten Attribute möglich. Es müssen nicht immer alle Attribute einer Klasse in einer Rolle verwendet werden (vgl. Abbildung 8.5 mit Abbildung 8.8).

Über dem Hauptarbeitsbereich befindet sich der Name der Operation, die mit dem visuellen Kontrakt annotiert ist. Unter dem Namen der Operation befindet sich eine Auswahlliste, die es erlaubt (für spätere Erweiterungen) einer Operation mehr als einen visuellen Kontrakt zuzuweisen. Zu diesem Zweck kann ein visueller Kontrakt auch einen Namen bekommen. Weiterhin kann einem visuellen Kontrakt eine Beschreibung zugefügt werden. Falls die mit einem visuellen Kontrakt annotierte Operation einen Rückgabeparameter besitzt, kann dieser mit dem Feld *Return* ausgewählt werden. Mit diesem Feld kann eine Rolle der Nachbedingung oder eine Variable (Abbildung 8.8), die einen Attributinhalt repräsentiert, ausgewählt werden. Die Auswahlliste zeigt nur vom Typ her geeignete Rollen und Variablen an.

[2]Die Visual Contract Workbench verwendet den Namen `this` (und nicht `self`) zur Bezeichnung des aktiven Objektes, da sie bisher sehr stark auf die Java-Codegenerierung (und DAML+OIL-Codegenerierung) ausgelegt ist. Zum Beispiel werden in der Visual Contract Workbench auch Java-spezifische Funktionen zum Vergleich von Attributinhalten mit anderen Werten angeboten.

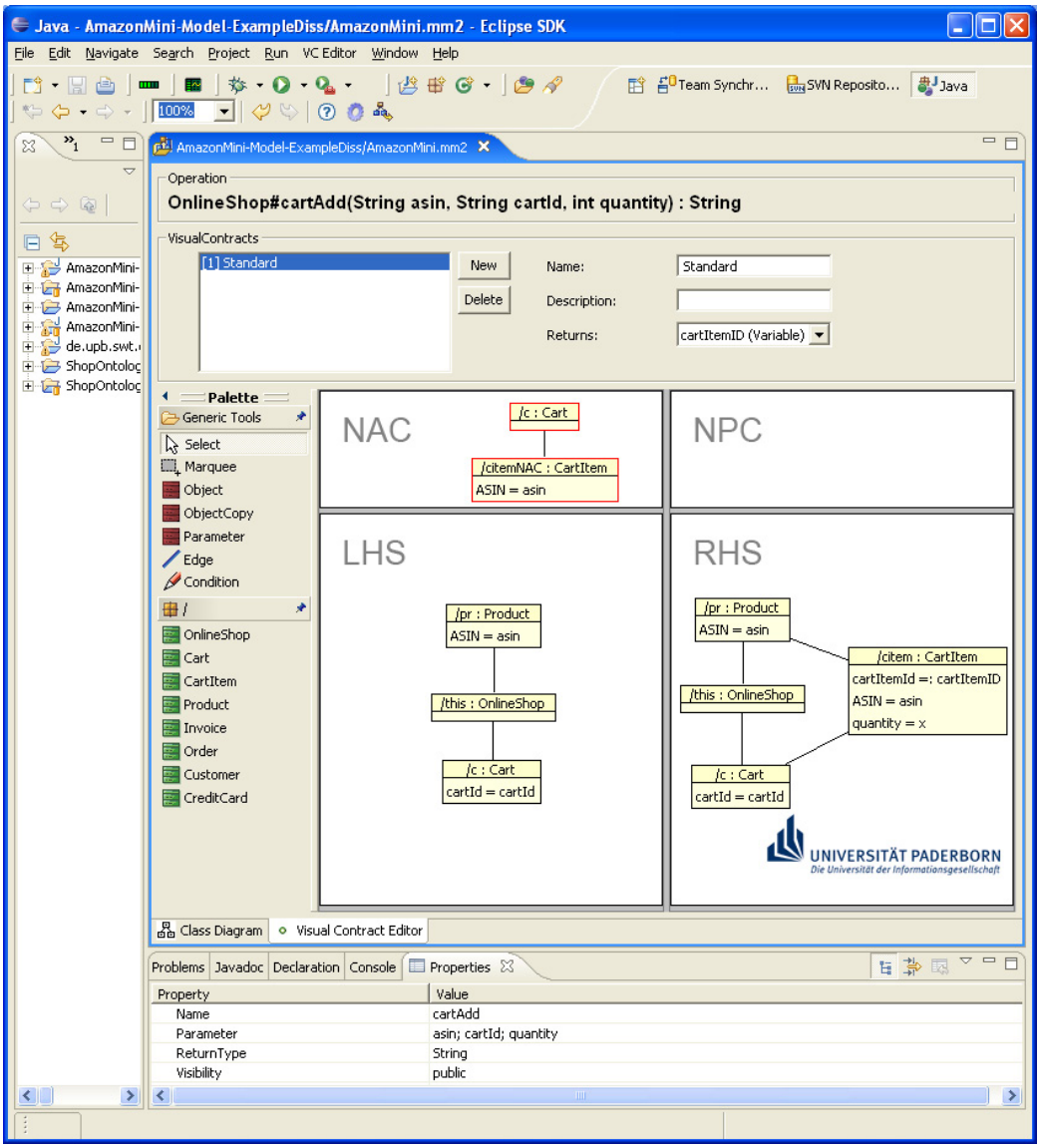


Abbildung 8.8: Visueller Kontrakt zur Detaillierung der Operation `cartAdd`

Links neben der Arbeitsfläche zum Editieren von Klassendiagrammen befindet sich eine Palette. In der Palette können zum einen Werkzeuge zum Selektieren einzelner (*Select*) oder einer Menge von Objekten (*Marquee*) ausgewählt werden. Zum anderen enthält diese Palette eine Menge von Notationselementen und Werkzeugen zum Editieren der Elemente eines visuellen Kontraktes. Mit dem Befehl *Object* kann ein neues Objekt bzw. eine neue Rolle in eines der Felder in dem Hauptarbeitsbereich eingefügt werden. Wenn eine existierende Rolle in ein weiteres Feld übernommen werden soll (wenn z.B. dieselbe Rolle in der Vor- und Nachbedingung verwendet werden soll) kann die Funktion *ObjectCopy* verwendet werden. Mit dem Befehl *Parameter* können Eingabeparameter der Operation in den visuellen Kontrakt übernommen werden. Mit dem Befehl *Edge* können Rollen über Konnektoren verbunden werden und mit dem Befehl *Condition* können einer Rolle Attribute, die durch ihren Typ vorgegeben sind, hinzugefügt werden.

Unter der Arbeitsfläche befindet sich ein *Properties View*. Über diesen View können die Eigenschaften eines im Hauptfenster selektierten Elementes editiert werden. Über den Properties View werden z.B. die Namen der Rollen oder die Inhalte der Attribute vergeben.

Modellierung öffentlicher Kontrakte

Mit der Produktfunktion *Model Public Contract* der Visual Contract Workbench kann ein Domänenexperte die Semantik eines Service oder eine Suchanfrage beschreiben. Ein Service oder eine Suchanfrage wird mit einer Vor- und einer Nachbedingung beschrieben. Abbildung 8.9 zeigt die Beschreibung eines Service zum Bestellen eines existierenden Einkaufswagens.

Die Produktfunktion *Model Public Contract* ist im Wesentlichen identisch zu der Produktfunktion *Model Private Contract* aus Abschnitt 8.2. Der Ansatz für die modellbasierte Suche aus Kapitel 6 erlaubt jedoch nicht die Verwendung von negativen Anwendungsbedingungen oder negativen Nachbedingungen und nicht die Verwendung von Attributen. Weiterhin werden die visuellen Kontrakte auf der semantischen Ebene keinen Operationen mit Signaturen zugeordnet. Die visuellen Kontrakte sind lediglich über der Ontologie getypt. Zur Verwaltung mehrerer semantischer Kontrakte enthält die Visual Contract Workbench eine Liste mit visuellen Kontrakten, die über dem Hauptarbeitsbereich angeordnet ist (vgl. Abbildung 8.9). Ansonsten verhält sich die Oberfläche der Visual Contract Workbench zur Modellierung von semantischen Kontrakten wie in der Beschreibung der Produktfunktion *Model Private Contract*.

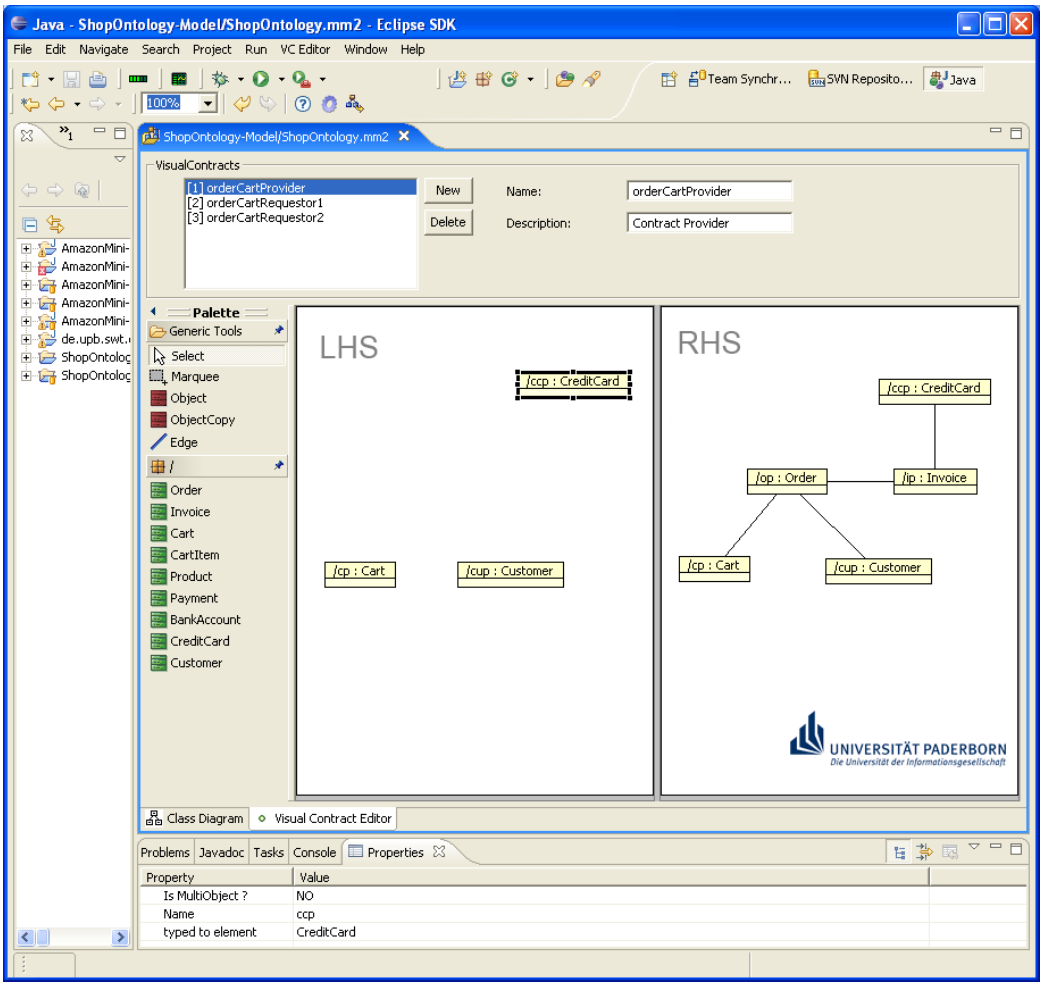


Abbildung 8.9: Visueller Kontrakt zur Beschreibung eines Service

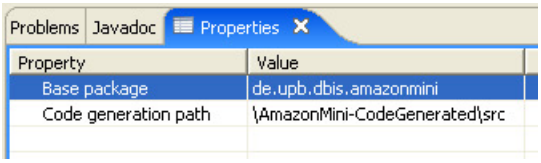


Abbildung 8.10: Pfad für Codegenerierung bestimmen

8.2.3 Codegenerierung

Mit der Produktfunktion *Generate Code* der Visual Contract Workbench kann ein Softwareentwickler aus den Modellen einer Anwendung Code generieren. Der Code besteht aus zwei Teilen. Erstens werden aus den UML-Klassen der Anwendung Java-Klassen generiert. Die Java-Klassen enthalten die spezifizierten Attribute und Methoden (jedoch keine Methodenrümpfe) sowie weitergehende Attribute und vollständige Methoden zur Verwaltung der Assoziationen. Bei der Codegenerierung kann sichergestellt werden, dass bereits manuell erstellter Code nicht überschrieben wird.

Zweitens werden zur Annotation der Operationen aus den visuellen Kontrakten JML-Assertions generiert, um den später hinzugefügten manuellen Code überprüfen zu können. Die generierten Code-Fragmente werden dann an den Programmierer übergeben, um den fehlenden Code zur Implementierung des Verhaltens der Anwendung zu integrieren.

Bevor der Softwareentwickler in der Visual Contract Workbench die Codegenerierung starten kann (Abbildung 8.11), muss er im *Properties View* das Zielprojekt[3] sowie das Zielpaket bestimmen (Abbildung 8.10). Danach kann die Codegenerierung über den entsprechenden Eintrag in der Menuleiste oder über einen Button in der Toolbar gestartet werden. Dabei kann der Entwickler wählen, ob er nur Java-Code aus dem Klassendiagramm oder auch zusätzlich JML-Annotationen erzeugen möchte (siehe Abbildung 8.12).

Die JML-Annotationen darf der Programmierer gemäß den Vorgaben in Abschnitt 2.5 nicht editieren. Daher werden die Java-Klassen und die JML-Annotationen in unterschiedlichen Dateien gespeichert, was von dem JML-Compiler auch unterstützt wird.

[3]In Eclipse können mehrere Entwicklungsprojekte gleichzeitig verwaltet werden. Der generierte Code muss sich nicht in einem gemeinsamen Projekt mit den Modellen befinden.

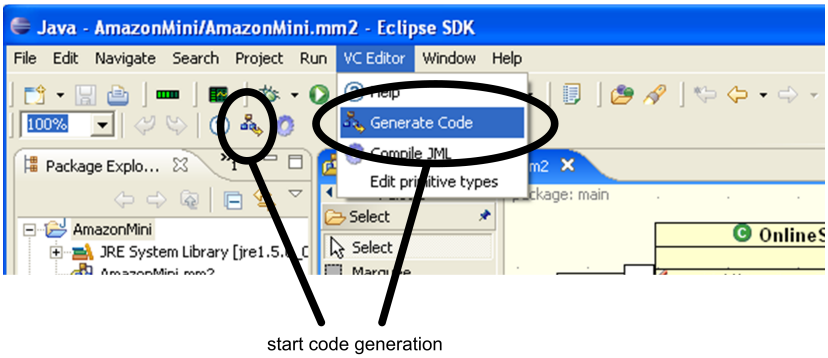


Abbildung 8.11: Java-Codegenerierung starten (1)

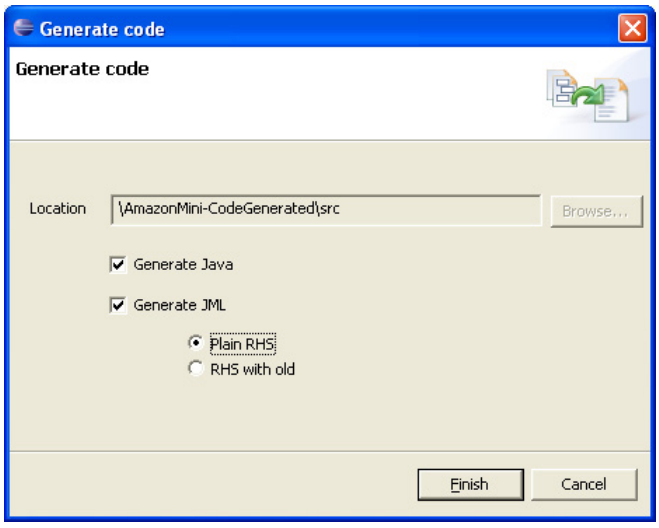


Abbildung 8.12: Java-Codegenerierung starten (2)

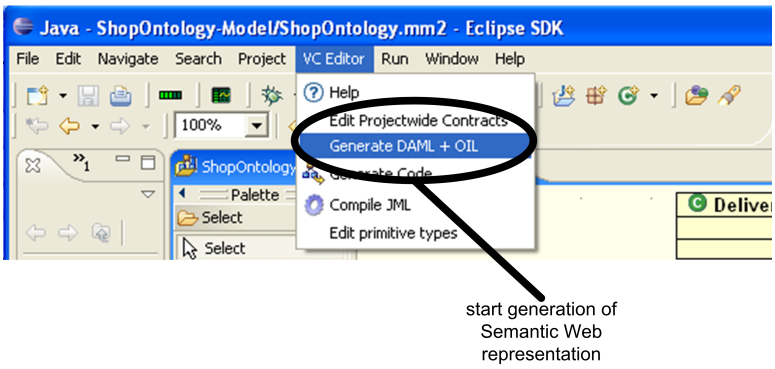


Abbildung 8.13: Start der Generierung der Semantic-Web-Repräsentation (1)

8.2.4 Generierung von Semantic-Web-Repräsentationen

Mit den Produktfunktionen *Generate DAML+OIL Ontology* bzw. *Generate Semantic Web Representation of Contracts* kann der Domänenexperte mit der Visual Contract Workbench eine DAML+OIL-Repräsentation der Ontologie generieren bzw. die visuellen Kontrakte in das in Abschnitt 7.3 beschriebene RDF-basierte Dateiformat exportieren.

Bevor der Domänenexperte die Semantic-Web-Repräsentation der Modelle generieren kann, muss er zunächst ein Zielverzeichnis bzw. ein Zielprojekt in der Eclipse-Umgebung als auch den Namensraum der modellierten Ontologie festlegen. Danach kann die Codegenerierung über einen entsprechenden Eintrag in der Menuleiste gestartet werden (Abbildung 8.13). Dabei kann der Domänenexperte wählen, ob er aus dem Klassendiagramm eine DAML+OIL-Ontologie oder eine Semantic-Web-Repräsentation der visuellen Kontrakte generieren möchte (siehe Abbildung 8.14).

8.2.5 Modelle als Vorlage verwenden

Mit den Produktfunktionen *View Private Contract* und *View Design Class Diagram* kann der Programmierer die Modelle zur Spezifikation des zu erstellenden Systems einsehen. Insbesondere muss der Programmierer bei der Implementierung der Methodenrümpfe die visuellen Kontrakte als Referenz für das Verhalten der Operationen verwenden. In unserer Visual Contract Workbench werden diese Funktionalitäten über die Produktfunktionen *Model Design Class Diagram* bzw. *Model Private Contract* abgedeckt. Die aktuelle Implementierung unterscheidet nicht zwischen den Rollen des Entwicklers

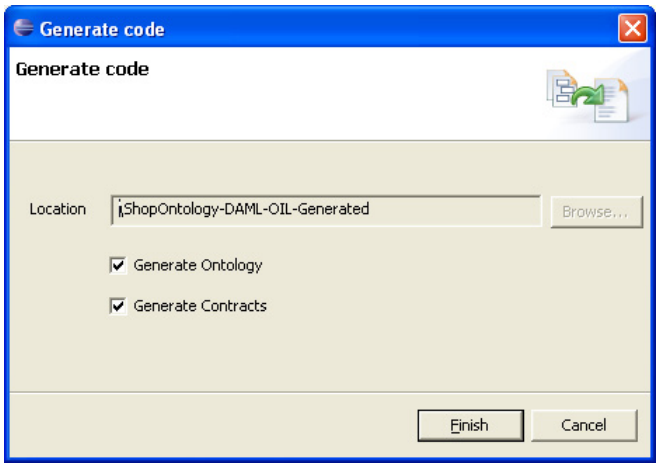


Abbildung 8.14: Start der Generierung der Semantic-Web-Repräsentation (2)

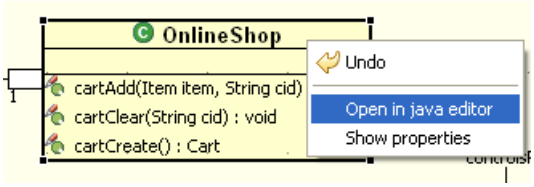


Abbildung 8.15: Öffnen des Java-Editors

und Programmierers.

Damit ein Programmierer leichter die zu implementierenden Codestellen findet, kann er das Klassendiagramm in der Visual Contract Workbench zu Navigation benutzen. Wenn er auf einer Klasse oder einer Operation die rechte Maustaste drückt, wird ein kontextsensitives Menu mit einem Menupunkt *Open in Java Editor* geöffnet (Abbildung 8.15). Bei der Betätigung dieses Menupunkts wird ein Java-Editor-Fenster geöffnet. Der Cursor wird in der Textdatei dabei direkt auf der ausgewählten Klasse bzw. der ausgewählten Operation plaziert. Weiterhin wird die angewählte Klasse bzw. Operation im Editor hervorgehoben dargestellt. In Abbildung 8.16 ist zum Beispiel über das Klassendiagramm die Operation `cartAdd` angewählt worden.

8.2.6 Bearbeiten des Java-Codes

Mit der Produktfunktion *Edit Java Classes* kann ein Programmierer den Source Code des zu erstellenden Systems editieren, um ein vollständiges, lauffähiges System zu erzeugen. Insbesondere muss der Programmierer das

Verhalten der modellierten Operationen implementieren. Weiterhin darf der Programmierer den existierenden Klassen neue Operationen hinzufügen oder dem System weitere Klassen hinzufügen.

Die Abbildung 8.16 zeigt einen Screenshot des Java-Editors. Es handelt sich hierbei um den Standard-Java-Editor von Eclipse. Der generierte Source Code enthält *Tasks* für den Programmierer (gekennzeichnet mit dem Schlüsselwort *ToDo*). Die Tasks werden bei der Codegenerierung dem Java-Code hinzugefügt, um zu kennzeichnen, welche Operationen ein Progammierer implementieren muss. Die Tasks werden von Eclipse in einer Taskliste (untere Teil in Abbildung 8.16) verwaltet. Weiterhin wird allen automatisch generierten Operationen ein Kommentar mit dem Schlüsselwort *generated* hinzugefügt.

8.2.7 Starten des Compilers

Mit den Produktfunktionen *Compile Java Classes* und *Compile Java Classes including JML* kann ein Programmierer den Source Code einer Anwendung — bestehend aus manuellem und automatisch generiertem Code — compilieren, um ausführbaren Java-Bytecode zu erzeugen. Wählt der Programmierer die Produktfunktion *Compile Java Classes* wird nur aus den Java-Klassen der Anwendung Bytecode erzeugt, mit der Produktfunktion *Compile Java Classes including JML* wird zusätzlich Code zur Überprüfung der JML-Assertions in den Bytecode eingefügt.

Wenn nur die Java-Klassen compiliert werden sollen, kann der in Eclipse verwendete Java-Compiler gestartet werden. Wenn zusätzlich aus den JML-Assertions Code generiert werden soll, muss der JML-Compiler verwendet werden. Der JML-Compiler kann aus der Visual Contract Workbench über den entsprechenden Eintrag der Menuleiste oder über einen Button in der Toolbar gestartet werden (vgl. Abbildung 8.17). Der Bytecode zum Überprüfen der JML-Assertions und damit der Code zum Überprüfen der visuellen Kontrakte wird automatisch von dem JML-Compiler erzeugt.

8.3 Architektur der Visual Contract Workbench

In diesem Abschnitt wird die Architektur der Visual Contract Workbench auf einer hohen Abstraktionsebene anhand Abbildung 8.18 beschrieben. Eine detaillierte Beschreibung einer frühen Version der Visual Contract Workbench

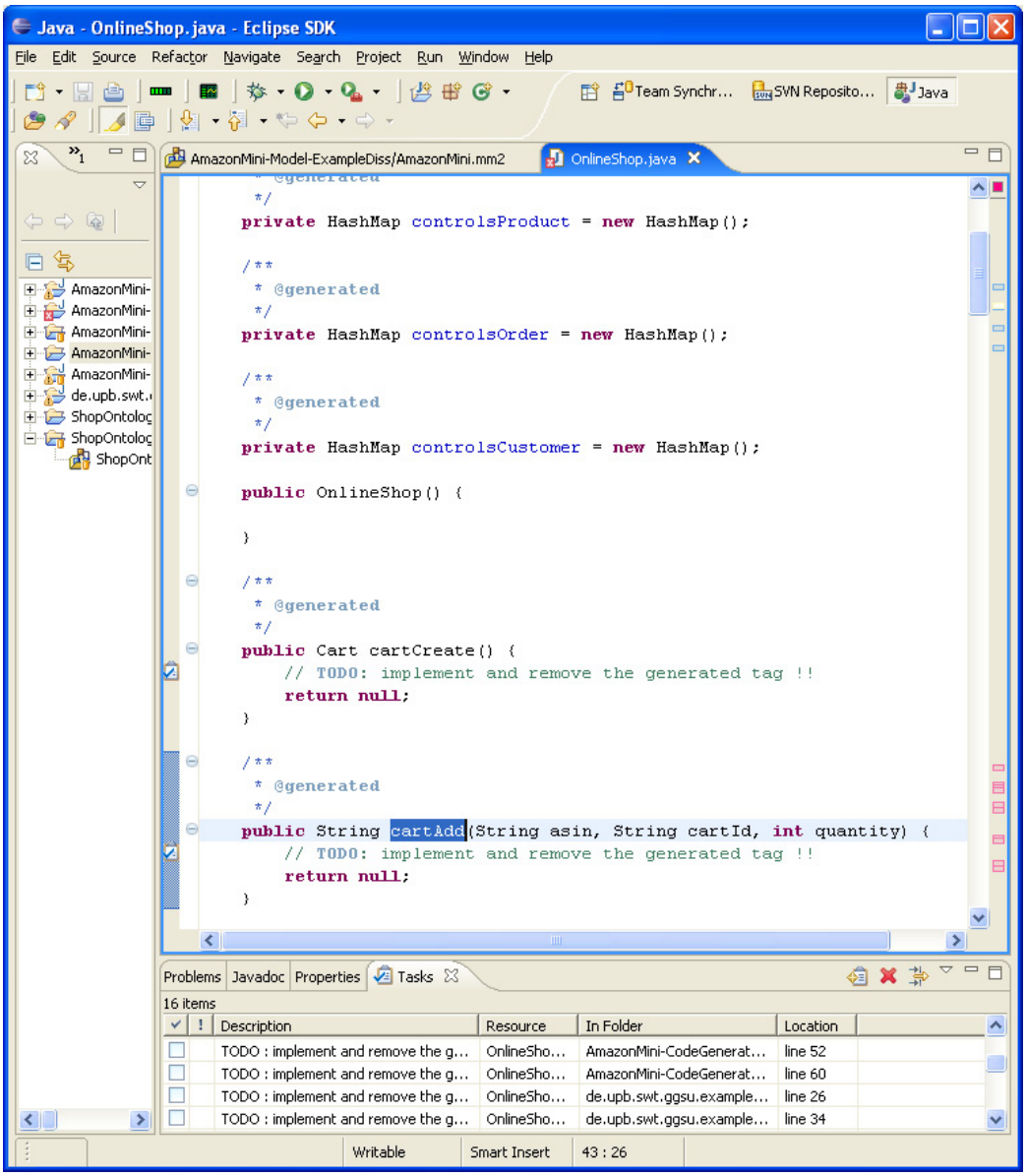


Abbildung 8.16: Editor für Source Code

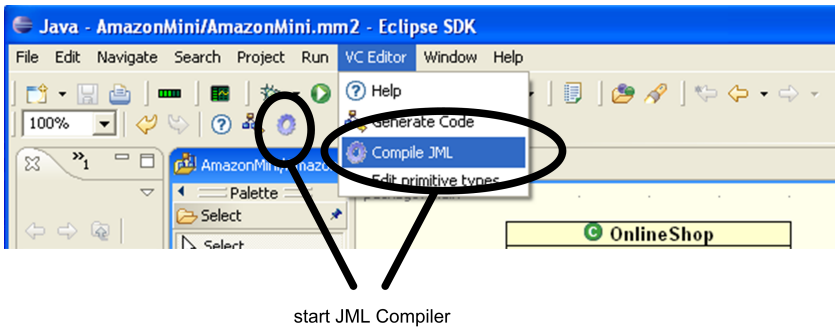


Abbildung 8.17: JML-Compiler starten

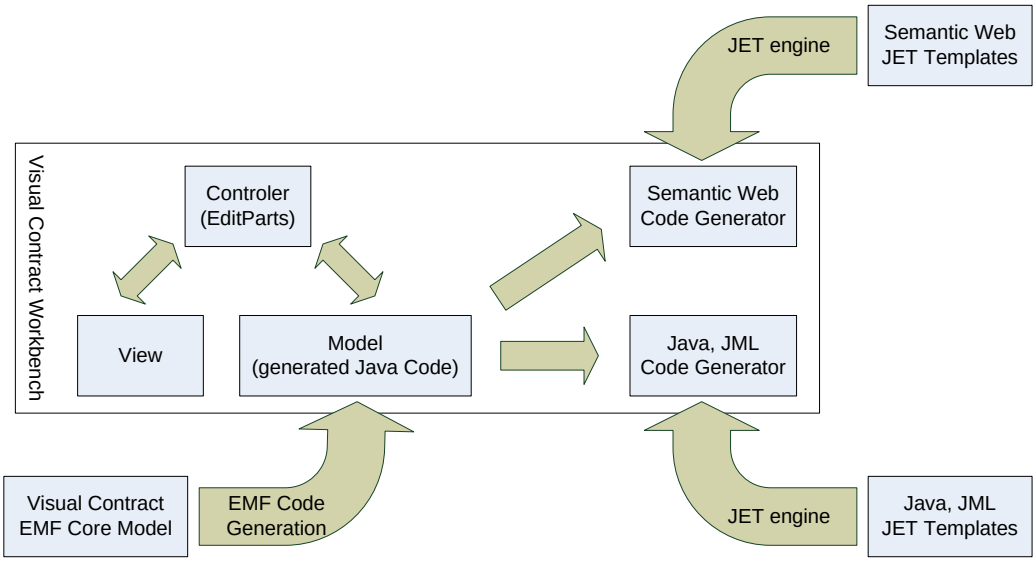


Abbildung 8.18: Architektur der Visual Contract Workbench

findet sich in [Wag05]. Die Visual Contract Workbench ist ein Eclipse-Plugin, das auf der Basis von GEF und EMF entwickelt wurde. Die Verwendung des GEF-Frameworks erfordert eine Unterteilung des Plugins in die drei Komponenten Model, View und Controller. Zur Realisierung der Codegenerierung werden zwei weitere Komponenten benötigt.

Die Entwicklung unserer Workbench haben wir mit dem Design eines EMF-Core-Modells begonnen. Das EMF-Core-Modell ist ein Metamodell zur Beschreibung unserer Modelle (Klassen und visuelle Kontrakte). Das EMF-Core-Modell entspricht im Wesentlichen dem in Kapitel 3 eingeführten Metamodell für visuelle Kontrakte bis auf einige Vereinfachungen oder Anpassungen aufgrund der technischen Vorgaben von EMF. Abbildung 8.19 zeigt den

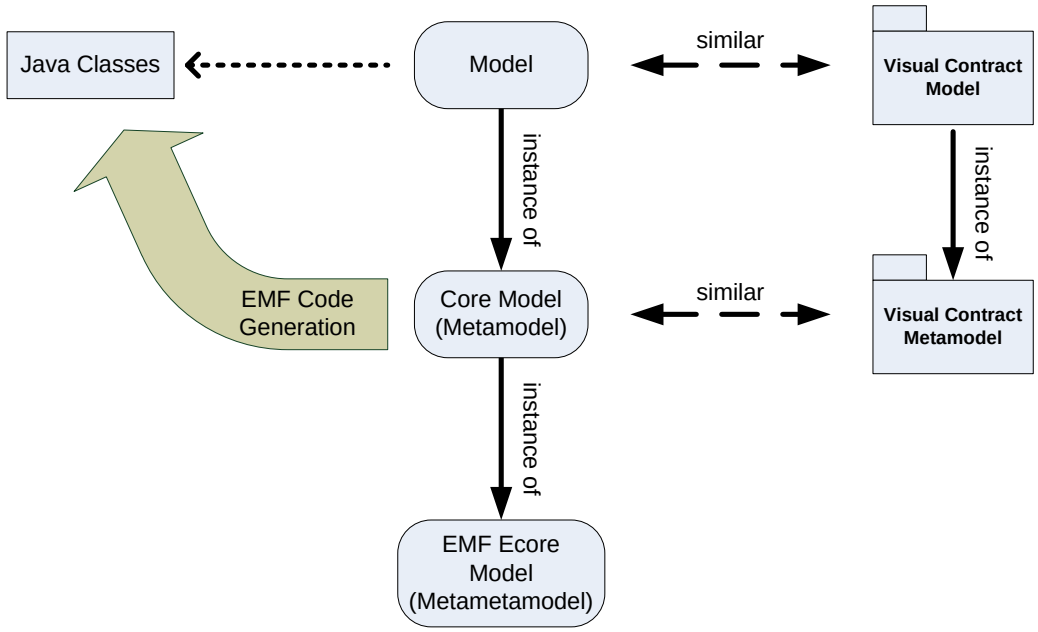


Abbildung 8.19: Zusammenhang EMF-Core-Modell und Metamodell für visuelle Kontrakte

Zusammenhang zwischen einem EMF-Core-Modell und dem Metamodell für visuelle Kontrakte aus Kapitel 3. Aus diesem EMF-Core-Modell haben wir die Komponente zur Verwaltung unserer Modelle mit dem Eclipse Modeling Framework generiert.

Zur Realisierung der View-Komponente sind verschiedene Draw2D Figures zur Darstellung der Modellelemente in den Klassendiagrammen und visuellen Kontrakten implementiert worden. Die Controller-Komponente enthält für nahezu jede Klasse aus der Modell-Komponente eine entsprechende `EditParts`-Klasse.

Zur Realisierung der Komponenten für die Codegenerierung haben wir eine Reihe von JET-Templates entwickelt. Als Vorlage für die JET-Templates zur Generierung von JML dienten die in Kapitel 5 eingeführten Compound Rules. Insbesondere die Zielregel einer Compound Rule ist eine Vorlage für ein JET-Template. Jedoch mussten hierzu einige Anpassungen vorgenommen werden. So mussten z.B. die Variablen, die eine Zielregel in einer Compound Rule mit einer Quellregel verknüpfen, in den JET-Templates durch Methodenaufrufe, die auf das Modell zugreifen, ersetzt werden. Aufgrund der Komplexität der Codegenerierung mussten Teile der Codegenerierung jedoch manuell implementiert werden. Insbesondere die Kontrollstrukturen der

Transformation aus Kapitel 5 und die zusätzlich eingeführten Objekte (um bereits bearbeitete Modellelemente zu markieren) konnten nicht direkt mit JET-Templates umgesetzt werden. Im Folgenden skizzieren wir kurz das Vorgehen zur Erzeugung der Komponente zur Generierung von Java-Code und JML-Annotationen. Die Komponente zur Generierung der Semantic-Web-Repräsentationen funktioniert ähnlich.

Die Komponente *Java, JML-Code-Generator* besteht aus zwei Teilen. Zum einen muss aus einem Klassendiagramm Java-Code erzeugt werden und zum anderen müssen aus den visuellen Kontrakten JML-Annotationen erzeugt werden.

Die Codegenerierung der Java-Klassen kann vollständig mit ineinander geschachtelten Templates realisiert werden. Listing 8.1 zeigt das Jet-Template zum Starten der Codegenerierung für Java-Klassen. Der rekursive Aufruf eines weiteren Jet-Templates erfolgt mit dem Befehl `<%@ include` (siehe z.B. Zeile 17).

In JET-Templates existieren weiterhin drei verschiedene Arten von Ausdrücken (*Directives*, *Expression* und *Scriptlets*), die für das Verständnis von Listing 8.1 notwendig sind. Zur Übersetzung des Templates in eine Java-Klasse (vgl. Abschnitt 8.1.3) benötigt die JET-Engine gewisse Umgebungsparameter, wie z.B. das Java-Paket in dem die übersetzte Implementierungsklasse abgelegt werden soll. Diese werden mit Hilfe einer Direktive am Beginn eines Templates gesetzt. Eine Direktive wird mit `<%@jet` eingeleitet.

JET-Expressions ermöglichen die Ausgabe von Werten einer Variable oder Methode. Eine JET-Expression wird mit `<%=` eingeleitet. In Listing 8.1 wird z.B. in Zeile 14 mit dem Ausdruck `<%= umlClass.getName() %>` der Name der aktuell bearbeiteten Klasse in das generierte Ziel eingesetzt.

Mit Scriptlets können Java-Code-Fragmente in JET-Templates eingefügt werden, wodurch eine größere Flexibilität bei der Codegenerierung gegeben ist. Scriptlets werden mit `<%` eingeleitet und mit `%>` abgeschlossen (siehe Zeilen 7-12).

Die Generierung der JML-Annotationen ist aufgrund der Teilgraphensuche, die mit den JML-Ausdrücken realisiert werden müssen, komplizierter als die Generierung der Java-Klassen. Eine einfache Schachtelung von JET-Templates ist hier nicht mehr ausreichend. Die notwendigen, komplexeren Kontrollstrukturen zur Generierung der JML-Annotationen werden mit Java-Klassen manuell implementiert und nicht aus den JET-Templates generiert. Die manuell implementierten Klassen realisieren hauptsächlich die Generie-

```
1 <%@ jet package="de.upb.swt.gssu.codegeneration"
2     class="GenerateClassForVC"
3     imports="de.upb.swt.ggsu.codegeneration.wrapper.*
4                         de.upb.swt.ggsu.codegeneration.*
5                           de.upb.swt.ggsu.model.*
6                           java.util.*" %>
7 <%
8 Iterator it;
9 CodeGenerationArgument gab =
10              (CodeGenerationArgument) argument;
11 UMLClass umlClass = (UMLClass)gab.getGraphObject();
12 %>
13
14 public class <%= umlClass.getName() %>
15 <%= gab.getExtendsAndImplementsString() %> {
16
17   <%@ include file="classAttributeDef.javajet" %>
18
19   <%@ include file="classAssociationDef.javajet" %>
20
21    public <%= umlClass.getName() %>() {
22
23    }
24
25   <%@ include file="classMethod.javajet" %>
26
27   <%@ include file="classAttributeMethod.javajet" %>
28
29   <%@ include file="classAssociationMethod.javajet" %>
30
31 }
```

Listing 8.1: JET-Template zur Generierung einer Java-Klasse

```
1 /*@ also
2         @ public normal_behavior
3         <%
4     //
5     for (VisualContractWrapper vc :
6     currentOperation.getVisualContracts())
7     {
8         %>
9     @ requires <%= vc.getRequires() %>
10
11    @ ensures <%= vc.getEnsures() %><%
12    }
13    %>
14   @*/
15 <%  OperationWrapper method = currentOperation; %>
16 <%@ include file="class_method_signature.javajet" %>;
```

Listing 8.2: JET-Template zur Generierung einer JML-Annotation

rung des Codes für die Teilgraphensuche. Zur Generierung ähnlicher JML-Strukturen können aus den manuell implementierten Klassen jedoch wiederum JET-Templates aufgerufen werden und umgekehrt können aus JET-Templates manuell implementierte Klassen aufgerufen werden.

Listing 8.2 zeigt ein Beispiel zur Generierung einer JML-Annotation für eine Operation. Das Template generiert als erstes Java-Kommentare, in welche die JML-Annotation eingebettet werden kann. Die Überprüfung der Vorbedingung (Nachbedingung) erfordert die Generierung komplexer Pfadausdrücke. Daher wird zur Generierung der Vorbedingung (Nachbedingung) aus dem JET-Template heraus die Operation `getRequires` (`getEnsures`) aufgerufen. Diese manuell implementierte Operation ruft wiederum andere manuell implementierte Operationen oder JET-Templates auf.

8.4 Fazit

In diesem Kapitel haben wir ein Werkzeug vorgestellt, um die in dieser Arbeit vorgestellten modellbasierten Vorgehensweisen zur Entwicklung von Softwarekomponenten für den Einsatz in serviceorientierten Architekturen zu unterstützen. Als erstes haben wir in diesem Kapitel die technologische Basis zur Realisierung unserer Visual Contract Workbench beschrieben. Da-

nach haben wir die notwendigen Produktfunktionen zur Unterstützung unserer Vorgehensweisen beschrieben und dokumentiert, wie diese in unserer Workbench umgesetzt werden. Zum Abschluss dieses Kapitels haben wir die Architektur der Visual Contract Workbench erläutert.

Die Visual Contract Workbench ist ein Eclipse Plugin, das mit Java 5.0 implementiert wurde. Das Plugin ist für die Eclipse Version 3.1.1 entwickelt worden. Der Code zur Verwaltung von Modellinformationen ist mit dem Eclipse Modeling Framework in der Version 2.1.1 aus einem entsprechendem EMF-Core-Modell generiert worden. Der graphische Teil der Visual Contract Workbench ist mit dem Graphical Editing Framework in der Version 3.1.1 implementiert worden. Der von der Workbench generierte Java-Code ist kompatibel zu Java 1.4 und kann einschließlich der generierten JML-Annotationen mit dem JML-Compiler in der Version 5.2 compiliert werden. Der JML-Compiler setzt dazu ein JSDK in der Version 1.4.1 oder 1.4.2 voraus. Der generierte DAML+OIL-Code ist kompatibel zur DAML+OIL Spezifikation vom März 2001 [CvHH+01].

Da es sich bei der Visual Contract Workbench nur um einen Forschungsprototypen handelt, sind nicht alle Konzepte in der aktuellen Implementierung verfügbar. So können zum Beispiel zwar visuelle Kontrakte mit Multiobjekten erstellt werden, eine entsprechende Codegenerierung wird jedoch noch nicht unterstützt. In zukünftigen Entwicklungen muss zudem die Bedienbarkeit der Visual Contract Workbench verbessert werden oder sie muss um eine Reengineering-Komponente ergänzt werden, falls ein bestehendes System weiterentwickelt werden soll. Trotz dieser Einschränkungen hat die Visual Contract Workbench die Umsetzbarkeit unseres Ansatzes gezeigt.

Kapitel 9

Zusammenfassung und Ausblick

In diesem Kapitel fassen wir als erstes die Ergebnisse dieser Arbeit zusammen. Danach beschreiben wir eine Evaluation von Teilen unseres Ansatzes, die wir im Rahmen einer Kooperation mit einem industriellen Partner durchführen konnten. Weiterhin geben wir einen Ausblick über mögliche Erweiterungen des hier entwickelten Ansatzes bevor wir die Arbeit mit einem Schlusswort abschließen.

9.1 Zusammenfassung der Ergebnisse der Arbeit

In dieser Arbeit haben wir einen Ansatz zur kontraktbasierten Modellierung, Implementierung und Suche in serviceorientierten Architekturen beschrieben. In diesem Ansatz berücksichtigen wir die zentrale Rolle der Softwareentwickler und der öffentlichen Servicebeschreibungen bei der Implementierung und Suche eines Service. Insbesondere erlauben wir in unserem Ansatz eine modellbasierte, semantische Beschreibung, die Überwachung der Korrektheit von Services und den Vergleich existierender Servicebeschreibungen. Hierzu basiert unser Ansatz auf drei wesentlichen Techniken.

Die Basis unseres Ansatzes ist die Verwendung visueller Kontrakte zur Spezifikation der Semantik von Services. Ein visueller Kontrakt besteht aus einem Paar von UML-Composite-Structure-Diagrammen. Mit der Verwendung von UML-Diagrammen benutzen wir eine Notation, die Softwareentwicklern vertraut ist und einfach in heutige Softwareentwicklungsprozesse zu integrieren ist.

Mit der neuen, innovativen Technik des Model-Driven Monitorings unterstützen wir die Überwachung der Korrektheit einer manuell erstellten Implementierung. Hierzu betten wir die visuellen Kontrakte in einen modellbasierten Softwareentwicklungsprozess ein und erlauben die Generierung von Runtime-Assertions aus den visuellen Kontrakten, welche die Korrektheit eines Systems gegenüber seiner Spezifikation mit visuellen Kontrakten zur Laufzeit überwachen.

Die modellbasierte Suche eines Service Requestors nach existierenden Services unterstützen wir mit der neuen, innovativen Technik des Model-Driven Matchings. Hierzu ermöglichen wir einem Service Requestor die Verwendung der visuellen Kontrakte zur Erstellung einer Suchanfrage und definieren einen automatisierbaren Kompatibilitätsbegriff zum Vergleich einer öffentlichen Servicebeschreibung mit einer Suchanfrage.

9.2 Evaluation

Im Rahmen eines Gemeinschaftsprojektes des Software Quality Lab (s-lab), der sd&m AG und des Fachgebietes Datenbank- und Informationssysteme des Institutes für Informatik der Universität Paderborn hatten wir die Möglichkeit, die praktische und wirtschaftliche Einsetzbarkeit verschiedener Bestandteile unseres Ansatzes im Rahmen einer realistischen, industriellen Fallstudie zu untersuchen [EGJ+06, LRE+06]. Als Fallbeispiel wurde in dieser Studie ein Prozess zum Abschluss eines Versicherungsvertrages ausgewählt. Die Erfahrungen der sd&m AG aus Projekten in Versicherungsgesellschaften haben sichergestellt, dass ein realistisches Szenario entstanden ist. Ziel des Projektes war die Evaluierung visueller Kontrakte in einem praktischen Kontext, um übertragbare und verwertbare Ergebnisse für deren Einsatz in einer serviceorientierten Architektur zu erzielen.

Der Hintergrund dieser Fallstudie ist, dass die sd&m AG sich im Rahmen der Fortentwicklung existierender Anwendungslandschaften intensiv mit serviceorientierten Architekturen beschäftigt. Hierbei wird in den bisherigen Projekten die Semantik von zu entwickelnden Enterprise Services[1] vorwiegend natürlichsprachlich festgehalten. Diese gängige Herangehensweise ist in der Praxis mit Risiken und hohen Aufwänden verbunden. In der Regel ist es notwendig, dass viele abstimmende Gespräche geführt werden müssen,

[1] Ein Enterprise Service ist ein Service, der eine einzelne Aktivität oder mehrere Aktivitäten in einem Geschäftsprozess unterstützt. Zur Realisierung eines Enterprise Service müssen die in einer Anwendungslandschaft existierenden Systeme integriert werden.

bis eine Formulierung der Schnittstellenbeschreibung gefunden wird, die korrekt im Sinne der Geschäftslogik ist und keine unzulässigen Spielräume für Interpretationen offen lässt. Dabei entstehen häufig umfangreiche Dokumente, deren Konsistenz im größeren Projektkontext kaum sichergestellt werden kann und die bei sich ändernden Anforderungen aufwendig angepasst werden müssen. Im Kontext einer serviceorientierten Architektur verschärft sich dieses Problem durch die zentrale Rolle, welche die Schnittstellenvereinbarungen einnehmen.

Der steigende Aufwand zur Verwaltung einer immer größeren Menge von Enterprise Services sowie der Wunsch nach einer werkzeuggestützten, automatisierbaren Verwaltung von Services hat auf der Seite der sd&m AG den Ausschlag gegeben, nach alternativen Beschreibungsmöglichkeiten zu suchen. Dabei bestehen seitens der sd&m AG verschiedene Anforderungen. Es wird eine intuitive, visuelle Notation bevorzugt, da bei der fachlichen Gestaltung einer SOA die Kunden sehr stark integriert sind. Diese sollte auf einer bekannten Notation wie z.B. der UML beruhen, um die Einarbeitungszeit der Softwareentwickler möglichst gering zu halten. Die Notation soll mächtig genug sein, um die relevanten Inhalte der typischen textuellen Dokumente der sd&m AG zur Beschreibung der Semantik der Entprise Services präzise ausdrücken zu können. Weiterhin soll auch überprüft werden, in wie weit es möglich ist, das Verhalten existierender, technischer Schnittstellen einer Anwendung vollständig zu beschreiben. Eine Turing-Vollständigkeit ist jedoch explizit nicht gefordert, da die Zielsetzung sich auf eine Unterstützung der Spezifikation beschränkt. Durch den Einsatz der Notation sollen auch Konsistenzprobleme früh im Projektverlauf entdeckt werden können. Die hohe Zahl der zu verwaltenden Services erfordert zudem für die Suche von Services geeignete Methodiken.

Die intuitive Notation unserer visuellen Kontrakte und die Verwendungen von bereits zuvor bekannten Diagrammen der UML stießen bei der sd&m AG auf großes Interesse. Die Mitarbeiter der sd&m AG, die mit dieser speziellen Modellierungsmethode zu Beginn unserer Fallstudie nicht vertraut waren, konnten schon nach kürzester Zeit visuelle Kontrakte korrekt interpretieren und erstellen. Auch wenn die Methodik noch nicht in einem Kundenprojekt eingesetzt wurde, ist die Einschätzung der sd&m-Mitarbeiter, dass die Kunden die visuellen Kontrakte bis zu einer gewissen Komplexitätsstufe verstehen werden.

Im Allgemeinen sind visuelle Kontrakte gut geeignet, um strukturelle Änderungen von Systemzuständen, beispielsweise das Löschen oder Erzeugen von Objekten, sowie das Knüpfen und Lösen von Beziehungen zwischen Objekten

zu beschreiben. In unserer Fallstudie hat sich herausgestellt, dass diese Art der Beschreibung auf der Ebene der Enterprise Services ausreichend ist. Das heißt, ein Großteil der textuellen Beschreibungen der Aktivitäten aus dem Fallbeispiel der Fallstudie konnte mit unseren visuellen Kontrakten wiedergegeben werden.

Bei dem Versuch die Semantik existierender, technischer Systeme, auf die z.B. zur Realisierung eines Enterprise Service zurückgegriffen werden kann, mit visuellen Kontrakten zu beschreiben haben sich unterschiedliche Ergebnisse gezeigt. Während es teilweise noch möglich war, die Semantik der Systeme mit visuellen Kontrakten ausreichend zu beschreiben, hat sich insbesondere bei der detaillierten, vollständigen Beschreibung der Operationen eines Partnersystems zur Beseitigung von Dubletten gezeigt, dass bei Schnittstellen mit einer sehr hohen Komplexität visuelle Kontrakte nicht ausreichend sind. Für eine vollständige Beschreibung der Schnittstellen existierender Systeme müssen die visuellen Kontrakte z.B. um Prozessbeschreibungen ergänzt werden. Bei diesen Problemen muss man sich jedoch auch noch einmal vor Augen führen, was das Ziel der visuellen Kontrakte ist: Das Ziel ist es, das Verhalten eines Service auf einer geeigneten Abstraktionsebene zu beschreiben. Es ist nicht das Ziel, die Implementierung eines Service vollständig zu beschreiben.

Weiterhin hat sich gezeigt, dass durch den Einsatz der visuellen Kontrakte die Qualität der verwendeten Modelle und Beschreibungen gestiegen ist. Bei der Erstellung der visuellen Kontrakte sind Fehler in den Modellen zur Beschreibung der Enterprise Services und in den Beschreibungen der Geschäftsaktivitäten sowie Inkonsistenzen zwischen beiden aufgefallen. Auch offene Fragestellungen sind deutlich geworden, die ansonsten erst bei der Implementierung eines Prozesses und damit sehr spät im Entwicklungsprozess auftauchen.

Der Fokus der Fallstudie war die Evaluierung visueller Kontrakte in einem praktischen Kontext. Jedoch konnten wir auch das Model-Driven Matching ansatzweise evaluieren. Die hierbei gemachten Erfahrungen deuten daraufhin, dass unser Matching-Konzept für die Suche nach (Enterprise) Services in einer serviceorientierten Architektur geeignet ist.

Die Fallstudie zeigt also ein durchaus positives Bild für den Einsatz unserer visuellen Kontrakte und des Model-Driven Matchings zur Beschreibung und zur Suche von Services. Es hat sich gezeigt, dass die visuellen Kontrakte in einem realen Anwendungsumfeld für serviceorientierte Architekturen praktisch und wirtschaftlich einsetzbar sind.

Das Model-Driven Monitoring ist nicht im Rahmen dieser Fallstudie unter-

sucht worden. Jedoch ist auch diese Technik während der Fallstudie und in späteren Diskussionsrunden bei der sd&m AG vorgestellt worden. Insbesondere die Idee, dass die Korrektheit einer manuellen Implementierung gegenüber einer unvollständigen Spezifikation überwacht werden kann, fand bei der sd&m AG großes Interesse. Ein Grund ist, dass im Rahmen der Fortentwicklung existierender Anwendungslandschaften aufgrund des enormen Zeitdrucks nicht die Möglichkeit besteht, die Semantik eines Systems vollständig zu spezifizieren. Nichtsdestotrotz soll die Korrektheit ausgewählter Services überwacht werden können. Eine Aussage der sd&m AG ist, dass die Technik des Model-Driven Monitoring in Zukunft weiter verfolgt werden soll. Jedoch ist hierzu eine Umsetzung des Model-Driven Monitoring mit aktuellen, in der sd&m AG verwendeten CASE-Tools notwendig.

Neben diesen eher konzeptionellen Ergebnissen, haben wir im Rahmen dieser Arbeit die Visual Contract Workbench entwickelt. Die Visual Contract Workbench erlaubt die Modellierung von Klassendiagrammen und visuellen Kontrakten sowie eine anschließende Codegenerierung nach Java bzw. JML und nach DAML+OIL. Mit diesem prototypischen Werkzeug kann unser Ansatz evaluiert und Softwareentwicklern näher gebracht werden.

9.3 Ausblick

Der beschriebene Ansatz kann in verschiedener Hinsicht erweitert werden. So ist es zum Beispiel aus Sicht eines Service Providers nicht immer erwünscht, feingranulare Servicebeschreibungen auf der Ebene von Operationen zu veröffentlichen. Stattdessen sollen auf einem Discovery Service eventuell semantische Beschreibungen zu Prozessen hinterlegt werden. Setzt sich ein Prozess aus einer Folge von Operationen zusammen, die einzeln mit visuellen Kontrakten spezifiziert wurden, so muss dem Service Provider eine methodische Unterstützung zur Bestimmung der Prozessbeschreibung aus den einzelnen semantischen Beschreibungen der Operationen zur Verfügung gestellt werden.

In Abschnitt 4.4 haben wir gezeigt, wie das Werkzeug JMLUnit zum Testen einer Anwendung eingesetzt werden kann. Neben dem Tool JMLUnit existieren diverse weitere JML-Tools, welche zum Teil sogar eine statische Verifikation ermöglichen. Gegenüber testbasierten Ansätzen haben statische Verifikationen z.B. den Vorteil, dass sie für alle möglichen Ausführungspfade einer manuellen Implementierung überprüfen, ob diese korrekt gegenüber einer JML-Spezifikation sind. Bei testbasierten Ansätzen werden nur die Aus-

führungspfade überprüft, die in den Testfällen bedacht wurden. Allerdings ist bei der statischen Verifikation nicht immer entscheidbar, ob eine Implementierung korrekt gegenüber einer JML-Assertion ist. Daher existieren verschiedene JML-Tools (siehe [BCC+05] für einen Überblick), die verschiedene Grade der Automation für unterschiedlich ausdrucksstarke Spezifikationen anbieten. In weitergehenden Arbeiten kann z.B. überprüft werden, in wie weit die vorhandenen JML-Tools für eine statische Verifikation einsetzbar sind, um die Korrektheit einer manuellen Implementierung gegenüber unseren generierten JML-Assertions zu verifizieren.

In dieser Arbeit besitzen eine Ontologie, die zur Erstellung der semantischen, öffentlichen Servicebeschreibung verwendet wird und das Klassendiagramm der Implementierung eines Service Providers eine gemeinsame Menge von Klassen. Ist diese gemeinsame Basis nicht gegeben, so muss ein Softwareentwickler bei der Erstellung der semantischen Servicebeschreibungen weitergehend unterstützt werden. Erstens muss ein Softwareentwickler eine Relation zwischen den Begriffen seines internen Vokabulars (den Begriffen im Klassendiagramm für die Implementierung) und der Ontologie herstellen können. Hierzu können z.B. existierende Erweiterungen der UML zur Darstellung von Abbildungen zwischen zwei Klassendiagrammen verwendet werden [HK03]. Zweitens muss diese Relation verwendet werden können, um aus den semantischen Beschreibungen, die über dem Klassendiagramm der Implementierung getypt sind, semantische, öffentliche Beschreibungen zu berechnen, die über der Ontologie getypt sind. Hier sind automatisierte Vorgehensweisen denkbar, die z.B. auf Schemaevolutionstechniken (bekannt aus dem Bereich der objektorientierten Datenbanken [BKKK87, KE98]) beruhen.

Andere mögliche Erweiterungen unseres Ansatzes haben sich aus einer Kooperation mit der Arbeitsgruppe von Prof. Dr. Odej Kao ergeben. In der Arbeitsgruppe von Prof. Dr. Kao ist ein Verfahren zur Auffindung von Grid-Ressourcen, welches auf Description Logics (DL) basiert, entwickelt worden [HHK04]. Das Verfahren zeichnet sich insbesondere dadurch aus, dass einer Ontologie zur Laufzeit neue Konzepte hinzugefügt werden können. Mit einem DL-System kann zusätzlich überprüft werden, ob ein Konzept ein anderes umfasst, so dass auch mit neuen Konzepten beschriebene Ressourcen mit bestehenden Suchanfragen gefunden werden können. Ein erster Vergleich [Fre05] unseres Ansatzes mit dem aus der Arbeitsgruppe von Prof. Dr. Kao hat gezeigt, dass unser Ansatz insbesondere von den erweiterbaren Ontologien profitieren kann, während der Ansatz aus [HHK04] von einer geeigneten, intuitiven Darstellung ähnlich der visuellen Kontrakte profitieren könnte.

Neben diesen Beispielen für mögliche konzeptionelle Erweiterungen entwi-

ckeln sich die in dieser Arbeit verwendeten Technologien auch ständig weiter. Aktuelle JML-Versionen unterstützen bisher nur Java bis Version 1.4.2. Ein Nachteil der Java Versionen bis einschließlich Version 1.4.2 ist beispielsweise, dass die Datenstrukturen zu Verwaltungen von Kollektionen (Listen, Hashtables etc.) prinzipiell offen für jeden Typ sind, da sie Objekte vom allgemeinsten Typ `Object` beim Speichern entgegennehmen und diesen auch als Rückgabe liefern. Seit Java 1.5 macht z.B. auch die `Collection`-API massiven Gebrauch der Java Generics [Bra04]. Dadurch wird eine bessere Typsicherheit gewährleistet, da mit Generics Kollektionen auf einen spezifischen Typ beschränkt werden können. Eine Unterstützung von JML für Java 1.5 ist bereits angekündigt. Unsere Codegenerierung muss dann zur Unterstützung von Java 1.5 ebenfalls angepasst werden, insbesondere die typsicheren Kollektionen wären in unserem generierten Code ein Gewinn.

Die Berechnung unseres Model-Driven Matchings basiert auf DAML+OIL. DAML+OIL besitzt jedoch bereits einen offiziellen Nachfolger (Web Ontology Language — OWL), der vom World Wide Web Consortium standardisiert wurde. Unsere Codegenerierung zur Repräsentation von Ontologien und visuellen Kontrakten in einem XML/RDF-basierten Dateiformat ist an diese Entwicklung anzupassen. Die Auswirkungen auf unseren Algorithmus zur Berechnung des Matching werden jedoch eher gering sein, da die Implementierung des Matching-Algorithmus auf RDF basiert und sowohl DAML+OIL also auch OWL auf RDF basieren.

9.4 Abschluss

Serviceorientierte Architekturen bieten ein großes Potential zur Integration von IT-Sytemen innerhalb und außerhalb eines Unternehmens. Diverse Veröffentlichungen und Migrationsprojekte zur Umsetzung serviceorientierter Architekturen zeigen, dass deren Potential sowohl im universitären Bereich als auch in der Industrie erkannt wurde.

Weiterhin leisten verschiedene Projekte aus Forschung und Industrie einen Beitrag, um den Erfolg serviceorientierter Architekturen zu erhöhen. Was uns hauptsächlich von existierenden Ansätzen unterscheidet ist, dass wir die Erstellung öffentlicher Beschreibungen eines Service nicht losgelöst von heutigen modellbasierten, objektorientierten Softwareentwicklungsprozessen betrachten. Weiterhin berücksichtigen wir, dass in heutigen Projekten der Softwareentwickler sowohl bei der Spezifikation als auch bei der Implementierung eines Service immer noch einen entscheidenden Erfolgsfaktor darstellt.

Anhang A

Beschreibung der Metamodellklassen zur Repräsentation visueller Kontrakte

Im Folgenden werden die Klassen des Metamodells für visuelle Kontrakte detailliert beschrieben. Die Beschreibung erfolgt in einem standardisierten Format, das dem Format der UML-Spezifikation zur Beschreibung von Metamodellklassen ähnelt. Die in diesem Kapitel verwendeten Referenzen auf die UML-Spezifikation beziehen sich auf das Dokument „Unified Modeling Language: Superstructure Version 2.0, Revised Final Adopted Specification (ptc/04-10-02)“ vom 8. Oktober 2004 [OMG04].

A.1 Klasse Constraint

Ein Constraint gehört zu einem `VCElement` (zu einer Rolle). Ein Constraint spezifiziert für eine Rolle den Inhalt eines spezifischen `StructuralFeature`, d.h. für ein Attribut. Bei dem Feature muss es sich um ein strukturelles Feature der Klasse handeln, über welches die Rolle getypt ist.

Generalisierung

- Constraint aus dem Paket Kernel, UML-Spezifikation, S. 56

Attribute Diese Metamodellklasse enthält keine zusätzlichen Attribute.

Assoziationen

- `definingFeature:StructuralFeature[0..1]` — Mit der Referenz auf die Metamodellklasse `StructuralFeature` wird spezifiziert, welche Inhalte ein Constraint referenzieren darf, da mit dieser Assoziation ein Attribut einer Klasse (über welche die Rolle getypt ist) referenziert wird und dieses Attribut einen Typ besitzt.
- `owningConnectableElement:VCElement[1]` — Die Rolle zu der dieser Constraint gehört. Submenge von `Element::owner` aus UML 2.0 Paket `Kernel`.
- `postElement:VCElement[0..*]` — Ordnet einem spezifischen Feature einer Rolle ein oder mehrere `ConnectableElement`s als Inhalt zu. Diesen Inhalt muss eine Instanz, welche zur Laufzeit die Rolle einnimmt, die diesen Constraint besitzt, nach der Ausführung der Operation für das spezifische Feature besitzen. Das referenzierte Element muss denselben Typ haben wie vom `StructuralFeature` vorgeschrieben.
- `postNCElement:VCElement[0..*]` — Ordnet einem spezifischen Feature einer Rolle ein oder mehrere `ConnectableElement`s als Inhalt zu. Diesen Inhalt darf eine Instanz, welche zur Laufzeit die Rolle einnimmt, die diesen Constraint besitzt, nach der Ausführung der Operation für das spezifische Feature nicht besitzen. Das referenzierte Element muss denselben Typ haben wie vom `StructuralFeature` vorgeschrieben.
- `postNCValue:ValueSpecification[0..*]` — Ordnet einem spezifischen Feature einer Rolle einen oder mehrere Werte als Inhalt zu. Diesen Inhalt darf eine Instanz, welche zur Laufzeit die Rolle einnimmt, die diesen Constraint besitzt, nach der Ausführung der Operation für das spezifische Feature nicht besitzen. Die referenzierte `postNCValue` muss denselben Typ haben wie vom `StructuralFeature` vorgeschrieben.
- `postValue:ValueSpecification[0..*]` — Ordnet einem spezifischen Feature einer Rolle einen oder mehrere Werte als Inhalt zu. Diesen Inhalt muss eine Instanz, welche zur Laufzeit die Rolle einnimmt, die diesen Constraint besitzt, nach der Ausführung der Operation für das spezifische Feature besitzen. Die referenzierte `ValueSpecification` muss denselben Typ haben wie vom `StructuralFeature` vorgeschrieben.
- `preElement:VCElement[0..*]` — Ordnet einem spezifischen Feature

einer Rolle ein oder mehrere `ConnectableElement`s als Inhalt zu. Diesen Inhalt muss eine Instanz, welche zur Laufzeit die Rolle einnimmt, die diesen Constraint besitzt, vor der Ausführung der Operation für das spezifische Feature besitzen. Das referenzierte Element muss denselben Typ haben wie vom `StructuralFeature` vorgeschrieben.

- `preNCElement:VCElement[0..*]` — Ordnet einem spezifischen Feature einer Rolle ein oder mehrere `ConnectableElement`s als Inhalt zu. Diesen Inhalt darf eine Instanz, welche zur Laufzeit die Rolle einnimmt, die diesen Constraint besitzt, vor der Ausführung der Operation für das spezifische Feature nicht besitzen. Das referenzierte Element muss denselben Typ haben wie vom `StructuralFeature` vorgeschrieben.
- `preNCValue:ValueSpecificaiton[0..*]` — Ordnet einem Feature einer Rolle einen oder mehrere Werte als Inhalt zu. Diesen Inhalt darf eine Instanz, welche zur Laufzeit die Rolle einnimmt, die diesen Constraint besitzt, vor der Ausführung der Operation für das spezifische Feature nicht besitzen. Die referenzierte `ValueSpecification` muss denselben Typ haben wie vom `StructuralFeature` vorgeschrieben.
- `preValue:ValueSpecification[0..*]` — Ordnet einem spezifischen Feature einer Rolle einen oder mehrere Werte als Inhalt zu. Diesen Inhalt muss eine Instanz, welche zur Laufzeit die Rolle einnimmt, die diesen Constraint besitzt, vor der Ausführung der Operation für das spezifische Feature besitzen. Die referenzierte `ValueSpecification` muss denselben Typ haben wie vom `StructuralFeature` vorgeschrieben.

Constraints

- Ein `Constraint` besitzt entweder einen Link auf ein `preElement` oder ein `preValue`.
- Ein `Constraint` besitzt entweder einen Link auf ein `postElement` oder ein `postValue`.
- Ein `Constraint` besitzt entweder einen Link auf ein `preNCElement` oder ein `preNCValue`.
- Ein `Constraint` besitzt entweder einen Link auf ein `postNCElement` oder ein `postNCValue`.

Semantik Ein Constraint setzt eine Rolle, ein Attribut der Klasse über welche die Rolle getypt ist und mögliche Inhalte des Attributes miteinander

in Beziehung. Ein Constraint beschreibt somit, welche Attributinhalte eine Instanz zur Laufzeit besitzen muss bzw. nicht besitzen darf, damit sie eine bestimmte Rolle einnehmen kann. Dabei wird zwischen Vor- und Nachbedingungen unterschieden. Die Werte bzw. Rollen, welche von einem Constraint referenziert werden, müssen zu dem `definingFeature` des Constraints typkonform sein. Ein Constraint fügt einer Rolle neben der Typdefinition somit zusätzliche Einschränkungen zu, die eine Instanz besitzen muss, um die Rolle einzunehmen.

A.2 Klasse DependentParameter

Ein `DependentParameter` ist eine Rolle in einem visuellen Kontrakt, die einen Pfad zu einem Parameter besitzt. Innerhalb einer Kollaboration oder eines visuellen Kontraktes kann ein Parameter wie eine Rolle behandelt werden.

Generalisierung

- `VCElement` aus dem Paket `VisualContracts`.

Attribute Diese Metamodellklasse enthält keine zusätzlichen Attribute.

Assoziationen Diese Metamodellklasse enthält keine zusätzlichen Assoziationen.

Constraints Ein `DependentParameter` ist über einen eindeutigen Pfad mit einem Parameter verbunden.

Semantik In der UML spezifiziert ein Parameter, wie Argumente an eine Operation übergeben bzw. von der Operation an den Client zurückgegeben werden. Beim Aufruf einer Operation wird ein Parameter an einen konkreten Wert oder konkrete Werte gebunden. In objektorientierten Systemen kann ein Parameter die Wurzel eines Objektbaums darstellen, die an eine Operation übergeben oder von einer Operation zurückgegeben wird. In unseren visuellen Kontrakten sind alle Objekte des Baumes, außer dem Wurzelknoten, `DependentParameter`.

Im UML-Metamodell findet sich eine entsprechende Klasse nicht. Wir haben diese Metamodellklasse mit aufgenommen, um die Elemente einer Baumstruktur in einem visuellen Kontrakt verwenden und darstellen zu können. Diese Elemente dürfen nicht mit den Rollen, die den Zustand eines Systems beschreiben (Vor-, Nach-, negative Vor- und Nachbedingungen) verwechselt werden.

A.3 Klasse NegativeCondition

Eine `NegativeCondition` ist eine `Collaboration` und definiert somit eine Menge von verlinkten Elementen (Rollen), welche über Konnektoren miteinander verbunden sind. Diese Rollen können zur Laufzeit von Instanzen eingenommen werden. Die Konnektoren definieren Kommunikationspfade zwischen den beteiligten Instanzen. Damit beschreibt eine `NegativeCondition` eine Sicht auf ein System.

Die Hauptaufgabe einer negativen Bedingung ist die Beschreibung eines Systemzustandes, der vor bzw. nach der Ausführung einer Operation nicht gegeben sein darf. D.h. es werden Objektstrukturen definiert, die in einem System nicht vorhanden sein dürfen. Zu diesem Zweck werden nur die nicht erlaubten Aspekte des Systems mit Hilfe von getypten Rollen einschließlich der Inhalte ihrer Attribute beschrieben, die für die Beschreibung der negativen Bedingung relevant sind.

Generalisierung

- `Collaboration` aus dem Paket `Collaborations`, UML-Spezifikation, S. 176

Attribute Diese Metamodellklasse enthält keine zusätzlichen Attribute.

Assoziationen

- `collaborationRole:VCElement[0..*]` — Referenziert eine Menge von Rollen, die zur Laufzeit von Instanzen eingenommen werden können, um diese Kollaboration zu erfüllen.

  Submenge von `Collaboration.collaborationRole` aus UML 2.0 Paket `InternalStructures`.

- `owningPost:Postcondition[0..1]` — Nachbedingung zu der die negative Anwendungsbedingung gehört.

  Submenge von `Element::owner` aus UML 2.0 Paket `Kernel`.

- `owningPre:Precondtion[0..1]` — Vorbedingung zu der die negative Anwendungsbedingung gehört.

  Submenge von `Element::owner` aus UML 2.0 Paket `Kernel`.

Constraints Metamodellklasse enthält keine zusätzlichen Constraints.

Semantik Eine `NegativeCondition` wird generell verwendet, um nicht erwünschte Objektstrukturen zu beschreiben. Dabei beschreibt eine negative Bedingung nur die Aspekte, die zur Beschreibung eines nicht erlaubten Systemzustandes notwendig sind, d.h. es werden keine vollständigen Systemzustände beschrieben. Daher kann ein Objekt zur Laufzeit verschiedene Rollen in verschiedenen Kollaborationen (d.h. in verschiedenen Vor-, Nachbedingungen und negativen Vor- und Nachbedingungen) einnehmen.

Hierzu definiert eine `NegativeCondition` eine Menge von Rollen zur Beschreibung einer negativen Bedingung. Die Rollen der negativen Bedingung können zur Laufzeit von Instanzen, die miteinander agieren, eingenommen werden. Die zur Erfüllung einer negativen Bedingung relevanten Beziehungen werden als Konnektoren zwischen den Rollen dargestellt. Die Rollen einer negativen Bedingung beschreiben die Bedeutung (Verwendung) der beteiligten Instanzen, während die Typen der Rollen beschreiben, welche Eigenschaften eine Instanz haben muss, um eine Rolle einnehmen zu können. Die Konnektoren definieren, welche Kommunikationspfade zwischen den Instanzen zur Laufzeit existieren müssen, damit sie die in der negativen Bedingung definierten Rollen einnehmen können.

Wenn zur Laufzeit eine Objektkonfiguration existiert, welche die in der negativen Bedingung definierten Rollen einschließlich der Konnektoren einnehmen kann, so ist die negative Bedingung nicht erfüllt.

Eine `NegativeCondition` muss nicht alle Eigenschaften der Instanzen, welche die in der negativen Bedingung definierten Rollen einnehmen, beschreiben. Genauso können zwischen den an der negativen Bedingung beteiligten Instanzen mehr Kommunikationspfade existieren als in der negativen Bedingung definiert sind. Eine negative Bedingung beschreibt also nur eine selektive Sicht auf eine nicht erlaubte Systemkonfiguration.

A.4 Klasse Parameter

Ein Parameter ist die Spezifikation eines Argumentes zur Übergabe von Informationen beim Aufruf oder bei der Beendigung einer Operation (einer `BehavioralFeature` laut UML-Spezifikation). Ein Paramter kann innerhalb einer Kollaboration oder eines visuellen Kontraktes wie eine Rolle behandelt werden.

Generalisierung

- `Parameter` aus dem Paket `Collaborations`, UML-Spezifikation, S. 188
- `VCElement` aus dem Paket `VisualContracts`

Attribute Diese Metamodellklasse enthält keine zusätzlichen Attribute.

Assoziationen Diese Metamodellklasse enthält keine zusätzlichen Assoziationen.

Constraints Metamodellklasse enthält keine zusätzlichen Constraints.

Semantik Ein Parameter spezifiziert, wie Argumente an eine Operation übergeben werden. Bei dem Aufruf einer Operation wird ein Parameter an einen konkreten Wert oder konkrete Werte gebunden.

Ein Parameter hat einen Namen, der den Parameter eindeutig innerhalb des Kontexts einer Operation oder visuellen Kontraktes identifiziert. Innerhalb eines visuellen Kontraktes wird der Parametername als Rollenname interpretiert.

A.5 Klasse Postcondition

Eine `Postcondition` ist eine `Collaboration` und definiert somit eine Menge von verlinkten Elementen (Rollen), welche über Konnektoren miteinander verbunden sind. Diese Rollen können zur Laufzeit von Instanzen eingenommen werden. Die Konnektoren definieren Kommunikationspfade zwischen den beteiligten Instanzen.

Die Hauptaufgabe einer Nachbedingung ist die Beschreibung eines Systemzustandes, der nach der Ausführung einer Operation gegeben sein muss. D.h. es werden Objektstrukturen definiert, die in einem System nach der Ausführung einer Operation vorhanden sein müssen. Zu diesem Zweck werden nur die notwendigen Aspekte des Systems — keine vollständigen Systemzustände — mit Hilfe von getypten Rollen einschließlich der Inhalte ihrer Attribute beschrieben, die für die Beschreibung der Nachbedingung relevant sind. Damit beschreibt eine `Postcondition` eine Sicht auf ein System.

Generalisierung

- `Collaboration` aus dem Paket `Collaborations`, UML-Spezifikation, S. 176

Attribute Diese Metamodellklasse enthält keine zusätzlichen Attribute.

Assoziationen

- `collaborationRole:VCElement[0..*]` — Referenziert eine Menge von Rollen, die zur Laufzeit von Instanzen eingenommen werden können, um diese Nachbedingung zu erfüllen.

  Submenge von `Collaboration.collaborationRole` aus UML 2.0 Paket `InternalStructures`.

- `npc:NegativeCondition[0..*]` — Referenziert eine Menge von Kollaborationen, die eine negative Anwendungsbedingung nach der Ausführung einer Operation darstellen.

  Submenge von `Element::ownedElement` aus UML 2.0 Paket `Kernel`

- `owningContract:VisualContract[1]` — Der visuelle Kontrakt zu dem diese Nachbedingung gehört.

  Submenge von `Element::owner` aus UML 2.0 Paket `Kernel`.

Constraints Metamodellklasse enthält keine zusätzlichen Constraints.

Semantik Eine `Postcondition` wird generell verwendet, um zu beschreiben, wie eine Menge von verlinkten Instanzen die Nachbedingung einer Operation erfüllen kann. Dazu beschreibt eine `Postcondition` nur die Aspekte,

die zur Beschreibung der Nachbedingung einer Operation notwendig sind, d.h. es werden keine vollständigen Systemzustände beschrieben. Daher kann ein Objekt zur Laufzeit verschiedene Rollen in verschiedenen Kollaborationen (d.h. in verschiedenen Vor-, Nachbedingungen und negativen Bedingungen) einnehmen.

Hierzu definiert eine `Postcondition` eine Menge von Rollen, die zur Erfüllung der Vorbedingung einer Operation notwendig sind. Die Rollen der Vorbedingung können zur Laufzeit von Instanzen, die miteinander agieren, eingenommen werden. Die zur Erfüllung einer Vorbedingung relevanten Beziehungen werden als Konnektoren zwischen den Rollen dargestellt. Die Rollen einer Vorbedingung beschreiben die Bedeutung (Verwendung) der beteiligten Instanzen, während die Typen der Rollen beschreiben, welche Eigenschaften eine Instanz haben muss, um eine Rolle einnehmen zu können. Die Konnektoren definieren, welche Kommunikationspfade zwischen den Instanzen zur Laufzeit existieren müssen, damit sie die in der Vorbedingung definierten Rollen einnehmen können.

Eine `Postcondition` muss nicht alle Eigenschaften der Instanzen beschreiben, welche die in der Nachbedingung definierten Rollen einnehmen. Genauso können zwischen den an der Nachbedingung beteiligten Instanzen mehr Kommunikationspfade existieren als in der `Postcondition` definiert sind. Eine Vorbedingung beschreibt also nur eine selektive Sicht auf eine Systemkonfiguration.

A.6 Klasse Precondition

Eine `Precondition` ist eine `Collaboration` und definiert somit eine Menge von verlinkten Elementen (Rollen), welche über Konnektoren miteinander verbunden sind. Diese Rollen können zur Laufzeit von Instanzen eingenommen werden. Die Konnektoren definieren Kommunikationspfade zwischen den beteiligten Instanzen.

Die Hauptaufgabe einer Vorbedingung ist die Beschreibung eines Systemzustandes, der vor der Ausführung einer Operation gegeben sein muss. D.h. es werden Objektstrukturen definiert, die in einem System vor der Ausführung einer Operation vorhanden sein müssen. Zu diesem Zweck werden nur die notwendigen Aspekte des Systems — keine vollständigen Systemzustände — mit Hilfe von getypten Rollen einschließlich der Inhalte ihrer Attribute beschrieben, die für die Beschreibung der Vorbedingung relevant sind. Damit

beschreibt eine `Precondition` eine Sicht auf ein System.

Generalisierung

- Collaboration aus dem Paket Collaborations, UML-Spezifikation, S. 176

Attribute Diese Metamodellklasse enthält keine zusätzlichen Attribute.

Assoziationen

- `collaborationRole:VCElement[0..*]` — Referenziert eine Menge von Rollen, die zur Laufzeit von Instanzen eingenommen werden können, um diese Vorbedingung zur erfüllen.

  Submenge von `Collaboration.collaborationRole` aus UML 2.0 Pakete `InternalStructures`.

- `nac:NegativeCondition[0..*]` — Referenziert eine Menge von Kollaborationen, die eine negative Anwendungsbedingung vor der Ausführung einer Operation darstellen.

  Submenge von `Element::ownedElement` aus UML 2.0 Paket `Kernel`.

- `owningContract:VisualContract[1]` — Der visuelle Kontrakt zu dem diese Vorbedingung gehört.

  Submenge von `Element::owner` aus UML 2.0 Paket `Kernel`.

Constraints Metamodellklasse enthält keine zusätzlichen Constraints.

Semantik Eine `Precondition` wird generell verwendet, um zu beschreiben, wie eine Menge von verlinkten Instanzen die Vorbedingung einer Operation erfüllen kann. Dazu beschreibt eine `Precondition` nur die Aspekte, die zur Beschreibung der Vorbedingung einer Operation notwendig sind, d.h. es werden keine vollständigen Systemzustände beschrieben. Daher kann ein Objekt zur Laufzeit verschiedene Rollen in verschiedenen Kollaborationen (d.h. in verschiedenen Vor-, Nachbedingungen und negativen Bedingungen) einnehmen.

Eine `Precondition` beschreibt eine Menge von Rollen, die zur Erfüllung der Vorbedingung einer Operation notwendig sind. Die Rollen der Vorbedingung können zur Laufzeit von Instanzen, die miteinander agieren, eingenommen werden. Die zur Erfüllung einer Vorbedingung relevanten Beziehungen werden als Konnektoren zwischen den Rollen dargestellt. Die Rollen einer Vorbedingung beschreiben die Bedeutung (Verwendung) der beteiligten Instanzen, während die Typen der Rollen beschreiben, welche Eigenschaften eine Instanz haben muss, um eine Rolle einnehmen zu können. Die Konnektoren definieren, welche Kommunikationspfade zwischen den Instanzen zur Laufzeit existieren müssen, damit sie die in der Vorbedingung definierten Rollen einnehmen können.

Eine `Precondition` muss nicht alle Eigenschaften der Instanzen beschreiben, welche die in der Vorbedingung definierten Rollen einnehmen. Genauso können zwischen den an der Vorbedingung beteiligten Instanzen mehr Kommunikationspfade existieren als in der `Precondition` definiert sind. Eine Vorbedingung beschreibt also nur eine selektive Sicht auf eine Systemkonfiguration.

A.7 Klasse VCElement

Ein `VCElement` beschreibt eine Rolle, die zur Laufzeit von Instanzen des Systems eingenommen werden kann.

Generalisierung

- `ConnectableElement` aus dem Paket `InternalStructures`, siehe UML-Spezifikation, S. 182

Attribute Diese Metamodellklasse enthält keine zusätzlichen Attribute.

Assoziationen

- `constraint:Constraint[0..*]` — Ein Constraint weist einem strukturellem Feature einer Rolle einen Wert oder mehrere Werte zu. Eine Rolle kann einen Constraint für jedes strukturelle Feature besitzen, das durch ihren Typ definiert ist. Es ist nicht notwendig für jedes strukturelle Feature des Typs einer Rolle einen Constraint zu definieren, da auch partielle Beschreibungen der Rollen zu einem Typ erlaubt sind.

Submenge von `Element::ownedElement` aus Paket UML 2.0 `Kernel`.

- `NegativeCondition[0..*]` — Negative Anwendungsbedingungen zu denen dieses `VCElement` gehört.

  Submenge von `Element::owner` aus UML 2.0 Paket `Kernel`.

- `owningConstraint:Constraint[0..*]` — Referenz auf eine Menge von Attributen (Constraints), die das `VCElement` als Inhalt besitzen.

- `Postcondition[0..1]` — Die Nachbedingung zu dem dieses `VCElement` gehört.

  Submenge von `Element::owner` aus UML 2.0 Paket `Kernel`.

- `Precondition[0..1]` — Die Vorbedingung zu dem dieses `VCElement` gehört.

  Submenge von `Element::owner` aus UML 2.0 Paket `Kernel`.

Constraints Metamodellklasse enthält keine zusätzlichen Constraints.

Semantik Ein VCElement beschreibt eine Rolle, die in einer Vor-, Nachbedingung, oder negativen Bedingung eines visuellen Kontraktes verwendet werden kann. Instanzen eines Systems können zur Laufzeit die beschriebene Rolle einnehmen.

Eine Rolle repräsentiert also eine Menge von Instanzen, die zur Laufzeit diese Rolle einnehmen können. Zur Einschränkung dieser Menge kann eine Rolle über eine Klasse getypt werden. Durch die Typisierung werden einer Rolle eine Menge von Eigenschaften (Attributen) zugeordnet, die eine Instanz zur Laufzeit besitzen muss. Weiterhin können den Attributen einer Rolle in einem visuellen Kontrakt noch Inhalte zugewiesen werden, wodurch die Menge der möglichen Instanzen, die zur Laufzeit diese Rolle einnehmen können noch weiter eingeschränkt werden.

`VCElemente` können über Konnektoren miteinander verknüpft werden. Damit können nicht nur einzelne Instanzen sondern auch verlinkte Instanzen (welche die entsprechenden Rollen einnehmen) beschrieben werden. Somit wird es möglich mit `VCElement`en eine Menge von Systemzuständen zu beschreiben.

A.8 Klasse VisualContract

Ein visueller Kontrakt ist eine `Collaboration` und beschreibt eine Menge von Rollen, die über Konnektoren miteinander verbunden sind. Zur Beschreibung einer Operation aggregiert ein visueller Kontrakt eine Vor- und eine Nachbedingung, die wiederum Kollaborationen sind und eine Teilmenge der Rollen, die von dem visuellen Kontrakt definiert wurden, enthalten. Diese Rollen können zur Laufzeit von Instanzen eingenommen werden. Wenn die Rollen einer Vorbedingung (Nachbedingung) zugeordnet sind, so müssen vor (nach) der Ausführung der Operation eine Menge von Instanzen existieren, welche diese Rollen einnehmen. Typischerweise konzentriert man sich bei der Spezifikation einer Operation in der Design-Phase nur auf die zur Beschreibung relevanten Inhalte. Details wie die Inhalte zu allen Attributen oder alle Attribute einer Rolle müssen nicht beschrieben werden.

Generalisierung

- `Collaboration` aus dem Paket `Collaborations` der UML 2.0 Spezifikation)

Attribute Diese Metamodellklasse enthält keine zusätzlichen Attribute.

Assoziationen

- `postcondition:Postcondition[1]` — Nachbedingung eines visuellen Kontraktes

  Submenge von `Element::ownedElement` aus UML 2.0 Paket `Kernel`.

- `precondition:Precondition[1]` — Vorbedingung eines visuellen Kontraktes

  Submenge von `Element::ownedElement` aus UML 2.0 Paket `Kernel`.

Constraints Metamodellklasse enthält keine zusätzlichen Constraints.

Semantik Visuelle Kontrakte werden verwendet, um zu beschreiben wie eine Kollektion von Objekten zur Erfüllung einer Aufgabe verwendet werden kann. Aus diesem Grund müssen in einem visuellen Kontrakt nur die

notwendigen Aspekte zur Erklärung einer Aufgabe beschrieben werden. Unnötige Aspekte können weggelassen werden.

Ein visueller Kontrakt definiert eine Menge von teilnehmenden Elementen, die zur Erfüllung einer bestimmten Aufgabe benötigt werden. Die in einem visuellen Kontrakt spezifizierten Rollen werden von Instanzen angenommen, wenn diese miteinander zur Laufzeit agieren. Bei den relevanten Beziehungen handelt es sich um Konnektoren (`Connector`) zwischen Rollen. Rollen in einem visuellen Kontrakt beschreiben die Verwendung von Instanzen, während die `Classifier`, welche die Typen für diese Rollen darstellen alle benötigten Eigenschaften der Instanzen spezifizieren. Somit beschreibt ein visueller Kontrakt, welche Eigenschaften eine Instanz haben muss, damit sie an dem visuellen Kontrakt teilnehmen kann: Eine Rolle (aufgrund ihres Typs) beschreibt die benötigte Menge an `Feature`s, die eine partizipierende Instanz haben muss. Die Konnektoren zwischen den Rollen spezifizieren, welche Kommunikationspfade (Links) zwischen den beteiligten Instanzen existieren müssen. Die Rollen können weiter eingeschränkt werden, indem ihren `Feature`s, d.h. ihren Attributen, über einen `Constraint` Inhalte zugewiesen werden.

Ein visueller Kontrakt kann einer Operation oder einem Classifier über die Klasse `CollaborationUse` hinzugefügt werden. Ein visueller Kontrakt, welcher auf diese Art und Weise verwendet wird, beschreibt wie diese Operation oder dieser `Classifier` realisiert wird durch eine Menge von zusammenarbeitenden Instanzen. Die Konnektoren in einem derartigen visuellen Kontrakt spezifizieren Links zwischen Instanzen. Verhalten, welches in dem visuellen Kontrakt spezifiziert wird, muss zur Laufzeit durch die zugehörige Operation ausgeführt werden.

Anhang B

Transformationsregeln zur Übersetzung visueller Kontrakte nach JML

In diesem Kapitel konstruieren wir eine formale, operationalisierbare Spezifikation der Übersetzung der visuellen Kontrakte nach JML. Das gwählte Verfahren ist in Kapitel 5 beschrieben. Die hier beschriebenen Transformationsregeln erzeugen aus der Vorbedingung eines visuellen Kontraktes eine JML-Spezifikation. Die Vorgehensweise zur Erzeugung einer JML-Spezifikation aus der Nachbedingung oder einer negativen Vor- bzw. Nachbedingung verläuft nahezu identisch und wird hier nicht weiter im Detail betrachtet. Zur Vereinfachung der Transformationsregeln gehen wir weiterhin davon aus, dass die Assoziationen im Klassendiagramm und somit auch die Links in den visuellen Kontrakten ungerichtet sind. Wenn die Quellregel einer Compound Rule keine Änderungen im Quellgraphen vornimmt, so stellen wir nur die linke Seite der Quellregel dar, um die Regeln übersichtlicher zu gestalten.

B.1 Grundgerüst einer JML-Spezifikation

Eine JML-Spezifikation besteht aus zwei Teilen, einem statischen und einem dynamischen Teil (vgl. auch Abschnitt 4.3.2). Die Java-Deklaration einer Operation selbst stellt den statischen Teil der Spezifikation dar. Das Verhalten der Operation wird mit Vor- und Nachbedingungen beschrieben. Die Verhaltensbeschreibung (JML-Annotation) der Operation ist eingebettet in

Java-Kommentare vor der Spezifikation der Operationssignatur (vgl. auch Abbildung 4.3.2).

Die folgenden Transformationsregeln erzeugen für eine Operation, die im Quellmodell mit einem visuellem Kontrakt annotiert ist, die Java-Signatur der Operation sowie einen Rahmen zur Spezifikation der Vor- und Nachbedingungen mit JML-Ausdrücken. In diesen Rahmen werden jeweils ein Nichtterminal für die Vorbedingung und ein Nichtterminal für die Nachbedingung eingefügt. Diese Nichtterminale werden in späteren Transformationsregeln durch konkrete Ausdrücke ersetzt oder gelöscht.

Die Transformation erfolgt in zwei Schritten. Als erstes wird die Operation einschließlich des Rahmens für die JML-Annotationen erzeugt und als zweites werden der Operation die Aufrufparameter — wenn vorhanden — hinzugefügt.

B.1.1 Erstellung der Operationen

Die Compound Rules op_1, op_2 und op_3 erzeugen für eine Operation mit einem visuellen Kontrakt eine Operationssignatur und einen Rahmen für die JML-Annotationen. Abhängig von den Eigenschaften der Operation wird entweder die Regel op_1, op_2 oder op_3 ausgeführt.

Die Compound Rule op_1 übersetzt eine Operation mit einem visuellen Kontrakt und einem einfachen Rückgabeparameter (Rückgabeparameter mit Kardinalität 1) in eine Operationssignatur mit einem Rahmen für die JML-Annotationen. Die Regel setzt voraus, dass in der Regel tr_1 (vgl. Abbildung 5.2) ein Nichtterminal `Methods(<c>)` erzeugt wurde. Auf der linken Seite der Quelltransformation wird gefordert, dass eine Operation einen visuellen Kontrakt besitzt, was dargestellt wird über eine Verbindung zwischen einem Objekt vom Typ `Operation` und `VisualContract` über ein Objekt `CollaborationUse`. Weiterhin wird in der Regel gefordert, dass die Operation einen Rückgabeparameter (siehe Objekte `Parameter` und `Classifier`) mit einer Kardinalität der Höhe eins besitzt (siehe Objekte `Parameter` und `LiteralUnlimitedNatural`). Die Klasse (Objekt `Class`) einer Operation wird benötigt, um die zu erzeugende Operation in den richtigen Klassenrahmen einzufügen, d.h. mit Hilfe des Identifier des Objekts `Class` wird das Nichtterminal auf der linken Seite der Zieltransformation initialisiert.

Die rechte Seite der Zieltransformation erzeugt als erstes einen Rahmen für die JML-Annotationen mit Nichtterminalen für die Vor- und Nachbedingungen, die in späteren Transformationsregeln aufgelöst werden. Danach wird die

Signatur der Operation erzeugt. Dazu werden die Sichtbarkeit der Operation, der Name der Operation und der Typ des Rückgabeparameters übernommen. Zwei weitere Nichtterminale sorgen dafür, dass die Parameter der Operation mit weiteren Regeln erstellt werden können und weitere Operationen in die JML-Datei eingefügt werden können.

Die Compound Rules op_2 und op_3 funktionieren im Prinzip ähnlich. Die Regel op_2 wird angewendet, wenn eine Operation im Quellmodell keinen Rückgabeparameter besitzt. Auf der linken Seite der Quelltransformation wird dies durch eine entsprechende negative Anwendungsbedingung dargestellt. Weiterhin ist in diesem Fall anstatt eines Typs für einen Rückgabeparameter im Zielmodell ein `void` einzufügen.

Die Regel op_3 wird angewendet, wenn eine Operation im Quellmodell einen Rückgabeparameter mit einer Kardinalität größer als eins besitzt. Auf der linken Seite der Quelltransformation wird dies durch die Existenz eines Rückgabeparameters ergänzt durch eine negative Anwendungsbedingung, die eine Kardinalität von eins ausschliesst, ausgedrückt. Die Zielregeln in op_2 und op_3 erzeugen wiederum ein Nichtterminal, um in späteren Transformationsschritten weitere Operationen in das Zielmodell einfügen zu können.

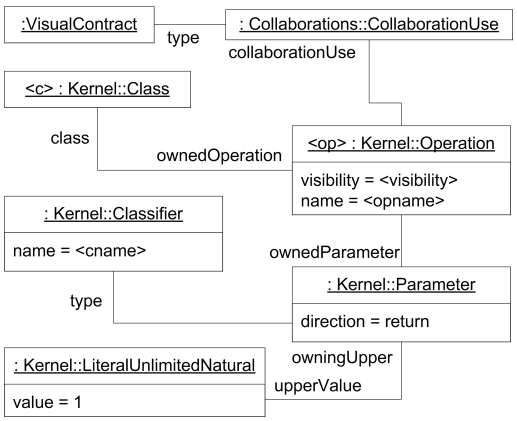
Die Regeln op_4 löscht das Nichtterminal `Methods(<c>)` aus dem Zielmodell. Diese Regel darf erst angewendet werden nachdem eine weitere Anwendung der Regeln op_1, op_2 und op_3 nicht mehr möglich ist. Ansonsten werden eventuell nicht für alle Operationen mit visuellen Kontrakten entsprechende Repräsentationen im Zielmodell erstellt.

Damit ergibt sich für die Ausführung der Regeln $op_1, \ldots, op_4$ folgender Kontrollausdruck: $\langle \{op_1, op_2, op_3\} \downarrow, op_4 \rangle$. Mit diesem Kontrollausdruck wird sichergestellt das die Regeln op_1, op_2 und op_3 solange in einer nichtdeterministischen Reihenfolge ausgeführt werden, bis sie nicht mehr angewendet werden können. Erst danach wird die Transformationsregel op_4 einmalig ausgeführt und somit das Nichtterminal `Methods(<c>)` aus dem JML-Code gelöscht.

B.1.2 Erstellung der Parameterliste

Die Compound Rules pa_1, pa_2, pa_3 und pa_4 erzeugen die Parameter einer Operation im Zielmodell. Die Regeln setzen voraus, dass in den Transformationsregeln op_1, op_2 oder op_3 ein Nichtterminal `Argsstart(<op>)` erzeugt wurde. Wenn die Operation mindestens einen Parameter besitzt, wird die Regel pa_1 ausgeführt, ansonsten wird die Regel pa_2 ausgeführt. Nach der Ausführung einer Regel pa_1 wird pa_3 so lange ausgeführt, bis alle Parameter

	$r_S : L_S ::= R_S$	$r_T : L_T ::= R_T$

op_1		`#Methods(<c>)# ::=`
		`/*@ public normal_behavior`
		`  @ #Pre(<op>)#`
		`  @ #Post(<op>)#`
		`  @*/`
		`  <visibility> <cname> <opname>`
		`  (#Argsstart(<op>)#);`
		
		`#Methods(<c>)#`

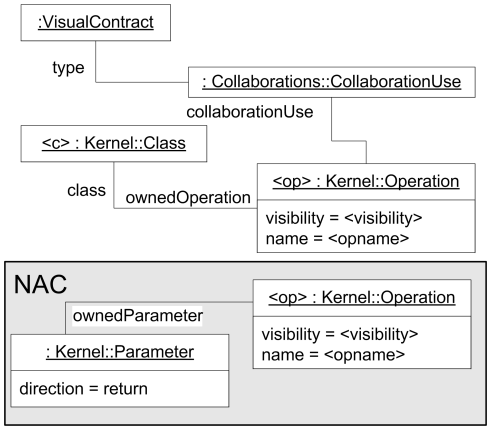
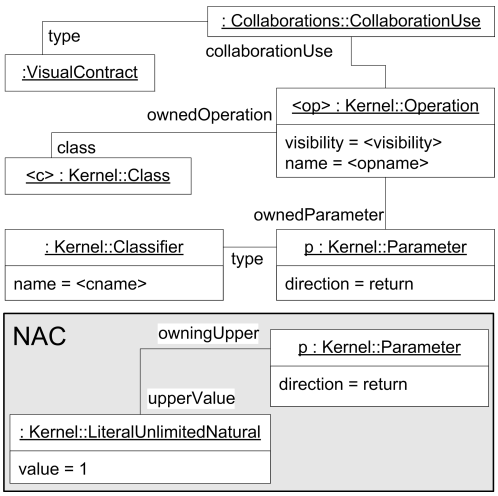
op_2		`#Methods(<c>) ::=`
		`/*@ public normal_behavior`
		`  @ #Pre(<op>)#`
		`  @ #Post(<op>)#`
		`  @*/`
		`  <visibility> void <opname>`
		`  (#Argsstart(<op>)#);`
		
		`#Methods(<c>)#`

Abbildung B.1: Compound Rules zur Erzeugung der Operationssignaturen (1)

	$r_S : L_S ::= R_S$	$r_T : L_T ::= R_T$

op_3		`Methods(<c>) ::=`
		`/*@ public normal_behavior`
		`  @ #Pre(<op>)#`
		`  @ #Post(<op>)#`
		`  @*/`
		`  <visibility> <cname>[] <opname>`
		`  (#Argsstart(<op>)#);`
		
		`#Methods(<c>)#`

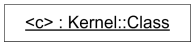
op_4		`#Methods(<c>)# ::=` ϵ

Abbildung B.2: Compound Rules zur Erzeugung der Operationssignaturen (2)

verarbeitet wurden. Zum Abschluss wird die Regel pa_4 ausgeführt, um das verbleibende Nichtterminal zu löschen.

Die Compound Rule pa_1 übersetzt einen Parameter (`Parameter`) einer Operation (`Operation`) in einen Parameter für eine Java-Operation. Dazu wird auch der Typ (`Classifier`) benötigt. Für die Zieltransformation wird der Name der Operation benötigt, um die Nichtterminale korrekt zu instanzieren. Mit der rechten Seite der Zieltransformation wird der Typname, der Parametername und ein Nichtterminal in das Zielmodell eingefügt. Aufgrund des Nichtterminals `Argscont(<op>)` können in Folgeregeln weitere Parameter für dieselbe Operation in das Zielmodell eingefügt werden.

Die Regel pa_2 löscht ein Nichtterminal aus dem Zielmodell. Die Regel darf in einem gegebenem Kontext nur ausgeführt werden, wenn die Regel pa_1 nicht ausgeführt werden kann. Das heisst die Regel darf nur ausgeführt werden, wenn die Operation keine Parameter besitzt.

Die Regel pa_3 übersetzt weitere Parameter (`Parameter`) einer Operation (`Operation`) in Parameter einer Java-Operation. Der Effekt der Regel ist nahezu identisch mit dem Effekt der Regel pa_1. Dem auf dem Zielmodell erzeugtem Ausdruck wird lediglich ein Komma vorangestellt, da es sich um Folgeparameter handelt. Wenn für eine Operation mehrere Parameter einer Operation in den JML-Code eingefügt werden müssen, so ist sicher zu stellen, dass die Reihenfolge identisch ist zu der Reihenfolge der Parameter in der Operation aus der entsprechenden Java-Klasse. Im UML-Klassendiagramm besitzt die Assoziation zwischen der Klasse `Operation` und `Parameter` eine Ordnung. Wir gehen in diesem Fall vereinfachend von der Annahme aus, dass die Parameter immer gemäß dieser vorgegebenen Ordnung sortiert sind, d.h. bei der Transformation gemäß dieser Reihenfolge bearbeitet werden.

Die Regel pa_4 löscht ein Nichtterminal aus dem Zielmodell. Die Regel darf in einem gegebenem Kontext nur ausgeführt werden, wenn keine weiteren Ausführungen der Regel pa_3 möglich sind.

Für die Ausführung der Regeln $pa_1, \ldots, pa_4$ wird folgender Kontrollausdruck benötigt: $\langle \{pa_1, pa_3\} \downarrow, pa_2, pa_4 \rangle$. Mit diesem Kontrollausdruck wird sichergestellt das die Regeln pa_1 und pa_3 und solange in einer nichtdeterministischen Reihenfolge ausgeführt werden, bis sie nicht mehr angewendet werden können. Erst danach werden die entsprechenden Nichtterminale aus dem JML-Code gelöscht.

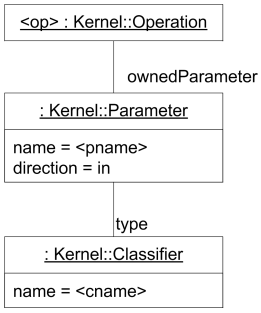
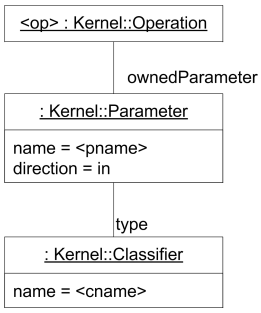
	$r_S : L_S ::= R_S$	$r_T : L_T ::= R_T$
pa_1		`#Argsstart(<op>)# ::=` `<cname> <pname> #Argscont(<op>)#`
pa_2		`#Argsstart(<op>)# ::=` ϵ
pa_3		`#Argscont(<op>)# ::=` `, <cname> <pname> #Argscont(<op>)#`
pa_4		`#Argscont(<op>)# ::=` ϵ

Abbildung B.3: Compound Rules zur Erzeugung der Parameterliste einer Operation

B.2 Erstellung der Vorbedingung

Die JML-Spezifikation einer Vorbedingung eines visuellen Kontraktes muss eine Teilgraphensuche in einem Systemgraphen durchführen. Der Systemgraph ist eine Repräsentation des Systemzustandes mit Objekten und Links. Dazu müssen alle Rollen einer Vorbedingung im Systemgraphen gefunden werden.

Generell spannen die Algorithmen für die Teilgraphensuche einen Suchbaum auf, der mittels eines Backtracking-Algorithmus durchlaufen wird. Auch wir verwenden diese Vorgehensweise, um die Rollen einer Vorbedingung an Laufzeitobjekte zu binden. Die Knoten in einem Schritt repräsentieren einen Schritt bei der Teilgraphensuche. In jeder Ebene des Suchbaums wird eine Rolle an ein Laufzeitobjekt des Systems gebunden. Ein Laufzeitobjekt kann an eine Rolle gebunden werden, wenn die Inhalte seiner Attribute den vorgegebenen Inhalten im visuellen Kontrakt entsprechen und wenn das Laufzeitobjekt mit weiteren Laufzeitobjekten gemäß den Vorgaben im visuellen Kontrakt verbunden ist.

Diese Vorgehensweise wird auch mit unseren Transformationsregeln umgesetzt, indem wir die Quantoren von JML entsprechend schachteln. Die Wurzel unseres Suchbaums stellt das Element `self` dar, das in jeder Vor- oder Nachbedingung vorhanden sein muss. Daher generieren wir als erstes JML-Code, um zu überprüfen, ob die Inhalte der Attribute des Elementes `self` den vorgegebenen Inhalten entsprechen.

Das Element `self` ist in einem visuellen Kontrakt mit weiteren Rollen, die im Metamodell mit `ConnectableElement`s dargestellt werden, verbunden. Nach der Überprüfung des Objekts `self` erzeugen wir JML-Code, um potentielle Laufzeitobjekte zu finden, welche den Rollen entsprechen. Dazu muss überprüft werden, welche Inhalte die Attribute der Laufzeitobjekte haben und mit welchen weiteren Laufzeitobjekten diese verbunden sind. Bei der Überprüfung der weiteren Laufzeitobjekte unterscheiden wir zwischen gebundenen (nur den Link überprüfen) und nicht gebundenene Laufzeitobjekten. Ungebundene Laufzeitobjekte müssen an eine weitere Variable gebunden werden.

B.2.1 Überprüfung der Variablen des Elements self

Die Compound Rules $prsv_1$ und $prsv_2$ erzeugen JML-Ausdrücke zur Überprüfung der Inhalte der Variablen des Elementes `self` aus dem visuellen Kontrakt (Objekt `this` in JML-Spezifikation).

Die Compound Rules $prsv_1$ und $prsv_2$ sind nahezu identisch. Die Regeln setzen voraus, dass in den Compound Rules op_1, ..., op_3 ein Nichtterminal `Pre(<op>)` erzeugt wurde. Wenn ein Attribut in einem visuellem Kontrakt einen elementaren Datentyp (Integer, String) als Inhalt besitzt, wird die Regel $prsv_1$ ausgeführt. Ansonsten hat das Attribut ein anderes `VCElement` als Inhalt. Ersichtlich wird dieser Unterschied an den linken Seiten der Quelltransformationen $prsv_1$ und $prsv_2$. Die Regeln werden ausgeführt, wenn das Element `self` ein Attribut besitzt, dem in der Vorbedingung (siehe entsprechende Assoziationen am Objekt `Constraint`) ein Constraint zugewiesen ist. Pro Attribut wird ein `requires` Ausdruck generiert. Aufgrund der JML-Semantik sind diese Ausdrücke verundet.

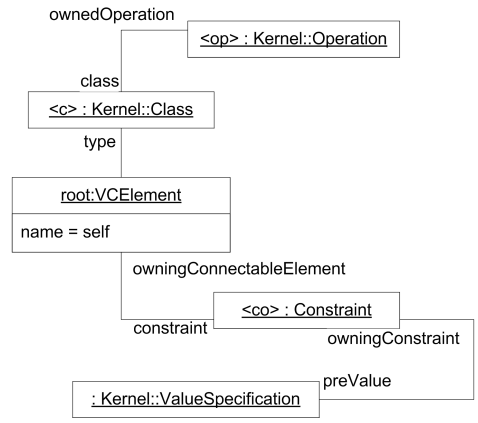
Der genaue Ausdruck der JML-Spezifikation zur Überprüfung der Inhalte wird in diesen Regeln nicht erzeugt, sondern lediglich vorbereitet. In späteren Regelanwendungen wird das Nichtterminal `PreConstraint(<co>)` entsprechend ersetzt.

B.2.2 Abhängige Elemente des Objekts self überprüfen

Die Compound Rules $prsr_1$, ..., $prsr_8$ erzeugen JML-Code zum Binden von Laufzeitobjekten an Rollen, die mit der Rolle `self` des visuellen Kontraktes verbunden sind. Dazu werden aus den visuellen Kontrakten entsprechende `\exists`-Quantoren generiert, welche die mit dem Laufzeitobjekt `this` verbundenen Objekte überprüfen. Für die korrekte Schachtelung der logischen Ausdrücke über Konjuktoren unterscheiden wir in unseren Regeln zwischen den zu generierendem JML-Ausdrücken für das erste zu bindende Laufzeitobjekt und folgenden Laufzeitobjekten. Als erstes wird eine der Regeln $prsr_1$, ..., $prsr_3$ ausgeführt, um den Code zum Binden der ersten Rolle zu erzeugen. Der Code für weitere Rollen wird mit den Regeln $prsr_4$, ..., $prsr_6$ erzeugt.

Die Compound Rule $prsr_1$ setzt voraus, dass in den Compound Rules op_1, ..., op_3 ein Nichtterminal `Pre(<op>)` erzeugt wurde. Bei der ersten Ausführung der Regel $prsr_1$ wird dieses Nichtterminal aus dem Zielmodell entfernt. Die linke Seite der Quelltransformation setzt voraus, dass im visuellen Kontrakt eine Rolle `self` mit einer weiteren Rolle (`<dependent>`) über einen `Connector` verbunden ist. Die Rolle `<dependent>` besitzt mindestens ein Attribut, dessen Inhalt zu überprüfen ist, da dem Attribut im entsprechenden visuellen Kontrakt in der Vorbedingung ein Wert (elementarer Datentyp) zugewiesen wurde (siehe Objekt `Constraint`). Die weiteren vorausgesetzten Informationen auf der linken Regelseite, wie z.B. der Typ der Rolle, werden benötigt, um einen korrekten JML-Ausdruck zu generieren.

	$r_S : L_S ::= R_S$	$r_T : L_T ::= R_T$

$prsv_1$		`#Pre(<op>)# ::=`
		`@ requires`
		`@ this.get#PreConstraint(<co>)#;`
		`@ #Pre(<op>)#`

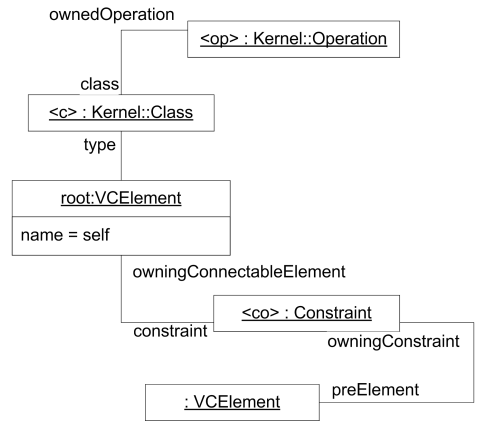
$prsv_2$		`#Pre(<op>)# ::=`
		`@ requires`
		`@ this.get#PreConstraint(<co>)#;`
		`@ #Pre(<op>)#`

Abbildung B.4: Inhalt der Variablen des Elements self überprüfen

Die rechte Seite der Quelltransformation führt keine Änderungen am Modell vor. Es wird lediglich jeweils ein Objekt `Pre-Processed` mit den Objekten `<root>`, `<dependent>` und dem `:Connector` verbunden. Durch diese Konstruktion ist in späteren Transformationsschritten die Information verfügbar, dass diese Objekte bereits bearbeitet wurden, bzw. dass bereits entsprechender JML-Code zur Bearbeitung dieser Objekte erzeugt wurde.

Die rechte Seite der Zieltransformation erzeugt einen `exists`-Ausdruck, der ausgehend vom Objekt `this` nach einem Objekt sucht, welches die Eigenschaften der Rolle `<dname>` erfüllt. Der `exists`-Ausdruck erzeugt eine Variabe `<dname>` vom Typ `<cname>`. Die Links von `this` zu Klassen vom Typ `<cname>` werden zum Binden dieser Variable abgesucht. Danach wird Code erzeugt, um die Eigenschaften der gesuchten Rolle zu überprüfen (Nichtterminal `PreCheck(<dependent>)`), um weitere vom Objekt `self` abhängige Rollen zu finden (Nichtterminal `PreFollow(<op>)`) und um von der neu gefundenen Rolle `<dname>` abhängige Rollen zu finden (Nichtterminal `PreRec(<dependent>)`).

Die Compound Rules $prsr_2$ und $prsr_3$ funktionieren ähnlich. Die Regel $prsr_2$ wird ausgeführt, wenn die zu suchende Rolle einen Attributinhalt referenziert, der selbst kein elementarer Datentyp sondern ein `VCElement` ist. Entsprechend ändert sich die linke Seite der Quelltransformation, während die rechte Seite der Zieltransformation nicht verändert werden muss.

Die Compound Rule $prsr_3$ wird ausgeführt, wenn in dem visuellen Kontrakt keine Aussagen über die Inhalte der Attribute der zu suchenden Rolle gemacht werden. Dies wird mit den negativen Anwendungsbedingungen auf der linken Seite der Quelltransformation beschrieben. Auf der rechten Seite der Zieltransformation wird das Nichtterminal `PreCheck(<dependent>)` nicht benötigt, da keine Attribute zu überprüfen sind.

Bei der Transformation der visuellen Kontrakte nach JML wird eine der Regeln $prsr_1$, ..., $prsr_3$ einmal ausgeführt. Danach werden die Transformationsregeln $prsr_4$, ..., $prsr_6$ ausgeführt, um Code zum Binden weiterer Rollen zu generieren. Die Unterscheidung wird notwendig, um eine korrekte Schachtelung der logischen Ausdrücke generieren zu können. Die Regel $prsr_4$ entspricht nahezu der Regel $prsr_1$, $prsr_5$ entspricht $prsr_2$ und $prsr_6$ entspricht $prsr_3$.

Für die Ausführung der Regeln $prsr_1, \ldots, prsr_8$ wird der Kontrollausdruck $\langle \{prsr_1, prsr_2, prsr_3\}, \{prsr_4, prsr_5, prsr_6\} \downarrow, prsr_7, prsr_8 \rangle$ benötigt. Mit diesem Kontrollausdruck wird sichergestellt, zuerst eine der Regeln $prsr_1$, ..., $prsr_3$ ausgeführt werden. Danach werden die Regeln $prsr_4$, ..., $prsr_6$ so-

lange in einer nichtdeterministischen Reihenfolge ausgeführt werden, bis sie nicht mehr angewendet werden können. Erst danach werden die entsprechenden Nichtterminale aus dem JML-Code gelöscht. Die Regel $prsr_7$ löscht das Nichtterminal `PreFollow(<op>)`, wenn alle mit dem Objekt `self` verknüpften Elemente abgearbeitet worden sind. Die Regel $prsr_8$ löscht das Nichtterminal `Pre(<op>)`, wenn dieses nicht von den Regeln $prsr_1, \ldots, prsr_3$ gelöscht worden ist. Dies kann vorkommen, wenn das Objekt `self` alleine auf der linken Seite eines visuellen Kontraktes steht.

B.2.3 Abhängige Elemente überprüfen

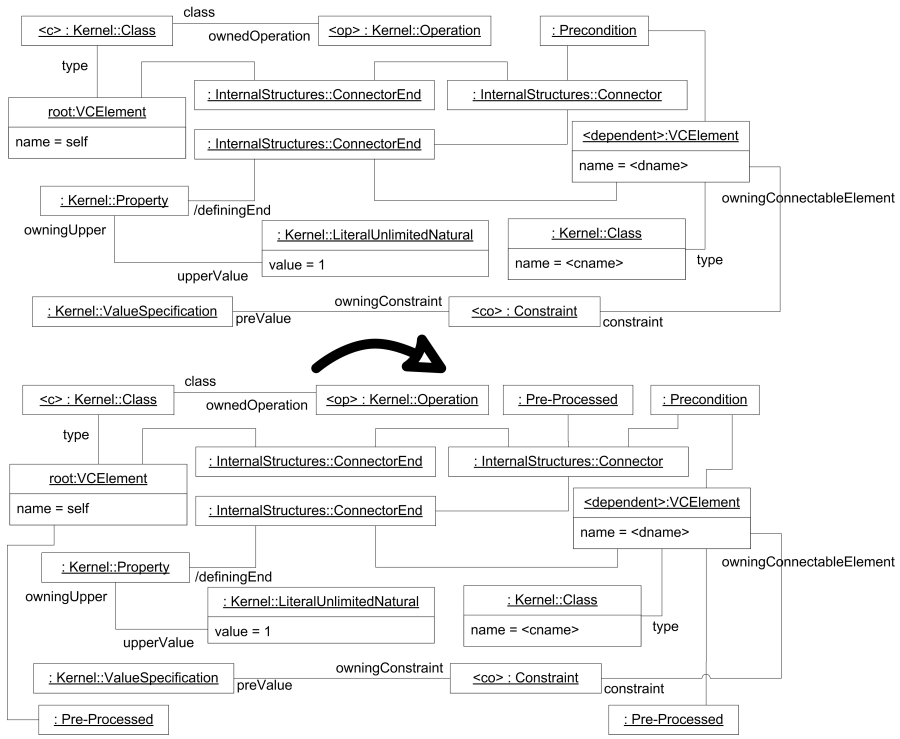
Die Compound Rules $prer_1, \ldots, prer_6$ erzeugen JML-Code zum Binden weiterer Laufzeitobjekte an Rollen, die nicht mit der Rolle `self` des visuellen Kontraktes verbunden sind. Dazu werden aus den visuellen Kontrakten entsprechende `\exists`-Quantoren zur Suche der Laufzeitobjekte ausgehend von bereits gefundenen Laufzeitobjekten generiert. Für die korrekte Schachtelung der logischen Ausdrücke über Konjuktoren unterscheiden wir ausgehend von einem bereits gefundenem Laufzeitobjekt in unseren Regeln zwischen den zu generierenden JML-Ausdrücken für das erste zu bindende Laufzeitobjekt und folgenden Laufzeitobjekten. Als erstes wird eine der Regeln $prer_1, \ldots, prer_3$ ausgeführt, um den Code zum Binden der ersten Rolle zu erzeugen. Der Code für weitere Rollen wird mit den Regeln $prer_4, \ldots, prer_6$ erzeugt.

Die Compound Rule $prer_1$ setzt voraus, dass mit den Compound Rules $prsr_1, \ldots, prsr_6$, $prer_1, \ldots, prer_6$ ein Nichtterminal `PreRec(<vc1>)` erzeugt wurde. Eine Ausführung der Regel $prer_1$ entfernt dieses Nichtterminal aus dem Zielmodell. Die linke Seite der Quelltransformation setzt voraus, dass im visuellen Kontrakt eine bereits gebundene Rolle `<vc1>` mit einer weiteren Rolle (`<vc2>`) über einen `Connector` verbunden ist. Die Rolle `<vc2>` besitzt mindestens ein Attribut, dessen Inhalt zu überprüfen ist, da dem Attribut im entsprechenden visuellen Kontrakt in der Vorbedingung ein Wert (elementarer Datentyp) zugewiesen wurde (siehe Objekt `Constraint`). Die weiteren vorausgesetzten Informationen auf der linken Regelseite, wie z.B. der Typ der Rolle, werden benötigt, um einen korrekten JML-Ausdruck zu generieren. Mit der negativen Anwendungsbedingung wird sichergestellt, dass die Rolle `<vc2>` nicht bereits in einer anderen Transformation bearbeitet wurde.

Die rechte Seite der Quelltransformation führt keine Änderungen am Modell durch. Es wird lediglich jeweils ein Objekt `Pre-Processed` mit den Objekten `<vc2>` und `:Connector` verbunden. Durch diese Konstruktion ist in späteren Transformationsschritten die Information verfügbar, dass diese Objekte

$prsr_1$

$r_S : L_S ::= R_S$



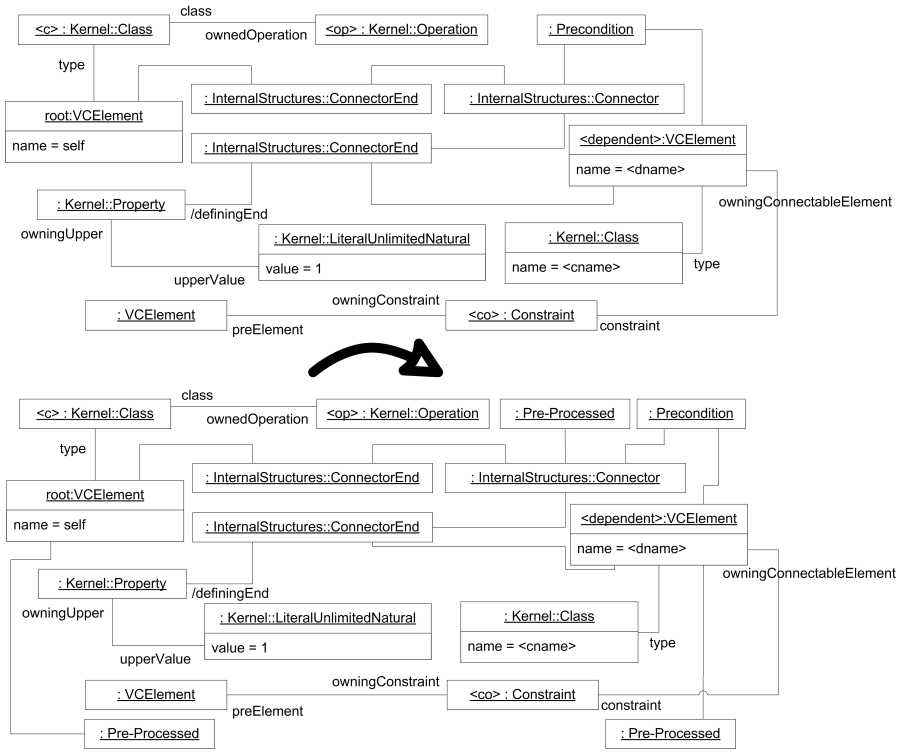
$r_T : L_T ::= R_T$

```
#Pre(<op>)# ::=
@ requires (\exists <cname> <dname>;
     this.get<cname>().values().contains(<dname>);
     #PreCheck(<dependent>)# && #PreFollow(<op>)#
     #PreRec(<dependent>)#);
```

Abbildung B.5: Abhängige Elemente des Elementes self überprüfen (1)

$prsr_2$

$r_S : L_S ::= R_S$



$r_T : L_T ::= R_T$

```
#Pre(<op>)# ::=
@ requires (\exists <cname> <dname>;
     this.get<cname>().values().contains(<dname>);
     #PreCheck(<dependent>)# && #PreFollow(<op>)#
     #PreRec(<dependent>)#);
```

Abbildung B.6: Abhängige Elemente des Elementes self überprüfen (2)

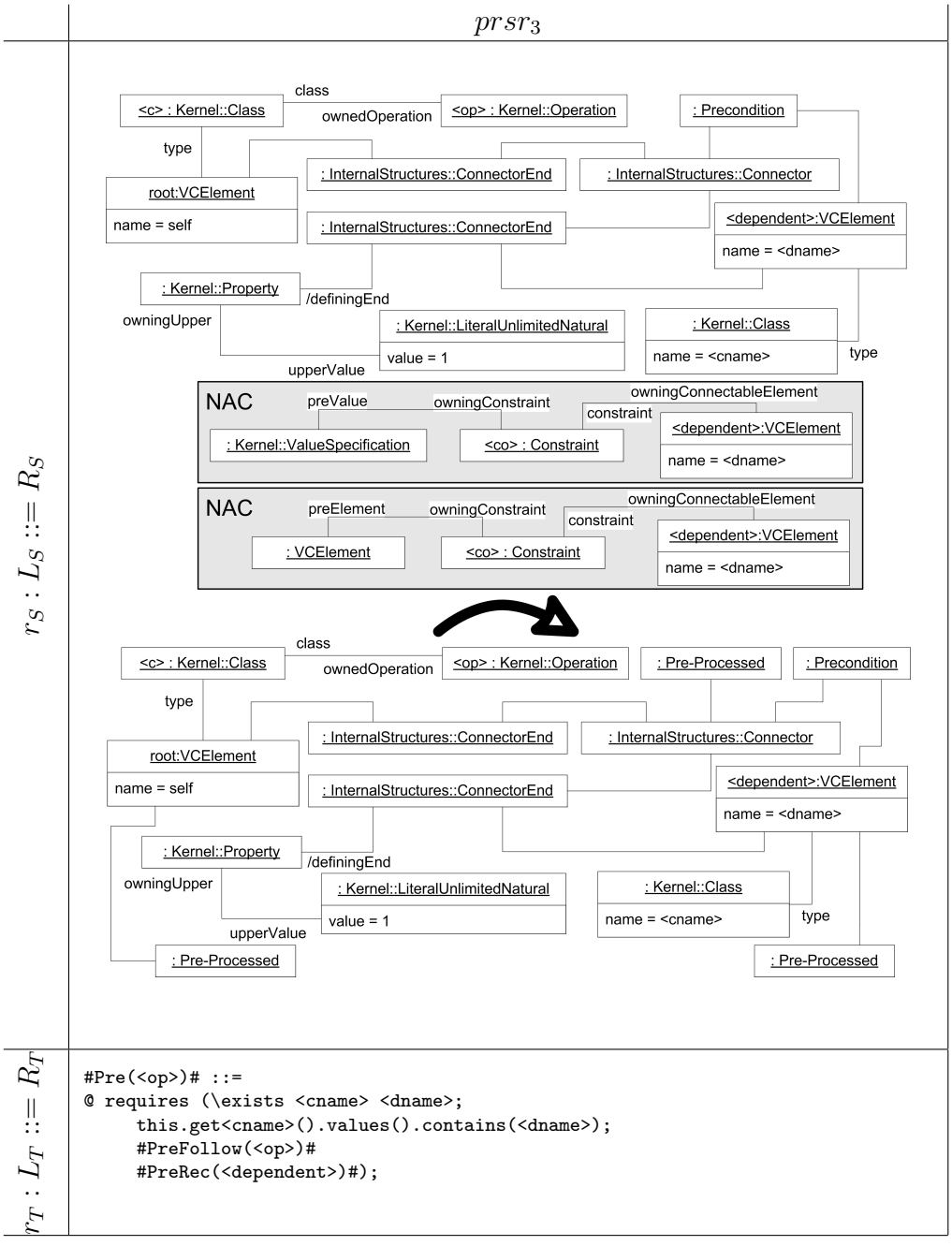
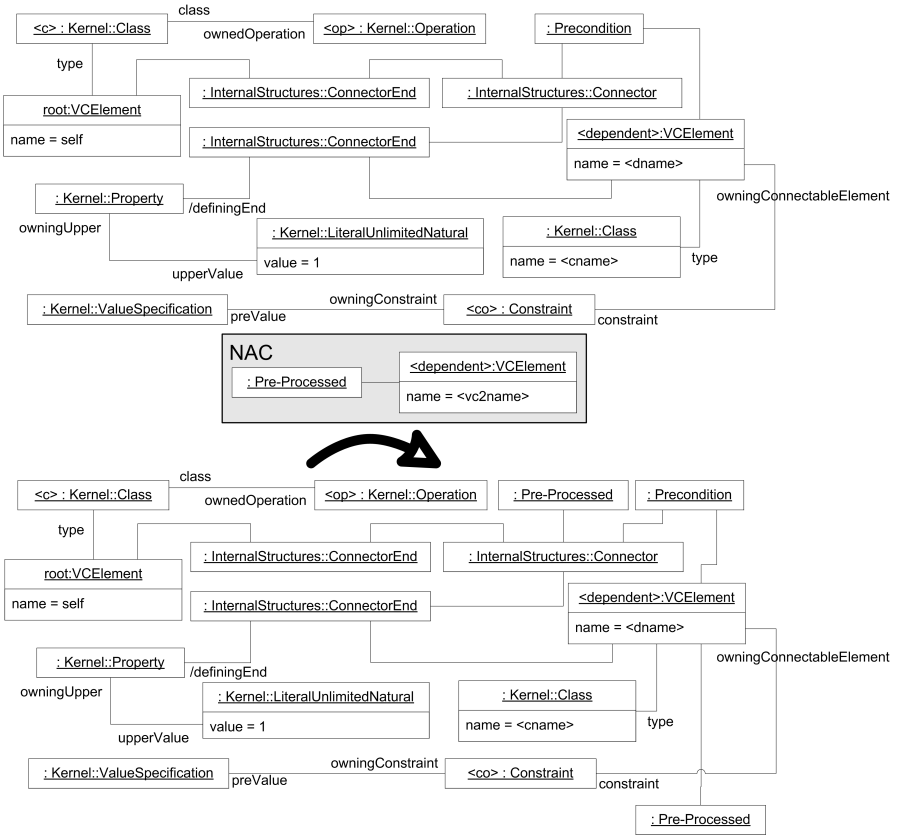


Abbildung B.7: Abhängige Elemente des Elementes self überprüfen (3)

$prsr_4$

$r_S : L_S ::= R_S$



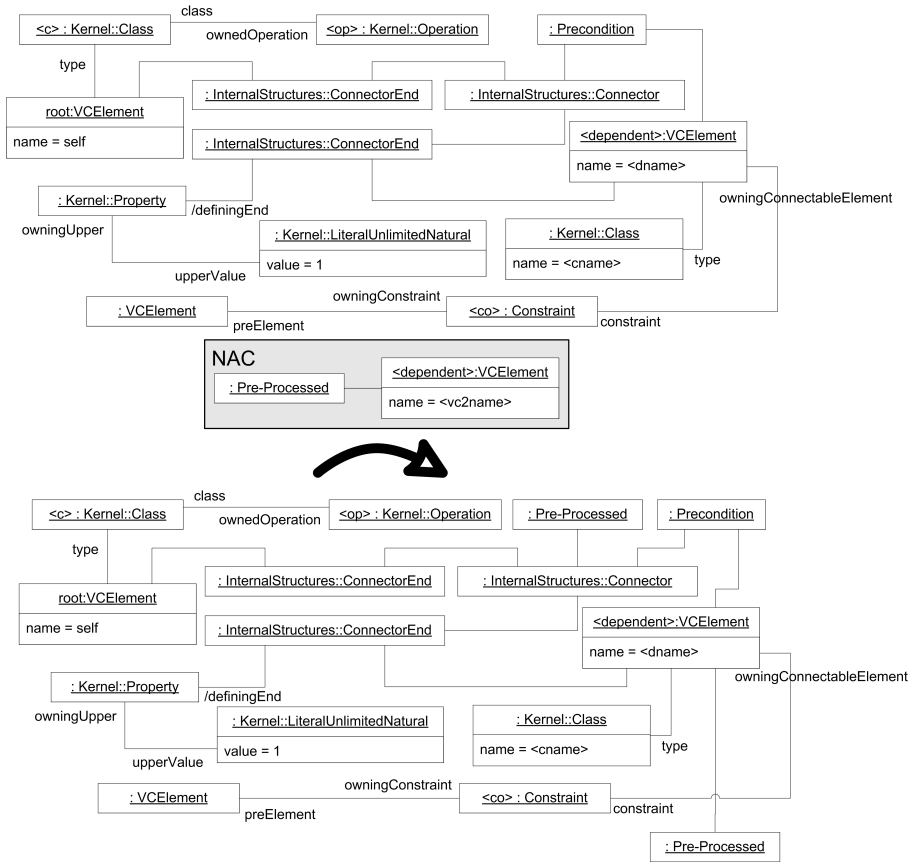
$r_T : L_T ::= R_T$

```
#PreFollow(<op>)# ::=
(\exists <cname> <dname>;
  this.get<cname>().values().contains(<dname>);
  #PreCheck(<dependent>)# && #PreFollow(<op>)#
  #PreRec(<dependent>)# ) &&
```

Abbildung B.8: Abhängige Elemente des Elementes self überprüfen (4)

$prsr_5$

$r_S : L_S ::= R_S$



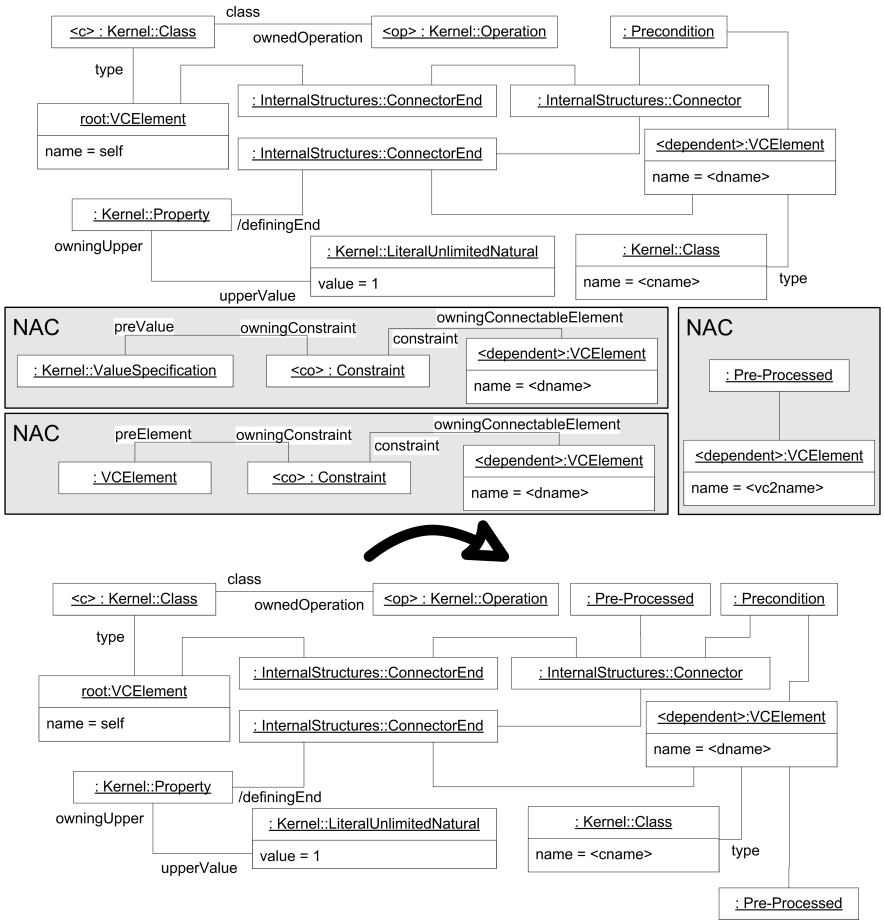
$r_T : L_T ::= R_T$

```
#PreFollow(<op>)# ::=
(\exists <cname> <dname>;
   this.get<cname>().values().contains(<dname>);
   #PreCheck(<dependent>)# && #PreFollow(<op>)#
   #PreRec(<dependent>)# ) &&
```

Abbildung B.9: Abhängige Elemente des Elementes self überprüfen (5)

$prsr_6$

$r_S : L_S ::= R_S$



$r_T : L_T ::= R_T$

```
#PreFollow(<op>)# ::=
(\exists <cname> <dname>;
   this.get<cname>().values().contains(<dname>);
   #PreFollow(<op>)#
   #PreRec(<dependent>)# ) &&
```

Abbildung B.10: Abhängige Elemente des Elementes self überprüfen (6)

	$r_S : L_S ::= R_S$	$r_T : L_T ::= R_T$
$prsr_7$	<op> : Kernel::Operation	#PreFollow(<op>)# ::= ϵ
$prsr_8$	<op> : Kernel::Operation	#Pre(<op>)# ::= ϵ

Abbildung B.11: Abhängige Elemente des Elementes self überprüfen (7)

bereits bearbeitet wurden, bzw. dass bereits entsprechender JML-Code zur Bearbeitung dieser Objekte erzeugt wurde.

Die rechte Seite der Zieltransformation erzeugt einen `exists`-Ausdruck, der ausgehend vom bereits gebundenem Objekt `<vc1>` (in Zieltransformation wird der Name der Rolle `<vc1name>` verwendet) nach einem Objekt sucht, welches die Eigenschaften der Rolle `<vc2>` erfüllt. Der Ausdruck erzeugt eine Variabe `<vc2name>` vom Typ `<cname>`. Die Links von `<vc1>` zu Klassen vom Typ `<cname>` werden zum Binden dieser Variable abgesucht. Danach wird Code erzeugt, um die Eigenschaften der gesuchten Rolle zu überprüfen (Nichtterminal `PreCheck(<vc2>)`), um weitere vom bereits gebundenem Objekt `<vc1>` abhängige Rollen zu finden (Nichtterminal `PreRecFollow(<vc1>)`) und um von der neu gefundenen Rolle `<vc2>` abhängige Rollen zu finden (Nichtterminal `PreRec(<vc2>)`).

Die Compound Rules $prer_2$ und $prer_3$ funktionieren ähnlich. Die Regel $prer_2$ wird ausgeführt, wenn die zu suchende Rolle einen Attributinhalt referenziert, der selbst kein elementarer Datentyp sondern ein `VCElement` ist. Entsprechend ändert sich die linke Seite der Quelltransformation, während die rechte Seite der Zieltransformation nicht verändert werden muss.

Die Compound Rule $prer_3$ wird ausgeführt, wenn in dem visuellen Kontrakt keine Aussagen über die Inhalte der Attribute der zu suchenden Rolle gemacht werden. Dies wird mit den zusätzlichen negativen Anwendungsbedingungen auf der linken Seite der Quelltransformation beschrieben. Auf der rechten Seite der Zieltransformation wird das Nichtterminal `PreCheck(<vc2>` nicht benötigt, da keine Attribute zu überprüfen sind.

Bei der Transformation der visuellen Kontrakte nach JML wird ausgehend von einer bereits gebundenen Rolle eine der Regeln $prer_1, \ldots, prer_3$ ein-

mal ausgeführt. Danach werden die Transformationsregeln $prer_4$, ..., $prer_6$ ausgeführt, um Code zum Binden weiterer Rollen zu generieren. Die Unterscheidung wird notwendig, um eine korrekte Schachtelung der logischen Ausdrücke generieren zu können. Die Regel $prer_4$ entspricht nahezu der Regel $prer_1$, $prer_5$ entspricht $prer_2$ und $prer_6$ entspricht $prer_3$.

B.2.4 Links zu besuchten Elementen überprüfen

Bei der Umsetzung der Vorbedingung eines visuellen Kontraktes in eine JML-Spezifikation kann bei der Teilgraphensuche ein bereits gebundenes Objekt neu gefunden werden. Falls der Link über den das bereits gebundene Objekt gefunden wurde noch nicht bearbeitet wurde, so muss dieser noch überprüft werden. Hierfür sind die Compound Rules $prer_7$ und $prer_8$ zuständig. Für die korrekte Schachtelung der logischen Ausdrücke über Konjuktoren unterscheiden wir ausgehend von einem bereits gefundenem Laufzeitobjekt in unseren Regeln, ob bereits zuvor nach einem Objekt gesucht wurde oder nicht. Falls zuvor noch kein anderes neues Objekt gebunden wurde, wird die Regel $prer_7$ ausgeführt, ansonsten wird die Regel $prer_8$ ausgeführt.

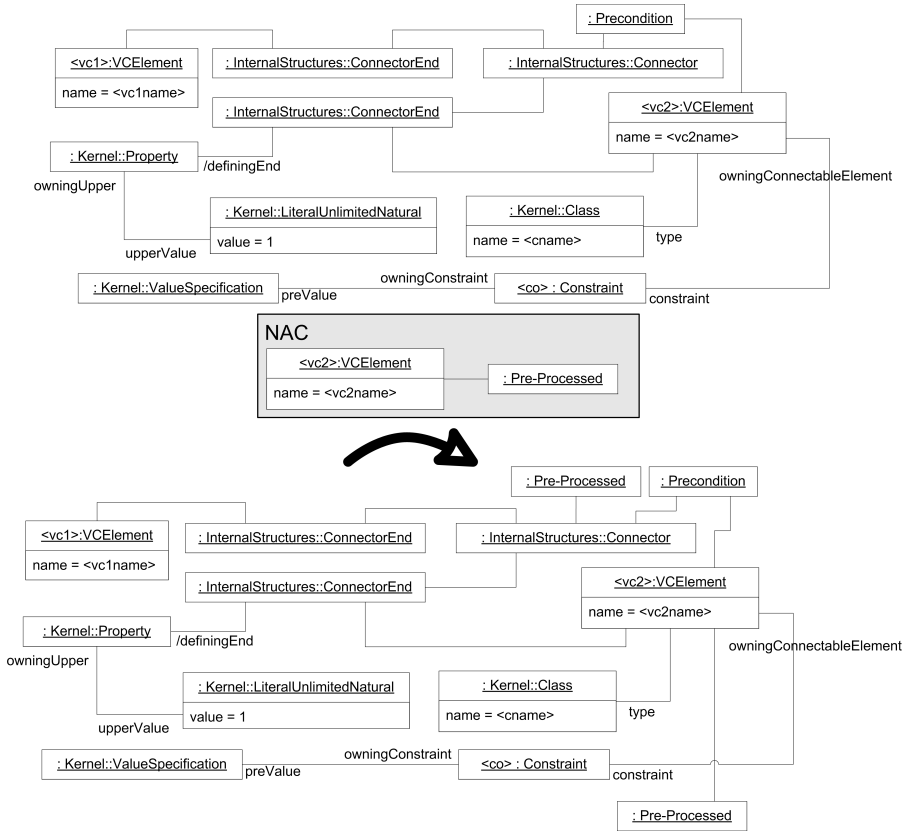
Die Compound Rule $prer_7$ setzt voraus, dass mit den Compound Rules $prsr_1$, ..., $prsr_6$, $prer_1$, ..., $prer_6$ ein Nichtterminal `PreRec(<vc1>)` erzeugt wurde. Eine Ausführung der Regel $prer_7$ entfernt dieses Nichtterminal aus dem Zielmodell. Die linke Seite der Quelltransformation setzt voraus, dass im visuellen Kontrakt eine bereits gebundene Rolle `<vc1>` mit einer weiteren Rolle (`<vc2>`) über einen `Connector` verbunden ist. Die Rolle `<vc2>` ist bereits gebunden (siehe Objekt `Pre-Processed`). Die weiteren vorausgesetzten Informationen auf der linken Regelseite, wie z.B. der Typ der Rolle, werden benötigt, um einen korrekten JML-Ausdruck zu generieren. Mit der negativen Anwendungsbedingung wird sichergestellt, dass der `Connector` noch nicht bearbeitet wurde.

Die rechte Seite der Quelltransformation führt keine Änderungen am Modell durch. Es wird lediglich ein Objekt `Pre-Processed` mit dem Objekt `:Connector` verbunden. Durch diese Konstruktion ist in späteren Transformationsschritten die Information verfügbar, dass dieses Objekt bereits bearbeitet wurde, bzw. dass bereits entsprechender JML-Code zur Bearbeitung dieses Objekts erzeugt wurde.

Die rechte Seite der Zieltransformation erzeugt einen Ausdruck, der überprüft, ob das zuvor an `<vc2>` (im JML-Ausdruck wird der Name des Elementes `<vc2name>` verwendet) gebundene Objekt über einen Link von `<vc1>`

$prer_1$

$r_S : L_S ::= R_S$



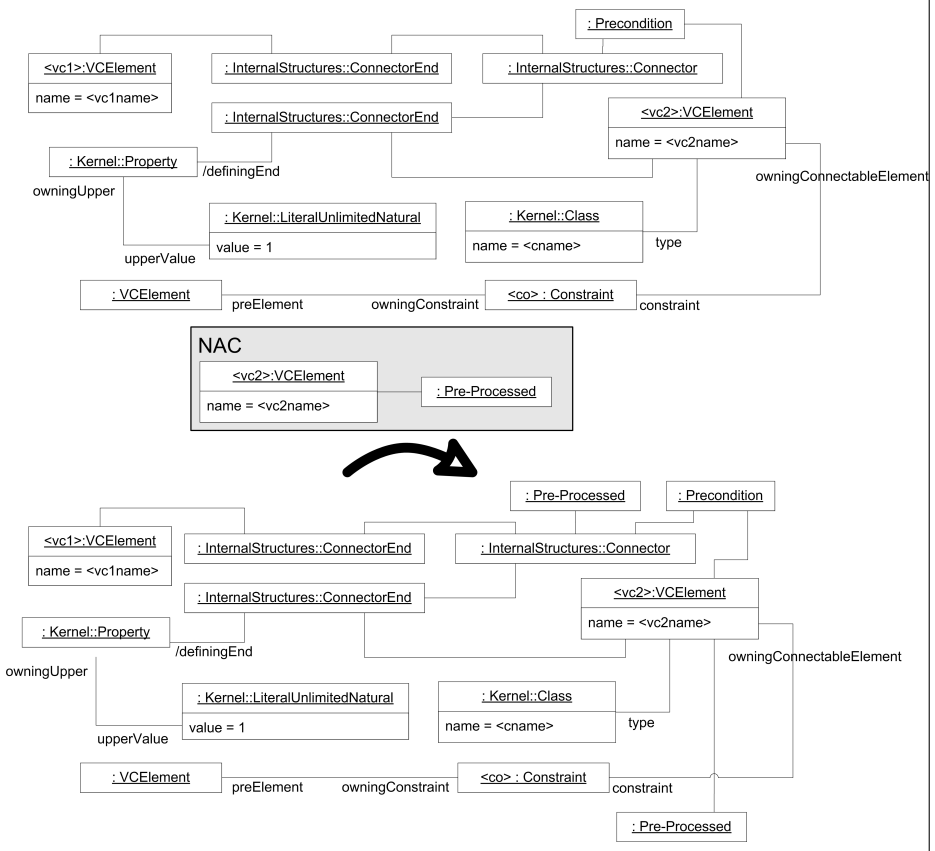
$r_T : L_T ::= R_T$

```
#PreRec(<vc1>)# ::=
(\exists <cname> <vc2name>;
    <vc1name>.get<cname>().values().contains(<vc2name>);
    #PreCheck(<vc2>)# && #PreRecFollow(<vc1>)#
    #PreRec(<vc2>)#)
```

Abbildung B.12: Abhängige Elemente überprüfen (1)

$prer_2$

$r_S : L_S ::= R_S$



$r_T : L_T ::= R_T$

```
#PreRec(<vc1>)# ::=
(\exists <cname> <vc2name>;
    <vc1name>.get<cname>().values().contains(<vc2name>);
    #PreCheck(<vc2>)# && #PreRecFollow(<vc1>)#
    #PreRec(<vc2>)#)
```

Abbildung B.13: Abhängige Elemente überprüfen (2)

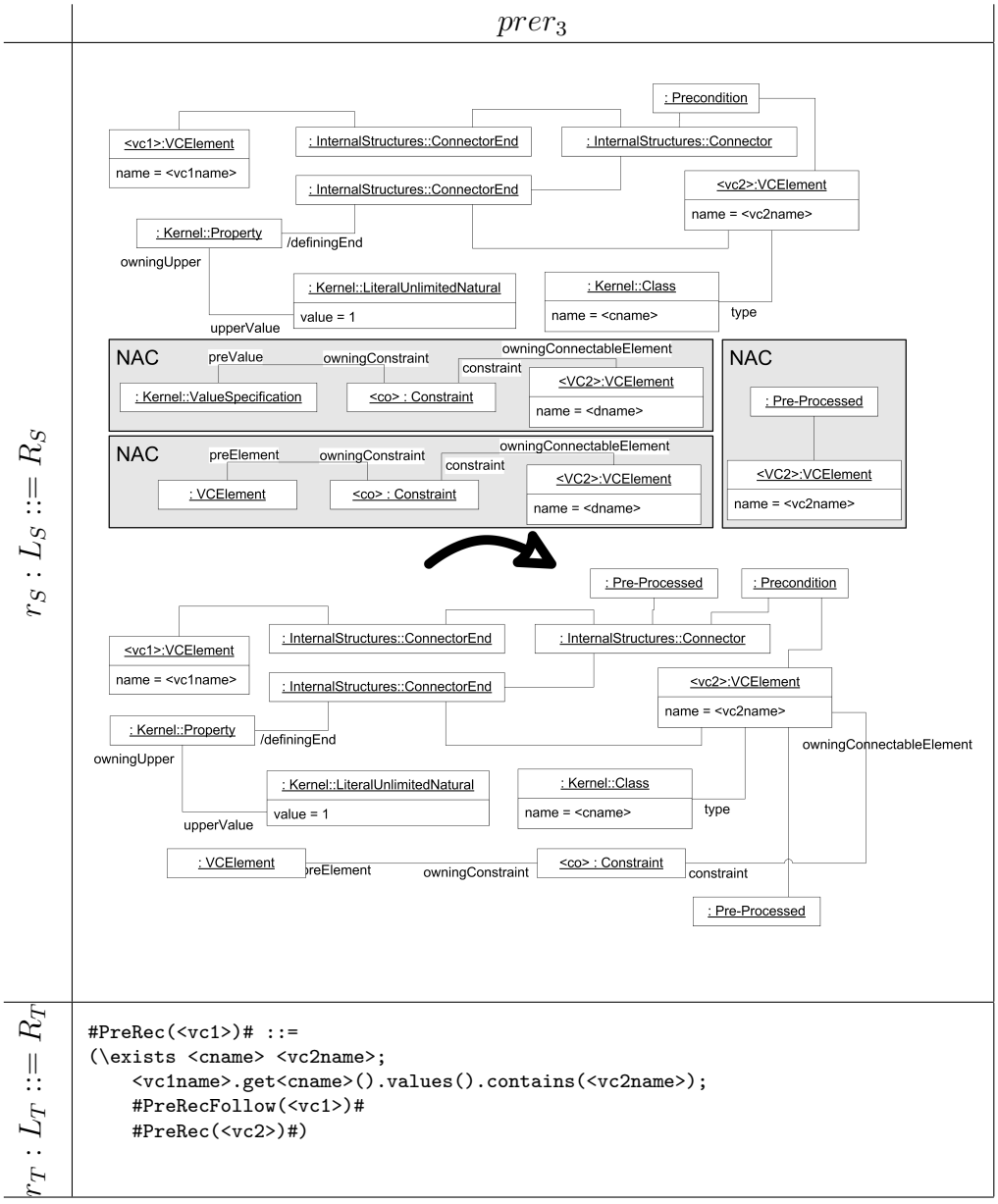
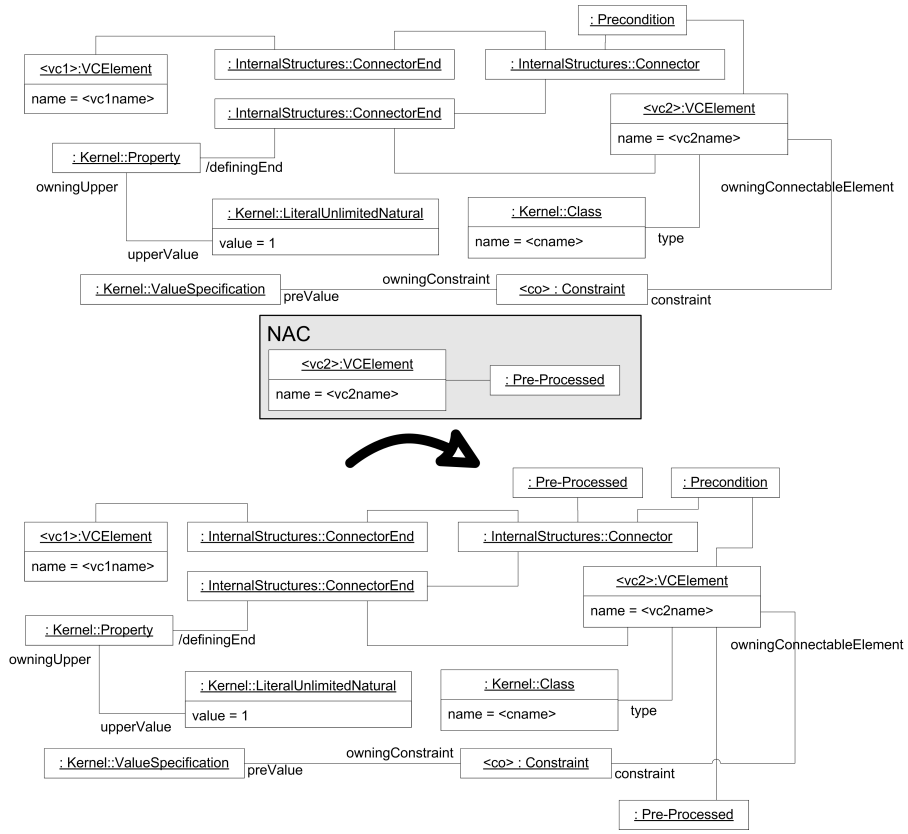


Abbildung B.14: Abhängige Elemente überprüfen (3)

$prer_4$

$r_S : L_S ::= R_S$



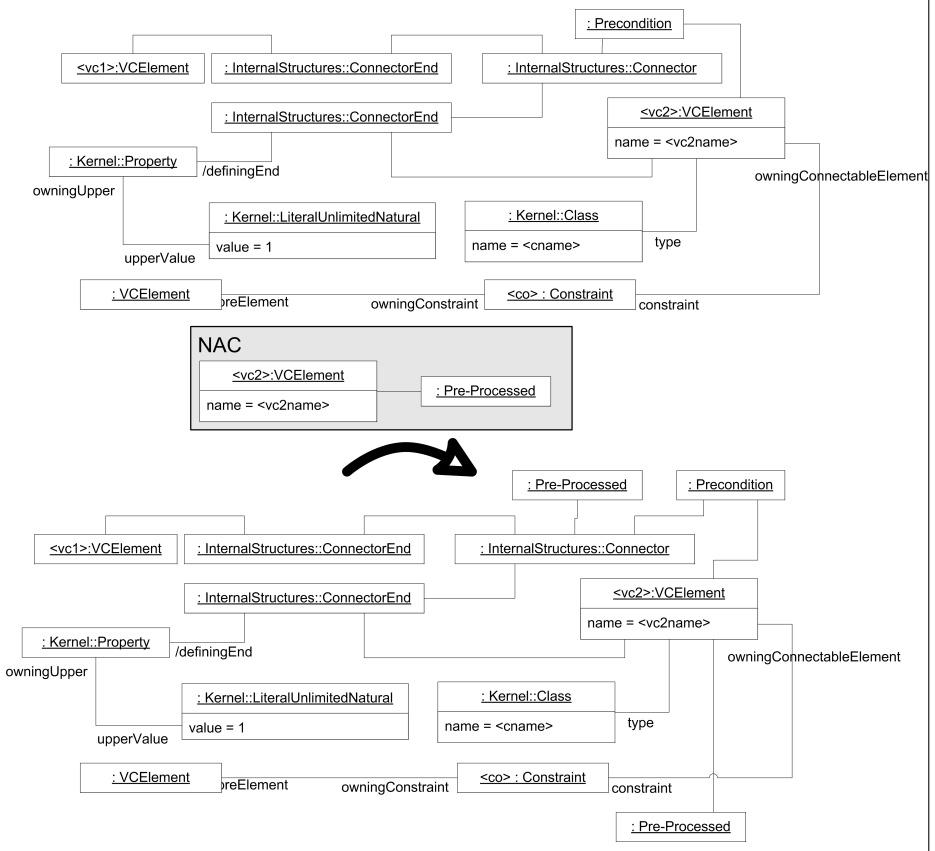
$r_T : L_T ::= R_T$

```
#PreRecFollow(<vc1>)# ::=
(\exists <cname> <vc2name>;
   <vc1name>.get<cname>().values().contains(<vc2name>);
   #PreCheck(<vc2>)# && #PreRecFollow(<vc1>)#
   #PreRec(<vc2>)#) &&
```

Abbildung B.15: Abhängige Elemente überprüfen (4)

$prer_5$

$r_S : L_S ::= R_S$



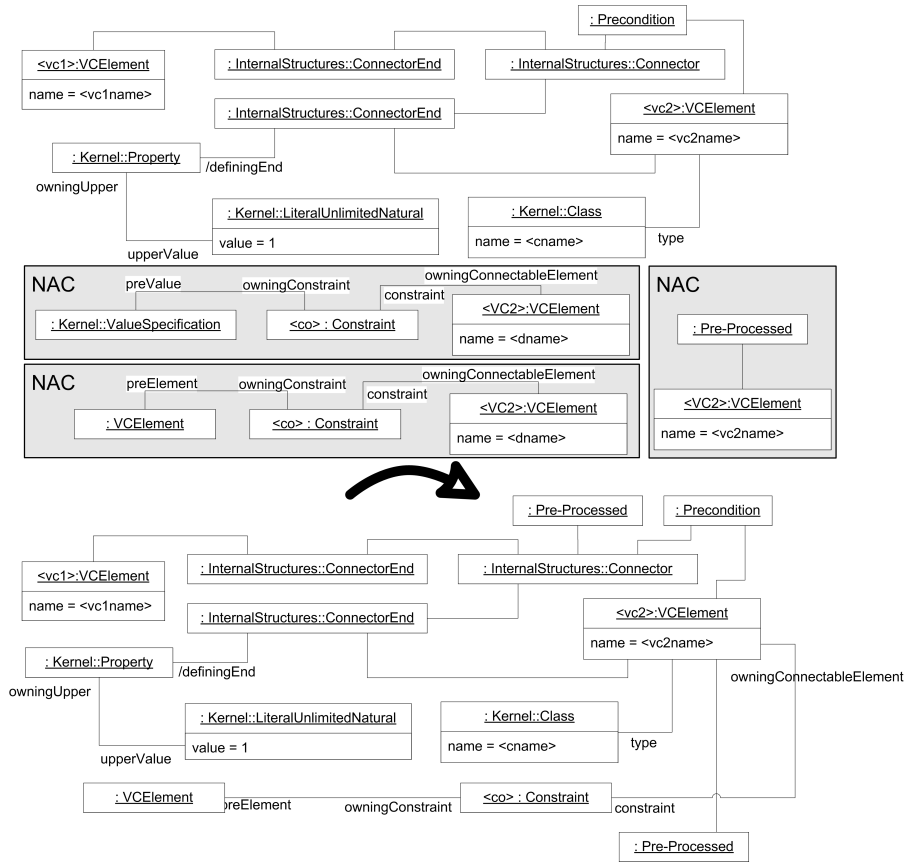
$r_T : L_T ::= R_T$

```
#PreRecFollow(<vc1>)# ::=
(\exists <cname> <vc2name>;
   <vc1name>.get<cname>().values().contains(<vc2name>);
   #PreCheck(<vc2>)# && #PreRecFollow(<vc1>)#
   #PreRec(<vc2>)#) &&
```

Abbildung B.16: Abhängige Elemente überprüfen (5)

$prer_6$

$r_S : L_S ::= R_S$



$r_T : L_T ::= R_T$

```
#PreRecFollow(<vc1>)# ::=
(\exists <cname> <vc2name>;
   <vc1name>.get<cname>().values().contains(<vc2name>);
   #PreRecFollow(<vc1>)#
   #PreRec(<vc2>)#) &&
```

Abbildung B.17: Abhängige Elemente überprüfen (6)

aus erreicht werden kann. Die Eigenschaften der gesuchten Rolle müssen hier nicht mehr überprüft werden, da diese an anderer Stelle beim Binden der Rolle überprüft werden. Um weitere vom bereits gebundenem Objekt `<vc1>` abhängige Rollen zu bestimmen wird das Nichtterminal `PreRecFollow(<vc1>)` eingefügt.

Die Compound Rule $prer_8$ funktioniert ähnlich. Bei der Transformation der visuellen Kontrakte nach JML wird ausgehend von einer bereits gebundenen Rolle die Regel $prer_7$ einmal ausgeführt. Danach wird die Transformationsregel $prer_8$ ausgeführt, um Code zum Binden weiterer Rollen zu generieren. Die Unterscheidung wird notwendig, um eine korrekte Schachtelung der logischen Ausdrücke generieren zu können.

B.2.5 Abschluss der Suche

Nachdem die Suchgraphen mit den Regeln $prer_1, \ldots, prer_8$ aufgebaut wurden, sind im Zielmodell noch Nichtterminale vorhanden, die mit den Regel $prer_9$ und $prer_{10}$ gelöscht werden.

Für die Ausführung der Regeln $prer_1, \ldots, prer9$ wird folgender Kontrollausdruck $\langle\{\{prer_1, prer_2, prer_3, prer_7\}\downarrow, \{prer_4, prer_5, prer_6, prer_8\}\downarrow\}\downarrow, prer_9\downarrow, prer_{10}\downarrow\rangle$ benötigt. Mit diesem Kontrollausdruck wird sichergestellt, dass zuerst die Regeln $prer_1, \ldots, prer_3, prer_7$ in einer nichtdeterministischen Reihenfolge ausgeführt werden, bis sie nicht mehr angewendet werden können. Danach werden die Regeln $prer_4, \ldots, prer_6, prer_8$ solange in einer nichtdeterministischen Reihenfolge ausgeführt werden, bis sie nicht mehr angewendet werden können. Der gesamte Ausdruck wird auch noch einmal wiederholt, bis er nicht mehr ausgeführt werden kann, da während der Ausführung der Teilausdrücke weitere Nichtterminale entstehen, die aufgelöst werden müssen.

Zum Abschluss werden die verbleibenden Nichtterminale mit den Regel $prer_9$, $prer_{10}$ gelöscht. Die Regel $prer_{10}$ löst das Nichtterminal `PreRec(<vc1>)` jedoch nicht nach ϵ sondern nach `true` auf. Der Grund ist, dass bei einer Auflösung des Nichtterminals nach ϵ die Verundung (`&&`) nicht korrekt aufrecht erhalten werden kann, während die Einfügung von `true` die korrekte Verundung aufrecht erhält. Die Semantik des generierten JML-Ausdrucks ändert sich durch das Einfügen eines `true` nicht. Die Regel $prer_{10}$ wird ausgeführt, wenn bei einer Rolle nur die Attributinhalte überprüft werden müssen, wenn eine Rolle also z.B. ein Blatt in dem Graphen einer Vorbedingung darstellt. Alternativ können die Regeln $prsr_1, \ldots, prsr_6$ und $prer_1, \ldots, prer_8$ mit leicht abgewandelten Quellregeln vervielfacht werden, um zu verhindern, dass im

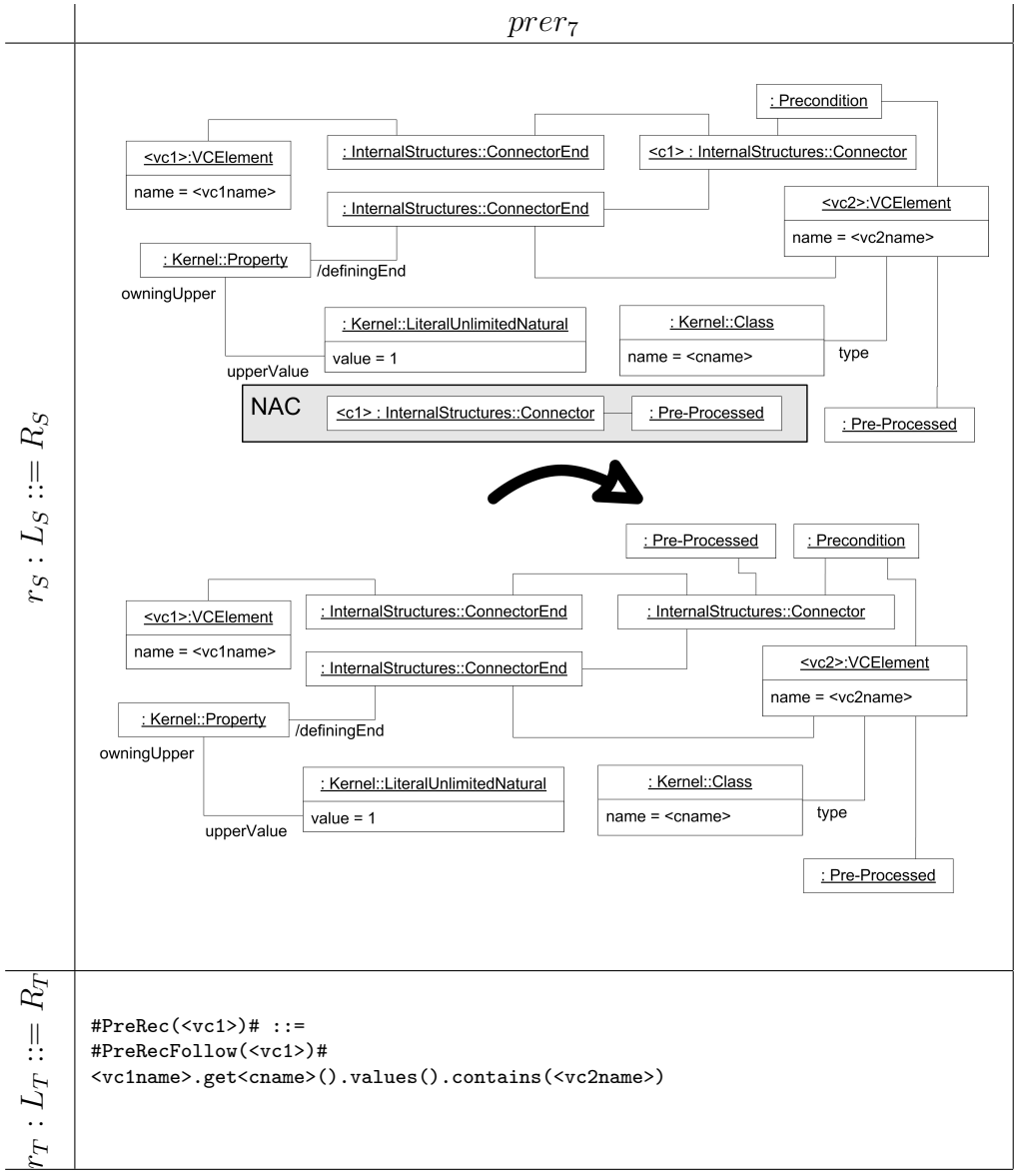


Abbildung B.18: Links zu bereits besuchten Elementen überprüfen (1)

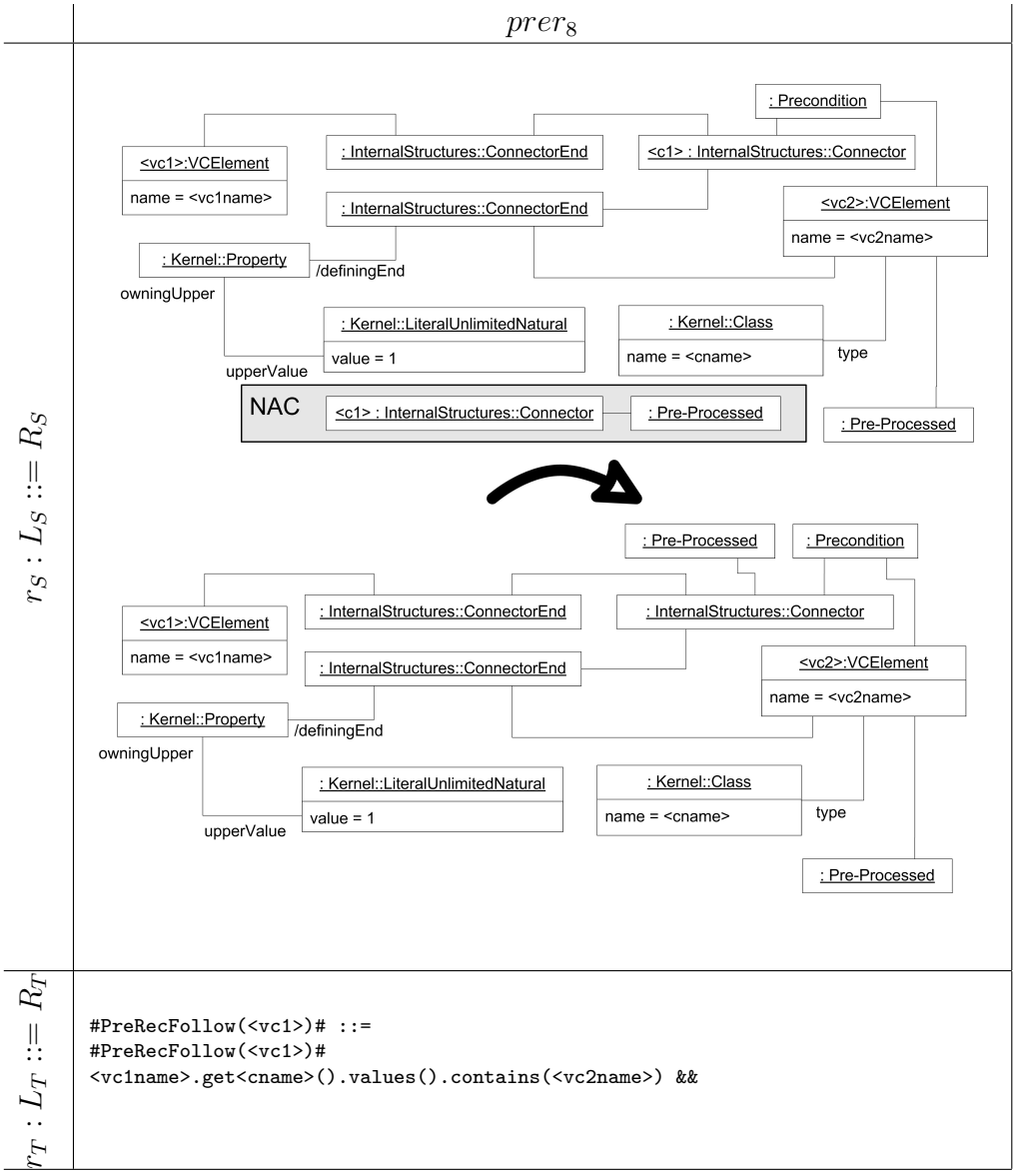


Abbildung B.19: Links zu bereits besuchten Elementen überprüfen (2)

	$r_S : L_S ::= R_S$	$r_T : L_T ::= R_T$
$prer_9$	<vc1>:VCElement	`#PreRecFollow(<vc1>)# ::=` ϵ
$prer_{10}$	<vc1>:VCElement	`#PreRec(<vc1>)# ::=` `true`

Abbildung B.20: Entfernen der Nichtterminale *PreRecFollow*

JML-Code ein `&&` eingefügt wird, wenn ein Blatt gefunden wurde.

B.2.6 Überprüfung der Attributinhalte — Vorbereitung

Beim Aufbau des Suchgraphen ist noch kein JML-Code zur Überprüfung der Attribute einer Rolle erzeugt worden (außer für das Objekt `self` bzw. `this`). Stattdessen wurde bisher nur ein Nichtterminal `PreCheck` als Platzhalter für eine Menge von Attributen, die überprüft werden müssen, eingefügt. In den Regeln $pre_1, \ldots, pre_4$ wird dieses Nichtterminal durch jeweils einen weiteren Platzhalter pro Constraint (d.h. pro zu überprüfendem Attribut) ersetzt. Die detaillierte Prüfung des Inhaltes eines Attributes wird in weiteren Regeln erzeugt. Beim ersten Ersetzen des Nichtterminals $PreCheck(< vc >)$ werden in Abhängigkeit vom referenziertem Inhalt des Attributes entweder pr_1 oder pr_2 aufgerufen. Für jedes weitere Attribut werden dann entweder pr_3 oder pr_4 aufgerufen. Durch diese Aufteilung der Regeln in initiale und folgende Regeln wird eine korrekte Schachtelung der logischen Ausdrücke über Konjunktoren sichergestellt. Nachdem alle Attribute bearbeitet wurden, müssen die von den Regeln $pre_1, \ldots, pre_4$ zusätzlich erzeugten und nicht mehr benötigten Nichtterminale mit der Regel pre_5 wieder gelöscht werden.

Die Compound Rule pre_1 setzt voraus, dass in vorherigen Transformationsschritten ein Nichtterminal `PreCheck(<vc>)` erzeugt wurde. Eine Ausführung der Regeln entfernt dieses Nichtterminal aus dem Zielmodell. Die linke Seite der Quelltransformation setzt voraus, dass im visuellen Kontrakt eine Rolle einen Attributinhalt (`Constraint`) besitzt, der ein elementarer Datentyp ist. Die rechte Seite der Quelltransformation führt keine Änderungen am Modell durch.

Die rechte Seite der Zieltransformation erzeugt einen Pfad zur Abfrage des

Attributinhaltes und fügt ein Nichtterminal `PreConstraint<c>`, welches in einem späteren Transformationsschritt noch aufgelöst wird, um den Code zum Abfragen des Attributinhaltes automatisch zu erzeugen. Das zweite erzeugte Nichtterminal `PreCheckFollow(<vc>)` ist ein Platzhalter, der durch weitere Abfragen von Attributinhalten ersetzt werden kann.

Die Compound Rule pre_2 funktioniert ähnlich. Die Transformationsregel pre_2 wird ausgeführt, wenn der referenzierte Attributinhalt selbst kein elementarer Datentyp sondern ein `VCElement` ist. Entsprechend ändert sich die linke Seite der Quelltransformation, während die rechte Seite der Zieltransformation nicht verändert werden muss. Auch die Transformationsregeln pre_3 und pre_4 funktionieren ähnlich. Sie werden jedoch nicht beim ersten gefundenen Attribut einer Rolle aufgerufen, sondern erst bei allen folgenden Attributen.

Damit ergibt sich folgender Kontrollausdruck für die Ausführung der Regeln $pre_1, \ldots, pre_5$: $\langle\{pre_1, pre_2\}\downarrow, \{pre_3, pre_4\}\downarrow, prer_5\downarrow\rangle$. Mit diesem Kontrollausdruck wird sichergestellt, dass zuerst die Regeln pre_1 und pre_2 in einer nichtdeterministischen Reihenfolge ausgeführt werden, bis sie nicht mehr angewendet werden können. Danach werden die Regeln $prer_3$ und pre_4 solange in einer nichtdeterministischen Reihenfolge ausgeführt werden, bis sie nicht mehr angewendet werden können. Da während der Ausführung der Regeln $pre_1, \ldots, pre_4$ weitere Nichtterminale entstehen, werden diese zum Abschluss mit der Regel pre_5 gelöscht.

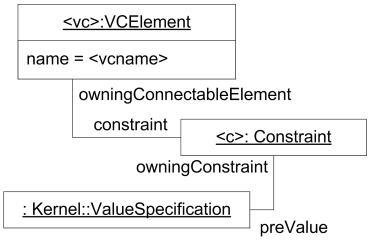
B.3 Überprüfung der Attributinhalte in Vorbedingung auflösen

Die Compound Rules $pat_1, \ldots, pat_6$ erzeugen JML-Code zum Überprüfen der Inhalte eines Attributes einer gebundenen Rolle. Der Code zum Überprüfen des Inhalts eines Attributes ist abhängig vom Datentyp des Attributes. Wir unterscheiden in unseren Compound Rules zwischen den Typen `Integer` und `String`. Für jeden weiteren Datentyp sind drei weitere Regeln einzuführen. Jede Regel ist einen der folgenden drei Fälle zuzuordnen, die wir bereits für die Typen `Integer` und `String` unterscheiden:

1. Der Attributinhalt ist ein konkreter Wert (pat_1, pat_2). Im Quellmodell wird der konkrete Wert an der Klasse `PrimitiveType` erkannt.

2. Der Attributinhalt ist ein Parameter (Klasse `Parameter` im Quellmodell) der Operation, und der Parameter ist ein primitiver Datentyp

	$r_S : L_S ::= R_S$	$r_T : L_T ::= R_T$

pre_1		`#PreCheck(<vc>)# ::=`
		`<vcname>.get#PreConstraint(<c>)#`
		`#PreCheckFollow(<vc>)#`

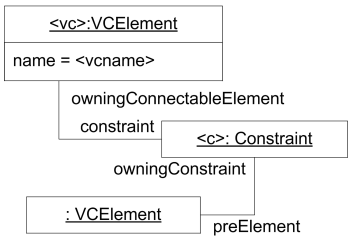
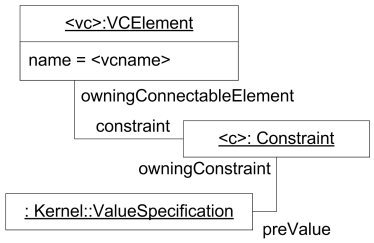
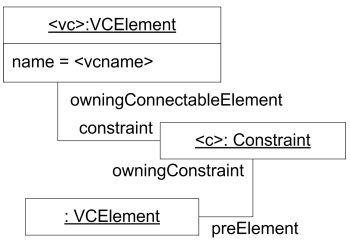
pre_2		`#PreCheck(<vc>)# ::=`
		`<vcname>.get#PreConstraint(<c>)#`
		`#PreCheckFollow(<vc>)#`

Abbildung B.21: Bearbeitung Attributinhalte vorbereiten (1)

	$r_S : L_S ::= R_S$	$r_T : L_T ::= R_T$

pre_3		`#PreCheckFollow(<vc>)# ::=`
		`&&`
		`<vcname>.get#PreConstraint(<c>)#`
		`#PreCheckFollow(<vc>)#`

pre_4		`#PreCheckFollow(<vc>)# ::=`
		`&&`
		`<vcname>.get#PreConstraint(<c>)#`
		`#PreCheckFollow(<vc>)#`

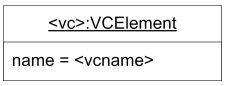
pre_5		`#PreCheckFollow(<vc>)# ::=` ϵ

Abbildung B.22: Bearbeitung Attributinhalte vorbereiten (2)

(pat_3, pat_4).

3. Der Attributinhalt ist eine Variable, die im visuellen Kontrakt verwendet wird (pat_5, pat_6). In diesem Fall wird im Zielmodell ein neues Nichtterminal `Variable(<vname>)` eingefügt, das in späteren Transformationsschritten wieder aufgelöst werden muss.

Die Transformationsregeln produzieren bis auf die datentypspezifischen Vergleichoperatoren nahezu identische Inhalte. Sie komplettieren den in den Regeln $pre_1, \ldots, pre_5$ vorbereiteten Pfadausdruck zum Zugriff auf ein Attribut mit dem konkreten Attributnamen `<sfname>`. Danach folgen die datentypspezifischen Vergleichsoperatoren (`equal` für `String`, `==` für `Integer`) sowie ein Vergleichswert `<lsname>`. Die Regeln `pat_5` und `pat_6` führen statt einem Vergleichswert eine weiteres Nichtterminal ein.

Alle sechs Transformationsregeln können in einer nichtdeterministischen Reihenfolge ausgeführt werden, bis sie nicht mehr angewendet werden können. Dies führt zu dem Kontrollausdruck: $\langle\{pat_1, pat_2, pat_3, pat_4, pat_5, pat_6\}\downarrow\rangle$

B.4 Bearbeitung von Variablen

In einem visuellen Kontrakt können Variablen verwendet werden, um die Inhalte von unterschiedlichen Attributen (in unterschiedlichen Elementen) miteinander zu vergleichen. In den vorherigen Kapiteln sind bereits Nichtterminale in die JML-Spezifikation als Platzhalter für Variablen eingefügt worden. Diese müssen jetzt durch die entsprechenden Pfadausdrücke ersetzt werden. Zur Vereinfachung gehen wir davon aus, dass ein Parameter einer Operation entweder ein elementarer Datentyp (siehe Regeln pat_1, pat_2) oder ein einzelnes Objekt, das keine weiteren Parameter referenziert, ist. Ein Übergabeparameter einer Operation kann also nicht die Wurzel eines Objektbaums sein. Hierfür sind die im folgenden beschriebenen Regeln $pv_1, \ldots, pv_3$ zuständig.

B.4.1 Variable ist Attributinhalt eines Operationsparameters

Bekommt eine Operation ein Objekt als Parameter übergeben, so können auch in den Attributinhalten von diesen Objekten Variablen verwendet werden. In der Regel werden die Variablen in den als Parameter übergebenen Objekten verwendet, um die Inhalte weiterer Elemente in einem visuellen

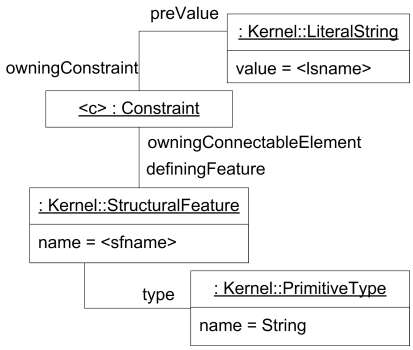
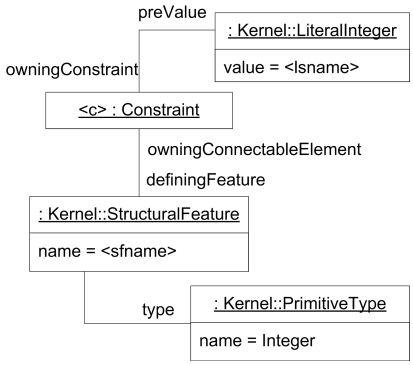
	$r_S : L_S ::= R_S$	$r_T : L_T ::= R_T$
pat_1		`#PreConstraint(<c>)# ::=` `<sfname>().equals(<lsname>)`
pat_2		`#PreConstraint(<c>)# ::=` `<sfname>() == <lsname>`

Abbildung B.23: Überprüfung der Attributinhalte auflösen (1)

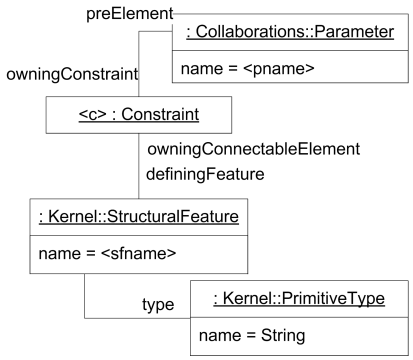
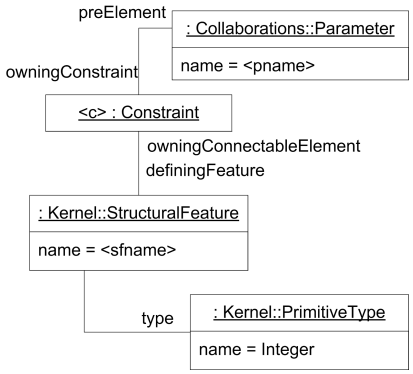
	$r_S : L_S ::= R_S$	$r_T : L_T ::= R_T$
pat_3		`#PreConstraint(<c>)# ::=` `<sfname>().equals(<lsname>)`
pat_4		`#PreConstraint(<c>)# ::=` `<sfname>() == <pname>`

Abbildung B.24: Überprüfung der Attributinhalte auflösen (2)

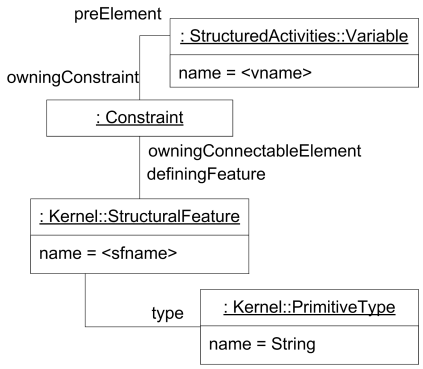
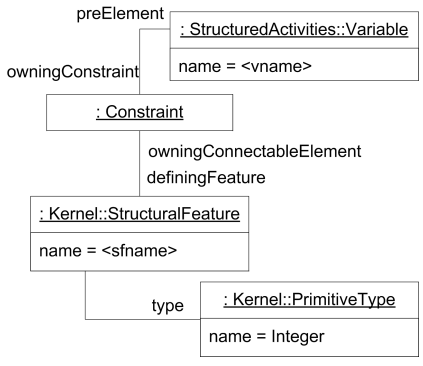
	$r_S : L_S ::= R_S$	$r_T : L_T ::= R_T$
pat_5		`#Pre(<op>)# ::=` `<sfname>().equals(#Variable(<vname>)#)`
pat_6		`#Pre(<op>)# ::=` `<sfname>() == #Variable(<vname>)#`

Abbildung B.25: Überprüfung der Attributinhalte auflösen (3)

Kontrakt weiter einzuschränken. Wenn eine Variable der Attributinhalt eines Objektes ist, welches als Parameter übergeben wird, so können in der JML-Spezifikation die Nichtterminale, die als Platzhalter für eine bestimmte Variable eingesetzt werden, ersetzt werden mit einem Zugriffspfad auf das entsprechende Attribut des Parameterobjektes.

Die linke Seite der Quelltransformation der Transformationsregel pv_1 setzt voraus, dass eine Variable `<vname>` der Inhalt eines Attributes `<sfname>` eines Parameterobjektes ist (siehe Objekte `Parameter`, `Constraint`, `Variable`, `StructuralFeature`).

Die rechte Seite der Quelltransformation führt keine Änderungen am Modell durch. Es wird lediglich ein Objekt `Pre-Processed` mit der Variable `<vname>` verbunden. Dadurch ist in späteren Transformationsschritten die Information verfügbar, dass diese Variable bereits abgearbeitet wurde.

Die rechte Seite der Zieltransformation erzeugt einen Pfadausdruck zum Zugriff auf den Inhalt des Attributes. Der Pfadausdruck ist eine einfache `get()`-Operation auf dem übergebenen Parameterobjekt `<pname>`. Zur Laufzeit gibt dieser Ausdruck den Inhalt des Attributes `<sfname>` zurück und repräsentiert damit den Referenzinhalt für die Variable `vname`.

B.4.2 Variable ist kein Attributinhalt eines Operationsparameters

Wenn die Variable kein Inhalt eines Attributes eines Operationsparameters ist, so ist der Referenzinhalt einer Variablen aus einem Laufzeitobjekt, welches eine Rolle einer Vorbedingung eines visuellen Kontraktes einnimmt, zu bestimmen. Hierfür sind die Compound Rules pv_2 und pv_3 zuständig.

Die Compound Rule pv_2 ist für jede Variable `<vname>` einmal auszuführen. Die Regel legt den Referenzinhalt für ein Attribut fest. Die linke Seite der Quelltransformation setzt voraus, dass eine Variable `<vname>` über einen `Constraint` an den Inhalt eines Attributes `<sfname>` einer Rolle `<dpname>` gebunden ist. Die negative Anwendungsbedingung stellt sicher, dass ein einmal erstellter Referenzinhalt für eine Variable nicht wieder überschrieben wird.

Die rechte Seite der Quelltransformation nimmt keine Änderungen am Modell vor. Es wird lediglich ein Objekt `Pre-Processed` mit dem Objekt `<var>` verbunden. Damit steht in späteren Transformationen die Information zur Verfügung, dass dieses Objekt bereits bearbeitet wurde. Weiterhin besitzt

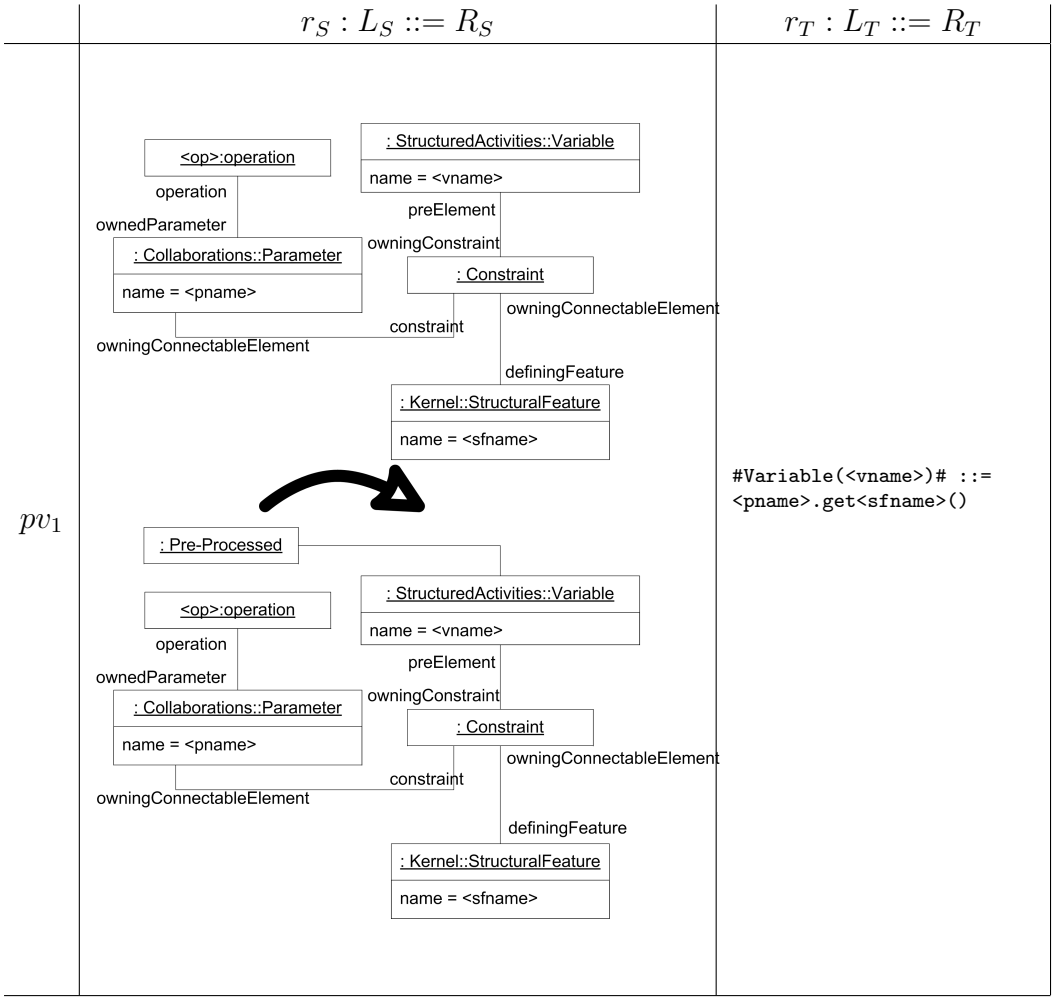


Abbildung B.26: Variable ist Attributinhalt eines Operationsparameters

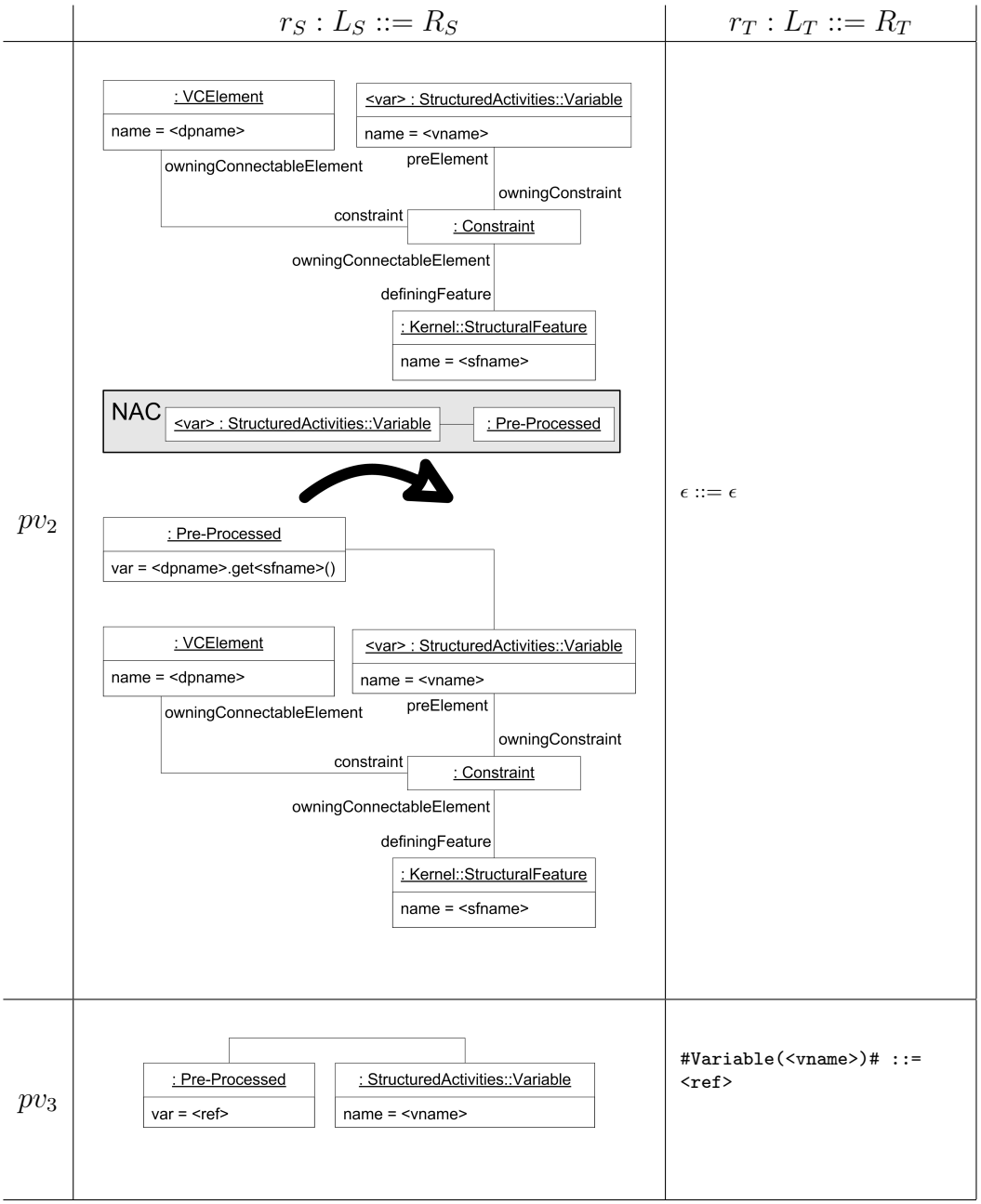


Abbildung B.27: Referenzinhalt für Variable bestimmen und einfügen

das Objekt `Pre-Processed` ein Attribut `var`. In diesem Attribut wird die Zugriffsmöglichkeit auf ein Attribut `<sfname>` eines Laufzeitobjektes, welches an die Rolle `<dpname>` gebunden ist, gespeichert. Bei dieser Zugriffsmöglichkeit handelt es sich um eine einfache `get`-Operation. Damit kann in späteren Transformationen auf den mit dieser Transformation festgelegten Referenzinhalt für eine Variable `<vname>` zurückgegriffen werden.

Auf dem Zielmodell werden während der Bestimmung der Referenzinhalte für die Variablen keine Änderungen durchgeführt.

Nachdem für alle Variablen die Referenzinhalte bestimmt wurden, können diese mit der Compound Rule pv_3 in das Zielmodell eingefügt werden. Die linke Seite der Quelltransformation setzt voraus, dass für eine Variable `<vname>` bereits ein Referenzinhalt `<ref>` gefunden wurde. Dieser ist im Attribut `var` des Objektes `Pre-Processed` gespeichert.

Die rechte Seite der Zieltransformation ersetzt die in vorherigen Regeln als Platzhalter eingeführten Nichtterminale `Variable(<vname>)` durch den Referenzinhalt `<ref>`. Die Referenzinhalte in den JML-Konstrukten verwenden Variablen, die zur Laufzeit über die `exists`-Quantoren an Objekte gebunden sind. Die Referenzinhalte werden an beliebigen Stellen im JML-Ausdruck eingefügt. Bei den hier beschriebenen Compound Rules haben wir nicht berücksichtigt, ob eine Variable, die in einem JML-Ausdruck eingefügt wird, auch an der entsprechenden Stelle bereits bekannt ist. Damit die generierten JML-Ausdrücke korrekt funktionieren muss der JML-Ausdruck der Operation nach der Transformation noch in eine pränexe Normalform gebracht werden. Das heißt, alle Quantoren sind nach vorne zu ziehen. Dann besteht der JML-Ausdruck aus einem Präfix, der alle Quantoren enthält und einem Postfix, der quantorenfrei ist. Die Verschiebung der Quantoren ist möglich, da die Variablen, die mit den Quantoren an Laufzeitobjekte gebunden werden, nicht frei sind. Das heißt die Variablen kommen nicht außerhalb des Wirkungsbereichs eines Quantors in einem JML-Ausdruck vor. Dies kann sichergestellt werden, indem ein Rollenname in einem visuellen Kontrakt nicht zweimal verwendet werden darf. Im Metamodell wird eine Rolle mit einem `VCElement` repräsentiert. Ein `VCElement` spezialisiert ein `ConnectableElement`, welches wiederum ein `NamedElement` spezialisiert. Ein `NamedElement` besitzt ein Attribut `name`, das gemäß UML-Spezifikation [OMG04] ein Identifier ist. Da wir aus diesem Attribut `name` Variablen generieren, sind auch die Variablen in der generierten JML-Spezifikation eindeutig und werden immer nur für einen Quantor verwendet. Ein Verschieben der Quantoren ist also möglich.

Der folgende Kontrollausdruck steuert die Ausführung der Regeln $pv_1, \ldots,$

pv_3: $\langle pv_1 \downarrow, pv_2 \downarrow, pv_3 \downarrow\rangle$. Mit diesem Kontrollausdruck wird sichergestellt, dass zuerst die Regel pv_1 ausgeführt wird, bis sie nicht mehr angewendet werden kann. Danach wird die Regel pv_1 solange ausgeführt werden, bis sie nicht mehr angewendet werden kann. Zum Abschluss werden die Nichtterminale zur Repräsentation von Variablen mit dem berechneten Pfadausdruck ersetzt (pv_3).

Anhang C

DAML+OIL

C.1 DAML+OIL Ontologie

```
 1 <?xml version="1.0"?>
 2
 3 <rdf:RDF
 4    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 5    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
 6    xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
 7    xmlns:xsd="http://www.w3.org/200/10/XMLSchema#"
 8    xmlns:dex="http://www.daml.org/2001/03/daml+oil-ex#"
 9    xmlns:exd="http://www.daml.org/2001/03/daml+oil-ex-dt#"
10    xmlns="http://shopOntology">
11
12 <!-- ######################################################
13       Ontology
14       ###################################################### -->
15   <daml:Ontology rdf:ID="shopOntology">
16      <daml:versionInfo>1.0</daml:versionInfo>
17        <rdfs:comment>Ontology for selling products
18        </rdfs:comment>
19      <daml:imports rdf:resource=
20          "http://www.daml.org/2001/03/daml+oil"/>
21   </daml:Ontology>
22
23 <!-- ######################################################
24        Class: Order
25       ###################################################### -->
```

```
26  <daml:Class rdf:ID="Order">
27    <rdfs:label>Order</rdfs:label>
28    <rdfs:comment>Class Order</rdfs:comment>
29    <!-- Association c -->
30    <rdfs:subClassOf>
31      <daml:Restriction>
32        <daml:onProperty rdf:resource="#c"/>
33        <daml:toClass rdf:resource="#Cart"/>
34        <daml:minCardinality>0</daml:minCardinality>
35        <daml:maxCardinality>1</daml:maxCardinality>
36      </daml:Restriction>
37    </rdfs:subClassOf>
38    <!-- Association oinv -->
39    <rdfs:subClassOf>
40      <daml:Restriction>
41        <daml:onProperty rdf:resource="#oinv"/>
42        <daml:toClass rdf:resource="#Invoice"/>
43        <daml:cardinality>1</daml:cardinality>
44      </daml:Restriction>
45    </rdfs:subClassOf>
46    <!-- Association da -->
47    <rdfs:subClassOf>
48      <daml:Restriction>
49        <daml:onProperty rdf:resource="#da"/>
50        <daml:toClass rdf:resource="#Customer"/>
51        <daml:cardinality>1</daml:cardinality>
52      </daml:Restriction>
53    </rdfs:subClassOf>
54  </daml:Class>
55
56 <!-- ######################################################
57       Class: Invoice
58      ###################################################### -->
59  <daml:Class rdf:ID="Invoice">
60    <rdfs:label>Invoice</rdfs:label>
61    <rdfs:comment>Class Invoice</rdfs:comment>
62    <!-- Association payBy -->
63    <rdfs:subClassOf>
64      <daml:Restriction>
65        <daml:onProperty rdf:resource="#payBy"/>
66        <daml:toClass rdf:resource="#Payment"/>
67        <daml:minCardinality>0</daml:minCardinality>
68        <daml:maxCardinality>1</daml:maxCardinality>
```

```
69          </daml:Restriction>
70        </rdfs:subClassOf>
71        <!-- Association oinv -->
72        <rdfs:subClassOf>
73          <daml:Restriction>
74            <daml:onProperty rdf:resource="#oinv"/>
75            <daml:toClass rdf:resource="#Order"/>
76            <daml:cardinality>1</daml:cardinality>
77          </daml:Restriction>
78        </rdfs:subClassOf>
79      </daml:Class>
80
81  <!-- ////////////////////////////////////////////////////////
82          Class: Cart
83        //////////////////////////////////////////////////////// -->
84      <daml:Class rdf:ID="Cart">
85        <rdfs:label>Cart</rdfs:label>
86        <rdfs:comment>Class Cart</rdfs:comment>
87        <!-- Association items -->
88        <rdfs:subClassOf>
89          <daml:Restriction>
90            <daml:onProperty rdf:resource="#items"/>
91            <daml:toClass rdf:resource="#CartItem"/>
92            <daml:minCardinality>0</daml:minCardinality>
93          </daml:Restriction>
94        </rdfs:subClassOf>
95        <!-- Association c -->
96        <rdfs:subClassOf>
97          <daml:Restriction>
98            <daml:onProperty rdf:resource="#c"/>
99            <daml:toClass rdf:resource="#Order"/>
100           <daml:minCardinality>0</daml:minCardinality>
101           <daml:maxCardinality>1</daml:maxCardinality>
102         </daml:Restriction>
103       </rdfs:subClassOf>
104     </daml:Class>
105
106 <!-- ////////////////////////////////////////////////////////
107         Class: CartItem
108       //////////////////////////////////////////////////////// -->
109     <daml:Class rdf:ID="CartItem">
110       <rdfs:label>CartItem</rdfs:label>
111       <rdfs:comment>Class CartItem</rdfs:comment>
```

```
112     <!-- Association pr -->
113     <rdfs:subClassOf>
114       <daml:Restriction>
115         <daml:onProperty rdf:resource="#pr"/>
116         <daml:toClass rdf:resource="#Product"/>
117         <daml:cardinality>1</daml:cardinality>
118       </daml:Restriction>
119     </rdfs:subClassOf>
120     <!-- Association items -->
121     <rdfs:subClassOf>
122       <daml:Restriction>
123         <daml:onProperty rdf:resource="#items"/>
124         <daml:toClass rdf:resource="#Cart"/>
125         <daml:cardinality>1</daml:cardinality>
126       </daml:Restriction>
127     </rdfs:subClassOf>
128   </daml:Class>
129
130 <!-- ###################################################
131        Class: Product
132       ################################################### -->
133   <daml:Class rdf:ID="Product">
134     <rdfs:label>Product</rdfs:label>
135     <rdfs:comment>Class Product</rdfs:comment>
136     <!-- Association pr -->
137     <rdfs:subClassOf>
138       <daml:Restriction>
139         <daml:onProperty rdf:resource="#pr"/>
140         <daml:toClass rdf:resource="#CartItem"/>
141         <daml:minCardinality>0</daml:minCardinality>
142       </daml:Restriction>
143     </rdfs:subClassOf>
144   </daml:Class>
145
146 <!-- ###################################################
147        Class: Payment
148       ################################################### -->
149   <daml:Class rdf:ID="Payment">
150     <rdfs:label>Payment</rdfs:label>
151     <rdfs:comment>Class Payment</rdfs:comment>
152     <!-- Association payBy -->
153     <rdfs:subClassOf>
154       <daml:Restriction>
```

```
155           <daml:onProperty rdf:resource="#payBy"/>
156           <daml:toClass rdf:resource="#Invoice"/>
157           <daml:minCardinality>0</daml:minCardinality>
158           <daml:maxCardinality>1</daml:maxCardinality>
159         </daml:Restriction>
160       </rdfs:subClassOf>
161     </daml:Class>
162
163 <!-- ###########################################################
164         Class: BankAccount
165       ########################################################### -->
166     <daml:Class rdf:ID="BankAccount">
167       <rdfs:label>BankAccount</rdfs:label>
168       <rdfs:comment>Class BankAccount</rdfs:comment>
169       <daml:subClassOf rdf:resource="#Payment"/>
170     </daml:Class>
171
172 <!-- ###########################################################
173         Class: CreditCard
174       ########################################################### -->
175     <daml:Class rdf:ID="CreditCard">
176       <rdfs:label>CreditCard</rdfs:label>
177       <rdfs:comment>Class CreditCard</rdfs:comment>
178       <daml:subClassOf rdf:resource="#Payment"/>
179     </daml:Class>
180
181 <!-- ###########################################################
182         Class: Customer
183       ########################################################### -->
184     <daml:Class rdf:ID="Customer">
185       <rdfs:label>Customer</rdfs:label>
186       <rdfs:comment>Class Customer</rdfs:comment>
187       <!-- Association da -->
188       <rdfs:subClassOf>
189         <daml:Restriction>
190           <daml:onProperty rdf:resource="#da"/>
191           <daml:toClass rdf:resource="#Order"/>
192           <daml:cardinality>1</daml:cardinality>
193         </daml:Restriction>
194       </rdfs:subClassOf>
195     </daml:Class>
196
197 <!-- ###########################################################
```

```
198       Associations
199      ##################################################### -->
200
201 <!-- Association: c -->
202   <daml:ObjectProperty rdf:ID="c">
203     <rdfs:comment>
204       Property for association "c"
205     </rdfs:comment>
206     <rdfs:domain rdf:resource="#Order"/>
207     <rdfs:range  rdf:resource="#Cart"/>
208   </daml:ObjectProperty>
209
210 <!-- Association: oinv -->
211   <daml:ObjectProperty rdf:ID="oinv">
212     <rdfs:comment>
213       Property for association "oinv"
214     </rdfs:comment>
215     <rdfs:domain rdf:resource="#Order"/>
216     <rdfs:range  rdf:resource="#Invoice"/>
217   </daml:ObjectProperty>
218
219 <!-- Association: items -->
220   <daml:ObjectProperty rdf:ID="items">
221     <rdfs:comment>
222       Property for association "items"
223     </rdfs:comment>
224     <rdfs:domain rdf:resource="#Cart"/>
225     <rdfs:range  rdf:resource="#CartItem"/>
226   </daml:ObjectProperty>
227
228 <!-- Association: pr -->
229   <daml:ObjectProperty rdf:ID="pr">
230     <rdfs:comment>
231       Property for association "pr"
232     </rdfs:comment>
233     <rdfs:domain rdf:resource="#CartItem"/>
234     <rdfs:range  rdf:resource="#Product"/>
235   </daml:ObjectProperty>
236
237 <!-- Association: da -->
238   <daml:ObjectProperty rdf:ID="da">
239     <rdfs:comment>
240       Property for association "da"
```

```
241       </rdfs:comment>
242       <rdfs:domain rdf:resource="#Customer"/>
243       <rdfs:range  rdf:resource="#Order"/>
244     </daml:ObjectProperty>
245
246 <!-- Association: payBy -->
247     <daml:ObjectProperty rdf:ID="payBy">
248       <rdfs:comment>
249         Property for association "payBy"
250       </rdfs:comment>
251       <rdfs:domain rdf:resource="#Invoice"/>
252       <rdfs:range  rdf:resource="#Payment"/>
253     </daml:ObjectProperty>
254 </rdf:RDF>
```

Listing C.1: DAML+OIL Repräsentation der Ontologie aus Abbildung 6.2

C.2 DAML+OIL Kontrakt

```
 1 <?xml version="1.0"?>
 2
 3 <rdf:RDF
 4     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 5     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
 6     xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
 7     xmlns:xsd="http://www.w3.org/200/10/XMLSchema#"
 8     xmlns:dex="http://www.daml.org/2001/03/daml+oil-ex#"
 9     xmlns:exd=
10           "http://www.daml.org/2001/03/daml+oil-ex-dt#">
11
12
13     <Rule rdf:ID="contract_orderCartProvider">
14       <hasLeftSide  rdf:resource=
15             "#contract_orderCartProvider_Precondition"/>
16       <hasRightSide rdf:resource=
17             "#contract_orderCartProvider_Postcondition"/>
18     </Rule>
19
20     <Left rdf:ID="contract_orderCartProvider_Precondition">
21       <CreditCard rdf:ID="ccp">
22       </CreditCard>
23       <Cart rdf:ID="cp">
24       </Cart>
```

```
25      <Customer rdf:ID="cup">
26      </Customer>
27    </Left>
28
29    <Right rdf:ID=
30                "contract_orderCartProvider_Postcondition">
31      <CreditCard rdf:ID="ccp">
32        <payBy rdf:resource="#ip"/>
33      </CreditCard>
34      <Cart rdf:ID="cp">
35        <c rdf:resource="#op"/>
36      </Cart>
37      <Customer rdf:ID="cup">
38        <da rdf:resource="#op"/>
39      </Customer>
40      <Order rdf:ID="op">
41        <c rdf:resource="#cp"/>
42        <da rdf:resource="#cup"/>
43        <oinv rdf:resource="#ip"/>
44      </Order>
45      <Invoice rdf:ID="ip">
46        <oinv rdf:resource="#op"/>
47        <payBy rdf:resource="#ccp"/>
48      </Invoice>
49    </Right>
50
51 </rdf:RDF>
```

Listing C.2: DAML+OIL Repräsentation des visuellen Kontraktes aus Abbildung 6.3

Literaturverzeichnis

[ACD+03] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, Ivana Trickovic, and Sanjivac Weerawarana. Business process execution language for web services version 1.1. Technical report, BEA Systems IBM Microsoft SAP AG Siebel Systems, 5 May 2003 2003.

[ACKM04] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijy Machiraju. *Web Services: Concepts, Architectures and Applications*. Data-Centric Systems and Applications. Springer-Verlag Berlin Heidelberg New York, 2004.

[ACO06] ACORD. Gobal insurance standards, 2006.

[AEH+96] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. Graph transformation for specification and programming. Technical Report 7/96, Universität Bremen, 1996.

[AEH+99] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. Graph transformation for specification and programming. *Science of Computer Programming*, 34(1):1–54, 1999.

[AFM+05] Rama Akkiraju, Joel Farrell, John Miller, Meenakshi Nagarajan, Marc-Thomas Schmidt, Amit Sheth, and Kunal Verma. Web service semantics - WSDL-S - version 1.0, April 2005.

[AGB+04] Daniel Austin, W. W. Grainger, Abbie Barbir, Christopher Ferris, and Sharad Garg. Web services architecture requirements - W3C working group note 11 february 2004, 2004.

[Agr03] Aditya Agrawal. Graph rewriting and transformation (GReAT): A solution for the model integrated computing (MIC) bottleneck. In *18th IEEE International Conference on Automated Software Engineering (ASE 2003)*, pages 364–368. IEEE Computer Society, 2003.

[AH00] Sergio Antoy and Dick Hamlet. Automatically checking an implementation against its formal specification. *IEEE Transactions on Software Engineering*, 26(1):55–69, January 2000.

[AK02] David Akehurst and Stuart Kent. A relational approach to defining transformations in a metamodel. In Jean-Marc Jézéquel and Heinrich Hussmann, editors, *UML 2002*, LNCS 2460, pages 243–258. Springer-Verlag Berlin Heidelberg, 2002.

[Ama05a] Amazon Services, Inc. Amazon e-commerce service: Developer guide, March 2005.

[Ama05b] Amazon Services, Inc. Amazon web services (aws), 2005. `http://www.amazon.com/webservices`.

[Amb01] Scott W. Ambler. 'an overview of object relationships', 'unidirectional object relationships', 'implementing one-to-many object relationships', 'implementing many-to-many object relationships'. a series of tips to be found at ibm developer works, 2001. `http://www-106.ibm.com/developerworks/`.

[Amb03] Scott Ambler. *Agile Database Techniques : Effective Strategies for the Agile Software Developer*. Wiley, October 2003.

[And] Andromda. `http://www.andromda.org/`.

[Bau04] Alexander Bausch. Plattformunabhängige Modelle mit Hilfe von Java Servlets ausführen, September 2004. Bachelorarbeit Universität Paderborn.

[BCC+03] Lilian Burdy, Yoonsik Cheon, David Cok, Michael D. Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan, M. Leino, and Erik Poll. An overview of JML tools and applications. In Thomas Arts and Wan Fokkink, editors, *Eighth International Workshop onFormal Methods for Industrial Critical Systems (FMICS'03)*, volume ENTCS, Røros, Norway, 2003. Elsevier Science.

[BCC+05] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An

overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, February 2005.

[Bec04] Dave Beckett. Rdf/xml syntax specification (revised) - W3C recommendation 10 february 2004, February 2004.

[Beh03] Wernher Behrendt. *Web Engineering: Systematische Entwicklung von Web-Anwendungen*, chapter Semantisches Web - Das Netz der Bedeutungen im Netz der Dokumente, pages 345–368. In Kappel et al. [KPRR03], October 2003.

[Bel02] Tom Bellwood. UDDI version 2.04 API specification. Technical report, OASIS, July 2002.

[BEL+03] Roswitha Bardohl, Hartmut Ehrig, Juan de Lara, Olga Runge, Gabi Taentzer, and Ingo Weinhold. Node type inheritance concept for typed graph transformation. Technical report, TU Berlin, Forschungsberichte des Fachbereichs Informatik, 2003.

[BELT04] Roswitha Bardohl, Hartmut Ehrig, Juan de Lara, and Gabriele Taentzer. Integrating meta-modelling aspects with graph transformation for efficient visual language definition and model manipulation. In Michel Wermelinger and Tiziana Margaria, editors, *Fundamental Approaches to Software Engineering, 7th International Conference, FASE 2004*, volume 2984 of *Lecture Notes in Computer Science*, Barcelona, Spain, 2004.

[BFM02] Christoph Bussler, Dieter Fensel, and Alexander Maedche. A conceptual architecture for semantic web enabled web services. *ACM SIGMOD Record*, 31(4):24 – 29, December 2002.

[BFMW01] Detlef Bartetzko, Clemens Fischer, Michael Möller, and Heike Wehrheim. Jass - java with assertions. volume 55 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2001.

[BG] Kent Beck and Erich Gamma. Junit. `http://www.junit.org`.

[BG98] Kent Beck and Erich Gamma. Junit test infected: Programmers love writing tests. *Java Report*, 3(7):37–50, 1998.

[BG04] Dan Brickley and R.V. Guha. RDF vocabulary description language 1.0: RDF schema - W3C recommendation 10 february 2004, February 2004.

[BGST05] Sven Burmester, Holger Giese, Andreas Seibel, and Matthias Tichy. Worst-case execution time optimization of story patterns

for hard real-time systems. In *Proc. of the 3rd International Fujaba Days 2005, Paderborn, Germany*, pages 71–78, September 2005.

[BHM+04] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Champion. Michael, Chris Ferris, and David Orchard. Web services architecture - W3C working group note 11 february 2004. Technical report, W3C, 2004.

[BHRT03] Boualem Benatallah, Mohand-Said Hacid, Christophe Rey, and Farouk Toumani. Semantic reasoning for web services discovery. In *WWW 2003 Workshop on E-Services and the Semantic Web (ESSW' 03)*, Budapest, Hungary, 2003.

[Bin00] Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns and Tools.* Addison-Wesley Longman Publishing Co., Inc., 2000.

[BKK+01] Kenneth Baclawski, Mieczyslaw K. Kokar, Paul A. Kogut, Lewis Hart, Jeffrey Smith, William S. Holmes III, Jerzy Letkowski, and Michael L. Aronson. Extending uml to support ontology engineering for the semantic web. In Martin Gogolla and Cris Kobryn, editors, *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools - 4th International Conference*, pages 342–360, Toronto, Canada, 2001. Springer-Verlag Berlin Heidelberg.

[BKK+02] Kenneth Baclawski, Mieczyslaw K. Kokar, Paul A. Kogut, Lewis Hart, Jeffrey Smith, Jerzy Letkowski, and Pat Emery. Extending the unified modeling language for ontology development. *Software and Systems Modeling*, 2002.

[BKKK87] Jay Banerjee, Won Kim, Hyoung-Joo Kim, and Henry F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In Umeshwar Dayal, editor, *International Conference on Management of Data*, pages 311–322, San Francisco, California, United States, 1987. ACM Press.

[BKPPT00] Paolo Bottoni, Manuel Koch, Francesco Parisi-Presicce, and Gabriele Taentzer. Consistency checking and visualization of OCL constraints. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October*

2000, Proceedings, volume 1939 of *LNCS*, pages 294–308, York, 2000. Springer.

[BKPPT01] Paolo Bottoni, Manuel Koch, Francesco Parisi-Presicce, and Gabriele Taentzer. A visualization of OCL using collaborations. In Martin Gogolla and Cris Kobryn, editors, *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, volume 2185 of *Lecture Notes In Computer Science*, pages 257–271. Springer-Verlag, 2001.

[BKPPT02] Paolo Bottoni, Manuel Koch, Francesco Parisi-Presicce, and Gabi Taentzer. Working on OCL with graph transformation. In H.J.Kreowski and P.Knirsch, editors, *Applied Graph Transformation (AGT'02)*, pages 1–10, 2002.

[BL94] T. Berners-Lee. Universal resource identifiers in WWW - request for comments 1630, June 1994.

[BL98] Tim Berners-Lee. Semantic web road map, September 1998.

[BL01] Lionel Briand and Yvan Labiche. A UML-based approach to system testing. In Martin Gogolla and Cris Kobryn, editors, *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools. 4th International Conference, Toronto, Canada, October 2001, Proceedings*, volume 2185 of *LNCS*, pages 194–208. Springer, 2001.

[BL02] Lionel Briand and Yvan Labiche. A UML-based approach to system testing. Technical Report TR SCE-01-01- Version 4, Software Quality Engineering Laboratory Systems and Computer Engineering Department, Carleton University, 1125 Colonel By Drive, Ottawa, Canada, K1S 5B6, 2002.

[BL05] David Booth and Canyang Kevin Liu. Web services description language (WSDL) version 2.0 part 0: Primer - W3C working draft 10 may 2005. Technical report, World Wide Web Consortium, May 2005. http://www.w3.org/TR/2005/WD-wsdl20-primer-20050510/.

[BLFM05] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifier (URI): Generic syntax - request for comments 3986, January 2005.

[BLHL01] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5):34–43, May 2001.

[BLS04] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The spec# programming system: An overview. In *CASSIS 2004*, volume 3362 of *LNCS*. Springer-Verlag, 2004.

[b+m] b+m ArchitectureWare. b+m generator framework. `http://www.architectureware.de/produkte/generator_framework.htm`.

[Boo03] Paul Boocock. Jamda model compiler framework - version 0.2, 2003. `http://jamda.sourceforge.net/docs/index.html`.

[Bor] Borland. Borland together technologies. `www.borland.com/together/`.

[Box97] Don Box. *Essential COM*. Addison-Wesley Professional, December 1997.

[BPM04] Paul V. Biron, Kaiser Permanente, and Ashok Malhotra. XML Schema part 2: Datatypes second edition - W3C recommendation 28 october 2004, October 2004.

[BPSM+04] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (XML) 1.0 (third edition) - W3C recommendation 04 february 2004, 2004.

[Bra04] Gilad Bracha. Generics in the java programming language, July 2004.

[BvHH+04] Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, Lynn Andrea Stein, and Franklin W. Olin. OWL web ontology language - reference - W3C recommendation 10 february 2004, February 2004. `http://www.w3.org/TR/owl-ref/`.

[CER02] Francisco Curbera, David Ehnebuske, and Dan Rogers. Using WSDL in a UDDI registry, version 1.07 UDDI best practice. Technical report, May 2002.

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *Proceedings of the OOPSLA'03 Workshop on the Generative Techniques in the Context Of Model-Driven Architecture*, 2003.

[Che06] Alexey Cherchago. *Service Specification and Matching based on Graph Transformation.* PhD thesis, University of Paderborn, 2006.

[CHL+05] Roberto Chinnici, Hugo Haas, Amy Lewis, Jean-Jacques Moreau, David Orchard, and Sanjiva Weerawarana. Web services description language (WSDL) version 2.0 part 2: Adjuncts - W3C working draft 10 may 2005, May 2005. http://www.w3.org/TR/2005/WD-wsdl20-adjuncts-20050510/.

[CHM+02] György Csertán, Gábor Huszerl, István Majzik, Zsigmond Pap, András Pataricza, and Dániel Varró. VIATRA: Visual automated transformations for formal verification and validation of UML models. In Julian Richardson, Wolfgang Emmerich, and Dave Wile, editors, *Proc. ASE 2002: 17th IEEE International Conference on Automated Software Engineering*, pages 267–270, Edinburgh, UK, September 23–27 2002. IEEE Press.

[CHRR04] Luc Clement, Andrew Hately, Claus von Riegen, and Tony Rogers. UDDI version 3.0.2 - UDDI spec technical committee draft, dated 20041019. Technical report, OASIS, October 2004.

[CL02a] Yoonsik Cheon and Gary T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In Hamid R. Arabnia and Youngsong Mun, editors, *Proceedings of the International Conference on Software Engineering Research and Practice (SERP '02)*, pages 322–328. Computer Science Research, Education, and Applications (CSREA) Press, June 2002.

[CL02b] Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In Boris Magnusson, editor, *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP)*, volume 2374 of *Lecture Notes in Computer Science*, pages 231–255. Springer-Verlag, 2002.

[Cla99] James Clark. XSL transformations (XSLT) version 1.0 - W3C recommendation 16 november 1999, November 1999. `http://www.w3.org/TR/1999/REC-xslt-19991116`.

[CMR96] A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26(3,4):241–266, 1996.

[CMR+97] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*, chapter Algebraic Approaches to Graph Transformation Part I: Basic Concepts and Double Pushout Approach, pages 163–245. World Scientific, 1997.

[CMRW05] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana. Web services description language (WSDL) version 2.0 part 1: Core language - W3C working draft 10 may 2005, May 2005. http://www.w3.org/TR/2005/WD-wsdl20-20050510/.

[Coa04] The OWL Services Coalition. DAML-S and related technologies, 2004.

[Cod] Codagen Technologies Inc. Codagen architect. `http://www.codagen.com/products/architect/default.htm`.

[Com] Compuware. Optimalj. `http://www.compuware.com/products/optimalj/`.

[CP99] Stephen Cranefield and Martin Purvis. UML as an ontology modelling language. In *Proceedings of the Workshop on Intelligent Information Integration, 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, 1999.

[Cra01] Stephen Cranefield. UML and the Semantic Web. In *The first Semantic Web Working Symposium*, 2001.

[CvHH+01] Dan Connolly, Frank van Harmelen, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn A. Stein. DAML+OIL (march 2001) reference description - W3C note 18 december 2001, March 2001.

[DAM] DAML Group. DAML Validator. `http://www.daml.org/validator/`.

[DCLK02] Asuman Dogac, Ibrahim Cingil, Gokce Laleci, and Yildiray Kabak. Improving the functionality of UDDI registries through web service semantics. In Alejandro P. Buchmann, Fabio Casati, Ludger Fiege, Mei-Chun Hsu, and Ming-Chien Shan, editors, *TES '02: Proceedings of the Third International Workshop on Technologies for E-Services*, volume Lecture Notes In Computer Science; Vol. 2444, pages 9–18, Hong Kong, China, 2002. Springer-Verlag.

[DGD05] Dragan Djuric, Dragan Gasevic, and Vladan Devedzic. Ontology modeling and MDA. *Journal of Object Technology*, vol. 4(no. 1):109–128, 2005.

[DHO01] Birgit Demuth, Heinrich Hussmann, and Sven Obermaier. Experiments with XMI based transformations of software models. In *WTUML: Workshop on Transformations in UML, ETAPS 2001 Satellite Event*, 2001.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.

[DOH+01] Marlon Dumas, Justin O'Sullivan, Mitra Heravizadeh, David Edmond, and Arthur ter Hofstede. Towards a semantic framework for service description. In Robert Meersman, Karl Aberer, and Tharam S. Dillon, editors, *Semantic Issues in E-Commerce Systems, IFIP TC2/WG2.6 9th Working Conference on Database Semantics*, volume 239 of *IFIP Conference Proceedings*, pages 277–291, Hong Kong, 2001. Kluwer.

[Dre] Dresden University of Technology. OCL compiler. `http://dresden-ocl.sourceforge.net/`.

[DSB+04] Mike Dean, Guus Schreiber, Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. OWL web ontology language reference - W3C recommendation 10 february 2004, February 2004.

[Ecl06a] Eclipse Consortium. Eclipse graphical editing framework (GEF) - version 3.1.1. http://www.eclipse.org/gef/, 2006.

[Ecl06b] Eclipse Consortium. Eclipse modeling framework (EMF) - version 2.1.2. http://www.eclipse.org/emf/, 2006.

[Ecl06c] Eclipse Consortium. Java emitter templates (JET). Eclipse Modeling Framework (EMF) - Version 2.1.1, http://www.eclipse.org/emf/, 2006.

[EEK99] H. Ehrig, G. Engels, and H. J. Kreowski, editors. *Handbook of Graph Grammars and Computing by Graph Transformations: Applications, Languages and Tools Volume 2*. World Scientific Publishing Company, October 1999.

[EGJ+06] Gregor Engels, Baris Güldali, Oliver Juwig, Marc Lohmann, and Jan-Peter Richter. Industrielle Fallstudie: Einsatz visueller Kontrakte in serviceorientierten Architekturen. In Bettina Biel, Matthias Book, and Volker Gruhn, editors, *Software Enginneering 2006, Fachtagung des GI Fachbereichs Softwaretechnik*, volume 79 of *Lecture Notes in Informatics*, pages 111–122. Köllen Druck+Verlag GmbH, 2006.

[EHHS00] Gregor Engels, Jan Hendrik Hausmann, Reiko Heckel, and Stefan Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In *UML 2000 - The Unified Modeling Language. Advancing the Standard: Third International Conference*, volume 1939 / 2000, pages 323–337, October 2000.

[EHSW99a] Gregor Engels, Roland Hücking, Stefan Sauer, and Annika Wagner. UML collaboration diagrams and their transformation to java. In *Proceddings of the UML 99 - The Unified Modeling Language: Beyond the Standard, Second International Conference*, volume 1723 / 1999, pages 473–488, October 1999.

[EHSW99b] Gregor Engels, Roland Hücking, Stefan Sauer, and Annika Wagner. UML collaboration diagrams and their transformation to java - extended version -. Technical Report TR-RI-99-208, University of Paderborn, Department of Computer Science, 1999.

[ELS05a] Gregor Engels, Marc Lohmann, and Stefan Sauer. Design by Contract zur semantischen Beschreibung von Web Services. In *GI Jahrestagung (2)*, pages 612–616, 2005.

[ELS05b] Gregor Engels, Marc Lohmann, and Stefan Sauer. Modellbasierte Entwicklung von Web Services mit Design by Contract. In *GI Jahrestagung (2)*, pages 491–495, 2005.

[ELSH06] Gregor Engels, Marc Lohmann, Stefan Sauer, and Reiko Heckel. Model-driven monitoring: An application of graph transformation for design by contract. In *International Conference on Graph Transformation ICGT 2006*, September 2006. accepted for publication.

[ELW03] Gregor Engels, Marc Lohmann, and Annika Wagner. *Web Engineering: Systematische Entwicklung von Web-Anwendungen*, chapter Entwicklungsprozess von Web-Anwendungen, pages 239–263. In Kappel et al. [KPRR03], October 2003.

[ELW06] Gregor Engels, Marc Lohmann, and Annika Wagner. *Web Engineering*, chapter The Web Application Development Process. Wiley, May 2006.

[EMK99] Hartmut Ehrig, Ugo Montanari, and Jörg Kreowski, editors. *Handbook of Graph Grammars and Computing by Graph Transformations: Concurrency, Parallelism, and Distribution Volume 3*. World Scientific Publishing Company, November 1999.

[EPS73] H. Ehrig, M. Pfender, and H.J. Schneider. Graph grammars: an algebraic approach. In *14th Annual IEEE Symposium on Switching and Automata Theory*, pages 167–180. IEEE, 1973.

[EPT04] Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer. Fundamental theory for typed attributed graph transformation. In Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg, editors, *Proc. 2nd Int. Conference on Graph Transformation (ICGT'04)*, volume Lecture Notes in Computer Science 3256, pages 161–177, Rome, Italy, 2004. Springer.

[ETH+03] H. Ehrig, G. Taentzer, R. Heckel, L. Baresi, G. Engels, and M. Pezze. *Graph Transformation and Software Engineering*, chapter Chapter 3: Formal Aspects of Graph Transformation. 2003. Informal Draft, To appear.

[FB02] D. Fensel and C. Bussler. The web service modeling framework WSMF. Technical report, Vrije Universiteit Amsterdam, 2002.

[FFJ+02] Alexander Felfernig, Gerhard Friedrich, Dietmar Jannach, Markus Stumptner, and Markus Zanker. Transforming UML domain descriptions into configuration knowledge bases for the semantic web. In B. Omelayenko and M. Klein, editors, *Proceedings of the Workshop on Knowledge Transformation for the Semantic Web KTSW 2002*, pages 11–18, July 2002.

[FHvH+00] Dieter Fensel, Ian Horrocks, Frank van Harmelen, Stefan Decker, Michael Erdmann, and Michel C. A. Klein. OIL in a nutshell. In *EKAW '00: Proceedings of the 12th European Workshop on Knowledge Acquisition, Modeling and Management*, pages 1–16, London, UK, 2000. Springer-Verlag.

[Fie00] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[Fin99] Frank Finger. Java-implementierung der OCL-basisbibliothek (in german). Technical report, 1999.

[Fin00] Frank Finger. Design and implementation of a modular ocl compiler (master thesis). Master's thesis, Technische Universität Dresden, 2000.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proc. Symposium in Applied Mathematics 19*, pages 19–31. American Mathematical Society, 1967.

[Flu03] Greg Flurry. Web services development and deployment with websphere v5 tools and technologies - part 2: Publishing and finding a web service in a uddi registry, 2003.

[FNT98] Thorsten Fischer, Jörg Niere, and Lars Torunski. Design and implementation of an integrated development environment for UML, Java, and Story Driven Modeling (in German). Master's thesis, University of Paderborn, 1998.

[FNTZ98] Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. Story diagrams: A new graph rewrite language based on the Unified Modeling Language. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Gregor Rozenberg, editors, *Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations (TAGT)*, volume 1764 of *Lecture Notes In Computer Science*, pages 296–309. Springer Verlag, 1998.

[Fow96] Martin Fowler. *Analysis Patterns - Reusable Object Model*. Addison Wesley, October 1996.

[Fre05] Thomas Freitag. Vergleich von Verfahren zur Auffindung von Diensten und Ressourcen. Studienarbeit, University of Paderborn, September 2005.

[FSS03] K. Falkovych, M. Sabou, and H. Stuckenschmidt. UML for the semantic web: Transformation-based approaches. *Knowledge Transformation for the Semantic Web*, 2003.

[FW04] David C. Fallside and Priscilla Walmsley. XML schema part 0: Primer second edition - W3C recommendation 28 october 2004, October 2004.

[GdCL03] Gonzalo Genova, Carlos Ruiz del Castillo, and Juan Llorens. Mapping UML associations into java code. *Journal of Object Technology*, 2(5):135–162, September-October 2003.

[GDD04] Dragan Gasevic, Dragan Djuric, and Vladan Devedzic. Tutorial - MDA standards for ontology development - 7th international conference on the unified modeling language uml 2004, October 2004. `http://ctp.di.fct.unl.pt/UML2004/tutorial02.htm`.

[GDD05] Dragan Gasevic, Dragan Djuric, and Vladan Devedzic. MDA standards for ontology development - half-day (3 hours) tutorial - 4th international semantic web conference (iswc 2005), November 2005. `http://www.sfu.ca/~dgasevic/Tutorials/ISWC2005/`.

[Ges01] Gesamtverband der Deutschen Versicherungswirtschaft e.V. VersicherungsAnwendungsArchitektur (VAA), 2001.

[GHG+93] J.V. Guttag, J.J. Horning, S.J. Garland, K.D. Jones, A. Modet, and Jeannette M. Wing. *Larch: Languages and Tools for Formal Specification*. 1993.

[GHK02] J. Gil, J. Howse, and S. Kent. Towards a formalization of constraint diagrams. In *IEEE CS International Symposium on Human-Centric Computing Languages and Environments (HCC 2001)*. IEEE Computer Society, 2002.

[GHM+03a] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. SOAP version 1.2 part 1: Messaging framework - W3C recommendation 24 june 2003. Technical report, World Wide Web Consortium, June 2003. http://www.w3.org/TR/2003/REC-soap12-part1-20030624/.

[GHM+03b] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. SOAP version 1.2 part 2: Adjuncts - W3C recommendation 24 june 2003. Technical report, World Wide Web Consortium, June 2003. http://www.w3.org/TR/2003/REC-soap12-part2-20030624/.

[GTW78] Joseph A. Gougen, Jim W. Thatcher, and E. G. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In Raymond T. Yeh, editor, *Current Trends in Programming Methodology*, pages 80–149. Prentice Hall, Englewood Cliffs, NJ, 1978.

[Gut77] John V. Guttag. Abstract data types and the development of data structures. *Communications of the ACM*, 20(6):396–404, June 1977.

[GW02] Nicola Guarino and Christopher Welty. Evaluating ontological decisions with ontoclean. *Communications of the ACM*, 45(2):61–65, February 2002.

[Hag03] Claus Hagen. *Enterprise Application Integration - Flexibilisierung komplexer Unternehmensarchitekturen*, volume Band I of *Enterprise Architecture*, chapter Integrationsarchitektur der Credit Suisse, pages 63–81. GITO Verlag Berlin, December 2003.

[Ham02] Ali Hamie. Towards verifying java realizations of OCL-constrained design models using JML. In *Proceedings of 6th IASTED International Conference on Software Engineering and Applications (SEA'2002)*, 2002.

[Ham04] Ali Hamie. Translating the object constraint language into the java modelling language. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 1531–1535. ACM Press, 2004.

[Hau03] Jan Hendrik Hausmann. Metamodelling relations - relating metamodels. In *Proceedings of the Metamodeling for MDA Workshop 2003*, Nov. 2003.

[HBR00] William Harrison, Charles Barton, and Mukund Raghavachari. Mapping uml designs to java. In *The 15th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications-OOPSLA 2000*, volume 35 of *ACM SIGPLAN Notices*, pages 178–187. ACM Press, 2000.

[HC05] Reiko Heckel and Alexey Cherchago. Application of graph transformation for automating web service discovery. In Jean Bezivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, volume 04101 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany, 2005. `http://drops.dagstuhl.de/portals/5/`.

[HCEL96] Reiko Heckel, Andrea Corradini, Hartmut Ehrig, and Michael Löwe. Horizontal and vertical structuring of typed graph transformation. *Math. Struc. in Comp. Science*, 1996.

[HCL04] Reiko Heckel, Alexey Cherchago, and Marc Lohmann. A formal approach to service specification and matching based on graph transformation. In M. Bravetti and G. Zavattaro, editors, *Proceedings of the First International Workshop on Web Services and Formal Methods (WSFM 2004)*, volume 105 of *Electronic Notes in Theoretical Computer Science*, pages 37–49. Elsevier B.V., December 2004.

[HD96] Eric Hammer and Norman Danner. *Logical Reasoning with Diagrams*, chapter Towards a model theory of Venn diagrams, pages 109–127. Oxford University Press, Inc., New York, NY, USA, 1996.

[HDF00] Heinrich Hussmann, Birgit Demuth, and Frank Finger. Modular architecture for a toolset supporting OCL. In Andy Evans, Stuart Kent, and Bran Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard: Third International Conference*, volume 1939 of *LNCS*, pages 278–293, York, UK, 2000. Springer-Verlag Heidelberg.

[HDF02] Heinrich Hussmann, Birgit Demuth, and Frank Finger. Modular architecture for a toolset supporting OCL. *Science of Computer Programming*, 44:51–69, 2002.

[Hec98] Reiko Heckel. *Open Graph Transformation Systems*. PhD thesis, Technical Univesity of Berling, 1998.

[Her04] Volker Herwig. UDDI in der praxis. *Java Spektrum*, 5(04):34–36, Oktober/November 2004.

[Hew] Hewlett-Packard Development Company. Jena - a semantic web framework for Java. `http://jena.sourceforge.net/`.

[HEWC01] Reiko Heckel, Hartmut Ehrig, U. Wolter, and Andrea Corradini. Double-pullback transitions and coalgebraic loose semantics for graph transformation systems. *APCS (Applied Categorical Structures)*, 9(1):83–110, 2001.

[HFB+00] Ian Horrocks, Dieter Fensel, Jeen Broekstra, Stefan Decker, Michael Erdmann, Carole Goble, Frank van Harmelen, Michel Klein, Steffen Staab, Rudi Studer, and Enrico Motta. OIL: The Ontology Inference Layer. Technical Report IR-479, Vrije Universiteit Amsterdam, Faculty of Sciences, September 2000.

[HHK04] Felix Heine, Matthias Hovestadt, and Odej Kao. Towards ontology-driven P2P grid resource discovery. In *5th IEEE/ACM International Workshop on Grid Computing*, 2004.

[HHL03] Jan Hendrik Hausmann, Reiko Heckel, and Marc Lohmann. Towards automatic selection of web services using graph transformation rules. In Robert Tolksdorf and Rainer Eckstein, editors, *Tagungsband Berliner XML Tage 2003*, pages 286–291. XML-Clearinghouse, October 2003.

[HHL04] Jan Hendrik Hausmann, Reiko Heckel, and Marc Lohmann. Model-based discovery of web services. In *IEEE International Conference on Web Services (ICWS'04)*, pages 324–331. IEEE Computer Society, 2004.

[HHL05] Jan Hendrik Hausmann, Reiko Heckel, and Marc Lohmann. Model-based development of web services descriptions enabling a precise matching concept. *International Journal of Web Services Research*, 2(2):67–84, April-June 2005.

[HHS01] Jan Hendrik Hausmann, Reiko Heckel, and Stefan Sauer. Towards dynamic meta modeling of UML extensions: An extensible semantics for UML sequence diagrams. In *HCC*, pages 80–87, 2001.

[HHS03] Jan Hendrik Hausmann, Reiko Heckel, and Stefan Sauer. Dynamic meta modeling with time: Specifying the semantics of multimedia sequence diagrams. In P. Bottoni and M. Minas, editors, *Proc. ICGT 2002 Workshop on Graph Transformation and Visual Modelling Techniques(GT-VMT 2002)*, volume 72 of *ENTCS*, 2003.

[HHS04] Jan Hendrik Hausmann, Reiko Heckel, and Stefan Sauer. Dynamic meta modeling with time: Specifying the semantics of multimedia sequence diagrams. *Software and System Modeling*, 3(3):181–193, 2004.

[HHT96] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3,4):287–313, 1996.

[HK03] Jan Hendrik Hausmann and Stuart Kent. Visualizing model mappings in UML. In *Proceedings of the 2003 ACM symposium*

on Software visualization, pages 169–178, San Diego, California, 2003. ACM Press.

[HKS01] Jan Hendrik Hausmann, Jochen Malte Küster, and Stefan Sauer. Identifying semantic dimensions of (UML) sequence diagrams. In Andy Evans, Robert France, Ana Moreira, and Bernhard Rumpe, editors, *Practical UML-Based Rigorous Development Methods - Countering or Integrating the eXtremists. Workshop of the pUML-Group held together with the UML 2001 October 1st, 2001 in Toronto, Canada*, volume P-7 of *LNI*, pages 142–157. German Informatics Society, 2001.

[HKT02a] Reiko Heckel, Jochen Malte Küster, and Gabriele Taentzer. Confluence of typed attributed graph transformation systems. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *ICGT '02: Proceedings of the First International Conference on Graph Transformation*, volume 2505 of *LNCS*, pages 161–176. Springer-Verlag, 2002.

[HKT02b] Reiko Heckel, Jochen Malte Küster, and Gabriele Taentzer. Towards automatic translation of UML models into semantic domains. In H.-J. Kreowski and P. Knirsch, editors, *Proceedings of the Appligraph Workshop on Applied Graph Transformation*, March 2002.

[HL03a] Reiko Heckel and Marc Lohmann. Model-based development of web applications using graphical reaction rules. In Mauro Pezzè, editor, *Fundamental Approaches to Software Engineering: 6th International Conference, FASE 2003*, volume 2621 of *Lecture Notes in Computer Science*, pages 170–183. Springer-Verlag Berlin Heidelberg, 2003.

[HL03b] Reiko Heckel and Marc Lohmann. Towards model-driven testing. In *International Workshop on Test and Analysis of Component-based Systems (TACoS'03)*, volume 82 of *Electronic Notes in Theoretical Computer Science*. Elsevier B.V., September 2003.

[HL04] Reiko Heckel and Marc Lohmann. Towards contract-based testing of web services. In Mauro Pezzè, editor, *Proceedings of the International Workshop on Test and Analysis of Component Based Systems (TACoS 2004)*, volume 116 of *Electronic Notes in Theoretical Computer Science*, pages 145–156. Elsevier B.V., January 2004.

[HL06] Reiko Heckel and Marc Lohmann. Model-driven development of reactive information systems: from graph transformation rules to JML contracts. *International Journal on Software Tools for Technology Transfer (STTT)*, 2006.

[HM00] James Hendler and Deborah L. McGuinness. The darpa agent markup language. *IEEE Intelligent Systems*, 15(6):67–73, November/December 2000.

[HM04] Patrick Hayes and Brian McBride. RDF semantics - W3C recommendation 10 february 2004, February 2004.

[Hoa69] C. Anthony R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[HP01] A. Habel and D. Plump. Computational completeness of programming languages based on graph transformation. In F. Honsell and M. Miculan, editors, *Foundations of Software Science and Computation Structures (FOSSACS)*, LNCS, pages 230–245. Springer-Verlag, 2001.

[HWG03] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *The C# Programming Language*. Microsoft .NET Development Series. Addison Wesley Professional, 2003.

[IBM] IBM. Rational Rose XDE Developer. `http://www-306.ibm.com/software/awdtools/developer/rosexde/`.

[Int] Interactive Objects. Arcstyler. `http://www.interactive-objects.com/`.

[Iow] Iowa State University. The java modeling language (JML) - JML home page. `http://www.cs.iastate.edu/~leavens/JML/index.shtml`.

[JBR99] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley Professional, 1999.

[Jon90] Cliff B. Jones. *Systematic software development using VDM*. Prentice-Hall International Series In Computer Science. Prentice-Hall, 2nd edition, 1990.

[JZC04] Hemant Jain, Huimin Zhao, and Nageswara Rao Chinta. A spanning tree based approach to identifying web services. *International Journal of Web Services Research*, 1(1):1–20, 2004.

[KE98] Silvia Kolmschlag and Gregor Engels. Unterstützung der flexibilität eines electronic commerce systems durch evolutionstechniken. In Stefan Conrad and Wilhelm Hasselbring, editors, *Workshop „Integration heterogener Softwaresysteme (IHS'98)" im Rahmen der GI-Jahrestagung Informatik '98*, pages 13–24, Magdeburg, 1998.

[Ken97a] Stuart Kent. Constraint diagrams: Visualizing assertions in oo modelling. Technical Report Technical Report ITCM97/C2, University of Brighton, 1997.

[Ken97b] Stuart Kent. Constraint diagrams: Visualizing invariants in object-oriented models. In *OOPSLA97*, 1997.

[KG98] Stuart Kent and Joseph Gil. Visualising action contracts in object-oriented modelling. In *IEE Proceedings: Software*, volume number 2-3, 1998.

[KH99] Stuart Kent and John Howse. Mixing visual and textual constraint languages. In Robert France and Bernhard Rumpe, editors, *Proceedings of UML'99*, volume 1723 of *Lecture Notes in Computer Science*, pages 384–398. Springer, 1999.

[KH02] Jernej Kovse and Theo Härder. Generic XMI-based uml model transformations. In *OOIS '02: Proceedings of the 8th International Conference on Object-Oriented. Information Systems*, pages 192–198, London, UK, 2002. Springer-Verlag.

[KJC04] Graham Klyne and Jeremy J. Carroll. Resource description framework (rdf): Concepts and abstract syntax - W3C recommendation 10 february 2004, February 2004.

[Knu68] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[KPRR03] Gerti Kappel, Birgit Pröll, Siegfried Reich, and Werner Retschitzegger, editors. *Web Engineering: Systematische Entwicklung von Web-Anwendungen.* dpunkt.verlag, October 2003.

[Kru99] Philippe Kruchten. *Der Rational Unified Process - Eine Einführung.* Addison-Wesley, 2. aufl. edition, 1999.

[KTW02a] Christiane Kiesner, Gabriele Taentzer, and Jessica Winkelmann. Visual OCL: A visual notation of the object constraint language. Technical report, 2002.

[KTW02b] Christiane Kiesner, Gabriele Taentzer, and Jessica Winkelmann. Visual OCL: Eine visuelle notation der object constraint language. Technical report, 2002.

[Kus00] S. Kuske. *Transformation Units - A Structuring Principle for Graph Transformation Systems.* PhD thesis, University of Bremen, 2000.

[Küs04] Jochen Malte Küster. *Consistency Management of Object-Oriented Behavioral Models.* PhD thesis, Faculty of Computer Science, Electrical Engineering, and Mathematics of the University of Paderborn, 2004.

[LBR05] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06-rev27, Department of Computer Science, Iowa State University, February 2005.

[LC04] Gary T. Leavens and Yoonsik Cheon. Design by contract with JML. Technical report, 05.04.2004 2004.

[LES06] Marc Lohmann, Gregor Engels, and Stefan Sauer. Model-driven monitoring: Generating assertions from visual contracts. In *accepted for publication at the 21st IEEE/ACM International Conference on Automated Software Engineering - ASE 2006 Demonstration Session.*, September 2006.

[LETE04] Juan de Lara, Claudia Ermel, Gabriele Taentzer, and Karsten Ehrig. Parallel graph transformation for model simulation applied to timed transition petri nets. In *In Proc. Graph Transformation and Visual Modelling Techniques (GTVMT) 2004*, 2004.

[LG03] Juan de Lara and Esther Guerra. Towards the uniform manipulation of visual and textual languages in AToM3. In *Proceedings of Jornadas de Programación y Lenguajes PROLE 2003*, pages 45–58, 2003.

[LKW93] M. Löwe, M. Korff, and Annika Wagner. *An algebraic framework for the transformation of attributed graphs*, chapter Term Graph Rewriting: Theory and Practice, pages 185–199. John Wiley & Sons Ltd, 1993.

[LRE+06] Marc Lohmann, Jan-Peter Richter, Gregor Engels, Baris Güldali, Oliver Juwig, and Stefan Sauer. Abschlussbericht: Semantische Beschreibung von Enterprise Services - Eine industrielle Fallstudie. Technical Report 1, Software Quality Lab , Unversity of Paderborn, May 2006.

[LSE05] Marc Lohmann, Stefan Sauer, and Gregor Engels. Executable visual contracts. In Martin Erwig and Andy Schürr, editors, *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, pages 63–70, 2005.

[LV02] Juan de Lara and Hans Vangheluwe. AToM3: A tool for multi-formalism and meta-modelling. In *FASE '02: Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering*, pages 174–188, London, UK, 2002. Springer-Verlag.

[Mar03] David Martin. DAML-S (version 0.9) walk-through, May 2003.

[Mar04] Benoît Marchal. Working XML: UML, XMI, and code generation, 2004.

[MBH+04] David Martin, Mark Burstein, Jerry Hobbs, Ora Lassila, Drew McDermott, Sheila A. McIlraith, Srini Narayanan, Massimo Paolucci, Bijan Parsia, Terry R. Payne, Evren Sirin, Naveen Srinivasan, and Katia Sycara. OWL-S: Semantic markup for web services - W3C member submission 22 november 2004, 22.11.2004 2004.

[Meg] David Megginson. Sax. `http://www.saxproject.org/`.

[Meh84] Kurt Mehlhorn. *Data Structures and Efficient Algorithms*, volume Graph algorithms and NP-completeness of *EATCS Monographs*. Springer-Verlag New York, Inc., New York, NY, USA, 1984.

[Mey92a] Bertrand Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10):40–51, 1992.

[Mey92b] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, second printing edition, 1992.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Englewood Cliffs, second edition, 1997.

[MF05] Thomas Meservy and Kurt D. Fenstermacher. Transforming software development: An MDA road map. *Computer*, 38(9):52–58, 2005.

[Mit03] Nilo Mitra. SOAP version 1.2 part 0: Primer - W3C recommendation 24 june 2003, Juni 2003. http://www.w3.org/TR/2003/REC-soap12-part0-20030624/.

[MM04] Frank Manola and Eric Miller. RDF primer - W3C recommendation 10 february 2004, February 2004.

[Mon70] Ugo G. Montanari. Separable graphs, planar graphs and web grammars. *Information and Control*, 16(3):243–267, May 1970.

[MSZ01] Sheila A. McIlraith, Tran Cao Son, and Honglei Zeng. Semantic web services. *IEEE Intelligent Systems (Special Issue on the Semantic Web)*, 16(2):46–53, 2001.

[MvH04] Deborah L. McGuinness and Frank van Harmelen. OWL web ontology language - overview - W3C recommendation 10 february 2004, February 2004. `http://www.w3.org/TR/2004/REC-owl-features-20040210/`.

[MW99] K. Mehner and A. Wagner. On the role of method families in aspect oriented programming. In *Proceedings of the ECOOP Workshop on Aspect Oriented Programming)*, volume 1743 of *Lecture Notes In Computer Science*, pages 305–306. Springer-Verlag, 1999.

[NDM70] Kristen Nygaard, Ole-Johan Dahl, and Bjørn Myhrhaug. The simula 67 common base language. Technical Report S-22, Norwegian Computing Center, Oslo, October 1970.

[Ner92] Jean-Marc Nerson. Applying object-oriented analysis and design. *Communication of the ACM*, 35(9):63 – 74, 1992.

[Nob96] James Noble. Some patterns for relationships. In *Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS Pacific 21)*. Prentice-Hall, 1996.

[Nob97] James Noble. Basic relationship patterns. In *Proceedings of the European Conference on Pattern Languages of Program Design (EuroPLOP'97)*, 1997.

[Obj04] Object Management Group. Metamodel and UML profile for java and EJB specification - version 1.0 - part of the UML profile for

enterprise distributed object computing (EDOC) specification, February 2004. formal/04-02-02.

[OMG01a] OMG (Object Management Group). Model Driven Architecture (MDA). Technical Report Document number ormsc/2001-07-01, July 9, 2001 2001.

[OMG01b] OMG (Object Management Group). Unified Modeling Language specification, version 1.4, 2001.

[OMG03a] OMG (Object Management Group). UML 2.0 infrastructure final adopted specifcation, 2003.

[OMG03b] OMG (Object Management Group). UML 2.0 OCL final adopted specification, 2003.

[OMG04] OMG (Object Management Group). UML 2.0 superstructure specification - revised final adopted specification, 2004.

[OMG05] OMG (Object Management Group). MOF 2.0/XMI mapping specification, v2.1, September 2005.

[OMG06] OMG (Object Management Group). Meta Object Facility (MOF) Core specification - version 2.0, January 2006.

[Paa95] J. Paakki. Attribute grammar paradigms - a high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):560–595, June 1995.

[Par02] Parasoft. jcontract - user's guide version 1.5, 2002.

[PKPS02] Massimo Paolucci, Takahiro Kawmura, Terry R. Payne, and Katia Sycara. Semantic matching of web services capabilities. In Ian Horrocks and James A. Hendler, editors, *Proceedings of the First International Semantic Web Conference on The Semantic Web*, volume Lecture Notes In Computer Science; Vol. 2342, pages 333–347, Sardinia, Italy, 2002. Springer-Verlag.

[PO99] Richard F. Paige and Jonathan S. Ostroff. A comparison of the business object notation and the unified modeling language. Technical Report CS-1999-04, York University, Department of Computer Science, 1999.

[Pop04a] Remko Popma. Jet tutorial part 1 (introduction to jet), May 2004. `http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html`.

[Pop04b] Remko Popma. Jet tutorial part 2 (write code that writes code), May 2004. `http://eclipse.org/articles/Article-JET2/jet_tutorial2.html`.

[PP98] Dennis K. Peters and David Lorge Parnas. Using test oracles generated from program documentation. *IEEE Transactions on Software Engineering*, 24(3):161–173, 1998.

[PR69] John L. Pfaltz and Azriel Rosenfeld. Web grammars. In *IJCAI*, pages 609–620, 1969.

[Pra71] T. W. Pratt. Pair grammars, graph languages and string-to-graph translations. *Journal of Computer and System Sciences*, 5:560–595, 1971.

[Pre98] Roger S. Pressman. Can internet-based applications be engineered? *IEEE Software*, vol. 15(no. 5):104–110, 1998.

[PS06] Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF - W3C working draft 20 february 2006, February 2006.

[PSHH04] Peter F. Patel-Schneider, Patrick Hayes, and Ian Horrocks. OWL web ontology language semantics and abstract syntax - W3C recommendation 10 february 2004, February 2004. `http://www.w3.org/TR/2004/REC-owl-semantics-20040210/`.

[Red00] Achut Reddy. Java coding style guide. Technical report, 2000.

[RG99] Mark Richters and Martin Gogolla. A metamodel for OCL. In Robert France and Bernhard Rumpe, editors, *Unified Modeling Language (UML'99)*, pages 156–171, 1999.

[RKL+05] Dumitru Roman, Uwe Keller, Holger Lausen, Jos de Bruijn, Rubén Lara, Michael Stollberg, Axel Polleres, Cristina Feier, Christoph Bussler, and Dieter Fensel. Web service modeling ontology. *Applied Ontology*, 1(1):77–106, 2005.

[Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*. World Scientific Publishing, February 1997.

[RPHB02] Balasubramaniam Ramesh, Jan Pries-Heje, and Richard Baskerville. Internet software engineering: A different class of processes. *Annals of Software Engineering*, 14:169–195, 2002.

[RSN03] Peter Roßbach, Thomas Stahl, and Wolfgang Neuhaus. Model Driven Architecture. *Java Magazin*, 9, 2003.

[Rum87] James Rumbaugh. Relations as semantic constructs in an object-oriented language. In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages and Applications*, pages 466–481, 1987.

[Sch91] Andy Schürr. *Operationales Spezifizieren mit programmierten Graphersetzungssystemen*. Deutscher Universitätsverlag, Wiesbaden, 1991.

[Sch94] Andy Schürr. Specification of graph translators with triple graph grammars. In Tinhofer, editor, *Proceedings WG94 International Workshop on Graph-Theoretic Concepts in Computer Science*, volume LNCS, pages 151–163. Springer-Verlag, 1994.

[Sea04] Andy Seaborne. RDQL - a query language for RDF - W3C member submission 9 january 2004. Technical report, W3C, 2004.

[Sev03] Neyir Sevilmis. Grafische Reaktionsregeln zur Beschreibung der Semantik von Web Services. Master's thesis, Universität Paderborn, 2003.

[SG00] Richard Soley and OMG Staff Strategy Group. Model Driven Architecture, November 2000.

[SH04] Munindar Singh and Michael N. Huhns. *Service-Oriented Computing*. John Wiley and Sons Ltd, November 2004.

[Smi97] B. Smith. Succeed-first or fail-first: A case study in variable and value ordering. In *Proceedings of PACT'97*, pages 321–330, 1997.

[Sof] Software Engineering Group, University of Paderborn. Fujaba tool suite. http://www.fujaba.de/.

[Spi92] J. M. Spivey. *The Z notation: a reference manual*. Prentice-Hall International Series In Computer Science. Prentice-Hall, 1992.

[SVSM03] Kaarthik Sivashanmugam, Kunal Verma, Amit Sheth, and John Miller. Adding semantics to web services standards. In Liang-Jie Zhang, editor, *Proceedings of the International Conference on Web Services, ICWS '03*, pages 395–401, Las Vegas, Nevada, USA, 2003. CSREA Press.

[SWLM04] Michael K. Smith, Chris Welty, and Deborah L. McGuinness. OWL web ontology language - guide - w3c recommendation 10 february 2004, February 2004. `http://www.w3.org/TR/owl-guide/`.

[Tae96] Gabriele Taentzer. *Parallel and Distributed Graph Transformation - Formal Description an Application to Communication-Based Systems*. PhD thesis, Technical University of Berlin, 1996.

[Tae04] Gabriele Taentzer. AGG: A graph transformation environment for modeling and validation of software. In John L. Pfaltz, Manfred Nagl, and Boris Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance: Second International Workshop*, volume Lecture Notes in Computer Science, pages 446–453, Charlottesville/Virgina, USA, 2004. Springer-Verlag Heidelberg.

[TBMM04] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML schema part 1: Structures second edition - W3C recommendation 28 october 2004, October 2004.

[UDD01a] UDDI Consortium. UDDI executive white paper. Technical report, OASIS, November 2001.

[UDD01b] UDDI Consortium. UDDI technical white paper. Technical report, OASIS, September 2001.

[vEB00] Ghica van Emde Boas. Fantastic, unique, uml tool for the java environment (fuut-je). In *OOPSLA '00: Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum)*, pages 109–110, New York, NY, USA, 2000. ACM Press.

[Wag02] Annika Wagner. A pragmatical approach to rule-based transformations within UML using XMI.difference. In *WITUML: Workshop on Integration and Transformation of UML models*, 2002.

[Wag05] Christian D. Wagner. Realisierung eines Editors für visuelle Kontrakte für Eclipse. University of Paderborn, July 2005.

[WD98] Kim Waldén and Enea Data. *Handbook of Object Technology*, chapter Business Object Notation (BON). CRC Press, 1998.

[Wen02] Stefan Wendler. Mapping XMI/UML to DAML+OIL: Automatische umsetzung von UML-klassendiagrammen in eine

DAML+OIL ontologie, 2002. `http://www.jdev.de/html/projects/uml2daml/mapping/uml2daml_mapping.html`.

[Wie00] Ralf Wiebicke. Utility support for checking OCL business rules in java programs. Master thesis, Technische Universität Dresden, 2000.

[Wil03] Edward D. Willink. UMLX: A graphical transformation language for mda. In Arend Rensink, editor, *CTIT Technical Report TR-CTIT-03-27*, pages 13–24, Enschede, The Netherlands, June 2003. University of Twente.

[WN94] Kim Waldén and Jean-Marc Nerson. *Seamless Object-Oriented Software ArchitectureArchitecture: Analysis and Design of Reliable Systems*. Prentice Hall, 1994.

[WS96] Wolfgang Weck and Clemens Szyperski. Do we need inheritance? In *Workshop on Composability Issues in Object-Orientation (at ECOOP'96)*, June 1996.

[WVV+01] H. Wache, T. Vögele, U. Visser, H. Stuckenschmidt, G. Schuster, H. Neumann, and S. Hübner. Ontology-based integration of information - a survey of existing approaches. In *Proceedings of IJCAI-01 Workshop: Ontologies and Information Sharing*, 2001.

[Zün96] Albert Zündorf. Graph pattern matching in progres. In J. Cuny, Hartmut Ehrig, Gregor Engels, and Gregor Rozenberg, editors, *5th. Int. Workshop on Graph-Grammars and their Application to Computer Science*, LNCS 1073, 1996.