Ein Verfahren zur Struktur- und Verhaltensanalyse von UML-Modellen durch Attribute

Christian Saad

TR 2008-17

(basierend auf der Diplomarbeit von Christian Saad)

August 2008

Universität Augsburg
Institut für Informatik

Institut für Informatik
Universität Augsburg
D-86135 Augsburg, Germany
http://www.informatik.uni-augsburg.de/vs

Kurzfassung

Metamodellierung ist ein konzeptioneller Ansatz zur Formalisierung der Objektstruktur einer Anwendungsdomäne, der aktuell in der Softwaretechnik, aber auch in anderen Bereichen der Informatik, große Verbreitung findet. Im Gegensatz zu der, durch das Metamodell festgelegten, abstrakten Syntax lässt sich die Semantik eines Modells durch die momentan zur Verfügung stehenden Methodiken jedoch nur unzureichend beschreiben und überprüfen.
Diese Arbeit entwickelt eine flexible und dynamische Technik zur statischen semantischen Analyse von Modellstrukturen, die auf einer Übertragung des auf kontextfreien Grammatiken definierten Attributierungskonzepts auf UML-Metamodelle beruht. Basierend auf diesem Verfahren wird zudem eine Variante des Datenflussanalyse-Algorithmus vorgestellt, die den Informationsfluss zwischen Modellelementen ermittelt, wodurch Erkenntnisse über das Verhalten auf Instanzebene gewonnen werden können. Der praktische Einsatz der beschriebenen Techniken wird anhand eines Anwendungsbeispiels und einer prototypischen Implementierung demonstriert.

Abstract

Meta modeling is a conceptual approach to the formalization of the object structure in an application domain and is currently gaining influence in software engineering as well as other areas of computer science. However, in contrast to the definition of the abstract syntax by a meta model, the currently available methods for description and validation of model semantics are not sufficient.
This thesis develops a flexible and dynamic technique for the semantic analysis of model structures that is based on an adaption of the concept used to attribute context-free grammars and is able to operate on UML meta models. Additionally, an attribute-based variant of the data flow analysis algorithm is presented, which allows to identify the information flow between model elements in order to study the model behavior on the instance level. The practical use of the described techniques will be demonstrated on the basis of an example use case and a prototypic implementation.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Metamodellierung, zum Beispiel mithilfe der Unified Modeling Language (UML), wird in vielen Bereichen der Informatik - insbesondere in der Softwaretechnik - zur Beschreibung der syntaktischen Struktur von Modellierungssprachen (DSL/DSM) und Softwaresystemen eingesetzt. Ein dabei häufig anzutreffendes Problem ist die zur Formalisierung semantischer Restriktionen unzureichende Ausdrucksmächtigkeit der Metasprachen. Die für diesen Zweck entwickelte Object Constraint Language (OCL) orientiert sich allerdings stark am statischen Aufbau der Modellelemente und ist zudem nicht geeignet, Informationen, die in der dynamischen Modellstruktur implizit vorliegen, abzuleiten. Zudem können durch die OCL keine Berechnungen, die die Betrachtung zyklischer Abhängigkeiten voraussetzen (z.B. eine Datenflussanalyse), implementiert werden.

Die Entwicklung eines Ansatzes zur semantischen Anreicherung von (UML-) Metamodellen hat folglich zum Zweck, die beschränkte Ausdrucksfähigkeit der abstrakten Syntax um Analysefähigkeiten zu erweitern, die die dynamische Aggregation, Auswertung und Überprüfung semantischer Daten erlauben und dabei den speziellen Eigenschaften der Metamodellierung Rechnung tragen.

Die Entscheidung, dieses Ziel durch Implementierung einer Attributierungstechnik zu erreichen, wurde durch die Überlegung motiviert, dass Attribute auf der syntaktischen Struktur von Sprachelementen operieren, und folglich also eine Methodik zur Verfügung stellen, die die im Modell enthaltenen, kontextabhängigen Informationen - im Gegensatz zur OCL - dynamisch zu verteilen bzw. zu aggregieren vermag, und die sich zugleich als invarianter gegenüber Änderungen der Metamodellstruktur erweist.

1.2 Ansatz

Ziel dieser Arbeit ist es, eine Technik zu entwickeln, die die Untersuchung statischer sowie dynamischer Eigenschaften eines UML-Modells erlaubt, um dadurch eine Basis für die Spezifikation semantischer Restriktionen bzw. Analysen auf Metaebene zu schaffen.

Der Ansatz, der für die Entwicklung einer Attributierungstechnik auf Metamodellen gewählt wurde, stützt sich auf das Konzept attributierter Grammatiken, die im Compilerbau zur semantischen Analyse von Quelltexten eingesetzt werden. Ein weiterer Schwerpunkt liegt auf der Datenflussanalyse - ebenfalls eine Technik aus dem Bereich Compilerbau - die es erlaubt zyklische Informationsflüsse in Programmen und - entsprechend angepasst - auch in Metamodellen zu untersuchen.

Ein generisches Konzept zu beschreiben und zu implementieren, das die Ausdrucksmächtigkeit der OCL erweitert, und dabei eine effektive und optimierte Struktur- und Verhaltensanalyse von Modellen ermöglicht, erfordert die Entwicklung von formellen Strukturen und Algorithmen auf vielen Ebenen, weshalb im zeitlich und räumlich begrenzten Rahmen dieser Arbeit nicht auf alle Gesichtspunkte im jeweils notwendigen Detailgrad eingegangen werden kann. Vielmehr soll stattdessen die grundlegende Vorgehensweise, sowie die sich stellenden Herausforderungen bei der Erarbeitung eines Attributierungskonzepts für Metamodelle gegeben werden, um damit die Basis für eine vertiefte Ausarbeitung des Themas zu schaffen. Hierzu werden an zahlreichen Stellen Konventionen exemplarisch eingeführt, mögliche Alternativen und potentielle Erweiterungen aufgezeigt, sowie die sich daraus ergebenden Konsequenzen diskutiert.

Das Grundlagenkapitel 2 beschreibt neben den wesentlichen Aspekten der Metamodellierung auch den Aufbau und die im Kontext dieser Arbeit relevanten Eigenschaften von Attributgrammatiken und der Datenflußanalyse.

Um den Einsatz in der Praxis zu demonstrieren erfolgt die Beschreibung der Attributierungsfragmente auch im Hinblick auf eine Softwareimplementierung, die das Analyseverfahren für das in Kapitel 3 vorgestellte Anwendungsbeispiel durchführt. Ein weiteres einfaches Anwendungsszenario dient dagegen in den folgenden Kapiteln als Fallbeispiel, anhand dessen einzelne Aspekte der Attributierungstechnik erläutert werden.

Der Hauptteil dieser Arbeit wird durch die Kapitel 4 und 5 gebildet. Ersteres entwickelt Syntax und Semantik der Metaattributierungssprache und beleuchtet dabei auch die Gemeinsamkeiten und Unterschiede mit der OCL bzw. Attributgrammatiken. Im zweiten Hauptkapitel wird dann beschrieben, wie eine gegebene Attributierung auf einem Modell ausgewertet werden kann, wobei auch der Sonderfall zyklischer Definitionen berücksichtigt wird.

Aufbauend auf diesen Konzepten wird die Attributierungstechnik in Kapitel 6 zunächst theoretisch auf das bereits erwähnte Anwendungsbeispiel angewandt, um sie dann in ein lauffähiges Auswertungsmodul umzusetzen welches unter anderem die Analyse des Fallbeispiels automatisiert.

Abschließend werden die entwickelten Methoden und Definitionen in Kapitel 7 zusammengefasst und mögliche Erweiterungen des Attributierungskonzepts diskutiert.

2 Grundlagen

In diesem Kapitel soll ein Überblick über die Grundlagen gegeben werden, die in dieser Arbeit für die Entwicklung eines Ansatzes zur Beschreibung und Auswertung von attributierten Metamodellen herangezogen wurden. Bei den in den nachfolgenden Abschnitten vorgestellten Konzepten handelt es sich um Techniken, die insbesondere im Compilerbau, einem Teilgebiet der praktischen Informatik, eingesetzt werden. Um diese Techniken in ihrem jeweiligen Anwendungskontext betrachten zu können, beginnt dieses Kapitel mit einer kurzen Übersicht über den strukturellen Aufbau und die Funktionsweise eines Übersetzers. Im Anschluss werden die für diese Arbeit relevanten Themen im Detail vorgestellt.

Compilerbau

Der *Compilerbau* (Übersetzerbau) ist ein Bereich der Informatik, dessen Grundlagen auf die Automatentheorie und die Theorie formaler Sprachen zurückgehen. Seine zentralen Aufgaben sind die theoretischen und praktischen Herausforderungen, die sich im Zuge der Entwicklung von computergestützten Übersetzungsmechanismen für formale Sprachen stellen. Die hierfür entwickelten Verfahren sind vielseitig und werden in verschiedensten Bereichen der Software-Entwicklung eingesetzt, wenn eine maschinelle Analyse von Texteingaben erforderlich ist. Angewendet werden sie unter anderem bei der Programmierung von Robotern und Werkzeugmaschinen, Job Control Languages oder Generatorsprachen. (vgl. [Job92, S. 1])

Bei einem *Compiler* (Übersetzer) handelt es sich um eine Anwendung, die *"ein in einer bestimmten Sprache - der Quell-Sprache - geschriebenes Programm liest und es in ein äquivalentes Programm einer anderen Sprache - der Ziel-Sprache - übersetzt"* [AEU88a, S. 1]. *Äquivalentes Programm* meint hier, dass die Übersetzung semantikerhaltend sein muss (vgl. [BH98, S. 4]). Da Compiler komplexe Softwaresysteme darstellen, wird der Übersetzungsvorgang meist in einfachere, über sinnvoll gewählte Schnittstellen verbundene, Teilsysteme zerlegt. Dabei haben sich bewährte Strukturierungsmechanismen herausgebildet, die heute bei der Übersetzung beinahe aller höheren Programmiersprachen angewendet werden, wodurch die Konstruktion und Implementierung von Übersetzern stark vereinfacht wurde.

Übersetzungsphasen

Der Übersetzungsprozess eines Compilers wird in der Fachliteratur in zwei Hauptphasen unterteilt (vgl. [AEU88a, S. 2ff], [BH98, S. 5ff], [WM92, S. 173ff]). Man spricht je nach Kontext von der Analyse- und Synthesephase oder vom Compiler Front-End bzw. Back-End[1]. Die Analysephase untersucht die syntaktische Struktur und die semantischen Eigenschaften des Quellprogramms, um daraus eine Zwischendarstellung zu erzeugen, die der Synthesephase wiederum als Grundlage für Optimierungen und die Codegenerierung dient. Daneben wird meist eine Symboltabelle verwaltet, die Informationen zu den erkannten Sprachelementen aggregiert und den Übersetzermodulen zur Verfügung stellt.

Die *Analysephase* lässt sich weiter in einzelne Teilphasen[2] aufgliedern:

Lexikalische Analyse
: Der sogenannte Scanner liest den Zeichenstrom der Eingabe und zerlegt diesen in lexikalische Einheiten[3] (Symbole, Token) der Sprache.

Syntaxanalyse
: Die gefundenen Symbole werden vom Parser anhand der syntaktischen Sprachstruktur[4] geordnet und auf Verletzungen der Syntax geprüft.

Semantische Analyse
: Anhand der Programmstruktur wird die Einhaltung der statischen Semantik überprüft.

Auch die *Synthesephase* besteht aus mehreren Teilprozessen:

Zwischencodeerzeugung
: Aufbauend auf dem Ergebnis der Analysephase wird eine maschinenunabhängige Darstellung generiert.

Codeoptimierung
: Es werden Optimierungen durchgeführt, um die Effizienz des Zwischencodes zu verbessern.

Codeerzeugung
: Der maschinenunabhängige Code wird in die Zielsprache übersetzt.

Anhang A.1 zeigt exemplarisch den Durchlauf einer Programmanweisung durch die einzelnen Übersetzungsphasen. Diese aufgabenorientierte Unterteilung (siehe Abbildung

[1] Beide Aufteilungen entsprechen sich weitestgehend. Die Aufteilung Front-End/Back-End legt allerdings stärkeres Gewicht auf die Unterscheidung zwischen quellsprachenabhängigen (Front-End) und zielsystemabhängigen (Back-End) Teilphasen (vgl. [AEU88a, S. 24f]).

[2] Sowohl für die lexikalische wie auch für die syntaktische Analyse existieren Generatoren wie *Lex* (vgl. [GE99, S. 33ff], [Pen94, S. 33ff]) oder *Yacc* (vgl. http://dinosaur.compilertools.net), die die zugehörigen Compilermodule auf Basis einer vorgegebenen Sprachdefinition erstellen können.

[3] Für die Definition von Sprachausdrücken werden meist reguläre Ausdrücke verwendet.

[4] Da die Ausdrucksmächtigkeit regulärer Ausdrücke für die Definition der Syntax meist nicht ausreicht, werden hier üblicherweise kontextfreie Grammatiken eingesetzt.

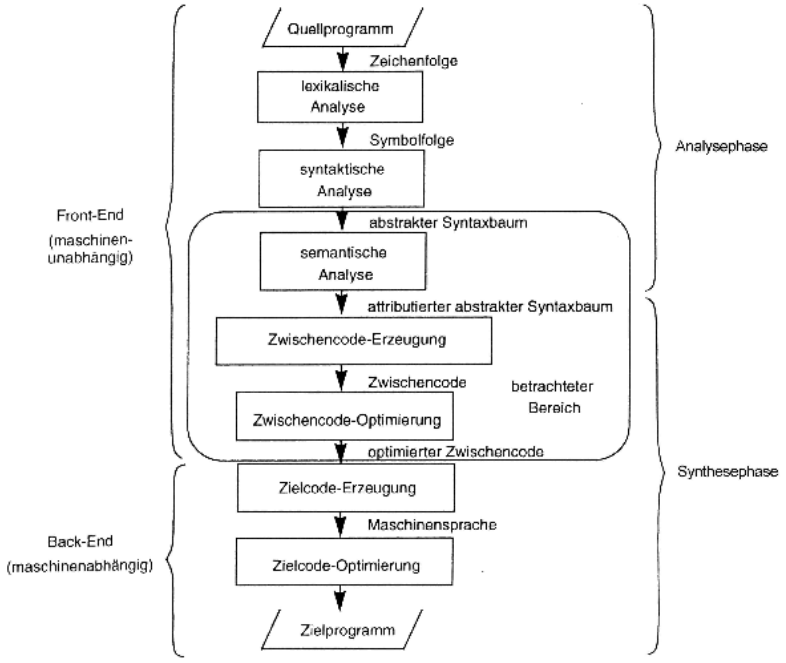


Abbildung 2.1: Phasen eines Compilers (Quelle: [BH98, S. 6], die Unterteilung in Analyse- und Synthesephase wurde gemäß [AEU88a] eingetragen)

2.1) stellt aber lediglich einen konzeptionellen Rahmen dar, da es in der Praxis aus Performancegründen notwendig sein kann verschiedene Phasen zu kombinieren, oder sie aber aufgrund hoher Komplexität weiter aufzuspalten. Im Weiteren sind insbesondere die Schnittstelle zwischen syntaktischer und semantischer Analyse, sowie Verfahren, die in der semantischen Analyse angewendet werden und Optimierungstechniken von Interesse.

Die Übergabe des Quellprogramms an die semantische Analyse erfolgt in Form eines Syntaxbaums, der die syntaktische Struktur der Eingabe repräsentiert. Aufbau und Eigenschaften dieser Bäume werden in 2.1 beschrieben.

Die *semantische Analyse* überprüft Eigenschaften, die über den rein syntaktischen Aufbau der Quellsprache hinausgehen und deshalb nicht durch kontextfreie Grammatiken ausgedrückt werden können. Häufig anzutreffende semantische Restriktionen entstehen beispielsweise durch die Typisierung[5] von Programmiersprachen. Das wichtigste Werkzeug zu deren Analyse sind attributierte Grammatiken, durch die sich statische Eigenschaften, d.h. Eigenschaften die bereits zur Übersetzungszeit erkennbar und für jede Ausführung des Programms konstant sind, überprüfen lassen. Ein weiteres Anwendungsgebiet von Attributgrammatiken sind syntaxgerichtete Definitionen, die die syntaktische Struktur der Eingabe als Basis für die Codegenerierung verwenden. Da At-

[5] Es muss sichergestellt werden, dass Operatoren nur auf kompatible Operanden angewandt werden (vgl. [AEU88a, S. 421ff]).

tributgrammatiken einen wichtigen Ausgangspunkt der Attributierungstechnik auf Metamodellen darstellen, wird auf sie in Abschnitt 2.2 näher eingegangen.

Im Gegensatz zu den statischen Eigenschaften wird das dynamische Verhalten eines Programms erst zur Laufzeit sichtbar und kann deshalb nur näherungsweise durch eine abstrakte Interpretation bestimmt werden. Deren Ergebnis wird meist für Heuristiken zur Optimierung des erzeugten Codes eingesetzt (vgl. [WM92, S. 176,351f]). Eine Technik, die dies umsetzt, ist die Datenfluss- bzw. Fixpunktanalyse, die aus dem Kontrollfluss des Programms optimierungsrelevante Informationen gewinnen kann, z.B. inwieweit Programmanweisungen aufeinander Einfluss nehmen. Diese Konzepte sind für die Auswertung der Metamodellattributierung nützlich und werden in 2.3 vorgestellt.

Da in dieser Arbeit Metamodelle den Platz kontextfreier Grammatiken als Basis für die Entwicklung einer Attributierungstechnik einnehmen, gibt Abschnitt 2.4 einen kurzen Einblick in die Grundlagen und Methoden der Metamodellierung anhand der Unified Modeling Language. Desweiteren wird die Formulierung semantischer Beschränkungen auf Modellebene mit Hilfe der Sprache OCL betrachtet.

2.1 Parse- und Syntaxbäume

Parse- bzw. Syntaxbäume bilden die Schnittstelle zwischen syntaktischer und semantischer Analyse. In diesen grammatikalischen Ableitungsbäumen wird die *„syntaktische Struktur eines Satzes festgehalten“* [Job92, S. 21]. Sie sind für die Beschreibung und Auswertung von Attributierungen von zentraler Bedeutung, weshalb ihr Aufbau in diesem Abschnitt vorgestellt wird.

Kontextfreie Grammatiken

Die Festlegung der Sprachsyntax erfolgt häufig durch Angabe einer kontextfreien Grammatik[6], wodurch die Programmstruktur vom Compiler vergleichsweise einfach und automatisiert ausgewertet werden kann[7]. Durch Generatoren können auf Basis vorgegebener Grammatiken vollautomatisch Parserprogramme erzeugt werden, die Eingaben in der beschriebenen Sprache analysieren[8].

Definition Parsebaum

[6] Eine formale Definition kontextfreier Grammatiken findet sich in [KV97, S. 12f].

[7] In der Praxis werden meist spezielle Klassen von Grammatiken mit eingeschränkter Ausdrucksmächtigkeit verwendet (z.B. LL- oder LR-Grammatiken), die besonders effizient geparst werden können (vgl. [AEU88a, S. 193ff, 263ff]).

[8] In [AEU88a, S. 314ff] wird gezeigt wie dies mit dem Parsergenerator Yacc durchgeführt werden kann.

Nach [AEU88a, S. 36] ist ein Parsebaum B für eine kontextfreie Grammatik $G = (N, T, P, S)$ formal wie folgt definiert:

- *Die Wurzel ist mit dem Startsymbol der Grammatik markiert.*
- *Jedes Blatt ist entweder mit einem Terminal oder mit ϵ markiert.*
- *Jeder innere Knoten ist mit einem Nichtterminal markiert.*
- *Wenn ein innerer Knoten mit dem Nichtterminal A markiert ist und die Nachfolger dieses Knotens von links nach rechts die Marken $X_1, X_2, ..., X_n$ tragen, dann ist $A \rightarrow X_1, X_2, ..., X_n$ eine Produktion. Hierbei stehen $X_1, X_2, ..., X_n$ jeweils für ein Symbol, das entweder Terminal oder Nichtterminal ist. In dem besonderen Fall einer Produktion $A \rightarrow \epsilon$ hat ein mit A markierter Knoten eventuell nur einen einzigen, mit ϵ markierten Nachfolger.*

Anschaulich lässt sich also sagen, *„daß durch die Verklebung von (Vorkommen von) Produktionen ein Ableitungsbaum entsteht“* [KV97, S. 13], der beschreibt, wie ein Eingabewort durch Anwendung der grammatikalischen Regeln erzeugt werden kann. Der Aufbau kann *Top down*, d.h. beginnend beim Startsymbol bis hinunter zu den Blättern oder *Bottom up*, also von den Blättern hinauf zum Startsymbol, erfolgen. Die Blätter des Herleitungsbaums von links nach rechts gelesen ergeben dann genau das Eingabewort.

Mehrdeutige Grammatiken

Beim Entwurf der Grammatiken ist darauf zu achten, dass diese nicht mehrdeutig sind, d.h. es dürfen zu einem gegebenen Wort nicht mehrere Ableitungsbäume existieren. Die Konsequenz wäre, dass die Sprache nicht mehr eindeutig interpretiert werden kann[9]. Lässt sich die Mehrdeutigkeit nicht vermeiden, so müssen für die problematischen Ableitungen spezielle Regeln vereinbart werden, die sicherstellen, dass Wörter der Sprache nach einem konsistenten Muster ausgewertet werden (vgl. [AEU88a, S. 37f, 301f]).

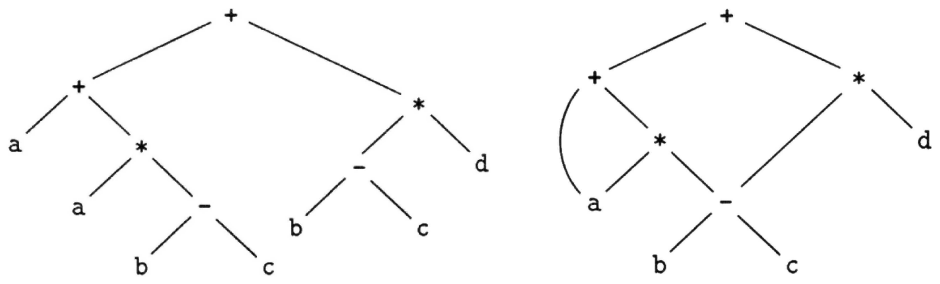
Abstrakte Syntax / Syntaxbaum

Da sich die Grammatik einer Sprache meist eng an deren äußerer Form orientiert, ist es für den weiteren Kompilierungsvorgang oft sinnvoll diese *konkrete Syntax* von uninteressanten Informationen, zum Beispiel vollständigen Ableitungsketten für Terminalsymbole, zu befreien. Das Ergebnis dieser Transformation wird als *abstrakte Syntax* bezeichnet, die nur noch die auftretenden Konstrukte und ihre Schachtelungsbeziehungen identifiziert. (vgl. [WM92, S. 36])

Die Unterscheidung zwischen abstrakter und konkreter Syntax findet sich entsprechend auch in der Darstellung des Ableitungsbaums wieder. Die Gliederung der Symbole anhand der konkreten Syntax wird als *konkreter Syntaxbaum* oder *Parsebaum* bezeichnet,

[9] Anhang A.2 zeigt eine mehrdeutige Grammatik mit verschiedenen Ableitungen.

während der *abstrakte Syntaxbaum* oder *Strukturbaum*[10] die auf die wesentlichen Konstrukte reduzierte Darstellung bezeichnet (vgl. [AEU88a, S. 60]). Dabei stellen *„die Operatoren die inneren Knoten [...] und die Operanden eines Operators die Söhne des Knotens für diesen Operator“* [AEU88a, S. 9] dar.



(a) Abstrakter Syntaxbaum (b) Gerichteter azyklischer Graph

Abbildung 2.2: Der Ausdruck $a + a * (b - c) + (b - c) * d$ dargestellt als abstrakter Syntaxbaum und als gerichteter azyklischer Graph (Quelle: [AEU88a, S. 356])

Bei Parse- bzw. Syntaxbäumen handelt es sich also um semantisch äquivalente Darstellungen derselben Eingabe, deren Verwendung vom jeweiligen Einsatzkontext bestimmt wird. Abbildung 2.2(a) zeigt einen (abstrakten) Syntaxbaum. Anhang A.3 verdeutlicht den Unterschied zwischen konkreter und abstrakter Syntax anhand eines Beispiels.

Gerichtete azyklische Graphen

Ein Syntaxbaum lässt sich, leicht modifiziert, auch als Gerichteter azyklischer Graph (GAG) (siehe Abbildung 2.2(b)) darstellen. Da es sich bei GAGs nicht um Bäume handelt, darf ein GAG-Knoten mehr als einen Vorgänger besitzen. Diese Eigenschaft ist dann von Vorteil, wenn zwei verschiedene Knoten auf einen identischen Nachfolger verweisen, so dass in einem Syntaxbaum in diesem Fall zwei identische Kopien dieses Nachfolgers angelegt werden müssten (mit entsprechenden negativen Auswirkungen auf die Performanz eines Auswertungsalgorithmus). Ein GAG erlaubt diese Redundanz zu vermeiden, indem derselbe Knoten bei Bedarf mehrfach referenziert werden kann.

2.2 Attributgrammatiken

Die Definition einer Programmiersprache erfordert die Festlegung *“wie die Programme der Sprache aussehen (die Syntax der Sprache) und welche Bedeutung diese Programme haben (die*

[10] Strukturbäume können bereits beim Parsen der Eingabe aus der konkreten Syntax erzeugt werden (vgl. [Kas90, S. 75f]).

Semantik der Sprache)" [AEU88a, S. 31]. Die durch eine kontextfreie Grammatik beschriebene Sprachsyntax kann beispielsweise in der erweiterten Backus Naur Form (EBNF) oder durch Syntaxdiagramme formuliert werden (vgl. [WS07, S. 197f]). Durch die formale Beschreibung wird eine Automatisierung der lexikalischen- sowie der syntaktischen Analyse ermöglicht.

Semantik einer Sprache

Die Semantik einer Sprache lässt sich - im Gegensatz zur Syntax - nur vergleichsweise schwer formal fassen und auf Korrektheit prüfen, da meist der Kontext, in dem die Sprachfragmente auftreten, von Bedeutung ist. Die festgelegte Semantik beschränkt die Menge der, durch die Grammatik erzeugten Wörter, auf eine Teilmenge der semantisch korrekten Ausdrücke. Es wird zwischen statischer und dynamischer Semantik unterschieden: Statische semantische Eigenschaften können zur Übersetzungszeit analysiert werden, wohingegen dynamische Eigenschaften erst zur Laufzeit sichtbar werden. Die Definition der statischen semantischen Anforderungen erfolgt meist textuell, für die automatisierte Überprüfung auf semantische Korrektheit wird oft auf eine Attributgrammatik (AG)[11] zurückgegriffen.

Syntaxgerichtete Definition

Neben der reinen Überprüfung semantischer Vorgaben können attributierte Grammatiken auch zur Codeerzeugung eingesetzt werden. Bei der *syntaxgerichteten Definition* handelt es sich um einen *„Formalismus mit dem Übersetzungen für programmiersprachliche Konstrukte beschrieben werden können"* [AEU88a, S. 41], wobei der Übersetzungsvorgang durch die syntaktische Struktur der Eingabe geleitet wird. In den Attributen werden Informationen über den Kontext der Programmsymbole gesammelt und daraus dann die entsprechenden Anweisungen erzeugt.[12]

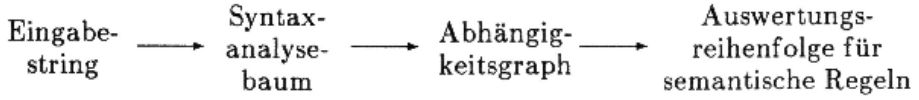


Abbildung 2.3: Ablauf der Analyse und Auswertung von Attributgrammatiken (Quelle: [AEU88a, S. 341])

Die folgenden Abschnitte definieren die für diese Arbeit im Zusammenhang mit attributierten Grammatiken relevanten Begriffe und beschreiben, analog zu Abbildung 2.3, wie

[11] Weitere Ansätze zur formalen Festlegung semantischer Anforderungen sind z.B. denotationale (vgl. [Gor79]) und operationelle (vgl. [WM92, S. 472]) Semantiken.

[12] [AEU88a, S. 42ff] zeigt die Übersetzung von Infix Ausdrücken in Postfix-Notation durch eine syntaxgerichtete Definition.

für Eingaben in Form von Ableitungsbäumen eine Abhängigkeitsanalyse, und, darauf aufbauend, eine Auswertung der Attribute durchgeführt werden kann.

2.2.1 Definition

Das grundlegende Prinzip einer Attributierung von Grammatiken wird durch folgende Aussage zusammengefasst:

> *Da Ableitungsbäume auf kontextfreien Grammatiken basieren, bietet es sich an, jedem Symbol der Grammatik eine Menge von Attributen als 'Informationsbehälter' zuzuordnen und jeder Produktion der Grammatik eine Menge von Regeln, die die Berechnung von Attributwerten aus den Werten von Attributen anderer in der Produktion vorkommender Symbole beschreiben. [...] Die Art der in Attributen aufbewahrten Informationen ist nicht näher spezifiziert.* [GE99, S. 120]

„Diese Erweiterung [kontextfreier Grammatiken] hat das Ziel, eine Möglichkeit bereitzustellen, Informationen von einem Knoten eines Ableitungsbaumes an einen anderen Knoten zu transportieren und sie mit dort vorliegenden Informationen zu verknüpfen." [KV97, S. 1]. Die Definition ist deklarativ: Es werden lediglich funktionale Abhängigkeiten festgelegt, jedoch keine Algorithmen zum Baumdurchlauf (vgl. [Kas90, S. 99]).

Definition Attributgrammatik

Die Basis einer Attributgrammatik ist eine kontextfreie Grammatik $G = (N, T, P, S)$ mit Nonterminalmenge N, Terminalmenge T, Produktionen P und Startsymbol S. Produktionen besitzen die Form $X_0 \rightarrow X_1 ... X_n$ mit $X_0 \in N$ und $X_i \in N \cup T, 1 \leq i \leq n$. Jedes (Non)Terminal der kontextfreien Grammatik wird mit einer Menge von Attributen verknüpft, die als Träger statischer semantischer Informationen fungieren. Ein Attribut ist entweder vom Typ *synthetisiert* (zusammengesetzt) oder vom Typ *ererbt*. Diese Eigenschaft ist konstant für alle Vorkommen des Attributs. Funktionale Abhängigkeiten werden zwischen den Attributvorkommen an den Grammatiksymbolen der Produktionen festgelegt. Diese Abhängigkeiten können als Berechnungsvorschriften interpretiert werden, die angeben wie sich der Wert eines konkreten Attributvorkommens aus den Werten anderer Vorkommen derselben Produktion berechnet. In einem Ableitungsbaum der Grammatik verbinden sich diese lokalen Abhängigkeiten zu einem globalen Informationsfluss. Die funktionalen Abhängigkeiten sind dabei immer so zu definieren, dass die Attributexemplare in jedem syntaktisch und statisch semantischen korrekten Programm ausgewertet werden können. (vgl. [WM92, S. 372]; [KV97, S. 1])

Formale Definitionen von Attributgrammatiken, die in der Literatur zu finden sind, weisen im Detail oft Unterschiede auf, die zentralen Konzepte sind jedoch stets identisch. Die folgende Definition $AG = (G, A, R)$ enthält die übereinstimmenden Elemente aus

[AEU88a, S. 343ff], [WM92, S. 372ff], [WG98, S. 183ff], [KV97, S. 21ff], [GE99, S. 120ff], [Pen94, S. 141ff] und [Kas90, S. 98ff]:

- G: Die zugrunde liegende kontextfreie Grammatik $G = (N, T, P, S)$.
- A: Eine endliche Menge von Attributen, bestehend aus den disjunkten Teilmengen der ererbten (Inh) und synthetisierten (Syn) Attribute, wobei jedes Attribut durch die Relation $A(X)$ einem Grammatiksymbol $X \in N \cup T$ eindeutig[13] zugeordnet ist.
- R: Eine endliche Menge von Attributionsregeln $\bigcup_{p \in P} R(p)$ der Form $X_i.a = f(...)$, die durch $R(p)$ jeweils einer Produktion $p \in P$ der Grammatik zugeordnet sind.

Die Menge $A(X)$ aller Attribute eines Symbols X entspricht der Vereinigung der (disjunkten) Teilmengen der ererbten und synthetisierten Attribute $Inh(X) \cup Syn(X)$, die X zugeordnet sind. Die Mengen aller ererbten und synthetisierten Attribute lassen sich damit definieren als $Inh = \bigcup_{X \in N} Inh(X)$ bzw. $Syn = \bigcup_{X \in N \cup T} Syn(X)$. Vereinfachend kann ein Attribut $a \in A(X)$ auch als $X.a$ notiert werden. Betrachtet man die Vorkommen eines Grammatiksymbols aus $X \in N \cup T$ in den Produktionen P, so existieren für jedes Auftreten von X entsprechende Attributvorkommen aller Attribute $a \in A(X)$.

Erweiterungen

Diese, bei allen Definitionen in der Fachliteratur übereinstimmenden Kernelemente, werden von einigen Autoren um individuelle Erweiterungen ergänzt. [WG98, S. 183ff] definiert eine Menge $B(p)$ von Bedingungen an einer Produktion $p \in P$. Diese berechnen aus den produktionslokal verfügbaren Attributen einen Rückgabewert vom Typ *boolean*, der signalisiert ob der analysierte Ausdruck der statischen Semantik entspricht. Da diese Bedingungen auch als semantische Regeln für ein *boolean* Attribut der linken Produktionsseite interpretiert werden können, ist eine gesonderte Betrachtung in der Praxis allerdings meist nicht nötig. Eine weitere Eigenschaft attributierter Grammatiken, die sich nicht in allen Definitionen findet, ist die Festlegung einer gültigen Wertedomäne D_a für jedes Attribut $a \in A$ (vgl. [WM92, S. 373]). Weiterhin wurden Klassen attributierter Grammatiken mit eingeschränkter Ausdrucksmächtigkeit[14] entworfen, die eine besonders effiziente Analyse unterstützen.

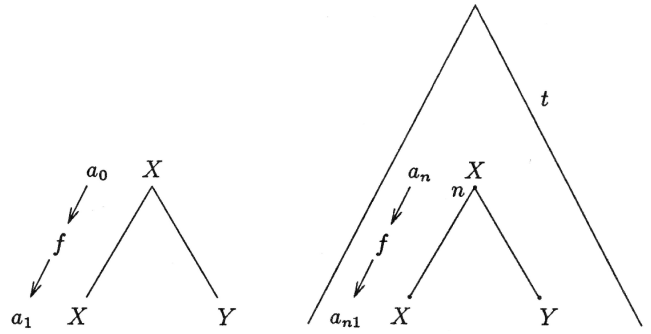
Attributexemplare

Sowohl für Attribute als auch für Attributvorkommen gilt, dass diese bereits durch die Attributierung einer Grammatik vollständig definiert sind. Wird nun für ein Wort der

[13] Um zu verdeutlichen, dass sie Träger derselben Art von Information sind, können verschiedene Attribute durchaus denselben Namen tragen.

[14] Zum Beispiel S- und L-attributierte Grammatiken (vgl. [AEU88a, S. 344ff, 363ff]).

Sprache der zugehörige Syntaxbaum generiert, so enthält jedes angewandte Vorkommen einer Produktion in diesem Baum Instanzen aller Attributvorkommen dieser Produktion. Für diese Instanzen, die als Attributexemplare oder auch als Attributinstanzen bezeichnet werden, wird ein Ergebniswert gemäß den Attributierungsregeln berechnet (siehe Abschnitte 2.2.3 und 2.2.4).



(a) Attributvorkommen zur Definitionszeit

(b) Attributexemplare im generierten Syntaxbaum

Abbildung 2.4: Attributvorkommen und Attributexemplare für eine Produktion $X \rightarrow XY$ und eine semantische Regel $a_1 = f(a_0)$ (Quelle: [WM92, S. 375])

Abbildung 2.4 verdeutlicht den Unterschied zwischen Attributvorkommen und Attributexemplaren anhand einer Produktion $p : X \rightarrow XY$ und eines Attributs $a \in Inh(X)$. In Bild 2.4(a) hat das Symbol X, und damit auch das ihm zugeordnete Attribut a zwei Vorkommen in der Produktion. Im Beispiel dargestellt ist auch deren Abhängigkeitsbeziehung für eine semantische Regel $a_1 = f(a_0)$, d.h. das Attributvorkommen a_1 (bei dem Vorkommen von X auf der rechten Produktionsseite) bezieht seinen Eingabewert von dem Attributvorkommen a_0 (bei dem Vorkommen von X auf der linken Produktionsseite). Bild 2.4(b) zeigt eine Instanz dieser Produktion als Teilbaum n eines Syntaxbaums t. An den Knoten dieser Instanz befinden sich die instantiierten Attributexemplare, die die Basis für die Anwendung semantischer Regeln bilden.

Synthetisierte / Ererbte Attribute

Synthetisierte Attribute zeichnen sich dadurch aus, dass der Informationsfluss zwischen den Attributexemplaren im Syntaxbaum von den Blättern zur Wurzel hin stattfindet. Dies ist sinnvoll, wenn Ergebnisse für Teilausdrücke berechnet werden sollen, die dann zu Gesamtergebnissen zusammengefasst werden, z.B. bei der Berechnung von arithmetischen Ausdrücken. Für synthetisierte Attribute an Terminalzeichen werden üblicher-

weise durch externe Regeln feste Werte[15] vorgegeben. Ererbte Attribute, deren Werte sich aus den Attributwerten ihrer Vorgänger- und Nachbarknoten ableiten, werden dagegen meist eingesetzt um Kontextinformation des jeweiligen Ausdrucks, zum Beispiel eine Liste deklarierter Variablen, weiterzuleiten. (vgl. [AEU88a, S. 344ff])

Normalform

Man unterscheidet bei Attributvorkommen zwischen definierenden und angewandten Vorkommen. Wenn für ein synthetisiertes Attribut a eines Symbols X innerhalb einer Produktion der Form $X_0 \rightarrow X_1...X_n$, der eine Attributionsregel für dieses Attribut zugeordnet ist, gilt $X = X_0$, dann bezeichnet man dieses Vorkommen als definierend. Dasselbe gilt für ein ererbtes Attribut, wenn $X \in X_1...X_n$. Alle anderen Vorkommen werden als angewandt bezeichnet. Definierende Vorkommen sind also synthetisierte Attribute der linken, bzw. ererbte Attribute der rechten Produktionsseite. Existiert für jedes definierende Attributvorkommen einer Produktion genau eine semantische Regel, deren Argumente ausschließlich angewandte Attributvorkommen sind, so ist die Attributgrammatik in Normalform. Dies kann wie folgt interpretiert werden: Definierende Vorkommen erhalten ihre Werte von angewandten Vorkommen derselben Produktion. Deren Werte werden wiederum aus den Vorkommen in Vorgänger- bzw. Nachfolgerknoten im Syntaxbaum abgeleitet. Die Normalform einer Grammatik spielt bei der Analyse der Abhängigkeiten (siehe dazu 2.2.3) eine wichtige Rolle.

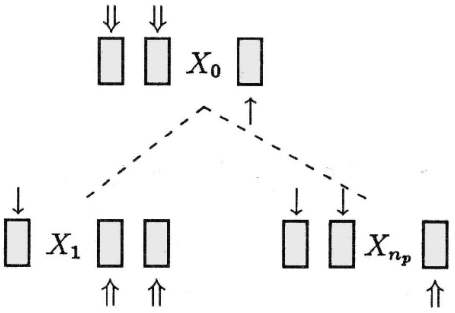


Abbildung 2.5: Produktion in Normalform: Ererbte Attributvorkommen stehen links, synthetisierte rechts der Grammatiksymbole. Einfache Pfeile symbolisieren den Informationsfluss innerhalb der Produktion, doppelte die von außen erhaltenen Informationen (Quelle: [WM92, S. 374])

Abbildung 2.5 zeigt einen Syntaxbaumausschnitt für ein Produktionsexemplar $X_0 = X_1...X_{n_p}$ in Normalform. Angewandte Attributexemplare erhalten ihre Werte von außerhalb, definierende Exemplare von innerhalb des jeweiligen Produktionsvorkommens.

[15] Diese können beispielsweise aus der lexikalischen Analyse oder der Symboltabelle des Compilers stammen.

2.2.2 Syntax

Nachdem in 2.2.1 die grundlegenden Bestandteile von Attributgrammatiken vorgestellt wurden, wird in diesem Abschnitt betrachtet, wie attributierte Grammatiken in der Praxis notiert werden können. Die Festlegung einer Deklarationssprache ermöglicht erst die automatische Analyse und Auswertung und wird dementsprechend bei der Konstruktion einer Attributierungstechnik auf Metamodellen ebenfalls eine wichtige Rolle spielen.

Notation für attributierte Grammatiken

Für die Formulierung von Attributgrammatiken existiert keine einheitliche Syntax. Da aber weitestgehend Einigkeit über deren grundlegende Struktur besteht, sind die Notationen verschiedener Autoren in ihrem Aufbau sehr ähnlich. Die Produktionen der zugrunde liegenden kontextfreien Grammatik werden dabei direkt um die daran angelegten semantischen Regeln erweitert.[16] (vgl. [WM92, S. 375f], [WG98, S. 184], [Kas90, S. 98])

Im Weiteren wird exemplarisch die Notation[17] aus [WM92] übernommen. Diese deckt alle relevanten Attributierungsbestandteile ab und verschafft so einen Überblick über die Anforderungen, die an eine Deklarationssprache für attributierte Grammatiken zu stellen sind. Selbstverständlich gilt dabei auch für diese Sprache, dass neben der syntaktischen Form auch statische semantische Vorgaben zu berücksichtigen sind, um eine korrekte Auswertung zu ermöglichen. So muss beispielsweise sichergestellt werden, dass die Mengen der synthetisierten und ererbten Attribute disjunkt sind und alle Argumente semantischer Regeln in der jeweiligen Produktion ein Vorkommen besitzen. Diese und weitere Anforderungen ergeben sich aber bereits aus der Definition attributierter Grammatiken, weswegen hier nicht weiter darauf eingegangen werden soll.

Die vorgestellte Schreibweise verbindet die Definition einer kontextfreien Grammatik, bestehend aus (Non)Terminalen und Produktionen, mit der Deklaration von Attributen und semantischen Regeln. Attribute werden durch die vorangestellten Schlüsselwörter *syn* bzw. *inh* als synthetisiert bzw. ererbt definiert. Daran anschließend wird die Liste der (Non)Terminale aufgeführt, denen das jeweilige Attribut zugeordnet ist[18], sowie der gültige Wertebereich (*domain*) festgelegt. Der nächste Abschnitt besteht aus einer Liste von Regeldefinitionen, wobei sich jede Regel aus jeweils einer Produktion und einer Liste von semantischen Regeln dieser Produktion zusammensetzt. Ein optionaler letzter Teil bietet die Möglichkeit Funktionen zu definieren, die in den semantischen Regeln auf Attributvorkommen angewendet werden können.

[16] Auch der Parsergenerator Yacc, der Attributierung eingeschränkt unterstützt, verwendet eine vergleichbare Notation (vgl. [GE99, S. 109ff]).

[17] Die formale Definition der Sprache in EBNF-Form findet sich in A.4.

[18] Tatsächlich handelt es sich gegebenenfalls um verschiedene Attribute gleichen Namens, da die Zuordnung von Attribut zu Symbol eindeutig sein muss.

attribute grammar Bin_to_Dec:
nonterminals $\{N, BIN, BIT\}$;
attributes **syn** l **with** BIN **domain int**;
syn v **with** N, BIN, BIT **domain real**;
inh r **with** BIN, BIT **domain int**;
rules

1 : $N \rightarrow BIN.\ BIN$
$N.v = BIN_1.v + BIN_2.v$
$BIN_1.r = 0$
$BIN_2.r = -\ BIN_2.l$

2 : $BIN \rightarrow BIN\ BIT$
$BIN_0.v = BIN_1.v + BIT.v$
$BIN_0.l = BIN_1.l + 1$
$BIN_1.r = BIN_0.r + 1$
$BIT.r = BIN_0.r$

3 : $BIN \rightarrow$
$BIN.v = 0$
$BIN.l = 0$

4 : $BIT \rightarrow 1$
$BIT.v = 2^{BIT.r}$

5 : $BIT \rightarrow 0$
$BIT.v = 0$

Abbildung 2.6: Attributgrammatik *Bin_to_Dec* (Quelle: [WM92, S. 381f])

Beispiel einer Attributgrammatik

Abbildung 2.6 zeigt eine attributierte Grammatik gemäß der vorgestellten Notation. Die Grammatik *Bin_to_Dec* führt eine Umrechnung von Binärzahlen (mit Nachkommastellen) der Form $n.m (n, m \in \mathbb{N})$ in Dezimalzahlen durch. Die Umrechnung basiert auf folgenden Attributen:

v $(value, synthetisiert)$
: Berechnet den Dezimalwert der einzelnen Binärziffern anhand ihres Ranges r (d.h. ihrer Position) und synthetisiert daraus den Gesamtwert.

l $(length, synthetisiert)$
: Berechnet die Länge des Nachkomma-Binärstrings. In negierter Form bestimmt diese den Rang des niederwertigsten Bits.

r $(rank, vererbt)$
: Berechnet den Rang der Binärziffern durch Vererbung. Der Rang eines Bits ist dabei wie folgt festgelegt (jeweils ausgehend vom Dezimalpunkt): Elemente der linken Seite besitzen einen aufsteigenden Rang $0, 1, 2, ..$, Elemente rechts einen absteigenden Rang $-1, -2, -3, ...$

Anhand dieses Beispiels ist zu erkennen, wie synthetisierte und ererbte Attribute in Kombination eingesetzt werden können, um ein Ergebnis zu ermitteln: Zunächst wird die Länge der Binärzahl rechts des Punktes synthetisiert und deren Rang mit ihrer negativen Länge initialisiert (der Rang der linken Binärzahl ist initial 0). Der Rang aller anderen Binärziffern wird bestimmt, indem der Rang des Vorgängers weitervererbt und dabei jeweils inkrementiert wird. Zuletzt kann der Dezimalwert jeder Ziffer berechnet werden, der im Syntaxbaum wiederum durch Addition bis zu dem Endergebnis an der Wurzel synthetisiert wird.

2.2.3 Abhängigkeitsanalyse

Nachdem eine kontextfreie Grammatik attributiert und der Syntaxbaum für ein Wort der Sprache generiert wurde, müssen im nächsten Schritt die an den Knoten dieses Baums anliegenden Attributexemplare gemäß den vorgegebenen semantischen Regeln ausgewertet werden. Eine wichtige Rolle spielt dabei die Betrachtung der Abhängigkeiten zwischen den Attributexemplaren. Diese entstehen aus den Argumenten semantischer Regeln und können durch Abhängigkeitsgraphen visualisiert werden.

Abhängigkeiten zwischen Attributvorkommen

Eine Abhängigkeit ist eine Beziehung zwischen den Attributvorkommen der Produktionen: Wird ein Attributvorkommen *b* als Argument der semantischen Regel eines Attributvorkommens *a* benötigt, so impliziert dies, dass *a* von *b* abhängt. Da *„durch die Attributierung [...] keinerlei Strategie für Besuche der Knoten im Syntaxbaum vorgegeben“* [Pen94, S. 138] ist, muss sichergestellt werden, dass die Berechnung von *a* erst erfolgt wenn der Wert von *b* verfügbar ist (vgl. [WM92, S. 382]).

Die Analyse der Abhängigkeiten legt also die Reihenfolge fest, in der die einzelnen Werte ausgewertet werden müssen, um für weitere Berechnungen zur Verfügung stehen. Die Abhängigkeiten können dabei in lokalem Kontext, d.h. innerhalb einer Produktion, global für den gesamten Syntaxbaum oder sogar für die gesamte Grammatik betrachtet werden.

Für die Menge aller Attributexemplare an den Knoten des Syntaxbaums ergibt sich ein Gleichungssystem. Ist dieses Gleichungssystem rekursiv (*zyklisch*), so kann es mehrere oder auch gar keine Lösung besitzen. Ist es nicht zyklisch, so existiert genau eine Lösung, d.h. jedem Attributexemplar kann ein eindeutiger Wert zugeordnet werden. Ist sichergestellt, dass kein aus einer Grammatik erzeugtes Gleichungssystem eine Rekursion enthalten kann, so wird diese Attributgrammatik als *wohlgeformt* bezeichnet. (vgl. [WM92, S. 375])

Produktionslokale Abhängigkeiten

Produktionslokale Abhängigkeiten bezeichnen die Beziehungen zwischen Attributvorkommen innerhalb einer Produktion. Sie können visualisiert werden, indem sie in die graphische Darstellung der jeweiligen Produktion eingetragen werden. Als Konvention wird festgelegt, dass ererbte Attribute links und synthetisierte rechts der Grammatiksymbole stehen. Für zwei Attributvorkommen *a* und *b*, wobei *a* von *b* abhängt, wird in den Graphen ein Pfeil von *b* nach *a* eingezeichnet. Sind alle Abhängigkeiten eingetragen, so erhält man den *produktionslokalen Abhängigkeitsgraphen*. Befindet sich die Attributgrammatik in Normalform, so ist sichergestellt, dass die lokalen Abhängigkeitsgraphen nicht zyklisch sind (vgl. [WM92, S. 384]).

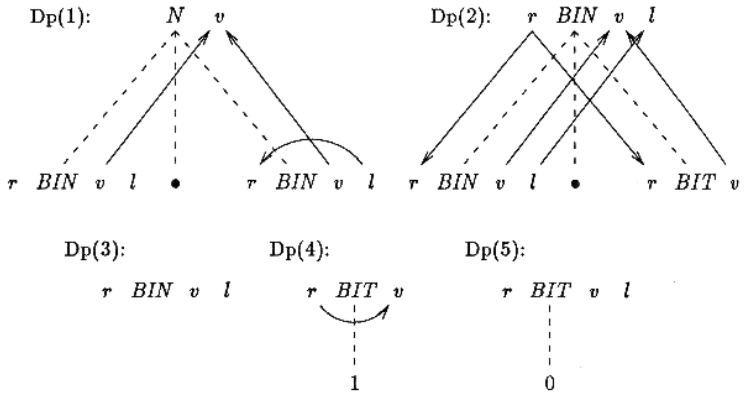


Abbildung 2.7: Produktionslokale Abhängigkeiten der Attributgrammatik *Bin_to_Dec* (nach [WM92, S. 384])

Für die in Abschnitt 2.2.2 vorgestellte Attributgrammatik *Bin_to_Dec* zeigt Abbildung 2.7 die zugehörigen produktionslokalen Abhängigkeitsgraphen D_p aller Produktionen $p \in P$.

```
for jeden Knoten n im Parsebaum do
    for jedes Attribut a des Grammatiksymbols bei n do
        konstruiere im Abhängigkeitsgraphen einen Knoten für a;
for jeden Knoten n im Parsebaum do
    for jede Semantikregel b := f(c1, c2, ..., ck), die mit der bei
        n benutzten Produktion assoziiert ist do
            for i := 1 to k do
                konstruiere eine Kante vom Knoten für ci
                    zum Knoten für b;
```

Abbildung 2.8: Algorithmus zur Abhängigkeitsanalyse in einem attributierten Syntaxbaum (Quelle: [AEU88a, S. 348])

Individueller Abhängigkeitsgraph

Soll das Gleichungssystem für die Berechnung der Attributwerte aufgestellt werden, so kann ausgehend von den lokalen Abhängigkeitsgraphen (die auf Basis der Attributierung vorausberechnet werden können) und dem Syntaxbaum des Eingabewortes (der erst zur Übersetzungszeit zur Verfügung steht) der *individuelle Abhängigkeitsgraph* erstellt werden. Dieser wird konstruiert, indem an jedem Knoten des Syntaxbaums der lokale Abhängigkeitsgraph der dort angewendeten Produktion eingesetzt wird. Alternativ kann auch der Algorithmus aus Abbildung 2.8 angewendet werden, der den individuellen Abhängigkeitsgraphen vollständig dynamisch erzeugt.

Abbildung 2.9 zeigt den individuellen Abhängigkeitsgraph der Attributgrammatik *Bin_to_Dec*, basierend auf den produktionslokalen Abhängigkeitsgraphen aus Abbildung 2.7 und dem Syntaxbaum für das Eingabewort „10.01“. Ausgehend von diesem Graphen kann nun die Auswertungsreihenfolge für die Attributexemplare bestimmt

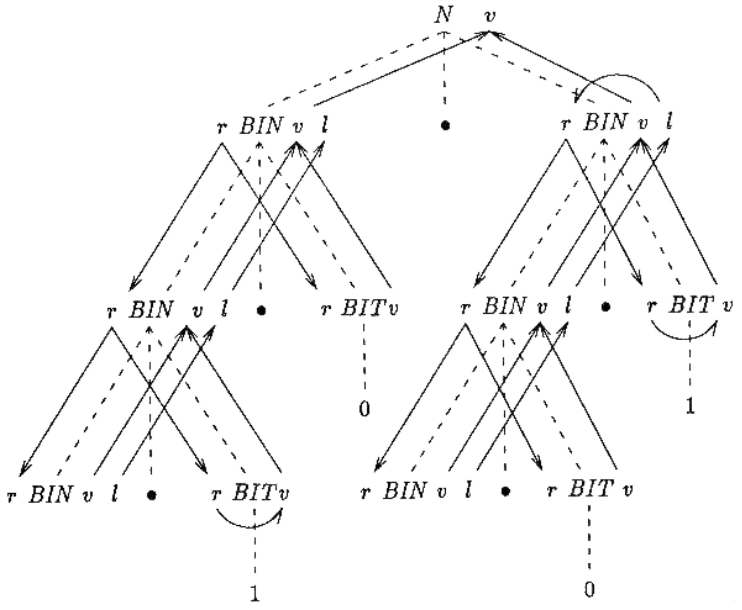


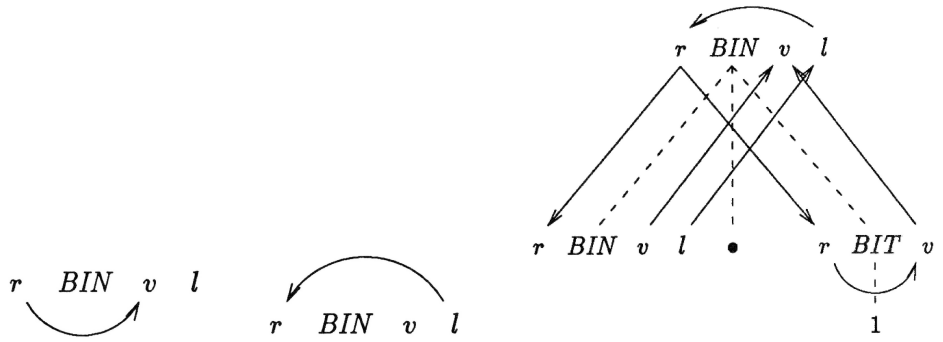
Abbildung 2.9: Individuellem Abhängigkeitsgraphen der Attributgrammatik *Bin_to_Dec* für das Wort „10.01“ (nach [WM92, S. 385])

werden.

Charakteristische Abhängigkeitsgraphen

Aus dem Kontext, in dem die Produktionsvorkommen auftreten, ergeben sich zusätzliche Abhängigkeiten, die aus dessen oberen und unteren Fragmenten im Syntaxbaum resultieren. Um diese Abhängigkeiten ebenfalls zu erfassen ist eine Erweiterung des produktionslokalen zum charakteristischen Abhängigkeitsgraphen notwendig. Für das Nonterminal X der linken Produktionsseite muss der obere charakteristische Graph berechnet werden. Dieser ist definiert durch alle Kanten von $a \in Syn(X)$ zu $b \in Inh(X)$, wenn ein (transitiver) Weg im oberen Baumfragment der Produktion von a nach b führt. Für die Symbole der rechten Produktionsseite wird jeweils der untere charakteristische Graph erzeugt. Komplementär zum oberen charakteristischen Graphen wird hierbei das jeweils untere Baumfragment des Symbols nach Kanten für $a \in Inh(X)$ zu $b \in Syn(X)$ durchsucht. Wird nun der produktionslokale Abhängigkeitsgraph um den charakteristischen oberen und die charakteristischen unteren Graphen ergänzt, so erhält man eine Abhängigkeitsrelation für alle Attributexemplare des jeweiligen Produktionsvorkommens. Ein effizienter Algorithmus zur Berechnung dieser Abhängigkeiten findet sich in [WM92, S. 389f].

Ein exemplarischer unterer und oberer charakteristischer Graph für die Produktion $BIN \rightarrow BIN.BIT$ ist in Abbildung 2.10(a) bzw. 2.10(b) zu sehen. Der untere charakteristische Graph wurde für den Knoten links unterhalb der Wurzel im Ableitungsbaum 2.9 berechnet, der obere charakteristische Graph für den Blattknoten BIN des rechten Teilbaumes der Wurzel.



(a) Unterer charakteristischer Graph zu BIN

(b) Oberer charakteristischer Graph zu BIN

(c) Produktionslokaler Abhängigkeitsgraph mit oberem und unterem charakteristischem Graphen

Abbildung 2.10: Charakteristische Graphen für BIN (Quelle: [WM92, S. 387,392])

Abbildung 2.10(c) zeigt für das oberste Produktionsexemplar der Produktion $BIN \rightarrow BIN.BIT$ des rechten Teilbaums aus 2.9 die Kombination aus produktionlokalem Abhängigkeitsgraphen, oberem und unterem charakteristischen Graphen.

Zyklenfreiheit

Voraussetzung für die Auswertung der Attribute ist die Zyklenfreiheit des Gleichungssystems. Wie beschrieben ist eine Attributgrammatik in Normalform in den lokalen Abhängigkeitsgraphen stets zyklenfrei. Die globale Zyklenfreiheit für ein bestimmtes Wort der Sprache lässt sich im individuellen Abhängigkeitsgraphen dieses Wortes leicht durch einen entsprechenden Suchalgorithmus testen. Eine Attributgrammatik ist jedoch nur dann zyklenfrei (semantisch wohldefiniert), *„falls kein Syntaxbaum mit einer zyklischen Abhängigkeit eines Attributvorkommens existiert“* [Pen94, S. 143]).[19]

Um Zyklenfreiheit zu garantieren müssen die Abhängigkeiten für beliebige Syntaxbäume analysiert werden. Dies ist schwierig, da der Aufbau der oberen und unteren charakteristischen Graphen, im Gegensatz zu den produktionslokalen Abhängigkeitsgraphen, von ihrer Position im Ableitungsbaum beeinflusst wird. Nach [JOR75] lässt sich die semantische Wohldefiniertheit einer Attributgrammatik effektiv prüfen. Dieser Test hat allerdings exponentielle Komplexität und ist daher nicht für die Praxis geeignet[20]. Aus diesem Grund wird üblicherweise auf Näherungslösungen zurückgegriffen (vgl. [WM92, S. 397ff]).

[19] Es ist aber durchaus möglich, dass eine zirkuläre Attributgrammatik zykelfreie Ableitungsbäume besitzt (vgl. [KV97, S. 32]).

[20] Der Algorithmus für den Test auf Zyklenfreiheit findet sich in [AEU88a, S. 410ff] und [KV97, S. 38ff].

2.2.4 Attributauswertung

Attributauswertung bezeichnet die Zuordnung von Werten zu den Attributexemplaren des attributierten Syntaxbaums gemäß den definierten semantischen Regeln.

> *Ein Attributauswerter ist ein Algorithmus, der den Strukturbaum auf bestimmten Wegen entlang der Kanten durchläuft und Attributwerte gemäß Attributregeln [...] berechnet [...]. Die Konstruktion des Algorithmus muß garantieren, daß alle Attributwerte von denen eine Attributregel oder Kontextbedingung abhängt, berechnet sind, bevor die Attributregel angewandt wird. Deshalb spielen die in den Attributregeln ausgedrückten Abhängigkeiten zwischen Attributen eine zentrale Rolle bei der Konstruktion.* [Kas90, S. 105]

Bewerteter Syntaxbaum

Die Ergebnisse des Auswertungsalgorithmus werden an den Knoten des Syntaxbaums notiert. Dazu werden diese um Datenfelder[21] erweitert, die die Attribute des jeweiligen Symbols aufnehmen. Entsprechend den semantischen Regeln können die Werte dann in einer gültigen Reihenfolge berechnet und in den Datenfeldern abgespeichert werden, wo sie als Eingabe für weitere Berechnungen bzw. als Endergebnis bereitstehen. Wurde die Auswertung für alle Attributexemplare durchgeführt, so spricht man von einem *bewerteten* (annotated) Syntax- bzw. Parsebaum (vgl. [AEU88a, S. 342]).

Auswertungsverfahren

Der Auswertungsvorgang kann nach [WM92, S. 398] in zwei Phasen aufgeteilt werden:

Strategiephase
: Basierend auf den Abhängigkeiten wird eine Auswertungsordnung bestimmt, die für jedes Attributexemplar die im Vorfeld durchzuführenden Berechnungen festlegt.

Auswertungsphase
: Wertet die Attributexemplare gemäß der Auswertungsreihenfolge aus. Eine *bedarfsgetriebene Auswertung* beginnt mit einer Menge von Attributexemplaren, deren Werte gewünscht sind und berechnet ihre Argumente durch Rekursion. Eine *datengetriebene Auswertung* beginnt mit bekannten Werten und berechnet in jeder Iteration alle Funktionen, deren Argumente vollständig vorliegen.

[21] In der Praxis werden aus Gründen der Performanz meist andere Ansätze verwendet (vgl. [WG98, S. 212f]).

Je nach angewandter Strategie zur Bestimmung der Auswertungsreihenfolge lassen sich die Auswertungsverfahren nach [WM92, S. 382] und [KV97, S. 47ff] in zwei Gruppen einteilen:

Dynamische Verfahren
: Setzen kein Wissen über den Aufbau des Gleichungssystems (und damit der Attributierung) voraus. Die Auswertungsordnung wird individuell für die jeweilige Attributgrammatik und den konkreten Ableitungsbaum bestimmt.

Statische Verfahren
: Nutzen Wissen über die Abhängigkeitsstruktur der Grammatik, um einen statischen Auswerter für beliebige Ableitungsbäume zu erzeugen.

Ein einfaches Beispiel für ein vollständig dynamisches Verfahren ist eine Eliminationsstrategie, die in jedem Schritt ein Attributexemplar sucht, das nur von bereits berechneten (oder vorgegebenen) Werten abhängt. Alternativ kann eine dynamische Auswertung auch durch das Festlegen einer totalen Ordnung auf den individuellen Abhängigkeitsgraphen erfolgen.

Der Nachteil der dynamischen Verfahren besteht darin, dass sie *„platz- und zeitaufwendig und [deshalb] in der Praxis nicht zu verwenden sind“* [WM92, S. 399]. Desweiteren wird vorausgesetzt, dass das Gleichungssystem zum Berechnungszeitpunkt vollständig vorliegt. In der Praxis sind die Syntaxanalyse und die semantische Analyse in Compilern aber oft stark verzahnt[22] um die Performanz zu erhöhen. Statische Verfahren basieren auf speziellen Klassen von Attributgrammatiken mit eingeschränkter Ausdrucksmächtigkeit[23], deren für eine gültige Auswertung nötige Besuchsfolgen sich bereits zur Definitionszeit aus der Grammatik ableiten lassen.

2.3 Datenflussanalyse

Im Compilerbau wird die Datenflussanalyse (DFA) eingesetzt um aus der Struktur des Quellprogramms optimierungsrelevante Daten zu gewinnen. Sie ist *„eine Art der abstrakten Interpretation von Programmen und wird zumeist für die intraprozedurale Analyse [...] verwendet. Dabei wird aus einem Programm ein Flussgraph gemäß den Programmkonstrukten aufgebaut und anschließend annotiert“* [BH98, S. 130]. Die Datenflussanalyse wird beispielsweise zur Elimination unerreichbaren Codes und der Freigabe nicht mehr benötigter Variablen eingesetzt.

[22] Bei der sogenannten parsergesteuerten Auswertung orientiert sich die Auswertungsstrategie an den speziellen Eigenschaften der jeweiligen Parserimplementierung. Auf diese Techniken wird in [WM92, S. 412ff] eingegangen.

[23] Diese Klassen, sowie die dazugehörigen statischen Auswerter, werden in [KV97, S. 48ff] besprochen.

Programmflussgraph

Der Kontrollfluss eines Programms lässt sich durch einen Programmflussgraphen visualisieren, der alle strukturell möglichen Ablaufpfade enthält (vgl. [Kas90, S. 197]). Dieser basiert auf einem Flussgraph, d.h. einem zusammenhängenden gerichteten Graphen $G = (N, E, n_{in}, n_{out})$ mit Knotenmenge N, Kantenmenge E und Eingangs- bzw. Ausgangsknoten n_{in} und n_{out}. Die Anweisungen bzw. die Grundblöcke des Programms werden durch eine Markierungsfunktion *prog* an den Knoten des Graphen annotiert[24] (vgl. [BH98, S. 131f], [AEU88b, S. 645ff]). Jeder Grundblock, der aus mehreren Programmanweisungen bestehen kann, erhält so einen Eingangs- und einen Ausgangspunkt im Graphen. Vom Startpunkt aus unerreichbare Knoten sind in keinem gültigen Ablaufpfad enthalten und können damit entfernt werden, ohne die Ausführungssemantik zu verändern.

Größter / Kleinster Fixpunkt

Die abstrakte Interpretation des Flussgraphen führt zu einer Abschätzung von Laufzeiteigenschaften der einzelnen Programmfragmente. Es kann sich aber nur um eine Annäherung an das tatsächliche Verhalten handeln, weil sämtliche zur Laufzeit möglichen Ablaufpfade in die Auswertung einbezogen werden müssen. Schleifen, die Rückwärtskanten und damit Zyklen im Programmflussgraph erzeugen, erfordern zudem eine iterative Vorgehensweise bei der Berechnung des Informationsflusses. Die Datenflussgleichungen werden dazu mit den Ergebnissen der jeweils letzten Iteration erneut ausgewertet, bis das System einen stabilen Fixpunkt erreicht. Nach [BH98, S. 133] müssen die Informationen, die entlang den Kanten des Graphen propagiert werden, eine Halbordnung bilden, um die Bestimmung eines Infimums (größter Fixpunkt) bzw. eines Supremums (kleinster Fixpunkt) zu ermöglichen. Zu beachten ist, dass im Zweifelsfall immer eine konservative Abschätzung getroffen werden muss, da die Semantik des Programms durch den Optimierungsprozess nicht verändert werden darf (vgl. [AEU88b, S. 747]).

Datenflussgleichungen

Die DFA verknüpft Informationen benachbarter Programmfragmente, je nach zu lösendem Datenflussproblem beispielsweise Variablen, Ausdrücke oder Zuweisungen. Die Informationen, die vor bzw. nach einer Anweisung B verfügbar sind, werden durch die Mengen $in(B)$ und $out(B)$ repräsentiert. Der Inhalt dieser Mengen wird durch Datenflussgleichungen definiert, deren Form von der Lösungsstrategie des jeweiligen Problems abhängt. Zu deren Formulierung stehen $gen(B)$, die Menge der in B erzeugten, und $kill(B)$, die Menge der in B zerstörten Informationen zur Verfügung. $Vorg(B)$ und $Nachf(B)$ enthalten die Vorgänger bzw. die Nachfolgerknoten von B. (vgl [Kas90, S. 202])

[24] Alternativ können die Anweisungen auch die Kanten des Graphen bilden.

Abhängig von der gewählten Analyserichtung und der gewünschten Fixpunktmenge (gesteuert durch den Konfluenzoperator $\bigcup$ bzw. $\bigcap$) lassen sich nach den Definitionen aus [AEU88b, S. 764] vier generische Gleichungssysteme aufstellen:

Größter Fixpunkt mit Vorwärtsanalyse

$$in(B) = \bigcap_{X \in Vorg(B)} out(X)$$
$$out(B) = gen(B) \cup (in(B) - kill(B))$$

Zur Ermittlung des *Infimums* wird die Eingabe von B aus dem Schnitt der Ausgabemengen seiner Vorgänger berechnet. Dies führt dazu, dass nur Informationen berücksichtigt werden, die B über *alle* Ausführungspfade erreichen. Die Ausgabe entspricht der Eingabe inklusive der in B erzeugten, aber ohne die in B zerstörten Informationen. $in(n_{in})$ wird mit allen Werten der Domäne initialisiert.

Größter Fixpunkt mit Rückwärtsanalyse

$$out(B) = \bigcap_{X \in Nachf(B)} in(X)$$
$$in(B) = gen(B) \cup (out(B) - kill(B))$$

Bei der Rückwärtsanalyse erfolgt der Informationsfluss in umgekehrter Richtung, d.h. von den nachfolgenden Knoten zu den Vorgängern. Die Auswertung beginnt hier beim Ausgangsknoten $out(n_{out})$, der ebenfalls mit dem gesamten Wertebereich initialisiert wird.

Kleinster Fixpunkt mit Vorwärtsanalyse

$$in(B) = \bigcup_{X \in Vorg(B)} out(X)$$
$$out(B) = gen(B) \cup (in(B) - kill(B))$$

Die Berechnung des kleinsten Fixpunkts (*Supremum*) erfolgt durch Vereinigung der Ausgaben der jeweiligen Vorgänger. Dadurch werden alle Informationen gesammelt, die B auf einem *beliebigen* Ausführungspfad erreichen. Der Initialwert von $in(n_{in})$ ist $\emptyset$.

Kleinster Fixpunkt mit Rückwärtsanalyse

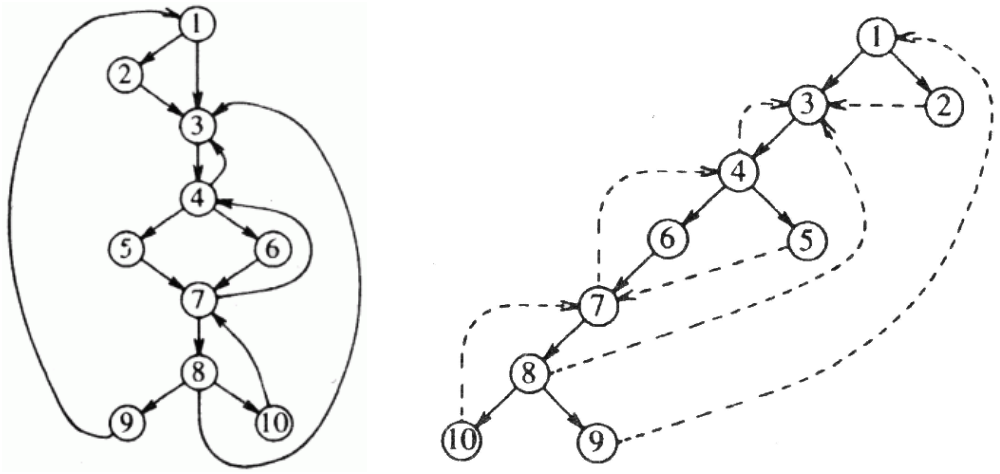
$$out(B) = \bigcup_{X \in Nachf(B)} in(X)$$
$$in(B) = gen(B) \cup (out(B) - kill(B))$$

Die Informationen werden in Rückwärtsrichtung weitergeleitet. Anfangswert für $out(n_{out})$ ist wiederum $\emptyset$.

Allgemeine Lösungsverfahren

Enthält der Flussgraph keine Zyklen, so kann die Berechnung des Gleichungssystems durch Substitution der Gleichungen erfolgen. Ist dies nicht der Fall, muss ein allgemeineres Verfahren angewendet werden. Nach [Kas90, S. 204] muss dieses die minimalste Lösung ermitteln, um sinnvolle Daten zur Optimierung bereitzustellen.

Ein iterativer Algorithmus nach [AEU88b, S. 765f] setzt die Mengen $in(B)$ und $out(B)$ auf geeignete Anfangswerte und wiederholt deren Berechnung bis keine Änderungen mehr auftreten, der Fixpunkt also erreicht wurde. Soll beispielsweise durch eine Vorwärtsanalyse der kleinste Fixpunkt gefunden werden, so wird $in(B) = \emptyset$ und $out(B) = gen(B)$ für alle B gesetzt, und deren Auswertung wiederholt bis ein stabiler Zustand erreicht ist.



(a) Programmflussgraph (b) Flussgraph in Tiefenordnung

Abbildung 2.11: Darstellung eines Flussgraphen in Tiefenordnung (Quelle: [AEU88b, S. 809f])

Ein verbesserter Algorithmus basiert auf der Darstellung des Flussgraphen in Tiefenordnung. Dazu wird der Graph in Tiefensuchordnung durchlaufen und aus den Vorwärtskanten eine zyklenfreie Baumdarstellung erzeugt, während gefundene Rück- und Kreuzkanten gesondert eingetragen werden. Abbildung 2.11(b) zeigt die Tiefenordnung des Graphen 2.11(a). Die Programmstruktur kann nun in die iterative Auswertung einbezogen werden, indem die Gleichungen in der Reihenfolge ihrer Tiefenordnung (bei einer Rückwärtsanalyse entgegen der Tiefenordnung) ausgewertet werden. Somit reduziert sich die Anzahl der nötigen Durchläufe. (vgl. [AEU88b, S. 823ff])

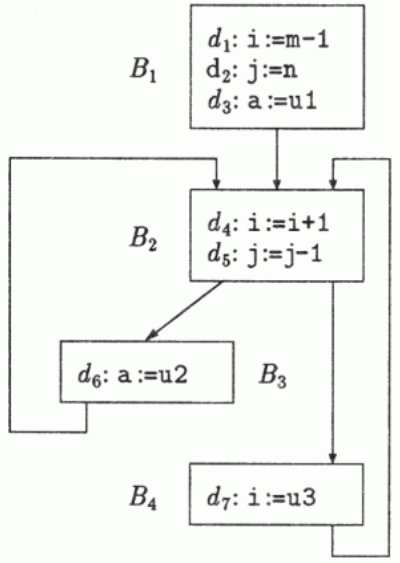
Beispiel: Verfügbare Definitionen

Eine typische Anwendung der Datenflussanalyse ist die Ermittlung verfügbarer Defini-

tionen (reaching definitions). Dabei handelt es sich um Zuweisungen an Variablen, die eine bestimmte Stelle im Programm erreichen, ohne auf diesem Weg zerstört worden zu sein. Da im Allgemeinen unentscheidbar ist, ob ein konkreter Pfad durchlaufen wird, müssen für eine konservative Abschätzung des Ergebnisses alle möglichen Pfade auf dem Weg zu einer Anweisung berücksichtigt werden. Damit ist klar, dass es sich um eine vorwärtsgerichtete Analyse des kleinsten Fixpunkts handeln muss. Die Menge $gen(B)$ bezeichnet hier die Menge der in B erzeugten, und $kill(B)$ die in B überschriebenen, und damit zerstörten, Variablendefinitionen. Nach [AEU88b, S. 760f] lässt sich daraus ein zyklenfreies Gleichungssystem erstellen, das aufgrund dieser Eigenschaft durch eine Attributierung auf den Kontrollstrukturen der Programmiersprache (vgl. [AEU88b, S. 746ff]) gelöst werden kann.

```
/* d1 */  i := m-1;
/* d2 */  j := n;
/* d3 */  a := u1;
          do
/* d4 */        i := i+1;
/* d5 */        j := j-1;
              if e1 then
/* d6 */            a := u2
              else begin
/* d7 */                    i := u3;

                  end
          while e2
```

(a) Programm mit Variablendefinitionen d_1-d_7



(b) Flussgraph des Programms

	gen	kill	in	out
B1	d_1, d_2, d_3	d_4, d_5, d_6, d_7		d_1, d_2, d_3
B2	d_4, d_5	d_1, d_2, d_7	$d_1, d_2, d_3, d_4, d_5, d_6, d_7$	d_3, d_4, d_5, d_6
B3	d_6	d_3	d_3, d_4, d_5, d_6	d_4, d_5, d_6
B4	d_7	d_1, d_4	d_3, d_4, d_5, d_6	d_3, d_5, d_6, d_7

(c) Stabiles Ergebnis der Datenflussanalyse nach drei Iterationen

Abbildung 2.12: Datenflussanalyse für das „verfügbare Definitionen"-Problem nach [AEU88b, S. 758,767f].

Abbildung 2.12(b) zeigt den Flussgraph des Programms 2.12(a). Das Ergebnis der DFA ist in 2.12(c) zu sehen. Die Mengen *gen* und *kill* werden lokal berechnet und sind konstant. So enthält $gen(B_2)$ beispielsweise die Definitionen d_4, d_5, da diese in B_2 erzeugt werden, $kill(B_2)$ dagegen d_1, d_2, d_7, weil deren Zuweisungen an die Variablen i und j durch

B_2 zerstört werden. Die endgültigen Werte für *in* und *out* werden im zweiten Berechnungsdurchlauf nach der Initialisierung erreicht und entsprechen damit dem Fixpunkt, an dem sich das System stabilisiert. Dass das Ergebnis korrekt sein muss, lässt sich an diesem Beispiel einfach nachvollziehen. $in(B_2)$ enthält beispielsweise alle existierenden Definitionen, da B_2 von allen Knoten direkt bzw. über Schleifen erreicht werden kann.

Weitere typische Einsatzgebiete der Datenflussanalyse sind die Berechnung verfügbarer Ausdrücke [AEU88b, S. 768ff], aktiver Variablen [AEU88b, S. 773ff] und die Typisierung in objektorientierten Sprachen [BH98, S. 135ff], wobei zum Teil auch alternative Gleichungssysteme zum Einsatz kommen.

2.4 Metamodellierung

Metamodellierung spielt in der Informatik, insbesondere im Bereich Softwareentwicklung, eine große Rolle. Durch einen fortschreitenden Trend zur Automatisierung des Entwicklungsprozesses, sowie durch immer komplexer werdende Softwaresysteme werden Techniken benötigt, die aus vorgegebenen Spezifikationen weitgehend selbstständig Softwarefragmente erzeugen können. Ein wichtiger Aspekt ist dabei die Minimierung der Ressourcen, die für einfache, aber zeitaufwändige und fehleranfällige Routinetätigkeiten aufgewendet werden müssen. Ein aktueller Ansatz, der intensiven Gebrauch von Metamodellierungstechniken macht, ist die Model Driven Architecture (MDA)[25] der Object Management Group (OMG).

In Abschnitt 2.4.1 werden die allgemeinen Prinzipien der Metamodellierung, sowie deren Implementierung durch die UML vorgestellt. In 2.4.2 wird dann beschrieben, wie durch OCL semantische Restriktionen auf Metamodellen formuliert werden können.

2.4.1 Unified Modeling Language

„A metamodel is a model used to model modeling itself“[Gro06a, S. 29]. Ein Metamodell ist also ein Formalismus, der eine bestimmte Struktur vorgibt, die wiederum eine Klasse von Modellen beschreibt. Ein solches Modell repräsentiert demnach eine konkrete Instanz seines definierenden Metamodells. Auch ein Metamodell selbst ist Instanz einer übergeordneten Modellierungsprache, dem Meta-Metamodell. Theoretisch lässt sich so eine unendliche Hierarchie von Modellen und Metamodellen aufbauen, die in der Praxis aber üblicherweise auf eine im jeweiligen Kontext sinnvolle Anzahl von Hierarchieebenen eingegrenzt wird. Je nachdem, ob die komplette Architektur oder nur Ausschnitte davon betrachtet werden, kann eine Modellierungsinstanz sowohl als Modell wie auch als Metamodell interpretiert werden.

[25] http://www.omg.org/mda

Beziehung zwischen Metamodell- und Modellelementen

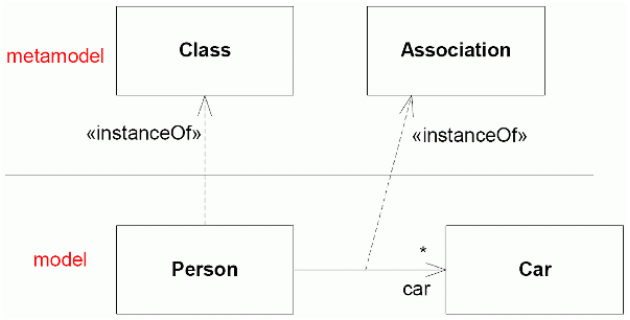


Abbildung 2.13: Beziehung zwischen Metamodell- und Modellelementen (Quelle: [Gro07a, S. 17])

Das Prinzip der Metamodellierung wird in Abbildung 2.13 illustriert. Jedes Element des Modells ist Instanz genau eines Metaobjekts (dargestellt durch eine *«instanceof»* Beziehung zwischen den Objekten).

Das in der Softwareentwicklung am weitesten verbreitete Metamodellierungsrahmenwerk ist die UML[26], die von der OMG[27] entwickelt wird. Sie definiert eine 4-Schichten Architektur, die aus den folgenden Hierarchieebenen besteht (vgl. [Gro07a, S. 16f]):

M3 Schicht
: Diese Schicht repräsentiert das Meta-Metamodell und ist damit das Fundament der Metamodellierungshierarchie. Der Zweck dieser Ebene ist es, eine Sprache zur Beschreibung von Metamodellen zu definieren. Innerhalb UML wird diese Aufgabe durch die Meta Object Facility (MOF) übernommen.

M2 Schicht
: Definiert Metamodelle, die den Aufbau der M1 Modelle beschreiben. Die Modellierungssprachen der Unified Modeling Language werden in dieser Ebene festgelegt.

M1 Schicht
: Instanzen der M2 Metamodelle, zum Beispiel Klassen- oder Aktivitätsdiagramme, beschreiben semantische Domänen.

M0 Schicht
: Repräsentiert konkrete Instanzen auf der untersten Ebene, beispielsweise Laufzeitobjekte in Programmen.

[26] Siehe dazu [Gro07a] und [Gro07b].
[27] http://www.omg.org

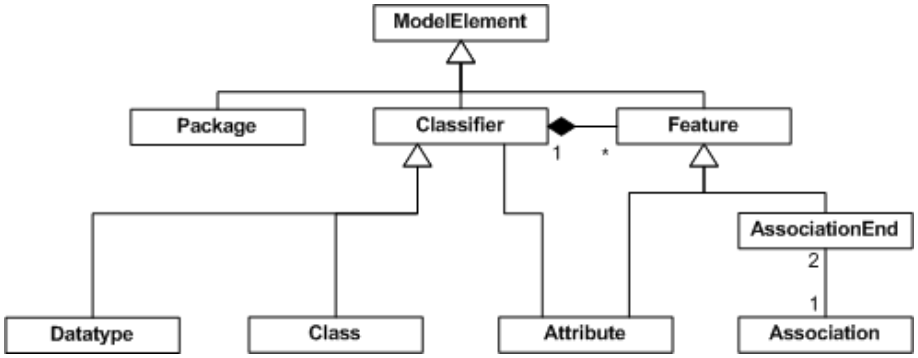


Abbildung 2.14: Vereinfachte Darstellung des UML Meta-Metamodells (nach [Wik08])

Abbildung 2.14 zeigt eine vereinfachte Sicht auf die wichtigsten Elemente des UML Meta-Metamodells. Alle Objekte sind eine Spezialisierung des generischen Modellelements `ModelElement`. Um die Elemente zu strukturieren, können sie in `Package`s organisiert werden. Klassen (`Class`) und Datentypen (`Datatype`) sind als `Classifier` definiert, die wiederum über `Feature`s, d.h. Attribute und Assoziationsenden, verfügen können. Der Typ eines Attributs (`Attribute`) wird wiederum durch einen `Classifier` festgelegt. Assoziationsenden (`AssociationEnd`) bilden Ankerpunkte für Assoziationen (`Association`), die jeweils zwei Enden verbinden. Der größte Teil dieser Modellobjekte wird durch die UML Infrastruktur spezifiziert. Diese wird unter anderem durch die MOF (vgl. [Gro06a]) importiert, um die UML zugrunde liegende Metamodellierungsarchitektur zu beschreiben. Diese Schicht ist reflektiv, d.h. sie bildet ihr eigenes Metamodell und kann damit durch sich selbst vollständig beschrieben werden, so das keine weitere Metaebene benötigt wird.

In Abbildung 2.15 werden die Zusammenhänge zwischen den Ebenen anhand eines konkreten Beispiels dargestellt. Die Elemente der Schicht M2 sind Instanzen des M3-Typs `Class`. Dabei ist es wichtig die Semantik der Konzepte, die in den einzelnen Ebenen festgelegt werden, strikt zu trennen. So repräsentieren die Modellelemente `Class` des M2- und des M3-Layers zwei verschiedene Typen die, trotz identischen Namens, eine unterschiedliche Semantik aufweisen. Die M3-Klasse repräsentiert eine Metaklasse von der andere Objekttypen abgeleitet werden können, während sie in M2 als Klasse im Sinne eines UML Klassendiagramms interpretiert werden kann. Das Benutzermodell, das in M1 dargestellt wird, entspricht der üblichen Modellierungssicht eines Entwicklers, der die Komponenten eines Softwaresystems entwirft. Auf der untersten Ebene befinden sich Objekte der Realität, die zur Laufzeit existieren.

Abstrakte/Konkrete Syntax und Semantik

Die Definition einer Modellierungssprache erfordert neben der Festlegung des Metamodells, das die syntaktischen Struktur (*abstrakte Syntax*) beschreibt, auch die damit verknüpfte Vorgabe einer (visuellen) Notation der Sprachelemente. Diese wird als *konkrete Syntax* bezeichnet. Beispielsweise legt die konkrete Syntax der UML-Klassendiagramme

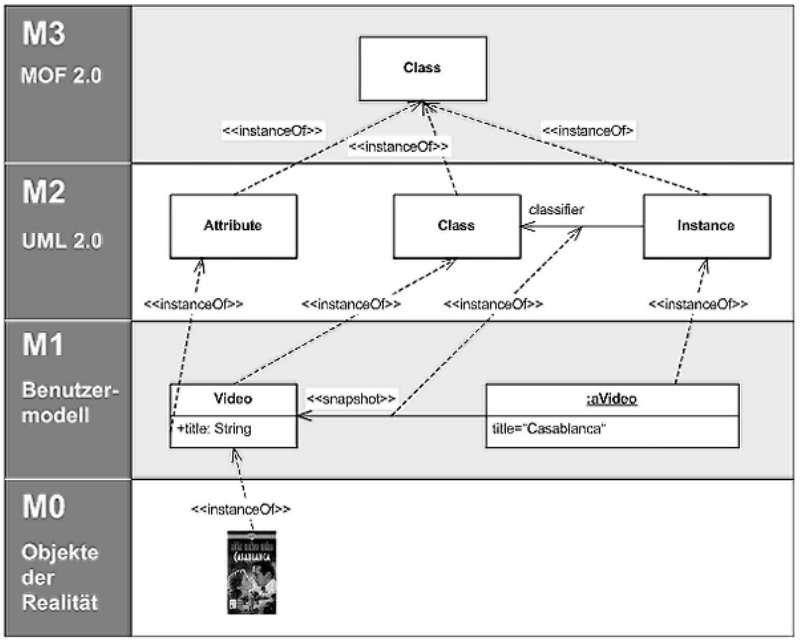


Abbildung 2.15: Die Ebenen der UML Metamodellierungsarchitektur (Quelle: [Gro07a, S. 19])

unter anderem die Notation einer Klasse als ein mit dem Klassennamen überschriebenes Rechteck mit Einträgen für Attribute und Operationen, fest. Wie bei anderen formalen Sprachen muss zusätzlich zur syntaktischen Beschreibung eine Semantik hinterlegt werden, die die Interpretation und die Laufzeiteigenschaften der Modellelemente beschreibt.

2.4.2 Object Constraint Language

Es zeigt sich beim Einsatz von Modellen - beispielsweise in Form von UML-Diagrammen - häufig, dass diese nicht ausdrucksstark genug sind, um alle relevanten Aspekte einer Spezifikation festzuhalten. Wie auch bei kontextfreien Sprachen muss die abstrakte Syntax um zusätzliche semantische Anforderungen ergänzt werden. Derartige Erweiterungen, die in textueller Form, also einer natürlichsprachlichen Beschreibung, festgehalten werden, sind tendenziell unvollständig, mehrdeutig oder inkonsistent. Aufgrund des fehlenden Formalismus eignen sich diese nicht zur automatisierten Verarbeitung im Entwicklungsprozess[28]. Auch die Verwendung algebraischer Semantiken (z.B. nach [RCA00]) ist hier problematisch, da diese auf komplexen mathematischen Zusammenhängen aufbauen und sich ihr Einsatz in der Praxis daher schwierig gestaltet.

Um der Herausforderung zu begegnen, sowohl die Ansprüche an eine formale Spezifikationssprache zu erfüllen, als auch die Anwendbarkeit im praktischen Umfeld zu ge-

[28] Anwendungsbeispiele hierfür wären etwa die Formulierung semantischer Restriktionen, automatisierte Konsistenzprüfungen oder Codegenerierung.

währleisten, wurde die Object Constraint Language entwickelt. Sie ist Teil der Unified Modeling Language: *„[OCL is] a formal language used to describe expressions on UML models. These expressions typically specify invariant conditions that must hold for the system being modeled [...]"*[Gro06b, S. 5]. Neben anwendungsspezifischen Restriktionen, z.B. der Einschränkung von Attributen auf bestimmte Wertebereiche, können auch komplexe Beschreibungen der Semantik[29] sowie programmiersprachenunabhängige Modellabfragen und Transformationen[30] implementiert werden.

Die OCL-Spezifikation [Gro06b] führt als wichtige Eigenschaften unter anderem auf:

- OCL ist eine pure Spezifikationssprache, d.h. die Auswertung von OCL-Ausdrücken hat keinerlei Seiteneffekte auf das zugrunde liegende Modell.
- Es handelt sich nicht um eine Programmiersprache, OCL kann keine Operationen durchführen und OCL-Ausdrücke sind nicht direkt ausführbar.
- Die Sprache ist getypt, d.h. jeder OCL Ausdruck hat einen definierten Typ und für Ausdrücke gilt, dass sie wohlgeformt bzgl. dieser Typen sein müssen.
- Die Auswertung eines Ausdrucks ist ein atomarer Prozess, d.h. der Status der Modellobjekte ändert sich während der Auswertung nicht.
- Es existieren verschiedene Arten von Restriktionen, die für unterschiedliche Zwecke eingesetzt werden: Invarianten müssen zu jedem Zeitpunkt gelten. Pre- und Postconditions beschreiben spezielle Zustandseigenschaften vor bzw. nach der Ausführung einer Operation.

Ein OCL Ausdruck ist immer im Kontext eines UML Modelltyps (`Classifier`) definiert. Dabei kann es sich zum Beispiel um Klassen oder deren Features - also Klassenattribute, Operationen oder Assoziationsenden - handeln. Werden OCL-Anweisungen innerhalb von UML-Diagrammen formuliert, so zeigt eine Verbindungslinie zum entsprechenden Modellobjekt den Kontext an. Außerhalb des Metamodells muss dieser jedoch explizit deklariert werden. Die Auswertung der OCL-Constraints im Modell erfolgt dann an allen Instanzen des jeweiligen Kontexts.

Die Formulierung von Restriktionen setzt voraus, dass auf Modellobjekte und deren Eigenschaften, beispielsweise die Instanzwerte der Klassenattribute, zugegriffen werden kann. Das Schlüsselwort `self` referenziert zu diesem Zweck die Kontextinstanz des OCL-Ausdrucks, weitere Objekte können durch eine Navigationstechnik adressiert werden, die sich an den Assoziationen orientiert. Durch den Rollennamen, der das gegenüberliegende Assoziationsende identifiziert, werden die diesbezüglich in Relation stehenden Objekte angesprochen. Ist ein solcher nicht definiert, kann stattdessen auch der

[29] [Zie06] verwendet OCL, um die Semantik von UML-Diagrammen formal festzuhalten.

[30] Die OMG definiert dazu das auf MOF und OCL basierende Query/View/Transformation-Framework (http://www.omg.org/cgi-bin/doc?ptc/2007-07-07)

Objekttyp am Assoziationsende angegeben werden, sofern dadurch keine Mehrdeutigkeiten, z.B. an reflexiven Assoziationen, entstehen. Abhängig von der Multiplizität des Assoziationsendes besteht das Ergebnis entweder aus einem einzelnen Objekt oder einer Objektmenge. Die Funktion `allInstances` auf einen Typ angewandt, liefert dagegen alle existierenden Instanzen dieses Typs zurück. Weiterhin verfügt OCL über eine Reihe von Operatoren und eine Bibliothek vordefinierter Funktionen (vgl. [Gro06b, S. 137]), die den Entwurf von Restriktionen erleichtern.

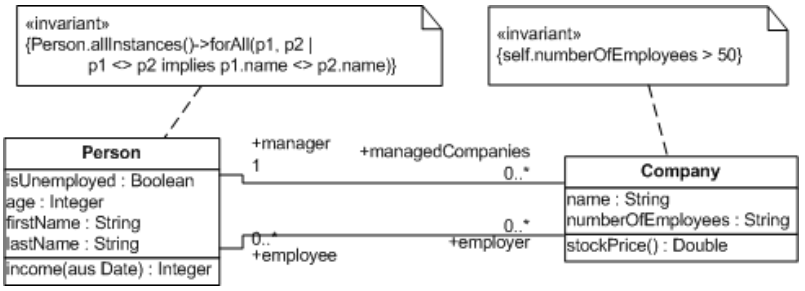


Abbildung 2.16: Ein Metamodell, das zwei OCL-Constraints enthält (vgl. [Gro06b, S. 7]

Abbildung 2.16 zeigt ein einfaches Klassendiagramm, das die Typen `Person` und `Company` definiert. Beiden wurde ein OCL-Constraint in Form einer Invariante zugeordnet, die zu jeder Zeit für alle Instanzen dieser Elemente gelten muss. Die Invariante der `Company`-Klasse greift über `self` auf den Wert des lokalen Klassenattributs `numberOfEmployees` zu, um sicherzustellen, dass die Anzahl der Firmenangestellten stets größer als 50 ist. Die Invariante von `Person` kombiniert das Schlüsselwort `allInstances` mit der `forAll`-Operation um eine Restriktion auf der Menge aller `Person`-Objekte zu definieren, die festlegt, dass deren Namen paarweise verschieden, Personennamen damit also eindeutig, sein müssen.

```
context Person::income(d : Date) : Integer
    post: result = 5000

context Company
    inv: self.manager.isUnemployed = false
    inv: self.employee->notEmpty()
```

Abbildung 2.17: OCL-Constraints außerhalb des Metamodells (Quelle: [Gro06b, S. 8ff])

In 2.17 sind zwei weitere Constraints auf den Klassen `Person` und `Company` zu sehen. Der Kontext der Ausdrücke wird hier nicht graphisch dargestellt, sondern ist Teil

der Deklaration. Die OCL-Ausdrücke können so auch außerhalb des Metamodells gespeichert werden. Im Kontext der Operation `income` der Klasse `Person` wurde eine Post-Condition definiert, die verlangt, dass der Rückgabewert dieser Operation nach jeder Ausführung genau 5000 beträgt. Die zweite Invariante navigiert über die Rollennamen `manager` und `employee` zu Instanzen von `Person` und belegt das Ergebnis dieser Abfrage mit Restriktionen. Anhand der Multiplizitäten ist für den Rückgabewert der `manager`-Assoziation klar, dass es sich um ein einzelnes Objekt vom Typ `Person` handeln muss, während die Navigation über die Rolle `employee` eine Menge von Instanzen dieser Klasse zurückliefert.

3 Anwendungsbeispiele

Dieses Kapitel stellt Beispiele vor, anhand derer im weiteren Verlauf dieser Arbeit demonstriert wird, wie das vorgestellte Attributierungskonzept in der Praxis angewendet werden kann, um strukturelle Analysen auf UML Metamodellen durchzuführen.

Neben einem, in 3.1 präsentierten, einfachen Beispiel, das in späteren Abschnitten verwendet wird, um verschiedene Aspekte der Attributierungstechnik zu visualisieren, beschreibt 3.2 einen komplexeren Anwendungsfall, in dem über eine Attributierung Aussagen zum Verhalten der Modellobjekte getroffen werden sollen.

3.1 Referenzbeispiel

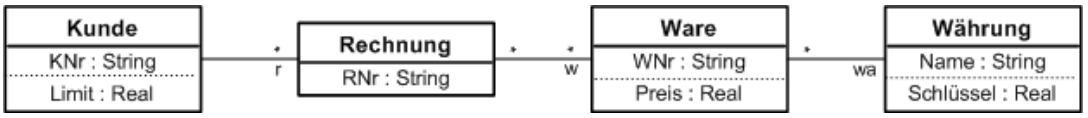


Abbildung 3.1: Metamodell eines einfachen Kundenverwaltungssystems

Das Beispiel aus Abbildung 3.1 wird im weiteren Verlauf zur Demonstration der einzelnen Aspekte des Attributierungskonzepts dienen. Es handelt sich dabei um ein einfaches Abrechnungssystem in UML-Klassendiagrammnotation. In diesem besitzt jeder *Kunde* neben einer eindeutigen ID (*KNr*) ein Umsatzlimit (*Limit*), das pro einzelner *Rechnung* nicht überschritten werden darf. Der Rechnungsumsatz berechnet sich aus den *Waren*, die dieser *Rechnung* zugeordnet sind. Deren *Preis* ist länderspezifisch, und muss zur Abrechnung erst durch den jeweiligen *Währungsschlüssel* in die Firmenwährung umgerechnet werden. Ziel ist es, über eine Attributierung zum Einen die Umsätze aller Kunden zu berechnen und anzuzeigen, zum Anderen soll die Einhaltung des Umsatzlimits überprüft werden.

Schwächen der OCL-Notation

Eine Betrachtung der äquivalenten OCL-Ausdrücke offenbart dabei bereits mehrere

Schwächen der Object Constraint Language, die durch Attributierung des Metamodells, wie in Kapitel 4 gezeigt, ganz oder zumindest teilweise behoben werden können:

- Um sicherzustellen, dass das Umsatzlimit pro Rechnung nicht überschritten wird, kann folgende OCL-Anweisung verwendet werden, die im Kontext des Rechnungsobjekts eine Invariante festlegt, die überprüft ob die Summe der, durch den Währungsschlüssel modifizierten, Warenpreise kleiner/gleich dem Kundenlimit ist:

  context Rechnung
  inv: self.w→collect(ware : Ware | ware.Preis *
  ware.wa.Schlüssel)→sum() <= self.kunde.Limit

  Die Aggregation der dazu notwendigen Navigationen in einem Ausdruck geht dabei zu Lasten der Übersichtlichkeit, was das intuitive Verständnis beeinträchtigt. Attribute beschränken den Transport von Modellinformationen und die Auswertung von Restriktionen dagegen auf ein lokales Umfeld, wodurch sich die Komplexität der einzelnen Ausdrücke stark reduziert.

- Da die Object Constraint Language nur die Formulierung von Restriktionen unterstützt, ist es nicht erlaubt Berechnungen durchzuführen, deren Ergebnis für eine weitere Verwendung zur Verfügung steht. Es spricht aber andererseits nichts dagegen, die über eine Attributierung berechneten Ergebnisse im Modellierungsvorgang oder sogar in Laufzeitumgebungen weiterzuverwenden.

- Lässt man letztere Einschränkung außer Acht, so kann eine (formal nicht korrekte) OCL-Anweisung formuliert werden, die den Gesamtumsatz des Kunden auswertet:

  context Kunde
  ?: self.r->collect(rechnung : Rechnung | rechnung.w→collect(ware :
  Ware | ware.Preis * ware.wa.Schlüssel)→sum())→sum()

  Dabei fällt zunächst auf, dass die Komplexität der OCL-Anweisung durch die erhöhte Reichweite der Navigation weiter zugenommen hat, und ihre Semantik nun endgültig nicht mehr auf den ersten Blick erkennbar ist. Obwohl aber auch klar ist, dass der größte Teil der Restriktion mit oben definierter Invariante übereinstimmt, kann diese jedoch nicht zur Formulierung dieses Ausdrucks wiederverwendet werden. Neben der Unübersichtlichkeit und dem erhöhten Definitions- und Wartungsaufwand wird dadurch auch die Performanz des Auswertungsvorgangs durch redundante Berechnungen stark beeinträchtigt. Werte, die für Attribute berechnet wurden, sind dagegen leicht weiterverwendbar und reduzieren damit die Komplexität der einzelnen Fragmente.

- Durch die Orientierung an der Struktur des Metamodells sind die OCL-Restriktionen starr und reagieren bereits gegenüber kleinen Änderungen des Aufbaus sehr empfindlich. Wird z.B. zwischen Rechnung und Ware zusätzlich ein Warenkorb modelliert, der die gekauften Waren aggregiert, so muss für alle existierenden OCL-Restriktionen geprüft werden, ob entsprechende Anpassungen notwendig sind, und diese dann an den jeweiligen Teilausdrücken implementiert werden. Bei einer Attributierung wird der Informationsfluss im Metamodell dagegen durch Abhängigkeiten repräsentiert. Dadurch lässt sich leicht erkennen, wie sich eine Änderung auf diesen auswirkt und welche Attribute davon betroffen sind. Gegebenenfalls kann dann die Attributierung ergänzt werden, wobei nur lokal an der geänderten Stelle eine Weiterleitung eingefügt werden muss, während alle anderen Ausdrücke ihre Gültigkeit beibehalten.
- Nicht explizit gefordert ist im Beispiel die Implementierung einer Prüfung auf Eindeutigkeit der Kundennummer. Dies könnte durch folgenden Eintrag in OCL realisiert werden:

  context Kunde
  inv: Kunde.allInstances()→forAll(k1, k2 | k1 <> k2 **implies** k1.KNr <> k2.KNr)

  Auch hier ist zur Formulierung einer vermeintlich einfachen semantischen Restriktion ein komplexer Ausdruck in mathematischer Schreibweise nötig, dessen Funktionalität in einer Attributierung leicht hinter einem Attribut mit vordefinierter Funktionalität verborgen werden könnte.

3.2 Implementierungsbeispiel (JWT)

Zur Demonstration der Anwendbarkeit von Attributierungen zur strukturellen Analyse von Modellen werden Geschäftsprozessen herangezogen, die mit Hilfe der JWT-Toolsuite modelliert werden.

Java Workflow Tooling (JWT)

Java Workflow Tooling (JWT) ist Teil der offenen Entwicklungsplattform Eclipse[1] von IBM. Das JWT-Projekt beinhaltet verschiedene Softwarewerkzeuge, die eine Plattform für die Entwicklung und das Management von flexiblen und interoperablen Geschäftsprozessen (Arbeitsabläufen) bereitstellen. Die Modellierung der Geschäftsprozesse erfolgt über die grafische Oberfläche des JWT Workflow Editors, der als Plugin in die Eclipse Plattform eingebunden ist.

[1] Siehe http://www.eclipse.org und http://www.eclipse.org/jwt

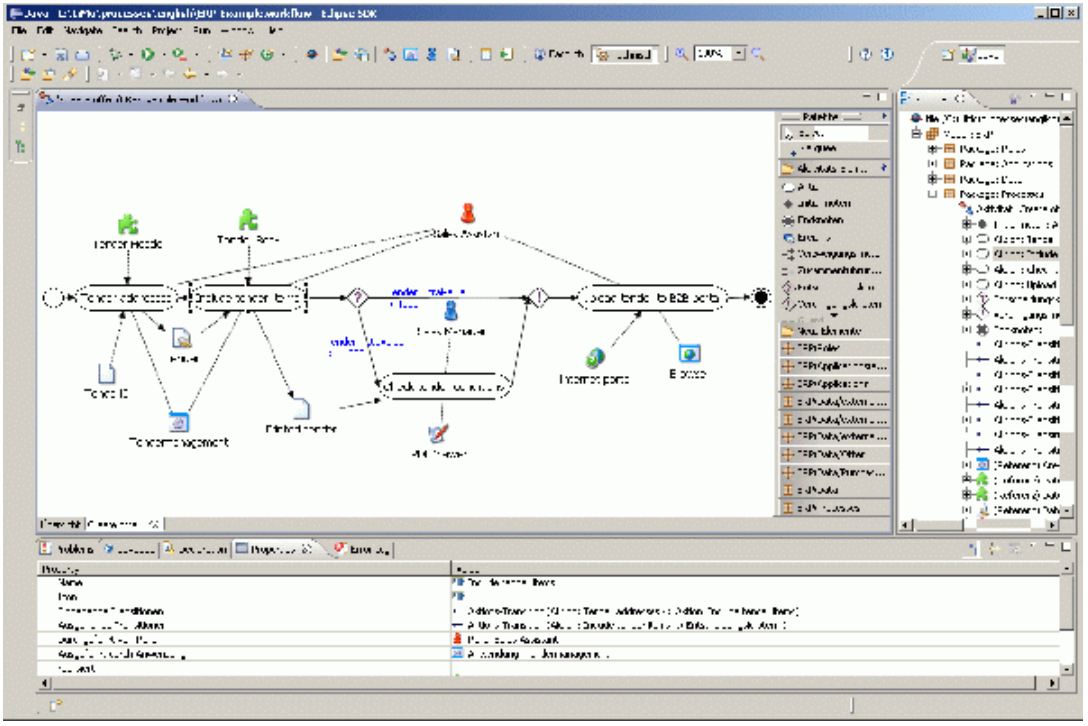


Abbildung 3.2: Workflow Editor des JWT Projekts

Abbildung 3.2 zeigt einen im Workflow Editor geladenen Geschäftsprozess (eine JWT-Aktivität) in der, über das Eclipse GEF-Framework realisierten, graphischen Darstellung. Die weiteren Bestandteile des Workflow Editors umfassen die Palette, über die der Aktivität zusätzliche Elemente hinzugefügt werden können, eine Toolbar mit Schnellzugriff auf die wichtigsten Programmfunktionen, ein Fenster, in dem die Eigenschaften der Modellobjekte bearbeitet werden können und, auf der rechten Seite, eine strukturelle Darstellung des Geschäftsprozesses.

Die Aktivität beginnt mit einem Initialknoten und führt über mehrere verbundene Einzelschritte, in JWT als Aktionen bezeichnet, zu einem finalen Zustand. Dabei kann der Prozessablauf auch bedingte Verzweigungen enthalten, so dass die Ausführung, abhängig vom jeweiligen Systemzustand zur Laufzeit, genau einem der vorgegebenen Pfade folgt. Ebenfalls ist die Modellierung von parallel zu durchlaufenden Ausführungssträngen möglich. Die Aktionen, die die einzelnen Aufgaben des Geschäftsprozesses repräsentieren, benötigen bzw. generieren Ressourcen verschiedenen Typs, z.B. Daten oder Anwendungen. Diese können ebenfalls modelliert und den Aktionen über Verbindungen (gestrichelte Linien) zugewiesen werden.

Die interne Darstellung der Prozesse basiert auf einem Metamodell (ein Ausschnitt ist in Abbildung 3.3 dargestellt), das über das Eclipse Modeling Framework (EMF) in den Workflow Editor eingebunden ist. Dies hat nicht nur den Vorteil, dass die Modellierungssprache leicht erweitert und an individuelle Bedürfnisse angepasst werden kann, sondern stellt zudem sicher, dass die modellierten Prozesse den syntaktischen Vorgaben der Beschreibungssprache genügen, also bezüglich dieser syntaktisch korrekt sind. Daneben

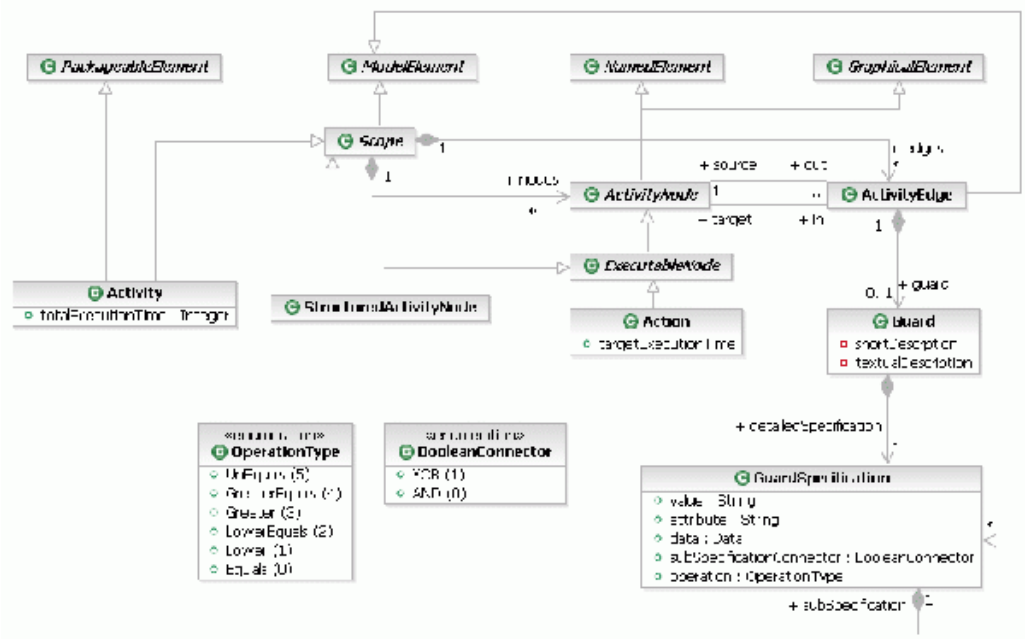


Abbildung 3.3: Das *Processes Package* des JWT-Metamodells definiert unter anderem Aktivitäten (*Activity*), die Aktionen (*Action*) und weitere Arten von Aktivitätsknoten (*ActivityNode*) beinhalten. Aktionen sind über gerichtete Aktivitätskanten (*ActivityEdge*) verbunden, auf denen Bedingungen (*Guards*) festgelegt werden können

können die modellierten Prozesse durch Modelltransformationen leicht in andere Prozessbeschreibungssprachen[2] übersetzt werden.

Ziel der Anwendung der Attributierungstechnik

Eine Attributierung des JWT Metamodells kann verschiedene Zielsetzungen haben: Zum Einen besteht die Möglichkeit zusätzliche semantische Restriktionen auf den Modellen zu formulieren, zum Anderen können aus der Modellstruktur Aussagen zum Verhalten der modellierten Prozesse gewonnen werden.

Ziel ist es, die in dieser Arbeit entwickelte Attributierungstechnik anzuwenden, um die Struktur von in JWT modellierten Aktivitäten zu analysieren. Konkret soll festgestellt werden, welche Aktionen eines Geschäftsprozess, der auch zyklische Abläufe enthalten kann, in jedem Fall durchgeführt werden müssen, und unter welchen Ablaufschritten zu welchem Zeitpunkt eine Auswahl getroffen werden muss. Im Einzelnen ergeben sich folgende Zielsetzungen für das, durch eine Attributierung zu implementierende, Analyseverfahren:

1. Für jede Aktion soll eine Liste der Vorgängeraktionen berechnet werden, deren Ausführung Voraussetzung für das Erreichen dieser Position im Prozessablauf ist.

[2] Zum Beispiel in die Business Process Modeling Notation (BPMN) (vgl. http://www.omg.org/spec/BPMN/1.1/PDF/), die ebenfalls einen OMG-Standard darstellt.

Dadurch können Aussagen über die Situation zu einem bestimmten Zeitpunkt, z.B. bezüglich des Ressourcen- oder des Zeiteinsatzes, getroffen werden, aber auch darüber, wie sich die modellierte Struktur des Prozesses auf dessen konkrete Durchführung auswirkt.

2. Manche Aktionen setzen voraus, dass ein spezifisches Ereignis, unabhängig vom konkreten Prozessablauf, zwingend eingetreten sein muss, bevor die Ausführung beginnen darf. Hier muss es möglich sein, in die Attributierung Restriktionen einbeziehen zu können, die dies sicherstellen.

3. Können über Verzweigungen verschiedene Ablaufpfade beschritten werden, so müssen diese Alternativen in die Betrachtung vollständig mit einbezogen werden.

4. Die beschriebene Auswertung soll nicht nur für einzelne Knoten, sondern auch für den gesamten Prozess erfolgen. Insbesondere kann dadurch anhand der minimalen Durchlaufpfade eine Untergrenze für den Ressourcenverbrauch angesetzt werden.

5. Unter bestimmten Bedingungen kann es sinnvoll sein Aktionen bzw. Teilabläufe in Geschäftsprozessen zu parallelisieren bzw. zu serialisieren um Ressourcen zu schonen oder den Prozessablauf zu beschleunigen. Eine Berechnung der minimalen und alternativen Abläufe ermöglicht es, dieses Optimierungspotential zu erschließen.

Weitere Zielsetzungen, die hier aber nicht weiter diskutiert werden sollen, bestünden beispielsweise in der Implementierung von semantischen Restriktionen, die unter anderem die korrekte Schachtelung von parallelen Abläufen sicherstellen könnten, in der automatischen Generierung von Knotennummerierungen gemäß ihrer Position im Ablauf, oder auch in der Berechnung aller möglichen Prozessabläufe zur Durchführung von Simulationen.

Einschränkungen der Object Constraint Language

Die Formulierung des oben beschriebenen Analyseverfahrens in der Object Constraint Language ist nicht möglich, da über diese keine dynamischen und iterativen Berechnungsalgorithmen implementiert werden können. Tatsächlich kann auf Grund der statischen Form der OCL-Restriktionen nicht einmal eine Menge der besuchten Aktionen geführt werden, da die Navigationen auf der Metastruktur festgelegt sind und damit eine begrenzte Reichweite besitzen.

4 Attributierte Metamodelle

In Abschnitt 2.2 wurde ein gängiges Attributierungskonzept beschrieben, das kontextfreie Grammatiken um semantische Attribute erweitert, wodurch aus der syntaktischen Struktur eines Wortes kontextabhängige Informationen gewonnen und verarbeitet werden können. In 2.4 wurden die Prinzipien der Metamodellierung sowie die Formulierung von Restriktionen durch die Object Constraint Language vorgestellt. Dieses Kapitel kombiniert diese Techniken, um daraus ein Konzept für die Struktur- und Verhaltensanalyse von (UML)-Metamodellen zu entwickeln, das auf einer Anreicherung der Modelle durch semantische Attribute basiert.

Die Übertragung des Attributierungskonzepts auf Metamodelle ist möglich, da kontextfreie Grammatiken und Metamodelle Ähnlichkeiten in ihren grundlegenden Eigenschaften aufweisen. Wörter, die die Sprache einer Grammatik bilden, folgen in ihrem strukturellen Aufbau den vorgegebenen Regeln der Grammatik. Metamodelle definieren eine abstrakte Syntax - formalisiert z.B. als XML-Schemata - die ebenfalls den Aufbau von Instanzen (entsprechend im XMI-Format) der modellierten Sprache beschreibt.[1] Abschnitt 4.1 geht auf die konzeptionellen Parallelen beider Techniken ein, und entwickelt die Basisanforderungen zur Attributierung von Metamodellen.

Die Anreicherung kontextfreier Grammatiken um Attributierungselemente erforderte die Hinterlegung von Attributierungssyntax und -semantik. Dadurch wurde beschrieben, wie Attribute mit grammatikalischen Symbolen zusammenhängen und wie eine gegebene Attributierung zu interpretieren ist. Äquivalent muss die Definition von Metamodellen erweitert werden, so dass diese die Festlegung und Auswertung von Attributierungen unterstützen. Dazu wird in 4.2 zunächst eine abstrakte Syntax vorgestellt, die Attributierungselemente in das Meta-Metamodell integriert. Darauf aufbauend wird eine konkrete Syntax in Form einer graphischen Notation beschrieben, sowie ein Konzept entwickelt, das die Spezifikation von implementierungsunabhängigen Berechnungsvorschriften erlaubt. Abgeschlossen wird die Syntaxdefinition durch eine textuelle Notation, durch die Attributierungen vollständig und unabhängig vom zugrunde liegenden Metamodell definiert und abgelegt werden können.

[1] Die Autoren von [AP04] implementieren beispielsweise - basierend auf den ähnlichen Eigenschaften von formaler Grammatiken und Metamodellen - Transformationen zwischen diesen Darstellungsformen.

Im nächsten Schritt wird den Sprachelementen eine Semantik zugeordnet, die festlegt welche Anforderungen an eine gültige Attributierung zu stellen sind und welche Bedeutung die einzelnen Attributierungsfragmente besitzten. Abschnitt 4.3 schafft damit die Voraussetzung für eine Abhängigkeitsanalyse der Attributierung, welche wiederum die Basis für deren Auswertung bildet. Abhängigkeitsanalyse und Auswertung werden in Kapitel 5 behandelt.

4.1 Anforderungen und Eigenschaften

Metamodelle und Grammatiken sind verwandte Konzepte in dem Sinne, als dass beide Techniken Methoden zur Modellierung von Sprachen definieren, deren Elemente einer vorgegebenen Struktur entsprechen. Um eine Metamodellierungssprache zu entwerfen, die eine Attributierung erlaubt, müssen zunächst einige Vereinbarungen über die Eigenschaften getroffen werden, die diese Sprache aufweisen muss.

Abschnitt 4.1.1 arbeitet Gemeinsamkeiten von Grammatiken und Metamodellen heraus, indem beide auf ihren jeweiligen Definitions- und Anwendungsebenen verglichen werden. In 4.1.2 werden die Anforderungen definiert, die im Kontext dieser Arbeit an die Entwicklung einer Metasprache gestellt werden. Zuletzt werden in 4.1.3 die zentralen Eigenschaften einer Attributierung auf Metamodellen der Object Constraint Language gegenübergestellt.

4.1.1 Vergleich der Abstraktionsebenen von attributierten Grammatiken und Metamodellen

Kontextfreie Grammatiken und Metamodellierung sind Techniken, deren Einsatz die Betrachtung unterschiedlicher Abstraktionsschichten voraussetzt. In diesem Abschnitt werden die grundlegenden Gemeinsamkeiten, die beide Ansätze verbinden, herausgearbeitet um eine Basis für eine Übertragung des Attributierungskonzepts auf Metamodelle zu schaffen.

Wie in 2.4.1 bereits angedeutet, ist im Bereich der Metamodellierung die Unterscheidung zwischen Modellierungs- und Anwendungsebene innerhalb einer gegebenen Metahierarchie vom Anwendungskontext abhängig. Ein Softwaretechniker, der die Struktur eines Programms in einem UML-Klassendiagramm modelliert, erstellt damit ein Metamodell der zur Laufzeit instantiierten Objekte. Die Bezeichnungen Meta-/Instanzebene können in der Abstraktionshierarchie aber auch verschoben werden, so dass beispielsweise das Klassendiagramm selbst als Instanz der Sprache der UML-Klassendiagramme interpretiert wird.

Die Klassifikation der Abstraktionsschichten kontextfreier Grammatiken lässt sich ähnlich vornehmen. Üblicherweise wird auf der Metaebene die Struktur einer Sprache durch

Angabe der Grammatik „modelliert“. Die Instanzebene umfasst dann alle Wörter dieser formalen Sprache. Im konkreten Anwendungskontext des Übersetzerbaus, in dem die Sprache durch alle gültigen Programme gebildet wird, kann eine konkrete Ausführung von Programmcode wiederum auch als Instanz des jeweiligen Programms aufgefasst werden.

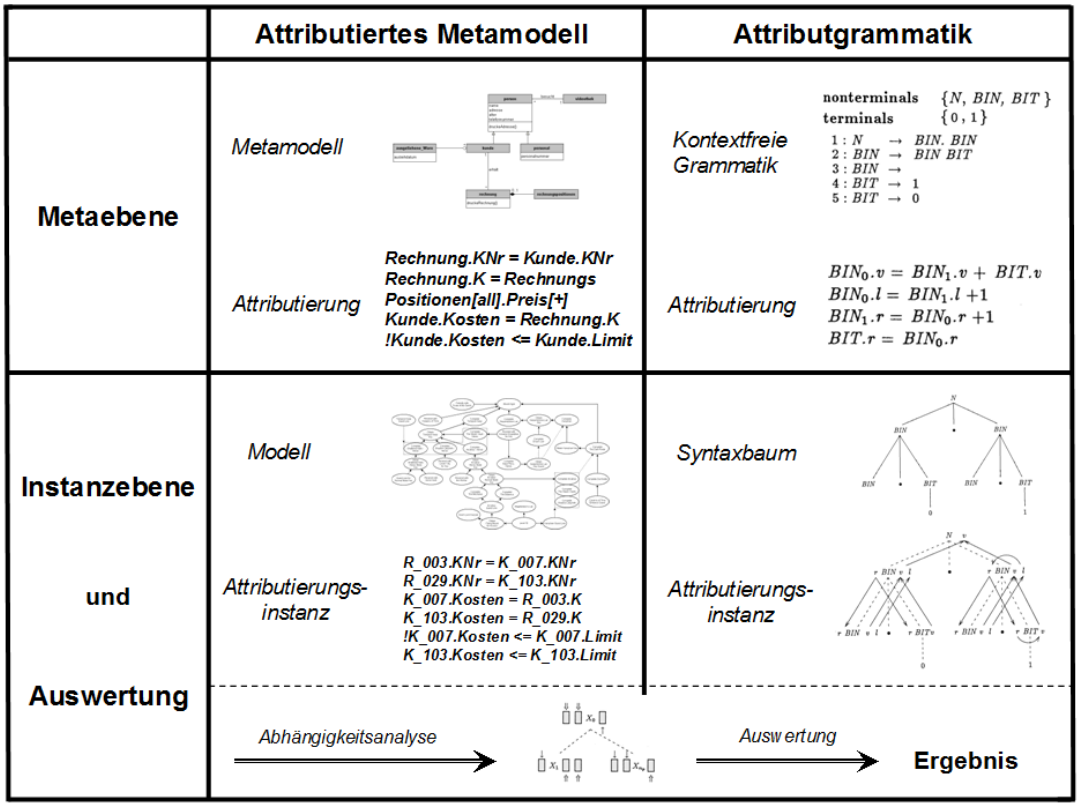


Abbildung 4.1: Gegenüberstellung attributierter Metamodelle und Attributgrammatiken anhand ihrer Abstraktionsschichten

Abbildung 4.1 zeigt eine Gegenüberstellung konkreter Meta- und Instanzebenen attributierter Metamodelle bzw. Grammatiken. Die Anordnung wurde dabei so gewählt, dass die Definition formaler Sprachen (durch Angabe von Attributgrammatiken) und die Erstellung attributierter Metamodelle die Metaebene bilden. Diese Einteilung basiert auf der Überlegung, dass die Metaebene die syntaktische Struktur sowie die statische Semantik der Zielsprache bestimmt.

Die Auswertung der Attributierung erfolgt jeweils auf Instanzebene, also auf einem Wort der Sprache bzw. einem Modell. Dies kann bei einer Attributgrammatik beispielsweise ein Programm sein bzw. dessen interne Repräsentation in Form eines Syntaxbaums. Wie in Kapitel 2 beschrieben, kann als Ergebnis dieser Auswertung die Überprüfung der Einhaltung semantischer Vorgaben oder auch die Generierung neuer Informationen stehen.

In der Abbildung nicht enthalten ist die Abstraktionsschicht, die den Aufbau der Metaebene selbst festlegt. Bei Attributgrammatiken handelt es sich dabei um die formale Definition kontextfreier Grammatiken, sowie deren Verknüpfung mit dem Attributierungskonzept (vgl. 2.2.1). Struktur und Semantik von Metamodellen werden dagegen üblicherweise durch die (rekursive) Angabe einer weiteren Metaebene beschrieben. Dieses Meta-Metamodell muss folglich neben der Metamodellierungssprache auch die Attributierungselemente und deren Verbindung zu den Sprachausdrücken definieren. Die Anforderungen an dieses Meta-Metamodell werden in 4.1.2 beschrieben und in 4.2.1 für eine konkrete Metasprache implementiert.

Zusammenfassend kann festgehalten werden, dass die Auswertung der Attributierung bei beiden Konzepten für Instanzen erfolgt, deren Struktur durch eine Metaebene festgelegt wird. Diese bildet durch Verknüpfung der Sprachelemente mit einer Attributierung den Grundstein für die semantische Analyse und baut selbst auf einer übergeordneten Attributierungssprache auf.

4.1.2 Anforderungen an die Struktur der Metasprache

Die grundlegende Eigenschaft, die eine Metamodellierungssprache (d.h. das Meta-Metamodell) besitzen sollte, ist die Implementierung einer einfachen, aber dennoch flexiblen, Struktur. Diese muss einerseits ausdrucksmächtig und konkret genug sein, um den Einsatz in verschiedenen Anwendungsdomänen zu gewährleisten, in ihrer Grundkonzeption aber dennoch abstrakt, um zukünftige domänenspezifische Adaptionen zu ermöglichen.

Um eigene Metasprachen zu entwickeln, bietet es sich an, auf UML bzw. MOF zurückzugreifen, die für diesen Zweck eigene Methoden definieren. Da im Rahmen dieser Arbeit nicht alle Modellierungselemente, die die Meta Object Facility zur Verfügung stellt, in das Attributierungskonzept aufgenommen werden können, ist eine Beschränkung auf deren wesentliche Elemente erforderlich. Die Auswahl ist dabei möglichst so zu treffen, dass die Ausdrucksmächtigkeit der Metasprache weitgehend erhalten bleibt. In Abbildung 2.14 wurde bereits der grundlegende Aufbau des UML M3-Modells vorgestellt. Eine Orientierung an dieser Struktur würde sowohl die Anforderungen bezüglich der geforderten Ausdrucksmächtigkeit erfüllen, als auch die Kompatibilität bereitstellen, die notwendig ist um das Attributierungskonzept zukünftig auf die gesamte MOF ausdehnen zu können.

Die zentralen Konzepte, die durch die Unified Modeling Language Infrastructure Library (vgl. [Gro07a, S. 27ff]) definiert werden, und die im Weiteren die Basis für attributierte Metamodelle bilden, sind:

Klasse
: Alle domänenspezifischen Elemente sind Instanzen dieses Konzepts.

Klassenattribut
Eine Eigenschaft, die eine Klasse näher charakterisiert.

Assoziation
Stellt einen semantischen Zusammenhang zwischen Klassen her, der durch Rollennamen und Multiplizitäten näher bestimmt werden kann.

Generalisierung
Ein Vererbungskonzept, durch das Eigenschaften der Elternklasse (Supertyp) auf Kinderklassen (Subtypen) übergehen.

Durch diese Sprachstruktur ergeben sich weitere Vorteile:

- Da in der Softwareentwicklung Metamodellierung auf Basis von UML-Techniken weit verbreitet ist, erleichtert der Einsatz dieser Konzepte das Verständnis für Personen, die bereits mit UML vertraut sind.
- Der gewählte Sprachausschnitt ist im Gegensatz zum vollen UML Sprachumfang zwar eingeschränkt, durch die Unterstützung der wichtigsten Konzepte lässt sich das Attributierungskonzept aber bereits auf viele bestehende Metamodelle anwenden.
- Eine Einschränkung auf grundlegende Sprachelemente ermöglicht es, bei der in den nächsten Abschnitten folgenden Definition einer Attributierung auf Metamodellen den Fokus auf die wesentlichen Herausforderungen zu richten.
- Das Vorhandensein elementarer, abstrakter Konzepte bildet die Basis für zukünftige Erweiterungen. So kann die Metasprache beispielsweise domänenspezifisch in Richtung UML Klassendiagramm erweitert werden, indem, äquivalent zu Klassenattributen, ein Konzept für Klassenoperationen definiert wird.
- Da UML die gewählten Elemente der Infrastructure Library sowohl auf M3- wie auch auf M2-Ebene importiert, können nicht nur attributierte Metamodelle der Schicht M2, sondern auch einfache attributierte M1-Modelle entworfen werden. Die Auswertung der Attributierung erfolgt dann äquivalent auf Ebene M1 bzw. M0.
- UML bzw. MOF definieren für die verwendeten Sprachelemente bereits Semantik und Notation.

Die beschriebene Struktur gilt es jetzt um Elemente zu erweitern, die eine geeignete Repräsentation einer Attributierung auf Metamodellen erlauben. Dabei handelt es sich gewissermaßen um Meta-Attribute, deren Semantik im Zusammenspiel mit den ursprünglichen Sprachelementen definiert wird.

4.1.3 Vergleich des Attributierungskonzepts mit OCL

Neben den grundlegenden Eigenschaften attributierter Grammatiken weist das Attributierungskonzept auf Metamodellen auch Eigenschaften auf, die an die in Abschnitt 2.4.2 vorgestellte Object Constraint Language erinnern. Da sich die Gebiete, in denen attributierte Metamodelle eingesetzt werden können, zumindest teilweise mit typischen Anwendungsfällen von OCL-Restriktionen überschneiden, ist eine Betrachtung der Gemeinsamkeiten sowie der Unterschiede beider Techniken angebracht.

Gemeinsame Eigenschaften

- Beide Konzepte operieren auf der syntaktischen Struktur von Metamodellen, die mit semantischen Restriktionen belegt werden kann.
- Neben der syntaktischen Struktur können auch weitere Laufzeiteigenschaften der Modelle, wie z.B. Werte von Klassenattributen, in die Formulierung der Constraints einfließen.
- Durch Implementierung auf MOF-Ebene können sie grundsätzlich für beliebige UML Metamodelle eingesetzt werden.
- Beide Verfahren setzen auf Navigationstechniken, die sich an den modellierten Assoziationen orientieren, um den Zugriff auf konkrete Instanzmengen zu ermöglichen. Die Multiplizitäten der Assoziationsenden erzwingen dabei für diese Anfragen eine Mengensemantik. Neben dem qualifizierten Zugriff können auch alle Instanzen eines Typs angesprochen werden.
- Sowohl OCL-Ausdrücke wie auch Metamodellattributierungen lassen sich innerhalb des zugrunde liegenden Metamodells oder extern definieren.
- Der Auswertungsprozess ist atomar, das bedeutet das zugrunde liegende Modell ist während der Auswertung konstant. Direkte Manipulationen des Ausgangsmodells durch den Auswertungsprozess sind damit nicht möglich.
- Die Ausdrucksmächtigkeit einer entsprechend formulierten Attributierungstechnik beinhaltet den Sprachumfang der Object Constraint Language als Teilmenge, da die in einem OCL-Constraint enthaltenen Navigationen und Bedingungen in entsprechende Attributadressierungen und Spezifikationen (siehe Abschnitt 4.2.3) transformiert werden können.

Unterschiede

- Das Implementierungskonzept attributierter Metamodelle beschränkt sich in der in dieser Arbeit besprochenen ersten Ausbaustufe auf Restriktionen. Guards, Pre-/Postconditions und weitere Elemente der Object Constraint Language werden derzeit (noch) nicht unterstützt.

- OCL-Ausdrücke sind an die Metamodellstruktur gebunden, wodurch die Analyse von Informationsflüssen im Modell nicht, oder nur rudimentär, möglich ist. Attributierungen sind hier viel eher geeignet Informationen aus und über die Struktur eines Metamodells zu gewinnen und zu aggregieren als die OCL-typischen statischen Navigationen.
- Die Anwendungsgebiete von Attributierungen sind breiter gefächert als die der Object Constraint Language. Neben der Eigenschaft auch komplexe Semantiken, die auf der Betrachtung dynamischer Informationsflüsse aufbauen, formal hinterlegen zu können, unterstützen Attributierungen die Durchführung von Datenflussanalysen und den syntaktisch gesteuerten Prozeduraufruf. Nicht zuletzt ist dadurch beispielsweise auch die Durchführung von Modelltransformationen denkbar.
- Wie in Abschnitt 3.1 bereits angedeutet, sind OCL-Restriktionen sehr anfällig gegenüber Änderungen an der Struktur des Metamodells, während der Informationsfluss bei Attributierungen oft nur lokal angepasst werden muss.
- Ebenfalls wurde demonstriert, wie in OCL formalisierte semantische Einschränkungen durch zahlreiche Navigationen und Selektionen zu komplexen und teilweise redundanten Ausdrücken führen, da Zwischenergebnisse nicht weiterverwendbar sind, während die Nutzung bereits berechneter Informationen gerade das Kernelement der Attributierungstechnik bildet.

4.2 Syntax

Die Syntax zur Attributierung von Metamodellen wird in mehreren Schritten entwickelt:

- Zu Beginn muss eine Sprache definiert werden, durch die Metamodelle mit einer Attributierung versehen werden können. Da Metamodelle selbst Instanzen eines Meta-Metamodells sind, wird ein solches, das die Anforderungen aus 4.1 erfüllt, in 4.2.1 beschrieben.
- In 4.2.2 wird eine konkrete Syntax für die syntaktischen Elemente der Metasprache in Form einer graphischen Notation vorgestellt.
- Semantische Regeln entsprechen Funktionen, deren Ergebnis sich aus den übergebenen Argumenten berechnet. Wie für diese Attributierungsregeln Funktionsdefinitionen hinterlegt werden können, beschreibt die Spezifikationssyntax aus 4.2.3.
- Abschließend wird in 4.2.4 erörtert, wie eine Attributierung in textueller Form vollständig repräsentiert werden kann.

4.2.1 Abstrakte Syntax

Die abstrakte Syntax der Metasprache wird durch ein Meta-Metamodell festgelegt, das die Basis für die Entwicklung einer Attributierungstechnik auf Metamodellen darstellt. Um dies zu erreichen, werden einer Modellierungssprache, die sich auf die Anforderungen aus 4.1.2 stützt, Elemente hinzugefügt, welche die Modellierung von semantischen Attributen ermöglichen. Das Ergebnis ist die syntaktische Struktur einer (Meta-) Modellierungssprache, die die Erstellung von Metamodellen und deren Anreicherung durch Attributierungen unterstützt.

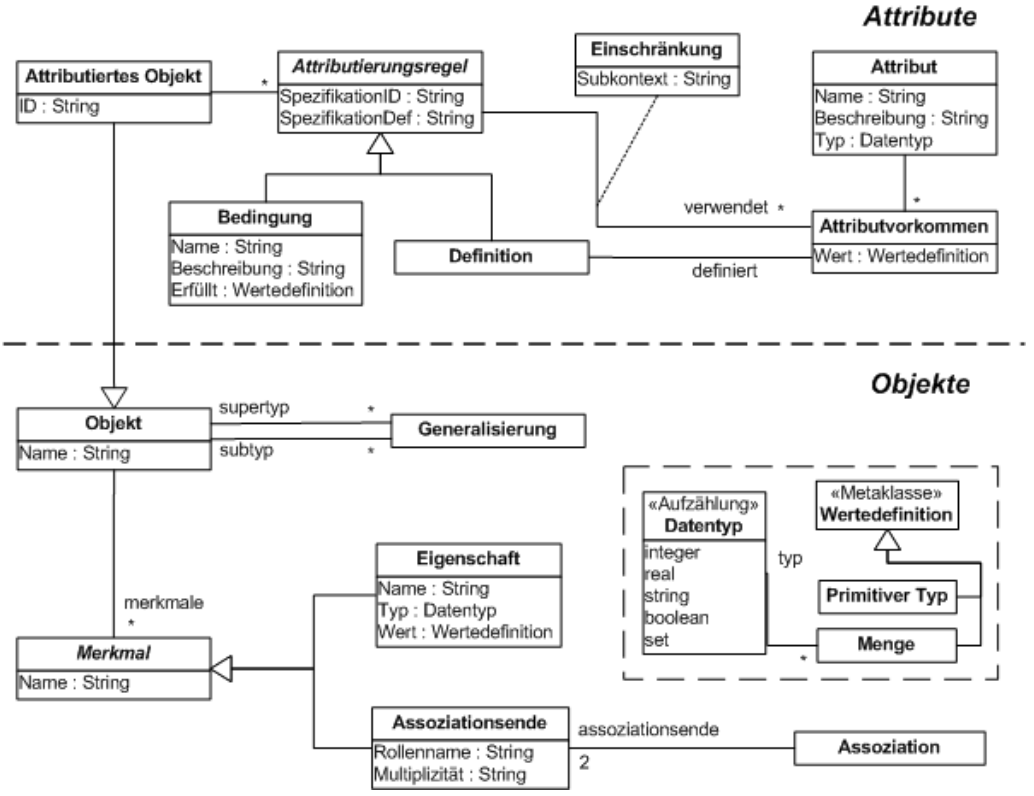


Abbildung 4.2: Meta-Metamodell, das eine MOF-ähnliche Sprache implementiert und deren Konstrukte um eine Attributierungsstruktur erweitert (nicht angegebene Multiplizitäten sind 1)

Der Einsatz einer MOF-ähnlichen Sprachstruktur erlaubt neben der Anlehnung an deren semantische Konventionen auch die Verwendung der im UML-Umfeld gebräuchlichen Klassendiagrammnotation. Abbildung 4.2 zeigt das Meta-Metamodell in dieser Darstellungsform, wobei die Sprachelemente in zwei jeweils logisch zusammenhängende Bereiche aufgeteilt sind.

Die untere Hälfte modelliert Objekte sowie deren Eigenschaften und Beziehungen in Anlehnung an die Basisstruktur der Meta Object Facility (siehe Abbildung 2.14), im oberen

Teil wird die Struktur der Attributierungskonstrukte definiert. Die Spezialisierung von `Objekt` durch den Typ `Attributiertes Objekt`, dessen Instanzen mit einem Satz von `Attributierungsregeln`[2] versehen werden können, bildet die Schnittstelle zwischen beiden Bereichen.

Objekte und Relationen

Die Objektumgebung des Meta-Metamodells definiert eine Sprache zur Modellierung der Anwendungsdomäne. Die Bedeutung der Elemente entspricht weitestgehend der ihrer jeweiligen UML-Gegenstücke und wird im Folgenden erläutert:

Objekt

Die Semantik eines `Objekts` orientiert sich an der einer UML-Klasse[3]. Es steht für Konzepte der Anwendungsdomäne und ist damit das zentrale Element für die Modellierung von Klassen, Aktionen, Akteuren etc. Objektinstanzen werden durch einen, innerhalb der Gruppe aller `Objekte` eindeutigen, `Namen` identifiziert. Ein `Objekt` kann über eine beliebige Anzahl an `Merkmalen` der Ausprägung `Eigenschaft` bzw. `Assoziationsende` verfügen. Über die `Generalisierung` wird ein Vererbungsmechanismus implementiert, der `Merkmale` eines (Objekt-) Supertyps auf den (Objekt-) Subtyp überträgt.

Merkmal

Ein `Merkmal` ist ein abstraktes Modellelement, das genau einem `Objekt` zugeordnet ist und dieses (in seinen Ausprägungen `Assoziationsende` bzw. `Eigenschaft`) näher charakterisiert. Der Typ `Merkmal` orientiert sich an den UML-Features bzw. Properties.

Eigenschaft

Eine `Eigenschaft` ist ein spezielles `Merkmal`, über das `Objekte` zusätzliche Informationen aufnehmen können. Konkret handelt es sich um ein Datenfeld, vergleichbar einem UML-Klassenattribut, das durch einen `Namen` definiert wird und einen Wert vom vorgegeben `Datentyp` aufnehmen kann.

Assoziationsende

Ein `Assoziationsende` ist die zweite Spezialisierungsform des Typs `Merkmal` und definiert den Endpunkt jeweils genau einer `Assoziation`. `Multiplizität` und `Rollenname` bestimmen die Eigenschaften der jeweiligen Assoziationsseite. Die Semantik entspricht dabei den üblichen UML Konventionen.

---

[2] Dieser Mechanismus erlaubt die flexible Erweiterung des Attributierungskonzepts auf beliebige Elemente einer Metasprache, indem für diese jeweils Spezialisierungen angelegt werden, die die Verbindung mit Attributierungsregeln ermöglichen.

[3] Die Bezeichnung „Objekt“ (statt „Klasse“) leitet sich vom Anspruch ab, beliebige Konzepte beliebiger Domänen zu repräsentieren.

Assoziation

Eine `Assoziation` entspricht einer UML-Association, das bedeutet sie ist eine Verbindung zwischen genau zwei `Assoziationsenden`, wodurch ein semantischer Zusammenhang zwischen den an der Relation beteiligten `Objekten` bzw. deren Instanzen hergestellt wird. Zur Laufzeit wird sie für die vorhandenen Instanzen unter Berücksichtigung der definierten `Multiplizitäten` instantiiert.

Generalisierung

Eine `Generalisierung` ist eine Relation zwischen zwei `Objekten` in der Rolle eines Super- und eines Subtyps. Der `subtyp` spezialisiert den `supertyp`, indem er dessen `Merkmale`[4] erbt und um eigene `Merkmale` erweitert. Dabei ist auch Mehrfachvererbung erlaubt. Das Generalisierungskonzept wurde von der superClass-Relation der UML übernommen.

Datentyp

Der `Datentyp` legt den zulässigen Wertbereich für Datenfelder fest. Die Grunddatentypen sind Integer, Real, String, Boolean und Set[5]. Der `Datentyp` Set bezeichnet eine Menge von Werten eines primitiven `Datentyps`.

Wertedefinition

Eine `Wertedefinition` (UML: Value Specification) nimmt den Laufzeitwert eines bestimmten `Datentyps` auf. Dabei kann es sich um die Instanz eines primitiven Typs handeln, z.B. um eine Integerzahl oder einen Text, oder aber um eine Menge von Instanzen eines festgelegten primitiven Typs.

Attributierung

Die definierte „klassische“ Metasprache gilt es um eine Attributierungsstruktur zu erweitern, deren Schnittstelle das `Attributierte Objekt` bildet. Dieser Entscheidung liegt folgende Überlegung zugrunde: In einer attributierten Grammatik werden semantische Regeln mit Produktionen assoziiert. Produktionsvorkommen bilden die Knoten des Ableitungsbaums und damit die Basis für die Auswertung der Attributierung. Die Knoten in einem Modell sind dagegen Instanzen des Typs `Objekt`, das sich damit als Träger semantischer Attributierungsinformationen qualifiziert. Diese bestehen aus einem Satz von `Attributierungsregeln`, die festlegen wie sich der Ergebniswert eines `Attributvorkommens` aus den Werten anderer `Attributvorkommen` berechnet. Die Elemente der Attributierungsstruktur sind im Einzelnen:

Attributiertes Objekt

Ein `Attributiertes Objekt` ist eine Spezialisierung eines „klassischen“ `Objekts`, dem durch die Verknüpfung mit `Attributierungsregeln` eine Attri-

[4] Die Vererbungssemantik der Attributierung, d.h. die Weiterleitung von Attributwerten innerhalb der Vererbungshierarchie, verlangt eine explizite Definition (siehe Abschnitt 4.3).

[5] Bis auf Set sind die Datentypen identisch mit den Basisdatentypen der OCL (vgl. [Gro06b, S. 10]).

butierung zuordnet werden kann. Die `ID` dient als Identifikation und muss daher eindeutig[6] für die Menge der `Attributierten Objekte` sein.

Attributierungsregel

Eine `Attributierungsregel` ist eine semantische Regel, die jeweils genau einem `Attributierten Objekt` zugeordnet ist. Es handelt sich um ein abstraktes Konzept, das in den Ausprägungen `Definition` und `Bedingung` auftritt. Wie bei Attributgrammatiken bestehen die Argumente dieser Regeln aus `Attributvorkommen`, die hier über die Relation `verwendet` assoziiert sind. Über die `Einschränkung` auf einen `Subkontext` können Restriktionen auf der Menge der als Argumente zu übergebenden Attributinstanzen festgelegt werden. Die Funktion, die auf die Argumente der Regel angewendet wird, wird durch die Klassenattribute `SpezifikationID` und `SpezifikationDef` festgelegt (siehe Abschnitt 4.2.3).

Definition

Eine `Definition` ist eine spezielle Form einer `Attributierungsregel`. Ihre Aufgabe ist es, einen Ergebniswert für das `Attributvorkommen`, das über die Relation `definiert` mit der `Definition` assoziiert ist, zu berechnen.

Bedingung

Eine `Bedingung` definiert kein `Attributvorkommen`, kann aber Laufzeitwerte von Attributexemplaren abfragen um daraus ein Ergebnis vom Typ Boolean zu generieren. Das Klassenattribut `Erfüllt` ist true genau dann, wenn die durch die Spezifikation festgelegte Funktion für die übergebenen Argumente den Wert true zurückliefert, und false, wenn die Anwendung der Funktion false ergibt. Eine `Bedingung` wird durch ihren `Namen` identifiziert, der innerhalb des definierenden `Attributierten Objekts` eindeutig sein muss.

Attribut

Ein `Attribut` definiert `Eigenschaften`, die eine Gruppe von `Attributvorkommen` charakterisieren. Diese Eigenschaften bestehen aus einem (eindeutigen) `Namen`, einer textuellen `Beschreibung` sowie einem `Datentyp`. Zu beachten ist dabei, dass `Attribute` hier als eine Art Typdefinition der `Attributvorkommen` auftreten. Verschiedene `Attributvorkommen`, die durch dasselbe `Attribut` charakterisiert werden, sind üblicherweise Träger derselben Art von Information, im Sinne der Auswertungssemantik aber voneinander unabhängig.

Attributvorkommen

`Attributvorkommen` repräsentieren Argumente bzw. das Ergebnis der jeweiligen `Attributierungsregeln`. Handelt es sich bei der Regel um eine `Definition`, so wird dem definierten Vorkommen ein Wert zugeordnet, wobei jedes `Attributvorkommen` genau einmal definiert werden muss. `Attributvorkommen`,

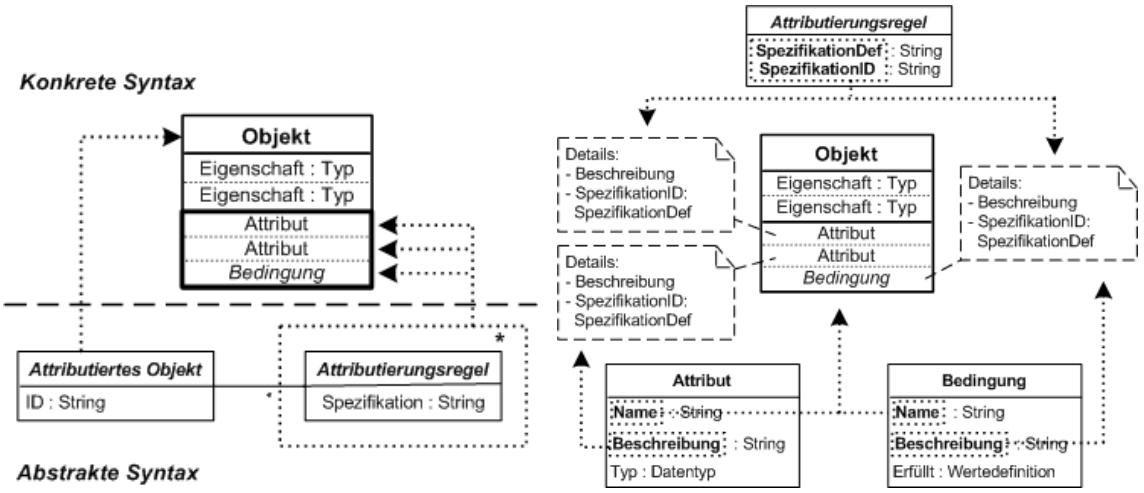
[6] Da der Objektname in der gegebenen Metasprache eindeutig sein muss, bietet es sich an, diesen automatisch als ID des Attributierten Objekts zu übernehmen.

die in einer Regel verwendet werden, stehen der Berechnungsvorschrift dabei als Eingabe zur Verfügung. Ein `Attributvorkommen` erhält durch eine Assoziation mit einem `Attribut` einen `Namen`, eine `Beschreibung` und einen `Datentyp` zugeordnet, wobei sich mehrere `Attributvorkommen` dieselbe Charakteristik teilen können.

4.2.2 Graphische Notation

Der Modellierungsvorgang verlangt nach einer für diesen Zweck optimierten Darstellung der Sprachkonstrukte: Der konkreten Syntax. Dieser Abschnitt erweitert die aus UML bekannte graphische Klassendiagrammnotation um eine geeignete Repräsentation der Attributierungselemente.

Darstellung Attributierter Objekte



(a) Metaobjekte in konkreter Syntax (b) Erweiterte konkrete Syntax

Abbildung 4.3: Konkrete Syntax für ein Attributiertes Objekt

Abbildung 4.3(a) zeigt die, an die Darstellung von UML-Klassen (vgl. [Gro07b, S. 25ff]) angelehnte, graphische Notation Attributierter Objekte. Die assoziierten Attributierungsregeln[7] werden ähnlich Klassenattributen im Objekt eingetragen[8]. Zur visuellen

[7] Es handelt sich dabei streng genommen um die Attributvorkommen, die durch die Attributierungsregeln im Kontext des jeweiligen Objekts definiert werden.

[8] Optional wäre denkbar, die Attributvorkommen je nach zugeordnetem Attributtyp einzufärben, um semantisch zusammenhängende Informationen auf den ersten Blick erkennbar zu machen.

Unterscheidung von Definitions- und Bedingungsregeln, wird für letztere eine kursive Schrift gewählt. Abbildung 4.3(b) zeigt eine erweiterte Darstellung, in der über Kommentarboxen zusätzliche Informationen eingeblendet sind. Die Beschriftung der Definitionsregeln wird vom Attributtyp des durch die Regel definierten Attributvorkommens ableitet, für Bedingungen wird deren Name als Beschriftung übernommen. Die optional einblendbaren Kommentare zeigen die Beschreibung des Attributs bzw. der Bedingung, sowie die SpezifikationID aus dem Supertyp Attributierungsregel.

Darstellung von Attributabhängigkeiten im Metamodell

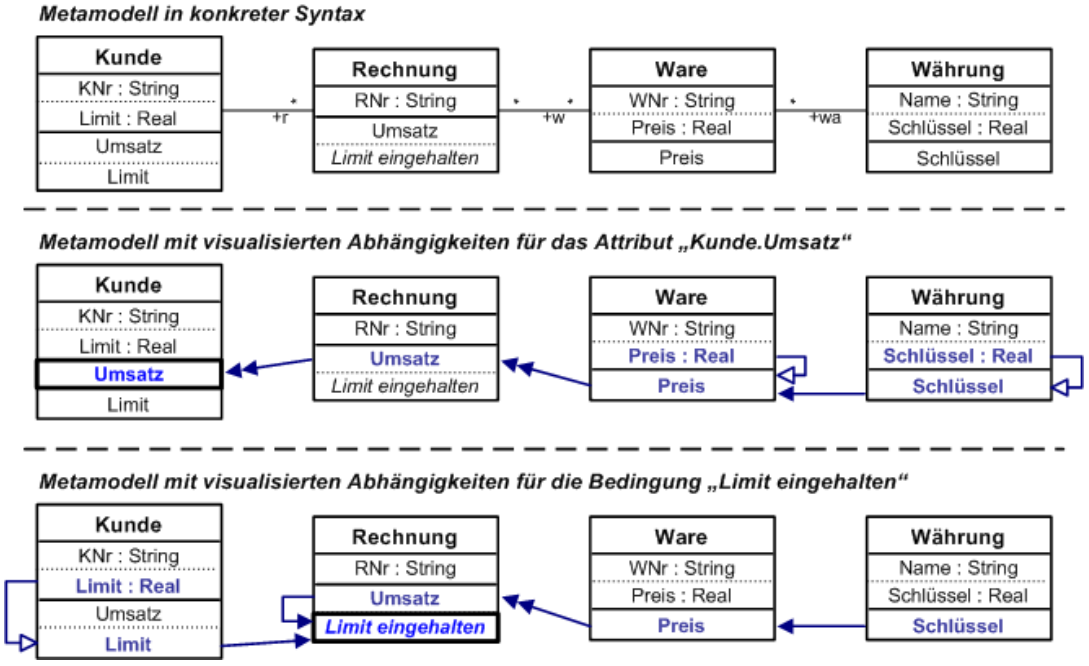


Abbildung 4.4: Darstellung von Abhängigkeiten zwischen Attributvorkommen auf Metamodellebene (im Beispiel aus 3.1)

Abbildung 4.4 demonstriert anhand des, um eine Attributierung erweiterten, Beispiels aus Abschnitt 3.1 die Visualisierung der Abhängigkeiten zwischen Attributvorkommen. Um den Gesamtumsatz des Kunden anzuzeigen, leitet die Attributierung den durch den *Währungsschlüssel* modifizierten *Preis* bis zum Attribut *Umsatz* im Objekt *Kunde*. Die Einhaltung des Rechnungslimits wird durch die Bedingung *Limit prüfen* mit Hilfe der Argumente *Umsatz* und *Limit* sichergestellt.

Die beiden unteren Darstellungen zeigen für das Attribut *Umsatz* am Objekt *Kunde* bzw. die Bedingung *Limit eingehalten* am Objekt *Rechnung* den Informationsfluss, der durch die

Abhängigkeiten zwischen den betroffenen Attributvorkommen generiert wird. Ein einfacher Pfeil symbolisiert die Weiterleitung genau eines Attributwerts, wohingegen ein Doppelpfeil anzeigt, dass an dieser Stelle, bedingt durch die Multiplizität der Assoziation, die Werte mehrerer Attributinstanzen übertragen werden können. Stammt der verwendete Wert nicht von einem Attribut sondern von einer Klasseneigenschaft, so ist die Pfeilspitze nicht gefüllt[9]. Der Subkontext, der die Abhängigkeit einer Attributierungsregel von einem Attributvorkommen näher spezifiziert, kann optional als Pfeilbeschriftung angezeigt werden.

Darstellung von Objektinstanzen

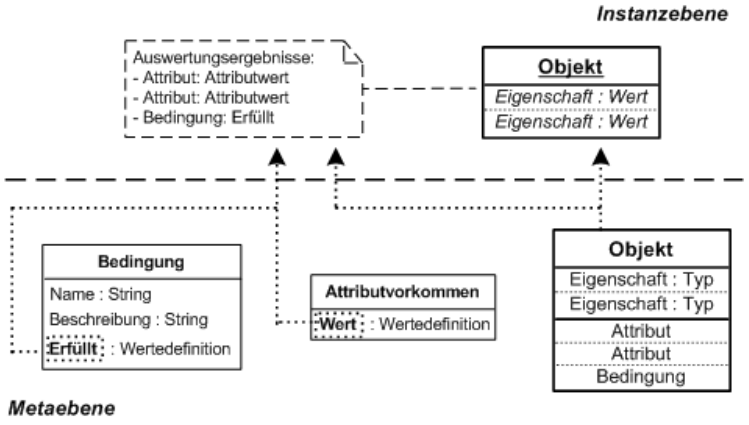


Abbildung 4.5: Darstellung Attributierter Objekte auf Instanzebene

Der Schritt zur Instanzebene wird in 4.5 vollzogen. Die Ergebnisse, die durch einen Attributauswerter für die Attributinstanzen berechnet wurden, werden durch Kommentare an den Laufzeitobjekten (vgl. UML-InstanceSpecification, [Gro07b, S. 211ff]) angezeigt. Diese bestehen aus einer Liste von (Attributname, Attributwert) bzw. (Bedingungsname, Erfüllt-Status) Einträgen.

4.2.3 Spezifikationssyntax

Bisher wurde offen gelassen, wie Attributierungsregeln mit Berechnungsvorschriften, hier als Spezifikationen bezeichnet, versehen werden können. Ein attributiertes Metamo-

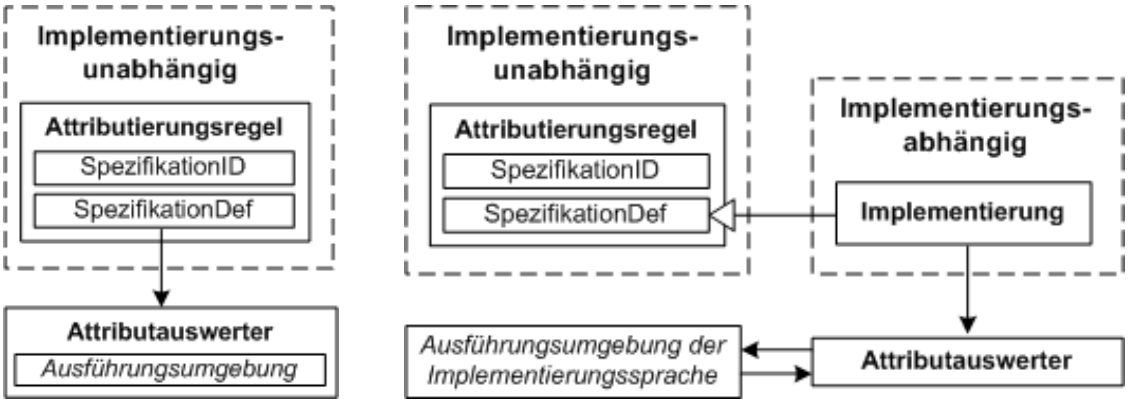
[9] Diese Darstellung hat nur Symbolcharakter, da Objekteigenschaften als statische Werte keine echten Abhängigkeiten erzeugen und damit bei Definition und Auswertung einer Attributierung nicht gesondert betrachtet werden müssen. Einer zukünftigen Erweiterung des Attributierungskonzepts steht es aber offen, auch Abhängigkeiten von Konstanten explizit zu machen.

dell ohne derartige Berechnungsvorgaben ist nicht als vollständig anzusehen, da zwar der Informationsfluss zwischen den Attributvorkommen ersichtlich ist, nicht aber die Art und Weise wie sich deren Ergebniswerte berechnen. Dieser Abschnitt schließt diese Lücke und demonstriert eine exemplarische Umsetzung des Spezifikationskonzepts. Dabei wird auch die Schnittstelle betrachtet, die einem Attributauswerter den Zugriff auf die Berechnungsvorschriften erlaubt.

Aus mathematischer Sicht handelt es sich bei semantischen Regeln einer Attributierung um Funktionen, die eine Anzahl von Eingabewerten auf genau einen Ausgabewert abbilden. Zur Formulierung dieser Funktionen wird bei Attributgrammatiken häufig eine einfache Syntax verwendet, die auf den Produktionen der Grammatik aufsetzt (siehe Abschnitt 2.2.2). Da Attributierungsregeln auf Metamodellen derselben Semantik folgen, d.h. die den Regeln zugeordneten Attributvorkommen bilden deren Eingabe- bzw. Ausgabevariablen, muss jede existierende *Attributierungsregel* mit einem Funktionsrumpf verbunden werden, der die Berechnung des Ergebniswerts abhängig von den Funktionsargumenten beschreibt. Die abstrakte Syntax definiert für diesen Zweck die Klassenattribute *SpezifikationID* und *SpezifikationDef*.

Anforderungen an das Spezifikationskonzept

Einsatzkontext der Metamodellierung ist meist die implementierungsunabhängige Darstellung von Anwendungsdomänen. Eine Festlegung auf eine proprietäre Implementierungssprache für Spezifikationen würde diese Eigenschaft zerstören.



(a) Die Spezifikationsdefinition ist in einem vom Attributauswerter interpretierbaren, plattformunabhängigen Format gegeben

(b) Die Spezifikationsdefinition liegt in einem proprietären Format vor, auf das über eine implementierungsunabhängige Schnittstelle zugegriffen werden kann

Abbildung 4.6: Zwei Ansätze für eine implementierungsunabhängige Spezifikationsarchitektur

Um Plattformunabhängigkeit zu garantieren, stehen folgende Ansätze zur Wahl[10]:

Implementierungsunabhängige Spezifikationsdefinition (Bild 4.6(a))
Die Spezifikationen der Regeln werden in der Attributierung in einem implementierungsunabhängigen Format[11] hinterlegt.

Einbindung einer externen Implementierung (Bild 4.6(b))
Ist der Einsatz proprietärer Systeme unumgänglich, so kann über die Spezifikations-ID auf eine externe Implementierung verwiesen werden[12].

Unabhängig von der gewählten Spezifikationsarchitektur ist zusätzlich eine Technik erforderlich, durch die die übergebenen Argumente innerhalb des Funktionsrumpfs adressiert werden können. Während sich Attribute in den Produktionen der Attributgrammatiken leicht durch ihre Bindung an (Non)Terminale identifizieren lassen, wird dieses Vorgehen durch die für Metamodelle typische Graphstruktur und die enthaltenen Assoziationsmultiplizitäten erschwert.

Bei der Betrachtung konkreter attributierter Grammatiken fällt auf, dass viele semantische Regeln identische Aufgaben erfüllen, beispielsweise Werte innerhalb des Ableitungsbaums zu transportieren[13] oder Attributvorkommen durch Konstanten zu initialisieren. Um den Erstellungsprozess effizienter zu gestalten, bietet es sich an, häufig verwendete Funktionen in eine Standardbibliothek aufzunehmen, aus der sie in eine Attributierung importiert werden können. Anhang B.2 zeigt, wie sich diese in die Spezifikationsarchitektur einfügt.

Integration der Spezifikationsfragmente

Die implementierungsunabhängigen Spezifikationsfragmente werden in den Klassenattributen *SpezifikationID* und *SpezifikationDef* einer Attributierungsregel abgelegt. *SpezifikationID* ist eine eindeutige Identifikation der Berechnungsvorschrift, die dem Attributauswerter als Referenz für den Funktionsaufruf dient. *SpezifikationDef* nimmt die plattformunabhängige Spezifikationsbeschreibung auf. Liegt diese in einem nicht maschinell interpretierbaren Format - beispielsweise in Textform - vor, so wird zur Auswertung zusätzlich eine externe Funktionsbibliothek benötigt, die für jede *SpezifikationID* eine aufrufbare Methode implementiert (siehe Abbildung 4.6(b)).

[10] Anhang B.1 demonstriert die Anwendung beider Techniken anhand eines Beispiels.

[11] Hierfür könnte beispielsweise ein OCL-Dialekt entwickelt werden. OCL ist als plattformunabhängige Sprache konzipiert, die vielfältige Operationen zur Verarbeitung von Modelldaten bereitstellt, und für die bereits geeignete Interpreter existieren.

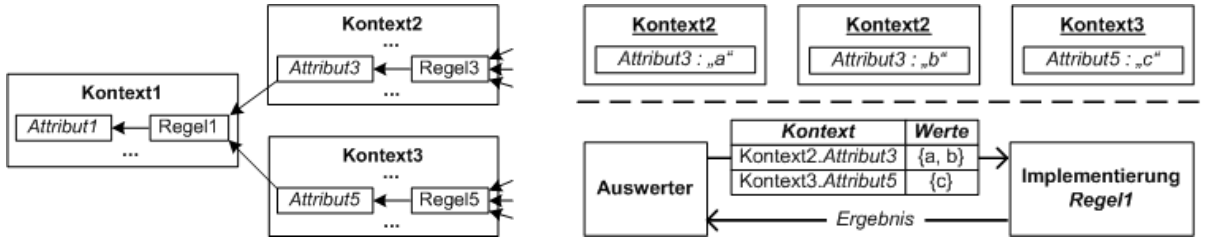
[12] Die Spezifikationsdefinition kann hier zur informellen (textuellen) Dokumentation der Berechnungssemantik verwendet werden.

[13] Die in manchen Systemen angewandte Technik, implizite Transferregeln für Attributwerte zu generieren, ist für attributierte Metamodelle nicht praktikabel, da durch Assoziationen induzierte Zyklen im Modellgraphen miteinander in Konflikt stehende Ausbreitungswege bedingen.

Übergabe und Referenzierung der Funktionsargumente

Die Zuordnung von Attributvorkommen zu einem Definitionskontext, sowie die Adressierung der Attributvorkommen wird in Abschnitt 4.3 ausführlich besprochen. Die für die Referenzierung relevanten Eigenschaften lassen sich wie folgt zusammenfassen:

- Ein Attributvorkommen wird durch den Namen seines definierenden Attributtyps identifiziert.
- Der Kontext von Attributierungsregeln und Attributvorkommen ist ihr definierendes Attributiertes Objekt.
- Die Adressierung eines Attributvorkommens in einem Kontext erfolgt gemäß der Syntax *Kontextname[optional: Subkontextname].Attributname*.
- Aufgrund der Assoziationsmultiplizitäten in Metamodellen liefert obige Adressierungssyntax auf Instanzebene - ähnlich einer OCL-Navigation - eine Menge von Attributwerten zurück.
- Der Zugriff auf Eigenschaften des Attributierten Objekts geschieht durch *Kontextname → Eigenschaftsname*.



(a) *Regel1* berechnet *Attribut1* in *Kontext1* und erhält zwei Argumentmengen mit den Instanzwerten von *Kontext2.Attribut3* und *Kontext3.Attribut5*

(b) Hier existieren von *Kontext2.Attribut3* zwei und von *Kontext3.Attribut5* ein Attributexemplar. Diese werden der Spezifikation, nach ihrer Adressierung gruppiert, übergeben

Abbildung 4.7: Prinzip der Argumentübergabe an die Spezifikationsimplementierung

Ein Attributvorkommen als Argument einer Attributierungsregel stellt dieser demnach eine bestimmte Teilmenge seiner Attributinstanzwerte zur Verfügung. Ob ein konkretes Attributexemplar in dieser Menge enthalten ist, wird durch den in der Adressierung angegeben Kontext bestimmt. Dieses Verfahren wird in Abbildung 4.7 anhand eines Beispiels illustriert. Die Übergabe von Objekteigenschaften erfolgt äquivalent für alle im lokalen Kontext verfügbaren Eigenschaften.

An dieser Stelle noch nicht implementiert, aber als potentielle Erweiterung nicht unerwähnt bleiben sollte die Möglichkeit, die zu übergebenden Argumente durch den Auswerter um zusätzliche Informationen anzureichern. Denkbar wäre beispielsweise die

Umsetzung eines Reflection-Mechanismus, indem unter entsprechenden Schlüsselwörtern Metadaten an die Spezifikationsimplementierungen ausgeliefert werden, die so Zugriff auf die Struktur des Metamodells und der Attributierung selbst erhalten.

Implementierung einer Spezifikationsarchitektur in Java

Die Bindung an eine konkrete Programmiersprache (siehe Abbildung 4.6(b)) ermöglicht eine verhältnismäßig schnelle und einfache prototypische Implementierung, da auf die Entwicklung einer dezidierten Sprache zur Formulierung von Berechnungsvorschriften verzichtet werden kann. Die Umsetzung der Spezifikationsarchitektur in Java lässt sich dabei leicht für andere Sprachen und Sprachmodelle portieren.

Es wird vereinbart, dass über die *SpezifikationID* einer Attributierungsregel jeweils eine Java-Methode gleichen Namens referenziert wird, die die entsprechende Berechnungsvorschrift implementieren muss. Die Methoden werden in einer Java-Klasse zusammengefasst, die den (externen) implementierungsabhängigen Teil der Metamodellattributierung repräsentiert. Die Spezifikationssemantik wird in Textform in der Eigenschaft *SpezifikationDef* dokumentiert.

Der Auswerter übergibt die Funktionsargumente über zwei *Hashtabellen* als (Schlüssel / Wert)-Paare codiert an die Implementierung:

Tabelle 1 - Attribute
: Enthält Attributinstanzen in der Form (Adressierung/Attributwerte) im Format (String/Set).

Tabelle 2 - Objekteigenschaften
: Übergibt die statischen Objekteigenschaften in der Form (Adressierung/Eigenschaftswert) als (String/Datentyp der Objekteigenschaft).

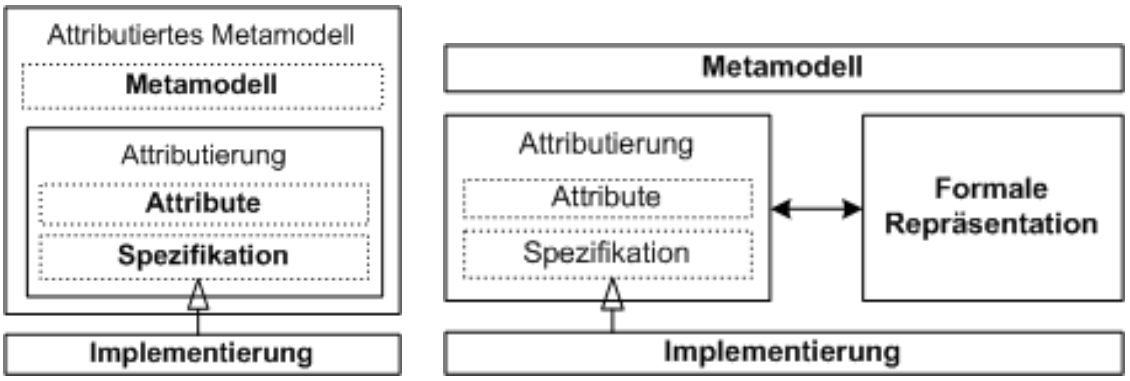
Der Implementierung der Regel *Umsatz* im Kontext *Rechnung* des Beispiels 4.4 wird also eine Attributtabelle übergeben, die unter dem Schlüssel *Ware.Preis* einen Eintrag mit den Preisen aller assoziierten Wareninstanzen besitzt. Die Eigenschaftstabelle enthält einen Eintrag unter *Rechnung→RNr* mit dem Wert dieser Objekteigenschaft. Aufgabe der Implementierung ist es, die Summe der Preise als Ergebnis an den Attributauswerter zurückzuliefern. Das Erstellen dieser Funktion kann vereinfacht werden, indem das Methodenskelett einer Spezifikationsimplementierung toolgestützt angelegt wird (siehe Anhang B.3).

Da davon auszugehen ist, dass in der Praxis viele Spezifikationen identische Aufgaben durchführen, bietet es sich an häufig gebrauchte Standardmethoden in eine Bibliothek auszulagern, um sie dann für die jeweilige Einsatzumgebung zu instantiieren. Dazu zählen beispielsweise Transferfunktionen, die Attributwerte über Assoziationen weiterleiten und auch die Belegung von Attributen mit Initialwerten. Ebenfalls können über Standardmethoden einfache arithmetische Operationen und Vergleichsoperatoren (für

Bedingungen) angeboten werden. Die Umsetzung kann durch Templates (siehe Anhang B.4) erfolgen, deren variable Bestandteile zur Definitionszeit an die Umgebung des jeweiligen Objekts angepasst werden. Wie Java-Methodentemplates für Standardfunktionen aussehen könnten zeigt Anhang B.5.

4.2.4 Textuelle Notation

Bisher wurde die Struktur der Attributierungssprache (abstrakte Syntax), eine graphische Repräsentation (konkrete Syntax) sowie eine Methode zur Deklaration von Berechnungsvorschriften (Spezifikationssyntax) entwickelt. Es gilt nun, diese Sprachfragmente in einer umfassenden Attributierungsnotation zu vereinen, die als einheitliche Schnittstelle gegenüber einem Attributauswerter dient.



(a) Die Attributierung ist im Metamodell enthalten

(b) Die Attributierung ist außerhalb des Metamodells definiert

Abbildung 4.8: Zwei Techniken Metamodelle mit einer Attributierung zu versehen

Durch die Anreicherung eines Meta-Metamodells um Attributierungselemente, wie in 4.2.1 beschrieben, können Attribute direkt im Metamodell deklariert werden. Metamodell und Attributierung bilden eine Einheit, lediglich die Implementierung der Berechnungsvorschriften erfolgt extern, um die Plattformunabhängigkeit des Metamodells zu wahren (siehe Abbildung 4.8(a)). Dadurch kann das attributierte Metamodell als Ganzes serialisiert[14] werden. Je nach Anwendungskontext kann es aber auch von Vorteil sein, die Attributierung außerhalb des Metamodells zu definieren. Bild 4.8(b) zeigt, wie ein Metamodell durch externe Attributierungsfragmente angereichert werden kann[15].

[14] Die Serialisierung kann in dem offenen Dateiformat XML Metadata Interchange (XMI) (vgl. http://www.omg.org/technology/documents/formal/xmi.htm) erfolgen, das von der OMG für den Einsatz mit UML bzw. MOF entwickelt wurde.

[15] Dieses Prinzip findet auch bei der Object Constraint Language Anwendung, deren Restriktionen in einer eigenen Datei abgelegt werden können.

Folgende Überlegungen sprechen für die getrennte Darstellungsform:

- Eine unabhängige Versionsverwaltung für das Metamodell und dessen Attributierung ist bei einer integrierten Darstellung nicht möglich.
- Ein Metamodell kann als Basis für unterschiedliche, auswechselbare Attributierungen fungieren. Es wird eine Technik zur dynamischen Bindung einer Attributierung an ein gegebenes Metamodell benötigt.
- Im Metamodell vorhandene Attributierungskonstrukte können sich störend auf vorhandene Entwicklungstools, beispielsweise Codegenerierungswerkzeuge, auswirken.
- Zur Speicherung einer Attributierung im Metamodell ist eine Modifikation des Meta-Metamodells und damit der bestehenden Entwicklungswerkzeuge notwendig. Durch eine externe Definition kann die Attributierungstechnik dagegen als Aufsatz für bestehende Techniken realisiert werden. Metamodelle können weiterhin mit „klassischen“ UML-Modellierungstools [16] entwickelt werden.
- Eine externe Repräsentation bietet eine unabhängige Schnittstelle zur Attributierungslogik, was die Konstruktion von Attributauswerten sowie die Überprüfung auf syntaktische und semantische Korrektheit, vereinfacht.

Die Architektur aus Bild 4.8(b) verlangt Sprachen zur Beschreibung folgender Sprachfragmente:

Metamodell
: Die UML-Metamodelle, die die Basis einer Attributierung bilden, können weiterhin im XMI-Format abgelegt werden.

Implementierung
: Die Implementierung kann in einem beliebigen, vom Attributauswerter unterstützten, Format erfolgen. Interoperabilität kann z.B. durch den Einsatz von Remote Procedure Call (RPC)-Aufrufen (vgl. RFC 1057 und 1831) erreicht werden.

Attributierung
: Die Attributierung besteht aus der Attributierungsstruktur und der Deklaration von Spezifikationen (siehe Abschnitte 4.2.1 und 4.2.3).

Eine geeignete Attributierungsnotation muss die Attributierungsstruktur sowie die Spezifikationen abbilden können. Die Sprache sollte dabei einfach aufgebaut und intuitiv verständlich sein. In ihren Grundzügen wurde sie deshalb der bereits vorgestellten Notation attributierter Grammatiken nachempfunden. Die verschiedenen Aspekte der Attributierung werden dabei in je einem eigenen Teilabschnitt beschrieben:

[16] Beispielsweise ArgoUML (http://argouml.tigris.org)

Allgemeine Informationen
Definiert Metadaten der Attributierung wie z.B.: Name, Autor, Version, Erstellungsdatum, Metamodell das der Attributierung zugrunde liegt, Java-Klasse mit Implementierung, etc.

Attributdeklarationen
Die Attribute werden als 3-Tupel, bestehend aus Attributname, Datentyp und Beschreibung, deklariert.

Attributierungsregeln: Definitionen
Die Deklaration der Attributierungsregeln erfolgt, ähnlich den OCL-Constraints, innerhalb ihres Kontexts, d.h. der ID des jeweiligen Attributierten Objekts. Nach der Angabe des Kontexts folgt die Liste der enthaltenen Definitionen. Eine Definition wird durch den Typ des Attributvorkommens identifiziert, das die Regel definiert. Weiterhin werden zu jeder Definition ihre Spezifikations-ID sowie eine Liste aller verwendeten Attributvorkommen in ihrer vollständigen Adressierung gespeichert.

Attributierungsregeln: Bedingungen
Die Notation einer Bedingung entspricht der einer Definition, mit dem Unterschied, dass eine Bedingung durch ihren Namen identifiziert und ihre Beschreibung angegeben wird.

Spezifikationen (optional)
Liegen die Spezifikationsdefinitionen in einem vom Attributauswerter unterstützen Format vor, so können sie in diesem Abschnitt, indiziert durch die jeweilige Spezifikations-ID, direkt angegeben werden.

Unterstützt das Modellierungswerkzeug/der Attributauswerter die Verwendung von Funktionstemplates, so können statt der Spezifikations-ID auch Templateaufrufe verwendet werden, für die durch einen Präprozessorlauf entsprechende Spezifikationsdefinitionen erzeugt werden.

Die Attributierung kann nun verhältnismäßig einfach auf syntaktische Korrektheit und die Einhaltung der statischen Semantik geprüft werden. Vor der Auswertung muss zusätzlich sichergestellt werden, dass die textuelle Beschreibung der Attributierung mit dem gegeben Modell kompatibel ist, d.h. die jeweils enthaltenen Elemente müssen übereinstimmen (siehe Abschnitt 5.1).

Die vollständige Attributierung aus Abbildung 4.4 in dieser Notation ist in Abbildung 4.9 zu sehen. Die Berechnungsvorschriften werden mit Hilfe der vorgegebenen Argumente aus der Standardbibliothek generiert. Die Syntax dieser Sprache in EBNF-Notation (vgl. [Med00, S. 34ff]) findet sich in Anhang B.6.

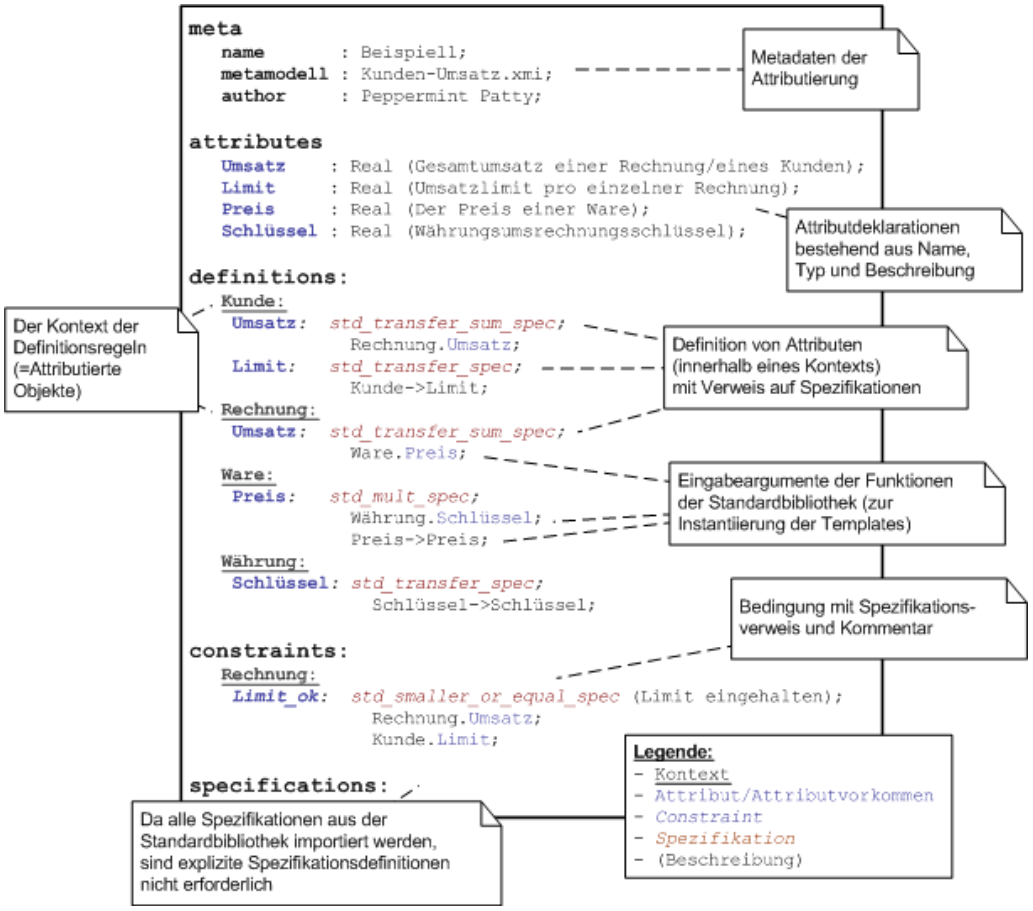


Abbildung 4.9: Textuelle Repräsentation der Attributierung aus Abbildung 4.4

4.3 Semantik

Neben der abstrakten Syntax, die den syntaktischen Aufbau einer Attributierung festlegt, wird für eine sinnvolle Interpretation dieser Sprachkonstrukte auch die im Folgenden definierte Semantik benötigt. Diese formuliert zusätzliche Restriktionen auf der Attributierungssprache, womit die Menge der syntaktisch korrekten Attributierungen auf die Teilmenge der semantisch korrekten Attributierungen eingeschränkt wird.

Da attributierte Grammatiken als Vorlage für die Implementierung des Attributierungskonzepts auf Metamodellen dienen, werden die wesentlichen Eigenschaften der zu definierenden Semantik in Abschnitt 4.3.1 aus einem Vergleich dieser beiden Techniken herausgearbeitet. Die Adressierung von Attributvorkommen, die bereits in Abschnitt 4.2.3 in Grundzügen vorgestellt wurde, stellt einen zentralen Teil der Attributierungssemantik dar, weshalb in 4.3.2 ausführlich darauf eingegangen wird. In Abschnitt 4.3.3 werden zuletzt die Abhängigkeiten behandelt, die durch Attributierungsregeln zwischen Attributvorkommen entstehen.

4.3.1 Vergleich zwischen Attributgrammatiken und attributierten Metamodellen

Für den Vergleich kontextfreier Grammatiken mit Metamodellen bietet sich eine Gegenüberstellung der Sprachkonstrukte auf den jeweiligen Definitions- und Instantiierungsebenen an. Diese Betrachtung, die in Abschnitt 4.1.1 bereits ansatzweise durchgeführt wurde, wird nun vertieft[17]. Die grundlegenden Bausteine entstammen der Definition attributierter Grammatiken (siehe Abschnitt 2.2) sowie der abstrakten Syntax für Metamodellattributierungen (siehe Abschnitt 4.2.1).

Das zentrale Element der Attributierung kontextfreier Grammatiken sind die Produktionen der zugrunde liegenden Grammatik. Auf Ebene der Sprachdefinition werden diese durch semantische Regeln angereichert, die auf den Vorkommen der Attribute an (Non)Terminalen innerhalb der Produktionen Berechnungsvorschriften festlegen. Daraus ergeben sich schließlich die Abhängigkeiten zwischen den Attributvorkommen.

Bei attributierten Metamodellen legt das - um Attributierungselemente erweiterte - Meta-Metamodell die syntaktische Attributierungsstruktur fest. Attributierten Objekten sind Attributierungsregeln zugeordnet, welche auf Metamodellierungsebene Berechnungsregeln für Attributvorkommen definieren. Die Abhängigkeiten, die zwischen Attributvorkommen entstehen, ergeben sich aus den Argumenten dieser Regeln, bei denen es sich um Vorkommen von Attributen an „benachbarten“ Attributierten Objekten handelt. Bei Attributgrammatiken besteht diese „Nachbarschaft“, die die Menge der erlaubten Eingabeargumente für semantische Regeln beschränkt, aus produktionslokalen Attributvorkommen. Für attributierte Metamodelle muss im Weiteren eine äquivalente Nachbarschaftsbeziehung definiert werden, die festlegt unter welchen Bedingungen Attributvorkommen als Argumente von Attributierungsregeln verwendet werden dürfen und wie diese aus dem jeweiligen Kontext heraus adressiert werden können.

Auf Instanzebene werden die Knoten in einem Modell (einem Ableitungsbaum) durch instantiierte Attributierte Objekte (Vorkommen von Produktionen) gebildet, deren semantische Regeln die lokalen Attribute gemäß der jeweiligen Berechnungsvorschrift aus den „benachbarten“ Attributwerten generiert.

Aus den bisherigen Beobachtungen lässt sich schließen, dass Attributierte Objekte und Attributierungsregeln in ihrer Funktion und ihrer Struktur den Produktionen und semantischen Regeln attributierter Grammatiken entsprechen. Die Semantik von Attributgrammatiken lässt sich damit zum größten Teil auf diese Elemente übertragen.

Unterschiede zwischen den Konzepten finden sich dagegen bei Definition und Zuordnung von Attributen. Bei attributierten Grammatiken sind Attribute stets eindeutig mit (Non)Terminalsymbolen verbunden, wobei Vorkommen der Attribute durch

[17] Anhang B.7 zeigt die beiden Techniken zugrunde liegenden Konzepte anhand der Darstellung der Attributierungsfragmente für ein konkretes Beispiel.

(Non)Terminal-Vorkommen innerhalb von Produktionen automatisch induziert werden. Wie in Abschnitt 2.2.2 beschrieben, wird die Eindeutigkeit dieser Zuordnung in der Praxis allerdings manchmal abgeschwächt, um verschiedene Sprachsymbole mit Attributen gleichen Namens versehen zu können. Diese flexiblere Gestaltungsmöglichkeit wurde direkt in die Definition der attributierten Metamodellen aufgenommen: Attribute sind ungebundene Spezifikationen, die konkrete Attributvorkommen charakterisieren (durch Name und Typ) anstatt sie wie bei Attributgrammatiken automatisch an den entsprechenden Symbolen zu erzeugen. Dadurch können Attribut(vorkommen) an verschiedenen Objekten als Repräsentanten eines gemeinsamen Konzeptes dargestellt werden.

Ein weiterer Unterschied, der in der Struktur von UML-Modellen begründet liegt, ist die fehlende Unterscheidung zwischen ererbten und synthetisierten Attributen. Da auf Instanzebene kein Wurzelelement identifiziert werden kann, ist die Übertragungsrichtung der Attributwerte vom gewählten Betrachtungspunkt abhängig.

Weitere Anforderungen, die an eine Attributierungssemantik auf Grammatiken sowie auf Metamodellen zu stellen sind, bestehen in einer konsistenten Typisierung der Attribute, sowie der Einhaltung von semantischen Vorgaben bei der Zuordnung von Attributvorkommen zu Attributierten Objekten bzw. Attributierungsregeln, worauf im folgenden Abschnitt eingegangen wird.

4.3.2 Adressierung und Kontext

Bei der Übertragung des Attributierungskonzepts müssen die besonderen Eigenschaften, die sich aus der Struktur von Metamodellen ergeben, berücksichtigt werden. Um Attributvorkommen zu adressieren, wurde in Abschnitt 4.2.3 bereits der Begriff des Definitionskontexts eingeführt, für den nun eine detailliertere Beschreibung folgt.

Definition des Kontextbegriffs und Adressierung von Attributen

Wie in 4.3.1 erläutert, existieren Attributvorkommen in Grammatiken innerhalb von Produktionen, wobei Vorkommen derselben Produktion wiederum Eingabewerte für semantische Regeln sein können. Die Produktionen stellen damit einen Definitionskontext für Attributvorkommen dar, d.h. sie bilden zur Laufzeit einen Container für die Instanzwerte aller im Kontext definierten Attribute.

Äquivalent zu grammatikalischen Produktionen übernehmen in Metamodellen Attributierte Objekte die Aufgabe, für Attributvorkommen und Bedingungen einen Definitionskontext bereitzustellen:

- Ein Attributiertes Objekt definiert ein Attributvorkommen *AV* eines Attributtyps *A*, wenn es eine Attributierungsregel *D* vom Typ Definition besitzt, für die gilt: *D* steht über *definiert* mit *AV* in Relation, und *AV* ist mit Attribut *A* assoziiert. Innerhalb eines Definitionskontexts darf jeder existierende Attributtyp durch maximal ein Attributvorkommen repräsentiert werden.
- Ein Attributiertes Objekt definiert eine Restriktion *R*, wenn es eine Attributierungsregel *R* vom Typ Bedingung besitzt. Der Name einer Bedingung muss im Kontext des Objekts eindeutig sein.

Aus dieser Festlegung und der in 4.2.1 gestellten Bedingungen an die Eindeutigkeit von Objekt- und Attributnamen lässt sich ein Adressierungsschema für Attributvorkommen ableiten: Die Kontexteinschränkung erfolgt durch Angabe der ID des Attributierten Objekts, das das jeweilige Attributvorkommen definiert. Als Trennzeichen folgt ein Punkt und danach die ID des Attributvorkommens, welche dem Namen seines charakterisierenden Attributs entspricht. So adressiert *Kunde.Umsatz* beispielsweise das Attributvorkommen vom Attributtyp *Umsatz* im Kontext des Objekts *Kunde*. Existieren zur Laufzeit mehrere Instanzen des Objekts *Kunde*, so liefert die Adressierung *Kunde.Umsatz* im Modell die Instanzwerte des Attributs *Umsatz* von allen *Kunde*-Instanzen zurück.

Beschränkung des Kontexts

In der Praxis ist die Verwendung aller existierenden Instanzwerte eines Attributvorkommens als Argumente einer Attributierungsregel meist nicht wünschenswert. Dies würde zahlreiche unnötige Abhängigkeiten generieren, welche den Auswertungsvorgang verkomplizieren oder sogar unmöglich machen. So benötigt die Attributierungsregel, die in Abbildung 4.4 für die Berechnung des Kundenumsatzes zuständig ist, dazu nur eine bestimmte Teilmenge der Instanzwerte von *Rechnung.Umsatz* als Eingabe, genauer gesagt sind nur die Rechnungsinstanzen von Interesse, die mit der jeweiligen Kundeninstanz assoziiert sind. Diese Beschränkung auf eine Teilmenge von Instanzen durch Assoziationen ist typisch für Metamodelle, da Assoziationen ja gerade mit Blick auf semantische Zusammenhänge in der Domäne definiert sind, weshalb auch die Navigation über Assoziationen ein zentrales Element der Sprache OCL ist.

Um auf der Menge der Attributinstanzen Einschränkungen vornehmen zu können, wird der Begriff „Subkontext“ eingeführt. Dieser wird in eckigen Klammern nach der Angabe des Kontexts notiert: *Kontext[Subkontext].Attribut*. Der Subkontext entspricht dabei standardmäßig, ähnlich wie bei der Object Constraint Language, dem Rollennamen des gegenüberliegenden Assoziationsendes. Der Rollenname schränkt die Menge aller Instanzen des Kontexts (und damit der darin enthaltenen Attributwerte) auf die Untermenge der über diese Assoziation verbundenen Objektinstanzen ein. Die oben erwähnte Teilmenge von *Rechnung.Umsatz*-Instanzwerten kann also durch *Rechnung[r].Umsatz* ausgedrückt werden. Bestehen mehrere Assoziationen zum Zielobjekt, so lassen sich die

durch die jeweiligen Rollennamen identifizierten Instanzmengen auch über die Notation *Kontext[Subkontext1 + Subkontext2].Attribut* vereinigen.

Um äquivalent zum OCL-Schlüsselwort *self* auf die Instanz des definierenden Kontexts selbst zuzugreifen, wird ein reservierter Subkontext mit der Bezeichnung *[loc]* (lokal) eingeführt. Dieser wird beispielsweise zur Definition der Bedingung *Limit eingehalten* benötigt, die den Wert des *Umsatz*-Attributs aus derselben Rechnungsinstanz als Eingabe verwendet. Die Adressierung im Kontext *Rechnung* wäre hier: *Rechnung[loc].Umsatz*.

In der gewählten Auswertungssemantik werden Generalisierungen als Spezialfälle von Assoziationen behandelt. Eine Weiterleitung von Attributwerten darf nur vom direkten Supertyp zum Subtyp erfolgen. Zur Adressierung der Supertypinstanz steht der reservierte Subkontext *[gen]* (Generalisierung) zur Verfügung. Um die Vererbung von Attributvorkommen von Eltern- an Kinderobjekte zu automatisieren, können die dazu notwendigen Transferregeln während des Auswertungsprozesses automatisch generiert werden (siehe Abschnitt 5.1). Treten durch Mehrfachvererbung Konflikte bei der Weiterleitung auf, oder sollen vererbte Werte im Subtyp manipuliert werden, so ist ein manueller Eingriff in die Vererbungssemantik durch Definition eigener Vererbungsregeln möglich. Assoziationen und Objekteigenschaften werden gemäß der üblichen UML-Semantik ebenfalls automatisch vererbt, so dass Attributierungsregeln vollen Zugriff auf Subkontexte und Eigenschaften des Supertyps haben.

Zur Auswertungszeit hängt das Ergebnis, das durch die Subkontext-Restriktion zurückgeliefert wird, davon ab, mit welchen Objektinstanzen die jeweilige Instanz des Kontexts assoziiert ist. Diese Einschränkung der gültigen Subkontextangaben wurde bereits mit dem Begriff „Nachbarschaft“ eines Attributierten Objekts grob umschrieben. Es wird gefordert, dass der angegebene Subkontext vom definierenden Attributierten Objekt aus direkt über eine Assoziation oder Generalisierung erreichbar sein muss. Es ist also *Rechnung[r].Umsatz* ein gültiges Argument für die Attributierungsregel, die den Kundenumsatz berechnet, nicht aber z.B. *Ware[w].Preis*, da dieser Subkontext vom Kontext des Kundenobjekts aus nicht verfügbar ist.

Konventionen zur Vereinfachung der Adressierungssyntax

Nachdem die grundlegenden Eigenschaften der Adressierung von Attributwerten eingeführt wurden, wird eine Variante der Adressierungsnotation vorgestellt, die den praktischen Einsatz erleichtert.

Es wird davon ausgegangen, dass nicht die Abfrage aller existierenden Instanzwerte den Regelfall darstellt, sondern der über einen Assoziations-Subkontext eingeschränkte Zugriff. Dies würde nach bisheriger Konvention immer die Angabe der Rollennamen erfordern, was sich zum einen negativ auf die Lesbarkeit der Attributierung auswirkt und zum anderen voraussetzt, dass sämtliche relevanten Assoziationsenden mit Rollenbezeichnern versehen wurden. Um dies zu umgehen, wird vereinbart, dass eine Adressie-

rung ohne weitere Einschränkung implizit stets durch den Standard-Subkontext ergänzt wird. Dieser beinhaltet die Vereinigung aller existierenden Assoziationen zum Zielobjekt. Die verkürzte Form *Rechnung.Umsatz* innerhalb des Objekts Kunde ist daher äquivalent zu *Rechnung[r].Umsatz*. Ebenfalls implizit ergänzt werden die reservierten Subkontexte *[loc]* und *[gen]*[18]. Durch diese Vereinfachung ist die Angabe eines Rollennamens nur noch nötig, wenn mehrere Assoziationen zum Zielobjekt existieren von denen nur eine Teilmenge selektiert werden soll. Um den Zugriff auf alle existierenden Instanzen eines Attributvorkommens auch weiterhin zu ermöglichen wird der Subkontext *[all]* eingeführt, der - ähnlich der OCL-Anweisung allInstances - dem uneingeschränkten Kontext entspricht. Die ursprüngliche Semantik von *Rechnung.Umsatz* lässt sich damit durch *Rechnung[all].Umsatz* reproduzieren.

Adressierung von Objekteigenschaften

In der abstrakten Syntax attributierter Metamodelle wurden Objekteigenschaften definiert, die an Objekten - ähnlich Klassenattributen - statische[19] Werte aufnehmen können. Um sie von Attributen unterscheiden zu können, werden sie mit → statt einem Punkt vom Kontext abgetrennt. *Kunde.Limit* bezeichnet die Menge der Instanzwerte des Attributvorkommens *Limit*, während *Kunde→Limit* den Wert der Objekteigenschaft *Limit* meint. Da für Objekteigenschaften angenommen wird, dass sie Träger lokaler Informationen sind, dürfen sie nur aus ihrem definierenden Kontext angesprochen werden, die explizite Angabe des Subkontexts *[loc]* ist aber nicht erforderlich.

Zusammenfassung der Adressierungssemantik

Es folgt eine kurze Übersicht über die definierten Adressierungsvarianten, wobei die Kontexteinschränkungen in absteigender Reihenfolge spezieller werden:

Kontext[all].Attribut
: Liefert alle existierenden Instanzwerte des Attributvorkommens in dem angegebenen Zielkontext zurück. Da keinerlei situationsabhängige Einschränkung gemacht wird, ist dieser Ausdruck unabhängig von der Umgebung, in der er definiert ist.

Kontext.Attribut
: Abhängig von der Umgebung in der dieser Ausdruck definiert wurde, werden alle existierenden Assoziationen zum Zielkontext verfolgt und die Vereinigung der Attributwerte als Ergebnis zurückgeliefert.

[18] Besitzt ein Objekt eine Assoziation zu seinem Supertyp, so entsteht ein Konflikt, da sowohl die Generalisierungsinstanz wie auch die assoziierten Instanzen gemeint sein könnten. In diesem Fall muss der Subkontext explizit definiert werden.

[19] Statisch meint hier, dass die Werte von Objekteigenschaften im Gegensatz zu Attributen nicht von anderen Werten im Modell abhängen und damit auch keine Abhängigkeiten erzeugen.

Kontext[Subkontext1+Subkontext2].Attribut
: Abhängig von der Umgebung werden die Zielkontext-Instanzen, die über Assoziationen mit den Rollennamen Subkontext1 und Subkontext2 erreichbar sind als Ergebnis interpretiert.

Kontext[Subkontext].Attribut
: Nur Zielkontext-Instanzen die über die Assoziation mit Rollennamen Subkontext verfügbar sind tragen zum Ergebnis bei.

Kontext[gen].Attribut
: Der Zielkontext ist ein Supertyp der aktuellen Umgebung. Die Abfrage bezieht sich auf die konkrete Instanz des Supertyps der Objektinstanz.

Kontext[loc].Attribut
: Adressiert wird die Instanz, für die die Auswertung vorgenommen wird.

Kontext→Eigenschaft
: Der Zugriff auf eine Eigenschaft erfolgt im lokalen Kontext. Der Ergebniswert ist der Wert der Eigenschaft in der jeweiligen Objektinstanz.

Anhang B.8 verdeutlicht die Funktionsweise der Adressierungstechnik anhand einiger Beispiele.

Erweiterung des Subkontexts

Die Möglichkeit, die Zielmenge der Instanzwerte über einen Subkontext genau festzulegen, bietet eine Schnittstelle für zukünftige Erweiterungen, die auch komplexere Selektionen unterstützen. So könnten hier beispielsweise spezielle Selektoroperationen oder sogar Teile der OCL-Navigationstechnik implementiert werden, um gezielt auf Teilmengen von Attributinstanzen zugreifen zu können, wodurch sich nicht zuletzt auch die Anzahl der Abhängigkeiten im Modell und damit auch die zur Auswertung benötigte Zeit reduzieren würde.

4.3.3 Abhängigkeiten und Auswertung

Die Auswertung einer Attributierung erfordert zunächst eine Abhängigkeitsanalyse, die die Attributvariablen unter Berücksichtigung ihrer gegenseitigen Abhängigkeiten in einem Gleichungssystem anordnet. Wie auch bei attributierten Grammatiken und OCL-Ausdrücken ist der Auswertungsprozess einer Modellattributierung atomar, d.h. das zugrunde liegende Modell ändert sich während des Berechnungsvorgangs nicht und alle berechneten Attributwerte sind endgültig. Enthält die Attributierung Bedingungen, so müssen diese alle zu wahr ausgewertet werden, ansonsten erfüllt das Modell nicht die durch die Attributierung festgelegten semantischen Anforderungen.

Abhängigkeiten entstehen durch Verwendung von Attributvorkommen als Argumente der Berechnungsvorschriften. Attributvorkommen, die durch einen konstanten Wert initialisiert werden, sowie Objekteigenschaften, sind unmittelbar verfügbar und rufen ihrerseits keine Abhängigkeiten hervor. Definitionen, die nur Konstanten als Argumente besitzen, bilden den Anfang einer Abhängigkeitskette. Für Bedingungen gilt, dass sie in jedem Fall am Ende einer solchen Kette stehen, da sie keine weiterverwendbaren Ergebniswerte bereitstellen. Der Test auf Zyklenfreiheit einer Attributierung für alle denkbaren Modellinstanzen wäre, wie auch bei Attributgrammatiken, in der Praxis oft aufwändig. Daneben kann eine zyklische Attributierung, die auf Metaebene Zyklen enthält, auf Instanzebene dennoch zyklenfrei sein, da problematische Objekte möglicherweise nicht instantiiert wurden. Daher wird die Forderung nach Zyklenfreiheit eingeschränkt auf folgende Prämissen:

- Ein Attributvorkommen darf nicht als Argument seiner definierenden Attributierungsregel auftreten.
- „Produktionslokal“, d.h. in direkter Nachbarschaft der einzelnen Attributierten Objekte dürfen keine Zyklen auftreten.
- Zyklen, die sich durch indirekte Abhängigkeiten ergeben, werden ignoriert.

Da Modellelemente die Struktur des Metamodells nicht vollständig abbilden müssen, ist wie bei OCL-Ausdrücken eine Toleranz gegenüber fehlenden Argumenten bzw. deren Antizipation wichtig. Existiert eine geforderte Attributinstanz zur Laufzeit nicht, so wird an ihrer Stelle ein *null*-Wert übergeben.

5 Auswertung der Attributierung

Die Berechnung der Attributinstanzwerte auf einem Modell der Anwendungsdomäne erfolgt auf Basis eines attributierten Metamodells und einer Implementierung der verwendeten Spezifikationen. Abschnitt 2.2.4 zeigt, dass dieser Vorgang, und damit auch der Aufbau des Attributauswerters, in zwei Teilphasen gegliedert werden kann: Die Strategiephase, die die Auswertungsordnung durch Analyse der Abhängigkeitsbeziehungen bestimmt, und die Auswertungsphase in der die Werte der Attribute gemäß den ermittelten Voraussetzungen berechnet werden.

Da auf dem in dieser Arbeit vorgestellten Attributierungskonzept bisher allerdings keine Einschränkungen definiert wurden, durch die sich Abhängigkeiten statisch aus der Struktur der Attributierung ableiten ließen, beschreibt Abschnitt 5.1 ein dynamisches Verfahren zur Abhängigkeitsanalyse auf Meta- und Instanzebene unter Berücksichtigung der speziellen Anforderungen attributierter Metamodelle. Die Berechnung der Attributwerte, basierend auf den aus der Abhängigkeitsanalyse gewonnenen Erkenntnissen, wird schließlich in 5.2 erläutert, wobei auch ein Ansatz zur Behandlung zyklischer Abhängigkeiten im Gleichungssystem der Attributierung vorgestellt wird, der auf Techniken der Datenflussanalyse aufsetzt, und die Analysefähigkeiten, insbesondere was die Untersuchung von Struktur- und Verhaltensmerkmalen der Modelle anbelangt, stark erweitert.

5.1 Abhängigkeitsanalyse

Die Abhängigkeitsanalyse hat zur Aufgabe, auf dem Gleichungssystem der semantischen Regeln, das die Attributvorkommen als Variablen enthält, eine gültige Auswertungsordnung festzulegen und diese in einer für den weiteren Auswertungsvorgang geeigneten Form zur Verfügung zu stellen. Auch können in dieser Phase an den gegebenen Attributierungen notwendige Ergänzungen automatisch vorgenommen und sie auf Einhaltung der statischen Semantik geprüft werden. Da zyklische Definitionen eine sinnvolle Interpretation der Attributierung im Normalfall schwierig oder sogar unmöglich machen, ist deren Entdeckung bzw. Ausschluss ebenfalls von Bedeutung.

Die Abhängigkeitsanalyse auf Ebene der attributierten Metamodelle, d.h. der Attributvorkommen, wird in Abschnitt 5.1.1 beschrieben. In 5.1.2 wird diese Betrachtung dann

für die Laufzeitinstanzen der Attribute, die Attributexemplare, wiederholt. Deren Abhängigkeitsbeziehungen sind von unmittelbarer Bedeutung für den sich daran anschließenden Auswertungsprozess.

5.1.1 Abhängigkeitsanalyse im Metamodell

Die Abhängigkeitsanalyse auf Metaebene ist der erste Schritt im Auswertungsprozess. Durch sie können zyklische Definitionen aufgedeckt, sowie die Attributierung automatisch vervollständigt und auf semantische Korrektheit hin überprüft werden. Wie bei Attributgrammatiken können die Abhängigkeiten entweder global oder aber auf ein lokales Umfeld beschränkt betrachtet werden. Wurde die Attributierung außerhalb des betreffenden Metamodells definiert (siehe Abschnitt 4.2.4), so muss vor der Auswertung zusätzlich sichergestellt werden, dass die Attributierung mit den Elementen des jeweiligen Metamodells kompatibel ist.

Vervollständigung und Tabellarische Darstellung

Das durch die Attributierung definierte Gleichungssystem kann in eine tabellarische Darstellung überführt werden, die alle im weiteren Auswertungsprozess benötigten Daten integriert. Jede Attributierungsregel erzeugt einen Tabelleneintrag, der das in ihr definierte Attributvorkommen[1], dessen Typ und Kontext, die zugehörigen Argumente mit ihrem jeweiligen Subkontext, sowie die Spezifikation enthält. Diese Form der Repräsentation ist im Gegensatz zu der für Menschen gedachten textuellen (4.2.4) bzw. graphischen (4.2.2) Notation, und der durch die abstrakte Syntax festgelegten formalen Struktur (4.2.1) für den Gebrauch im weiteren Auswertungsprozess optimiert.

Insbesondere können an dieser Stelle auch automatisch durchzuführende Manipulationen und Ergänzungen der Attributierung vorgenommen werden, sofern das Attributierungskonzept dies vorsieht. So müssen beispielsweise, um die vollständige Weiterleitung von Attributwerten innerhalb einer Generalisierungshierarchie zu garantieren (siehe 4.3.2), für alle Attributvorkommen eines Supertyps, die in dessen Subtypen nicht manuell überschrieben werden, Transferregeln erzeugt und eingetragen werden.

Die Darstellung 5.1 zeigt das Gleichungssystem der Attributierung auf dem Beispiel aus 3.1. Jedem Attributvorkommen wurde eine global eindeutige *ID* zugeordnet. Die Argumente, die eine Attributierungsregel verwendet, sind in Form einer Liste von *(Subkontext, Attributvorkommen)*-Tupeln gegeben. Um den Suchaufwand bei der rekursiven Ermittlung von Abhängigkeitsbeziehungen zu reduzieren, wird das Attributvorkommen durch eine Verknüpfung mit der *ID* seines Tabelleneintrags repräsentiert.

[1] Bedingungen generieren „virtuelle“ Attributvorkommen vom Typ boolean, die den Namen der Bedingung tragen. Eine Unterscheidung zwischen Definitionen und Bedingungen ist dann nicht mehr erforderlich.

ID	Kontext	Attribut	Argumente	Eigenschaften
1	Kunde	Umsatz	[r] 3	KNr, Limit
2	Kunde	Limit		KNr, Limit
3	Rechnung	Umsatz	[w] 5	RNr
4	Rechnung	Limit eing.	[loc] 3 2	RNr
5	Ware	Preis	[wa] 6	WNr, Preis
6	Währung	Schlüssel		Name, Schlüssel

Abbildung 5.1: Attributierung in tabellarischer Form (um die Übersichtlichkeit zu erhöhen wurde auf die Darstellung von Typisierung und Spezifikationen verzichtet)

Überprüfung auf Korrektheit

Während dem Aufbau des Gleichungssystems kann die Attributierung auf Kompatibilität mit einem konkreten Metamodell, sowie auf Einhaltung der festgelegten statischen Attributierungssemantik (siehe Abschnitt 4.3) geprüft werden. Für jeden Tabelleneintrag muss hierzu untersucht werden, ob der zugehörige Definitionskontext im Metamodell vorhanden ist und ob die benötigten Argumente in den angegebenen (Sub-)Kontexten verfügbar sind. Weiterhin müssen die Konstrukte bezüglich ihrer Typisierung übereinstimmen. Falls gewünscht, kann in diesem Schritt ebenfalls geprüft werden, ob Implementierungen bzw. Definitionen der verwendeten Spezifikationen vorliegen, was Voraussetzung für eine erfolgreiche Auswertung ist.

Objektlokalität

Die Analyse von Abhängigkeiten im produktionslokalen Umfeld spielt bei der Untersuchung attributierter Grammatiken eine wichtige Rolle (siehe 2.2.3). Für Attributgrammatiken in Normalform ist im lokalen Abhängigkeitsgraphen Zyklenfreiheit garantiert. Die speziellen Eigenschaften, die Attributierungen auf Metamodellen auszeichnen - der Verzicht auf die Unterscheidung zwischen synthetisierten und vererbten Attributen, sowie die für Metamodelle charakteristische Graphstruktur - erschweren jedoch hier die Formulierung einer äquivalenten Normalform. Nicht zuletzt würde dies auch die Modellierungsfreiheit beschränken, da der Informationsfluss stets nur von angewandten zu definierenden Attributvorkommen erlaubt ist. Ein Attributwert, der von einem *Objekt A* zu einem *Objekt B* übertragen wurde, könnte damit nicht mehr zurück zu *Objekt A* geleitet werden.

Um diese Einschränkung zu umgehen wird stattdessen für eine eingeschränkte „Umgebung“ der Attributierten Objekte ein Test auf Zyklenfreiheit durchgeführt. Die „Umgebung“ wird in Anlehnung an die Sprachregelung bei Attributgrammatiken im Folgenden als „objektlokal“ bezeichnet. Sie umfasst das betrachtete Attributierte Objekt als Hauptelement sowie sämtliche Objekte, deren Attributvorkommen Argumente semanti-

sche Regeln des Hauptobjekts bilden[2]

Der objektlokale Abhängigkeitsgraph

Der objektlokale Abhängigkeitsgraph wird für alle existierenden Attributierten Objekte aufgebaut. Da auf Metaebene keine Teilmengen von Attributexemplaren betrachtet werden können, repräsentiert jedes Attributvorkommen stets die Gesamtheit seiner möglichen Instanzen. Ein Modell kann daher zyklenfrei sein, obwohl sein Metamodell eine zyklische Definition enthält, falls die beteiligten Instanzmengen disjunkt oder nicht vorhanden sind. Dennoch ist der objektlokale Abhängigkeitsgraph eine brauchbare Annäherung, da bereits im attributierten Metamodell möglicherweise problematische Definitionen aufgedeckt bzw. ausgeschlossen werden können.

Das Prinzip, produktionslokale Abhängigkeiten zwischen den Attributvorkommen im Produktionsgraphen einzutragen, soll auch für die objektlokale Umgebung gelten. Die Notation entspricht Abbildung 4.4, wobei nur die relevanten Größen - Objekte, Attribute und Abhängigkeiten - dargestellt werden. Attributierungsregeln aller lokalen Objekte werden wie folgt untersucht[3]:

- Ist das Argument einer Attributierungsregel ein Attributvorkommen innerhalb der betrachteten Umgebung, so wird dies durch einen Abhängigkeitspfeil visualisiert, der optional mit dem Subkontext beschriftet werden kann.
- Ist das Argument nicht Teil der Umgebung, so wird die Quelle durch ein anonymes externes Objekt symbolisiert.
- Sind keine Argumente vorhanden, so wird dies durch die Abhängigkeit von einer Konstante angezeigt.

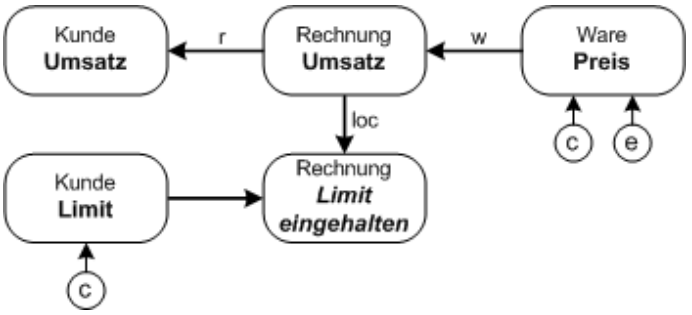
Abbildung 5.2(a) zeigt die objektlokale Umgebung für das Objekt *Rechnung*. Die Objekteigenschaften *Limit* und *Preis* werden durch Konstanten ersetzt, das Argument des Attributvorkommens *Preis*, das sich außerhalb der objektlokalen Umgebung befindet, wird als externer Wert angezeigt.

Reduziert man diese Darstellung auf die Attributvorkommen und ihre Abhängigkeitsbeziehungen, so erhält man den Graphen aus Abbildung 5.2(b), der in einer Adjazenzliste abgelegt werden kann. Um Zyklen zu detektieren kann darauf eine modifizierte Tiefensuche mit Laufzeitkomplexität $O(|V| + |E|)$ durchgeführt werden, wobei $|V|$ die Anzahl

[2] Durch die Adressierung auf Subkontextbasis sind dies stets entweder mit dem Hauptelement in Relation (Assoziation/Generalisierung) stehende Objekte, oder Objekte deren Vorkommen über den Subkontext *[all]* adressiert werden.

[3] Um zusätzliche Informationen zur Auswertungsreihenfolge - und damit möglicherweise einen Ansatz zur Formulierung einer statischen Auswertungsstrategie - zu gewinnen, können in einer Erweiterung dieses Verfahrens zusätzlich die indirekten Berechnungsabhängigkeiten zwischen den lokalen Attributvorkommen eingetragen werden (siehe 2.2.3: Berechnung des oberen bzw. des unteren charakteristischen Graphen).

(a) Objektlokale Umgebung.



(b) Reduzierte Darstellung des Abhängigkeitsgraphen als GAG

Abbildung 5.2: Objektlokale Abhängigkeiten des Objekts Rechnung aus Abbildung 4.4

der Knoten und $|E|$ die Anzahl der Kanten im Graph repräsentiert (vgl. [VT04, S. 26f, 92ff]). Aufgrund der linearen Komplexität des Suchalgorithmus sowie der Einschränkung auf ein lokales Umfeld kann davon ausgegangen werden, dass die Durchführung des Tests auf Zyklenfreiheit für alle Elemente des Metamodells in der Praxis ausreichend effizient ist.

Der globale Abhängigkeitsgraph

Der globale Abhängigkeitsgraph repräsentiert die gesamte Struktur der Abhängigkeiten im Metamodell und kann aus den lokalen Graphen zusammengesetzt werden. Im Gegensatz zu diesen zeigt er auch indirekte Zyklen zwischen Attributvorkommen an. Anhang C.1 enthält den globalen Graphen für das obige Beispiel, wobei die Abhängigkeiten nach Attributen und Definitionskontext geordnet wurden.

Die schon bei der Analyse der objektlokalen Umgebung existente Gefahr Zyklen zu detektieren, die in Modellen nicht auftreten, gilt allerdings für die globale Betrachtung durch die Einbeziehung indirekter Zyklen in verstärktem Maß. Als praktisches Anwendungsszenario für den globalen Abhängigkeitsgraphen kommt daher insbesondere die Visualisierung der Abhängigkeiten während der Definitionsphase (wie in Abbildung 4.4) in Frage, wobei nur die jeweils relevante Teilmenge der Abhängigkeiten eingeblendet wird.

5.1.2 Abhängigkeitsanalyse im Modell

Wurde für ein attributiertes Metamodell eine Abhängigkeitsanalyse durchgeführt, so kann darauf aufbauend eine detaillierte Aufstellung der individuellen Abhängigkeiten

für ein gegebenes Modell erfolgen[4]. In diesem Abschnitt wird beschrieben, wie die Abhängigkeitsbeziehungen zwischen Attributinstanzen aus dem Analyseergebnis der Metaebene abgeleitet werden können und welche Besonderheiten dabei zu berücksichtigen sind.

Attributabhängigkeiten im Modell

Auf Modellebene werden Instanzen von Attributvorkommen - die Attributexemplare - betrachtet, deren Abhängigkeitsbeziehungen im Umfeld der Modellelemente den objektlokalen Abhängigkeiten auf Metaebene entsprechen. Zur Darstellung des Analyseergebnisses bietet sich wieder die bekannte tabellarische Anordnung an, aus der der Auswertungsalgorithmus dann die individuellen Abhängigkeitsgraphen gewinnen kann. Da die Tabelleneinträge nun Attributexemplare statt Vorkommen repräsentieren, muss eine zweite *ID*-Spalte hinzugefügt werden, durch die Instanzen desselben Attributvorkommens unterschieden werden können. Der Aufbau erfolgt in drei sequentiell oder verzahnt auszuführenden Durchläufen:

Erzeugen der Attributexemplare
: Zuerst werden alle Objektinstanzen des Modells in beliebiger Reihenfolge betrachtet. Der Kontext jeder Instanz wird durch ihren Objekttyp bestimmt. Indem alle Einträge, die diesem Definitionskontext entsprechen, aus der Metaabhängigkeitstabelle in die Modelltabelle übernommen werden, entsteht eine Aufstellung der vorhandenen Attributexemplare. Jedesmal wenn dabei ein Kontext wiederholt identifiziert wird muss die darin zuletzt verwendete Instanz-ID inkrementiert werden.

Verknüpfung der Einträge gemäß ihrer Abhängigkeiten
: Im zweiten Schritt werden die Abhängigkeiten zwischen den Attributexemplaren hergestellt. Dazu werden die Argumente der Attributexemplare in ihrem jeweiligen Subkontext aus der Metaabhängigkeitstabelle ausgelesen und mit den Attributinstanzen im Modell abgeglichen. Auf diese Weise lässt sich für jedes Argument die betreffende Menge der Attributexemplare gewinnen. Da diese wiederum einzelne Tabelleneinträge adressieren, können die Argumente als Liste von *(Adressierung/Attributexemplar-ID)*-Tupeln angegeben werden. Können keine zugehörigen Instanzen identifiziert werden, so wird dies durch ein einziges Tupel angezeigt, das der Adressierung den Wert *null* zuweist.

Eintrag der Objekteigenschaften
: Die Objekteigenschaften werden ebenfalls als Tupel gespeichert, indem jeder Eigenschaft der aus dem Modell gewonnene konstante Wert zugewiesen wird.

[4] Wie in Abschnitt 2.2.4 für Attributgrammatiken dargelegt, kann diese Analyse auch vollständig dynamisch auf dem Modell durchgeführt werden. Zu beachten ist dabei, dass in diesem Fall automatische Ergänzungen der Attributierung, die z.B. die Attributweiterleitung in Generalisierungshierarchien sicherstellen, in die Modellanalyse integriert werden müssen.

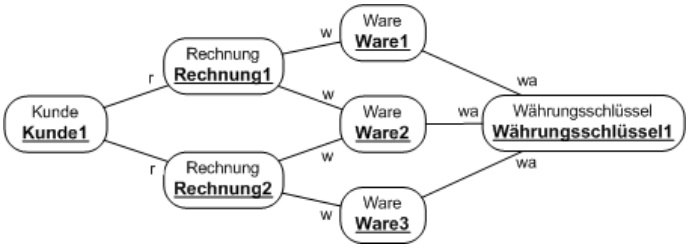


Abbildung 5.3: Ein Modell des Kunden/Umsatz-Referenzbeispiels aus Abschnitt 3.1

Das Ergebnis dieses Vorgehens für eine Instanz des Kunden-Metamodells 5.3 zeigt 5.4. Das aus dem Metamodell gewonnene Gleichungssystem wurde für die einzelnen Modellelemente instantiiert, wie exemplarisch an konkreten Elementen gezeigt wird:

- Die im Kontext *Rechnung* verfügbaren Attribute *Umsatz* (ID 3) und *Limit eingehalten* (ID 4) wurden für die Objekte *Rechnung1* und *Rechnung2* jeweils einmal mit unterschiedlicher Instanz-ID erzeugt.
- Um die Attributinstanzen für das Argument *Preis* zu erhalten, werden die Objektinstanzen im Subkontext *[w]* betrachtet. Dies sind *Ware1* und *Ware2* für *Rechnung1* bzw. *Ware2* und *Ware3* bei *Rechnung2*. Die IDs der zugehörigen *Preis*-Exemplare sind *5.1* und *5.2* bzw. *5.2* und *5.3*. Diese werden mit Subkontext in die Argumentliste des *Umsatz* Attributs eingetragen.
- Die Werte der Objekteigenschaften werden als Konstanten in die Tabelle übernommen.

Behandlung von Generalisierungen

Es ergeben sich aus der Vererbungssemantik durch Weiterleitung von Attributwerten innerhalb von Generalisierungshierarchien von Eltern- an Kindobjekte einige Spezialfälle, die gesondert betrachtet werden müssen.

Da die Supertypen einer spezialisierten Subtypinstanz nicht explizit im Modell enthalten sind, ist die Abhängigkeitsstruktur an diesen Objekten auf den ersten Blick unvollständig und muss deshalb während der Analyse ergänzt werden. Abbildung 5.5 verdeutlicht diese Problematik. In dem einfachen attributierten Metamodell 5.5(a) wird das Attribut *X* von *Objekt C* zu *Objekt A* geleitet, das *X* wiederum an seinen Subtyp *Objekt B* vererbt. Zwei Instanzen dieses Metamodells sind in 5.5(b) zu sehen. Die Auswertung des ersten Modells ist unproblematisch, da die bestehenden Attributierungsregeln die Abhängigkeiten zwischen *A* und *C* abdecken. Das zweite Modell enthält hingegen Subtyp *B*, dessen Interaktion mit *C* nur implizit in der Attributierung enthalten ist. Um dieses Problem zu umgehen, muss der Analysealgorithmus diese Objektinstanz als quasieigenständiges Modellobjekt simulieren, d.h. die Elterninstanz vom Typ *A* muss durch Generierung eines entsprechenden „Dummy-Objekts“ explizit gemacht werden. Die An-

ID1	ID2	Kontext	Attribut	Modellelement
1	1	Kunde	Umsatz	Kunde1
2	1	Kunde	Limit	Kunde1
3	1	Rechnung	Umsatz	Rechnung1
4	1	Rechnung	Limit eingehalten	Rechnung1
3	2	Rechnung	Umsatz	Rechnung2
4	2	Rechnung	Limit eingehalten	Rechnung2
5	1	Ware	Preis	Ware1
5	2	Ware	Preis	Ware2
5	3	Ware	Preis	Ware3
6	1	Währung	Schlüssel	Währungsschlüssel1

(a) Die Attributexemplare des Modells

ID	Attribute	Eigenschaften
1.1	r 3.1 r 3.2	
2.1		Limit *const*
3.1	w 5.1 w 5.2	
4.1	[loc] 3.1 2.1	
3.2	w 5.2 w 5.3	
4.2	[loc] 3.2 2.1	
5.1 5.2 5.3	wa 6.1 wa 6.1 wa 6.1	
6.1		Schlüssel *const*

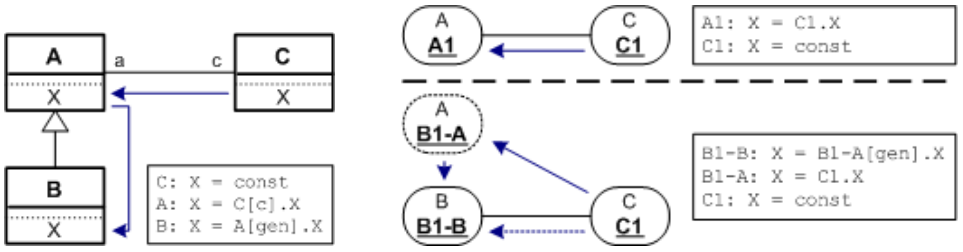
(b) Verknüpfung der Attributexemplare und Objekteigenschaften (nur die relevanten Objekteigenschaften werden angezeigt)

Abbildung 5.4: Abhängigkeitstabelle der Analyse auf Instanzebene (basierend auf dem Ergebnis der Metaabhängigkeitsanalyse aus Abbildung 5.1)

wendung dieser Technik für die gesamte Supertyphierarchie der Objekte ermöglicht es die bestehenden Attributierungsregeln unveränderten einzusetzen.

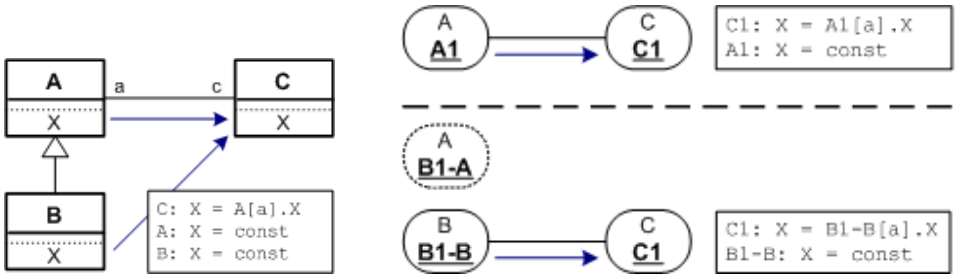
Ein weiteres Problem in diesem Kontext zeigt 5.6. *X* wird hier in entgegengesetzter Richtung von *Objekt A* bzw. *Objekt B*[5] zu *Objekt C* weitergeleitet. Unproblematisch ist wiederum die Auswertung des ersten Modells aus 5.6(b) während die Weiterleitung von *Objekt B* nach *Objekt C* jedoch durch keine Regel gewährleistet wird. Auch das „Dummy-Objekt" *B1-A*, das *Objekt B* in seiner Supertyp-Ausprägung repräsentiert, schafft an dieser Stelle keine Abhilfe. Vielmehr muss der Analysealgorithmus sicherstellen, dass beim Abgleich der Regelargumente mit den Instanzen im Modell auch Subtypen der in den Regeln definierten Kontexte berücksichtigt, und die Tabelleneinträge entsprechend mo-

[5] Eine Instanz von *Objekt B* überschreibt die Attributwerte seiner (impliziten) Elterninstanz.



(a) (b)

Abbildung 5.5: Die Generalisierungshierarchie muss auf Modellebene durch „Dummy-Objekte“ vervollständigt werden



(a) (b)

Abbildung 5.6: Attributierungsregeln gelten stets auch für Subtypen

difiziert, werden. Im Beispiel wird das Attribut *X* explizit von *Objekt A* angefordert. Da Kontext *Objekt B* aber in *Objekt A* enthalten ist, zählen dessen Instanzen zu den Argumenten dieser Regel.

Zyklentest auf Modellebene

Es wird davon ausgegangen, dass die Kompatibilität des annotierten Modells eine gültige Instanz seines definierenden Metamodells ist, und die Einhaltung der Attributierungssemantik bereits überprüft wurde. Damit stellt sich noch die Frage nach der Notwendigkeit eines Tests auf Zyklenfreiheit. Sofern das Metamodell zyklenfrei ist, kann auf diesen per Definition verzichtet werden. Ist dies nicht der Fall, so wäre eine Analyse auf Modellebene zwar denkbar, aber aufgrund der Anzahl der Einträge im Gleichungssystem, die ein Vielfaches der Anzahl der Einträge auf Metaebene umfassen kann, aber potentiell zu zeitaufwändig. Spielt dies keine Rolle, so kann der Test zeitgleich zur Berechnung der Attributwerte durchgeführt werden (siehe Abschnitt 5.2.1).

5.2 Attributauswertung

Nachdem durch die Abhängigkeitsanalyse die Voraussetzung für die Berechnung der Attributwerte in Form einer korrekten und vollständigen Beschreibung der Abhän-

gigkeitsbeziehungen geschaffen wurde, wird im zweiten Schritt ein konkreter Auswertungsalgorithmus auf diese Eingabedaten angewendet. Die Vorgehensweise dieses Algorithmus variiert je nach Eigenschaften des übergebenen Gleichungssystems und Grad der Optimierung.

Wie die Auswertung einer Attributierung in einer Art und Weise durchgeführt werden kann, die den Eigenschaften nicht-zyklischer Gleichungssysteme Rechnung trägt, zeigt Abschnitt 5.2.1. In 5.2.2 wird eine Variante dieser Methode vorgestellt, die unter bestimmten Voraussetzungen eine sinnvolle Interpretation zyklischer Definitionen ermöglicht.

5.2.1 Auswertung eines attributierten Modells

Liegt das Gleichungssystem der Attributierung vollständig vor, so kann darauf eine Auswertung erfolgen. Als deren Ergebnis steht ein bewertetes Modell, an dessen Objektinstanzen die Werte, die für die Attributexemplare berechnet wurden, sichtbar sind. Vorraussetzung für die erfolgreiche Auswertung ist das Vorliegen eines semantisch korrekten, nicht-rekursiven Gleichungssystems. Der Berechnungsvorgang selbst ist atomar, d.h. die Werte der Attributinstanzen und der Objekteigenschaften bleiben für eine Auswertung konstant. Der Algorithmus weist sowohl Eigenschaften von daten- als auch von bedarfsgetriebenen Strategien auf. Ähnlich zur bedarfsgetriebenen Auswertung erfolgt die Berechnung für ein gewähltes Attributexemplar, indem dessen Abhängigkeitsbeziehungen bis zu bereits bekannten Werten rekursiv zurückverfolgt werden. Von der datengetriebener Vorgehensweise wird die iterative Komponente übernommen, da die Berechnung wiederholt wird, bis Ergebnisse für alle Exemplare vorliegen.

Algorithmus zur Auswertung einer nichtzyklischen Attributierung

Bevor die Auswertung begonnen werden kann, muss die tabellarische Darstellung der Modellabhängigkeiten um eine Spalte erweitert werden, die die berechneten Ergebnisse aufnimmt. Da Zyklenfreiheit vorausgesetzt wird, ist sichergestellt, dass das Gleichungssystem eine oder mehrere, voneinander unabhängige und endliche Abhängigkeitsketten besitzt. Diese repräsentieren die Beziehungen zwischen den Exemplaren in Form eines Abhängigkeitsbaums, ausgehend vom zu berechnenden Wurzelelement hin zu bereits bekannten Werten an den Blättern. Da die Berechnung einen atomaren Prozess darstellt, ist es unerheblich ob die Auswertung bei dem tatsächlichen Wurzelelement einer Abhängigkeitskette beginnt oder bei einem seiner (in)direkten Kindknoten. In letzterem Fall zerfällt die Kette in einzelne Teilbäume, deren Auswertung getrennt erfolgt. Um die redundante Berechnung von bereits vorliegenden Werten zu vermeiden bietet es sich an, diese nicht in Baumform sondern als Gerichteter azyklischer Graph mit einem Wurzelelement (siehe Abbildung 2.2(b)) zu verarbeiten. Die Auswertung selbst erfolgt von den Blättern zur Wurzel, wobei sämtliche gewonnenen Werte - nicht nur das Endergebnis an

der Wurzel - sofort in die Ergebnisspalte der Tabelle übernommen werden. Als einfache Implementierung dieses Algorithmus bietet sich ein zweistufiges Vorgehen an:

1. Zunächst werden alle Einträge ausgewertet, die keine Abhängigkeiten besitzen oder ausschließlich von konstanten Werten, d.h. von Objekteigenschaften, abhängen. Die Ergebnisse werden in die Tabelle aufgenommen, womit sie als markiert/-berechnet gelten.

2. Die (Teil-)Abhängigkeitsketten werden dann durch Traversierung in Tiefensuchordnung berechnet, indem die Argumente jedes Knotens vor dessen Auswertung rekursiv ermittelt werden. Die Abhängigkeiten eines (zufällig) als Wurzelelement bestimmten Tabelleneintrags werden dazu rekursiv bis zu einem Knoten zurückverfolgt, dessen Eingabewerte bereits vollständig vorliegen. Dieser wird ausgewertet und gilt als markiert nachdem das Ergebnis in die Tabelle übertragen wurde. Der Algorithmus kehrt dann zum Vorgänger zurück und startet den Suchlauf für den nächsten unmarkierten Kindknoten erneut bis auch hier alle Argumentwerte vorliegen. Die Auswertung einer Abhängigkeitskette endet mit der Ermittlung des Wertes am Wurzelknoten. Aus der Menge der zu diesem Zeitpunkt unmarkierten Tabelleneinträge wird dann ein neues Wurzelelement gewählt und der Vorgang wiederholt bis der Wert jedes Attributexemplars berechnet wurde.

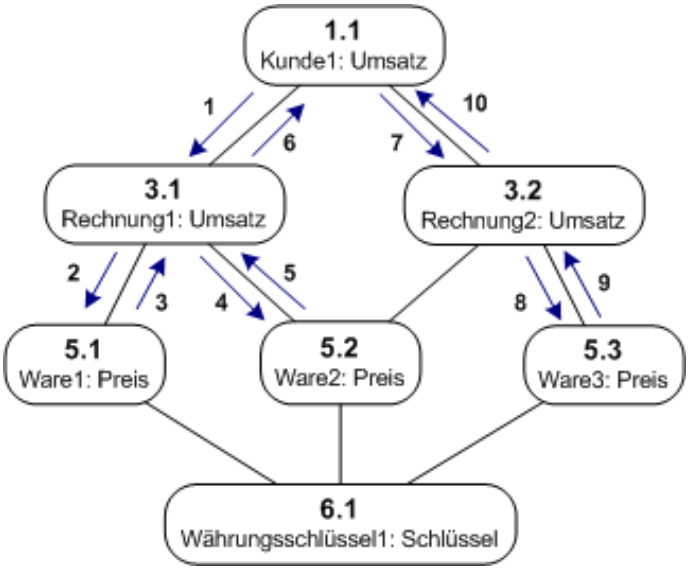


Abbildung 5.7: Attributauswertung am Beispiel des Attributexemplars *Umsatz* der Objektinstanz *Kunde1*

Im Beispiel aus Abbildung 5.7 wurde dieser Algorithmus für die Berechnung des Attributexemplars *1.1* angewendet. Die Wahl dieses Exemplars als Wurzelelement führt zur Betrachtung einer vollständigen Abhängigkeitskette, da es selbst nicht als Argument ei-

ner Attributierungsfunktion auftritt[6].

Es werden zuerst die Werte der Attribute bestimmt, die keine Abhängigkeiten aufweisen. Das betrifft im Beispiel unter anderem den *Währungsschlüssel*, der aus einer konstanten Objekteigenschaft ableitet wird. Danach startet die rekursive Nachverfolgung der Abhängigkeiten von *1.1*, die schließlich bei Knoten *5.1* endet, da dessen einziges Argument *6.1* bereits im ersten Schritt berechnet wurde. Der Knoten *5.1* wird ausgewertet und der Algorithmus kehrt zum Vorgänger *3.1* zurück. Da der Wert seines Arguments *5.2* noch unbekannt ist wird der Tiefensuchlauf für dieses gestartet. Nachdem für *5.2* das Ergebnis ermittelt wurde kann auch für *3.1* der Attributwert bestimmt und zum Vorgänger *1.1* zurückgekehrt werden, dessen noch nicht berechnetes Argument *3.2* als nächstes betrachtet wird. Aus dem Beispiel geht auch hervor, wie durch die sofortige Übertragung der Ergebnisse eine zweifache Berechnung von *5.2* vermieden wird.

Berechnung der Attributwerte auf Basis der Spezifikationen

Die Berechnung der Attributwerte erfolgt über die angegebenen Spezifikationen. Diese müssen in Abhängigkeit von ihrer Implementierung aufgerufen werden, wobei die Argumentwerte in dem durch die Spezifikationsarchitektur definierten Format zu übergeben sind. In Anknüpfung an das in 4.2.3 vorgestellte Implementierungskonzept, das die Argumente in Form von Hashtabellen erwartet, kann dies wie folgt geschehen:

1. Für jeden in der Argumentliste enthaltenen Subkontext wird eine Menge angelegt, die alle Werte dieses Subkontexts enthält.
2. Der Subkontext wird zur vollständigen Adressierung erweitert.
3. Die Wertemengen werden in der Hashtabelle abgelegt, die Adressierung fungiert dabei als Schlüssel.
4. Die Objekteigenschaften werden äquivalent in einer zweiten Hashtabelle gespeichert.
5. Aus der übergebenen Spezifikationsimplementierung wird die Methode aufgerufen, die der jeweiligen Spezifikations-ID des Attributs entspricht. Als Argumente werden die generierten Hashtabellen übergeben, der Rückgabewert der Methode entspricht dem Ergebnis der Auswertung.

Komplexität der Auswertung

Aufgrund der Zyklenfreiheit terminiert der Algorithmus in jedem Fall, da nach max.

[6] Wäre dagegen beispielsweise *3.1* als Wurzelelement gewählt worden, so würde die Auswertung zuerst für den Teilgraphen unter *3.1* erfolgen, während die weiteren Knoten dann erst in späteren Schritten betrachtet werden. Das Endergebnis wird dadurch nicht beeinflusst.

$n-1$ Suchvorgängen ein Eintrag mit konstantem Wert erreicht wird. Eine naive Auswertung, die die Attributliste wiederholt abgearbeitet und in jedem Durchlauf nur Einträge mit bekannten Argumentwerten berücksichtigt, hätte im schlechtesten Fall den Aufwand $O(\frac{n*(n+1)}{2})$ (falls in jedem Durchlauf nur ein Wert berechnet werden kann). In der Praxis kann aber davon ausgegangen werden, dass Attributexemplare durch die Weiterleitung von Werten innerhalb des Modells lange Abhängigkeitsketten bilden, wodurch eine redundante Betrachtung von Attributexemplaren vermieden werden kann. Der oben vorgestellte Algorithmus trägt dieser Eigenschaft Rechnung, indem in jedem Tiefensuchlauf möglichst viele Werte auf einmal ermittelt werden. Zusätzlicher Aufwand entsteht nur durch das Lokalisieren bereits berechneter Argumentknoten. Dementsprechend lässt sich ein (unrealistisches) worst-case Szenario konstruieren, bei dem nur der erste Tabelleneintrag frei von Abhängigkeiten ist und die übrigen Einträge von allen ihren jeweiligen Vorgängern abhängen. Auf die Betrachtung der n Attributexemplare im ersten Durchlauf folgt dann die Auswertung der restlichen $n-1$ Elemente, wobei insgesamt $\frac{n*(n-1)}{2}$ Suchvorgänge auf die Betrachtung bereits bekannter Argumente verwendet werden. Eine realistische Schätzung des durchschnittlichen Aufwands würde die Analyse einer nennenswerten Menge an Metamodellattributierungen voraussetzen. Die Erfahrungen, die im Rahmen dieser Arbeit gemacht wurden, deuten aber eher auf eine lineare Komplexität im praktischen Einsatz hin.

Verbindung mit der Abhängigkeitsanalyse auf Modellebene

Die Auswertungsphase ist konzeptionell sehr stark mit der Abhängigkeitsanalyse auf Modellebene verknüpft. Es ist möglich, beide Prozesse ineinander zu verzahnen, um die Effizienz zu steigern und die benötigten Laufzeitressourcen zu minimieren. Auch die Veränderung des Auswertungsalgorithmus zu einem rein bedarfsgetrieben Verfahren wäre denkbar, bei dem vom Benutzer einzelne Attribute zur Auswertung bestimmt werden. Ebenso kann die Auswertung auch mit einer Zyklenfreiheitsanalyse auf Modellebene verbunden werden, indem der Tiefensuchlauf um eine derartige Analysekomponente ergänzt wird.

5.2.2 Behandlung zyklischer Attributabhängigkeiten

Bisher wurde stets Wert auf die Zyklenfreiheit des Gleichungssystems der Attributierung gelegt, da zyklische Definitionen keine sinnvolle Auswertung erlauben (siehe Abschnitt 2.2.3). Diese Beschränkung wird jetzt fallengelassen, um eine Möglichkeit zur Analyse des Informationsflusses in Modellen zu implementieren. Die Auswirkungen auf die Attributierungssemantik und die notwendigen Anpassungen des Auswertungsalgorithmus sind Themen dieses Abschnitts.

Zyklische Abhängigkeiten

Enthalten die Attributexemplare zyklische Abhängigkeiten[7], so enden die in 5.2.1 postulierten Abhängigkeitsketten nicht in Blättern mit bereits bekannten Werten, sondern erreichen während des rekursiven Aufbaus der Kette bereits besuchte Knoten. Damit ist klar, dass rekursive Abhängigkeitsketten keine eindeutige Wurzel besitzen können, ihr Aufbau ist vielmehr von dem als Wurzelelement gewählten Element abhängig. Zyklen unterscheiden sich damit insofern von Rekursionen, als dass sie keinen klar definierten Eintrittspunkt und keine Abbruchsbedingung besitzen. Die Auswertung zyklischer Attributierungen erfordert daher die Festlegung einer eigenen Auswertungssemantik und eines daran angepassten Berechnungsalgorithmus.

Datenflussanalyse und Metamodelle

In Abschnitt 2.3 wurde als Technik zur Untersuchung der Informationsweitergabe in Programmen die Datenflussanalyse vorgestellt. Da die Entwicklung eines Attributierungskonzepts für Metamodelle nicht zuletzt den Anspruch hat, eine Technik zu entwerfen, die im besonderen Maße die syntaktische Struktur von Modellen berücksichtigt, bietet sich hier eine Interpretation zyklischer Abhängigkeiten an, die die Ausbreitung von Informationen im Modell entlang der Abhängigkeitsbeziehungen, äquivalent zum Kontrollfluss eines Programms, beschreibt. Im Vergleich zu Kontrollflussgraphen weisen Metaattributierungen allerdings keine expliziten Einstiegs- und Ausstiegspunkte auf. Vielmehr können mehrere unabhängige Schleifen und zyklenfreie Abhängigkeitsketten existieren, die jeweils einen eigenen Informationsfluss abbilden, dessen Flussrichtung von der Attributierung abhängt. Die Definition einer DFA ist damit hochgradig modell- und damit domänenspezifisch.

Anforderungen an den Auswertungsalgorithmus

Die Basis des Auswertungsalgorithmus bildet die in 5.1.2 beschriebene Vorgehensweise, die um die Funktionalität zur Abschätzung zyklischer Informationsflüsse erweitert wird. Da die Werte der Attributexemplare während der Auswertung nicht mehr konstant sein müssen, sondern über mehrere Berechnungsläufe betrachtet werden, ist es erforderlich die Struktur der Abhängigkeitskette bis zum Vorliegen des Endergebnisses aufrechtzuerhalten. Durch eine gesonderte Verwaltung der Rückwärtskanten kann so die für den Auswertungsvorgang benötigte Tiefenordnung[8] der Abhängigkeiten erzeugt werden. Rückwärtskanten repräsentieren Referenzen auf Knoten, deren Wert erst in der jeweils nächsten Iteration verfügbar ist. Da die Abhängigkeitskette in jedem Fall an Konstanten oder an durch Referenzen symbolisierten Rückkanten endet, ist sichergestellt,

[7] Diese können auch verschachtelt sein, so dass eine Kette mehrere Zyklen enthalten kann.

[8] Da die Auswertung bottom-up erfolgt, handelt es sich im Prinzip um eine umgekehrte Darstellung des tatsächlichen Informationsflusses.

dass für einen beliebig gewählten Ausgangsknoten ein endlicher Abhängigkeitsgraph in Tiefenordnung erstellt werden kann, welcher den Kontrollfluss der DFA ersetzt.

Wie beschrieben, ist die Flussrichtung bereits implizit durch das Gleichungssystem in Form einer Attributierung vorgegeben, so dass eine gesonderte Betrachtung der Analyserichtung nicht erforderlich ist. Die Berechnung selbst wird durch eine wiederholte Auswertung der Tiefenordnung erreicht, wobei nach jeder Iteration die aktuellen Werte entlang, der durch Referenzen symbolisierten, Rückkanten propagiert werden.

Es stellt sich weiterhin die Frage, ob die Wahl des Repräsentanten einer zyklischen Abhängigkeitskette in Form des Wurzelelements Einfluss auf das Endergebnis hat. Per Definition müssen die Ketten unabhängig vom gewählten Wurzelelement alle Attributexemplare des Zyklus enthalten, Unterschiede in der Menge der enthaltenen Knoten sind nur für nichtzyklische Bestandteile und abgekoppelte Zyklen möglich. Dieser Unterschied ist aber vernachlässigbar, da diese in einem späteren Auswertungsschritt mit den jeweiligen Endergebnissen berechnet werden. Der Fixpunkt ist unabhängig von der Wahl des Wurzelelements, so dass diese keinen Einfluss auf das Endergebnis hat.

Konfluenzoperatoren

Die Datenflussgleichungen werden durch die Spezifikationen implementiert. Der Konfluenzoperator, der den zu berechnenden Fixpunkt als Infimum oder als Supremum definiert, ist damit ebenfalls Teil der Attributierung und hat keine Auswirkungen auf den Auswertungsalgorithmus. Eine Besonderheit ergibt sich nur bei der Initialisierung, da hierbei je nach Typ des zu berechnenden Fixpunkts entweder die leere Menge oder die gesamte Wertedomäne eingesetzt werden muss. Um dies vollständig in der Attributierung zu codieren, wird festgelegt, dass Referenzen durch den Auswerter initial mit dem reservierten Wert *ref* belegt werden, der innerhalb der Spezifikationen dann je nach Kontext entweder als $\emptyset$ oder als Domäne D_a interpretiert wird.

Eine Spezifikation zur Berechnung des Supremums hat demnach die Form:

1. Erzeuge eine Liste aller Mengen, die in den Argumenten übergeben wurden.
2. Bilde die Vereinigung dieser Mengen, behandle dabei *ref* als leere Menge.
3. Entferne lokal zerstörte Informationen und füge neu generierte Informationen hinzu.
4. Liefere die Menge als Ergebnis zurück.

Das Infimum kann wie folgt implementiert werden:

1. Erzeuge eine Liste aller Mengen, die in den Argumenten übergeben wurden.
2. Bilde den Schnitt dieser Mengen, behandle dabei *ref* als Domäne (d.h. ein Schnitt mit *ref* erhält die Ursprungsmenge).

3. Entferne lokal zerstörte Informationen und füge neu generierte Informationen hinzu.

4. Liefere die Menge als Ergebnis zurück.

Auswertungsalgorithmus für zyklische Attributierungen

Der Auswertungsalgorithmus für zyklische Abhängigkeiten:

1. Äquivalent zur nichtzyklischen Auswertung werden in einem ersten Durchlauf alle Attributexemplare berechnet, die keine Abhängigkeiten besitzen.

2. Für einen gewählten Eintrag wird dessen Tiefendarstellung generiert. Treten dabei keine Zyklen in Erscheinung, kann der Standardalgorithmus angewandt und mit der Wahl des nächsten Exemplars fortgefahren werden. Falls jedoch Rückkanten gefunden werden, so werden diese nicht eingetragen. Stattdessen wird eine Referenz des Ursprungsknotens angelegt, der der Ergebniswert *ref* zugeordnet wird.

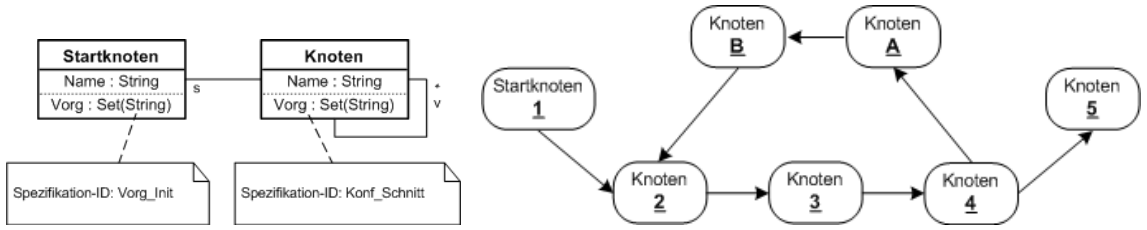
3. Die Auswertung erfolgt nun nach dem Standardalgorithmus, wobei die Referenzen als eigenständige Knoten betrachtet werden.

4. Die Werte der referenzierten Knoten werden nun auf die Referenzen übertragen, d.h. entlang der Rückkanten propagiert. Sind alle aktuellen Werte mit den Referenzwerten der letzten Iteration identisch, so wurde der Fixpunkt erreicht. In diesem Fall terminiert die Berechnung für das gewählte Attributexemplar und die Endwerte werden in die Ergebnistabelle übertragen. Hat sich jedoch ein Wert geändert, so beginnt die Auswertung beim vorherigen Schritt erneut.

Dieser Algorithmus hat den Vorteil, dass er weiterhin kompatibel mit der Auswertung nichtzyklischer Abhängigkeiten ist. Zyklische und nichtzyklische Definitionen können daher auch gemeinsam in einer Attributierung auftreten. Durch die Implementierung der Datenflussgleichungen in Spezifikationen kann der Informationsfluss zudem, z.B. durch attributabhängige *gen*/*kill* Mengen, sehr detailliert festgelegt werden. Weiterhin ist auch eine Kombination von Supremum/Infimum-Berechnungen denkbar.

Im Gegensatz zur nichtzyklischen Auswertung werden durch den vollständigen Aufbau der Tiefenordnung und durch mehrere Durchläufe pro betrachteter Abhängigkeitskette jedoch mehr Ressourcen verbraucht, was durch entsprechende Optimierungen allerdings zum Teil kompensiert werden könnte.

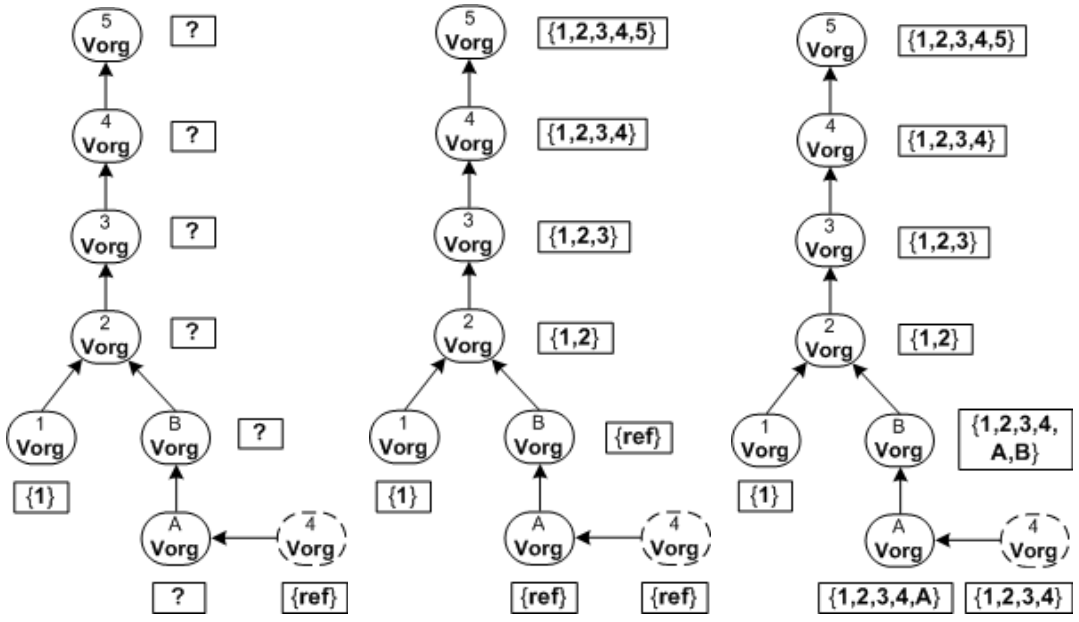
Beispiel für eine zyklische Auswertung



(a) Zyklische Metaattributierung (b) Modell mit Zyklus

Abbildung 5.8: Zyklische Metaattributierung und Modell

Abbildung 5.8(a) zeigt ein Metamodell, das ein stark vereinfachtes Ablaufdiagramm beschreibt, dessen Knoten ausgehend von einem initialen Startknoten über eine Vorgängerrelation verbunden sind. Das Attribut *Vorg* wird zu Beginn mit der leeren Menge initialisiert und soll nach der Berechnung an jedem Knoten die Menge der Schritte enthalten, die zwingend gemacht werden müssen um diesen Knoten zu erreichten. Da es sich dabei um die Analyse des größten Fixpunkts in Vorwärtsrichtung handelt, wurde dieses Attribut mit einer Spezifikation versehen, die den entsprechenden Konfluenzoperator (Schnitt) implementiert (siehe Anhang C.2). In 5.8(b) ist ein entsprechendes Modell zu sehen, das einen Zyklus enthält.



(a) Initiale Werte (b) Erster Durchlauf (c) Zweiter Durchlauf

Abbildung 5.9: Iterative Fixpunktberechnung auf der Attributierung

Der Fortschritt des Auswertungsprozesses ist in Abbildung 5.9 zu sehen. Die Tiefenordnung wurde für das Wurzelelement *5* aufgebaut, wobei die Rückkante von *4* nach *A* durch einen Referenzknoten ersetzt wurde, der mit dem Wert *ref* initialisiert wird. Wei-

terhin kann der Wert für *1* berechnet werden, da hier keine Abhängigkeiten vorliegen. Im ersten Durchlauf werden die Knoten im Standardverfahren bottom-up ausgewertet, wobei der Wert *ref* von der Schnittoperation in Knoten *2* als gesamte Wertedomäne interpretiert wird. Zu Beginn des nächsten Durchlaufs wird der, für Knoten *4* berechnete, Wert auf den Referenzknoten übertragen und die Auswertung erneut durchgeführt. Da der Wert des Knotens *4* nun mit dem in der Referenz gespeicherten Wert übereinstimmt wurde der Fixpunkt erreicht, da bei einer erneuten Auswertung das Ergebnis konstant bliebe. Dasselbe Endergebnis würde auch bei der Wahl eines beliebigen anderen Wurzelknotens erreicht (siehe Anhang C.3).

6 Implementierung des Anwendungsbeispiels

Nachdem in den Kapiteln 4 und 5 die Attributierung eines Metamodells und die Auswertung auf Modellebene in der Theorie besprochen wurden, werden diese Techniken auf den in Abschnitt 3.2 beschriebenen Anwendungsfall angewandt. Dazu werden zunächst die Anforderungen an eine entsprechende Attributierung des JWT-Metamodells festgelegt, bevor dann das Basiskonzept für deren Umsetzung erläutert und die Implementierung auf den einzelnen Metaobjekten definiert wird.

6.1 Theoretische Aspekte

Die Attributierung erfolgt auf einer, für das gewählte Anwendungsszenario relevanten, Teilmenge der JWT-Metaklassen. Grundlegende Bedingung für eine erfolgreiche Analyse ist die Beachtung der semantischen Beschränkungen durch die gegebenen JWT Modelle. Dazu zählen unter anderem das Einhalten der definierten Multiplizitäten, eine korrekte Schachtelung verzweigter Abläufe und die Existenz jeweils genau eines Einstiegs- und Ausstiegspunkts. Da die Überprüfung dieser Eigenschaften auf Basis einer Attributierung zwar prinzipiell möglich, aber nicht Teil der in 3.2 formulierten Aufgabenstellung ist, wird für das Eingabemodell ein syntaktisch und semantisch korrekter Aufbau vorausgesetzt.

Aktionen des Geschäftsprozesses werden durch ihren Namen identifiziert. Verschiedene Aktionsinstanzen an unterschiedlichen Positionen im Prozess repräsentieren dieselbe Aufgabe, sofern ihre Namen identisch sind. Diese semantische Äquivalenz muss durch das Analyseverfahren implementiert werden.

Da Geschäftsprozesse Zyklen in Form von Rückkanten enthalten können, muss ein entsprechend angepasster Auswertungsalgorithmus zur Anwendung kommen. Weil die Ausbreitung von Informationen über die Struktur des Modells untersucht werden soll, bietet sich der Einsatz der in 5.2.2 eingeführten Technik zur Datenflussanalyse an. Die Datenflussgleichungen sind dabei an die konkrete Problemstellung anzupassen. An jedem der betrachteten Modellknoten müssen demnach folgende Datenflussmengen ausgewertet werden:

MinNS ***(Minimal Node Set)***

Bei $MinNS$ handelt es sich um die Menge der (Vorgänger-)Aktionen, die *mindestens einmal* ausgeführt wurden, wenn dieser Knoten erreicht wird (handelt es sich bei dem betrachteten Knoten um eine Aktion, so ist diese ebenfalls enthalten). Das Ergebnis $MinNS\_O$ wird aus den Eingabemengen $MinNS\_I_i$ der i Vorgängerknoten ermittelt. Ziel ist die Berechnung des größten Fixpunkts durch Anwendung von Schnittbildung als Konfluenzoperator. Eine Ausnahme bildet die Zusammenführung paralleler Abläufe, da hier immer alle Pfade beschritten werden. Die Initialisierung an durch Zyklen induzierten Referenzen erfolgt durch die gesamte Domäne (alle Aktionen).

AltNS ***(Alternative Node Set)***

Enthalten alle Wege zum Zielknoten bedingte Verzweigungen, ist es also zwingend notwendig auf dem Weg zum Finalknoten zwischen verschiedenen Ablaufpfaden zu wählen, so werden die *spezifischen Aktionen der alternativen Ablaufpfade* in der Menge $AltNS$ gesammelt. Diese ist damit eine Sammlung der Knotenmengen, von denen *jede* das minimale Set $MinNS$ um einen der möglichen Alternativpfade zu einem vollständigen und minimalen Ablauf ergänzt. Bei sequentiell bzw. verschachtelt angeordneten Verzweigungen erweitern sich die Wahlmöglichkeiten, weswegen die Kombination der in $AltNS$ enthaltenen Mengen durch Bildung des kartesischen Produkts erfolgt. Der Initialwert an Zyklen entspricht der Menge, die als einziges Element die leere Menge enthält.

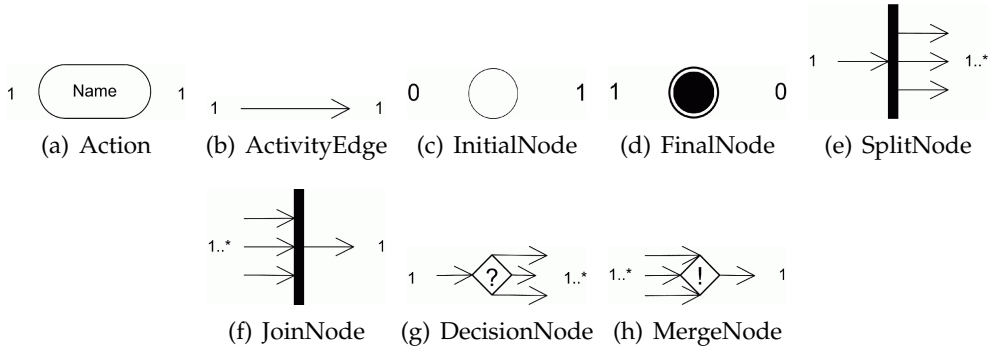
FinNS ***(Final Node Set)***

Das kartesische Produkt der Aktionsmenge $MinNS$, das die auf allen Pfaden durchzuführenden Aktionen enthält, mit den Aktionsmengen aus $AltNS$, von denen jede die Aktionen beschreibt, die auf einer der möglichen minimalen Ablaufvarianten erreicht werden, ergibt die Menge aller minimalen Prozessdurchläufe $FinNS$. Da die Berechnung ausschließlich im lokalen Kontext erfolgt, treten keine zyklischen Abhängigkeiten auf.

Basierend auf diesen Datenflussmengen lassen sich die 5 in Abschnitt 3.2 formulierten Ziele wie folgt erreichen:

1. Die bei jedem Ablauf auszuführenden Aktionen an einem beliebigen Punkt des Geschäftsprozesses entsprechen der Menge $MinNS\_O$ am jeweiligen Modellelement.
2. Über die Attributierung können auf der Knotenmenge $MinNS\_O$ Bedingungen festgelegt werden, die das Enthaltensein einer vorgegebenen Aktion fordern.
3. Die minimal zu beschreitenden alternativen Pfade werden in den $AltNS$ Mengen gesammelt und über die Berechnung von $FinNS$ in das Endergebnis einbezogen.
4. Das Ergebnis der Analyse für den gesamten Prozess lässt sich am *FinalNode* des Modells ablesen.

5. An den Mengen *MinNS*, *AltNS* und *FinNS* der verschiedenen Verzweigungs- und Zusammenführungsknoten lässt sich erkennen, welche Aktionen zum jeweiligen Zeitpunkt in jedem Fall oder aber nur unter bestimmten Bedingungen durchgeführt werden. Aus einer paarweisen Betrachtung zusammengehöriger Knoten lassen sich mögliche Modifikationen des Ablaufs ableiten. Eine Aktion, die an einem *SplitNode* nicht, beim entsprechenden *ForkNode* aber auf jeden Fall erreicht wurde, könnte so beispielsweise vor oder hinter diesen parallelen Ablauf verschoben werden.



(a) Action (b) ActivityEdge (c) InitialNode (d) FinalNode (e) SplitNode

(f) JoinNode (g) DecisionNode (h) MergeNode

Abbildung 6.1: Die attributierten Elemente des JWT-Metamodells (Darstellung mit Eingangs- und Ausgangsmultiplizitäten)

Die beschriebenen Datenflussmengen *MinNS*, *AltNS* und *FinNS* berechnen sich für die relevanten Metaobjekte (Abbildung 6.1) wie folgt:

InitialNode

$$MinNS\_O = \{\}$$
$$AltNS\_O = \{\{\}\}$$

Der Initialknoten stellt den Einstieg in den Prozess dar und initialisiert deshalb beide Datenflussmengen.

ActivityEdge, SplitNode, DecisionNode und FinalNode

$$MinNS\_O = MinNS\_I_0$$
$$AltNS\_O = AltNS\_I_0$$

Diese Elemente verfügen über jeweils genau einen Vorgängerknoten i_0, dessen Ergebnisse $MinNS\_I_0$ und $AltNS\_I_0$ sie unverändert übernehmen.

Action

$$MinNS\_O = MinNS\_I_0 \cup Action.Name$$
$$AltNS\_O = AltNS\_I_0$$

Die Aktion übernimmt ebenfalls die Ausgabe des (einzigen) Vorgängers i_0, fügt sich aber zusätzlich selbst zur Menge der minimal besuchten Knoten $MinNS$ hinzu.

JoinNode

$$MinNS\_O = \bigcup_{i \in Vorg}(MinNS\_I_i)$$
$$AltNS\_O = \bigcup_{i \in Vorg}(AltNS\_I_i)$$

Bei der Zusammenführung paralleler Abläufe erfolgt eine Vereinigung der Minimal- und Alternativmengen der i Vorgänger, da mit dem Erreichen der Zusammenführung alle parallelen Ausführungspfade beschritten wurden.

MergeNode

$$MinNS\_O = \bigcap_{i \in Vorg}(MinNS\_I_i)$$
$$Kompl_i = MinNS\_I_i \setminus MinNS\_O$$
$$wenn\ (\forall i \in Vorg:\ Kompl_i \neq \{\})$$
$$\ dann\ AltNS\_O = \bigcup_{i \in Vorg}(AltNS\_I_i \times Kompl_i)$$
$$\ sonst\ AltNS\_O =$$
$$\bigcup_{i \in Vorg \wedge Kompl_i = \{\}}(AltNS\_I_i)$$

Bei der Zusammenführung alternativer Ablaufwege muss davon ausgegangen werden, dass nur einer dieser Pfade durchlaufen wurde. Aus diesem Grund berechnet sich die Menge der minimal, d.h. auf jedem Pfad, besuchten Knoten $MinNS$ durch Schnittbildung der Vorgänger-Actionsets. Die Ermittlung der alternativen Aktionsmengen gestaltet sich aufwändiger, da die notwendige Vereinigung der Alternativen aller eingehenden Pfade dazu führen würde, dass diese auch über Rückkanten propagiert würden. Dadurch würden dann anstatt den *notwendigen* Alternativen (größter Fixpunkt) alle *möglichen* Alternativen (kleinster Fixpunkt) berechnet. Um dies zu verhindern, müssen Ergebnisse, die über Rückkanten erhalten werden, verworfen werden, was eine Methode zur Zyklenerkennung innerhalb der Datenflussgleichungen erfordert. Zur Ermittlung der Alternativen werden zunächst die Komplementmengen $Kompl_i$ durch Subtraktion des minimalen Sets $MinNS\_O$ von allen Eingabemengen $MinNS\_I_i$ berechnet. Standardmäßig wird $Kompl_i$ (die für den Pfad i charakteristischen Aktionen) mit jeder der bereits bestehenden Alternativmengen aus $AltNS\_I_i$ durch Bildung des kartesischen Produkts kombiniert. Alle diese Mengen werden zum Endergebnis $AltNS\_O$ zusammengefasst. Existiert jedoch mindestens ein Komplement $Kompl_i$, das der leeren Menge entspricht, so handelt es sich bei allen Vorgängern mit nicht leeren Komplemen-

ten vermutlich[1] um Rückkanten, da der minimale Pfad $MinNS\_O$ vollständig in einem der eingehenden Pfade $MinNS\_I_i$ enthalten ist. In diesem Fall werden nur die Mengen $AltNS\_I_i$ mit leerem $Kompl_i$ in die Berechnung einbezogen (da $Kompl_i = \{\}$ kann auf Bildung des kartesischen Produkts verzichtet werden).

Alle Knoten

$$FinNS = MinNS\_O \times AltNS\_O$$

Die Menge aller minimalen, vollständigen Ausführungsalternativen $FinNS$ berechnet sich an allen Knotentypen lokal aus dem kartesischen Produkt des größten Fixpunktes ($MinNS\_O$) und den in $AltNS\_O$ enthaltenen Aktionsmengen.

6.2 Implementierung des Auswertermoduls in Java

Da der Anwendungsfall eine Auswertung von JWT-Prozessen erfordert, welche auf einem EMF-Metamodell basieren, liegt es nahe das Auswertermodul ebenfalls auf dieses Framework abzustimmen[2].

Architektur

Abbildung 6.2 zeigt die Architektur der Attributierungsplattform. Das unabhängige Auswertermodul wird über eine Plugin-Schnittstelle in JWT eingebunden, das selbst wiederum ein Plugin der Eclipse-Plattform darstellt. Die JWT-Schnittstelle übernimmt die Steuerung des Auswerters und versorgt diesen außerdem mit den Modelldaten des ausgewählten JWT-Prozesses. Die Attributierung, inklusive der Spezifikationsimplementierungen, wird dagegen vom Attributauswerter selbst geladen.

Der interne Aufbau des Auswerters ist in Abbildung 6.3 dargestellt. Um eine generische Testplattform zur Verfügung zu stellen, wurde die Gesamtfunktionalität in einzelne Teilfunktionen aufgespalten. Diese kommunizieren über vorgegebene Schnittstellen und sind dadurch beliebig austauschbar.

[1] Diese Methode zur Zyklenerkennung versagt in einigen Sonderfällen, da relevante Pfade evtl. nicht mit Aktionen markiert sind bzw. Aktionen dieselben Namen besitzen. Dennoch ist sie für die meisten Szenarien anwendbar, weshalb sie im Rahmen dieses Beispiels als ausreichend betrachtet werden kann. Eine zuverlässige, aber umständlichere Lösung zur Zyklenerkennung wäre die Berechnung des kleinsten Fixpunkts auf allen Knoten, wobei jeder Knoten durch eine eindeutige ID repräsentiert wird.

[2] Weitere Softwarebibliotheken, auf die bei der Implementierung zurückgegriffen wurde, umfassen JavaCSV (http://sourceforge.net/projects/javacsv/) zum Parsen der Attributierung, sowie JGraph (http://www.jgraph.com/) und JGraphT (http://jgrapht.sourceforge.net/) zur Visualisierung der internen Berechnungsvorgänge.

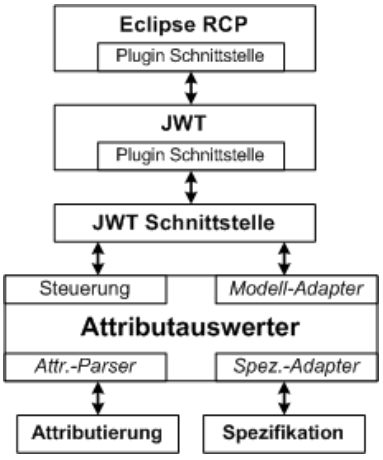


Abbildung 6.2: Architektur der Auswertungsplattform

Attributierungsparser

So ist beispielsweise das *Attributierungsparser*-Modul dafür zuständig die Attributierung einzulesen und in die interne Darstellung als *Regelsatz* zu überführen. In der prototypischen Implementierung übernimmt diese Aufgabe ein einfacher CSV-Parser, der die Attributierung in einer, ähnlich der in Abschnitt 5.1.1 gezeigten, tabellarischen Form erwartet. Denkbar sind aber auch Parsermodule, die Attributierungen direkt aus Metamodellen oder in der in 4.2.4 vorgestellten Notation einlesen.

Modelladapter

Der *Modelladapter* ist dafür zuständig aus dem Modell eine Liste mit allen auszuwertenden EMF-Objekten zu generieren. Der hier implementierte *Modelladapter* liest dazu alle Objekte aus der vom Benutzer ausgewählten Aktivität aus und übergibt diese an den weiteren Auswertungsvorgang.

Spezifikationsadapter

Der *Spezifikationsadapter* ist dagegen für die Anbindung der Spezifikationen zuständig und muss eine Methode zur Verfügung stellen, um eine Spezifikation über den übergebenen Namen aufzurufen und ihr die bereitgestellten Argumente zu übermitteln, sowie das Ergebnis zurückzuliefern. In der Beispielimplementierung übernimmt diese Aufgabe ein Modul, das Spezifikationen in Form von Java-Klassen über das Reflectioninterface lädt und Spezifikationsimplementierungen durch Methodenaufruf aufführt.

Instantiierungsmodul

Der nächste Schritt im Auswertungsvorgang wird vom *Instantiierungsmodul* durchgeführt, das den Regelsatz auf die übergebenen Modellobjekte anwendet, um die in 5.1.2 beschriebene Tabelle, den Instanzsatz, zu erzeugen. Die prototypische Implementierung beherrscht dabei bereits alle grundlegenden Attributierungskonzepte bis auf die Unterstützung von Generalisierungen, Typisierung und dem Subkon-

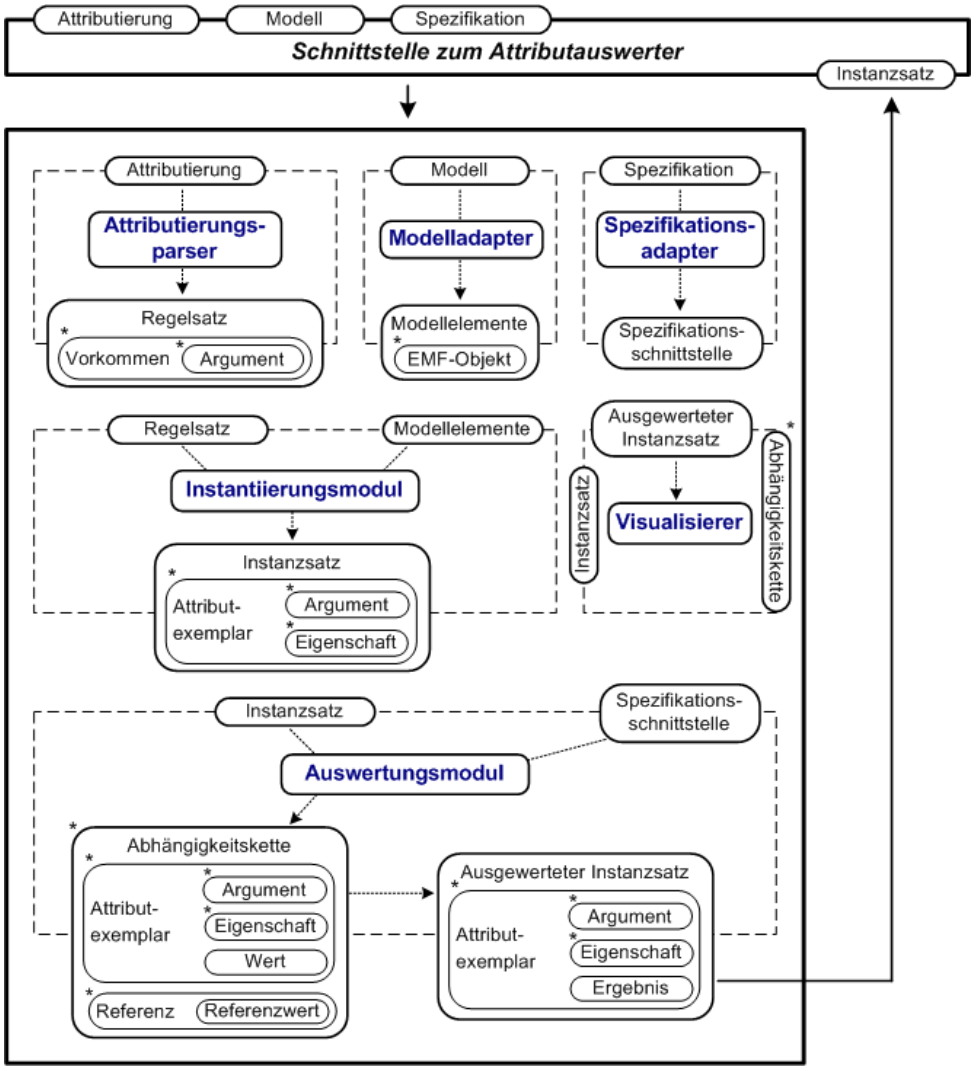


Abbildung 6.3: Interner Aufbau des Auswertermoduls

text [all].

Auswertermodul

Spezifikationsadapter und Instanzsatz bilden dann die Eingaben für das eigentliche Auswertermodul, das die Attributexemplare wie in Abschnitt 5.2 beschrieben durch Rückverfolgung ihrer Abhängigkeiten und Aufruf der Spezifikationen in der entsprechenden Reihenfolge auswertet. Die konkrete Implementierung unterstützt sowohl die normale als auch die zyklische Auswertung und verfügt über die beschriebenen Optimierungsmechanismen.

Ergebnis und Visualisierer

Als Ergebnis wird ein Instanzsatz zurückgeliefert, der für alle Attributexemplare die berechneten Endwerte enthält. Zusätzlich können die Zwischenergebnisse und weitere Debuginformationen von einem Visualisierer graphisch aufbereitet werden. Im konkreten Fall werden die einzelnen Schritte des Auswertungsprozesses

inklusive aller Zwischenergebnisse (Abhängigkeitsketten in tabellarischer und graphischer Form) im HTML-Format dokumentiert.

Einsatz

Die Anwendung des Auswertermoduls wird im Weiteren anhand zweier einfacher Beispiele demonstriert.

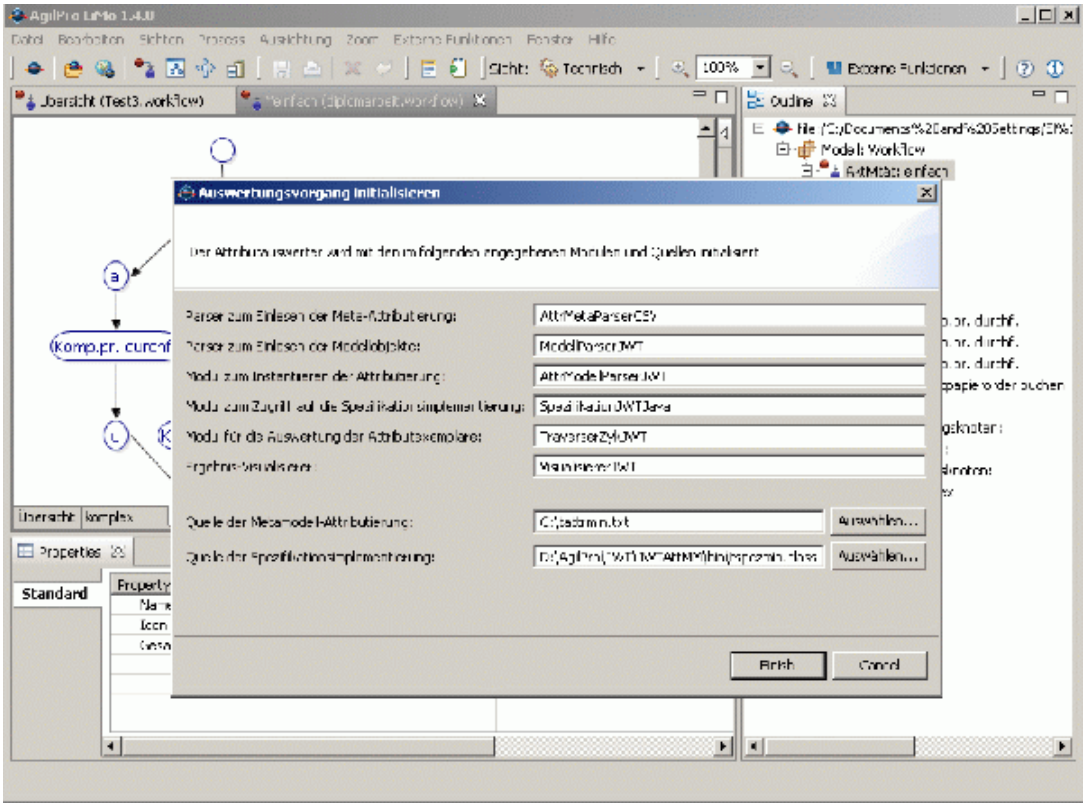


Abbildung 6.4: Integration des Auswertermoduls in den JWT Workflow Editor

Abbildung 6.4 zeigt die Integration des Auswerters in Form eines Plugins für den Workflow Editor des JWT Projekts. Wird das Plugin über seinen Menüeintrag aktiviert, so erfolgt zunächst über den dargestellten Dialog die Konfiguration des Auswerters. Diese Konfiguration umfasst die Angabe konkreter Implementierungen der einzelnen Teilmodule, sowie die Quelle der Attributierung und ihrer Spezifikationen. Im konkreten Beispiel ist die Attributierung in Tabellenform (CSV Format) gegeben und wird von einem entsprechenden Attributierungsparser eingelesen. Die Spezifikationen liegen als Methoden in einer Java-Klasse vor, die vom Spezifikationsadapter über das Java Reflection Interface aufgerufen werden. Das zu analysierende Modell entspricht dem im graphischen Editor aktuell geladenen Prozess, dessen Elemente (EMF-Objekte) dem Modelladapter von der JWT Pluginschnittstelle automatisch als Liste übergeben werden.

Der Ablauf des Auswertungsprozesses wird in einem HTML-Log detailliert festgehalten. Die Initialisierung des Auswertermoduls und der durch den Attributierungsparser eingelesene Regelsatz für die Analyse der alternativen Prozessabläufe ist in Abbildung 6.5 zu sehen. Die für die Berechnung der Fixpunktmenge *MinNS* (größter Fixpunkt) zuständigen Methoden der Spezifikationsimplementierung zeigt Abbildung 6.6. Dabei wurden einige DFA-Standardfunktionen, beispielsweise die Schnittbildung auf den Eingabemengen, in gesonderte Methoden ausgelagert (siehe Anhang D.1).

Ziel ist es nun, die Analyse für den abstrakten Beispielprozess aus Abbildung 6.7(a) durchzuführen. Ausgehend von einem Entscheidungsknoten sind drei verschiedene Abläufe möglich, wobei jeder Ablauf die Aktion *Kompetenzprüfung durchführen* enthält, woraufhin dann ein Wertpapierordner gebucht wird. Die Aktionen *a* und *c* sind in zwei der drei Abläufe enthalten, allerdings in verschiedener Reihenfolge. Im dritten Ablauf finden sich stattdessen die Aktionen *b* und *d*.

Die vom Auswerter im ersten Schritt vorgenommene Instantiierung der Attributierung auf dem Modell ist als Log-Ausschnitt in Anhang D.2) abgebildet. Das Auswertungsergebnis ist in Abbildung 6.7(b) dargestellt, wobei die Menge $FinNS$ vom Visualisier an den Kontrollknoten des Prozesses annotiert wurde. Wie erwartet ergeben sich dabei zwei alternative Abläufe: Ohne Aussage über die konkrete Ausführungsreihenfolge bilden *a* und *c* die charakteristischen Aktionen des einen Pfades (d.h. der linke und der rechte Ausführungspfad sind bezüglich der aufgerufenen Aktionen äquivalent), während *d* und *b* den zweiten möglichen Ablauf charakterisieren. Eine weitere Information, die sich aus dem Ergebnis ableiten lässt, ist dass die Aktion *Kompetenzprüfung durchführen* vor der Verzweigung nicht, nach der Zusammenführung aber in jedem Fall ausgeführt wurde.

Abbildung 6.8(a) zeigt einen weiteren Prozess, der im Gegensatz zum vorangehenden Beispiel einen Zyklus enthält. Aus dem Auswertungsergebnis aus Abbildung 6.8(b) lassen sich nun unter anderem folgende Informationen ableiten:

- Aktion *x* wird in jedem beliebigen Prozessdurchlauf mindestens einmal ausgeführt
- Die Aktionen *1*, *3* und *x* bilden den einzigen minimalen Ablaufpfad
- *x* muss ebenfalls durchlaufen werden, um in den Zyklus einzutreten
- Wurde der Zyklus betreten, so wird immer auch *a* ausgeführt
- Aktion *a* wird in jedem Fall vor Aktion *b* ausgeführt

In Abbildung 6.9 ist die automatisch generierte und mittels des JGraphT-Frameworks visualisierte Abhängigkeitskette für das Attribut *MinNS* am *FinalNode* zu sehen. Das gewählte Wurzelelement ist dabei schwarz gefärbt und die Endpunkte (Blätter) des Abhängigkeitsbaums dunkelblau, sofern es sich um Konstanten handelt und rot bei an zy-

klischen Abhängigkeiten erzeugten Referenzen. 6.9(a) zeigt den initialen Status dieser Kette und 6.9(b) das Endergebnis nach der dritten Iteration[3].

[3] Der Ablauf der Tiefensuche für die erste Berechnungsiteration ist in Anhang D.3 dargestellt, die Ergebnisse nach Iteration 1 und 2 in Anhang D.4

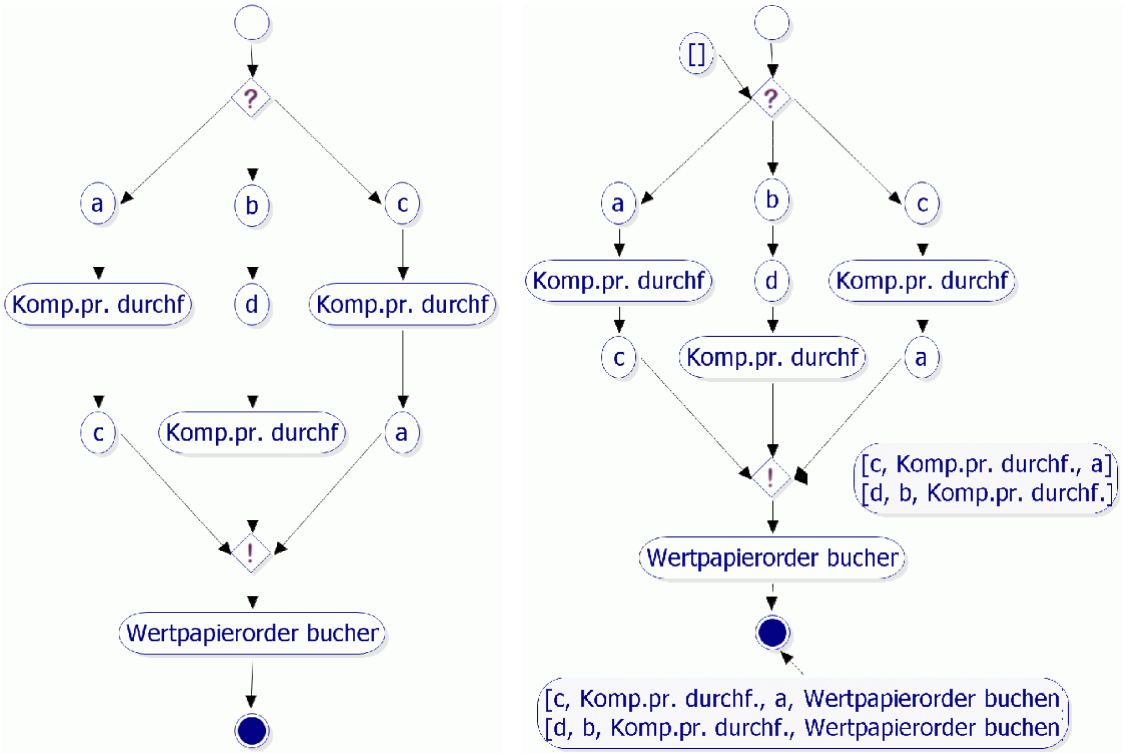
```
[Auswerter] Eingabe   Container MetaModell:
org.eclipse.jwt.we.model.processes.impl.ActivityImpl@10b23cf (name:
einfach, icon: ) (totalexecutiontime: 0)
[Auswerter] Modul geladen - Parser Metaattributierung: AttrMetaParserCSV
[Auswerter] Modul geladen - Parser Modellattributierung:
AttrModellParserJWT
[Auswerter] Modul geladen   Parser EMF Modellobjekte: ModellParserJWT
[Auswerter] Modul geladen   Spezifikationscontainer: SpezifikationJWTJava
[Auswerter] Modul geladen - Traverser: TraverserZykJWT
[Auswerter] Eingabe - Attributierungsquelle: C:\lattrmin.txt
[Auswerter] Eingabe   Spezifikations Quelle: D:\AgilPro\JWT\cWTAttMM
\bin\testspez3.class
[Auswerter] Auswertungsvorgang gestartet
[AttrMetaParserCSV] Attributierungsregeln einlesen
```

Der Regelsatz der Attributierung definiert 22 Attributvorkommen				
ID	**Kontext**	**Attribut**	**Spezifikation**	**Argumente**
0	Action	MinNS	*minns_action*	(1:in)
1	ActivityEdge	MinNS	*minns_transfer*	(0:source) (2:source) (3:source) (4:source) (5:source) (6:source) (7:source)
2	InitialNode	MinNS	*minns_init*	
3	FinalNode	MinNS	*minns_transfer*	(1:in)
4	ForkNode	MinNS	*minns_transfer*	(1:in)
5	JoinNode	MinNS	*minns_join*	(1:in)
6	DecisionNode	MinNS	*minns_transfer*	(1:in)
7	MergeNode	MinNS	*minns_merge*	(1:in)
8	Action	AltNS	*altns_transfer*	(9:in)
9	ActivityEdge	AltNS	*altns_transfer*	(8:source) (10:source) (11:source) (12:source) (13:source) (14:source) (15:source)
10	InitialNode	AltNS	*altns_init*	
11	FinalNode	AltNS	*altns_transfer*	(9:in)
12	ForkNode	AltNS	*altns_transfer*	(9:in)
13	JoinNode	AltNS	*altns_join*	(9:in)
14	DecisionNode	AltNS	*altns_transfer*	(9:in)
15	MergeNode	AltNS	*altns_merge*	(16:in) (7:loc)
16	ActivityEdge	AltNSloc	*altnsloc_init*	(9:loc) (1:loc)
17	ForkNode	FinNS	*finnsloc*	(12:loc) (4:loc)
18	JoinNode	FinNS	*finnsloc*	(13:loc) (5:loc)
19	DecisionNode	FinNS	*finnsloc*	(14:loc) (6:loc)
20	MergeNode	FinNS	*finnsloc*	(15:loc) (7:loc)
21	FinalNode	FinNS	*finnsloc*	(11:loc) (3:loc)

Abbildung 6.5: Ausschnitt aus dem Analyse-Log: Initialisierung des Auswertermoduls und Einlesen des Regelsatzes

```
/**
 * InitialNode>MinNS: Initialisieren mit leerer Menge
 */
public static Object minns_init(HashMap<String, HashSet> attribute,
        HashMap<String, Object> eigenschaften)
{
    return new HashSet();
}


/**
 * SplitNode/DecisionNode/FinalNode/ActivityEdge>MinNS: MinNS-Menge des (einzigen)
 * Vorgängers weiterleiten.
 */
public static Object minns_transfer(HashMap<String, HashSet> attribute,
        HashMap<String, Object> eigenschaften)
{
    return DFA_transfer(attribute, "MinNS");
}


/**
 * Action>MinNS: ActionName hinzufügen und weiterleiten
 */
public static Object minns_action(HashMap<String, HashSet> attribute,
        HashMap<String, Object> eigenschaften)
{
    // Name der Aktion
    String actionName = (String) eigenschaften.get("name");

    // MinNS-Menge des (einzigen) Vorgängers
    Object minNS_in_0 = DFA_transfer(attribute, "MinNS");

    // Falls Eingabe nicht -REF- ist: Aktion zum Ergebnis hinzufügen
    if (minNS_in_0 instanceof Set)
    {
        ((Set) minNS_in_0).add(actionName);
    }

    return minNS_in_0;
}


/**
 * JoinNode>MinNS: Mengen der Vorgänger vereinigen
 */
public static Object minns_join(HashMap<String, HashSet> attribute,
        HashMap<String, Object> eigenschaften)
{
    return DFA_vereinigung(attribute, "MinNS");
}


/**
 * MergeNode>MinNS: Eingabemengen aller Vorgänger schneiden
 */
public static Object minns_merge(HashMap<String, HashSet> attribute,
        HashMap<String, Object> eigenschaften)
{
    return DFA_schnitt(attribute, "MinNS");
}
```

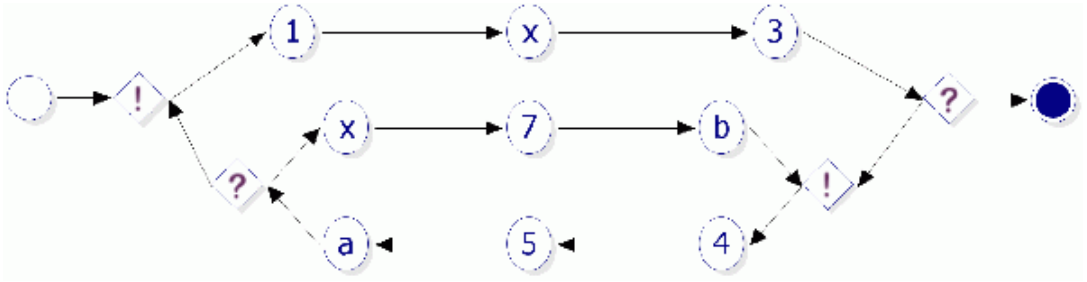
Abbildung 6.6: Auszug aus der Spezifikationsimplementierung zur Berechnung der Fixpunktmenge *MinNS*



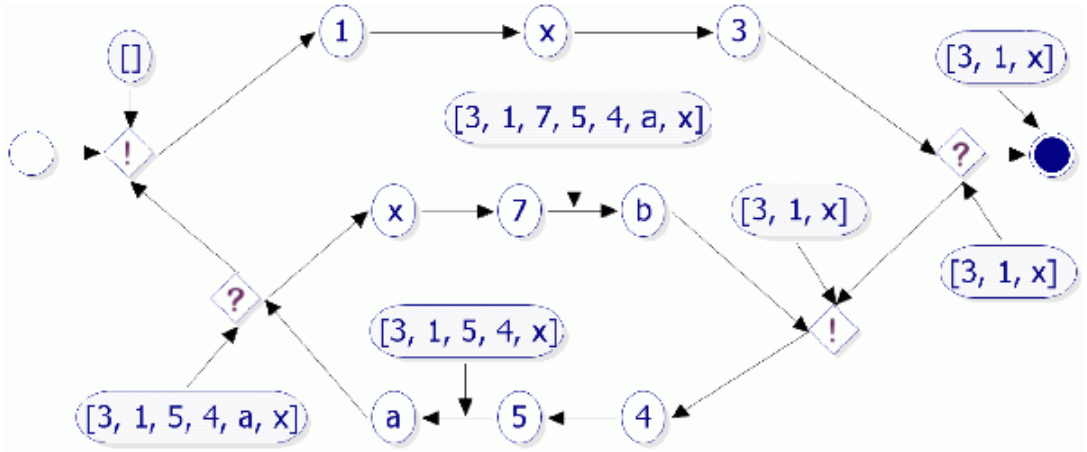
(a) Ein Geschäftsprozess mit alternativen Abläufen

(b) Der Geschäftsprozess mit den visualisierten Auswertungsergebnissen

Abbildung 6.7: Ein Geschäftsprozess mit alternativen Abläufen, für den das Ergebnis der Auswertung an den Kontrollknoten notiert wurde

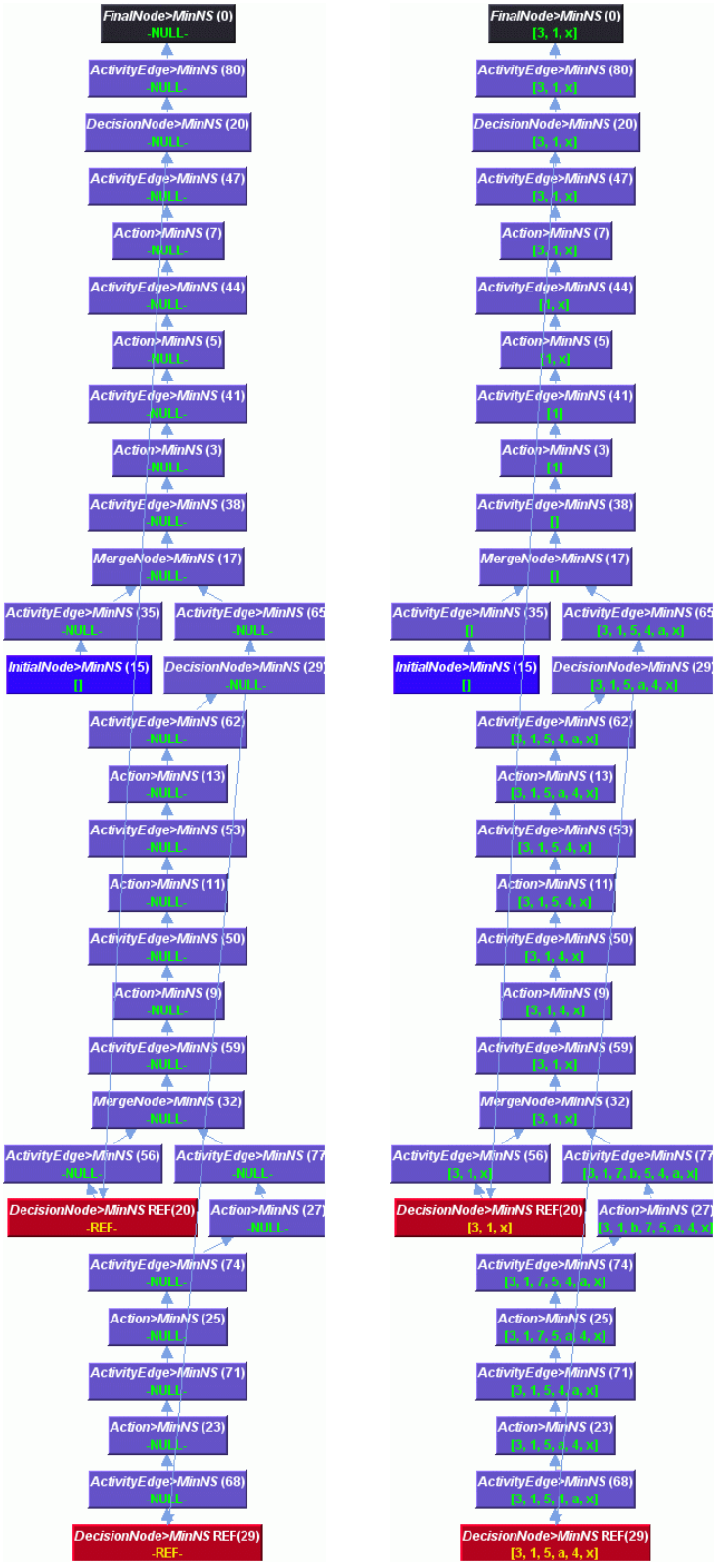


(a) Ein Geschäftsprozess mit einem zyklischen Ablauf



(b) Die visualisierten Ergebnisse der Auswertung

Abbildung 6.8: Ein Geschäftsprozess mit einem zyklischen Ablauf und dessen Auswertung



(a) Initiale Abhängigkeitskette (b) Fixpunkt nach dritter Iteration

Abbildung 6.9: Visualisierung einer automatisch generierten und ausgewerteten Abhängigkeitskette mittels JGraphT

7 Zusammenfassung und Ausblick

7.1 Zusammenfassung

In dieser Arbeit wurde eine Attributierungstechnik eingeführt, die eine dynamische Analyse von UML-Modellen erlaubt. Zielsetzung war, eine einfache und intuitive Methode zur Struktur- und Verhaltensanalyse von Modellen zu entwickeln, deren Ausdrücke nicht an die statische Struktur der Metamodelle gebunden sind und damit nicht die Limitationen der Object Constraint Language aufweisen. Zusätzlich sollte ein Verfahren zur Analyse des Informationsflusses in Modellen geschaffen werden.

Nach Beschreibung der grundlegenden Konzepte - Metamodellierung, attributierte Grammatiken und Datenflussanalyse - wurden zunächst die Parallelen zwischen formalen Grammatiken und Metamodellen, beides Techniken zur Formalisierung einer Sprachsyntax, aufgezeigt. Darauf aufbauend wurden Attributierungselemente in eine reduzierte Version der MOF-Metasprache eingefügt. Zur Unterstützung des Modellierungsvorgangs wurde für diese Elemente dann eine graphische sowie eine textuelle Notation vorgestellt. Die Einführung einer dezidierten Spezifikationssyntax ermöglicht es die semantischen Regeln flexibel mit ihren entsprechenden Berechnungssemantiken zu versehen. Um eine sinnvolle Interpretation der beschriebenen Syntaxelemente zu erreichen, wurde deren Semantik aus einem Vergleich mit attributierten Grammatiken abgeleitet, wobei die besondere Aufmerksamkeit der Adressierung der Attributvorkommen/-exemplare galt. Da zur Auswertung eine genaue Kenntnis der vorliegenden Abhängigkeitsstruktur erforderlich ist, musste zunächst ein Verfahren zur Abhängigkeitsanalyse auf Meta- und Modellebene entwickelt werden, durch das die Attributierung instantiiert, validiert und in eine für die weitere Verarbeitung vorteilhafte Darstellung überführt werden kann. Der eigentliche Auswertungsalgorithmus wurde in einer einfachen Form für nichtzyklische Abhängigkeiten, und in einer für die Bedürfnisse einer Datenflussanalyse auf Modellen angepassten Version vorgestellt. Diese Methoden wurden auf das Fallbeispiel angewandt, dessen Verhaltensanalyse dann auf einer Implementation eines generischen Attributauswerters für EMF-Modelle ausgeführt wurde.

7.2 Ausblick

Aus dem anfangs an diese Arbeit gestellten Anspruch, die Möglichkeiten sowie die Herausforderungen einer Attributierungstechnik für Metamodelle aufzuzeigen, lässt sich ableiten, dass eine Vielzahl von Ansatzpunkten zur Erweiterung und Ergänzung der vorgestellten Konventionen existiert.

Um den Einsatz auf breiter Basis zu gewährleisten, wäre es beispielsweise erforderlich das Attributierungskonzept auf die gesamte MOF auszudehnen und es mit einer umfangreichen Typisierung zu versehen. Die Ergänzung um weitere Restriktionstypen wie Guards und Pre-/Postconditions schließt dabei die (noch) bestehende Lücke zu OCL-Ausdrücken. Daneben steht die Entwicklung einer plattformunabhängigen Spezifikationssyntax, durch die Spezifikationsimplementierungen leichter beschrieben und direkt in die Attributierung integriert werden können. Ebenfalls kann der Spezifikationsaufwand durch eine Erweiterung der Standardbibliothek um häufig verwendete Funktionen reduziert werden. Hier kann auch ein Reflectionmechanismus entwickelt werden, der innerhalb der Spezifikationen Zugriff auf die Metadaten der Attributierung ermöglicht. Auch die Adressierungstechnik bietet Raum für Erweiterungen, beispielsweise durch spezifischere Subkontext-Selektoren oder aber durch das Verwenden einer OCL-basierten Adressierungssyntax. Nicht zuletzt stellt in der Praxis auch die Performanz des Auswertungsalgorithmus eine optimierungsrelevante Komponente dar.

Eine standardisierte Programmierschnittstelle zu Laufzeitsystemen eröffnet weitere Anwendungsszenarien für attributierte Metamodelle. Beispielsweise kann von Programmen auf die Ergebnisse des Auswertungsvorgangs zugegriffen werden, um diese etwa in Produktivsystemen weiterzuverarbeiten. Durch einen syntaxgesteuerten prozeduralen Aufruf ließen sich weiterhin Modelltransformationsverfahren implementieren. Werden für Attribute zusätzlich Metadaten hinterlegt, so kann die Attributierung beispielsweise in einzelne Layer gegliedert, die Sichtbarkeit individuell festgelegt, einzelne Attribute zur Auswertung ausgewählt oder Prioritäten auf die Einhaltung der Restriktionen verteilt werden.

Die Entwicklung einer, auf das Erstellen und Editieren von Attributierungen spezialisierten, Anwendung kann deren Handhabung dabei komfortabler gestalten. Attributierungen, die ein Modell auf seinem Metamodell validieren oder eine generische Datenflussanalyse durchführen, ließen sich automatisiert generieren. Auch bei der häufig anzutreffenden Attributweiterleitung kann ein Softwarewerkzeug den Modellierer durch automatisches Erstellen und Visualisieren von Transferregeln unterstützen.

Generell lässt sich festhalten, dass die Attributierung von Metamodellen einen vielversprechenden Ansatz zur Erweiterung der abstrakten Syntax um semantische Elemente darstellt.

Literaturverzeichnis

[AEU88a] AHO, Alfred V. ; ETHIS, Ravi ; ULLMANN, Jeffrez D.: *Compilerbau*. Bd. 1. Addison-Wesley, 1988. – ISBN 3–89319–150–X. – German translation of "Compilers, Principles, Techniques and Tools"

[AEU88b] AHO, Alfred V. ; ETHIS, Ravi ; ULLMANN, Jeffrez D.: *Compilerbau*. Bd. 2. Addison-Wesley, 1988. – ISBN 3–89319–150–X. – German translation of "Compilers, Principles, Techniques and Tools"

[AP04] ALANEN, Marcus ; PORRES, Ivan: A Relation between Context-Free Grammars and Meta Object Facility Metamodels / TUCS. 2004. – Forschungsbericht

[BH98] BAUER, Bernhard ; HÖLLERER, Riitta: *Übersetzung objektorientierter Programmiersprachen*. Springer, 1998. – ISBN 978–3540642565

[GE99] GÜTING, Ralf H. ; ERWIG, Martin: *Übersetzerbau. Techniken, Werkzeuge, Anwendungen*. Springer, 1999. – ISBN 978–3540653899

[Gor79] GORDON, Michael J. C.: *The Denotational Description of Programming Languages: An Introduction*. Secaucus, NJ, USA : Springer-Verlag New York, Inc., 1979. – ISBN 0387904336

[Gro06a] GROUP, Object M.: Meta-Object Facility 2.0 Specification / Object Management Group. Version: January 2006. http://www.omg.org/docs/formal/06-01-01.pdf. 2006 (Version 2.0). – Specification

[Gro06b] GROUP, Object M.: OCL Specification, v2.0 / Object Management Group. Version: Mai 2006. http://www.omg.org/cgi-bin/doc?formal/2006-05-01. 2006 (Version 2.0). – Specification

[Gro07a] GROUP, Object M.: Unified Modeling Language 2.1.2 Infrastructure Specification / Object Management Group. Version: November 2007. http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF/. 2007 (Version 2.1.2). – Specification

[Gro07b] GROUP, Object M.: Unified Modeling Language 2.1.2 Superstructure Specification / Object Management Group. Version: November 2007. http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF/. 2007 (Version 2.1.2). – Specification

[Job92] JOBST, Fritz: *Compilerbau: Von der Quelle zum professionellen Assemblertext*. Hanser, 1992. – ISBN 3446160957

[JOR75] JAZAYERI, Mehdi ; OGDEN, William F. ; ROUNDS, William C.: The intrinsically exponential complexity of the circularity problem for attribute grammars. In: *Commun. ACM* 18 (1975), Nr. 12, S. 697–706. http://dx.doi.org/http://doi.acm.org/10.1145/361227.361231. – DOI http://doi.acm.org/10.1145/361227.361231. – ISSN 0001–0782

[Kas90] KASTENS, Uwe: *Übersetzerbau*. Oldenbourg, 1990. – ISBN 978–3486207804

[KV97] KÜHNEMANN, Armin ; VOGLER, Heiko: *Attributgrammatiken. Eine grundlegende Einführung*. Vieweg, 1997. – ISBN 978–3528055820

[Med00] MEDUNA, Alexander: *Automata and Languages: Theory and Applications*. Springer, 2000. – ISBN 978–1852330743

[Pen94] PENNER, Volker: *Konzepte und Praxis des Compilerbaus*. Vieweg, 1994. – ISBN 978–3528054151

[RCA00] REGGIO, G. ; CERIOLI, M. ; ASTESIANO, E.: *An algebraic semantics of UML supporting its multiview approach*. 2000

[VT04] VOLKER TURAU, Friedrich V.: *Algorithmische Graphentheorie*. Oldenbourg, 2004. – ISBN 978–3486200386

[WG98] WAITE, William M. ; GOOS, Gerhard: *Compiler Construction*. Springer, 1998. – ISBN 978–3540642565

[Wik08] WIKIPEDIA: *Meta Object Facility — Wikipedia*. http://de.wikipedia.org/wiki/Meta_Object_Facility. Version: 2008. – [Online; Stand 4. Juli 2008]

[WM92] WILHELM, Reinhard ; MAURER, Dieter: *Übersetzerbau - Theorie, Konstruktion, Generierung*. Springer, 1992. – ISBN 3–540–55704–0

[WS07] WERNER, Dieter ; SCHNEIDER, Uwe: *Taschenbuch der Informatik*. 6. Fachbuchverlag Leipzig, 2007

[Zie06] ZIEMANN, Paul: *An Integrated Operational Semantics for a UML Core Based on Graph Transformation*. Logos Berlin, 2006. – ISBN 978–3832510718

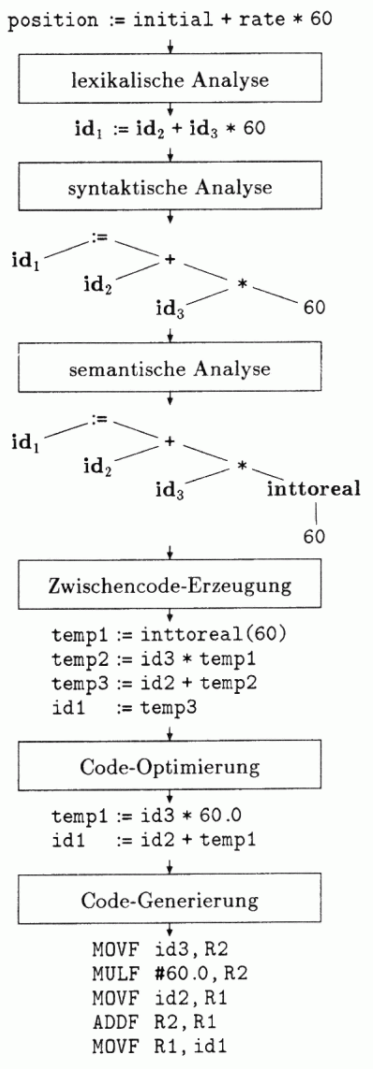
Abbildungsverzeichnis

Abkürzungsverzeichnis

AG Attributgrammatik
AMM Attributiertes Metamodell
BNF Backus Naur Form
BPMN Business Process Modeling Notation
CSV Comma Separated Value
DFA Datenflussanalyse
DSL Domänenspezifische Sprache (Domain-specific language)
DSM Domänenspezifisches Modell
EBNF Erweiterte Backus Naur Form
EMF Eclipse Modeling Framework
GAG Gerichteter azyklischer Graph
GEF Graphical Editing Framework
JWT Java Workflow Tooling
KFG Kontextfreie Grammatik
MDA Model Driven Architecture
MOF Meta Object Facility
OCL Object Constraint Language
OMG Object Management Group
RPC Remote Procedure Call
RFC Requests for Comments
UML Unified Modeling Language
XMI XML Metadata Interchange

A Attributgrammatiken

A.1 Compilerphasen



Quelle: [AEU88a, S. 15]

Eine Anweisung durchläuft die verschiedenen Phasen des Kompilierungsvorgangs. Die Bezeichner werden phasenübergreifend in einer Symboltabelle verwaltet (nicht abgebildet).

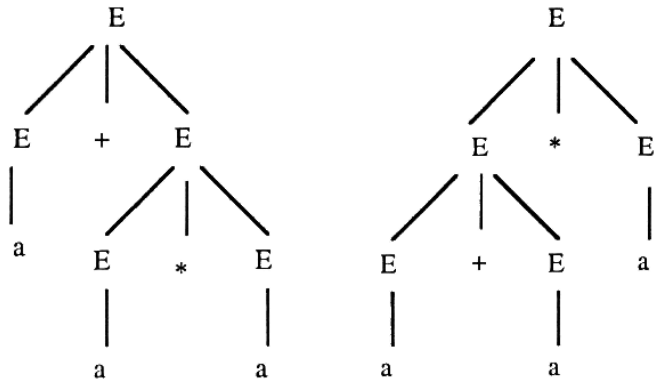
A.2 Mehrdeutige kontextfreie Grammatiken

$$E \rightarrow E+T \mid T$$
$$T \rightarrow T*F \mid F$$
$$F \rightarrow (E) \mid a$$

(a) Mehrdeutige kontextfreie Grammatik

$$a + a * a$$

(b) Ein Wort der Grammatik



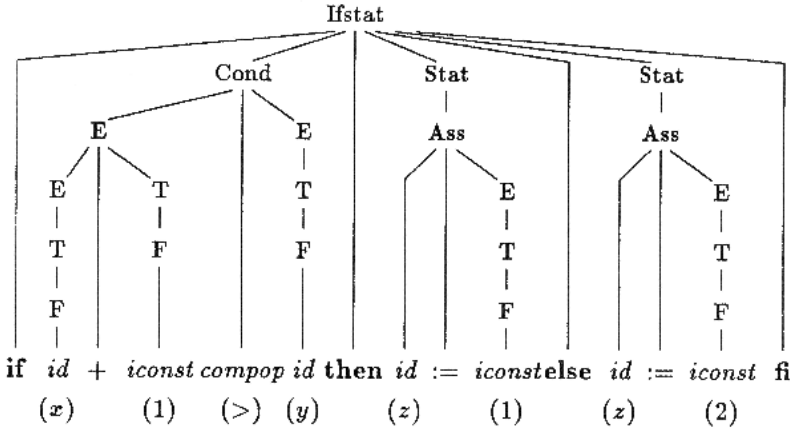
(c) Zwei Ableitungsbäume, die jeweils eine andere Interpretation desselben Wortes beinhalten. Dies führt dazu, dass die Prioritäten zwischen „+“ und „*“ nicht korrekt berücksichtigt werden.

Quelle: [Pen94, S. 49f]

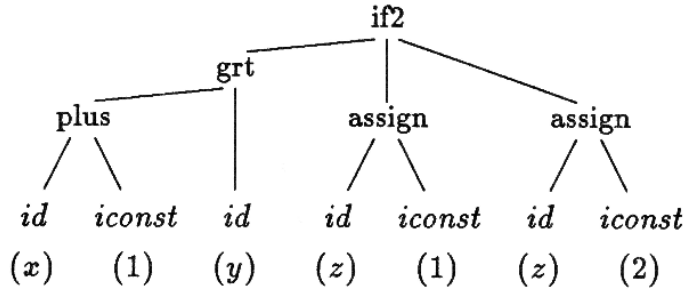
A.3 Konkrete und Abstrakte Syntax

```
if  x + 1 > y
then  z := 1
else  z := 2
fi
```

Programmausschnitt einer fiktiven Sprache



Parsebaum der konkreten Syntax (für die entsprechende Grammatik). Die konkrete Syntax orientiert sich bei der Herleitung am Aufbau der Grammatik.



Der Syntaxbaum orientiert sich an der abstrakten Syntax. Diese erfasst die wesentliche Struktur der Programmkonstrukte.

Quelle: [WM92, S. 355]

A.4 Eine Notation für Attributgrammatiken

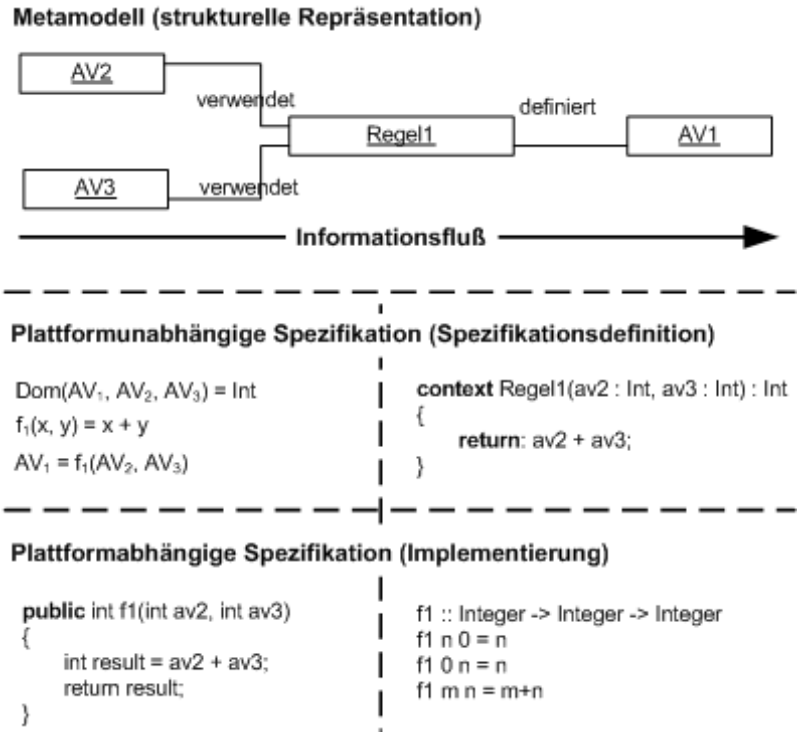
AttrGramm	⟶	*GrammName; Nonts; Attrs; Rules; Functions*
GrammName	⟶	**attribute grammar** *Name*
Nonts	⟶	**nonterminals** *NonEmptyNamList*
Attrs	⟶	**attributes** *AttrDeclList*
AttrDeclList	⟶	*AttrDeclList AttrDecl* | ε
AttrDecl	⟶	*Direction Name* **with** *NonEmptyNamList DomSpec*
Direction	⟶	**inh** | **syn**
DomSpec	⟶	**domain** *Domain*
Rules	⟶	**rules** *RuleList*
RuleList	⟶	*RuleList Rule* | *Rule*
Rule	⟶	*CfRule SemRuleList*
CfRule	⟶	*Name ' → ' NamList*
SemRuleList	⟶	*SemRuleList SemRule* | ε
SemRule	⟶	*Attr ' = ' Name (AttrList)* | *Attr ' = ' Attr* | *Attr ' = ' Const*
AttrList	⟶	*AttrList Attr* | *Attr*
Attr	⟶	*Name.Name* | *Name Index.Name*
Functions	⟶	**functions** *FunctDefList*
FunctDefList	⟶	*FunctDefList FunctDef* | ε
NamList	⟶	*NonEmptyNamList* | ε
NonEmptyNamList	⟶	*NonEmptyNamList Name* | *Name*

Quelle: [WM92, S. 375f]

Grammatik einer Deklarationssprache für Attributgrammatiken in EBNF-Form.

B Metamodellattributierung

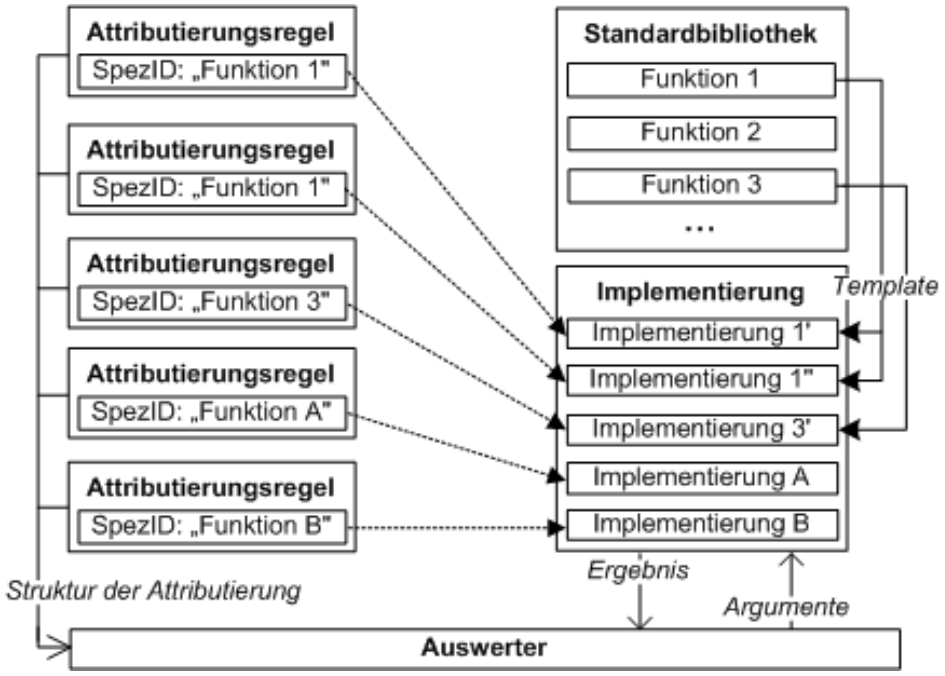
B.1 Spezifikationsdefinitionen und Implementierung



Das Metamodell definiert die Attributierungsregel *Regel1*, die die Attributvorkommen *AV2* und *AV3* als Argumente erhält und daraus einen Wert für *AV1*, hier die Summe von *AV2* und *AV3*, berechnet. Das Ziel ist, die Implementierungsunabhängigkeit des Metamodells zu erhalten.

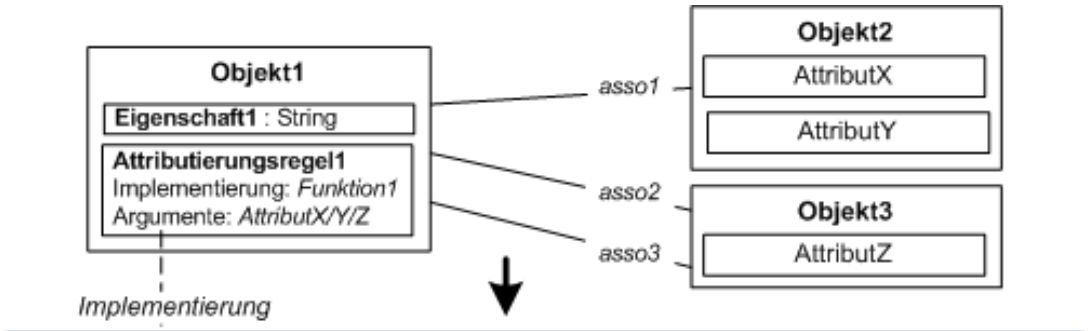
Eine Spezifikation, die in einem plattformunabhängige Format (im Beispiel als mathematische Funktion und in einem an OCL angelehnten Pseudocode) formuliert wurde, kann direkt im Metamodell abgelegt werden. Konkrete plattformabhängige Implementierungen, beispielsweise in Java oder Haskell, sollen dagegen außerhalb des Metamodells definiert werden.

B.2 Einbindung von Funktionen einer Standardbibliothek in die Implementierung



Die Implementierung setzt sich aus manuell definierten Funktionen und für den jeweiligen Einsatzkontext instantiierten Templates der Standardbibliothek zusammen. Die *SpezifikationID* der Attributierungsregeln verweist auf die Implementierung der Funktionen. Der Attributauswerter ruft die Funktionsimplementierungen gemäß der Attributierungsstruktur auf und übergibt diesen die benötigten Eingabewerte und weist das Funktionsergebnis den internen Attributvariablen zu.

B.3 Automatische Generierung eines Java-Methodenskeletts

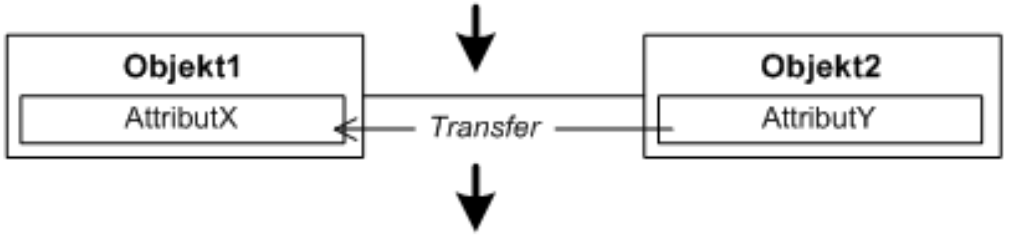



```
public Object Funktion1(Hashtable attr, Hashtable prop)
{
    // read object properties from hash table
    String Eigenschaft1 = (String)prop.get("Objekt1.Eigenschaft1");

    // read object attributes from hash table
    Set Objekt2_AttributX = (Set)attr.get("Objekt2.AttributX");
    Set Objekt2_AttributY = (Set)attr.get("Objekt2.AttributY");

    Set Objekt3_AttributZ = (Set)attr.get("Objekt3.AttributZ");
    Set Objekt3_asso2_AttributZ = (Set)attr.get("Objekt3[asso2].AttributZ");
    Set Objekt3_asso3_AttributZ = (Set)attr.get("Objekt3[asso3].AttributZ");

    $CODE$

    return $RESULT$;
}
```

Um den Attributierungsvorgang effizienter zu gestalten, kann für Attributierungsregeln automatisch ein leeres Methodenskelett erzeugt werden, das alle verfügbaren Attribute und Objekteigenschaften, die unter den definierten Subkontexten verfügbar sind, aus den übergebenen Tabellen extrahiert. Die Objekteigenschaft *Eigenschaft1* wird im Beispiel automatisch in der gleichnamigen Variablen abgelegt. Für die Attribute werden entsprechend Sets definiert, die deren Instanzwerte aufnehmen. Eine Besonderheit ergibt sich, wenn ein Attribut über mehrere Assoziationen verfügbar ist. So kann *Objekt1* auf drei verschiedene Instanzmengen von *AttributZ* zugreifen: Die über *asso2* verbundenen Instanzen, die über *asso3* verbundenen Instanzen sowie die Vereinigung dieser Teilmengen.

B.4 Instantiierung von Spezifikationstemplates

```
public Object transfer(Hashtable attr, Hashtable prop)
{
    Set sourceValues = (Set)attr.get($SOURCECONTEXT$);
    Object targetValue = sourceValue.iterator().next();
    return targetValue;
}
```




```
public Object transfer(Hashtable attr, Hashtable prop)
{
    Set sourceValues = (Set)attr.get("Objekt2.AttributY");
    Object targetValue = sourceValue.iterator().next();
    return targetValue;
}
```

Dieses Methodentemplate liefert den Eingabewert unverändert als Ergebnis zurück. Im Metamodell soll der Wert von *AttributY* auf *AttributX* transferiert werden. Dazu muss der variable Bestandteil *$SOURCECONTEXT$* des Templates durch die Adressierung des Quellwerts ersetzt werden. Diese Ersetzung kann durch Modellierungswerkzeuge automatisiert werden.

B.5 Templates einer Spezifikations-Standardbibliothek

Die folgenden Methodentemplates sollen demonstrieren, wie eine Standardbibliothek für Attributierungsspezifikationen (in Java) aufgebaut sein könnte.

```
public Object transfer(Hashtable attr, Hashtable prop)
{
    Set sourceValues = (Set)attr.get($SOURCECONTEXT$);
    Object targetValue = sourceValue.iterator().next();
    return targetValue;
}
```

Die Transferfunktion überträgt den Wert eines Attributvorkommens des Quellkontexts auf eine andere Attributinstanz. Per Definition muss im Quellkontext genau eine Attributinstanz vorhanden sein.

```
public Object const(Hashtable attr, Hashtable prop)
{
    $ATTRIBUTETYPE$ result = $CONST_X$;
    return result;
}
```

Die Definition einer Konstante verwendet keine Attribute als Argumente. Der Wert der Konstante sowie der Typ des Zielattributvorkommens wird beim Instantiieren des Templates eingesetzt.

```
public Object equal_const(Hashtable attr, Hashtable prop)
{
    Set values = (Set)args.get($CONTEXT$);
    Object value = values.iterator().next();
    return value.equals($CONST_X$);
}
```

Der Wert des übergebenen Attributvorkommens wird mit einem konstanten Wert verglichen, der beim Instantiieren des Templates eingesetzt wird. Diese Funktion ist, wie auch die Transferfunktion, für genau einen Eingabewert definiert und kann für entsprechende größer/kleiner/ungleich Vergleichsoperationen modifiziert werden.

```
public Object equal(Hashtable attr, Hashtable prop)
{
    Set values1 = (Set)args.get($CONTEXT1$);
    Set values2 = (Set)args.get($CONTEXT2$);
    Object value1 = values1.iterator().next();
    Object value2 = values2.iterator().next();
    return value1.equals(value2);
}
```

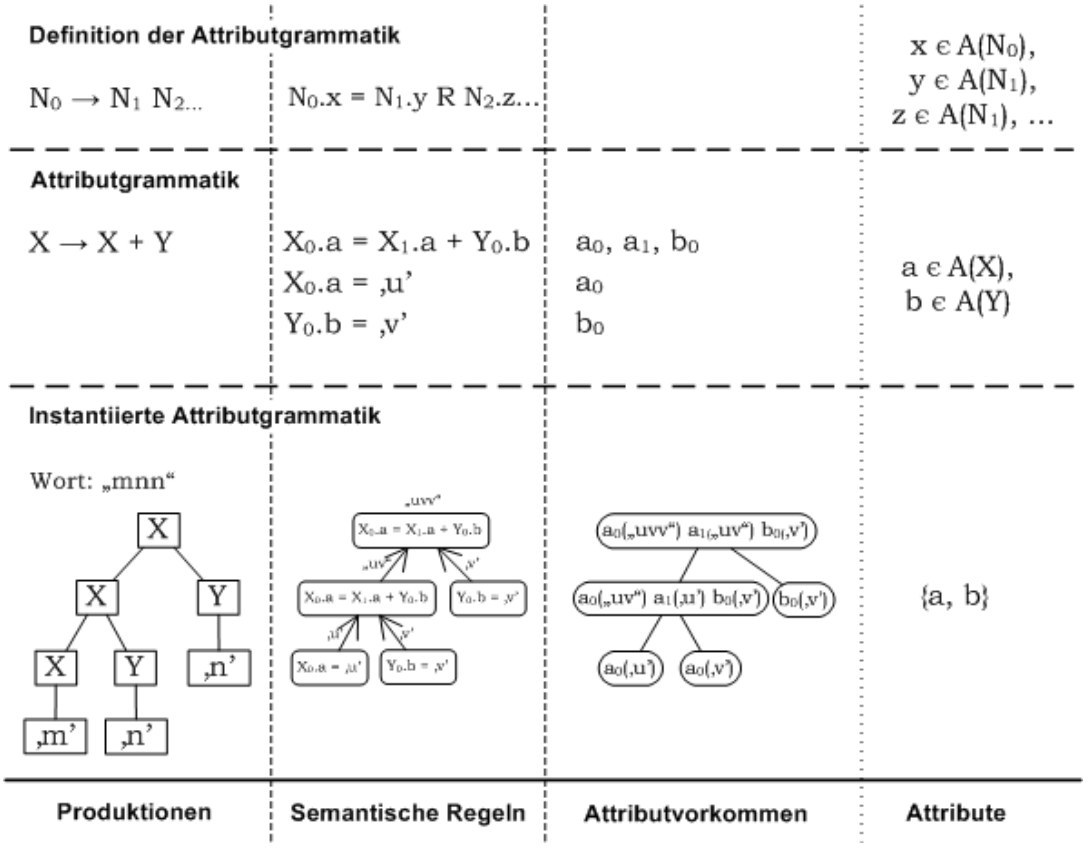
Entspricht der Vergleichsfunktion mit einer Konstanten, der Vergleich findet aber zwischen zwei Attributinstanzen statt.

B.6 Attributierungssyntax für Metamodelle

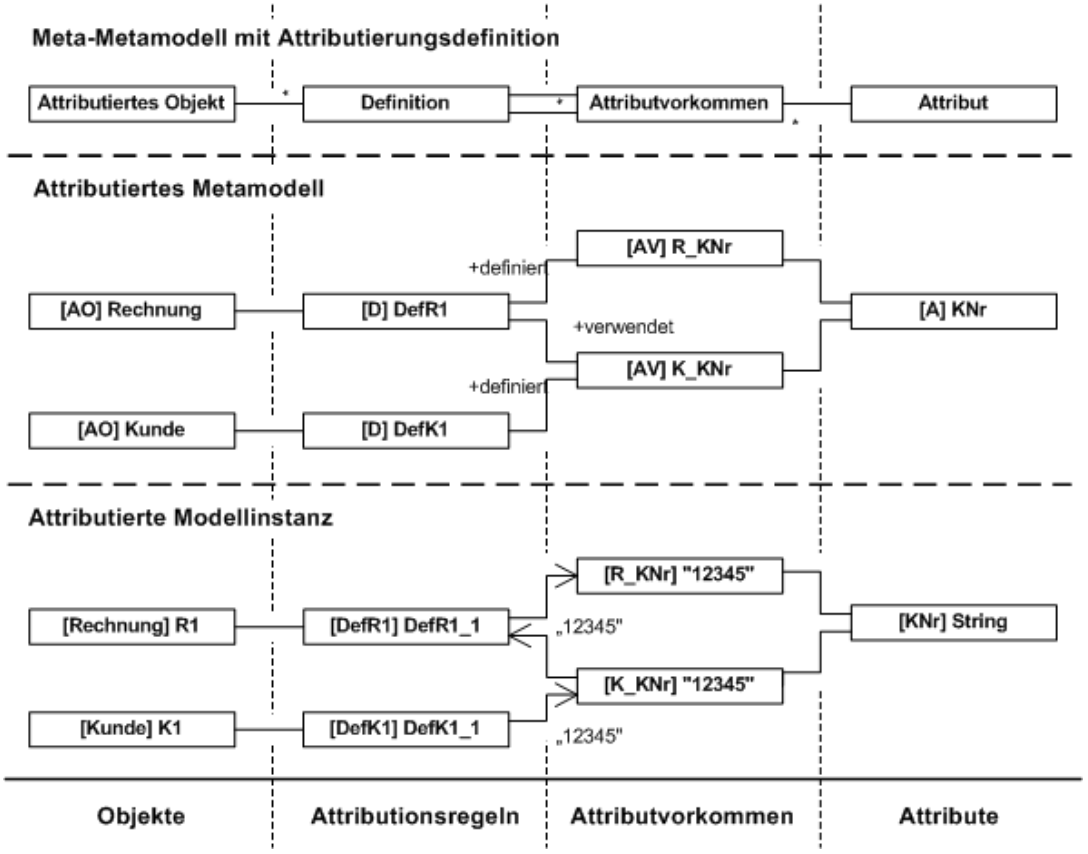
```
MetaAttr  ::= meta attrs defs cons specs;

meta      ::= "meta" { meta_tag };
meta_tag  ::= meta_id ":" meta_value ";";
meta_id   ::= String;
meta_value::= String;

attrs     ::= "attributes" { attr_tag };
attr_tag  ::= attr_id ":" attr_type "(" comment ")" ";";
attr_id   ::= String;
attr_type ::= ( "Integer" | "Real" | "String" );

defs      ::= "definitions" { definition };
definition::= context ":" { dspec };
dspec     ::= attr_id ":" spec_id ";" { argument };

cons      ::= "constraints" { constraint };
constraint::= context ":" { cspec };
cspec     ::= attr_id ":" spec_id ";" "(" comment ")" { argument };

specs     ::= { specification };
specification ::= spec_id ":" spec_def ";";
spec_def  ::= String;

comment   ::= String;
context   ::= String;
subcontext::= String;
prop_id   ::= String;
argument  ::= context ( attribute | property )
attribute ::= [ "[" subcontext "]" ] "." attr_id ";";
property  ::=  "->" prop_id ";";
spec_id   ::= (std_lib | String);
std_lib   ::= ( "std_transfer_spec" | ... );
```

Diese Definition in EBNF legt eine Syntax zur Formulierung von Attributierungen auf Metamodellen fest. Ihr Aufbau ist an A.4 angelehnt und besteht aus mehreren Abschnitten, in denen je eine Gruppe von Attributierungseigenschaften festgelegt wird: Metadaten, Attributdeklarationen, Definitionen, Bedingungen und Spezifikationen. Diese Sprache bildet die Grundlage für die Attributierung 4.9.

B.7 Konzeptioneller Vergleich zwischen Attributgrammatiken und attributierten Metamodellen anhand eines Beispiels

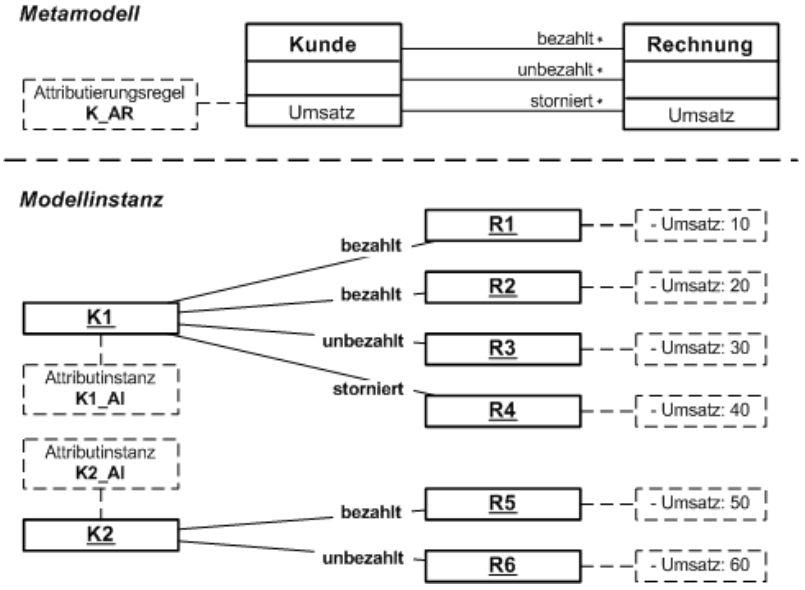


Definitions- und Instantiierungsebenen einer Attributgrammatik



Definitions- und Instantiierungsebenen eines attributierten Metamodells: Der Wert des Attributs KNr wird innerhalb des Objekts Kunde definiert (K_KNr) und an das Objekt Rechnung (R_KNr) weitergeleitet.

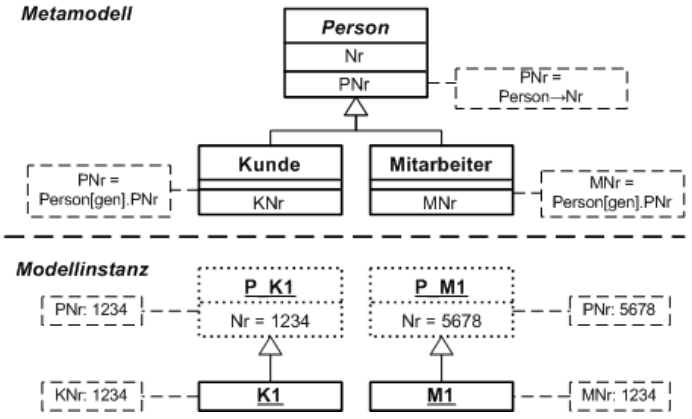
B.8 Adressierung von Attributwerten über ihren Kontext



Die Abbildung zeigt anhand eines einfachen Metamodells und entsprechenden Modellinstanzen für unterschiedliche Adressierungsarten, welche Objektinstanzen (und damit auch welche Attributwerte) jeweils zurückgeliefert werden.

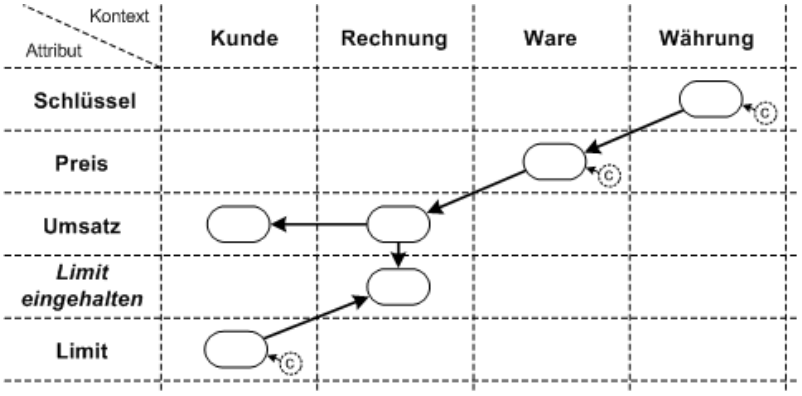
Attributierungsregel K_AR	**Ergebnis K1_AI**	**Ergebnis K2_AI**
Rechnung[all].Umsatz	R1, R2, R3, R4, R5, R6	R1, R2, R3, R4, R5, R6
Rechnung.Umsatz	R1, R2, R3, R4	R5, R6
Rechnung[unbezahlt + storniert].Umsatz	R3, R4	R6
Rechnung[bezahlt].Umsatz	R1, R2	R5

Die folgende Abbildung zeigt das Initialisieren des Attributs *PNr* durch den Wert der Objekteigenschaft *Nr*. Zur Laufzeit übernehmen die Objekte *K1* und *M1* vom Typ *Kunde* bzw. *Mitarbeiter* den Wert, den das Attribut *PNr* in ihrem jeweiligen (virtuellen) Supertyp *Person* besitzt.


Metamodell
Person
Nr
PNr
PNr =
Person→Nr
Kunde
KNr
Mitarbeiter
MNr
PNr =
Person[gen].PNr
MNr =
Person[gen].PNr
Modellinstanz
P_K1
Nr = 1234
P_M1
Nr = 5678
PNr: 1234
PNr: 5678
K1
M1
KNr: 1234
MNr: 1234

C Auswertung der Attributierung

C.1 Globale Abhängigkeiten im Metamodell



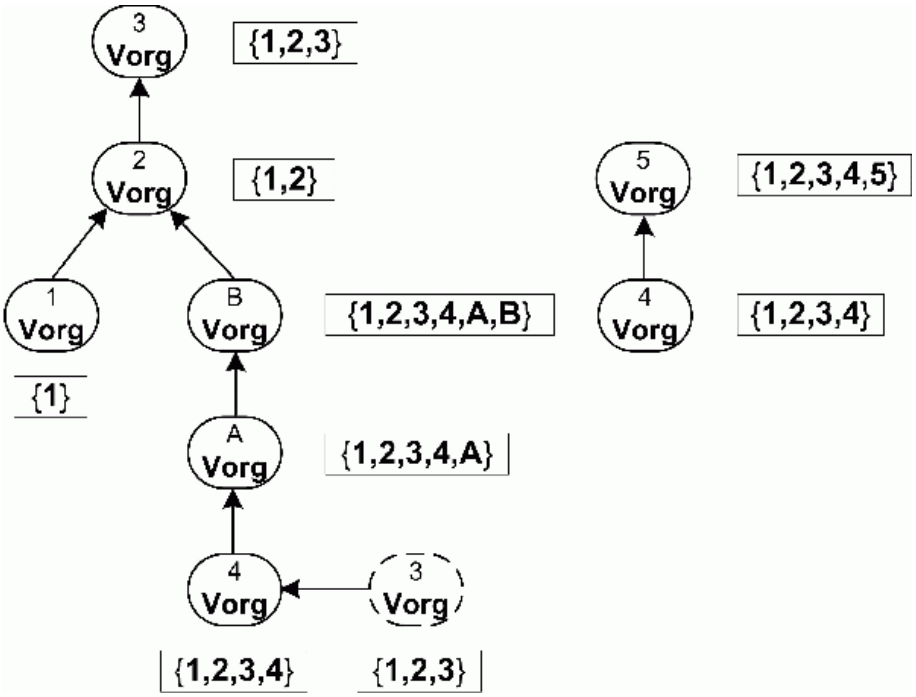
Die Grafik zeigt die vollständigen Abhängigkeiten, die im Metamodell des Kundenumsatzbeispiels enthalten sind. Die Darstellung erfolgt in Form eines Graphen, dessen Knoten in einem Gitternetz angeordnet sind. Dieses wird durch die definierten Attributtypen und den vorhandenen Definitionskontexten gebildet und ist eine detaillierte Aufstellung der Abhängigkeitsbeziehungen, die zwischen den Attributvorkommen des Metamodells existieren.

C.2 Spezifikationen für eine Auswertung zyklischer Attributierungen

```
public Object Konf_Schnitt(Hashtable attr, Hashtable prop)
{
    // Die Vorgängermengen in sourceValues vereinen
    Set sourceValues1 = (Set)attr.get("Startknoten[s].Vorg");
    Set sourceValues2 = (Set)attr.get("Knoten[v].Vorg");
    Set sourceValues = sourceValues1.addAll(sourceValues2);

    // Ergebnis initialisieren
    Set result = null;

    // Schnittoperator auf alle Eingabemengen anwenden
    for (Iterator i = sourceValues.iterator(); i.hasNext();)
    {
        Set iter = i.next();

        // Erste Vorgängermenge in das Ergebnis übernehmen
        if (result == null)
        {
            result = iter;
        }
        // Ansonsten Schnittoperator anwenden, „ref" ignorieren
        else if (!iter == REF);
        {
            result.retainAll(iter);
        }
    }

    // Aktuellen Knoten hinzufügen
    result.add(prop.get("Knoten->Name"));

    // Schnittmenge zurückliefern
    return result;
}
```

Zwei Spezifikationen, die die Datenflussmenge initialisieren und durch den Schnitt-Konfluenzoperator weiterleiten.

C.3 Alternative Wahl des Wurzelelements in einer zyklischen Attributierung



Die Wahl des Knotens *3* als Wurzelelement führt zunächst zur Betrachtung einer Teilkette. Steht deren Ergebnis nach mehreren Iterationen zur Verfügung, so kann im nächsten Schritt auch für Knoten *5* ein Ergebnis berechnet werden. Das Endergebnis ist dabei unabhängig von der Wahl des Wurzelknotens.

D Anwendungsbeispiel

D.1 Auszug aus der Spezifikationsimplementierung

```
/**
 * Menge der Eingabemengen aller Vorgänger zurückliefern.
 */
public static HashSet DFA_filtern(HashMap<String, HashSet> attribute, String attribut)
{
    HashSet in_Set = new HashSet();
    for (Iterator iterator = attribute.keySet().iterator(); iterator.hasNext();)
    {
        String key = (String) iterator.next();

        if (key.contains(attribut))
        {
            in_Set.addAll(attribute.get(key));
        }
    }
    return in_Set;
}

/**
 * Transferiere die Menge des (einzigen) Vorgängers bzw -REF-.
 */
public static Object DFA_transfer(HashMap<String, HashSet> attribute, String attribut)
{
    // Menge der von Vorgängern erhaltenen MinNS-Mengen
    HashSet in_Set = DFA_filtern(attribute, attribut);

    // Menge des ersten (und einzigen) Vorgängers auslesen
    Object in_O = in_Set.iterator().next();

    // Eingabemenge des (einzigen) Vorgängers transferieren
    if (in_O instanceof Set)
    {
        HashSet result = new HashSet();
        result.addAll((Set) in_O);
        return result;
    }
    // Falls Referenz übergeben: Referenz zurückliefern
    else if (in_O instanceof String && ((String) in_O).equals("-REF-"))
    {
        return "-REF-";
    }
    return null;
}
```

```
/**
 * Transferiert die Menge des (einzigen) Vorgängers bzw -REF-.
 */
public static Object DFA_vereinigung(HashMap<String, HashSet> attribute,
        String attribut)
{
    // Menge der von Vorgängern erhaltenen MinNS Mengen
    HashSet in_Set = DFA_filtern(attribute, attribut);

    HashSet vereinigung = new HashSet();

    for (Iterator iterator = in_Set.iterator(); iterator.hasNext();)
    {
        Object in_x = iterator.next();

        if (in_x instanceof Set)
        {
            vereinigung.addAll((Set) in_x);
        }
        else if (in_x instanceof String && ((String) in_x).equals("-REF-"))
        {
            return "-REF-";
        }
    }
    return vereinigung;
}

/**
 * Transferiert die Menge des (einzigen) Vorgängers bzw -REF-.
 */
public static Object DFA_schnitt(HashMap<String, HashSet> attribute, String attribut)
{
    // Menge der von Vorgängern erhaltenen MinNS-Mengen
    HashSet in_Set = DFA_filtern(attribute, attribut);

    // -REF- entfernen
    in_Set.remove("-REF-");

    // Eingabemengen schneiden
    if (in_Set.size() != 0)
    {
        // Elemente der ersten Vorgängermenge in Ergebnis laden
        HashSet schnitt = new HashSet();
        schnitt.addAll((HashSet) in_Set.iterator().next());

        // Mit allen weiteren Vorgängermengen schneiden
        for (Iterator iterator = in_Set.iterator(); iterator.hasNext();)
        {
            Object argument = iterator.next();

            schnitt.retainAll((HashSet) argument);
        }
        return schnitt;
    }
    // Eingabe bestand nur aus -REF-
    else
    {
        return "-REF-";
    }
}
```

D.2 Log Auszug: Generierter Instanzsatz

```
[ModellParserJNT] Modellelemente aus Container einlesen
[ModellParserJNT] Es wurden 29 Objekte eingelesen
|AttrModellParserJNT| Attributexemplare generieren
```

Die instantiierte Attributierung enthält 76 Attributexemplare					
ID	**Kontext**	**Attribut**	**Wert**	**Argumente**	**Eigenschaften**
0	InitialNode	MinNS	"-NULL-"		[name:null] [icon:null]
1	InitialNode	AltNS	"-NULL-"		[name:null] [icon:null]
2	Action	MinNS	"-NULL-"	(34:in)	[name:a] [icon:null] [targetexecutiontime:0]
3	Action	AltNS	"-NULL-"	(35:in)	[name:a] [icon:null] [targetexecutiontime:0]
4	DecisionNode	MinNS	"-NULL-"	(31:in)	[name:null] [icon:null]
5	DecisionNode	AltNS	"-NULL-"	(32:in)	[name:null] [icon:null]
6	DecisionNode	FinNS	"-NULL-"	(5:loc) (4:loc)	[name:null] [icon:null]
7	Action	MinNS	"-NULL-"	(37:in)	[name:b] [icon:null] [targetexecutiontime:0]
8	Action	AltNS	"-NULL-"	(38:in)	[name:b] [icon:null] [targetexecutiontime:0]
9	Action	MinNS	"-NULL-"	(40:in)	[name:c] [icon:null] [targetexecutiontime:0]
10	Action	AltNS	"-NULL-"	(41:in)	[name:c] [icon:null] [targetexecutiontime:0]
11	Action	MinNS	"-NULL-"	(49:in)	[name:Komp.pr. durchf.] [icon:null] [targetexecutiontime:0]
12	Action	AltNS	"-NULL-"	(50:in)	[name:Komp.pr. durchf.] [icon:null] [targetexecutiontime:0]
13	Action	MinNS	"-NULL-"	(46:in)	[name:d] [icon:null] [targetexecutiontime:0]
14	Action	AltNS	"-NULL-"	(47:in)	[name:d] [icon:null] [targetexecutiontime:0]
15	Action	MinNS	"-NULL-"	(43:in)	[name:Komp.pr. durchf.] [icon:null] [targetexecutiontime:0]
16	Action	AltNS	"-NULL-"	(44:in)	[name:Komp.pr. durchf.] [icon:null] [targetexecutiontime:0]
17	Action	MinNS	"-NULL-"	(52:in)	[name:c] [icon:null] [targetexecutiontime:0]
18	Action	AltNS	"-NULL-"	(53:in)	[name:c] [icon:null] [targetexecutiontime:0]
19	Action	MinNS	"-NULL-"	(58:in)	[name:Komp.pr. durchf.] [icon:null] [targetexecutiontime:0]
20	Action	AltNS	"-NULL-"	(59:in)	[name:Komp.pr. durchf.] [icon:null] [targetexecutiontime:0]
21	Action	MinNS	"-NULL-"	(64:in)	[name:a] [icon:null] [targetexecutiontime:0]
22	Action	AltNS	"-NULL-"	(65:in)	[name:a] [icon:null] [targetexecutiontime:0]
23	MergeNode	MinNS	"-NULL-"	(55:in) (61:in) (67:in)	[name:null] [icon:null]
24	MergeNode	AltNS	"-NULL-"	(57:in) (63:in) (69:in) (23:loc)	[name:null] [icon:null]
25	MergeNode	FinNS	"-NULL-"	(24:loc) (23:loc)	[name:null] [icon:null]
26	Action	MinNS	"-NULL-"	(70:in)	[name:Wertpapierorder buchen] [icon:null] [targetexecutiontime:0]

27	Action	AltNS	"-NULL-"	(71:in)	[name:Wertpapierorder buchen] [icon:null] [targetexecutiontime:0]
28	FinalNode	MinNS	"-NULL-"	(73:in)	[name:null] [icon:null]
29	FinalNode	AltNS	"-NULL-"	(74:in)	[name:null] [icon:null]
30	FinalNode	FinNS	"-NULL-"	(29:loc) (28:loc)	[name:null] [icon:null]
31	ActivityEdge	MinNS	"-NULL-"	(0:source)	
32	ActivityEdge	AltNS	"-NULL-"	(1:source)	
33	ActivityEdge	AltNSloc	"-NULL-"	(32:loc) (31:loc)	
34	ActivityEdge	MinNS	"-NULL-"	(4:source)	
35	ActivityEdge	AltNS	"-NULL-"	(5:source)	
36	ActivityEdge	AltNSloc	"-NULL-"	(35:loc) (34:loc)	
37	ActivityEdge	MinNS	"-NULL-"	(4:source)	
38	ActivityEdge	AltNS	"-NULL-"	(5:source)	
39	ActivityEdge	AltNSloc	"-NULL-"	(38:loc) (37:loc)	
40	ActivityEdge	MinNS	"-NULL-"	(4:source)	
41	ActivityEdge	AltNS	"-NULL-"	(5:source)	
42	ActivityEdge	AltNSloc	"-NULL-"	(41:loc) (40:loc)	
43	ActivityEdge	MinNS	"-NULL-"	(9:source)	
44	ActivityEdge	AltNS	"-NULL-"	(10:source)	
45	ActivityEdge	AltNSloc	"-NULL-"	(44:loc) (43:loc)	
46	ActivityEdge	MinNS	"-NULL-"	(7:source)	
47	ActivityEdge	AltNS	"-NULL-"	(8:source)	
48	ActivityEdge	AltNSloc	"-NULL-"	(47:loc) (46:loc)	
49	ActivityEdge	MinNS	"-NULL-"	(2:source)	
50	ActivityEdge	AltNS	"-NULL-"	(3:source)	
51	ActivityEdge	AltNSloc	"-NULL-"	(50:loc) (49:loc)	
52	ActivityEdge	MinNS	"-NULL-"	(11:source)	
53	ActivityEdge	AltNS	"-NULL-"	(12:source)	
54	ActivityEdge	AltNSloc	"-NULL-"	(53:loc) (52:loc)	
55	ActivityEdge	MinNS	"-NULL-"	(17:source)	
56	ActivityEdge	AltNS	"-NULL-"	(18:source)	
57	ActivityEdge	AltNSloc	"-NULL-"	(56:loc) (55:loc)	
58	ActivityEdge	MinNS	"-NULL-"	(13:source)	
59	ActivityEdge	AltNS	"-NULL-"	(14:source)	
60	ActivityEdge	AltNSloc	"-NULL-"	(59:loc) (58:loc)	
61	ActivityEdge	MinNS	"-NULL-"	(19:source)	
62	ActivityEdge	AltNS	"-NULL-"	(20:source)	
63	ActivityEdge	AltNSloc	"-NULL-"	(62:loc) (61:loc)	
64	ActivityEdge	MinNS	"-NULL-"	(15:source)	
65	ActivityEdge	AltNS	"-NULL-"	(16:source)	
66	ActivityEdge	AltNSloc	"-NULL-"	(65:loc) (64:loc)	
67	ActivityEdge	MinNS	"-NULL-"	(21:source)	
68	ActivityEdge	AltNS	"-NULL-"	(22:source)	
69	ActivityEdge	AltNSloc	"-NULL-"	(68:loc) (67:loc)	
70	ActivityEdge	MinNS	"-NULL-"	(23:source)	
71	ActivityEdge	AltNS	"-NULL-"	(24:source)	
72	ActivityEdge	AltNSloc	"-NULL-"	(71:loc) (70:loc)	
73	ActivityEdge	MinNS	"-NULL-"	(26:source)	
74	ActivityEdge	AltNS	"-NULL-"	(27:source)	
75	ActivityEdge	AltNSloc	"-NULL-"	(74:loc) (73:loc)	

D.3 Log Auszug: Erste Iteration zur Berechnung der Abhängigkeitskette

```
[TraverserZykJWT] Starte iterative Berechnung
[TraverserZykJWT] Beginne Berechnungsiteration 1
[TraverserZykJWT] Hinabsteigen für Argument: 80
[TraverserZykJWT] Hinabsteigen für Argument: 20
[TraverserZykJWT] Hinabsteigen für Argument: 47
[TraverserZykJWT] Hinabsteigen für Argument: 7
[TraverserZykJWT] Hinabsteigen für Argument: 44
[TraverserZykJWT] Hinabsteigen für Argument: 6
[TraverserZykJWT] Hinabsteigen für Argument: 41
[TraverserZykJWT] Hinabsteigen für Argument: 3
[TraverserZykJWT] Hinabsteigen für Argument: 38
[TraverserZykJWT] Hinabsteigen für Argument: 17
[TraverserZykJWT] Hinabsteigen für Argument: 35
[TraverserZykJWT] Berechne Exemplar der Abhängigkeitskette: 35
[SpezifikationJWTJava] Starte Implementierung minns_transfer
[TraverserZykJWT] Hinabsteigen für Argument: 65
[TraverserZykJWT] Hinabsteigen für Argument: 29
[TraverserZykJWT] Hinabsteigen für Argument: 62
[TraverserZykJWT] Hinabsteigen für Argument: 13
[TraverserZykJWT] Hinabsteigen für Argument: 53
[TraverserZykJWT] Hinabsteigen für Argument: 11
[TraverserZykJWT] Hinabsteigen für Argument: 50
[TraverserZykJWT] Hinabsteigen für Argument: 9
[TraverserZykJWT] Hinabsteigen für Argument: 59
[TraverserZykJWT] Hinabsteigen für Argument: 32
[TraverserZykJWT] Hinabsteigen für Argument: 56
[TraverserZykJWT] Berechne Exemplar der Abhängigkeitskette: 56
[SpezifikationJWTJava] Starte Implementierung minns_transfer
[TraverserZykJWT] Hinabsteigen für Argument: 77
[TraverserZykJWT] Hinabsteigen für Argument: 27
[TraverserZykJWT] Hinabsteigen für Argument: 74
[TraverserZykJWT] Hinabsteigen für Argument: 25
[TraverserZykJWT] Hinabsteigen für Argument: 71
[TraverserZykJWT] Hinabsteigen für Argument: 23
[TraverserZykJWT] Hinabsteigen für Argument: 68
[TraverserZykJWT] Berechne Exemplar der Abhängigkeitskette: 68
[SpezifikationJWTJava] Starte Implementierung minns_transfer
[TraverserZykJWT] Berechne Exemplar der Abhängigkeitskette: 23
[SpezifikationJWTJava] Starte Implementierung minns_action
[TraverserZykJWT] Berechne Exemplar der Abhängigkeitskette: 71
[SpezifikationJWTJava] Starte Implementierung minns_transfer
[TraverserZykJWT] Berechne Exemplar der Abhängigkeitskette: 25
[SpezifikationJWTJava] Starte Implementierung minns_action
[TraverserZykJWT] Berechne Exemplar der Abhängigkeitskette: 74
[SpezifikationJWTJava] Starte Implementierung minns_transfer
```

```
[TraverserZykJWT] Berechne Exemplar der Abhängigkeitskette: 27
[SpezifikationJWTJava] Starte Implementierung minns_action
[TraverserZykJWT] Berechne Exemplar der Abhängigkeitskette: 77
[SpezifikationJWTJava] Starte Implementierung minns_transfer
[TraverserZykJWT] Berechne Exemplar der Abhängigkeitskette: 32
[SpezifikationJWTJava] Starte Implementierung minns_merge
[TraverserZykJWT] Berechne Exemplar der Abhängigkeitskette: 59
[SpezifikationJWTJava] Starte Implementierung minns_transfer
[TraverserZykJWT] Berechne Exemplar der Abhängigkeitskette: 9
[SpezifikationJWTJava] Starte Implementierung minns_action
[TraverserZykJWT] Berechne Exemplar der Abhängigkeitskette: 50
[SpezifikationJWTJava] Starte Implementierung minns_transfer
[TraverserZykJWT] Berechne Exemplar der Abhängigkeitskette: 11
[SpezifikationJWTJava] Starte Implementierung minns_action
[TraverserZykJWT] Berechne Exemplar der Abhängigkeitskette: 53
[SpezifikationJWTJava] Starte Implementierung minns_transfer
[TraverserZykJWT] Berechne Exemplar der Abhängigkeitskette: 13
[SpezifikationJWTJava] Starte Implementierung minns_action
[TraverserZykJWT] Berechne Exemplar der Abhängigkeitskette: 62
[SpezifikationJWTJava] Starte Implementierung minns_transfer
[TraverserZykJWT] Berechne Exemplar der Abhängigkeitskette: 29
[SpezifikationJWTJava] Starte Implementierung minns_transfer
[TraverserZykJWT] Berechne Exemplar der Abhängigkeitskette: 65
[SpezifikationJWTJava] Starte Implementierung minns_transfer
[TraverserZykJWT] Berechne Exemplar der Abhängigkeitskette: 17
[SpezifikationJWTJava] Starte Implementierung minns_merge
[TraverserZykJWT] Berechne Exemplar der Abhängigkeitskette: 38
[SpezifikationJWTJava] Starte Implementierung minns_transfer
[TraverserZykJWT] Berechne Exemplar der Abhängigkeitskette: 3
[SpezifikationJWTJava] Starte Implementierung minns_action
[TraverserZykJWT] Berechne Exemplar der Abhängigkeitskette: 41
[SpezifikationJWTJava] Starte Implementierung minns_transfer
[TraverserZykJWT] Berechne Exemplar der Abhängigkeitskette: 5
[SpezifikationJWTJava] Starte Implementierung minns_action
[TraverserZykJWT] Berechne Exemplar der Abhängigkeitskette: 44
[SpezifikationJWTJava] Starte Implementierung minns_transfer
[TraverserZykJWT] Berechne Exemplar der Abhängigkeitskette: 7
[SpezifikationJWTJava] Starte Implementierung minns_action
[TraverserZykJWT] Berechne Exemplar der Abhängigkeitskette: 47
[SpezifikationJWTJava] Starte Implementierung minns_transfer
[TraverserZykJWT] Berechne Exemplar der Abhängigkeitskette: 20
[SpezifikationJWTJava] Starte Implementierung minns_transfer
[TraverserZykJWT] Berechne Exemplar der Abhängigkeitskette: 80
[SpezifikationJWTJava] Starte Implementierung minns_transfer
[TraverserZykJWT] Berechne Exemplar der Abhängigkeitskette: 0
[SpezifikationJWTJava] Starte Implementierung minns_transfer
[TraverserZykJWT] Berechnungsiteration beendet
```

D.4 Log Auszug: Zwischenergebnisse der iterativen Berechnung