Beispielgetriebene Entwicklung domänenspezifischer Modellierungssprachen

Bastian Roth

UNIVERSITÄT BAYREUTH
INSTITUT FÜR INFORMATIK

Beispielgetriebene Entwicklung domänenspezifischer Modellierungssprachen

Von der Universität Bayreuth zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Abhandlung

von
Bastian Josef Roth
geboren in Kemnath

1. Gutachter: Prof. Dr.-Ing. Stefan Jablonski
2. Gutachter: Prof. Dr. rer. pol. Ulrich Frank

Tag der Einreichung: 27. August 2014
Tag des Kolloquiums: 05. Dezember 2014

Für Lisa.

Zusammenfassung

Aufgrund der stetig wachsenden Komplexität von Softwaresystemen gewinnt der Einsatz domänenspezifischer Modellierungssprachen (DSMLs) im Kontext der Softwareentwicklung zunehmend an Bedeutung. Die formale Spezifikation von DSMLs ist jedoch eine aufwändige und daher oftmals mit Fehlern behaftete Aufgabe. In weiten Teilen liegt dies an der Top-Down Vorgehensweise, bei der zunächst die Realität verstanden werden muss, um darauf basierend eine DSML spezifizieren zu können. Erst wenn eine Sprache existiert, können mit ihrer Hilfe Modelle erstellt werden, die die Realität beschreiben. Beim Entwickeln von DSMLs müssen demzufolge die Modelle zunächst „vorgedacht" werden, was dem Sprachentwickler eine doppelte Typabstraktion abverlangt, nämlich von der Realität über die Modelle hin zur Modellierungssprache.

Dadurch motiviert ist im Lauf der letzten Jahre eine neue Bewegung entstanden, die es erlaubt DSMLs bottom-up zu entwerfen. Modelle können dabei ohne eine bereits vorhandene Sprache skizziert und daraus eine DSML automatisiert abgeleitet werden. Für das Skizzieren kommen verbreitete Zeichenwerkzeuge, wie z.B. Microsoft Visio oder Dia, zum Einsatz. DSMLs werden von derartigen Werkzeugen allerdings nur in visueller Hinsicht und somit äußerst rudimentär unterstützt. Deshalb muss eine abgeleitete Sprache zunächst in ein konkretes Modellierungssystem importiert werden. Bei der iterativen Entwicklung von DSMLs ist der Benutzer also dazu gezwungen, zwischen dem Zeichenwerkzeug und dem Modellierungssystem bedarfsweise zu wechseln. Außerdem geht durch den Technologiebruch der Bezug zwischen den einzelnen Beispielmodellen und den daraus induzierten Sprachartefakten verloren. Im Regelfall ist es nicht möglich, die Visualisierungsinformationen in das Modellierungssystem zu übernehmen. Eine konkrete Visualisierung ist demzufolge auch weiterhin separat innerhalb dieses Systems zu definieren.

Um der Problematik zu begegnen, wird im Rahmen dieser Arbeit ein ganzheitlicher Ansatz mit Werkzeugunterstützung propagiert, der sowohl das Skizzieren von Beispielmodellen (freies Modellieren) als auch das Erstellen von Modellen auf Basis einer gegebenen DSML (stringentes Modellieren) innerhalb eines Modellierungssystems integriert. Neben grafischen Sprachen werden ebenso DSMLs adressiert, die der textuell-projizierenden Visualisierungsform angehören. Dank der Integration erlaubt der vorgestellte Ansatz sowohl die manuelle Anpassung einer formalen Sprachspezifikation als auch die automatisierte Ableitung von Sprachänderungen basierend auf modifizierten oder zusätzlichen Beispielmodellen. Darüber hinaus wird der Benutzer auf Stellen in der induzierten DSML hingewiesen, die Optimierungspotenzial aufweisen und daher durch bestimmte Maßnahmen (z.B. Anwendung von Einfachvererbung) verfeinert werden können.

Abstract

Owing to the continuously growing complexity of software systems, in the context of software development the usage of domain-specific modelling languages (DSMLs) gains increasingly in importance. However, the formal specification of DSMLs is a sophisticated and oftentimes error-prone task. Mainly, this is due to the top-down practice, which first requires to understand the reality in order to specify a DSML based upon it. Only after a language exists models can be created that describe the reality using this language. Hence, during DSML development models need to be "pre-thought", which claims the language developer from a double type abstraction (namely from the reality across the models towards a modelling language).

As a consequence, during recent years a new approach was discovered, which allows for designing DSMLs the bottom-up way. Following this, models can be sketched in the absence of an existing language. Those models are utilized to automatically derive a DSML. For creating sketches popular drawing tools like Microsoft Visio or Dia are used. However, such tools merely support DSMLs in terms of their notation and thus highly rudimentary. Therefore, a derived language has to be imported into a concrete modelling system first. During the iterative development of DSMLs users are forced to switch between the drawing tool and the modelling systems as needed. Furthermore, because of the technology split, the link gets lost between the particular example models and the induced language artefacts. Generally, it is not possible to transfer the notational information into the modelling system. Consequently, a concrete notation has to be specified separately further on.

To address this problem, in the context of this work a holistic approach combined with tool support is introduced that integrates sketching of example models (free modelling) as well as the creation of models based on a given DSML (stringent modelling) within one modelling system. Beyond diagram languages, even DSMLs are considered, which follow the textual-projectional visualization. Thanks to the integration, the presented approach allows for the manual adaption of a formal language specification as well as for automatically deriving language changes resting on modified or additional example models. Moreover, users are pointed to locations of an induced DSML with potential for optimization. Thereby, adequate improvements (e.g., applying single inheritance) are suggested whenever possible.

Danksagung

Diese Dissertation wäre wohl niemals ohne die Unterstützung von vielen netten Menschen entstanden, bei denen ich mich an dieser Stelle von ganzem Herzen bedanken möchte.

Allen voran gilt mein Dank meinem Doktorvater Prof. Dr.-Ing. Stefan Jablonski, der mir durch das entgegengebrachte Vertrauen die Chance zur Promotion eröffnet hat. Während meiner Arbeit am Lehrstuhl stand er mir stets mit Rat und Tat zur Seite und hat durch zahlreiche Anregungen, Hinweise und Diskussionen richtungsweisend zum vorliegenden Resultat beigetragen. Durch seine Offenheit und Herzlichkeit schuf er ein äußerst harmonisches Arbeitsklima, für das ich ihm ebenfalls sehr dankbar bin.

Ganz herzlich bedanken möchte ich mich auch bei all meinen Kollegen für die zahlreichen fachlichen Gespräche sowie hilfreiche und konstruktive Kritik. Namentlich hervorheben möchte ich Matthias Jahn, Michael Zeising, Stefan Schönig und Lars Ackermann, zu denen während meiner Zeit am Lehrstuhl ein freundschaftliches Verhältnis entstanden ist, das auch außerhalb der Arbeitszeit regelmäßig gepflegt wird. Besonderer Dank gilt hierbei Matthias. Zusammen haben wir die Modellierungsplattform „Model Workbench" entwickelt, die als Implementierungsgrundlage für die Prototypen unserer beiden Dissertationen dient.

Gleichermaßen möchte ich Christine Leinberger, Kerstin Haseloff und Bernd Schlesier danken. Sie waren bzw. sind für die Sicherstellung eines reibungslosen Ablaufs am Lehrstuhl zuständig. Das Wörtchen „waren" bezieht sich auf unsere ehemalige Sekretärin Christine, die leider im August 2013 nach langer schwerer Krankheit verstorben ist. An dieser Stelle möchte ich allen Angehörigen, Verwandten und Freunden von Christine mein herzliches Beileid aussprechen. Mit der Aufgabe der Sekretärin ist nun Kerstin betraut, bei der ich mich ebenfalls für die netten und ermunternden frühmorgendlichen Gespräche bedanken möchte.

Weiterer Dank gebührt Prof. rer. pol. Ulrich Frank für die Übernahme des Zweitgutachtens sowie wertvoller Anmerkungen und Diskussionen bei den beiden Forschungskolloquien in Heiligenstadt in Oberfranken und Essen.

Diese Promotion wurde im Rahmen des Projekts „Kompetenzzentrum für praktisches Prozess- und Qualitätsmanagement" (KpPQ) durchgeführt, das vom Europäischen Fonds für regionale Entwicklung (EFRE, Fördernummer 1502/89304-01/2012) gefördert wurde. Daher möchte ich mich ebenso bei den Entscheidungsträgern dieses Fonds bedanken.

Ein riesengroßes Dankeschön möchte ich auch meinen Eltern und meinen beiden Schwestern aussprechen, die mich auf meinem gesamten Lebensweg und in jeder Lebensphase tatkräftig unterstützt haben. Meinem Vater und meiner Schwester Isabella möchte ich nochmals dafür danken, dass sie die nicht beneidenswerte Aufgabe übernommen haben, diese Arbeit auf sprachliche und grammatikalische Fehler zu überprüfen.

„Das Beste kommt zum Schluss“ ist der Titel eines Filmes mit Jack Nicholson und Morgan Freeman. Deshalb möchte ich zu guter Letzt meiner Ehefrau Lisa von ganzem Herzen danken. Sie ist stets eine verständnisvolle Zuhörerin und hat mir zudem den notwendigen Freiraum für diese Arbeit verschafft, insbesondere in der heißen Endphase. Du glaubst gar nicht, wie sehr du dadurch zur Fertigstellung dieser Dissertation beigetragen hast.

Inhaltsverzeichnis

1 Einführung und Motivation

Software durchdringt heutzutage sämtliche Bereiche des alltäglichen Lebens und damit sowohl den privaten Sektor als auch das breitgefächerte Feld von Industrie und Wirtschaft. Aufgrund der Vielfalt an unterschiedlichen Einsatzgebieten gleicht kein IT-Projekt dem anderen. Hinzu kommt, dass die Komplexität von Softwarelösungen wegen steigender Anforderungen der Anwender ebenfalls stetig zunimmt. Um der Komplexitätsproblematik zu begegnen, hat sich in der Informatik „Abstraktion" als Standardvorgehensweise etabliert [40]. Ziel der Abstraktion ist die Verbergung von Informationen, die zur Lösung bestimmter Aufgaben irrelevant oder zumindest von nebensächlicher Bedeutung sind. Häufig tritt im Kontext der Abstraktion der Begriff der „Abstraktionsebene" auf. Abstraktionsebenen repräsentieren verschiedene, aufeinander aufbauende Stufen mit unterschiedlichem Abstraktionsgrad. Dabei verbirgt eine Abstraktionsebene *n* die Informationen einer unterhalb angeordneten Abstraktionsebene *n-1*, die über einen geringeren Abstraktionsgrad verfügt. Ein klassisches Beispiel für den Einsatz von Abstraktionsebenen ist das Schichtenmodell [44], das bei vielen Softwaresystemen zur Gliederung der Systemarchitektur Verwendung findet (z.B. bei typischen Client-Server-Architekturen wie Webanwendungen [44], aber auch beim ANSI/SPARC Modell zur Implementierung von Datenbanksystemen [68]).

Orthogonal zur Abstraktion auf architektonischer Ebene gibt es die als mittlerweile selbstverständlich erachtete Abstraktion auf Ebene der Programmiersprachen. Ausgehend beispielsweise von der Hochsprache C++ [134] abstrahiert diese von Assembler und Assembler abstrahiert erneut von der Darstellung in Binärform [116]. Letztere ist schließlich von einem Rechner direkt ausführbar. Spätestens seit Bewegungen wie Model Driven Software Development (MDSD) [132] gibt es die Möglichkeit, eine weitere Abstraktionsschicht oberhalb von Hochsprachen zu nutzen. Mit der Unified Modeling Language (UML) [111] existiert ein Standard, der dieser Schicht zuzuordnen ist und verschiedene grafische Sprachen (Diagrammarten) für die objektorientierte Modellierung und Beschreibung von Softwaresystemen umfasst. Diese Diagrammarten sind spezifisch für die einzelnen Aspekte eines Systems (z.B. Klassendiagramme zur feingranularen Definition der Systemarchitektur und Sequenzdiagramme für die Beschreibung von Verhalten auf Klassenebene). Eines der ursprünglichen Ziele der UML war die Bereitstellung einer programmiersprachen- und plattformunabhängigen Modellierungssprache, aus deren Modellen schließlich plattformspezifischer Code in einer bestimmten General Purpose Language (GPL) wie Java [53] oder C++ [134] generiert werden sollte. Insbesondere die Plattformunabhängigkeit stellte sich jedoch als nur schwer umsetzbar heraus, da jede Plattform Eigenheiten aufweist, deren Visualisierung innerhalb von Diagrammen durchaus sinnvoll ist. Deshalb wurden in Version 2.0 des UML Standards Profile als leichtgewichtiger Erweiterungsmechanismus eingeführt [48]. Vereinfacht ausgedrückt stellt ein Profil eine Menge von Stereotypen dar, die auf Konstrukte der UML (z.B. Klassen und Pakete) anwendbar sind. Ein typischer und im Java-Umfeld oft eingesetzter Stereotyp ist der der Enterprise Java Bean (EJB). Eine entsprechend annotierte Klasse wird auf diese Weise automatisch als EJB deklariert, was bei der späteren Code-Generierung berücksichtigt werden kann [56].

Die mithilfe der UML erstellten Diagramme dienen neben der eigentlichen Spezifikation des Softwaresystems auch zu dessen Dokumentation. Sie können demnach als Kommunikationsmittel unter IT-Spezialisten verwendet werden. Für die Kommunikation mit Kunden und anderen Stakeholdern gilt das aber nur bedingt, weil diese im Regelfall nicht über ausreichende UML Kenntnisse verfügen. Hier

sollten stattdessen fachspezifische Termini benutzt werden, die jedoch nicht mit Mitteln der UML abgebildet werden können. Ein prominentes Beispiel sind Organigramme, die zur Beschreibung der organisatorischen Struktur von Unternehmen eingesetzt werden [34]. Als Ergänzung aber auch als Alternative zur UML hat sich daher eine Unterbewegung von MDSD gebildet, die Domain-Specific Modeling (DSM) [76] genannt wird. In deren Fokus steht die Entwicklung von Softwaresystemen auf Basis von Domain-Specific Languages bzw. Domain-Specific Modeling Languages (DSMLs). Innerhalb einer DSML sind die Begriffe und Konzepte der betreffenden Domäne sowie deren Beziehungen zueinander definiert. Demzufolge ist eine DSML spezifisch für ein bestimmtes Problemfeld, aber kann im Gegensatz zu GPLs nicht für die Lösung beliebiger Problemstellungen eingesetzt werden. Der Grundgedanke hinter DSM ist somit die formale Beschreibung einer Lösung eines Problems unter Verwendung eines fachspezifischen Vokabulars, wobei diese Beschreibung von einem Rechner weiterverarbeitet werden kann (z.B. zur Code-Generierung oder Interpretation). Beschreibungen dieser Art werden allgemeinhin als Modelle betitelt [81]. Während des Einsatzes einer DSML – und somit bei der Modellierung – können ausschließlich diejenigen Konzepte instanziiert werden, die innerhalb der Sprache spezifiziert sind [49]. Wird ein weiteres Konstrukt innerhalb der DSML benötigt, so muss sie entsprechend erweitert werden.

Sowohl die Erstellung als auch die Erweiterung einer DSML ist eine komplexe und zeitaufwändige Aufgabe, weil die meisten existierenden Ansätze tiefgehende Erfahrungen im Bereich der Sprachspezifikation (z.B. Meta-Modellierung) erfordern [24, 73]. Bei diesen Ansätzen gilt es die Konzepte der Sprache (abstrakte Syntax), die Visualisierungsform (konkrete Syntax), ein zugehöriges Mapping und gegebenenfalls einzuhaltende Regeln (Constraints) formal zu spezifizieren [24]. Schon allein aus ökonomischer Betrachtung ist es jedoch sinnvoll, eine Reduktion des Aufwands beim Entwickeln von Modellierungssprachen anzustreben [45]. Übersteigt dieser Entwicklungsaufwand denjenigen beim herkömmlichen Softwarebau (sprich mithilfe von GPLs), dann verfehlt der Einsatz von DSMLs sein Ziel.

1.1 Problemstellung und Lösungsskizze

Bevor eine DSML eingesetzt werden kann, muss diese erst definiert werden. Hier zeigt sich bereits das größte Defizit von herkömmlichen Ansätzen, da zunächst die Konzepte der Sprache „vorgedacht" werden müssen. Andernfalls können diese nicht in Modellen instanziiert und verwendet werden [49]. Hierzu ist allerdings eine doppelte Klassenbildung bzw. Typextraktion notwendig, und zwar von den beobachteten Instanzen der Realität über die Modelle (erste Klassenbildung) hin zur Modellierungssprache (zweite Klassenbildung).

Problem 1 *Vordenken der Konzepte einer Sprache*

Die **erste Klassenbildung** ist ein durchaus **natürlicher Vorgang**, der im täglichen Sprachgebrauch sehr oft angewendet wird, ohne dass die betreffenden Akteure dies bemerken. Ein Beispiel liefert der Satz: „In Fabriken nimmt der Einsatz von Maschinen stetig zu.". Darin repräsentieren die beiden Substantive „Fabrik" und „Maschine" unterschiedliche Klassen von realen Objekten. Die für diese Aussage beobachteten Instanzen werden also nicht explizit aufgeführt (je nach Größe der Stichprobe würde dies ohnehin einen enormen Aufwand bedeuten), sondern stattdessen in Klassen eingeteilt und anschließend die Namen der Klassen für eine Beschreibung der realen Welt verwendet.

Auf die Idee, eine neue natürliche Sprache zu definieren und damit eine **zweite Klassenbildung** durchzuführen, käme in der Realität niemand, weil dies keine nennenswerten Vorteile in der zwischenmenschlichen Kommunikation mit sich brächte. Anders verhält es sich im Kontext der Modellierung. Hier ist es durchaus üblich für eine Anwendungsdomäne eine spezifische DSML zu entwickeln, weil auf

diese Weise auch Domänenexperten ohne **tiefgehende Modellierungskenntnisse** in der Lage sind, mit vertrauten Begriffen und Formen Sachverhalte ihrer Domäne auszudrücken, die direkt von Rechnern verarbeitet werden können.

Als Beispieldomäne wird die Organisationsstruktur von Unternehmen betrachtet, die bequem mithilfe von Organigrammen abgebildet werden kann. Kämen stattdessen Objektdiagramme der UML [111] zum Einsatz, müssten sämtliche Strukturen als Objekte modelliert werden, was nicht der von der Geschäftsleitung gewohnten Visualisierungsform entspricht. Das hätte in vielen Fällen negative Auswirkungen auf die Akzeptanz der erstellen Diagramme, selbst dann, wenn die Modelle den realen Strukturen entsprechen. Abbildung 1-1 zeigt dazu dieselbe Organisationsstruktur in zwei unterschiedlichen Visualisierungen, nämlich auf der linken Seite als typisches Organigramm und rechts in Form eines Objektdiagramms. Letztere Darstellungsform ist aufgrund der vielen Verbindungslinien unübersichtlicher und demzufolge schwieriger zu verstehen. Hinzu kommt, dass die linke Variante im Managementumfeld weit verbreitet und somit dem Zielklientel vertraut ist. Eine Diskussion auf Basis der Organigrammnotation ist daher mit großer Wahrscheinlichkeit einfacher zu führen als unter Verwendung eines Objektdiagramms.

Die **Definition einer DSML** erfordert stets **Wissen über die später zu entwickelnden Modelle**, das praktisch im Vorfeld erahnt werden muss. Erst dann können die Konstrukte der DSML mittels einer zweiten Klassenbildung spezifiziert werden. Vor allem bei der initialen Anforderungsanalyse während der Entwicklung eines Softwaresystems ist dieses „Vordenken" ein nur schwer zu überwindendes Hindernis, da die Konzepte nur unzureichend bekannt sind [113]. Bei einem typischen IT-Projekt nehmen die IT-Spezialisten die Anforderung der Domänenexperten auf und versuchen dabei gleichzeitig durch ständiges Hinterfragen ein möglichst exaktes Bild von der zugrundeliegenden Domäne zu erhalten. Aufgrund der anfänglichen Sprachbarriere zwischen IT- und Anwendungsdomäne sowie den damit einhergehenden Fehldeutungen handelt es sich hierbei um einen iterativen Prozess.

Innerhalb der **Anforderungsanalyse** werden deshalb oftmals verbreitete **Office-Werkzeuge** wie Microsoft Word und Microsoft PowerPoint eingesetzt, um eine Beschreibung der Domäne anzufertigen [31]. Diese Programme haben den Vorteil der Flexibilität hinsichtlich beliebiger Änderungen an den erstellten Skripten und Diagrammen. Gleichzeitig weisen sie jedoch den Nachteil auf, dass die resultierenden Texte und Zeichnungen keine verwertbaren Strukturinformationen besitzen. Eine Wiederverwendung dieser informalen Artefakte im weiteren Software-Entwicklungszyklus ist mithin nicht oder nur in sehr eingeschränktem Maße möglich [31, 49, 113]. Eine populäre Vorgehensweise, um aus textuellen Artefakten strukturelle Informationen zu gewinnen, ist die von Rumbaugh in [123] eingeführte Substantivmethode. Mit deren Hilfe werden in den verschiedenen, im Zuge der Anforderungsanalyse erstellten Dokumenten alle Substantive markiert, da sie bei Verwendung der objektorientierten Programmierung als potentielle Klassenkandidaten in Frage kommen. Nachdem dies manuell geschieht, gibt es keine im Rechner manifestierte Beziehung zwischen den ursprünglichen

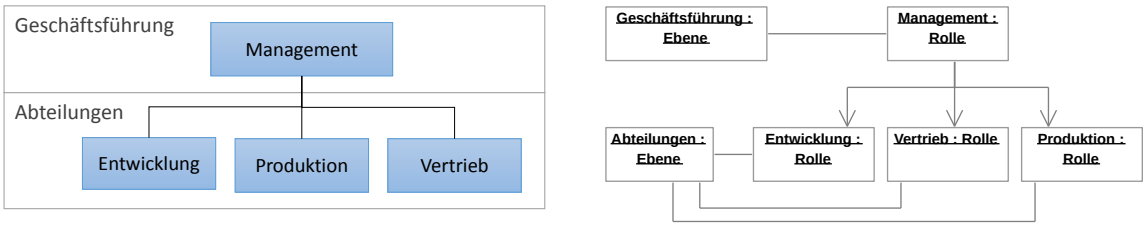

Abbildung 1-1: Beispielhafte Organisationsstruktur eines Unternehmens in verbreiteter Organigrammnotation (links) und als UML Objektdiagramm (rechts)

Artefakten und den letztlich abgeleiteten Klassen. Das ist jedoch ein erstrebenswerter Zustand, weil es auf diese Weise möglich ist, den abgeleiteten Code auf die zugrundeliegenden Anforderungen zurückzuführen.

Alternativ zur Definition neuer DSMLs kann es in einigen Fällen sinnvoll sein, auf **existierende DSMLs** zurückzugreifen und diese gegebenenfalls **an spezifische Bedürfnisse anzupassen**. Nach Auffassung der Autoren von [73] sollte diese Vorgehensweise sogar bevorzugt werden, weil dadurch in vielen Fällen eine Zeitersparnis durch Wiederverwendung erreicht werden kann. Allerdings muss dazu der IT-Spezialist zunächst die Sprachspezifikation in weiten Teilen verstehen, was wegen der Komplexität bei vielen DSMLs einen nicht zu unterschätzenden Aufwand bedeutet. Dieses Problem lässt sich nie ganz vermeiden, doch soll der initiale Aufwand des Verstehens reduziert werden, so dass auf einfache Weise eine Erweiterung der Sprache durchgeführt werden kann. Zunächst geht es auch hier um die Beseitigung des zusätzlichen, imaginären Klassenbildungsschritts, der mit dem Vordenken der Sprache (siehe Problem 1) vergleichbar ist. Darüber hinaus muss in Erfahrung gebracht werden, an welcher Stelle und mit welchen Mitteln die Sprachspezifikation anzupassen ist. Dieser Umstand stellt das zweite Problem dar, das es zu lösen gilt.

Problem 2 *Identifikation notwendiger Änderungen an einer existierenden Sprache*

Ein Beispiel für eine domänenspezifische Erweiterung ist die Einführung eines weiteren Entitätstyps in die Entity-Relationship (ER)-Sprache [19]. In vielen Anwendungen in der Praxis ist es üblich, für den Primärschlüssel einer Entität ein Surrogatattribut zu definieren, das neben der Identifikation keine weitere Funktion besitzt. Deshalb soll der neue Typ automatisch über einen Surrogatschlüssel verfügen, ohne dass er explizit in Form eines Attributs spezifiziert werden muss. Das hat zum einen den Vorteil des geringeren Aufwands beim Anlegen einer Entität und zum anderen steigert seine Verwendung die Übersichtlichkeit von ER-Diagrammen, da lediglich die fachlichen Attribute visualisiert sind. Nachdem es sich hierbei um eine Erweiterung der ER-Sprache handelt, können auch weiterhin reguläre Entitäten mit benutzerdefiniertem Primärschlüssel modelliert werden. Zur Veranschaulichung zeigt Abbildung 1-2 das Datenmodell einer stark vereinfachten Personalverwaltung in zwei verschiedenen Ausprägungen, nämlich einerseits als Standard ER-Diagramm und anderseits als erweitertes ER-Diagramm. Die Erweiterung ergibt sich daraus, dass nun der besagte zusätzliche Entitätstyp verwendet wird, der in Form eines grünen Rechtecks mit abgerundeten Ecken dargestellt ist und nicht mehr über explizit modellierte Primärschlüsselattribute verfügen muss. Die beiden grünen Entitäten weisen im Gegensatz zu ihren blauen Pendants somit nicht mehr das Attribut `person_id` bzw. `address_id` auf. Aus Gründen der Übersichtlichkeit wurde auf die Beschriftung von Kanten verzichtet.

Beide im Vorfeld geschilderten Probleme 1 und 2 sind der bisherigen Beschränkung auf die **Top-Down Vorgehensweise** bei der Definition, Manipulation und Verwendung von DSMLs geschuldet. Bevor ein Sprachkonstrukt verwendbar ist, muss dieses zunächst formal spezifiziert werden. Das Verwenden von bereits definierten Konstrukten bei gleichzeitiger Beachtung von durch eine Sprache vorgegebenen Rahmenbedingungen und Constraints bezeichnen wir im Folgenden als **stringentes Modellieren**.

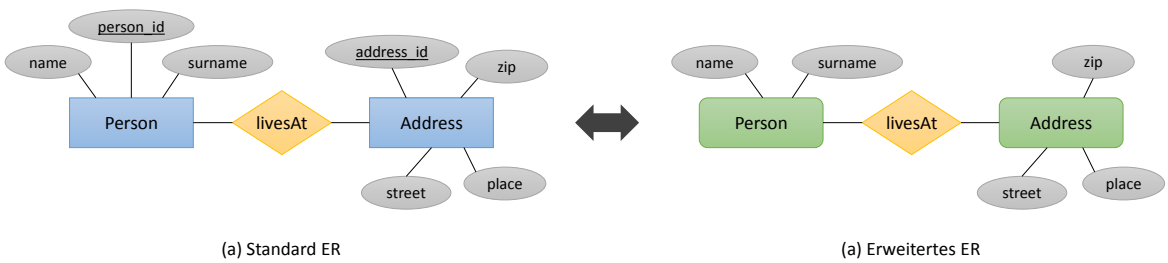

Abbildung 1-2: Beispieldiagramm für Standard ER und erweitertes ER

Derartige Constraints führen bei ER-Diagrammen beispielsweise dazu, dass zwei Attribute unterschiedlicher Entitäten nicht direkt miteinander verbunden werden dürfen.

Im Buch „Sprachbasierte Informatik" [112] schildert Ortner einen alternativen Ansatz zur Entwicklung von Sprachen, der auf dem Prinzip des Erlanger Konstruktivismus [88] beruht. Dieser sieht vor, dass alle Sprachen, die zum Entwurf eines Softwaresystems nötig sind, aus der praktischen Anwendung des Systems selbst rekonstruiert werden können bzw. sollen. Hierzu liefert Ortner in [112] und [143] ein theoretisches Vorgehensmodell für Entwickler, das die Sprachrekonstruktion beschreibt. Nach Ortners Ansicht widerspricht die Sprachrekonstruktion dem herkömmlichen Top-Down Modellierungsprozess, der als ersten Schritt die Definition von konkreter und abstrakter Syntax fordert. Erst danach können unter Verwendung dieser Syntaxen Modelle erstellt werden. Im Gegensatz dazu beschreibt Sprachrekonstruktion die Induktion einer Sprache aus bereits existierenden Artefakten und damit eine **Bottom-Up Vorgehensweise**. Der Benutzer muss also die Möglichkeit erhalten, beliebige Konstrukte verwenden und diese frei miteinander kombinieren sowie parametrieren zu können. Dies entspricht der **freien Modellierung**, wie sie – bewegt man sich in der grafischen Welt – von Programmen wie Microsoft Visio, Microsoft PowerPoint oder Dia zur Erstellung von Diagrammen unterstützt wird. Bezogen auf das zuvor genannte Beispiel eines ER-Diagramms bedeutet dies, dass es durchaus möglich ist, zwei Attribute verschiedener Entitäten miteinander zu verbinden.

Ein **weiteres Anwendungsfeld der freien Modellierung** ist die vorwiegend in Großbritannien populäre **Soft Systems Methodology** (SSM) [18]. Bei ihr handelt es sich um eine Analyse- und Designmethodik zum Zweck der Systemanalyse. Insbesondere die im Rahmen von SSM entworfenen konzeptuellen Modelle können als Ausgangspunkt einer neuen Modellierungssprache dienen. Die freie Modellierung ist daher eine durchaus verbreitete Vorgehensweise bei der Erstellung von Modellen.

Die **generelle Problematik** des herkömmlichen Modellierungsprozesses (Beschränkung auf Top-Down Vorgehensweise und stringentes Modellieren) ist bereits **in der DSM-Community allgemein erkannt**, was zahlreiche Veröffentlichungen zu dieser Thematik zeigen [5, 23, 27, 31, 49, 93, 113, 124]. Die einzelnen Ansätze und Systeme hinter diesen Publikationen werden in Kapitel 3 über verwandte Arbeiten näher analysiert und beschrieben. Vorgreifend kann schon gesagt werden, dass zum gegenwärtigen Zeitpunkt kein System existiert, das sowohl freies als auch stringentes Modellieren gleichermaßen unterstützt. Freies Modellieren ermöglicht eine natürliche und intuitive Erstellung von DSMLs (bottom-up), während stringentes Modellieren ein durch die Sprache limitiertes, aber dafür geführtes Arbeiten gestattet (top-down). Demzufolge ist es vorteilhaft beide Vorgehensweisen zu kombinieren.

Um von freier und stringenter Modellierung gleichermaßen profitieren zu können, ist eine **tiefe Integration im Rahmen einer Werkzeugunterstützung** erforderlich. Sie ermöglicht zunächst die initiale Erstellung einer DSML, indem beispielhaft grafische oder textuelle Modelle skizziert werden und daraus die Sprache abgeleitet wird (bottom-up). Anschließend können weitere, auf ihr basierende Modelle gebaut werden (top-down). Reicht die Ausdrucksstärke der Sprache irgendwann nicht mehr aus oder

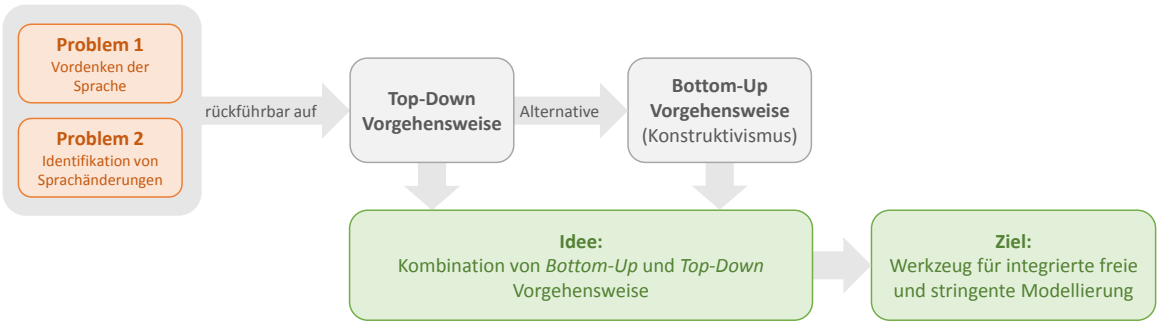

Abbildung 1-3: Von der Problemstellung zur Lösungsskizze

sind sonstige Änderungen erforderlich, kann die DSML analog zur konstruktivistischen Idee durch freie Modellierung und anschließende Sprachinduktion erweitert oder modifiziert werden. Darüber hinaus können Sprachänderungen auch der Top-Down Vorgehensweise folgend bewerkstelligt werden. Insbesondere für den Fall der direkten Interpretation von Modellen sind Anpassungsmöglichkeiten auf Sprachebene unverzichtbar, da nur IT-Experten mit Kenntnissen über die Implementierungslogik (z.B. Code-Generierung) gegebenenfalls notwendige Optimierungen vornehmen können. Eine zusammenfassende Übersicht über die gesamten Argumentationskette angefangen bei der Problemstellung hin zur Lösungsskizze zeigt Abbildung 1-3.

1.2 Anforderungen

Die vorliegende Arbeit entstand im Umfeld der Arbeitsgruppe „Meta-Modellierung" des Lehrstuhls für Angewandte Informatik IV. DSMLs werden deshalb mithilfe von Meta-Modellen definiert. Eine genaue Begründung hierfür ist den Abschnitten 2.1.2 und 2.1.3 zu entnehmen.

Für die Herleitung der Anforderungen an ein System, das freie und stringente Modellierung zulässt (Abschnitt 1.1), ist die **orthogonale Klassifikation** von zentraler Bedeutung. Auf ihr basiert die in [142] eingeführte und in Abschnitt 2.2.3 näher beschriebene linguistische Meta-Sprache (LML). Die ursprüngliche Definition der orthogonalen Klassifikation stammt von Atkinson und Kühne [9] und beschreibt eine Trennung von Struktur und Inhalt bei (Meta-)Modellen. Struktur meint in diesem Fall die interne Repräsentation von Modellen und Meta-Modellen und damit das Abspeicherungsformat der jeweils enthaltenen Elemente. Mit der LML können daher sowohl Modelle als auch zugehörige Meta-Modelle spezifiziert werden und stellen somit linguistische Instanzen dar. Die Beziehung zwischen einem Modell und dem zugehörigen Meta-Modell steht orthogonal zur vorher genannten linguistischen Instanziierung und wird als inhaltliche bzw. ontologische Instanziierung bezeichnet. Die Konstrukte einer DSML liegen also in einem Meta-Modell als sogenannte Typen vor, während die zugehörigen Instanzen Teil der darauf basierenden Modelle sind. Weiterführende Erläuterungen zu dem Meta-Modellierungsmuster „orthogonale Klassifikation" können Abschnitt 2.2.1.1 entnommen werden.

Insgesamt werden in diesem Kapitel drei Anforderungen an ein System hergeleitet, das sowohl freie als auch stringente Modellierung unterstützt. Dabei können lediglich die beiden ersten Anforderungen direkt aus der Kombination von freier und stringenter Modellierung abgeleitet werden. Sie liefern eine konkrete Herangehensweise an das Problem, wie DSMLs konstruktivistisch entwickelt und eingesetzt werden können, und zwar einerseits aus ontologischer (Anforderung 1) und andererseits aus linguistischer Sicht (Anforderung 2). Die letzte Anforderung (Anforderung 3) stellt hingegen eine Ergänzung der beiden anderen dar, da sie auf einer (semi-)automatischen Berücksichtigung von Qualitätskriterien der jeweils abgeleiteten Sprache beruht.

1.2.1 Konstruktivistische Entwicklung von DSMLs im ontologischen Kontext

Das Hauptaugenmerk von Anforderung 1 liegt auf der Konzeption eines Werkzeugs, das die nahtlose Integration von freier und stringenter Modellierung im ontologischen Kontext erfüllt. Ontologischer Kontext bedeutet, dass sämtliche linguistischen Artefakte der LML verborgen sind und der Benutzer sich deswegen vollständig auf die jeweilige Domäne und deren Vokabular konzentrieren kann. Ontologische Modellierung entspricht demzufolge der typischen Erstellung von Modellen mithilfe von DSMLs, die über eine abstrakte Syntax und eine konkrete Syntax verfügen. Auf den Punkt gebracht kann die erste Anforderung wie folgt formuliert werden:

Anforderung 1 *Konstruktivistische Entwicklung von DSMLs im ontologischen Kontext*

Der Benutzer arbeitet dabei stets mit visuellen Artefakten (vorwiegend grafisch oder textuell), die von ihren Wesensmerkmalen her der konkreten Syntax zuzuordnen sind. Während der Modellierung kann er auf bereits existierende Konstrukte zurückgreifen oder bei Bedarf neue erstellen, indem er sie direkt innerhalb des Dokuments definiert und gleichzeitig verwendet. Ein Beispiel für eine domänenspezifische Erweiterung ist bereits in Abschnitt 1.1 für Problem 2 vorgestellt worden. Es handelt sich um einen zusätzlichen Entitätstyp bei ER-Diagrammen, dessen Instanzen automatisch ein identifizierendes Surrogatattribut besitzen. Konträr zur gewohnten Vorgehensweise soll die ER-Sprache nicht durch formales Spezifizieren dieses Konstrukts erweitert werden, sondern stattdessen der freien Modellierung folgend durch direktes Zeichnen einer neuen Instanz innerhalb eines ER-Diagramms (z.B. wie in Abbildung 1-2 als grünes Rechteck mit abgerundeten Ecken). Die eigentliche Sprachanpassung erfolgt in einem nächsten Schritt über einen automatischen Mechanismus. Der Modellierer kann sich also weiterhin auf die Domäne konzentrieren und muss nicht den Modellierungsfluss unterbrechen, um den neuen Entitätstyp in der abstrakten Syntax als konzeptuelles Element und in der konkreten Syntax als visuelle Repräsentation zu spezifizieren.

Neben den grafischen DSMLs existiert das breite und gleichberechtigte Feld der textuellen DSMLs. Der Hauptunterschied zwischen beiden Arten liegt in der konkreten Syntax. Während sich diese Syntax bei grafischen Sprachen aus Formen zusammensetzt [28], die zueinander in Beziehung gesetzt werden können, erfolgt bei textuellen Sprachen eine Aneinanderreihung beliebiger Zeichen zu Text. Nachdem der Fokus auf DSMLs und damit auf formalen Sprachen liegt, wird das komplexe Gebiet der Analyse natürlicher Sprachen nicht berücksichtigt. Texte formaler Sprachen werden mithilfe von Parsern analysiert und in ein allgemeines Datenformat – eine baumartige Struktur – überführt [1]. Die Blätter des resultierenden Parse-Baums sind dann die einzelnen Tokens (typischerweise Wörter und Interpunktionszeichen) des Eingabetexts.

Demgemäß entsprechen die Tokens bei textuellen Sprachen weitgehend den Formen auf grafischer Seite. Strukturell betrachtet können textuelle und grafische Sprachen sowie darauf basierende Modelle im selben Format abgelegt werden. Daher kann auch die Interpretation und Weiterverarbeitung dieser Sprachen und Modelle auf dieselbe Weise erfolgen.

```
1  // Standard ER (1. Fall)
2  entity Person
3      identifier integer person_id
4      string name
5      string surname
6
7  // Erweitertes ER (2. Fall)
8  identified entity Person
9      string name
10     string surname
```

Listing 1-1: Textuelles Beispiel für Standard ER und erweitertes ER

Angelehnt an das Beispiel aus Abbildung 1-2 zeigt Listing 1-1 die Entität `Person` in zwei unterschiedlichen Ausprägungen, jedoch unter Verwendung einer textuellen DSML. Im ersten Fall folgt sie den Standardregeln der ER-Modellierung, weshalb die Entität das explizit spezifizierte Primärschlüsselattribut `person_id` besitzt. Im zweiten Fall hingegen ist dieses Attribut nicht mehr vorhanden, da die Entität stattdessen als `identified` deklariert ist und somit implizit darüber verfügt. Ausgangspunkt ist also die obige Entität inklusive einer textuellen ER-Sprache, die die bei ER-Diagrammen typischen Regeln vorgibt. Diese ER-Sprache soll nun analog zur freien Modellierung bei grafischen DSMLs erweitert werden können, und zwar indem einfach vor das Schlüsselwort `entity` ein weiteres

Schlüsselwort `identified` geschrieben und als solches deklariert wird. Die eigentliche Sprachanpassung geschieht auch hier separat und automatisch, so dass keine Unterbrechung des Modellierungsflusses erforderlich ist.

Gemäß der beiden Arten von DSMLs (grafisch und textuell) lässt sich die erste Anforderung in das konstruktivistische Entwickeln von grafischen Sprachen (**Anforderung 1a**) und das konstruktivistische Entwickeln von textuellen Sprachen (**Anforderung 1b**) unterteilen. Als Herausforderung ergibt sich die Entwicklung eines Ansatzes mit Werkzeugunterstützung, der sowohl freies als auch stringentes Modellieren im Rahmen von grafischen sowie textuellen DSMLs ermöglicht. Aus konzeptioneller Hinsicht soll dabei besonderer Wert auf die Extraktion von Gemeinsamkeiten gelegt werden, damit eine Kombination von grafischer und textueller Welt möglich ist. Die spezifische Umsetzung einer Werkzeugunterstützung für die Erfüllung der beiden Teilanforderungen ist weniger eine konzeptionelle Herausforderung, sondern vielmehr eine Problemstellung im Bereich des Software Engineerings. Aufgrund ihrer Komplexität und Neuartigkeit wird diese Problemstellung im Rahmen dieser Dissertation eingehend behandelt.

1.2.2 Konstruktivistische Entwicklung von DSMLs im linguistischen Kontext

Analog zur ersten Anforderung ist bei Anforderung 2 das Ziel die Konzeption eines Werkzeugs, das die nahtlose Integration von freier und stringenter Modellierung nun aber im linguistischen Kontext erfüllt. Linguistischer Kontext meint die Verwendung der LML zur Erstellung von Modellen mit oder ohne vorgegebene Sprache. Diese Sprache ist ebenfalls wieder mithilfe der LML formuliert. Im Unterschied zur ontologischen Modellierung ist die konkrete Syntax bei der linguistischen Variante schon durch das LML vorgegeben und muss nicht erst abgeleitet bzw. spezifiziert werden. Demzufolge bedeutet freie, linguistische Modellierung nicht, dass in einem weiteren Schritt eine neue LML abgeleitet wird, sondern dass die Ableitung ebenso in ontologischer Richtung geschieht, allerdings beschränkt auf die abstrakte Syntax. Die zweite Anforderung lässt sich kurz und prägnant folgendermaßen in Worte fassen:

Anforderung 2 *Konstruktivistische Entwicklung von DSMLs im linguistischen Kontext*

Mit konstruktivistischer Entwicklung von DSMLs im linguistischen Kontext sollen Modellierungsspezialisten auf die gleiche Weise unterstützt werden wie Domänenexperten ohne tiefgehende Modellierungserfahrung, die aber im ontologischen Kontext arbeiten. Ein typischer Anwendungsfall ist die Bereitstellung von Konfigurations- oder Testdaten, ohne sich hierfür eine abstrakte Syntax zu überlegen und diese im Vorfeld manuell definieren zu müssen. Weil die **abstrakte Syntax automatisiert** aus den vorgegebenen Beispielmodellen **induziert** wird, adressiert diese zweite Anforderung ebenfalls Problem 1 (das Vordenken der Sprache).

Angewendet auf das obige Beispiel einer Erweiterung der ER-Sprache (Abbildung 1-2) zeigt Abbildung 1-4 das ursprüngliche und das erweiterte Beispielmodell in Form linguistischer Konstrukte. Dazu wird die bekannte Notation der UML Objektdiagramme [111] verwendet, weil Aufbau und Struktur der LML erst in Abschnitt 2.2.2 beschrieben werden. Diese Notation repräsentiert somit eine vereinfachte, grafische Alternative zur LML. Wie bei Objektdiagrammen üblich, werden Objekte durch Rechtecke visualisiert, die mit dem jeweiligen Namen sowie dem Objekttyp (durch einen Doppelpunkt voneinander getrennt) beschriftet sind. Existierende Typen, die im Diagramm Verwendung finden und in der abstrakten Syntax definiert sind, sind `Entity`, `Attribute` und `Relationship`. Einen Typ `IdentifiedEntity` dagegen gibt es noch nicht. Er muss erst in einem weiteren Schritt abgeleitet und in die abstrakte Syntax der ER-Sprache integriert werden.

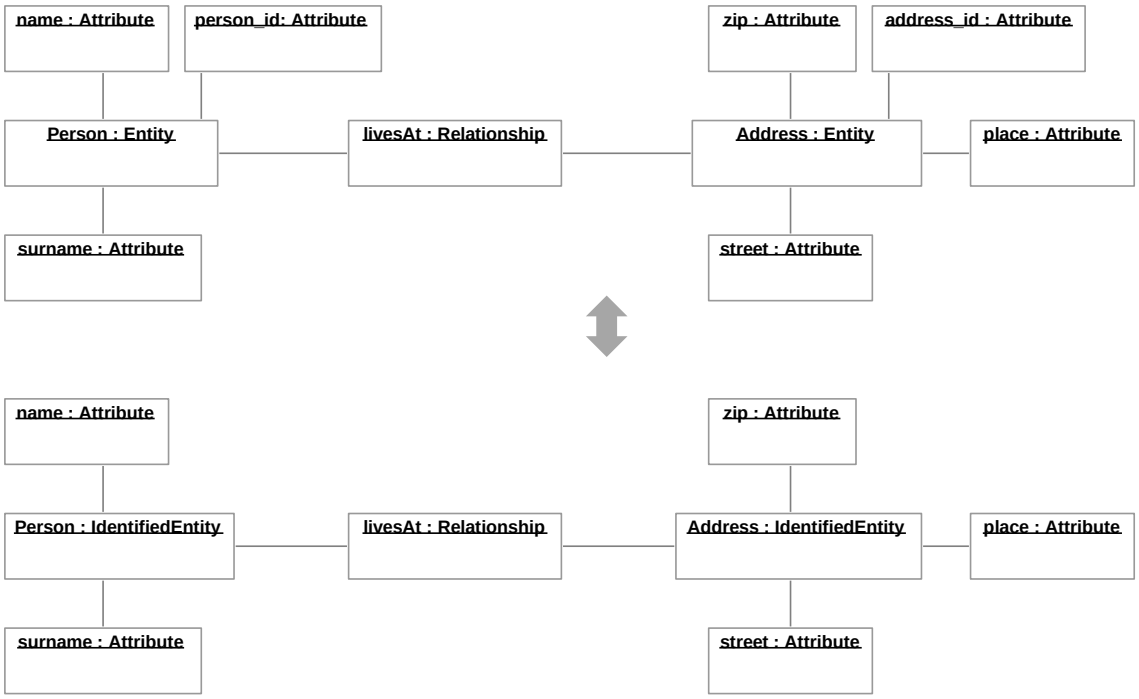

Abbildung 1-4: Linguistisches Beispiel für Standard ER (oben) und erweitertes ER (unten) in UML Objektdiagramm-Notation

1.2.3 Berücksichtigung von Qualitätskriterien

Anforderung 3 gilt der Qualität einer abgeleiteten DSML und resultiert daher nicht direkt aus den zwei Problemen aus Abschnitt 1.1. Allerdings ist sie die logische Konsequenz aus Anforderung 1 und Anforderung 2, bei denen das Ableiten einer Sprache einen wesentlichen Aspekt darstellt. Sie kann folgen-dermaßen prägnant formuliert werden:

Anforderung 3 *Berücksichtigung von Qualitätskriterien*

Für die Weiterverarbeitung einer Sprache (Interpretation, Code-Erzeugung etc.) ist insbesondere ihre abstrakte Syntax von Bedeutung, die in Form eines Meta-Modells vorliegt. Die dritte Anforderung richtet sich demnach primär an den Modellierungsexperten, der direkt in Kontakt mit diesem Meta-Modell steht. Deshalb ist es wichtig, dass es ausdrucksstark aber dennoch leicht verständlich ist. Um das zu erreichen, haben Bertoa und Vallecillo in [13] Qualitätskriterien ermittelt, die von jedem Meta-Modell beachtet werden sollten.

Ein typischer Ansatz, um die Qualität eines Meta-Modells zu steigern, ist die **Verwendung von Entwurfsmustern**, weil sie in der Regel oft für bestimmte Problemszenarien eingesetzt werden und mithin einen großen Bekanntheitsgrad aufweisen. Das Standardwerk zu dieser Thematik ist das Buch der Gang of Four [50], in dem zahlreiche Entwurfsmuster für die objektorientierte Programmierung beschrieben sind. In Anlehnung daran greifen Cho und Gray einige Muster auf [22], die in vielen grafischen DSMLs zum Einsatz kommen, und wenden sie direkt beim Ableiten eines Meta-Modells an. Dies ist nur deshalb möglich, da die Diagramme weitere Informationen enthalten (z.B. mögliche Unterscheidung zwischen Knoten und Kanten), auf die direkt zugegriffen werden kann. Folglich sind die unterstützten Muster stark auf grafische Sprachen fokussiert und können nicht generisch auf beliebige Meta-Modelle angewendet werden.

Nachdem die vorliegende Arbeit auf der LML aufsetzt, kann jedes (Meta-)Modell von den integrierten Sprachmustern (Einfachvererbung, Enumerationen und Powertypen [108], siehe Abschnitt 2.2.1) profitieren. Im Gegensatz zu den auf grafische Sprachen spezialisierten Entwurfsmustern ist die automatische Anwendung von Sprachmustern allerdings nicht realisierbar, weil den linguistischen Beispielmodellen die notwendigen Informationen fehlen. Alternativ sollen für jedes von der LML unterstützte **Sprachmuster** Regeln aufgestellt werden, anhand derer Konstellationen von Meta-Modellelementen identifiziert werden, die als Kandidat für die **Anwendung** des jeweiligen Musters in Frage kommen. Beispielsweise soll ein Automatismus erkennen können, auf welche Menge von Elementen es sinnvoll ist das Powertyp-Muster (Abschnitt 2.2.1.4) anzuwenden. Die gefundenen Kandidaten können dem Benutzer schließlich als Vorschlag für die Einführung des entsprechenden Sprachmusters unterbreitet werden. Allerdings ist anzumerken, dass das Qualitätsempfinden stets subjektiven Vorlieben unterworfen ist. Deswegen ist es unumgänglich eine Einflussnahme auf den Identifizierungsprozess zu ermöglichen, so dass Art und Anzahl der Vorschläge an die Bedürfnisse des Modellierungsexperten heranreichen.

1.3 Eigener Beitrag

Das primäre Ziel dieser Arbeit ist die Unterstützung eines neuartigen, auf Prinzipien des Konstruktivismus beruhenden Modellierungsprozesses, mit dessen Hilfe DSMLs erstellt, angepasst und verwendet werden können. Bei der Umsetzung der Bottom-Up Vorgehensweise beschränken wir uns auf die **Entwicklung der strukturellen DSML-Komponenten** (also abstrakte und konkrete Syntax). Für die Entwicklung von Constraints basierend auf Beispielen käme man nicht daran vorbei, sogenannte negative Beispielmodelle bereitzustellen [133]. Dabei handelt es sich um Modelle, die ungültig im Sinne der Domäne sind, da sie bestimmte Regeln verletzen. Die Bereitstellung negativer Beispiele ist aus unserer Sicht aber nicht praktikabel, weil dadurch der Benutzer seinen Fokus von den tatsächlichen Objekten der Realität lösen und sich auf Fälle konzentrieren müsste, die außerhalb der betreffenden Domäne liegen.

Das **neuartige Vorgehensmodell** bildet den Kern des eigenen Beitrags und ist in Abbildung 1-5 dargestellt. Die einzelnen Schritte sind in Form von Pfeilen angegeben, welche den Informations- und Aktionsfluss zwischen den drei für DSM relevanten Meta-Ebenen symbolisieren. Unten angefangen

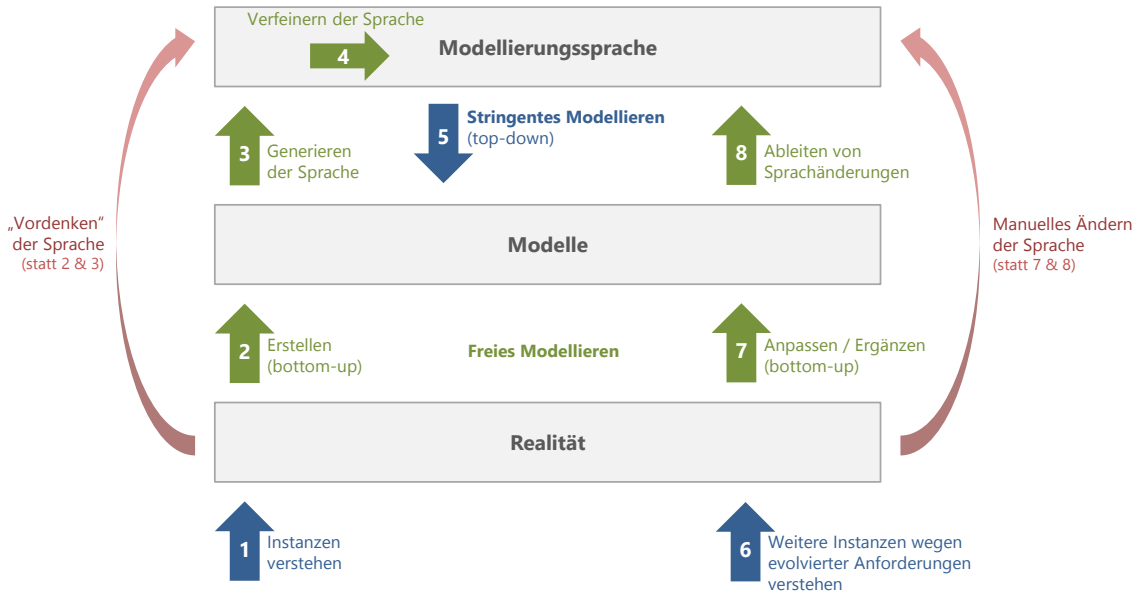

Abbildung 1-5: Neuartiger Modellierungsprozess für die Entwicklung und Verwendung von DSMLs

zeigt sich die Ebene der Realität, die die existierenden Entitäten der realen Welt beinhaltet. Darüber liegt die Ebene der Modelle, deren Elemente als Typen der zuvor genannten Entitäten aufgefasst werden können. An oberster Stelle befindet sich die Ebene der Modellierungssprache, die wiederum die Typen der darunter angesiedelten Modellelemente definiert.

Die eigentliche Vorgehensweise ist wie folgt zu deuten: Liegt noch keine DSML vor, so muss stets mit Schritt 1 begonnen und die Entitäten der realen Welt verstanden werden. Danach kann der Benutzer bereits durch freies Modellieren initiale Beispielmodelle skizzieren (Schritt 2), aus denen in Schritt 3 automatisiert eine Sprache induziert wird. Darüber hinaus kann ebenfalls durch einen semi-automatischen Mechanismus eine Verfeinerung der Sprache (z.B. unter Anwendung von Sprach- und Entwurfsmustern) durchgeführt werden (Schritt 4). Alternativ zu den Schritten 2 und 3 kann sich der Benutzer auch für die Top-Down Vorgehensweise entscheiden und die Modellierungssprache formal auf oberster Ebene spezifizieren. Hierfür ist jedoch ein Vordenken dieser Sprache erforderlich, da wegen des gewählten Ansatzes noch keine Modelle gebaut werden konnten. Ist dann ein erster Sprachentwurf vorhanden, kann stringent modelliert und auf diese Weise die DSML verwendet werden.

Reicht die Ausdrucksstärke einer Modellierungssprache nicht mehr aus, weil sich beispielsweise die Anforderungen geändert haben, dann gilt es zunächst diesen Wandel zu verstehen (Schritt 6). Anschließend kann der Benutzer erneut entscheiden, ob er einfach die existierenden Beispielmodelle entsprechend modifizieren bzw. neue Modelle erstellen möchte (Schritt 7) oder ob er stattdessen die Sprache manuell anpassen will. Wählt er die erste Variante, erfolgt im sich anschließenden Schritt 8 eine automatische Ableitung von Sprachänderungen. Die angepasste DSML kann nun ebenfalls gemäß Schritt 4 verfeinert werden. Danach beginnt der Zyklus von vorne und es kann solange stringent modelliert werden (Schritt 5), bis eine weitere Änderung der Sprache unausweichlich ist.

Neben dem Vorgehensmodell lässt sich der **eigene Beitrag** noch weiter untergliedern. Die extrahierten Unterpunkte können dabei wiederum in zwei Kategorien eingeordnet werden, nämlich in solche, die konzeptueller Natur sind, und solche, die dem Software Engineering zuzurechnen sind. Die Punkte der konzeptuellen Kategorie können wie folgt differenziert werden.

- Für das automatische Ableiten eines Meta-Modells aus einem gegebenen Satz von Beispielmodellen werden Algorithmen benötigt. Nachdem als Datenformat für die Abspeicherung von Modellen und Meta-Modellen die LML zum Einsatz kommt, wird besonderer Wert darauf gelegt, dass die von ihr bereitgestellten Möglichkeiten (z.B. Generalisierung/Spezialisierung, Erweiterter Powertyp) innerhalb der Algorithmen berücksichtigt und genutzt werden. Das Generieren der abstrakten Syntax ist sowohl im linguistischen als auch im ontologischen Kontext von essentieller Bedeutung, weshalb der erste Beitrag (Bereitstellung von Ableitungsalgorithmen) indirekt von Anforderung 1 und Anforderung 2 bedingt wird.
- Basierend auf Anforderung 1 werden im ontologischen Kontext grafische und textuelle Editoren unterschieden. Für beide wird jedoch ein gemeinsames Datenmodell konzipiert, das für spezifische Belange des jeweiligen Editortyps ergänzt werden kann. Damit ist der Ansatz auf eine Integration von grafischer und textueller Welt vorbereitet und bietet gleichzeitig eine Allgemeingültigkeit, von der weitere Arten von DSMLs (z.B. in Tabellenform) profitieren können.
- Die Umsetzung von Anforderung 3 liefert ein Framework, das für jedes geeignete und von der LML unterstützte Sprachmuster Regeln anbietet, die – angewendet auf ein Meta-Modell – Konstellationen von Modellelementen zurückliefern, welche mithilfe des betreffenden Musters verfeinert werden können. Ein Modellierungsexperte wird dadurch auf Stellen im Meta-Modell hingewiesen, die Verfeinerungspotential besitzen. Durch das zweckmäßige Einsetzen derartiger Sprachmuster wird die Qualität von Meta-Modellen positiv beeinflusst [61, 103].

Die verbleibenden Aspekte des eigenen Beitrags betreffen die Umsetzbarkeit der konstruktivistischen Idee für die Entwicklung von DSMLs. Ein hoher Stellenwert wird hierbei der Benutzerfreundlichkeit und Handhabbarkeit der resultierenden Werkzeugunterstützung beigemessen, die wesentlich für die Akzeptanz des neuartigen Modellierungsprozesses ist. Die folgenden vier Punkte weisen deshalb einen starken softwaretechnischen Bezug auf.

- Aufgrund von Anforderung 1a wird ein grafischer Editor konzipiert, der sowohl freies als auch stringentes Modellieren unterstützt. Im Fokus steht dabei, dass der Benutzer nicht in seinem Arbeitsfluss behindert wird und sich vollständig auf die Modellierung der Domäne konzentrieren kann. Dies bedingt die Unterstützung von Arbeitsweisen (z.B. Drag & Drop) und die Bereitstellung von Hilfsfunktionen (z.B. Ausrichten von Elementen an anderen Objekten), wie sie von verbreiteten Werkzeugen (Microsoft Visio, Dia, Visual Paradigm, MetaEdit+ etc.) angeboten werden.
- Wegen Anforderung 1b wird ein textueller Editor konzipiert, der sowohl freies als auch stringentes Modellieren unterstützt. Auch hier liegt der Fokus darauf, dass der Benutzer in seinem Arbeitsfluss nicht behindert wird. Erreicht wird dies vorrangig dadurch, dass – wie für Text-Editoren üblich – sämtliche Eingaben und Manipulationen am Dokument mittels einer Tastatur erfolgen können. Dabei kann der Benutzer auf Features zurückgreifen, die er von strukturellen Texteditoren gewohnt ist (z.B. Syntaxhervorhebung und Code-Vervollständigung).
- Bei der Modellierung im linguistischen Kontext ist die konkrete Syntax bereits durch die LML vorgegeben. Die Spezifikation der LML in Abschnitt 2.2 ist allerdings auf stringentes Modellieren ausgerichtet und bietet daher nicht die notwendige Flexibilität für die freie Variante. Demzufolge wird eine diesbezügliche Dynamisierung der LML eingeführt und der textuelle Editor für die Unterstützung der freien Modellierung entsprechend erweitert (Anforderung 2).
- Anforderung 3 motiviert die Schaffung einer Möglichkeit für den Benutzer, Einfluss auf den Ableitungs- und den Verfeinerungsprozess nehmen zu können. Der Fokus liegt hierbei auf der Identifikation von Stellen, an denen die Einflussnahme sinnvoll ist.

1.4 Struktur der Arbeit

Die vorliegende Arbeit erstreckt sich über insgesamt neun Kapitel. Im ersten Kapitel wurde bereits die Problematik beim herkömmlichen Entwickeln von DSMLs geschildert und mit dem beispielgetriebenen Ansatz eine grobe Lösungsskizze inklusive relevanter Anforderungen vorgestellt. Dieser Ansatz liegt im Zentrum der späteren Ausführungen.

Vor der Vertiefung des Ansatzes werden in Kapitel 2 zunächst einige Grundlagen erklärt, die für das Verständnis der weiteren Erläuterungen Bewandtnis haben. Daran schließt sich Kapitel 3 an, innerhalb dessen es um die Analyse von Arbeiten geht, die sich im weitesten Sinne mit neuartigen Methoden zur Entwicklung von Modellierungssprachen befassen. Ansätze, die dabei mit unserem vergleichbar sind, werden zudem im Hinblick auf die in Abschnitt 1.2 aufgestellten Anforderungen evaluiert.

Kapitel 4 liefert mit der beispielgetriebenen Entwicklung von DSMLs im linguistischen Kontext eine Lösung für Anforderung 2. Außerdem wird mit der automatisierten Unterbreitung von Verfeinerungsvorschlägen für induzierte Meta-Modelle auch Anforderung 3 adressiert. Sie findet aber ebenso in späteren Kapiteln Beachtung.

Die Kapitel 5, 6 und 7 betreffen allesamt Anforderung 1 und damit die beispielgetriebene Entwicklung von Modellierungssprachen im ontologischen Kontext. Im fünften Kapitel wird zuerst eine allgemeine Herangehensweise vermittelt und ein gemeinsames Meta-Modell präsentiert, das zur Beschreibung

beliebiger Visualisierungen (sowohl grafisch als auch textuell) eingesetzt werden kann. Kapitel 6 beschäftigt sich dann mit einer konkreten Lösung für grafische DSMLs (Anforderung 1a), während Kapitel 7 textuelle DSMLs behandelt (Anforderung 1b).

In Kapitel 8 wird der Prototyp vorgestellt, der im Rahmen dieser Dissertation entwickelt wurde. Darüber hinaus wird ein Anwendungsszenario präsentiert, das die Entwicklung textueller Modellierungssprachen nach dem in Kapitel 7 eingeführten beispielgetriebenen Ansatz erläutert. Zum Schluss erfolgt in Kapitel 9 ein Resümee über die gesamte Arbeit sowie ein Ausblick auf potentiell zukünftige Forschungs- und Ingenieurstätigkeiten im Umfeld der beispielgetriebenen Entwicklung von DSMLs.

Im Verlauf der Arbeit sind einzelne Textfragmente unterschiedlich formatiert. `Äquidistanter Text` bezieht sich stets auf Beschriftungen von Elementen, die in Diagrammen, Listings oder Tabellen vorkommen. **Fett gedruckter Text** hebt die zentralen Begriffe des jeweiligen Absatzes hervor. Er soll dabei helfen, schnell eine gewünschte Stelle im Dokument zu finden, ohne dass hierfür Absätze komplett gelesen werden müssen. *Kursiver Text* wird schließlich dazu verwendet, anderweitig wichtige Informationen hervorzuheben, die nicht in die beiden zuvor erwähnten Kategorien passen.

2 Grundlagen

Dieses Kapitel gibt eine kurze Einführung in grundlegende Themengebiete und Methoden. Den Anfang macht eine Übersicht über domänenspezifische Modellierung. Dabei werden zunächst die allgemeinen und im DSM-Umfeld üblichen Termini erklärt. Anschließend wird auf die beiden möglichen Techniken zur Definition von DSMLs eingegangen, wobei der Schwerpunkt auf Meta-Modellierung liegt, denn sie bildet den Kontext für das im darauffolgenden Abschnitt vorgestellte linguistische Meta-Modell. Dieses wiederum repräsentiert die Basisdatenstruktur für sämtliche Modelle, die in den späteren Kapiteln gezeigt werden, und stellt gleichzeitig erweiterte Mechanismen für fortgeschrittene Modellierungsaktivitäten zur Verfügung. Des Weiteren wird die Notation der verschiedenen Diagramme und Dokumente erläutert, die in den einzelnen Abbildungen und Listings Verwendung finden.

2.1 Domänenspezifische Modellierung

Eine kurze Erläuterung, was sich hinter domänenspezifischer Modellierung verbirgt, wurde bereits in der Einleitung gegeben. Demnach liegt der Schwerpunkt auf der Bereitstellung und anschließenden Verwendung von Sprachen, die spezifisch für die Beschreibung von Lösungen innerhalb eines bestimmten Anwendungsgebiets sind. Die einzelnen Termini solcher Sprachen müssen nicht zwingend textueller Natur sein, sondern können auch durch grafische Objekte repräsentiert werden. Primär werden DSMLs deshalb in textuelle und grafische Sprachen unterteilt (letztere werden oftmals auch Diagrammsprachen genannt).

Trotz dieser Differenzierung weisen alle DSMLs zahlreiche elementare Gemeinsamkeiten auf. In den folgenden Unterabschnitten werden diejenigen Gemeinsamkeiten näher betrachtet, die im weiteren Verlauf der Dissertation von Relevanz sind. Den Anfang macht dabei die Einführung diverser Termini aus dem Umfeld von DSMLs. Anschließend werden die beiden Techniken vorgestellt, auf deren Grundlage eigene DSMLs spezifiziert werden können. Einerseits handelt es sich hierbei um formale Grammatiken und andererseits um das breite Spektrum der Meta-Modellierung.

2.1.1 Allgemeine Terminologie

Eine **DSML** setzt sich immer aus den **Komponenten** „konkrete Syntax", „abstrakte Syntax" und einem „Mapping" zwischen diesen beiden zusammen. Häufig ergänzt wird eine DSML um zusätzliche „semantische Regeln", die die Bedeutung der einzelnen, in der abstrakten Syntax definierten Konzepte festgelegt [24]. Neben der Vorgabe von bestimmten Rahmenbedingungen (Constraints), die die Einsetzbarkeit dieser Konzepte einschränkt, wird damit auch assoziiert, wie die Konzepte zu interpretieren oder in eine ausführbare Programmiersprache zu transformieren sind. Der Fokus dieser Arbeit liegt jedoch auf der Struktur von Sprachen, weshalb semantische Regeln (unter anderem auch aus Komplexitätsgründen) nur am Rande beachtet werden.

Die **konkrete Syntax** repräsentiert die Notation und bildet damit die **visuelle Schnittstelle einer Sprache** zum Benutzer. Potentiell darf es zu jeder Sprache mehrere konkrete Syntaxen geben. Diese werden im Regelfall dann genutzt, wenn verschiedene Sichten auf einen bestimmten Sachverhalt

erforderlich sind. Jede konkrete Syntax stellt eine Ansammlung visueller Konstrukte bereit, die der Benutzer während der Modellierung verwendet und daraufhin miteinander kombiniert sowie parametriert. Differenzieren lassen sie sich anhand der eingesetzten Visualisierungsform in grafische und textuelle Konstrukte. Erstere sind typischerweise grafische Formen oder Formenverbünde, die zueinander in Beziehung gesetzt werden können. An manchen Stellen enthalten sie auch Platzhalter für die Eingabe von Text. Textuelle Konstrukte hingegen können als Vorlagen mit wohldefinierten Lücken für beliebigen Text oder andere Verwendungen von Konstrukten aufgefasst werden. Verwendet, kombiniert und parametriert werden visuelle Konstrukte innerhalb von **Dokumenten**, die ihrerseits somit Instanzen einer DSML repräsentieren. Bei der Instanziierung resultiert stets ein sogenannter konkreter Syntaxbaum (CST) [41]. Er spiegelt die visuelle Struktur des Dokuments wider und wird im Umfeld des Compilerbaus auch als Parsebaum bezeichnet [1, 115]. Insbesondere im grafischen Kontext ist der Begriff „Baum" allerdings nicht ganz korrekt, da neben hierarchischen oft auch andere Beziehungen (z.B. Anhaften) zum Einsatz kommen. Daher ist dort die allgemeinere Bezeichnung konkreter Syntaxgraph (CSG) passender.

Je ein Beispiel für die grafische und die textuelle Welt zeigt Abbildung 2-1. Die linke Seite stellt eine konkrete grafische Syntax inklusive einem Beispielmodell dar. Sie umfasst zwei Konstrukte, nämlich einen grünen Pfeil sowie ein grünes Rechteck, das sowohl über ein statisches Männchen in der linken oberen Ecke als auch über einen textuellen Platzhalter verfügt. Die konkrete textuelle Syntax enthält dagegen nur ein Konstrukt, welches aus dem festen Schlüsselwort „state" und einem darauffolgenden textuellen Platzhalter besteht, dessen Inhalt den betreffenden Zustand (`state`) mit einem Namen versieht.

Die **abstrakte Syntax** ist eine Datenstruktur, die die semantisch relevanten Konzepte einer Sprache sowie deren Zusammenhänge beschreibt [139]. Dadurch gibt sie strukturelle Regeln vor, an die sich die etwaige Instanzen halten müssen. Im Gegensatz zur konkreten Syntax ist sie gänzlich unbelastet von jedweden visuellen Details (z.B. Farbgebung oder Interpunktionssymbole). Instanzen einer abstrakten Syntax werden üblicherweise als abstrakter Syntaxbaum (AST) [115] im textuellen oder **abstrakter Syntaxgraph (ASG)** [118] im grafischen Kontext bezeichnet. Nachdem Graph jedoch der allgemeinere Begriff ist, wird dieser im Folgenden einheitlich benutzt.

Anstelle von „abstrakte Syntax" wird häufig auch der Terminus „**Domänen-Meta-Modell**" benutzt, weil dieses Modell ein die Domäne beschreibendes Meta-Modell darstellt. In logischer Konsequenz werden Instanzen davon auch als **Domänenmodell** bezeichnet (als Alternative zu ASG).

Damit der Rechner weiß, welches Konstrukt der konkreten Syntax zu welchem Konzept der abstrakten Syntax gehört, ist es notwendig, die Inhalte beider Komponenten aufeinander abzubilden. Der dafür

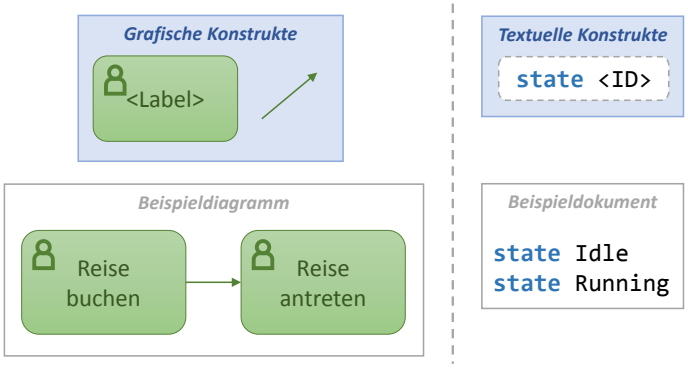

Abbildung 2-1: Beispiele für konkrete grafische und konkrete textuelle Syntax

eingesetzte Mechanismus wird als **Mapping** bezeichnet. Ohne derartige Mappings ist es nicht möglich, im Hintergrund (simultan und automatisch) ein Domänenmodell aufzubauen, während der Benutzer innerhalb eines Dokuments visuell mit Konstrukten der konkreten Syntax arbeitet.

Als zentraler Aspekt der vorliegenden Dissertation gilt das beispielgetriebene Entwickeln von DSMLs. Um dies zu ermöglichen, müssen zur Laufzeit neue Sprachen erstellt oder bestehende erweitert und angepasst werden können. Das wiederum soll weitgehend mit denselben Werkzeugen umsetzbar sein, wie auch das typische Modellieren und somit das Verwenden von DSMLs. Hierfür ist es erforderlich, dass zwei Modellierungsmodi unterschieden werden.

Unter **stringenter Modellierung** wird die traditionelle Vorgehensweise verstanden, bei der eine Sprache vollständig spezifiziert vorliegt. Es müssen sämtliche durch die Sprache vorgegebenen Rahmenbedingungen eingehalten werden. Folglich dürfen auch nur solche visuellen Konstrukte zum Einsatz kommen, die bereits vollständig definiert sind (d.h., die über ein Mapping mit einem Konzept der abstrakten Syntax verbunden sind). Ein Modifizieren der DSML ist in diesem Modus nicht möglich.

Demgegenüber steht die **freie Modellierung**. Sie gestattet ein weitgehend beliebiges Erstellen, Erweitern und Adaptieren einer DSML unter Bereitstellung von Beispielszenarien. Derartige Szenarien werden ebenfalls innerhalb eines Dokuments visuell modelliert, wobei in einem späteren Schritt die diversen Sprachartefakte (semi-)automatisch abgeleitet und in der betreffenden DSML konsolidiert werden. Selbst wenn bereits eine Sprache zugrunde liegt, brauchen die vorgegebenen Rahmenbedingungen nicht oder nur in wenigen Ausnahmefällen berücksichtigt werden. Dabei auftretende Konfliktsituationen führen beim nachgeschalteten Ableitungsprozess zu einer automatisierten Anpassung der DSML. Ein typisches Beispiel ist das nachträgliche Entfernen von nicht länger benötigten Konstrukten und der damit einhergehenden Fragestellung, wie mit bereits existierenden Instanzen verfahren werden soll.

2.1.2 Formale Grammatiken

Formale Grammatiken dienen unter anderem dazu, formale Sprachen (z.B. DSMLs) zu beschreiben [1]. Sie stellen im klassischen textuellen Fall Wortersetzungsregeln (oft auch Produktionsregeln genannt) bereit, die auf einen Text angewendet werden um einerseits zu prüfen, ob er dieser Sprache genügt. Andererseits werden die Regeln zum Parsen des Texts eingesetzt, was im Erfolgsfall dazu führt, dass ein CST entsteht. Er ist allerdings noch mit visuellen, für die betreffende konkrete Syntax spezifischen Artefakten verunreinigt. Diese Artefakte sind für eine Weiterverarbeitung allerdings belanglos, weshalb der CST im nächsten Schritt üblicherweise in einen AST als notationsunabhängige Datenstruktur überführt wird. Das geschieht in der Regel unter Zuhilfenahme einer Baumgrammatik, die eine Baumdarstellung in eine andere überführt [115]. In einem abschließenden Schritt wird aus dem AST durch Auflösung von Referenzen ein ASG, der auch über Querverbindungen verfügen kann. Zur Veranschaulichung ist dieser Prozess in Abbildung 2-2 skizziert.

Die ersten DSMLs waren allesamt **textueller** Natur und entstanden bereits Ende der 1950er Jahre [92]. Spezifiziert wurden sie mithilfe **kontextfreier Grammatiken** [17], einer speziellen Art formaler Grammatiken. Zur selben Zeit veröffentlichte auch Backus seine Backus-Naur-Form (BNF) [12], die einen weit

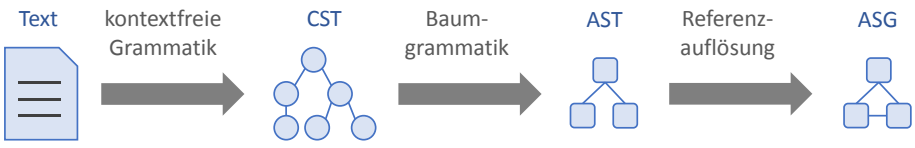

Abbildung 2-2: Analyse von Text mithilfe formaler Grammatiken

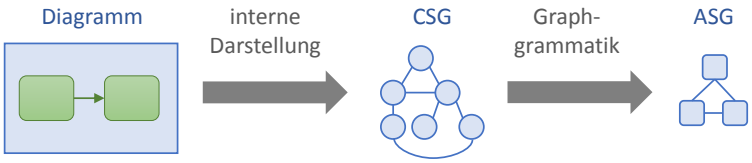

Abbildung 2-3: Analyse eines Diagramms mithilfe einer Graphgrammatik

verbreiteten Formalismus zur Spezifikation solcher Grammatiken darstellt. Im Übrigen handelt es sich hierbei ebenfalls um eine DSML, deren Anwendungsdomäne das besagte Spezifizieren ist.

Im Verlauf der 1980er Jahre wurde der grammatikbasierte Ansatz dahingehend adaptiert und erweitert, dass er auch auf **grafische Sprachen** in Verbindung mit der Bearbeitung von Diagrammen angewendet werden kann [54]. Den Ausgangspunkt bildet die Tatsache, dass Diagramme sich intern immer als attributierter Graph [36] darstellen lassen [118]. Jedes Diagramm liegt demnach schon als CSG vor [99] und nicht als reine Baumstruktur (CST). Mithilfe einer **Graphgrammatik** und den darin definierten Graphersetzungsregeln kann ein solcher CSG schließlich in einen ASG überführt werden (siehe Abbildung 2-3). Die Überprüfung, ob ein gegebenes Diagramm einer bestimmten grafischen Sprache genügt, findet somit erst bei der Transformation von CSG in ASG statt.

Gemeinhin übernehmen **Grammatiken** also die Aufgabe, Texte oder Diagramme in einen ASG zu transformieren. Mithin bilden sie die **Mapping-Komponente einer DSML**. Im Grafischen funktioniert dies direkt unter Verwendung einer Graphgrammatik, wohingegen bei Texten typischerweise zwei Grammatiken mit darauffolgender Referenzauflösung eingesetzt werden.

Graphartige, attributierte Datenstrukturen wie der ASG werden in objektorientierten Programmiersprachen stets durch ein Objektmodell abgebildet. Im engeren Sinne handelt es sich dabei um ein Meta-Modell, das die abstrakte Struktur einer bestimmten Sprache beschreibt. Dokumente, die sich dieser Sprache bedienen, repräsentieren somit immer eine Instanz des der Sprache zugrundeliegenden Meta-Modells. Folglich kann gesagt werden, dass auch bei der Verwendung formaler Grammatiken **Meta-Modelle** zum Einsatz kommen, nämlich **zur Definition abstrakter Syntaxen**. Die Entwickler und Autoren von DiaGen [97] verfolgten denselben Gedanken, als sie ihr Werkzeug auf Meta-Modelle zur Darstellung der abstrakten Syntax umstellten [98]. Das Resultat trägt den Namen DiaMeta.

Ein **Defizit** bei der Verwendung von formalen Grammatiken zur Spezifikation von DSMLs ist, dass **verschiedene Techniken zur Umsetzung der einzelnen DSML-Komponenten** genutzt werden. Dies erhöht zwangsläufig die Komplexität, weil jede Technik separat zu implementieren ist. Abhilfe schafft die ausschließliche Fokussierung auf Meta-Modellierung (Abschnitt 2.1.3), weil damit alle Komponenten durch (Meta-)Modelle ausgedrückt werden können [83].

2.1.3 Meta-Modellierung

Bei der Meta-Modellierung stehen die Begriffe **Modell und Meta-Modell im Mittelpunkt**. Gemäß der Definition von Seidewitz [126] handelt es sich bei einem Meta-Modell um ein Modell, das Informationen über eine Klasse von Modellen bereitstellt. Dabei beschreibt es die verwendbaren Elemente innerhalb dieser Modellklasse inklusive aller möglichen Eigenschaften und Beziehungen zwischen den Elementen.

Ein Meta-Modell kann demnach als abstrakte Grammatik für eine bestimmte Klasse von Modellen angesehen werden, was auch die Auffassung von Ober und Prinz [107] widerspiegelt. Abstrakt ist sie deshalb, weil keine konkrete visuelle Notation vorhanden ist. Sie muss – falls erforderlich – in einem

zusätzlichen Schritt spezifiziert und an das Meta-Modell angebunden werden. Doch auch diese Anbindung kann mithilfe weiterer Meta-Modelle erfolgen, was schließlich dazu führt, dass weitgehend **alle Komponenten einer DSML als Meta-Modelle formalisiert** und ausgedrückt werden [24]. Die konkreten Modelle der DSML sind dann Instanzen dieser Meta-Modelle. Dadurch ist jedes beteiligte, grobgranulare Artefakt ein Modell, was eine einheitliche Verarbeitung dieser Artefakte ermöglicht.

Dieser ganzheitliche, auf Modellen basierende Ansatz hat sich insbesondere bei grafischen Sprachen etabliert. Deutlich wird das durch einen Blick auf populäre Frameworks (z.B. Eclipse GMP [55] und Microsoft MSDK [101]) und Systeme (z.B. MetaCase MetaEdit+ [96]) zur Spezifikation grafischer DSMLs, die alle auf dem Prinzip der Meta-Modellierung aufsetzen. Mit sogenannten projizierenden Editoren wie MPS von JetBrains [94] und Domain Workbench von Intentional Software [128] trifft das auch vermehrt für textuelle DSMLs zu. Für eine Erklärung, was genau unter einem projizierenden Editor zu verstehen ist, sei hier auf Kapitel 7 verwiesen. Es ist also ein **Trend** zu beobachten, der weg von Grammatiken und stattdessen **hin zu (Meta-)Modellen** geht.

Nichtsdestotrotz wird im textuellen Umfeld der auf Grammatiken beruhende Ansatz weiterhin verbreitet bleiben. Primär liegt das an der großen Popularität von Frameworks wie Eclipse Xtext [153] und Spoofax [74], die mit einem großen Repertoire an Features aufwarten.

2.1.3.1 Meta Object Facility

Als Grundlage der Meta-Modellierung gilt heutzutage der von der Object Management Group (OMG) spezifizierte Standard Meta Object Facility (MOF) [110]. Er dient der standardisierten Verwaltung von Metadaten innerhalb einer **mehrschichtigen Metadaten-Architektur**. Die einzelnen Schichten werden in Form von implizit zueinander in Beziehung stehenden Modellen wiedergegeben. Die impliziten Beziehungen zwischen zwei Modellen entstehen durch explizite `instanceOf`-Referenzen zwischen Elementen beider Modelle. In der Literatur wird ein Modell oftmals als Ebene bezeichnet, da dieser Begriff das Konzept einer mehrschichtigen Architektur am besten beschreibt. Im Kontext der Meta-Modellierung findet man daher häufig eine geschichtete Darstellung von Modellen, bei denen Elemente einer niedrigen Ebene *A* Instanzen von Elementen (auch als Typen bezeichnet) der nächsthöheren Ebene *B* sind. Mit anderen Worten ausgedrückt repräsentiert der Inhalt von Ebene *B* das Meta-Modell zum einem auf Ebene *A* beschriebenen Modell.

Jedes Meta-Modell ist gleichsam ein Modell, das durch ein anderes Meta-Modell beschrieben wird. In Bezug auf das ursprüngliche Modell handelt es sich bei letzterem also um das sogenannte Meta-Meta-Modell. Prinzipiell lässt sich dieser Prozess beliebig oft fortsetzen, wodurch eine unendliche Abfolge

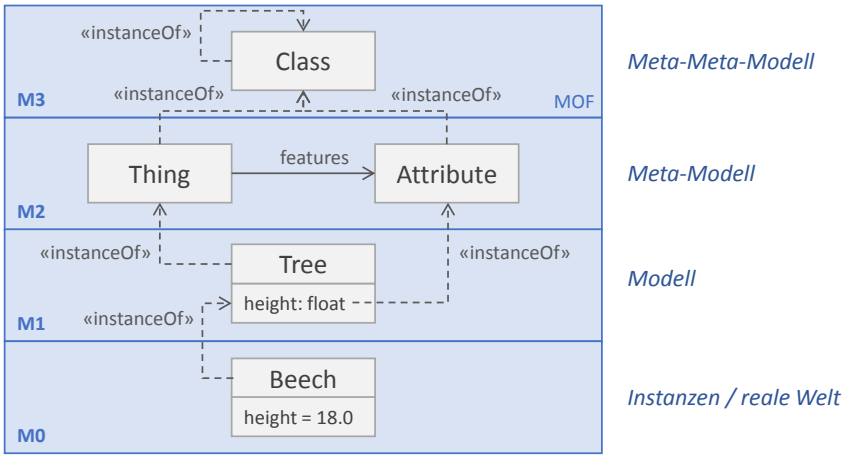

Abbildung 2-4: Beispiel einer Meta-Modell Hierarchie in Anlehnung an MOF

von Meta-Ebenen entsteht. Für die oberste Ebene (auch MOF-Ebene genannt) gilt deshalb die Konvention der Selbstbeschreibung, d.h. ihre Elemente sind gleichzeitig Instanzen von sich selbst. Zusätzlich muss eine Reihe weiterer Restriktionen erfüllt sein, die sowohl [110] als auch [8] zu entnehmen sind. Beispielsweise muss jedes Element einer Ebene *n* eine Instanz eines Elements der Ebene *n + 1* sein. Referenzen in umgekehrter Richtung sind nicht zulässig. Die Benennung bzw. Nummerierung der einzelnen Ebenen geschieht absteigend, wobei die MOF-Ebene als `M3` bezeichnet wird. `M0` als unterste Ebene umfasst letztendlich Objekte, die eine Entsprechung in der (virtuellen) Realität aufweisen.

Ein stark vereinfachtes Beispielszenario einer Meta-Ebenen Hierarchie veranschaulicht Abbildung 2-4. Vereinfacht bedeutet es, dass `M3` auf ein Element namens `Class` reduziert wurde. Zur Erfüllung der MOF-Spezifikation müssten darin jedoch alle Elemente enthalten sein, die die minimale Variante des Standards vorschreibt. Die Möglichkeit, dass die drei unteren Ebenen frei spezifizierbar sind, wurde in Abbildung 2-4 voll ausgeschöpft.

2.1.3.2 Qualitätskriterien für Meta-Modelle

Meta-Modelle müssen gewissen **Qualitätsansprüchen genügen**. Andernfalls ist deren Verständlichkeit beeinträchtigt, was eine spätere Verarbeitung und Evolution ungemein erschwert [13, 89]. Dies gilt ausdrücklich auch in Fällen, in denen Meta-Modelle (semi-)automatisch anhand von Beispielmodellen generiert werden, denn die erzeugten Artefakte müssen im Nachhinein stets von einem Modellierungsexperten auf Validität und Konformität hinsichtlich der abzubildenden Domäne überprüft werden. Ist dabei das Meta-Modell adäquat strukturiert (siehe nächster Absatz), sind Anpassungen durch den Experten nur in geringem Umfang notwendig. Gleichzeitig sind zu erledigende Modifikationen auf einfache Weise möglich.

Um die Qualität von Meta-Modellen bewerten zu können, haben Bertoa und Vallecillo in [13] ein Framework präsentiert, das eine Ansammlung diesbezüglicher **Qualitätskriterien** bietet. Wir beschränken uns an dieser Stelle auf solche Kriterien, die – basierend auf den durch die Beispielmodelle zur Verfügung stehenden Informationen – **weitgehend objektiv messbar** sind und denen damit automatisiert begegnet werden kann. Alle übrigen Kriterien (unter anderem Präzision im Sinne einer Weiterverarbeitung, Verständlichkeit durch andere Personen, Ästhetik, Beständigkeit und Gültigkeit über die Zeit etc.) müssen ohnehin von derjenigen Person (oder Gruppe von Personen) eingehalten werden, die für die Weiterverarbeitung der auf dem jeweiligen Meta-Modell beruhenden Modelle verantwortlich sind.

Im späteren Verlauf berücksichtigt werden die im Folgenden vorgestellten, objektiv messbaren Qualitätskriterien:

- *Vollständigkeit*: Das Meta-Modell umfasst alle Konzepte, die zur Beschreibung der betreffenden Domäne erforderlich sind.
- *Redundanz*: Jedes Konzept und Feature ist nicht mehr als einmal vorhanden, d.h. Redundanzen sind zu vermeiden. Andernfalls hätte dies negative Auswirkungen auf die nachträgliche Änderbarkeit, da z.B. beim geänderten Verständnis eines bestimmten Sachverhalts mehrere Artefakte anzupassen wären, die dasselbe semantische Objekt beschreiben.
- *Relevanz*: Es sind lediglich solche Artefakte im Meta-Modell enthalten, die für die jeweiligen Anwendungsszenarien benötigt werden.
- *Einfachheit*: Unnötige Verkomplizierungen der Struktur sind zu vermeiden (z.B. Verzicht auf tiefe und komplexe Generalisierungs-/Spezialisierungshierarchien, sofern nicht zwingend notwendig).

- *Prägnanz* (treffender ist der englische Begriff „Conciseness", für den im Deutschen keine adäquate Übersetzung existiert): Ergibt sich im Wesentlichen unter Berücksichtigung der Kriterien *Redundanz*, *Relevanz* und *Einfachheit*. Zur Steigerung der Prägnanz eines Meta-Modells wird häufig auf Sprach- und Entwurfsmuster zurückgegriffen [22, 120, 124]. Konkrete Lösungen, wann Sprachmuster auf vorhandene Meta-Modelle angewendet werden können, werden in Abschnitt 4.4 erörtert.
- *Selbstbeschreibung*: Die einzelnen Artefakte des Meta-Modells weisen aussagekräftige Namen auf, wodurch die Verständlichkeit für Dritte gesteigert wird.
- *Natürlichkeit*: Eng verwandt mit der *Selbstbeschreibung*. Zusätzlich wird gefordert, dass verwendete Bezeichnungen auf ein (möglichst allgemeingültiges) Glossar zurückzuführen sind.

Die letzten zwei Kriterien sind zwar nicht objektiv messbar, doch ist es zweckmäßig, den Benutzer bei der Benennung von Meta-Modellartefakten eine passable Werkzeugunterstützung zu bieten, die bereits während der freien Modellierung greift. Insbesondere von Bedeutung ist dies bei der Entwicklung von DSMLs im ontologischen Kontext, wie in Abschnitt 5.3.1.4 ersichtlich wird. Deshalb und weil beide Kriterien essentiell für die Verständlichkeit eines Meta-Modells sind, wurden sie ebenfalls aufgeführt.

Die aufgelisteten Qualitätskriterien sind vor allem für das Meta-Modell von Bedeutung, welches die **abstrakte Syntax** definiert. Denn dieses Meta-Modell bildet die Grundlage für die Weiterverarbeitung der späteren Domänenmodelle. Meta-Modellartefakte der konkreten Syntax sowie der Mapping-Komponente sind aufgrund ihres rein visuellen Charakters nicht oder höchstens in seltenen Ausnahmen relevant.

2.2 Linguistisches Meta-Modell

Das linguistische Meta-Modell (LMM) und der damit verknüpfte Meta-Modellierungsansatz kann als Erweiterung der mehrschichtigen Metadaten-Architektur MOF (Abschnitt 2.1.3.1) angesehen werden. Es dient der Repräsentation beliebiger Ebenenhierarchien und liefert damit die Basis für das in Abschnitt 2.2.1.1 vorgestellte Modellierungsmuster „orthogonale Klassifikation". Erstmalig wurde das LMM von Volz in [142] spezifiziert und später von Jahn in [66] weiterentwickelt. Die kommenden Betrachtungen beruhen auf der ergänzten Variante von Jahn, die auch die konzeptuelle Basis der in Kapitel 8 präsentierten *Model Workbench* verkörpert.

Neben den aus MOF bekannten Modellierungsmustern (auch Sprachmuster genannt) „Generalisierung/Spezialisierung" sowie „Enumerationen" unterstützt das LMM eine Reihe erweiterter Sprachmuster, die im ersten Unterabschnitt erläutert werden. Anschließend folgt eine Vorstellung der Struktur des LMM sowie zugehöriger Visualisierungsformen, die in den späteren Kapiteln Verwendung finden.

2.2.1 Erweiterte Modellierungsmuster

Die nachfolgend präsentierten erweiterten Modellierungsmuster werden allesamt vom LMM unterstützt bzw. bilden dessen Grundlage. Allerdings wird nur auf diejenigen eingegangen, die auch im weiteren Verlauf dieser Arbeit zum Einsatz kommen.

2.2.1.1 Orthogonale Klassifikation

Das von Atkinson und Kühne entwickelte und als orthogonale Klassifikation [9] betitelte Modellierungsmuster erlaubt die Einordnung von *instanceOf*-Beziehungen in zwei orthogonal zueinander stehende Meta-Modell Stacks. Der linguistische Meta-Modell Stack beinhaltet ein Meta-Modell, das beschreibt,

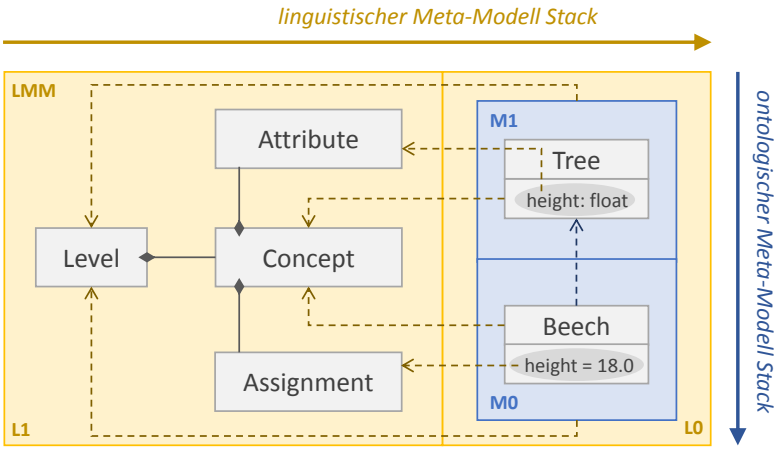

Abbildung 2-5: Orthogonale Klassifikation skizziert an einem einfachen Beispiel

wie Meta-Modellhierarchien der Anwendungsdomäne abzuspeichern sind. Ein dazu orthogonaler, ontologischer Meta-Modell Stack hingegen umfasst beliebig viele Modelle, die Beziehungen untereinander ausschließlich über ihren Inhalt ausdrücken. Alle Elemente des ontologischen Stacks sind demzufolge Instanzen von Elementen des linguistischen Meta-Modells. Gleichzeitig kann ein Element des ontologischen Stacks eine Instanz eines anderen Elements des ontologischen Stacks darstellen.

Zur Verdeutlichung und in Anlehnung an die in Abschnitt 2.2.2 beschriebene Struktur des LMM zeigt Abbildung 2-5 ein Beispiel der orthogonalen Klassifikation. Aus Übersichtlichkeitsgründen wurde auf die Benennung der als gestrichelte Pfeile visualisierten *instanceOf*-Beziehungen verzichtet. Als Besonderheit bietet das LMM (linguistische Ebene `L1`) die Möglichkeit, beliebige ontologische Ebenen (Modelle) auf `L0` zu spezifizieren. Beispiele hierfür sind die beiden Ebenen `M1` und `M0`, die wiederum linguistische Instanzen von `Level` sind.

Das in der Grafik als `Assignment` bezeichnete Element entspricht einer Wertzuweisung an ein vorher deklariertes Attribut. Analog dazu besitzt auch das `height`-Attribut mit `Attribute` einen korrespondierenden Typ auf der linguistischen Ebene `L1`. Das Konzept `Beech` auf ontologischer Ebene `M0` weist sowohl eine linguistische (ockerfarbene) als auch eine ontologische (blaue) *instanceOf*-Beziehung auf, nämlich zum einen auf `Concept` und zum anderen auf `Tree`.

Als Fazit bleibt, dass der gesamte Inhalt des ontologischen Meta-Modell Stacks Dank der orthogonalen Klassifikation frei definiert werden kann. Das LMM bietet somit die Möglichkeit, sämtliche (Meta-) Modelle im selben Format abzulegen, was eine einheitliche Verarbeitung gestattet. Alle weiteren Sprachmuster werden direkt durch das LMM (Ebene `L1`) bereitgestellt und können daher bei der Konstruktion ontologischer Ebenenhierarchien eingesetzt werden.

2.2.1.2 Clabject alias Konzept

Das Akronym „Clabject" besteht aus den beiden Begriffen „Class" und „Object" und drückt aus, dass ein Element mit dieser Nomenklatur sowohl eine Klasse als auch ein Objekt repräsentiert. Nach der Definition von Atkinson und Kühne [10] wird der Klassenteil als Typ-Facette und der Objektteil als Instanz-Facette bezeichnet. Innerhalb der Typ-Facette erfolgt die Deklaration von Attributen, wohingegen der Inhalt der Instanz-Facette aus der *instanceOf*-Relation des Clabjects zu dessen Typ resultiert. Demgemäß enthält letztere Wertzuweisungen an Attribute, die den Typ-Facetten des mit *instanceOf* referenzierten Clabjects zuzüglich aller Generalisierungen entstammen.

Der Grund für die Einführung dieses Sprachmusters fußt darauf, dass ein Element einer Ebene *n* Instanz eines Elements aus Ebene *n + 1* ist, aber gleichzeitig der Typ eines Elements aus Ebene *n – 1*. Folglich handelt es sich bei Clabjects um die einzelnen Konzepte einer Ebene, die die Clabjects einer darunterliegenden Ebene beschreiben. Anstelle des Kunstworts Clabject wird im Folgenden der neutrale Begriff „Konzept" verwendet. Demnach wird das entsprechende Konstrukt des LMM als `Concept` bezeichnet. Für eine bessere Verständlichkeit ist dennoch an einigen Stellen die Unterscheidung in Klasse und Objekt sinnvoll, weshalb manchmal zwischen *Meta-Konzept* und *Instanz-Konzept* oder einfacher zwischen *Typ* und *Instanz* differenziert wird.

2.2.1.3 Instanzspezialisierung

Instanzspezialisierung [67] ist ein Sprachmuster, das auf der prototypbasierten Vererbung beruht. Diese wiederum ist bekannt aus prototypbasierten Programmiersprachen [105] wie ECMAScript und Lua. Im Gegensatz zur klassenbasierten Vererbung werden bei der prototypbasierten Variante nicht Attribute vererbt, sondern Zuweisungen von Werten an solche Attribute. Bezogen auf Konzepte handelt es sich bei der Instanzspezialisierung um eine Beziehung zwischen den Instanz-Facetten zweier Konzepte, wodurch das spezialisierende Konzept (häufig *Verwendung* genannt) alle Wertzuweisungen des Prototypenkonzepts (oft *Definition* genannt) erbt.

Veranschaulicht werden soll dieser Sachverhalt an dem Beispiel in Abbildung 2-6. Es zeigt ein Objektdiagramm mit drei Instanzen des (nicht explizit abgebildeten) Konzepts `HSBColor`, die verschiedene Grüntöne im HSB-Farbraum repräsentieren. Dieser setzt sich aus den Attributen `hue` (Farbwert), `saturation` (Sättigung) und `brightness` (absolute Helligkeit) zusammen. Das mittig platzierte Konzept `Green` stellt ein sattes, leuchtendes Grün als Prototypen dar. Linker Hand befindet sich das Konzept `LightGreen`, welches die Instanz-Facette von `Green` spezialisiert und dabei den Sättigungsgrad reduziert und die Helligkeit auf den Maximalwert `100` setzt. Der Farbwert wird jedoch unberührt übernommen. Das rechts liegende Konzept `DarkGreen` spezialisiert ebenfalls die Instanz-Facette von `Green`, wobei lediglich die Helligkeit auf den Wert `30` verringert wird. Farbwert und Sättigung bleiben erhalten und gelten somit weiterhin auch für `DarkGreen`.

Jede Instanzspezialisierungsbeziehung trägt die Beschriftung „concreteUseOf", um sie von anderen Beziehungen wie *instanceOf* und *extends* unterscheiden zu können. Der Name rührt vom ursprünglichen Einsatzgebiet dieser Beziehung, das als Typ-Verwendungskonzept bezeichnet wird [64, 142]. Darin wird das spezialisierende Konzept als Verwendung und der Prototyp als Typ bezeichnet.

In Ergänzung zur prototypbasierten Vererbung bietet die Instanzspezialisierung zusätzliche Möglichkeiten, die es erlauben, das Überschreiben von Werten auf Spezialisierungsseite zu limitieren. Dies ist im weiteren Verlauf der Arbeit allerdings nicht von Bedeutung, weshalb hier auf eine nähere Betrachtung verzichtet wird. Details können der Dissertation von Jahn [66] entnommen werden.

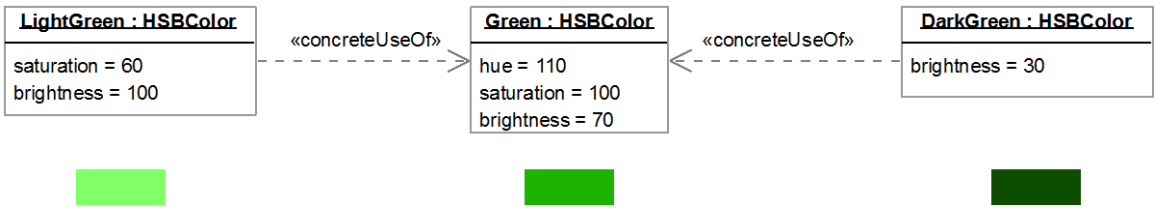

Abbildung 2-6: Beispiel für Instanzspezialisierung

2.2.1.4 *Erweiterter Powertyp*

Eingeführt wurde der erweiterte Powertyp von Jablonski et al. in [65]. Seine Grundlage bildet der von Odell in [108] vorgestellte (reguläre) **Powertyp**. Letzterer begegnet dem typischen Problem bei der Abstraktion eines bestimmten Sachverhalts in ein Modell. Hierzu kann im Rahmen der Meta-Modellierung entweder auf **Instanziierung** oder **Generalisierung** zurückgegriffen werden, was zwei verschiedenen Sichtweisen auf das Modell entspricht. Durch Einführung einer neuen, als „partitions" bezeichneten Beziehungsart ermöglicht der Powertyp die Abbildung dieser beiden Sichtweisen.

Da jeder erweiterte Powertyp gleichzeitig ein regulärer Powertyp ist, genügt ein Beispiel des Erstgenannten. Es ist in Abbildung 2-7 veranschaulicht. Der Powertyp wird darin vom Konzept `NodeKind` repräsentiert, wobei seine Attribute vorerst ignoriert werden können. Das Zielkonzept `Node` der ausgehenden `partitions`-Kante hingegen heißt partitionierter Typ. Verwendet wird ein Powertyp über die Instanziierung durch andere Konzepte. Die einzelnen Instanziierungsbeziehungen werden im Diagramm von gestrichelten Pfeilen symbolisiert. Nicht visualisiert dagegen werden die aufgrund der `partitions`-Kante implizit vorliegenden Generalisierungsbeziehungen, die zwischen den Powertyp-Instanzen und `Node` bestehen. Diese Powertyp-Instanzen sind somit gleichermaßen Instanzen von `NodeKind` und Spezialisierungen von `Node`, wodurch sie auch alle Attribute von letzterem erben.

Wegen der bestehenden Instanziierungsbeziehungen liegen `NodeKind` und `Node` logisch betrachtet eine Meta-Ebene höher als die übrigen sechs Konzepte. Dieser Umstand ist in der vorliegenden Arbeit jedoch nicht von Bedeutung, so dass auf die Einzeichnung der beteiligten Meta-Ebenen verzichtet wurde.

Die Attribute des Konzepts `NodeKind` sind zwar für einen regulären Powertyp durchaus legitim, haben jedoch keinerlei Auswirkung auf das Verhalten seiner Instanzen. Anders verhält es sich mit dem **erweiterten Powertyp**. Dort können boolesche Attribute des Powertyps dazu genutzt werden zu **steuern, welche Attribute vom partitionierten Typ geerbt werden** sollen. Die Verknüpfung, welches Attribut des Powertyps zu welchem Attribut des partitionierten Typs gehört, wurde im Beispiel über den Namen hergestellt. So wird mithilfe von `supportsTitle` angegeben, ob das `title`-Attribut geerbt wird oder nicht. Demzufolge verfügen ausschließlich `ElectronicTask`, `PaperTask` und `DetachedTask` über einen Titel. `Exit` erbt sogar kein einziges Attribut von `Node`.

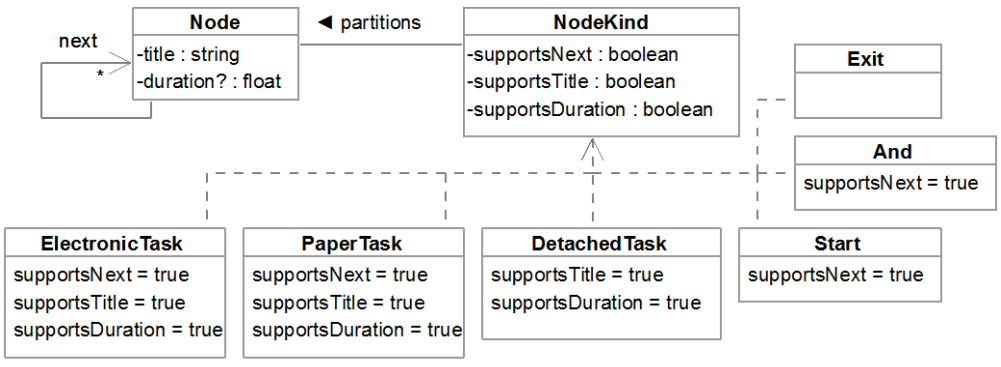

Abbildung 2-7: Beispiel für erweiterten Powertyp

2.2.2 Aufbau

Der Aufbau des LMM wird hier nur insoweit beschrieben, wie er in den weiteren Ausführungen von Belang ist. Auch die schematische Grafik in Abbildung 2-8 beschränkt sich auf die relevanten Komponenten.

Die Basis eines jeden (Meta-)Modells bildet eine **Ebene**, die ihrerseits andere Ebenen referenzieren kann. Bei der *instanceOf*-Referenz handelt es sich um die klassische, von MOF bekannte Beziehung, die ausdrückt, dass die Ausgangsebene das Modell und die Zielebene das dazugehörige Meta-Modell darstellt. Darüber hinaus können Ebenen andere Ebenen mittels *alignedWith* referenzieren. Diese Beziehung besagt, dass die partizipierenden Ebenen auf einer Stufe angesiedelt und gleichzeitig alle Konzepte der Zielebene innerhalb der Ausgangsebene verfügbar sind. Diese Beziehungsart ist vergleichbar mit dem Importmechanismus bei Programmiersprachen wie Java oder C#.

Jede Ebene enthält genau ein **Paket**. Pakete können beliebig viele weitere Pakete umfassen, so dass mit ihrer Hilfe Ebenen hierarchisch strukturiert werden können. Daneben besitzen sie keine weitere Funktion.

Innerhalb von Paketen liegen außerdem die wichtigsten Komponenten des LMM, und zwar die Konzepte. Jedes **Konzept** darf potentiell beliebige andere Konzepte referenzieren. Welche Restriktionen dabei gelten ist der Dissertation von Jahn zu entnehmen [66]. Die beiden Beziehungsarten *instanceOf* und *extends* entsprechen in ihrer Semantik denjenigen bei MOF. Erstere besagt, dass das Ausgangskonzept die Instanz des Zielkonzepts ist. Dieses stellt somit den zugehörigen Typ dar. Hinter *extends* hingegen verbirgt sich die typische, aus der traditionellen objektorientierten Programmierung geläufige Spezialisierung von Klassen. Der Zweck sowie die Semantik von *concreteUseOf* und *partitions* wurde bereits in den Abschnitten 2.2.1.3 bzw. 2.2.1.4 erläutert.

Neben diversen Referenzen besitzen Konzepte üblicherweise **Attribute** oder **Zuweisungen** (unter Umständen auch beide Komponenten). Die Attribute zählen hierbei zur Typ-Facette eines Konzepts, während Zuweisungen der Instanz-Facette angehören. Bei der stringenten Modellierung bezieht sich jede Zuweisung auf genau ein Attribut (für freie Modellierung siehe den Anfang von Kapitel 4). Es muss dafür explizit oder implizit (durch Konzeptreferenzen eingebracht) der Typ-Facette desjenigen Konzepts entstammen, das als Typ des Eigentümerkonzepts der Zuweisung gilt. Die genauen Regeln für das implizite Einbringen von Attributen in die Typ-Facette eines Konzepts können unter dem Stichwort „virtuelle Attribute" ebenfalls in [66] nachgeschlagen werden.

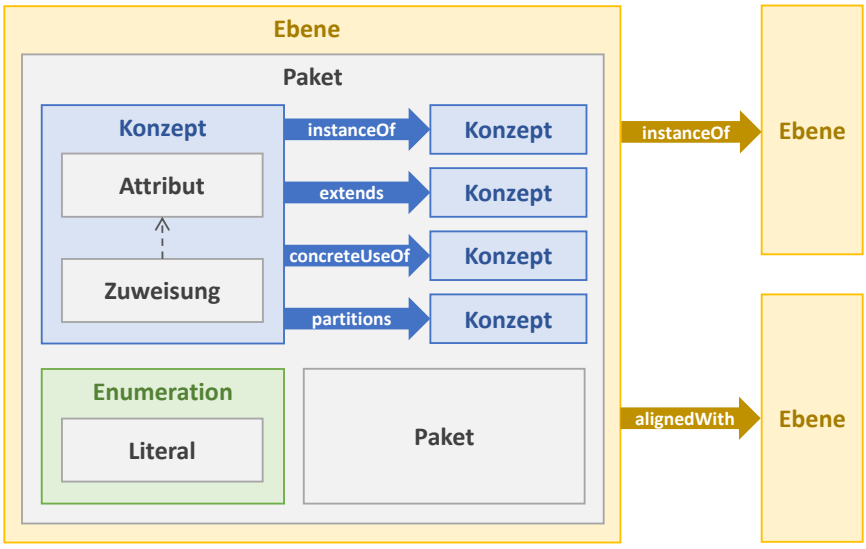

Abbildung 2-8: Struktur des linguistischen Meta-Modells

Des Weiteren dürfen innerhalb von Paketen auch Enumerationen liegen. Sie repräsentieren den aus vielen Programmiersprachen bekannten Aufzählungstyp. Jede **Enumeration** setzt sich aus einer beliebigen Anzahl von Enumerationsliteralen zusammen, die in Zuweisungen als Werte verwendet werden können.

Dem vorgestellten Ansatz liegt die Annahme zugrunde, dass alle zusammengehörenden, mithilfe des LMM formulierten (Meta-)Modelle in einem gemeinsamen **Modellrepositorium** abgelegt sind. Die Abspeicherungsstruktur ist dabei ein **gerichteter Graph**, so dass sämtliche im Repositorium enthaltenen Modelle bequem traversiert und angefragt werden können. Beispielsweise können dadurch alle Instanzen eines gegebenen Meta-Konzepts auf einfache Weise ermittelt werden.

2.2.3 Notation

Das LMM bildet die abstrakte Syntax der **linguistischen Metasprache (LML)**. Als Sprache verfügt die LML zusätzlich über eine konkrete Syntax, die laut der Spezifikation von Volz in [142] textueller Natur ist. Im Rahmen dieser Arbeit wurde sie etwas vereinfacht, indem das syntaktische Rauschen reduziert wurde. Dies drückt sich dadurch aus, dass die geschweiften Klammern zur Festlegung von Blockstrukturen wegfallen und stattdessen auf semantisches Einrücken gesetzt wird.

Die einzelnen Beispiellistings der vorliegenden Dissertation zeigen ausschließlich Konzepte sowie deren Instanz-Facette, weshalb hier lediglich auf deren Notation kurz eingegangen wird. Listing 2-1 umfasst die beiden Instanz-Konzepte `Booking` und `Travel`. Jedes Instanz-Konzept verweist an erster Stelle auf ein Meta-Konzept, das typischerweise in einer mit *instanceOf* referenzierten Meta-Ebene deklariert ist. Die Aufführung eines anderen Konzepts an erster Stelle entspricht der *instanceOf*-Beziehung zwischen Konzepten (d.h. `Booking` und `Travel` sind Instanzen von `Task`). Anschließend folgt der Name des Konzepts, über den es von anderen Konzepten referenziert werden kann. Ferner darf jedes Konzept Zuweisungen beinhalten, die eingerückt und unterhalb des betreffenden Konzepts liegen müssen. Das Instanz-Konzept `Booking` umfasst zwei derartige Zuweisungen, wobei dem zugrundeliegenden Attribut `function` eine Zeichenkette und `successors` mit `Travel` ein anderes Konzept zugewiesen wird.

```
1  Task Booking
2    function = "Reise buchen"
3    successors = Travel
4
5  Task Travel
6    function = "Reise antreten"
```

Listing 2-1: Beispiel eines Modells in LML-Notation

Insbesondere bei Modellen, deren Konzepte zahlreiche Querverbindungen aufweisen, ist die Struktur schwer auf Anhieb zu erfassen. Solche Beziehungen sind nicht als Verbindungslinien visualisiert, sondern durch die bloße Nennung des referenzierten Konzepts (siehe dazu Zeile 3 in Listing 2-1, in der auf das Konzept `Travel` verwiesen wird). Deshalb wird ein Großteil der im Folgenden abgebildeten LMM Modellfragmente mithilfe der UML-Notation für Klassen- und Objektdiagrammen visualisiert.

Wie diese Notation für das obige Beispiel aussieht, ist in Abbildung 2-9 illustriert. Links befindet sich die Deklaration des Meta-Konzepts `Task`, welches in einem Paket `Package` und dieses wiederum in einer Ebene `MetaLevel` liegt. Neben dem String-Attribut `function` besitzt es ein referentielles Attribut `successors`, dessen Typ selbst wieder `Task` ist. Auf der rechten Seite steht ein zugehöriges Beispielmodell, das mit den Instanz-Konzepten `Booking` und `Travel` den gleichen Inhalt aufweist wie Listing 2-1. Die *instanceOf*-Beziehungen der Konzepte sind allerdings nicht in Form von

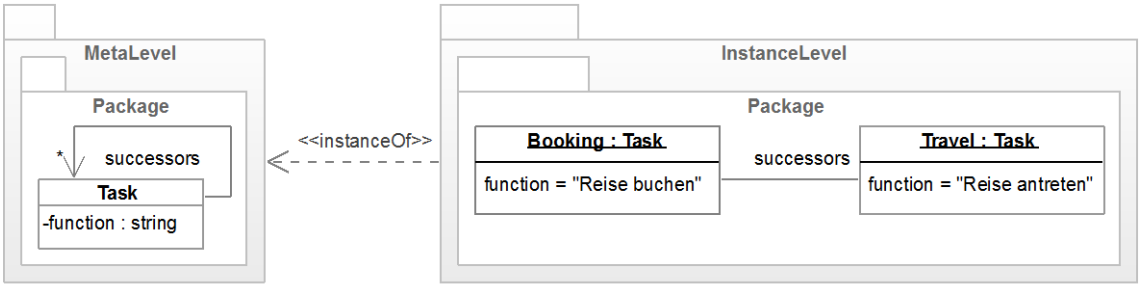

Abbildung 2-9: Beispiel für Meta-Modell und Modell als UML Klassen- bzw. Objektdiagramm

Verbindungslinien dargestellt, sondern – wie für Objektdiagramme üblich – ebenfalls durch bloße Nennung des Meta-Konzeptnamens. Die `successors`-Referenz hingegen wird von einer eigenständigen Linie mit entsprechender Beschriftung repräsentiert.

Ein weiterer wichtiger Aspekt, der im Zusammenhang mit der Notation der LML steht, ist die **Unterscheidung in linguistische und ontologische Modellierung**. Die Begriffe „linguistisch" und „ontologisch" beziehen sich hierbei nicht auf den jeweils gleichnamigen Meta-Modell Stack, sondern auf die verwendete konkrete Syntax. Beim linguistischen Modellieren wird die generische konkrete Syntax der LML eingesetzt, wohingegen beim ontologischen Modellieren eine eigene Notation festzulegen ist, die spezifisch für die betreffende Domäne ist. Wie schon in Abschnitt 1.2.2 erwähnt, wird in beiden Fällen allerdings stets entlang des ontologischen Meta-Modell Stacks modelliert.

In den nachfolgenden Ausführungen werden **Konzepte meistens direkt innerhalb der Ebenen** platziert. Dies bricht zwar mit der Spezifikation des LMM, doch vereinfacht es die Darstellung und damit die Lesbarkeit.

2.3 Notation der Ablaufdiagramme

In den späteren Kapiteln werden wiederholt **Ablaufdiagramme** dazu eingesetzt, einen Überblick über die jeweils vorkommenden Abläufe zu geben. Ihre Notation orientiert sich dabei stark an der von Aktivitätsdiagrammen der UML. Aufgrund ihres **informellen Charakters** erheben sie jedoch keineswegs Anspruch auf Vollständigkeit und Korrektheit im Sinne der UML, sondern dienen lediglich Illustrationszwecken. Sämtliche in den einzelnen Ablaufdiagrammen verwendeten Konstrukte sind in Abbildung 2-10 aufgeführt und werden nachfolgend kurz beschrieben.

- *Startpunkt*: An der Stelle startet der abgebildete Algorithmus.
- *Endpunkt*: An der Stelle terminiert der abgebildete Algorithmus, sobald alle durch Schleifen hervorgerufenen Iterationen dort angelangt sind.
- *Datenobjekte(e)*: Hierbei handelt es sich um die eigentlich Daten, die der Algorithmus konsumiert und/oder produziert. In manchen Fällen wurde aus Übersichtlichkeitsgründen bewusst davon abgesehen, sämtliche Datenobjekte ins Diagramm mit aufzunehmen.
- *Aktion*: Eine für den Algorithmus atomare Operation, die er ausführt.

Abbildung 2-10: Notation der Ablaufdiagramme

- *Entscheidung*: Dient zur Aufspaltung des Ablaufs basierend auf einer Entscheidung. Das spätere Zusammenfließen kann in jedem beliebigen anderen Konstrukt erfolgen. Dort setzt der Algorithmus fort, sobald ein zuvor aufgespaltener Ausführungspfad eintrifft.
- *Schleife*: Bezieht sich immer auf eine Menge von Datenobjekte, die in sie hineinfließen und über die letztlich iteriert wird. Um welche Datenobjekte es sich handelt, steht informell im Kopf des Schleifenkonstrukts (z.B. „Für alle Elemente:").

3 Verwandte Arbeiten

Dieses Kapitel widmet sich diversen Arbeiten, die auf unterschiedliche Weisen mit dieser Dissertation verwandt sind und im Folgenden kategorisiert vorgestellt werden. Eine zentrale Aufgabe beim Entwerfen von DSMLs im ontologischen Kontext ist die Definition einer konkreten Syntax. Für die Umsetzung der beispielgetriebenen DSML-Entwicklung ist es daher von Vorteil, wenn diese Definition ebenfalls anhand von Beispielen erfolgen kann. Welche Herangehensweisen und Lösungen es dafür schon gibt, steht im Mittelpunkt des Abschnitts 3.1.

Der Abschnitt 3.2 befasst sich mit der Ableitung einer abstrakten Syntax von einer gegebenen konkreten Syntax. Ein derartiges Verfahren kommt in Kapitel 5 zum Einsatz.

In Abschnitt 3.3 geht es um die Vorreiter der beispielgetriebenen Meta-Modellierung und damit auch der beispielgetriebenen Entwicklung von Modellierungssprachen. Hierbei handelt es sich um die Rekonstruktion von Meta-Modellen aus einer möglichst großen Anzahl an positiven und auch negativen Beispielmodellen. Die dort analysierten Ansätze haben ihren Ursprung in der Rekonstruktion von Grammatiken, weshalb auch darauf eingegangen wird.

Der Fokus von Abschnitt 3.4 liegt auf Arbeiten, die im weitesten Sinne die beispielgetriebene Entwicklung von DSMLs verfolgen. Meist ist das vorrangige Ziel allerdings der Erhalt eines Meta-Modells, das die abstrakte Syntax beschreibt. Deswegen ist dieser Abschnitt mit „Beispielgetriebene (Meta-)Modellierung" betitelt. Bei einigen Ansätzen wird dennoch auch die konkrete Syntax berücksichtigt, die dann jedoch stets grafischer Natur ist.

Den Abschluss macht ein Fazit, bei dem insbesondere die Systeme des vierten Abschnitts verglichen werden. Darüber hinaus wird ein zusammenfassendes Resümee gegeben, bei dem auch auf die nutzbringenden Aspekte der übrigen verwandten Arbeiten eingegangen wird.

3.1 Entwurf konkreter Syntaxen nach dem WYSIWYG-Prinzip

Die gesamte Interaktion zwischen Benutzer und DSML findet über die konkrete Syntax statt. Sie bildet demnach die Schnittstelle zur Außenwelt und sollte deshalb verständlich und intuitiv verwendbar sein. Dem zuträglich ist, wenn bereits während der Entwurfsphase einer DSML die einzelnen Konstrukte visualisiert werden können. Die Anzeige einer externen Vorschau wäre die einfachste Variante. Noch besser geeignet ist die Verfolgung des in der Informatik häufig eingesetzten **WYSIWYG**-Prinzips (Akronym für „What You See Is What You Get") [152], bei dem das erwartete Resultat direkt visuell bearbeitet werden kann. Ein bekanntes Beispiel für WYSIWYG sind HTML-Editoren, wie Adobe Dreamweaver oder TinyMCE. Ihnen gegenüber stehen einfache Texteditoren, die die Eingabe des vollständigen HTML-Quellcodes erfordern. Eine Vorschau der erstellten HTML-Seite kann bei reinen Texteditoren nur durch Öffnen mit einem Web-Browser erfolgen.

Als Pionier beim Entwurf grafischer Konstrukte nach dem WYSIWYG-Prinzip gilt das von MetaCASE kommerziell vertriebene **MetaEdit+** [96, 136]. Es handelt sich um eine Plattform zur Entwicklung und Verwendung grafischer DSMLs gemäß der weit verbreiteten Top-Down Vorgehensweise. Die Entwicklung von DSMLs ist dabei klar von deren Verwendung getrennt, indem zwei unterschiedliche Teilsysteme zum Einsatz kommen. Das zur Erstellung von Modellierungssprachen dienende System heißt

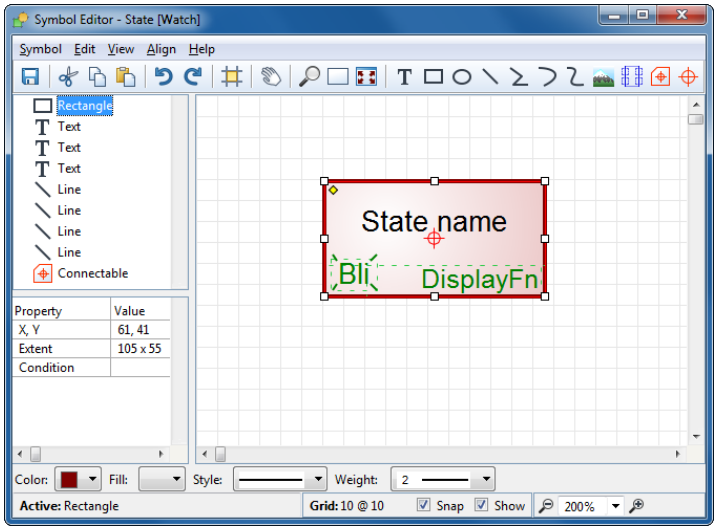

Abbildung 3-1: Symbol Editor der MetaEdit+ Workbench [149]

MetaEdit+ Workbench und stellt einen eigenen Symbol Editor bereit, mit dessen Hilfe Konstrukte für die konkrete Syntax gebaut werden können (Abbildung 3-1). Der typische Workflow gestaltet sich aber so, dass zunächst die abstrakte Syntax definiert wird (bei MetaEdit+ als konzeptuelles Modell bezeichnet) und danach erst die zugehörigen grafischen Konstrukte. Bezogen auf das Beispiel aus Abschnitt 2.2.3 bedeutet das, dass erst das Konzept `Task` deklariert werden muss, bevor eine entsprechende, visuelle Repräsentation (z.B. ein Rechteck mit abgerundeten Ecken) angegeben werden kann. Eine DSML muss daher vollständig „vorgedacht" werden, so dass sie beim Modellieren eingesetzt werden kann.

MetaCASE hat allerdings erkannt, dass eine gute **Werkzeugunterstützung zur Definition grafischer Konstrukte** für eine **positive Nutzererfahrung** wichtig ist. Beispielhaft zu nennen sind die mit Version 5.0 eingeführten Templates [75]. Sie ermöglichen ein komfortables Anlegen dynamischer Regionen innerhalb eines Konstrukts, an denen z.B. andere Konstrukte andocken können. Mit ihrer Hilfe lassen sich also die in einigen Sprachen vorkommenden Ports (z.B. in UML Komponentendiagrammen) bequem umsetzen. Einige der von MetaEdit+ unterstützten Möglichkeiten zur Bearbeitung grafischer Konstrukte wurden von Vogler aufgegriffen [141] und haben damit Einzug in die zur vorliegenden Dissertation gehörenden Implementierung (Abschnitt 8.2) gefunden.

Auch im wissenschaftlichen Umfeld gibt es mit dem auf Eclipse basierenden **Marama** ein System zur Top-Down Entwicklung und Verwendung grafischer DSMLs, das die Erstellung visueller Konstrukte nach dem WYSIWYG-Prinzip unterstützt. Es ist aus der Standalone-Applikation Pounamu [157] hervorgegangen, um Integrierbarkeit mit dem Eclipse Modeling Framework (EMF) [35] gewährleisten zu können. Der Vorteil von Marama gegenüber MetaEdit+ ist, dass alle grafischen Konstrukte einer Sprache in der gleichen Shape Designer Instanz (Pendant zum Symbol Editor bei MetaEdit+) entworfen werden. Dadurch erhält man eine grobe Vorschau, wie die einzelnen Formen in den späteren Diagrammen tatsächlich miteinander in Beziehung stehen können (Abbildung 3-2). Konträr zu MetaEdit+ sind die Interaktionsmöglichkeiten mit den einzelnen Konstrukten allerdings stark limitiert, was aber dem prototypischen Charakter akademischer Werkzeuge geschuldet ist.

Gegenwärtig wird Marama nicht mehr weiterentwickelt, wie der Webseite des Projekts [91] zu entnehmen ist. Der Quellcode steht jedoch unter einer Open Source Lizenz der Öffentlichkeit zur Verfügung.

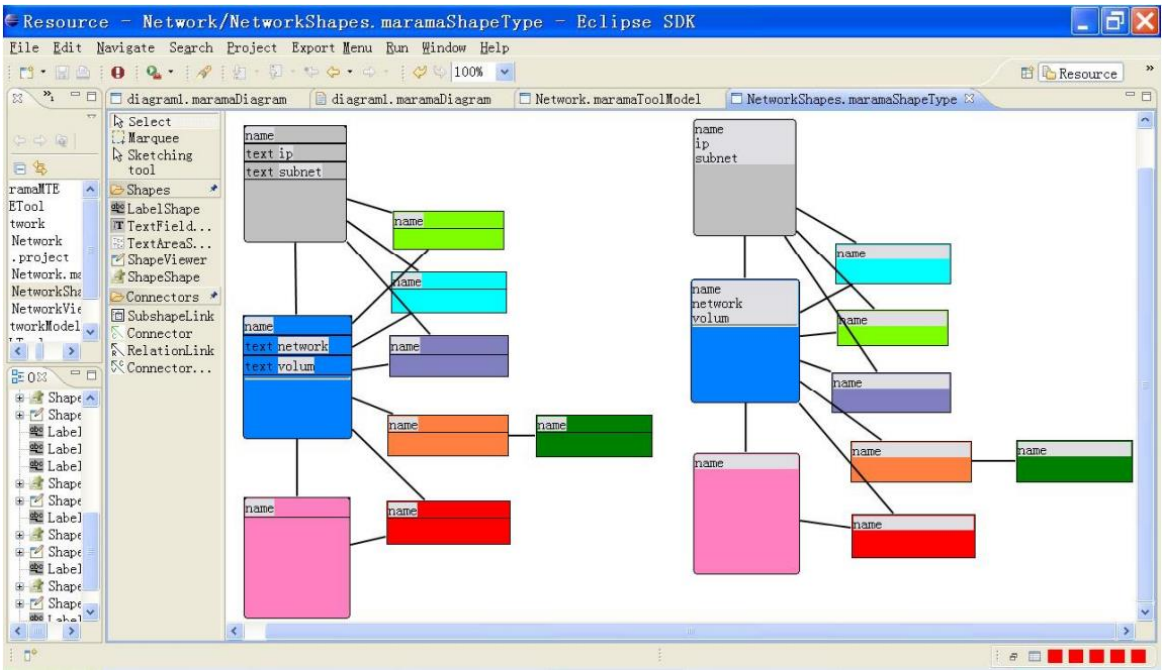

Abbildung 3-2: Shape Designer von Marama [59]

3.2 Ableiten einer abstrakten Syntax von einer konkreten Syntax

Die Ableitung einer abstrakten Syntax in Form eines Meta-Modells aus einer gegebenen konkreten Syntax ist ebenfalls Gegenstand dieser Dissertation (Abschnitt 5.3.2), weshalb im Folgenden verwandte Arbeiten zu dieser Thematik diskutiert werden. Als erste Publikation, die sich damit befasst, gilt nach unseren Erkenntnissen der 1997 erschienene Artikel „Abstract Syntax from Concrete Syntax" [147]. Wile präsentiert darin ein **auf Heuristiken beruhendes Rahmenwerk zur automatisierten Generierung eines objektorientierten Modells** (= Meta-Modell) basierend auf einer konkreten, textuellen Syntax, die in WBNF – einer Erweiterung von BNF (Abschnitt 2.1.2) – vorliegt. Die zentrale Neuerung von WBNF ist die Möglichkeit, Produktionsregeln annotieren und damit den Ableitungsprozess aktiv beeinflussen zu können. Den Ausgangspunkt des Rahmenwerks bildet eine Klassifikation von Produktionsregelmustern. Für jedes derartige Muster existiert eine Abbildungsvorschrift, die die konkreten Auswirkungen auf eine zu generierende abstrakte Syntax vorgibt. Ein exemplarischer Anwendungsfall folgt später bei der Beschreibung von Xtext, da deren Formalismus zur Definition von Grammatiken ähnlich, aber leichter verständlich ist.

Mit **MontiCore** [79] liefert Krahn einen Ansatz, der zwar auf demjenigen von Wile aufsetzt, doch verläuft die Ableitung einer abstrakten Syntax stark geradlinig. Das bedeutet, dass lediglich solche Artefakte im Meta-Modell generiert werden, zu denen es ein explizit spezifiziertes Pendant innerhalb der zugrundeliegenden Grammatik gibt. Es wird beispielsweise nur dann eine Generalisierungshierarchie eingeführt, wenn dies ausdrücklich angegeben ist. Diese auf den ersten Blick als Defizit anmutende Gegebenheit ist allerdings eine bewusste Designentscheidung von Krahn, der die damit propagierte Vorgehensweise als „Grammatikbasierte Erstellung von domänenspezifischen Sprachen" bezeichnet. Im Gegensatz zu Wile liegt der Fokus nämlich auf der kombinierten Spezifikation von konkreter und abstrakter Syntax innerhalb einer Grammatik [80].

Etwas anders verhält es sich mit **Xtext** [153], einem Framework zur Entwicklung textueller Domänensprachen. Es kann als Erweiterung von Wiles Ansatz betrachtet werden. Denn wie in der Dokumentation unter dem Punkt „Ecore Model Inference" [154] beschrieben, wird beim Generieren des Meta-

```
Process:
    'process' name=QualifiedName '{'
        tasks+=Task*
    '}';

HumanTask:
    'human' name=QualifiedName;

ScriptTask:
    'script' name=QualifiedName;

Task:
    HumanTask | ScriptTask;
```

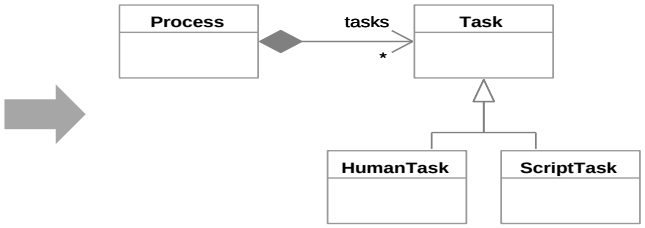

Abbildung 3-3: Beispiel einer (gekürzten) Xtext-Grammatik inklusive daraus abgeleitetem Meta-Modell

Modells versucht, auch komplexe Strukturen zu erzeugen (z.B. eine adäquate Typhierarchie). Einen Ausschnitt einer Xtext-Grammatik inklusive daraus abgeleitetem Meta-Modell zeigt Abbildung 3-3. Die Ausbildung der Generalisierungshierarchie findet laut Dokumentation deshalb statt, weil die Produktionsregel `Task` als Alternative der beiden Regeln `HumanTask` und `ScriptTask` spezifiziert ist.

Da alle vorgestellten **Ansätze** und Systeme auf Grammatiken beruhen, sind sie **nur bedingt** auf unsere, vollständig auf Meta-Modellen basierende Methode **übertragbar**. Allerdings können einige grundlegende Ideen, wie das Annotieren visueller Konstrukte oder die Einführung einer Generalisierungshierarchie nach dem Substitutionsprinzip, übernommen werden. Bei den untersuchten Ansätzen nicht vorgesehen ist außerdem die inkrementelle Ableitung von Änderungen an der abstrakten Syntax, nachdem die ursprüngliche Spezifikation der konkreten Syntax manipuliert wurde.

3.3 Rekonstruktion von Meta-Modellen

Anstelle von Grammatiken werden zunehmend Meta-Modelle für die Spezifikation von DSMLs eingesetzt. Analog dazu ging ebenso die Rekonstruktion von Meta-Modellen aus der Rekonstruktion von Grammatiken hervor. Eine aktuelle Studie [133] aus dem Jahr 2012 gibt einen Überblick über den Werdegang sowie den Status quo der **Rekonstruktion von Grammatiken** aus formalen Texten und Diagrammen. Ihr zufolge können drei maßgebliche Methoden differenziert werden, die auch kombiniert verwendbar sind. Vorweg zu erwähnen ist, dass die Ausgangsmodelle im Kontext der Sprachrekonstruktion üblicherweise Sätze oder Beispielsätze genannt werden.

Die erste Methode wird als „**Language identification in the limit**" (dt. „Spracherkennung im Grenzbereich") [52] bezeichnet. Sie erfordert eine möglichst große Menge an positiven und negativen Beispielsätzen, aus denen schließlich nach dem Brute-Force Prinzip eine Grammatik geschlussfolgert wird. Bei dem in der vorliegenden Arbeit vorgestellten Ansatz wird jedoch das Ziel verfolgt, dass bereits aus einer sehr kleinen Menge von Beispielmodellen (oftmals genau ein derartiges Modell) eine DSML abgeleitet werden kann. Des Weiteren ist die Bereitstellung negativer Beispiele nicht praktikabel, weil dadurch der Benutzer seinen Fokus von den tatsächlichen Objekten der Realität lösen und sich auf Fälle konzentrieren müsste, die außerhalb der betreffenden Domäne liegen.

Crepinsek et al. propagieren mit der Verfolgung eines **evolutionistischen Ansatzes** eine verbesserte Alternative [69] zum Brute-Force Prinzip, die aber weiterhin auf der Spracherkennung im Grenzbereich fußt. Im Wesentlichen wird dabei so vorgegangen, dass basierend auf einer gegebenen Menge positiver Beispielsätze eine Reihe von Grammatiken generiert wird. Letztere werden anschließend zufällig miteinander rekombiniert, so dass neue Grammatiken entstehen. Von denen werden die besten Vertreter ausgewählt (d.h. diejenigen, die eine möglichst große Anzahl positiver Sätze akzeptieren

und gleichzeitig möglichst viele negative Sätze ablehnen) und einer erneuten, zufälligen Rekombination unterzogen. Der Algorithmus terminiert, sobald alle positiven Beispielsätze angenommen und sämtlichen negativen Sätze abgewiesen werden.

„**Teacher and Queries**" (dt. „Lehrer und Befragung") ist die zweite Methode zur Rekonstruktion formaler Sprachen. Hierbei wird davon ausgegangen, dass eine als Lehrer betitelte Einheit zur Verfügung steht, die dem Ableitungsalgorithmus eine Reihe bestimmter Fragen hinsichtlich der einzelnen Beispielsätze beantworten kann [6]. In der Praxis wird der Lehrer häufig durch einen Menschen repräsentiert [133]. Genau diese Vorgehensweise greifen wir in Abschnitt 4.6 auf, wenn es um die Kompensation von lückenhaftem Domänenwissen geht.

Beim „**Probably Approximately Correct (PAC) learning**" (dt. „Wahrscheinlich annähernd richtiges Lernen") handelt es sich um die dritte Methode zur Sprachrekonstruktion [137]. Sie entlehnt zwar Konzepte aus den beiden anderen Verfahren, kann jedoch nicht verlässlich garantieren, dass die ermittelte Grammatik korrekt ist und zu allen vorgegebenen Beispielsätzen passt (daher das P in PAC). Wegen dieser Unsicherheit ist die PAC-Methode für unsere Zwecke nicht geeignet. Denn ein fälschlicherweise als richtig ausgegebenes Meta-Modell hätte unter Umständen zur Folge, dass frei modellierte Beispielmodelle invalidiert und damit irrtümlich als inkorrekt gekennzeichnet würden.

Mit dem MetAmodel Recovery System (**MARS**) [70] existiert eine Lösung, die die Techniken der Grammatikrekonstruktion auf die **Rekonstruktion von Meta-Modellen** überträgt. Grundlage bildet die oben vorgestellte, um evolutionistische Maßnahmen erweiterte Spracherkennung im Grenzbereich. Um allerdings Techniken der Grammatikrekonstruktion anwenden zu können, müssen die Beispielmodelle in eine einheitliche, textuelle Darstellung überführt werden. Die Ergebnisse einer auf formalen Texten beruhenden Rekonstruktion sind zunächst Produktionsregeln, die es anschließend in entsprechende Meta-Modellartefakte zu transformieren gilt. Hierbei wird auf Abbildungen zurückgegriffen, die stark denen der Ansätze aus Abschnitt 3.2 ähneln (z.B. führt die Regel `BASE → SUB1 | SUB2` zur Etablierung einer Generalisierungshierarchie).

Neben der Berücksichtigung von Beispielmodellen wird zudem der Quellcode eines gegebenenfalls vorhandenen Modellinterpreters ausgewertet. Die daraus extrahierbaren Informationen können primär zur Namens- und Typbestimmung herangezogen werden (insbesondere relevant bei abgeleiteten Attributen). Diese Vorgehensweise stellt eine Möglichkeit dar, um **Domänenwissen** zu **kompensieren**, das nicht aus den Beispielmodellen gewonnen werden kann. Die Existenz eines Modellinterpreters ist bei unserem Ansatz jedoch nicht gewährleistet, da wir nicht eine Rekonstruktion von Meta-Modellen anstreben, sondern deren aktive Entwicklung basierend auf einem neuartigen Vorgehensmodell.

Bei zunehmender Größe der Beispielmodelle liefert die ursprüngliche Fassung von MARS sehr komplexe Meta-Modelle als Resultat. Geschuldet ist dies dem Umstand, dass nicht alle in den Beispielmodellen verfügbaren Informationen in die Textdarstellung überführt werden. Deshalb wurde mit **Extended MARS** [86] eine Erweiterung von MARS entwickelt, die diese fehlenden Informationen ebenfalls berücksichtigt. Nichtsdestotrotz benötigt auch Extended MARS eine große Menge an Beispielmodellen, um zufriedenstellende Resultate zu erzielen.

Als **Fazit** bleibt festzuhalten, dass lediglich die zweite Methode (Lehrer und Befragung) zur Sprachrekonstruktion in ihren Wesenszügen für unsere Zwecke geeignet ist, weil bei den anderen Verfahren stets mindestens eine Grundvoraussetzung nicht erfüllt werden kann. Der Benutzer, der die Beispielmodelle erstellt, kann als Lehrer betrachtet werden. Er interagiert ohnehin ständig mit dem Modellierungssystem. Bei etwaigen Mehrdeutigkeiten kann das System deshalb unmittelbar Feedback einholen (Abschnitt 4.6).

3.4 Beispielgetriebene (Meta-)Modellierung

Beispielgetriebene Meta-Modellierung bedeutet die Entwicklung von Meta-Modellen auf der Basis von Beispielmodellen. Sie stellt einen zentralen Aspekt bei der beispielgetriebenen Entwicklung von DSMLs dar, die – wie in unserem Fall – auf Meta-Modellen beruhen (Abschnitt 2.1.3). Die beispielgetriebene Meta-Modellierung ist von dem aus den 1980er Jahren stammenden „Programming by Example"-Ansatz [58] inspiriert, bei dem es um die Erstellung von Computerprogrammen mithilfe von Beispielen geht, ohne dafür Code schreiben zu müssen.

Voraussetzung für beispielgetriebene Modellierung ist die Möglichkeit, **frei modellieren** zu können. Ein Großteil der vorhandenen, grafischen DSML-Systeme, die die Top-Down Vorgehensweise verfolgen, verwirklicht dies zumindest rudimentär (z.B. GMP [55] und MetaEdit+ [96]), während sämtliche nicht-projizierenden, textuellen Systeme keine diesbezügliche Unterstützung mitbringen. Bei den frei modellierten Elementen handelt es sich allerdings um rein visuelle, informelle Artefakte, die keinen Bezug zur abstrakten Syntax haben. Ein solcher Bezug kann konzeptbedingt nachträglich auch nicht hergestellt werden. Deshalb scheiden alle derartigen DSML-Werkzeuge aus und werden nicht näher betrachtet.

Die Autoren von **DiaMeta** [98], ein System zur Entwicklung von Diagrammeditoren auf der Basis von Meta-Modellen, verwenden mit **„Freihand-Modellierung"** eine recht ähnliche Begrifflichkeit wie „freies Modellieren". Im Gegensatz zu unserem Ansatz bezwecken die Entwickler von DiaMeta damit nicht eine Änderung der zugrundeliegenden Modellierungssprache, sondern wollen dem Anwender eine bessere Benutzererfahrung bieten. Erreicht werden soll dies durch Bewilligung eines größeren Bearbeitungsspielraums als vom zugrundeliegenden Meta-Modell vorgegeben. Erst im Zuge einer nachgeschalteten, auf Graphgrammatiken basierenden Validierung wird überprüft, ob das erstellte Modell dem Meta-Modell genügt. Falls dem nicht so ist, werden ungültige Artefakte entsprechend gekennzeichnet. Eine Beziehung zwischen den informalen (fehlerträchtigen) Artefakten und Elementen der abstrakten Syntax kann aber ebenso wenig geschaffen werden.

Im Folgenden werden verschiedene Ansätze erörtert, die im weitesten Sinne der beispielgetriebenen Meta-Modellierung zugerechnet werden können. Die Diskussion erfolgt dabei unter Rückbezug auf die in Abschnitt 1.2 aufgestellten Anforderungen. Durch die farbige Markierung der jeweiligen Anforderung ist erkennbar, ob diese der betreffende Ansatz erfüllt (grün), teilweise erfüllt (gelb) oder nicht erfüllt (orange).

3.4.1 Clafer

Mit „beispielgetriebene Modellierung" existiert ein Ansatz [155], der ähnlich betitelt ist wie die vorliegende Dissertation. Ziel dessen ist jedoch weniger eine allumfassende Werkzeugunterstützung, bei der eine DSML (semi-)automatisch anhand von Beispielen induziert wird, als vielmehr eine durch Beispiele getriebene Vorgehensweise zur **manuellen, iterativen Entwicklung von Modellierungssprachen** (primär der abstrakten Syntax).

Das Hauptaugenmerk liegt dabei auf der Mehrstufigkeit, die sich folgendermaßen ausdrückt. Im ersten Schritt werden zusammen mit einem Domänenexperten Beispielmodelle erstellt, wofür die leichtgewichtige textuelle Sprache **Clafer** [7] zum Einsatz kommt. Sie ähnelt von ihrem Charakter der in Abschnitt 2.2.3 präsentierten LML, da mit ihrer Hilfe sowohl Modelle als auch zugehörige Meta-Modelle spezifiziert werden können. Dennoch ist das Vorhandensein eines Meta-Modells nicht erforderlich, um entsprechende Modelle zu erstellen. Freie Modellierung im linguistischen Kontext wird somit nativ unterstützt, wodurch der Grundstein für Anforderung 2 gelegt ist. Die automatische Induktion eines Meta-Modells wird dabei jedoch nicht ermöglicht.

Ist die erste Iteration mit der Erstellung von Beispielmodellen abgeschlossen, dann entwirft ein Modellierungsspezialist darauf beruhend ein passendes Meta-Modell (in [7] als Abstraktion bezeichnet). Anhand dessen werden im Anschluss weitere (sowohl positive als auch negative) **Beispielmodelle automatisiert generiert und ebenso automatisiert validiert**. Auf Basis der Validierungsergebnisse wird schließlich das zugrundeliegende Meta-Modell vom Modellierungsspezialisten korrigiert. Dieser Prozess wird solange wiederholt, bis das resultierende Meta-Modell die jeweilige Domäne adäquat beschreibt.

Bei der erläuterten Vorgehensweise handelt es sich neben der Anwendung von etablierten Sprach- und Entwurfsmustern um eine weitere Möglichkeit, die zur **Qualitätssicherung bei Meta-Modellen** beiträgt und demzufolge Anforderung 3 erfüllt. Zu dem Zweck wird sie auch von dem in Abschnitt 3.4.3 vorgestellten MLCDB-Ansatz aufgegriffen und eingesetzt. Nachdem die Definition einer eigenen, konkreten Syntax nirgends Beachtung findet, wird Anforderung 1 in ihrer Gesamtheit nicht adressiert.

3.4.2 BITKit

Die Idee hinter BITKit ist, die **anfängliche Anforderungsaufnahme bei der Entwicklung eines Softwareprodukts** derart zu unterstützen, dass man Anforderungen so frei modellieren kann wie mit gängigen Präsentationswerkzeugen (z.B. Microsoft PowerPoint oder OpenOffice Impress) [31]. Auch die Benutzeroberfläche erinnert an derartige Applikationen, was anhand des Screenshot in Abbildung 3-4 ersichtlich ist. Während der Modellierungsphase wird eine abstrakte Syntax aus dem Inhalt des Diagramms und eventuellen Benutzereingaben vom System abgeleitet. Dabei erweitert BITKit eine bereits fest vorgegebene Menge an Basistypen wie *Entity*, *Set* oder *Relationship* um die modellierten Konstrukte aufzunehmen [113]. Folglich beschränkt sich BITKit auf zwei Meta-Ebenen, von denen nur die Instanzebene direkt beeinflusst werden kann. Durch diese Einschränkung kann die Meta-Ebene nur von BITKit selbst manipuliert werden. Meta-Modellierung ist somit nicht möglich, was die Erfüllung von Anforderung 2 verhindert.

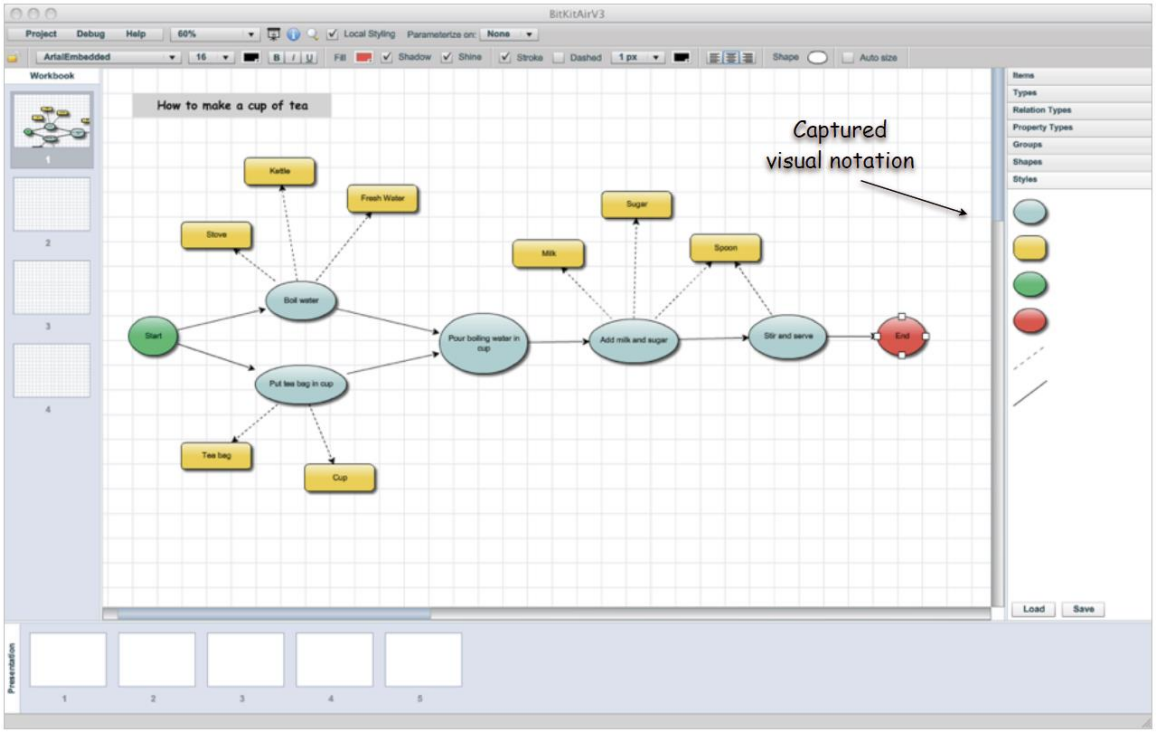

Abbildung 3-4: Screenshot der Benutzeroberfläche von BITKit

Da BITKit **keine vollständige Meta-Modellierungsumgebung** darstellen soll, beschäftigt sich das System nicht mit der Erzeugung prägnanter und kompakter Meta-Modelle. Anforderung 3 wird daher ebenso wenig erfüllt. Vielmehr steht bei diesem Projekt eine auf Wiederverwendung und Konsistenz setzende, einfache Modellierungsweise im Vordergrund, die diejenige von typischen Office-Werkzeugen ergänzt. Wird beispielsweise die Farbe einer konkreten Instanz eines bestimmten Typs geändert, so wirkt sich dies ebenfalls auf alle anderen, zugehörigen Instanzen aus.

Anforderung 1a kann als teilweise erfüllt betrachtet werden, weil BITKit zwar **im grafisch ontologischen Kontext angesiedelt** ist und dort die Entwicklung grafischer Modellierungssprachen ermöglicht. Allerdings können visuelle Konstrukte nur geringfügig an die Bedürfnisse der Anwender angepasst werden. So ist es z.B. nicht möglich, komplexe Konstrukte durch Komposition zu erstellen. Nicht unterstützt wird die textuelle Modellierung, weshalb Anforderung 1b nicht verwirklicht ist.

3.4.3 MLCBD

Cho beschreibt mit „**Modeling Language Creation By Demonstration (MLCBD)**" [21] (dt. „Erstellung von Modellierungssprachen durch Demonstration") einen Ansatz für den Entwurf von DSMLs anhand grafischer Beispiele. Diese Beispiele werden zunächst mit einem Diagrammwerkzeug als Fronend-Modul erstellt. Anschließend werden sie dem Backend-Modul übergeben, das daraus die verschiedenen Komponenten einer DSML induziert.

Das *Frontend-Modul* ist prototypisch mittels Microsoft Visio implementiert (Abbildung 3-5). Als generisches Zeichenwerkzeug unterstützt Visio die **freie Modellierung** von Haus aus. Außerdem wartet Visio mit einer umfangreichen Auswahl an mitgelieferten Standardformen auf, die als Ausgangspunkt für die zukünftigen, visuellen Konstrukte der beabsichtigten Sprache dienen. Wird eine neue, bislang noch nicht verwendete Form im Diagramm platziert, dann landet sie zudem in einer separaten Palette, die die einzelnen Konstrukte der konkreten Syntax umfasst. Diese Konstrukte können darin mit einem aussagekräftigen Namen versehen werden, wie es Abbildung 3-5 (links) zeigt. Das gewährleistet ein besseres Wiederfinden der selbigen.

Die **Erkennung von eigenständigen Konstrukten der konkreten Syntax** findet im *Backend-Modul* statt. Dieses Modul verwendet Techniken der Graph-Theorie [36], weshalb zunächst alle Beispielmodelle in eine Menge attributierter Graphen transformiert werden. Die Knoten des Graphen stammen von den verschiedenen Formen, die innerhalb der zugehörigen Diagramme liegen. Durch Vergleich der einzelnen Knoten miteinander anhand ihrer Attribute werden schließlich Kandidaten für Konstrukte der konkreten Syntax identifiziert und in der oben erwähnten Palette des Frontend-Moduls angezeigt. Analoges trifft für Kanten des Graphs zu, die ihren Ursprung in visualisierten Verbindungslinien haben.

Eine weitere, zentrale Aufgabe des *Backend-Moduls* bildet die **Ableitung einer abstrakten Syntax** in Form eines Meta-Modells. Der Vorgang beruht ebenso auf der zuvor genannten Graph-Darstellung. Diese Darstellung wird mit Entwurfsmustern abgeglichen, die daher ebenfalls in Graph-Repräsentation vorliegen und sich schon in anderen grafischen Sprachen etabliert haben [22]. Beim Auffinden passender Muster werden entsprechende Artefakte innerhalb der abstrakten Syntax generiert. Das kann zur Erzeugung von Meta-Modellelementen führen, die dieselbe semantische Entität widerspiegeln. In einem sich anschließenden Kombinierungsschritt werden derartige Mehrfachnennungen schließlich eliminiert.

Aufgrund der automatischen Einführung von Entwurfsmustern und wegen der Möglichkeit, visuelle Konstrukte annotieren zu können (z.B. mit einem aussagekräftigen Namen), ist sichergestellt, dass das resultierende Meta-Modell gewissen Qualitätsansprüchen genügt. Anforderung 3 kann demnach als erfüllt betrachtet werden. Allerdings birgt die automatisierte Einführung von Entwurfsmustern die

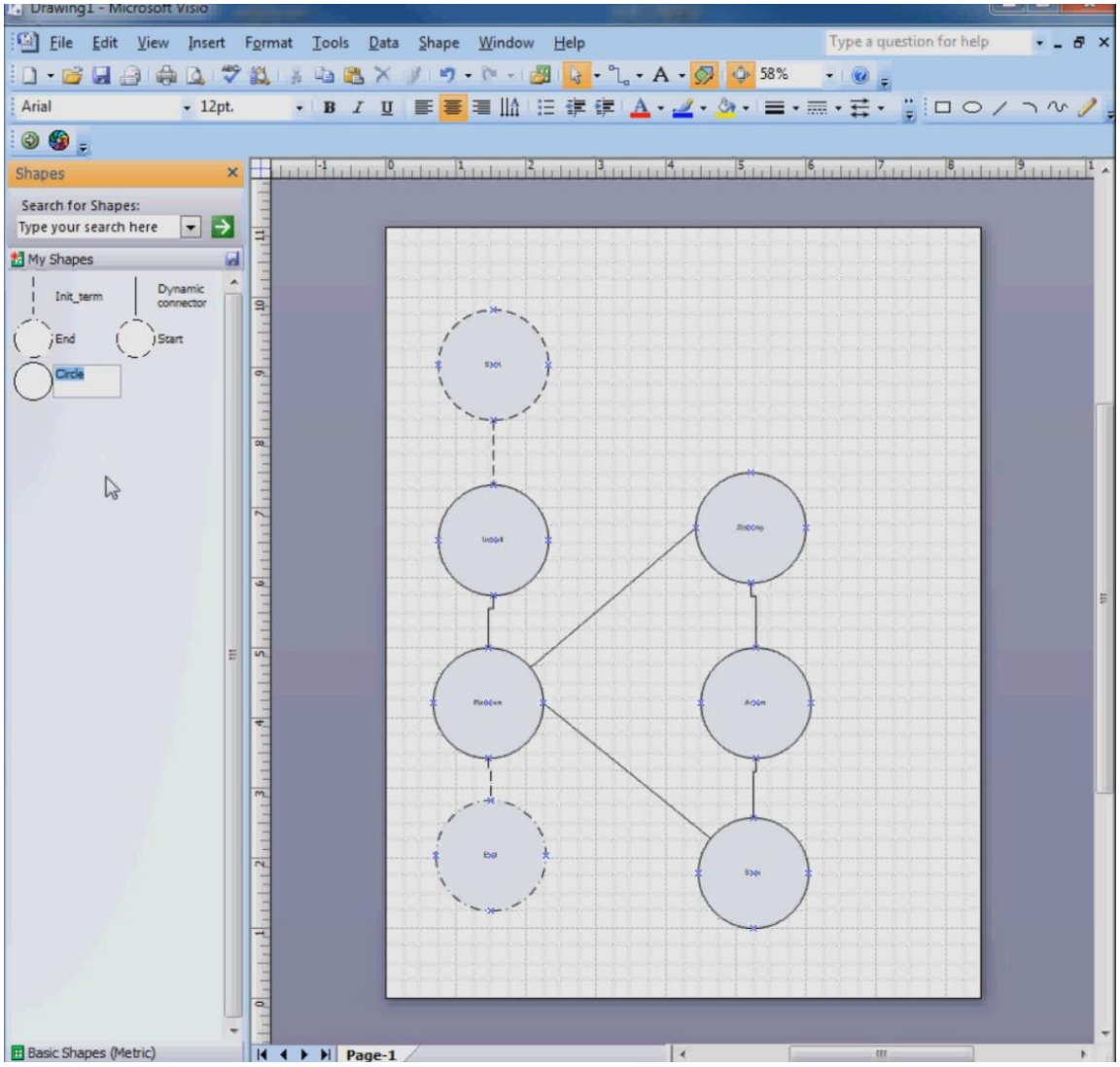

Abbildung 3-5: Screenshot des auf Microsoft Visio basierenden MLCBD-Werkzeugs [145]

Gefahr von Fehlinterpretationen in sich, weswegen wir deren Anwendung lediglich empfehlen (Abschnitt 4.4) und eine direkte Methode beim Generieren der abstrakten Syntax präferieren (Abschnitte 4.3 und 5.3.2). Es bleibt dadurch dem Benutzer überlassen, ob er ein vorgeschlagenes Muster einführt oder ignoriert.

Ebenfalls im *Backend-Modul* verortet ist eine Komponente, die dem Defizit begegnet, dass häufig nur eine verhältnismäßig kleine Menge an Beispielmodellen für die Ableitung einer abstrakten Syntax zur Verfügung steht. Sie verfolgt dabei das **Prinzip der beispielgetriebenen Modellierung**, wie in Abschnitt 3.4.1 vorgestellt. Basierend auf dem abgeleiteten Meta-Modell werden also automatisiert Beispielmodelle generiert, die im Anschluss vom Benutzer zu verifizieren sind.

Das *Backend-Modul* ist unter Verwendung von Makros implementiert. Gleiches gilt für die Unterstützung der **stringenten Modellierung**, deren zuzurechnende Makros das abgeleitete Meta-Modell sowie darin enthaltene statische Constraints (z.B. Kardinalitäten und Typinformationen) auswertet und daraufhin bestimmte Modellierungstätigkeiten unterbindet.

Unterstützt wird nur das initiale Ableiten einer DSML. Kommen neue Beispielmodelle hinzu, die nicht zur bisherigen Modellierungssprache passen, dann muss die Sprache erneut vollständig abgeleitet

werden. Demzufolge erfüllt MLCDB Anforderung 1a nur partiell. Darüber hinaus wird nur die verbindungsbasierte Klasse grafischer Sprachen [28] verfolgt, d.h. ein Verschachteln von Formen ist nicht möglich.

Nachdem der Fokus auf grafischen DSMLs liegt und sich deshalb für eine Umsetzung mit Microsoft Visio entschieden wurde, wird Anforderung 1b nicht Genüge getan. Entsprechendes trifft für Anforderung 2 zu, da Visio nicht als Meta-Modellierungswerkzeug konzipiert ist und somit nichts Vergleichbares wie die LML zur Verfügung steht.

3.4.4 MetaBUP

MetaBUP [95] ist ein noch in der Entwicklungsphase steckendes Werkzeug zum **inkrementellen Entwurf von Meta-Modellen nach dem Bottom-Up Prinzip** für beliebige Modellierungsplattformen (z.B. EMF [35]). Alle zugehörigen wissenschaftlichen Artikel [87, 124] wurden während der Entstehung der vorliegenden Arbeit publiziert, so dass letztere nur eingeschränkt von den Erkenntnissen jener Artikel profitieren kann. Nachfolgend werden die Parallelen von MetaBUP mit unserem Ansatz aufgezeigt, wobei sich auf die besagten Quellen bezogen wird.

Der von MetaBUP propagierte Modellierungsprozess (Abbildung 3-6) beginnt mit **frei modellierten Beispielmodellen**, die entweder **als Diagramme oder als Textfragmente**, in einer eigens dafür entworfenen Sprache vorliegen. Diese Sprache dient zur formalen Beschreibung und gleichzeitigen Annotation von Beispielmodellen. Zur einheitlichen Verarbeitung werden auch alle Diagramme in diese Textdarstellung überführt. Als Klientel für Diagramme gelten Domänenexperten, die jene mithilfe von Zeichenwerkzeugen wie Microsoft Visio oder Dia erstellen. Die Textfragmente hingegen werden von Modellierungsspezialisten produziert, weshalb diese Spezialisten auch in der Lage sind, die von Diagrammen abstammenden Fragmente um zusätzliche Informationen anzureichern (z.B. in Form von Annotationen).

Im nächsten Schritt wird aus den frei erstellten Textfragmenten ein technologieunabhängiges und damit neutrales Meta-Modell induziert. Kommen später weitere Textfragmente hinzu, dann wird ein bereits vorhandenes Meta-Modell inkrementell angepasst. Sollten hierbei nicht zweifelsfrei auflösbare

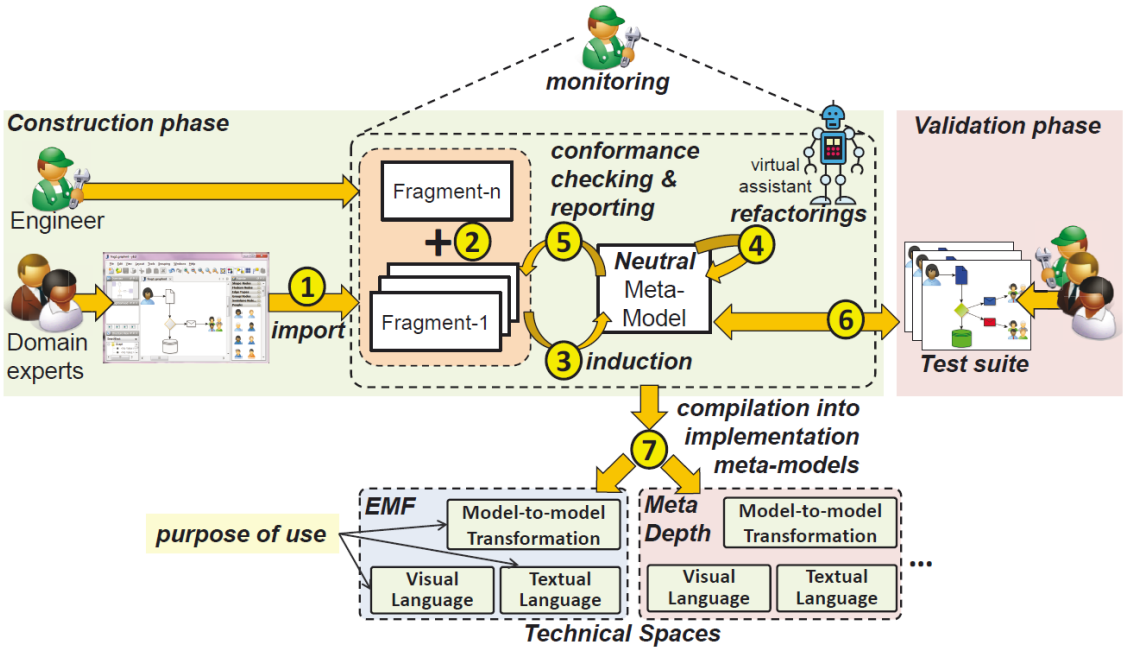

Abbildung 3-6: Workflow der Bottom-Up Meta-Modellierung mit MetaBUP [87]

Konflikte auftreten, so wird Feedback vom Benutzer eingefordert, indem ihm im Optimalfall verschiedene Lösungsstrategien angeboten werden. Auf diese Weise wird die **beispielgetriebene Entwicklung von Meta-Modellen im linguistischen Kontext** vollständig unterstützt (Anforderung 2).

Nach der inkrementellen Ableitung eines Meta-Modells tritt ein als virtueller Assistent bezeichnetes Analysemodul in Aktion, das im Meta-Modul nach Konstellationen von Modellelementen mit Verfeinerungspotential sucht. Werden solche Elemente identifiziert, dann wird dem Benutzer ein Vorschlag unterbreitet, eine entsprechende Refaktorisierung vorzunehmen, die wiederum vom System geleitet wird. Beispielsweise wird empfohlen, zwei Klassen zu einer zu verschmelzen, wenn sich beide gegenseitig über eine 1-zu-1-Beziehung referenzieren. Demzufolge wird unter Verwendung automatisch durchgeführter Analysen versucht, die **Qualität des generierten Meta-Modells sicherzustellen**, was mit der Erfüllung von Anforderung 3 einhergeht.

Zur Qualitätssicherung trägt auch das automatisierte Testen bzw. **Validieren von zusätzlichen Textfragmenten** bei, die ihren Ursprung abermals in von Domänenexperten frei modellierten Diagrammen haben können. Dabei können sowohl positive als auch negative Beispielmodelle durch den Ableitungsalgorithmus überprüft werden, der anstelle einer Anpassung des Meta-Modells Auskunft darüber gibt, ob die Eingabe den Erwartungen entspricht. Im Falle von positiven Beispielen wird als Ausgabe erwartet, dass keine Evolution des Meta-Modells notwendig ist, während bei negativen Beispielen mit dem Gegenteil gerechnet wird. Eine derartige Funktionalität liegt außerhalb des Untersuchungsspektrums dieser Dissertation, weil explizit die Berücksichtigung von Constraints ausgeklammert wurde (Abschnitt 1.3). Dennoch ist es lohnenswert diesen Aspekt in künftigen Arbeiten zu intensivieren.

Ebenso im Zusammenhang mit Constraints stehen die sogenannten **Domänen-Annotationen**. Sie bieten dem Modellierer eine einfache Möglichkeit, Artefakte der Textfragmente mit **häufig verwendeten Restriktionen** zu annotieren, wobei sich die Annotationen erst auf die später davon abgeleiteten Elemente des Meta-Modells auswirken. Ein typisches Beispiel ist die Annotation `@acyclic`, mit der ausschließlich Referenzen versehen werden dürfen. Sie besagt, dass es nicht gestattet ist Zyklen zu formulieren, die durch Instanzen dieser Referenz zustande kommen.

Daneben gibt es die **Design-Annotationen**, deren Zweck die einfache, auf Beispielebene angesiedelte **Umsetzung von Entwurfsentscheidungen** ist. Exemplarisch zu nennen ist die `@bidirectional`-Annotation, die bedingt, dass für eine entsprechend annotierte Referenz zwei bidirektionale Attribute im Meta-Modell erzeugt werden. Aufgrund des Ausschlusses von Constraints unterstützen wir zwar keine Domänen-Annotationen; anders sieht es hingegen mit den Design-Annotationen im ontologischen Kontext aus (Abschnitt 5.3.1.4).

Wie bereits oben erwähnt, wird ein **technologieunabhängiges Meta-Modell** induziert. Das bedeutet allerdings, dass es nicht möglich ist, Instanzen (Modelle) dieses Meta-Modells zu erzeugen, wodurch eine direkte Weiterverarbeitung ausgeschlossen ist. Hierzu muss es zunächst auf eine konkrete Technologie (z.B. EMF) abgebildet werden. Der damit verbundene **Technologiebruch** bedingt, dass die frei modellierten Textfragmente nicht als zukünftige Instanz-Modelle eingesetzt werden können. Ebenso wenig werden die visuellen Bausteine der bisherigen, von Domänenexperten gezeichneten Diagramme für eine konkrete, grafische Syntax innerhalb der Zieltechnologie verwendet. Soll bei der Realisierung mit einer bestimmten Technologie ebenfalls eine grafische Sprache umgesetzt werden, dann muss die konkrete Syntax von Grund auf neu definiert werden. Deshalb gilt Anforderung 1a als lediglich teilweise erfüllt. Da frei modellierte, textuelle Modelle ohne eine vorgegebene, konkrete Syntax nicht adressiert werden, erfährt Anforderung 1b keinerlei Beachtung.

3.4.5 FlexiSketch

FlexiSketch [150] ist ein weiteres, in der Entstehung befindliches Werkzeug, das als Android-App implementiert ist und den Entwurf von Meta-Modellen auf der Basis von Diagrammen thematisiert. Primäres Ziel ist jedoch die Erhebung und **Modellierung von Anforderungen mithilfe von Freihandzeichnungen** [151]. Demzufolge liegt der Fokus nicht auf der Erzeugung qualitativ hochwertiger Meta-Modelle, sondern auf der Ableitung einfacher Meta-Modelle, die im XML-Format exportiert und daraufhin extern weiterverarbeitet werden können. Der Qualitätsaspekt ist somit ausgelagert und wird nicht von FlexiSketch berücksichtigt (Anforderung 3).

Durch eine **automatische Skizzenerkennung** liegen die Freihandzeichnungen bereits als semi-formale Modelle vor. Diese Erkennung geht allerdings nicht so weit, dass sie grafische Konstrukte als eigenständige Typen identifiziert und in einer separaten Palette ablegt. Das muss der Benutzer manuell erledigen, indem er auf ein Kontextmenü zurückgreift [151]. Danach erst kann er die abgelegten Konstrukte im Diagramm wiederverwenden. Ein nachträgliches Modifizieren bereits vorhandener Konstrukte ist nicht möglich, weshalb Anforderung 1a nur teilweise erfüllt wird. Anforderung 1b dagegen gilt als gänzlich nicht erfüllt, weil textuelles Modellieren nicht unterstützt wird. Es ist lediglich erlaubt, die verschiedenen Elemente im Diagramm zu beschriften, wie in Abbildung 3-7 illustriert.

Aufgrund der Formalisierung von Skizzen inklusive Bereitstellung einer Exportfunktion für ein induziertes Meta-Modell umgeht FlexiSketch einen Medienbruch. Nichtsdestotrotz liegt genau wegen dieser Exportfunktionalität ein **Technologiebruch** vor, denn für eine Weiterverarbeitung eines exportierten Meta-Modells muss dieses zunächst in ein externes Meta-Modellierungssystem importiert werden. Nachdem der Export offensichtlich auch Darstellungsinformationen umfasst (siehe `<appearance>`-Tags im abgebildeten Beispielexport in [150]), können die gelieferten Visualisierungsdaten innerhalb des Meta-Modellierungssystems für eine konkrete, grafische Syntax wiederverwertet werden. Beispielgetriebene Entwicklung von Meta-Modellen im linguistischen Kontext ist wegen der Verlagerung von Meta-Modellierungsaufgaben in eine andere Applikation nicht möglich (Anforderung 2). Eine linguistische Metasprache könnte allenfalls dort zur Verfügung gestellt werden und läge somit außerhalb von FlexiSketch.

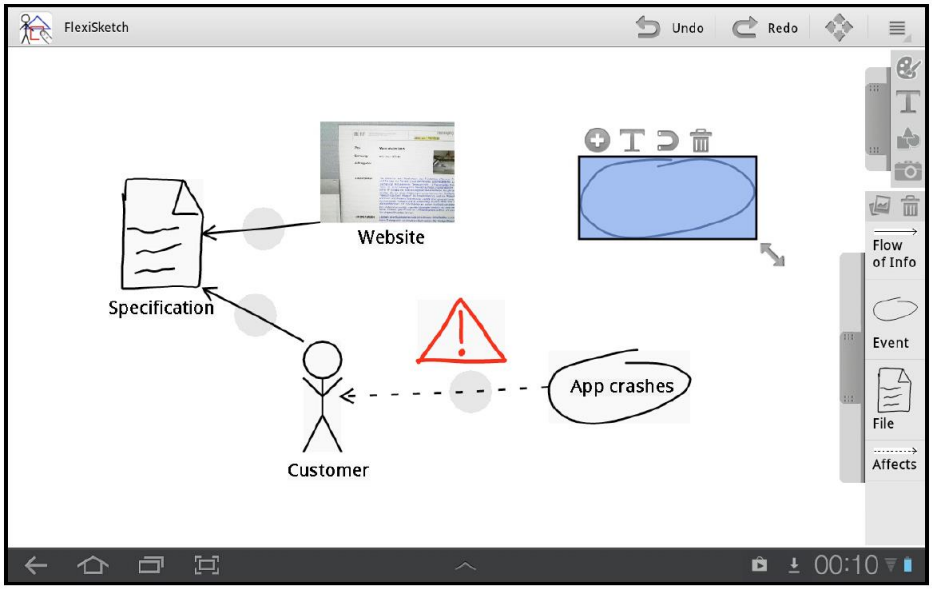

Abbildung 3-7: Screenshot der FlexiSketch Android-App [150]

3.5 Fazit

Die im Kontext der beispielgetriebenen (Meta-)Modellierung betrachteten Ansätze und Werkzeuge legen ihr Hauptaugenmerk auf unterschiedliche Aspekte. Deshalb werden sie nachfolgend gegenübergestellt und einem abschließenden Vergleich unterzogen, wobei insbesondere Kriterien berücksichtigt werden, die für diese Arbeit von Relevanz sind. Da kein System die beispielgetriebene Entwicklung textueller DSMLs (Anforderung 1b) unterstützt, sind immer grafische Modellierungssprachen gemeint, sofern allgemein von DSMLs die Rede ist.

Allen in den Abschnitten 3.3 und 3.4 untersuchten Ansätzen liegt die **Closed World Assumption** (dt. „Annahme zur Weltabgeschlossenheit") zugrunde. Sie besagt, dass mit den aktuell zur Verfügung stehenden Informationen die Welt in ihrer Gesamtheit beschrieben ist. Andernfalls wäre es nicht möglich, auf der Basis von wenigen Beispielmodellen irgendwelche Aussagen über die Realität zu treffen, was die automatische Induktion eines Meta-Modells unmöglich macht.

Außerdem beruhen sämtliche Ansätze, die automatisiert ein Meta-Modell ableiten, auf der Annahme, dass **gleichlautende Artefakte denselben semantischen Hintergrund** haben. MLCDB geht noch einen Schritt weiter und überträgt diese Annahme auch auf das Aussehen von grafischen Konstrukten. Dort gilt z.B., dass zwei Kreise auf demselben visuellen Konstrukt basieren, sofern sich ihre Styling-Eigenschaften gleichen.

Die eigentliche Gegenüberstellung erfolgt in Tabelle 3-1. Die hierbei eingesetzten Farben werden auf dieselbe Weise verwendet, wie zur Hervorhebung der Anforderungen innerhalb der vorherigen Abschnitte. In der nun folgenden Beschreibung wird lediglich auf die Punkte eingegangen, die bisher noch nicht behandelt wurden oder mit den Informationen in der Tabelle nicht vollständig verständlich sind.

Hinter dem Kriterium „linguistisches Bottom-Up" verbirgt sich im Wesentlichen Anforderung 2. Dasselbe gilt für den Punkt „Qualitätssicherung", der stattdessen Anforderung 3 adressiert. In beiden Fällen entsprechen die Ergebnisse denen der vorausgegangenen Analysen.

Tabelle 3-1: Gegenüberstellung der evaluierten Ansätze zur beispielgetriebenen Meta-Modellierung

	Clafer	BITKit	MLCBD	MetaBUP	FlexiSketch
Linguistisches Bottom-Up	freie Modellierung	nein	nein	ja	nein
Qualitätssicherung	primäres Ziel	nein	ja (Entwurfsmuster & manuelle Validierung)	ja (Verfeinerungsvorschläge & Auto-Validierung)	nein
Ableiten einer abstrakten Syntax	nein	intern als ER-Spezialisierung	ja, als Visio-Makros (nur vollständig)	ja (inkrementell)	inkrementell, aber rudimentär
Annotationen zur Anreicherung	nein	ja	ja	ja (große Anzahl)	ja
Definition einer konkreten Syntax	nein	einfache Formen (Änderungen möglich)	einfache Formen & Bilder (Änderungen nicht möglich)	nein	komplett manuell (Änderungen nicht möglich)
Weiterverarbeitung der Artefakte	geplant	nein	nein	mit anderer Technologie	mit anderer Technologie
Berücksichtigung von Constraints	ja, mit Constraintssprache	nein	rudimentär	ja, mit Annotationen	nein

Die am weitesten fortgeschrittene Methode beim **Ableiten einer abstrakten Syntax** stellt *MetaBUP* bereit. Das liegt vor allem daran, dass der Ableitungsprozess inkrementell funktioniert und mithilfe einer Vielzahl von Annotationen gesteuert werden kann. Inkrementell ist auch der Algorithmus von *FlexiSketch*. Allerdings weist das induzierte Meta-Modell noch starkes Verfeinerungspotential auf, da die in den Beispielmodellen erkannten Strukturen identisch übernommen werden. Es findet lediglich ein Pruning statt, bei dem mehrfach vorkommende, aber gleichartige Artefakte entfernt werden. Auch die Autoren von *BITKit* kommen zu dem Resultat, dass es sich beim Ableiten einer abstrakten Syntax um ein inkrementelles Verfahren handelt [113]. Sie liefern jedoch keine konkrete Lösung. Zudem dient das abgeleitete Meta-Modell primär der systeminternen Unterbreitung von Handlungsempfehlungen sowie zugehörigen Validierungszwecken und ist deshalb nicht für die Außenwelt zugänglich. *MLCBD* hingegen unterstützt das inkrementelle Ableiten einer abstrakten Syntax nicht. Kommen neue Beispielmodelle hinzu oder werden vorhandene Modelle angepasst, dann wird das bisherige Meta-Modell verworfen und eine vollständige Neu-Generierung initiiert. Wurde das Meta-Modell zwischenzeitlich manuell modifiziert (durch Manipulation der Visio-Makros), dann gehen diese Änderungen verloren.

Das **Annotieren von Artefakten der Beispielmodelle** wird von allen Werkzeugen unterstützt, die automatisiert ein Meta-Modell ableiten können. Diese Funktion dient dazu, Beispielmodelle mit zusätzlichem Domänenwissen anzureichern und damit die Qualität der generierten Meta-Modelle zu steigern. In den Zellen von *BITKit* und *FlexiSketch* zur Qualitätssicherung ist dennoch ein „nein" vermerkt, da sie hierfür keine weiterführenden Maßnahmen bieten, die halbwegs mit den anderen Ansätzen vergleichbar sind.

Das **Definieren von Konstrukten für eine konkrete, grafische Syntax** wird von dreien der evaluierten Systeme nur in beschränktem Umfang unterstützt. Am besten schneidet *BITKit* ab, weil es auch Änderungen an bereits existierenden Konstrukten gestattet. Allerdings erlaubt kein Werkzeug die Komposition visueller Konstrukte unter Verwendung vorgegebener Formen. Ein typisches Anwendungsszenario für eine solche Komposition ist aus BPMN bekannt. Dort ist eine Aufgabe als Benutzeraufgabe gekennzeichnet, sofern das die Aufgabe repräsentierende Rechteck über ein kleines Männchen in der linken oberen Ecke verfügt. Veranschaulicht ist dies am grafischen Beispiel in Abbildung 2-1.

Die direkte **Weiterverarbeitung der einzelnen Artefakte** (Beispielmodelle sowie generiertes Meta-Modell) wird derzeit von keinem Werkzeug unterstützt. Geplant ist es jedoch mit *Clafer* im Kontext der beispielgetriebenen Modellierung [7]. Bei *MetaBUP* und *FlexiSketch* muss hierfür auf externe Technologien zurückgegriffen werden, wobei Letztgenanntes auch Elemente der konkreten Syntax zur Weiterverwendung vorsieht.

Als abschließendes Kriterium ist die **Berücksichtigung von Constraints** aufgeführt, obwohl das im Rahmen dieser Dissertation explizit ausgeklammert wurde (Abschnitt 1.3). Nichtsdestotrotz nehmen Constraints bei DSMLs einen hohen Stellenwert ein. Eine geeignete und zugleich einfache Lösung für die Integration von Constraints in die beispielgetriebene Meta-Modellierung liefert MetaBUP. Es ist daher ratsam, diese Lösung in den in der vorliegenden Arbeit vorgestellten Ansatz zu integrieren.

Zusammenfassend bleibt festzuhalten, dass sich der Großteil der Aktivitäten im Umfeld der beispielgetriebenen Entwicklung von DSMLs auf grafische Sprachen konzentriert. Ebenfalls im Fokus steht die Induktion eines Meta-Modells, das die abstrakte Syntax der zu erstellenden DSML repräsentiert. Mit den in Abschnitt 3.1 präsentierten Werkzeugen existieren auch Lösungen, die ein angenehmes Definieren einer konkreten, grafischen Syntax ermöglichen und daher bei der Realisierung von Anforderung 1a bedacht werden. Aus den in den Abschnitten 3.2 und 3.3 vorgestellten Arbeiten lassen sich nur Methoden übertragen, die ohnehin schon von den Ansätzen in Abschnitt 3.4 genutzt werden (z.B. Einfordern von Feedback durch den Benutzer, Annotieren visueller Konstrukte oder Einführung einer Generalisierungshierarchie nach dem Substitutionsprinzip).

4 Beispielgetriebene Entwicklung im linguistischen Kontext

Im vorausgehenden Kapitel 3 wurden diverse verwandte Arbeiten analysiert und diskutiert, die Teillösungen für die in Abschnitt 1.2 geschilderten Anforderungen liefern oder sie zumindest adressieren. Keiner dieser Ansätze erfüllt allerdings alle drei Anforderungen in ihrer Gesamtheit. Nachdem der linguistische Kontext eine Teilmenge des ontologischen Kontexts darstellt, wird mit der Behandlung von Anforderung 2 begonnen. Dazu wird in diesem Kapitel eine Methodik eingeführt, die die beispielgetriebene Entwicklung von DSMLs im linguistischen Kontext ermöglicht. Im Fokus steht hierbei eine auf Heuristiken fußende **Induktion eines Meta-Modells** aus einer Menge frei modellierter Beispielmodelle. Auch Änderungen, die im freien Modus an Beispielmodellen vollzogen werden, können auf ein bereits zugrundeliegendes Meta-Modell übertragen werden. Dieses Meta-Modell kann als abstrakte Syntax einer DSML aufgefasst werden, wobei die Beispielmodelle ontologische Instanzen dieser DSML sind. Die konkrete Syntax ist durch die in Abschnitt 2.2.3 vorgestellte LML fest vorgegeben. Sie ist zwar für linguistisches Modellieren gedacht, doch nachdem sie naturgemäß auch für die Beschreibung von Beispielmodellen eingesetzt werden kann, repräsentiert sie ebenfalls eine in ontologischer Hinsicht generische konkrete Syntax.

Zur Umsetzung der beispielgetriebenen Entwicklung müssen einige Voraussetzungen gelten, die in Abschnitt 4.1 erörtert werden. Danach wird ein Beispielmodell beschrieben, das im weiteren Verlauf dieses Kapitels fortwährend für exemplarische Erläuterungen aufgegriffen wird. Wie auf Basis von Beispielmodellen eine abstrakte Syntax abgeleitet werden kann und welche Schwierigkeiten dabei zu erwarten sind, ist Gegenstand des darauffolgenden Abschnitts 4.3. Das so erzeugte Meta-Modell besitzt in den meisten Fällen Verfeinerungspotential. Diesem Umstand wird dadurch Rechnung getragen, dass für bestimmte Konstellationen von Modellelementen Vorschläge unterbreitet werden, die die Anwendung eines spezifischen Sprachmusters (gegebenenfalls auch mehrerer) empfehlen (Abschnitt 4.4). Im Anschluss daran geht es in Abschnitt 4.5 um die Verarbeitung von frei geänderten Beispielmodellen, die schon über eine abstrakte Syntax verfügen. Derartige Manipulationen sollen ebenfalls (weitgehend) automatisiert auf diese Syntax übertragen werden. Nachdem der Rechner beim Ableiten einer abstrakten Syntax lediglich auf Informationen zurückgreifen kann, die in den Beispielmodellen stecken, wird abschließend darauf eingegangen, wie lückenhaftes Domänenwissen kompensiert und damit das automatisch generierte Resultat verbessert werden kann (Abschnitt 4.6). Der Inhalt der Abschnitte 4.1 sowie 4.3 bis 4.6 bildet den eigenen, vornehmlich konzeptuellen Beitrag innerhalb von Kapitel 4.

4.1 Voraussetzungen

Grundlegend erforderlich ist, dass die **Closed World Assumption** gilt, da sonst basierend auf einer Menge von Beispielmodellen keine Aussagen über ein zugehöriges Meta-Modell getroffen werden können. Dieselbe Annahme setzen auch alle untersuchten Systeme voraus, die der beispielgetriebenen (Meta-)Modellierung angehören (Abschnitt 3.4).

Für die Unterstützung der freien Modellierung ist es wichtig, dass das statisch typisierte **LMM** (Abschnitt 2.2) **um Schemalosigkeit erweitert** wird. Die dafür notwendigen Anpassungen werden anhand des kurzen Beispiels aus Listing 4-1 hergeleitet, da es trotz seines geringen Umfangs bereits offenlegt, welche Informationen allgemein aus linguistischen Beispielmodellen extrahierbar sind. Denn für die freie Modellierung relevant sind ausschließlich Konzepte und deren Instanz-Facette. Das Modell zeigt eine Instanz `Sample` vom Typ `Task`, die über zwei Zuweisungen `title` und `duration` verfügt. Weder der benannte Typ `Task`, noch die durch die beiden Zuweisungen angegebenen Attribute existieren innerhalb des (noch nicht vorhandenen) Meta-Modells. Demzufolge müssen Konzepte einen Typnamen besitzen können, wodurch der Typ der Instanz in Form eines Strings festgelegt ist. Damit ist aber noch keine Instanziierungsbeziehung gegeben. Anders als in [142] beschrieben müssen Zuweisungen ebenfalls über einen Namen verfügen können und brauchen nicht zwangsläufig mit einem Attribut verknüpft sein.

```
1 Task Sample
2   title = "Warten auf Godot"
3   duration = 2.5
```

Listing 4-1: Beispiel zur Demonstration des dynamischen LMM

Um aus einer Menge von Beispielmodellen eine adäquate abstrakte Syntax ableiten zu können, ist es notwendig, folgende heuristische Annahme hinsichtlich dieser Modelle zu treffen: **Namensgleichheit bedeutet auch semantische Äquivalenz**. Je nach Art des Artefakts müssen hierbei noch zusätzliche Randbedingungen beachtet werden.

Nach dem Beispiel aus Listing 4-1 sind die eine Art von Artefakten **Konzepte mit Typnamen** (und damit Instanzen), wobei **kein entsprechendes Meta-Konzept** innerhalb der abstrakten Syntax vorliegt, das von den erstgenannten Konzepten instanziiert wird. Dennoch wird bei zwei oder mehreren Konzepten mit demselben Typnamen angenommen, dass sie implizit dasselbe Meta-Konzept instanziieren, dessen Name dem spezifizierten Typnamen entspricht.

Analoges gilt für **Zuweisungen**, zu denen **kein Attribut** deklariert ist. Deswegen wird auch hier implizit angenommen, dass mehreren gleichnamigen Zuweisungen dasselbe Attribut zugrunde liegt. Diese Hypothese ist ein zentraler Aspekt beim späteren Verfeinern der abstrakten Syntax (Abschnitt 4.4). Ohne ein zugehöriges Attribut können bei derartigen Zuweisungen beliebige Werte auf der rechten Seite angegeben werden. Die literalen Typen der zugewiesenen Werte (also String, Integer, Float, Boolean oder qualifizierender Name) können mithilfe eines regulären Ausdrucks ermittelt werden. Bei qualifizierenden Namen muss weiter differenziert werden, ob es sich um eine Konzeptreferenz, ein Enumerationsliteral oder um einen nicht näher spezifizierbaren Zeiger handelt. Die Vorgehensweise bei Zuweisungen mit gleichem Namen aber unterschiedlicher Typisierung wird in Abschnitt 4.3.1 erläutert.

Diese auf Namensgleichheit beruhende Annahme ist aus Gründen der Eindeutigkeit, die computergestützte Systeme stets erfordern, durchaus akzeptabel. Nachdem somit Mehrdeutigkeiten zu vermeiden sind, trägt Eindeutigkeit auch dazu bei, das Risiko späterer Fehldeutungen zu reduzieren.

4.2 Beispielmodell

Vor der eigentlichen Einführung der einzelnen Algorithmen zur Ableitung einer abstrakten Syntax, wird ein linguistisches Beispielmodell (Listing 4-2) vorgestellt, das in den folgenden Abschnitten für exemplarische Erläuterungen herangezogen wird. Das Modell ist mithilfe der LML frei formuliert (d.h. es gibt

kein zugrundeliegendes Meta-Modell) und repräsentiert einen Prozess zur Planung eines Konferenzbesuchs. Er dient lediglich Demonstrationszwecken und erhebt keinen Anspruch auf inhaltliche Vollständigkeit.

Der Ablauf des Prozesses gestaltet sich wie folgt: Nachdem eine passende Konferenz gesucht und gefunden wurde, muss ein entsprechender Reisekostenantrag eingereicht werden. Erst im Anschluss darf ein Hotel gebucht und danach die An- und Abreise geplant werden. Parallel zu den beiden zuletzt genannten Schritten kann bereits die Registrierung bei der Konferenz erfolgen. Die Nachfolgerbeziehungen sind in Form von `next`-Zuweisungen realisiert. Außerdem verfügt jeder Prozessschritt über einen `title` und kann mit einer Zeitangabe (`duration`) versehen sein. Die einzelnen Schritte werden dahingehend unterschieden, ob sie elektronisch (`ElectronicTask`), in Papierform (`PaperTask`) oder in unbestimmter Form nebenbei (`DetachedTask`) zu erledigen sind. Für die erwähnte parallele Abarbeitung gibt es ein `Split`- und ein `Join`-Element, das den Kontrollfluss einerseits aufspaltet und andererseits wieder zusammenführt. Durch das `And` wird ausgedrückt, dass die Prozessschritte beider Stränge ausgeführt werden müssen und es sich somit nicht um alternative Pfade handelt. Schließlich gibt es noch zwei weitere Elemente, die Start- und Endpunkt des Prozesses festlegen.

```
1  Start S
2    next = Search
3
4  ElectronicTask Search
5    title = "Suche nach passender Konferenz"
6    next = Request
7    duration = 1
8
9  PaperTask Request
10   title = "Einreichung eines Reisekostenantrags"
11   next = Split
12   duration = 0.5
13
14 And Split
15   next = Register, Booking
16
17 ElectronicTask Register
18   title = "Registrierung bei Konferenz"
19   duration = 0.3
20   next = Join
21
22 ElectronicTask Booking
23   title = "Buchung eines Hotels"
24   next = Organize
25
26 ElectronicTask Organize
27   title = "Planung von An- und Abreise"
28   duration = 0.6
29   next = Join
30
31 And Join
32   next = E
33
34 DetachedTask Inform
35   title = "Informieren über Inhalte"
36   duration = 1
37
38 Exit E
```

Listing 4-2: Frei modelliertes Beispielmodell im linguistischen Kontext

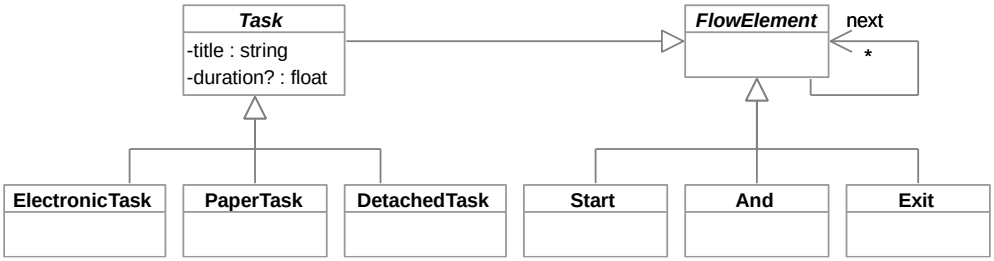

Abbildung 4-1: Gültiges Meta-Modell für linguistischen Beispielprozess

Ein zu diesem Beispielmodell passendes Meta-Modell ist in Abbildung 4-1 visualisiert. Gemessen an den Kriterien von Bertoa und Vallecillo (Abschnitt 2.1.3.2) verfügt es über ein passables Maß an Qualität. Mit `ElectronicTask`, `PaperTask`, `DetachedTask`, `Start`, `And` und `Exit` enthält es genau die Konzepte, die im Beispielmodell Verwendung finden (Vollständigkeit). Dasselbe gilt für die drei Attribute `title`, `duration` und `next`, die aufgrund von Generalisierung genau einmal deklariert werden und damit nicht redundant vorhanden sind (Prägnanz). Ferner ist auch durch die Benennung der Basiskonstrukte `Task` und `FlowElement` deren Intention ersichtlich (Selbstbeschreibung).

Das Meta-Modell räumt dem Modellierer jedoch mehr Freiheiten ein, als im zugrundeliegenden Beispielmodell ausgedrückt wurden. Im Beispiel ist der `DetachedTask Inform` völlig losgelöst vom gesamten Kontrollfluss. Laut Meta-Modell darf er jedoch durch das `next`-Attribut in den Kontrollfluss eingebettet sein und damit sowohl über Vorgänger als auch Nachfolger verfügen. Ähnliches trifft für Instanzen von `Exit` zu. Im Beispielmodell ist die einzige `Exit`-Instanz `E` lediglich Ziel eines Kontrollflusses, hat selbst aber keinen Nachfolger. Das Meta-Modell gestattet es jedoch, dass auch nach solchen Instanzen eine andere Aufgabe folgen kann. Ein Vorteil der Entwurfsentscheidung, dass alle Entitäten eines Prozesses Flusselemente darstellen, ist die Möglichkeit, alle Entitäten auf dieselbe Weise verarbeiten zu können (z.B. bei der Anbindung einer ontologischen konkreten Syntax wie in Abschnitt 5.2 beschrieben). Der dadurch eingebrachte Modellierungsspielraum kann allerdings durch die Bereitstellung externer Constraints eingeschränkt werden. Dadurch ist beispielsweise gezielt zu erreichen, dass Instanzen von `Exit` keine Nachfolger besitzen dürfen.

Derartige **Annahmen** hinsichtlich einem höheren Maß an Flexibilität können nicht dem Beispielmodell entnommen werden. Sie **erfordern tiefgreifendes Wissen über die jeweilige Domäne** und auch darüber, wie Modelle weiterverarbeitet werden. Demzufolge kann das Meta-Modell, wie es Abbildung 4-1 zeigt, nicht automatisch generiert werden, sondern lediglich bis zu einem bestimmten Punkt angenähert werden. Für eine weitere Verfeinerung können dem Benutzer allerdings Vorschläge unterbreitet werden, welche Stellen im Meta-Modell er näher betrachten sollte, weil dort Indizien vorliegen, die z.B. für die Anwendung eines bestimmten Sprachmusters sprechen. Nähere Details dazu finden sich im Anschluss an die Ableitung einer initialen abstrakten Syntax in Abschnitt 4.4.

4.3 Ableiten einer initialen abstrakten Syntax

In diesem Abschnitt wird die Generierung eines initialen Meta-Modells basierend auf einer Menge von Beispielmodellen vorgestellt. Das resultierende Meta-Modell entspricht der abstrakten Syntax einer bis dato noch nicht definierten DSML. Die konkrete Syntax hingegen ist durch das LML bereits fest vorgegeben. Der entsprechende, auch als **Bottom-Up** bezeichnete **Algorithmus** wurde bereits in [120] veröffentlicht und adressiert Anforderung 2. Er weist gewisse Parallelen zu den in [87, 124] geschilderten Verfahren auf, geht allerdings detaillierter auf die potentiell auftretenden Probleme sowie mögliche

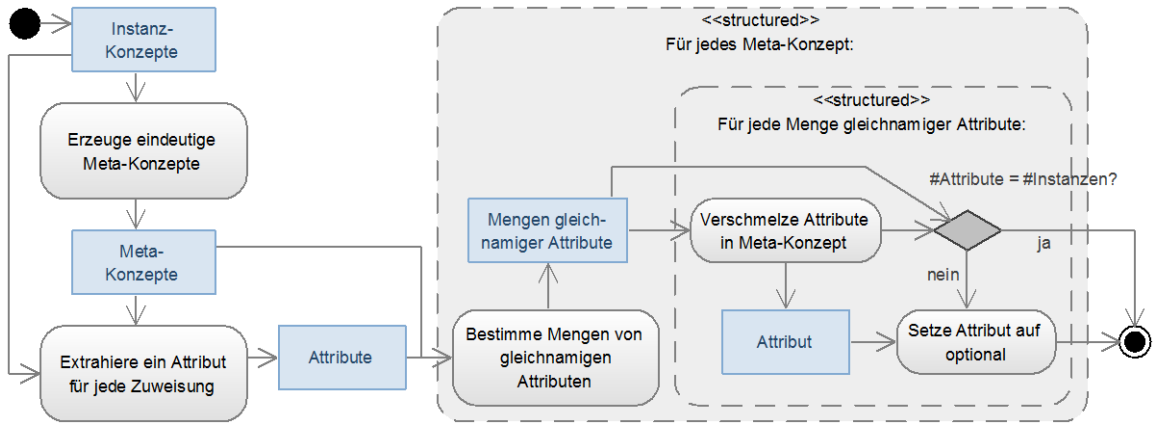

Abbildung 4-2: Aktivitätsdiagramm des initialen Bottom-Up Algorithmus

Lösungen ein. Nicht berücksichtigt wird in [87, 124] beispielsweise der Umgang mit Mehrfachvererbung, wie sie explizit in Abschnitt 4.3.2 adressiert wird.

Als **Input** des Algorithmus (Abbildung 4-2) dienen **sämtliche Instanz-Konzepte** der Beispielmodelle. Für jeden eindeutigen Typnamen wird zunächst ein separates Meta-Konzept generiert. Danach wird für jede Zuweisung ein entsprechendes Attribut erzeugt, ohne dass dieses einem der zuvor generierten Konzepte zugeteilt wird. Hierbei kann bereits die obere Multiplizitätsgrenze des Attributs bestimmt werden. Sie wird auf 1 gesetzt, wenn nur ein Wert zugewiesen wird, andernfalls auf *. Zur Ermittlung des Typs finden reguläre Ausdrücke Verwendung. Bei Werten, die den literalen Datentypen Boolean, Integer, Float und String entsprechen, ist das Resultat stets eindeutig. Wird als Wert jedoch lediglich ein Name angegeben, so ist eine weitere Differenzierung notwendig, da der Wert entweder ein anderes Instanz-Konzept ist oder einen Pointer repräsentiert. Wird eine Instanz mit dem Namen gefunden, der dem zugewiesenen Wert entspricht, dann wird als Attributtyp das zur Instanz gehörende Meta-Konzept gesetzt. Andernfalls wird der Typ des Attributs als Pointer deklariert.

Für jedes Meta-Konzept werden anschließend **Mengen gleichnamiger Attribute berechnet**, die als Grundlage für die eigentliche Attributdeklaration im jeweiligen Meta-Konzept dienen. Welche Attribute zu welchem Meta-Konzept gehören, kann unter Berücksichtigung der zugrundeliegenden Instanz-Konzepte festgestellt werden.

Für das Beispiel aus Listing 4-2 zeigt Tabelle 4-1 die abgeleiteten Meta-Konzepte sowie die zugehörigen Mengen gleichnamiger Attribute. Zur besseren Nachvollziehbarkeit enthält die Tabelle zusätzlich die zugrundeliegenden Instanz-Konzepte inklusive der aus den jeweiligen Zuweisungen generierten Attribute. Am deutlichsten nachvollziehbar ist die Struktur anhand des Meta-Konzepts `ElectronicTask`. Zu ihm gibt es die vier Instanz-Konzepte `Search`, `Register`, `Booking` und `Organize`. Mit Ausnahme von `Booking` verfügt jedes dieser Konzepte über Zuweisungen an Attribute mit den Namen `title`, `duration` und `next`. `Booking` hingegen besitzt nur Zuweisungen an die Attribute `title` und `next`. Bei der Zusammenfassung aller Attribute der besagten vier Instanz-Konzepte resultieren drei Mengen gleichnamiger Attribute, und zwar je eine Menge für `title`, `duration` und `next`. Da `title` und `next` in allen vier Instanz-Konzepten vorkommen bestehen die jeweiligen Attributmengen aus vier Attributen, während die `duration`-Menge lediglich drei Einträge umfasst.

Nach der Bestimmung der Attributmengen werden **alle Attribute pro Menge zu einem einzigen Attribut verschmolzen** und dieses dem jeweiligen Meta-Konzept hinzugefügt. Das Verschmelzen von Attributen ist eine nicht-triviale Operation, weshalb darauf in Unterabschnitt 4.3.1 näher eingegangen wird.

Tabelle 4-1: Aus Beispielmodell initial abgeleitete Meta-Konzepte und zugehörige Attributmengen

Meta-Konzepte	Instanz-Konzepte	Attribute	Attributmengen
Start	S	next: ElectronicTask	{ next: ElectronicTask }
ElectronicTask	Search	title: string duration: integer next: PaperTask	{ title: string, title: string, title: string }
	Register	title: string duration: float next: And	{ duration: integer, duration: float, duration: float }
	Booking	title: string next: ElectronicTask	{ next: PaperTask, next: And, next: ElectronicTask, next: And }
	Organize	title: string duration: float next: And	
PaperTask	Request	title: string duration: float next: And	{ title: string } { duration: float } { next: And }
And	Split	next[]: ElectronicTask	{ next[]: ElectronicTask, next: Exit }
	Join	next: Exit	
DetachedTask	Inform	title: string duration: integer	{ title: string } { duraton: integer }
Exit	E		

Abschließend wird überprüft, ob die **Anzahl der Attribute** der ursprünglichen Menge **mit der Anzahl der Instanzen** des jeweiligen Meta-Konzepts **übereinstimmt**. Ist dies der Fall, terminiert der Algorithmus. Andernfalls ist die Attributanzahl kleiner als die Anzahl der Instanz-Konzepte, so dass das Attribut als optional deklariert wird.

Der **Algorithmus ist auf eine beliebige Menge von Instanz-Konzepten anwendbar**, sofern diese innerhalb von Modellen liegen, die mithilfe des LMM formuliert sind. Einschränkungen gibt es nach unseren Erkenntnissen keine, wie zahlreiche Beispielszenarien aus diversen Anwendungsdomänen gezeigt haben.

4.3.1 Verschmelzen von Attributen

Das Verschmelzen mehrerer gleichnamiger Attribute ist die zentrale Aktion beim Ableiten einer abstrakten Syntax, weil hierbei die Informationen und Constraints aus verschiedenen Attributen in einem Attribut zusammengefasst werden. Es erfolgt damit eine **Konsolidierung** des aus den Beispielmodellen gewonnenen **Domänenwissens**. Bei der Attributverschmelzung gilt es die Teile der Attribute zu beachten, die während der initialen Erzeugung explizit gesetzt werden. Dabei handelt es sich um den Namen, den Typ sowie die Multiplizität. Da alle Attribute in der Ausgangsmenge denselben Namen aufweisen, wird dieser Name auch für das resultierende Attribut übernommen.

Bei der **Multiplizität** müssen im Zuge der Verschmelzung hingegen zwei unterschiedliche Werte beachtet werden, nämlich 1 und 1..*. Die Multiplizität wird bei den initial erzeugten Attributen auf 1 gesetzt, sofern die zugrundeliegende Zuweisung genau einen Wert umfasst. Sind mehr als ein Wert auf der rechten Seite angegeben, so wird die Multiplizität mit 1..* festgelegt. Demnach kann während

der Attributverschmelzung ausschließlich die obere Grenze der Multiplizität bestimmt werden. Hierbei wird stets der maximale Wertebereich übernommen, d.h. 1..* wird 1 gegenüber bevorzugt. Angewendet auf das Beispiel aus Tabelle 4-1 bedeutet dies für die `next`-Attributmenge von Meta-Konzept `And`, dass als Multiplizität 1..* resultiert. Die untere Grenze wird erst in einem sich anschließenden Schritt adressiert. Sie wird nur dann auf 0 gesetzt, wenn es mehr Instanzen zum aktuell verarbeiteten Meta-Konzept gibt als Attribute in der momentan behandelten Attributmenge (siehe dazu den Entscheidungsknoten sowie dessen Nachfolger in Abbildung 4-2). Dann nämlich können auch Instanzen existieren, die nicht über eine Zuweisung an das aktuelle Attribut verfügen. Ein Beispiel ist das `duration`-Attribut von `ElectronicTask` in Tabelle 4-1, da es lediglich in drei von vier Instanzen vorkommt.

Das **Zusammenführen verschiedener Attributtypen** ist weitaus komplexer. Unterschieden wird zwischen literalen und referentiellen Attributen. Zu den literalen Attributen zählen im LMM auch Attribute mit Pointer als Datentyp. Werden zwei oder mehr Attribute mit unterschiedlichen literalen Typen gefunden, so findet eine automatische Typkonvertierung statt, die mit derjenigen von dynamischen Programmiersprachen (z.B. JavaScript [39]) vergleichbar ist. Dabei wird stets der Typ mit dem größten Wertebereich übernommen, was zur Folge hat, dass die zugewiesenen Werte aus einem kleineren Wertebereich entsprechend konvertiert werden müssen.

Die Titelzeile von Tabelle 4-2 listet alle unterstützten literalen Datentypen auf, wobei der Wertebereich von links nach rechts zunimmt. Überdies enthält die Tabelle Beispiele für die **Konvertierung** verschiedener **literaler Werte** von einem Typ mit kleinem Wertebereich hin zu einem Typ mit größerem Wertebereich. Ein Sonderfall liegt vor, sobald der Datentyp Pointer im Kontext einer automatischen Typkonvertierung auftritt. Dieser kann ausschließlich in einen String umgewandelt werden. Eine Kompatibilität mit anderen Datentypen ist nicht gegeben, was bei einem entsprechenden Szenario zum Abbruch des Ableitungsprozesses führt.

Tabelle 4-2: Literale Datentypen mit Beispielen für die Konvertierung von Werten

	Boolean	**Integer**	**Float**	**Pointer**	**String**
Boolean	`false / true`	`0 / 1`	`0 / 1`	-	`"false" / "true"`
Integer	-	`3 / -2`	`3 / -2`	-	`"3" / "-2"`
Float	-	-	`0.5 / -3e6`	-	`"0.5" / "-3e6"`
Pointer	-	-	-	`X1 / A.B.c`	`"X1" / "A.B.c"`

In Tabelle 4-1 ist eine Typkonvertierung für das `duration`-Attribut von `ElectronicTask` erforderlich. Denn dieses Attribut ist einmal als Integer und in zwei Fällen als Float deklariert. Der resultierende Typ ist wegen des größeren Wertebereichs Float.

Weisen zwei oder mehr zu verschmelzende **Attribute unterschiedliche Meta-Konzepte als Typ** auf, so muss ebenfalls ein gemeinsames Meta-Konzept zur Typisierung des konsolidierten Attributs bestimmt werden. Dieser als **Liskovsches Substitutionsprinzip** bezeichneter Anwendungsfall ist typisch für das Sprachmuster „Generalisierung" [85]. Ist bereits ein gemeinsames Basiskonzept vorhanden, so wird es als Typ des resultierenden Attributs festgesetzt. Andernfalls muss ein derartiges Basiskonzept zuerst eingeführt und kann dann als Attributtyp verwendet werden.

Bezogen auf Tabelle 4-1 betrifft dies die `next`-Attributmenge von `ElectronicTask` sowie die `next`-Attributmenge von `And`. Demzufolge muss sowohl für `ElectronicTask`, `PaperTask`

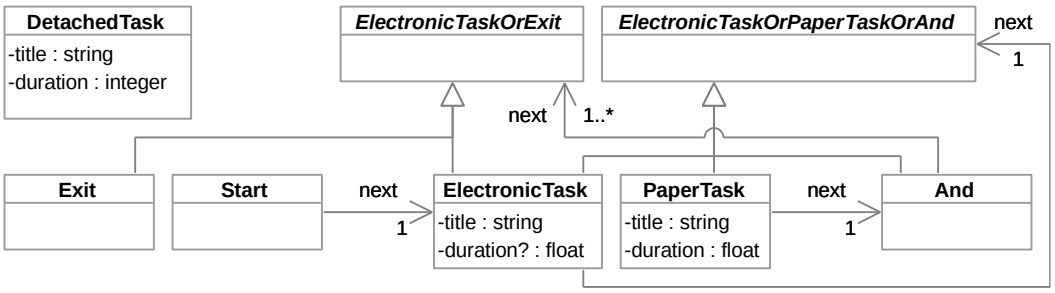

Abbildung 4-3: Initial abgeleitetes Meta-Modell für linguistischen Beispielprozess

und `And` als auch für `ElectronicTask` und `Exit` ein eigenes Basiskonzept angelegt werden. Diese Basiskonzepte sind in Abbildung 4-3 als `ElectronicTaskOrPaperTaskOrAnd` und als `ElectronicTaskOrExit` dargestellt.

Die **automatische Namensgebung** erfolgt durch die Konkatenation der Namen der einzelnen Ausgangskonzepte, wobei zwischen zwei Namen stets „Or" eingefügt wird. Nachdem das Diagramm das initial abgeleitete Meta-Modell für den Beispielprozess aus Listing 4-2 zeigt, sind darin bereits alle Attributmengen verschmolzen und dem jeweiligen Meta-Konzept hinzugefügt. Ein Fragezeichen hinter dem Namen eines literalen Attributs kennzeichnet dieses als optional (siehe `duration`-Attribut).

Treten gleichnamige Attribute auf, die sowohl literaler als auch referentieller Natur sind, wird der Meta-Modell Generierungsprozess abgebrochen und ein entsprechender Fehler gemeldet. Grund dafür ist, dass literale Werte nicht in Referenzen (und umgekehrt) konvertiert werden können. Prinzipiell wäre es zwar für Strings und Referenzen möglich, doch ist die Intention von Referenzen und Werten gänzlich verschieden.

4.3.2 Eliminierung von Mehrfachvererbung

Wie aus Abbildung 4-3 ersichtlich, kann die oben vorgestellte Vorgehensweise zur Einführung von Mehrfachvererbung führen, was in manchen Fällen unerwünscht ist, da dieses Sprachmuster potentielle Gefahren in sich birgt (z.B. Namenskollisionen) [129]. Deshalb kann durch Konfiguration dem initialen Ableitungsprozess eine zusätzliche Aktion nachgeschaltet werden, die Mehrfachvererbung aus dem generierten Meta-Modell entfernt. Um die Komplexität des Meta-Modells nicht künstlich zu steigern, wird **Mehrfachvererbung durch das Sprachmuster „Einfachvererbung" ersetzt**.

Die Überführungsstrategie startet mit der **Suche nach Konzeptverbünden mit Mehrfachvererbung**. Für jeden derartigen Verbund werden alle Basiskonzepte ermittelt und zu einem gemeinsamen Basiskonzept zusammengeführt (umsetzbar z.B. mithilfe des „Merge Classes"-Operator [62]). Die Namensbildung folgt dem oben beschriebenen Schema, d.h. Konkatenation der Namen mit verbindendem „Or". Um die Länge des Namens etwas zu beschränken, werden gemeinsame Teil-Strings nur einmal aufgeführt.

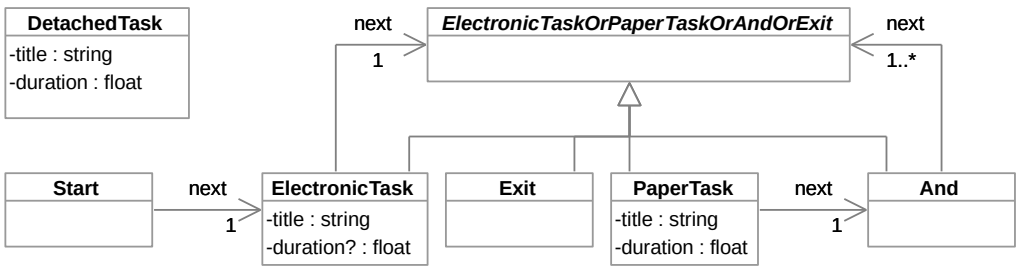

Abbildung 4-4: Initial abgeleitetes Meta-Modell mit Einfachvererbung für linguistischen Beispielprozess

Abbildung 4-4 zeigt die entsprechend modifizierte Variante des Meta-Modells aus Abbildung 4-3. Der Verbund mit Mehrfachvererbung besteht anfänglich aus `ElectronicTask`, `Exit`, `PaperTask`, `And`, `ElectronicTaskOrExit` und `ElectronicTaskOrPaperTaskOrAnd`. Die zwei zuletzt genannten Meta-Konzepte sind die Basistypen, die in Abbildung 4-4 zu `ElectronicTaskOrPaperTaskOrAndOrExit` zusammengeführt werden. Der Teil-String „ElectronicTaskOr" käme bei einer einfachen Konkatenation der Konzeptnamen doppelt vor und wird daher beim Anhängen des zweiten Namens ignoriert.

Da Basiskonzepte durch nachträgliche manuelle Änderungen ebenfalls Attribute besitzen können, ist ein Namenskonflikt nicht auszuschließen. Diese Attribute müssen dann analog zu dem in Abschnitt 4.3.1 erläuterten Verfahren verschmolzen werden. Nachdem das erneut zu mehr als einem Basiskonzept pro Meta-Konzept führen kann, muss die neu eingeführte Mehrfachvererbung wiederum eliminiert werden. Dieser **Zyklus** terminiert jedoch spätestens dann, wenn ein globales Basiskonzept gefunden wurde, das als Generalisierung für alle anderen Meta-Konzepte dient.

Alternativ zu obiger Strategie kann anstelle des Zusammenführens von Basiskonzepten die **Vererbungshierarchie erweitert** und dadurch ein Oberkonzept für die bisherigen Basiskonzepte eingeführt werden. Bei einem großen Konzeptverbund mit zahlreichen Basiskonzepten kann dies zu einer komplexen Vererbungshierarchie führen. Dies wiederum kann aufgrund der großen Anzahl zusätzlicher Konzepte die Verständlichkeit und somit die Qualität des Meta-Modells beeinträchtigen (Prägnanz-Kriterium, Abschnitt 2.1.3.2). Die Komplexität des Meta-Modells wird mit erstgenannter Lösung dagegen nur geringfügig erhöht, weshalb jene präferiert wird. Durch Konfiguration kann das Standardverhalten jedoch auf die besagte Alternative umgestellt oder die automatisierte Verschmelzung gänzlich deaktiviert werden.

4.4 Verfeinerung der abstrakten Syntax

Betrachtet man das initial abgeleitete Meta-Modell aus Abbildung 4-4, so sind zwar gewisse Parallelen zu der erwarteten Variante aus Abbildung 4-1 erkennbar, doch enthält das automatisiert generierte Meta-Modell zahlreiche Redundanzen, die die Verständlichkeit beeinträchtigen (z.B. werden die Attribute `title` und `duration` insgesamt drei Mal deklariert). Zudem enthält die erwartete Variante bereits ergänztes Domänenwissen, das im generierten Resultat fehlt. Ein Beispiel ist das Konzept `Task`, das als Generalisierung angibt, welche Arten von Informationen sämtliche Aufgaben beinhalten müssen bzw. können. Im konkreten Fall sind das einerseits der Titel einer Aufgabe und andererseits eine Zeitangabe, wie lange die Durchführung der betreffenden Aufgabe in etwa dauert. Daraus erwächst das Bedürfnis das generierte Resultat derart umzubauen, dass es weitgehend dem erwarteten Modell entspricht.

Nachdem abgeleitete Meta-Modelle erheblich größer sein können als dies im obigen Beispiel der Fall ist, ist es gemäß Anforderung 3 wünschenswert den Modellierungsexperten, der später das Meta-Modell weiterverarbeitet, **auf Stellen mit Optimierungspotential hinzuweisen**. Im vorliegenden Fall handelt es sich um die Beachtung des in Abschnitt 2.1.3.2 beschriebenen Prägnanz-Kriteriums. Dieselbe Strategie verfolgen auch López-Fernández et al. [87], um die Qualität des generierten Meta-Modells zu verbessern. Dass Sprachmuster zur Steigerung der Qualität von Meta-Modellen dienen können, erläutern Henderson-Sellers und Gonzalez-Perez in [61] sowie Neumayr et al. in [103].

Als Heuristik greifen wir hierzu erneut auf das **Prinzip gleichbedeutender Attribute** zurück, da laut der Annahme aus Abschnitt 4.1 sich hinter gleichnamigen Attributen mit großer Wahrscheinlichkeit dieselbe semantische Eigenschaft verbirgt. Dieses Prinzip wird nun **großflächig angewendet**, indem in einem gegebenen Meta-Modell nach Mengen von Konzepten gesucht wird, die möglichst viele

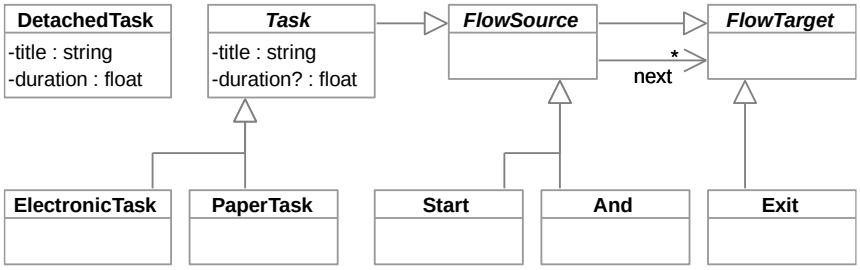

Abbildung 4-5: Alternatives, ebenfalls mit Einfachvererbung verfeinertes Meta-Modell für linguistischen Beispielprozess

gleichnamige Attribute deklarieren. Solche Mengen repräsentieren dann Kandidaten für die Einführung verallgemeinernder Sprachmuster. Das LMM (Abschnitt 2.2) unterstützt diesbezüglich die beiden Muster „Einfachvererbung" und „erweiterter Powertyp". Letzteres Muster dient vornehmlich zur Vermeidung komplexer Vererbungshierarchien und insbesondere der Vermeidung von Mehrfachvererbung [142]. Die Auswirkungen des erweiterten Powertyps sind demnach umfangreicher als die der reinen Einfachvererbung. Deshalb sind Vorschläge für jedes der beiden Sprachmuster auf unterschiedlichem Weg zu berechnen, was in den folgenden Unterabschnitten erläutert wird.

Die Anwendung von Sprachmustern kann schließlich manuell erfolgen oder unter Einsatz des von Jahn in seiner Dissertation [66] entwickelten Frameworks zur benutzerorientierten Evolution von Meta-Modellen. Darin werden unter anderem Strategien mit Werkzeugunterstützung vorgestellt, die die Einführung von Einfachvererbung und des erweiterten Powertypmusters stark vereinfachen. Eine **vollständig automatisierte Einführung von Sprachmustern ist aufgrund fehlenden Domänenwissens nicht hilfreich** und würde in den seltensten Fällen zu den erwarteten Ergebnissen führen. Verdeutlicht wird das durch die in Abbildung 4-1, Abbildung 4-5 und Abbildung 4-6 dargestellten Meta-Modelle, die alle drei valide hinsichtlich des Beispielprozesses aus Listing 4-2 sind, aber unterschiedlichen Ansprüchen für die spätere Verarbeitung genügen. Es sind noch zahlreiche weitere Alternativen denkbar, die die genannte Validitätsbedingung erfüllen. Welche Entwurfsentscheidungen man trifft, hängt jedoch stark vom jeweiligen Einsatzzweck und mithin von der betreffenden Domäne ab.

Eine andere Art der Vereinheitlichung kann auch mit dem Sprachmuster **Enumeration** erreicht werden. Sie bezieht sich jedoch nicht auf Konzepte, sondern auf einen literalen Datentyp mit eingeschränktem Wertebereich. Die Grundlage bilden auch hier gleichnamige Attribute. Wann ein Vorschlag für die Einführung einer Enumeration unterbreitet wird, kann Abschnitt 4.4.3 entnommen werden.

Das LMM unterstützt noch weitere Sprachmuster (z.B. Deep Instantiation und Materialisierung), doch diese sind nur für Meta-Ebenenhierarchien mit mehr als zwei Ebenen von Relevanz. Im Rahmen dieser

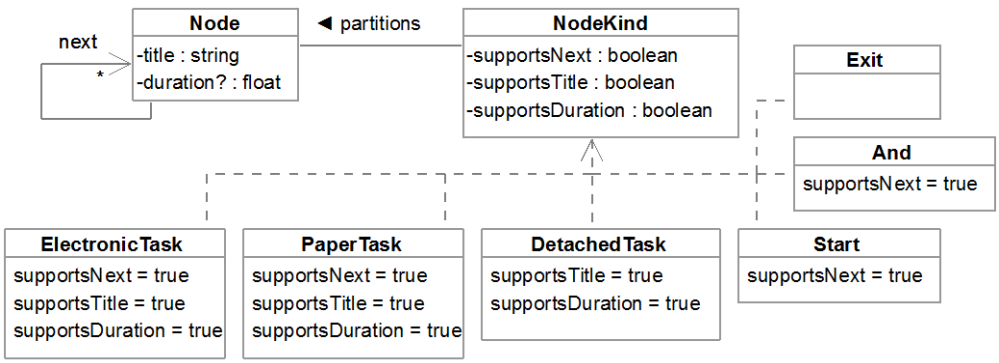

Abbildung 4-6: Alternatives, mit einem erweiterten Powertyp verfeinertes Meta-Modell für linguistischen Beispielprozess

Arbeit beschränken wir uns jedoch auf eine Modell- und eine Sprachebene, weshalb auf diese Muster im Folgenden nicht für die Unterbreitung entsprechender Verfeinerungsvorschläge eingegangen wird.

Im verwandten MLCBD-Ansatz (Abschnitt 3.4.3) findet ebenfalls eine Verfeinerung des Meta-Modells statt. Hierbei kommen anstelle von Sprachmustern Entwurfsmuster zum Einsatz. Allerdings beschränkt sich dieser Ansatz auf „demonstrierte" grafische DSMLs und somit auf den in Abschnitt 6 beschriebenen ontologischen Kontext. So diskutieren Cho und Gray in [22] **Entwurfsmuster**, die insbesondere **für grafische Sprachen** sinnvoll sind und deshalb automatisch in ein Meta-Modell, das die abstrakte Syntax einer grafischen DSML definiert, eingebaut werden können. Dies ist möglich, da der Benutzer die konkrete Syntax ebenfalls neu festlegt. Aus ihr können dann Annahmen über die betreffende Domäne abgeleitet werden (z.B. durch Verschachtelung oder Verbindung von Elementen), die im linguistischen Kontext nicht verfügbar sind.

Die in den kommenden Unterabschnitten präsentierten **Algorithmen** sind **allgemeingültig für Meta-Modelle anwendbar**, die mithilfe des LMM formuliert sind. Einschränkungen gibt es nach unseren Erkenntnissen keine, wie zahlreiche Beispielszenarien mit mehreren Dutzend Meta-Konzepten und Attributen gezeigt haben.

4.4.1 Einfachvererbung als Verfeinerungsvorschlag

Die einfachste Form eines Musters zur Umsetzung von Verallgemeinerung ist die Einfachvererbung. Um einen diesbezüglichen Verfeinerungsvorschlag zu unterbreiten, kann – wie anfangs im übergeordneten Abschnitt 4.4 erwähnt – lediglich auf gleichbedeutende Attribute zurückgegriffen werden. Attribute, die dieselbe Bedeutung haben und somit einander entsprechen, werden im Folgenden als **korrespondierende Attribute** betitelt. Zwei Attribute korrespondieren, wenn sie mindestens im Namen und in der Attributart (also literal oder referentiell) übereinstimmen. Je nach Konfiguration werden auch Typ und Multiplizität der zu vergleichenden Attribute beachtet (weitere Details dazu folgen in Abschnitt 4.6.2). Im Gegensatz zu den oben vorgestellten gleichnamigen Attributen sind korrespondierende Attribute in unterschiedlichen Konzepten deklariert.

Die beschriebene **Korrespondenz ist eine Äquivalenzrelation**, da sie reflexiv (jedes Attribut korrespondiert mit sich selbst), symmetrisch (wenn Attribut *a* mit Attribut *b* korrespondiert, dann korrespondiert auch *b* mit *a*) und transitiv (wenn Attribut *a* mit Attribut *b* und Attribut *b* mit Attribut *c* korrespondieren, dann korrespondiert auch *a* mit *c*) ist. Es ist also zweckmäßig korrespondierende Attribute als Mengen darzustellen, da keine Aussage über die Reihenfolge der Attribute hinsichtlich der Korrespondenzrelation getroffen werden kann. Bezogen auf das Meta-Modell aus Abbildung 4-4 ergeben sich folgende Attributmengen, falls neben dem Attributnamen keine weiteren Informationen auf Gleichheit geprüft werden:

- `{ DetachedTask.title: string, ElectronicTask.title: string, PaperTask.title: string }`
- `{ DetachedTask.duration: float, ElectronicTask.duration?: boolean, PaperTask.duration: boolean }`
- `{ Start.next: ElectronicTask, ElectronicTask.next[]: ElectronicTaskOrPaperTaskOrAndOrExit, PaperTask.next: And, And.next: ElectronicTaskOrPaperTaskOrAndOrExit }`

Wird zusätzlich die Multiplizität beachtet, so entfallen jeweils die Vertreter von `ElectronicTask` in der `duration`- und in der `next`-Attributmenge. Die Dauer ist hier nämlich als optional angegeben, während sie in den anderen Fällen verpflichtend ist. Beim `next`-Attribut dagegen dürfen mehrere

Nachfolger zugewiesen werden, was im Unterschied zu den anderen Konzepten eine höhere Kardinalität bedeutet.

Die Aussage hinter einer Menge korrespondierender Attribute ist, dass die deklarierenden Konzepte der in der Menge enthaltenen Attribute diese Attribute als genau eine Korrespondenz (Gemeinsamkeit) aufweisen. Bei der ersten aufgeführten Menge bilden die drei `title`-Attribute eine Korrespondenz der Konzepte `DetachedTask`, `ElectronicTask` und `PaperTask`. Gleiches gilt für die Menge der drei `duration`-Attribute. Damit besitzen `DetachedTask`, `ElectronicTask` und `PaperTask` genau zwei Gemeinsamkeiten, die durch zwei Mengen korrespondierender Attribute gegeben sind.

Der Sachverhalt von **Attributmengen mit denselben Korrespondenzen** lässt sich generalisieren. Sind zwei Mengen korrespondierender Attribute gleich groß und stimmen die deklarierenden Konzepte dieser Attribute im Vergleich zwischen den Mengen überein, dann wird von einer Abhängigkeit zwischen diesen Attributmengen hinsichtlich der gemeinsamen Elternkonzepte gesprochen. Diese Abhängigkeitsbeziehung bildet analog zur Korrespondenz von Attributen eine Äquivalenzrelation.

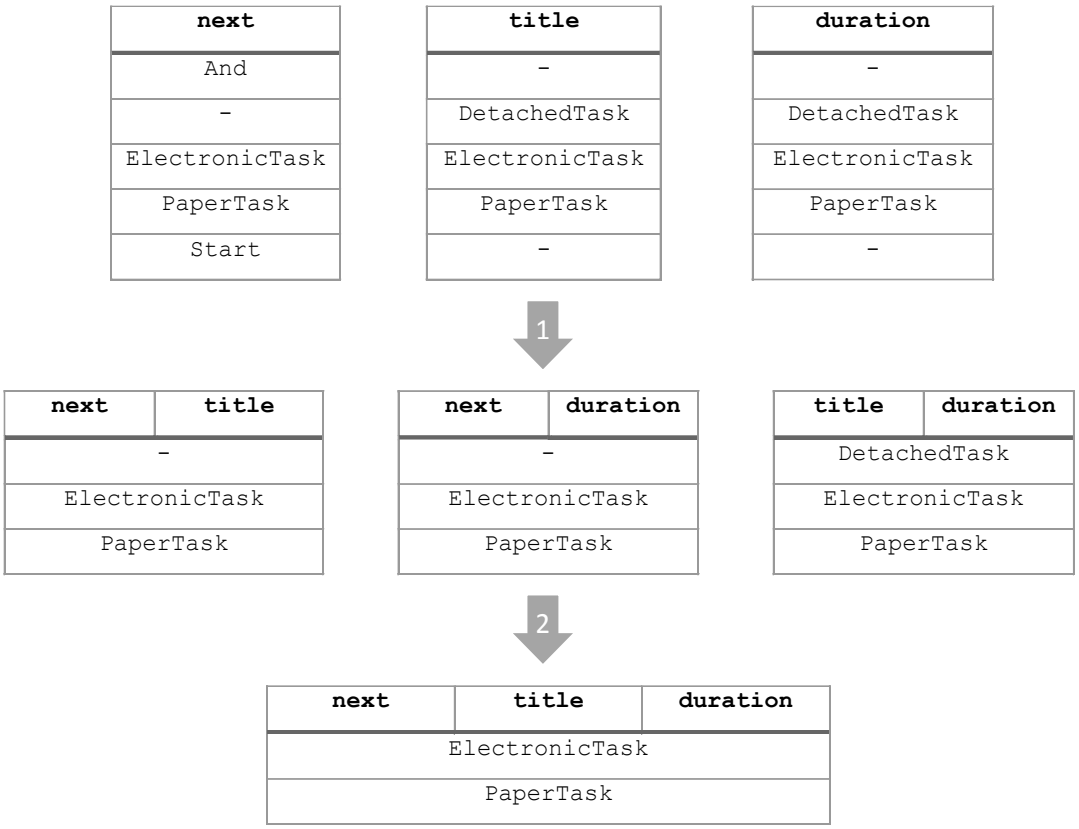

Abbildung 4-7: Beispiel für die Bestimmung abhängiger Mengen korrespondierender Attribute

Die Visualisierung dieses Sachverhalts kann in Tabellenform erfolgen, wie in Abbildung 4-7 geschehen. Dabei enthält die erste Tabellenzeile für jede Menge korrespondierender Attribute genau einen Eintrag mit dem gemeinsamen Namen der jeweils beinhalteten Attribute. In den Zeilen darunter hingegen sind all diejenigen Konzepte aufgelistet, die basierend auf den in der Titelzeile zusammengefassten Attributmengen voneinander abhängig sind. Diejenigen Zellen mit einem „-“ als Inhalt dienen lediglich

einer besseren visuellen Vergleichbarkeit und besitzen keine inhaltliche Bedeutung. Jede dieser Tabellen repräsentiert einen Kandidaten für die Anwendung eines verallgemeinernden Sprachmusters und dient damit zur Verfeinerung des Meta-Modells in Bezug auf die in der Tabelle aufgelisteten Konzepte und Attribute.

In einem Meta-Modell können in der Regel viele verschiedene solcher Kandidaten ausfindig gemacht werden. Deshalb ist es wichtig, die ermittelten **Kandidaten** zu **gewichten** und dem Benutzer – absteigend sortiert nach dieser Gewichtung – für die Verfeinerung vorzuschlagen. Die Gewichtung erfolgt hierbei anhand der Anzahl abhängiger Mengen korrespondierender Attribute. Ein Kandidat ist daher besser als ein anderer, wenn er über zahlenmäßig mehr solcher Attributmengen verfügt. Danach erst ist die Anzahl der deklarierenden Konzepte von Relevanz, die als Sekundärfaktor in die Gewichtung einfließt. Begründen lässt sich das damit, dass eine in der Realität tatsächlich vorhandene Gemeinsamkeit wahrscheinlicher ist, wenn zwei oder mehr Konzepte über eine möglichst große Anzahl an Überschneidungen (korrespondierende Attribute) verfügen. In Abbildung 4-7 ist dies für die unterste Tabelle der Fall, weil sie alle drei Attributmengen `next`, `title` und `duration` als Korrespondenzen der Konzepte `ElectronicTask` und `PaperTask` angibt.

Der **Algorithmus zur Bestimmung sämtlicher Kandidaten**, für die eine Verfeinerung in Frage kommt, ist in Abbildung 4-8 in Form eines Aktivitätsdiagramms visualisiert. Er beginnt mit der Suche nach korrespondierenden Attributen in einer gegebenen Menge von Ebenen. Die genauen Kriterien, ob zwei Attribute korrespondieren, werden extern durch eine vom Benutzer anpassbare Konfiguration festgelegt (Abschnitt 4.6.2).

Die gefundenen **Mengen korrespondierender Attribute** werden dann in eine Datenstruktur überführt, die **als Abhängigkeitstupel** bezeichnet wird und deren Inhalt den Tabellen aus Abbildung 4-7 gleichkommt. Grund dafür ist lediglich die Verwendung einer einheitlichen Datenstruktur für den weiteren Verlauf des Algorithmus. Der erste Eintrag eines derartigen Tupels umfasst die abhängigen Mengen korrespondierender Attribute und entspricht damit der ersten Zeile in den Tabellen des Beispiels. Der zweite Eintrag hingegen enthält diejenigen Konzepte, die zu jeder Menge des ersten Eintrags genau ein deklariertes Attribut besitzen. Diese Konzepte befinden sich in den weiteren Zeilen der Beispieltabellen. Die oben beispielhaft aufgelisteten Mengen korrespondierender Attribute sind äquivalent zu den drei ersten Tabellen aus Abbildung 4-7, die als Abhängigkeitstupel vorliegen.

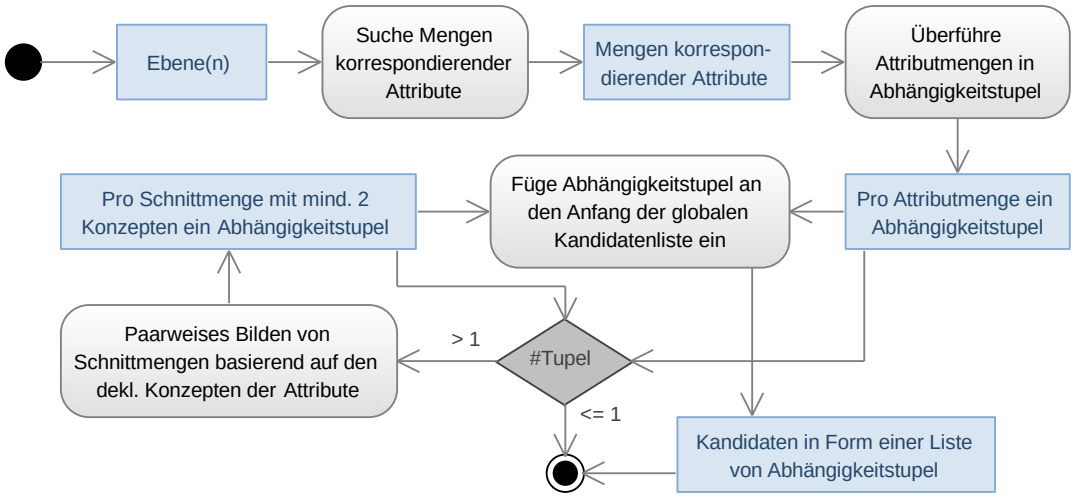

Abbildung 4-8: Aktivitätsdiagramm zur Berechnung von Kandidaten für die Anwendung von Einfachvererbung

Im nächsten Schritt werden die initial erzeugten Abhängigkeitstupel an den Anfang der Ergebnisliste eingefügt, die beim späteren Terminieren des Algorithmus alle Kandidaten umfasst, die während der Ausführung identifiziert wurden. Gleichzeitig wird darauf geachtet, dass Kandidaten mit zahlenmäßig mehr Konzepten weiter vorne in der Liste eingeordnet werden als diejenigen mit weniger Konzepten. Auf diese Weise ist sichergestellt, dass die resultierende **Kandidatenliste absteigend sortiert** ist. Der erste Eintrag ist somit stets der Kandidat mit der momentan größten Wahrscheinlichkeit für eine in der realen Welt tatsächlich auftretende Gemeinsamkeit.

Liegen mindestens zwei Abhängigkeitstupel vor, dann erfolgt eine paarweise Kombination durch **Bildung von Schnittmengen der deklarierenden Konzepte**. Dabei werden nur für diejenigen Schnittmengen neue Abhängigkeitstupel erzeugt, die mindestens zwei Konzepte enthalten, weil bei weniger als zwei Konzepten keine Abhängigkeit vorliegt. Der Kombinationsschritt wird so lange wiederholt, bis maximal ein Tupel übrig ist. Danach endet der Algorithmus und er liefert eine nach der oben beschriebenen Gewichtung geordnete Liste von Verfeinerungskandidaten als Resultat.

Angewendet auf das Beispiel aus Abbildung 4-7 sieht die Liste wie folgt aus, wobei aus Gründen der Übersichtlichkeit in der verwendeten Darstellung ausschließlich die Namen der korrespondierenden Attribute angegeben sind: `({next, title, duration}, {title, duration}, {next, title}, {next, duration}, {next}, {title}, {duration})`.

Damit erhält ein Modellierungsexperte als ersten Vorschlag ein gemeinsames Basiskonzept für die Konzepte `PaperTask` und `ElectronicTask` einzuführen, das die drei korrespondierenden Attribute `next`, `title` und `duration` deklariert. Möchte der Experte das nicht tun, kann er zur nächsten Empfehlung weiternavigieren, die ein gemeinsames Basiskonzept für die Konzepte `DetachedTask`, `PaperTask` und `ElectronicTask` basierend auf den Attributen `title` und `duration` vorschlägt. Dies kann er solange fortführen, bis er letztlich das ausschließlich auf `duration` beruhende Abhängigkeitstupel erreicht.

4.4.2 Erweiterter Powertyp als Verfeinerungsvorschlag

Nachdem der erweiterte Powertyp (Abschnitt 2.2.1.4) über einen größeren Geltungsbereich als die Einfachvererbung verfügt, wird pro Meta-Modell **maximal eine Empfehlung** gegeben, wo dieses Muster zur Qualitätssteigerung eingesetzt werden könnte. Das ist dann der Fall, wenn es mehrere Konzepte gibt, die sich in verschiedenen Attributen überschneiden, die Schnittmengen selbst aber möglichst selten äquivalent sind.

Der **Algorithmus** (Abbildung 4-9), der berechnet, ob Indizien für die Einführung eines erweiterten Powertyps sprechen, basiert auf den Abhängigkeitstupeln des Algorithmus aus Abbildung 4-8. Ausgangspunkt sind hierbei die Mengen korrespondierender Attribute des in der Kandidatenliste enthaltenen ersten und somit besten Kandidaten, der für die Anwendung von Einfachvererbung spricht. Im Beispiel aus Abbildung 4-7 handelt es sich demnach um `next`, `title` und `duration`. Treten diese korrespondierenden Attribute in einigen anderen Konzepten ebenfalls auf (d.h. in Konzepten, die nicht im besten Kandidaten vorkommen), dann wird eine Empfehlung für die Einführung eines Powertyps gegeben, da eine auf Generalisierung beschränkte Lösung zu einer komplexen Vererbungshierarchie führen könnte.

Es wird **zu jeder Attributmenge aus dem besten Abhängigkeitstupel A** das entsprechende **initiale Abhängigkeitstupel B aus der Kandidatenliste gesucht**. Alle initialen Tupel befinden sich aufgrund der Sortierung am Listenende. Daraufhin wird überprüft, ob B mehr Konzepte umfasst als A. Ist das der Fall, dann heißt dies, dass das aktuelle Attribut in noch weiteren Konzepten deklariert ist, als in

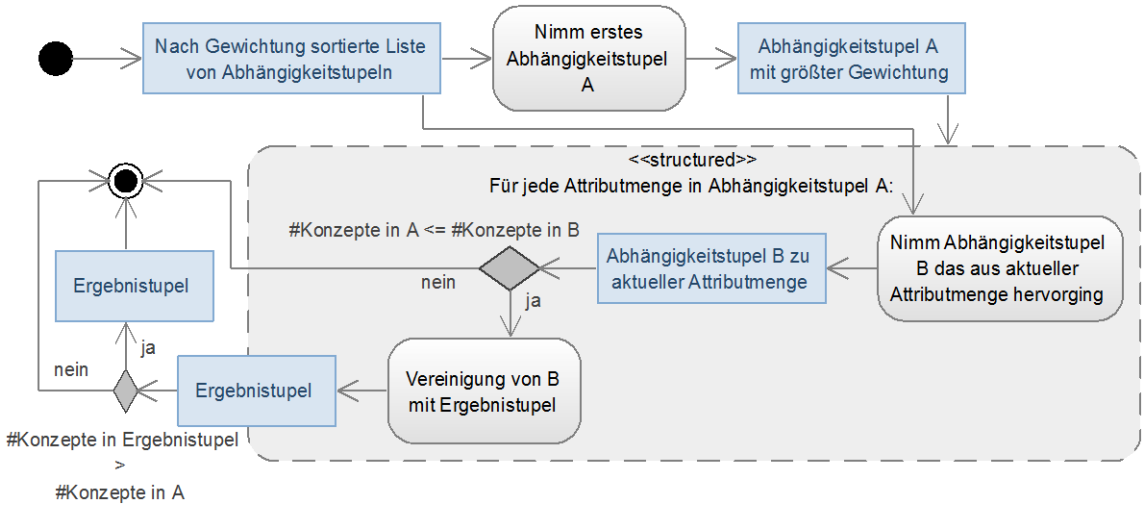

Abbildung 4-9: Aktivitätsdiagramm für die Berechnung eines Vorschlags zur Einführung eines erweiterten Powertyps

Tupel A enthalten sind. Das ist ein Indiz für einen erweiterten Powertyp, weil die Gegebenheit auf eine zusätzliche Gemeinsamkeit mit anderen Konzepten hindeutet.

Aufgrund dessen erfolgt eine **Vereinigung des Ergebnistupels mit dem Abhängigkeitstupel B**, was dem Hinzufügen der Attributmenge und der Konzepte aus B zum Ergebnistupel gleichkommt. Im Lauf der ersten Iteration ist noch kein Ergebnistupel vorhanden, weshalb es zunächst angelegt wird. Der Aufbau des Ergebnistupels ist demzufolge identisch mit dem der Abhängigkeitstupel, d.h. der erste Eintrag umfasst alle betreffenden Attributmengen und der zweite beinhaltet sämtliche Konzepte, in denen die Attribute des ersten Eintrags deklariert sind.

Falls die Anzahl der Konzepte im Ergebnistupel kleiner oder gleich der Anzahl der Konzepte in Tupel A ist, dann liefert der Algorithmus kein Resultat und somit keine Empfehlung für die Einführung eines erweiterten Powertyps. Nur **wenn das Ergebnistupel mehr Konzepte umfasst als der beste Kandidat für Einfachvererbung**, dann gibt es Gemeinsamkeiten zwischen bestimmten Konzepten, die allerdings nicht für alle Konzepte des Kandidaten gelten. Dies kann bei striktem Einsatz des Sprachmusters „Einfachvererbung" zu einer mehrstufigen und komplexen Vererbungshierarchie führen. Deshalb wird in diesem Fall das berechnete Ergebnistupel als **Empfehlung für die Einführung eines erweiterten Powertyps** ausgegeben. Sind dagegen beide Konzeptmengen gleich groß, erfolgt keine Ausgabe eines derartigen Vorschlags.

Abbildung 4-10 zeigt die Ergebnistupel für die zu Abbildung 4-7 gehörende Kandidatenliste in den verschiedenen Zuständen während der Ausführung des vorgestellten Algorithmus. Abhängigkeitstupel A entspricht hierbei dem untersten Tupel aus Abbildung 4-7. Als erste Attributmenge wird `next` verarbeitet, was dem ersten Abhängigkeitstupel (jetzt B) entspricht. Nachdem die Anzahl der Konzepte von B größer ist als die Anzahl der Konzepte von A, werden alle Konzepte von B (also `And`, `ElectronicTask`, `PaperTask` und `Start`) sowie die Menge korrespondierender `next`-Attribute dem Ergebnistupel hinzugefügt.

Die nächste Iteration bezieht sich auf die `title`-Attributmenge, mit der analog verfahren wird. Zu ihr gehört das zweite Abhängigkeitstupel (jetzt B), dessen Konzeptanzahl wieder größer als die entsprechende Anzahl von Konzepten in Tupel A ist. Bei der sich anschließenden Vereinigung beider Tupel wird das Ergebnistupel um die Attributmenge `title` sowie das Konzept `DetachedTask` ergänzt.

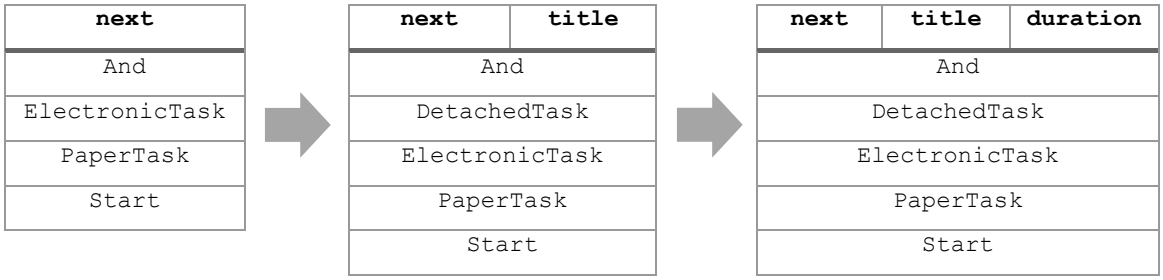

Abbildung 4-10: Beispiel für die Bestimmung eines Ergebnistupels, das auf die Einführung eines erweiterten Powertyps hindeutet

In der letzten Iteration erfolgt unter Verarbeitung der `duration`-Attributmenge dasselbe Prozedere, wodurch lediglich diese Attributmenge dem Ergebnistupel hinzugefügt wird. Alle betreffenden Konzepte sind bereits Teil des Tupels. Zum Schluss wird festgestellt, dass die Anzahl der Konzepte des Ergebnistupels größer ist als die Konzeptanzahl des Abhängigkeitstupels A, weshalb das Ergebnistupel als Empfehlung für die Einführung eines erweiterten Powertyps ausgegeben wird. In Abbildung 4-10 handelt es sich um die Tabelle auf der rechten Seite.

4.4.3 Enumeration als Verfeinerungsvorschlag

Eine Enumeration stellt einen Datentyp mit stark eingeschränktem Wertebereich dar [14]. Sie umfasst in der Regel nur wenige Literale, die als Werte bei Zuweisungen in Frage kommen. Deshalb ergibt der Vorschlag eine Enumeration als Datentyp einzuführen **nur bei korrespondierenden Attributen** Sinn, **deren zugehörige Zuweisungen wiederholt über identische Werte verfügen**. Wegen derselben lexikalischen Struktur von Pointer und Enumerationsliteral kann der Benutzer lediglich bei abgeleiteten Attributen mit Datentyp Pointer alternativ eine Enumeration gemeint haben. Folglich gelten im aktuellen Kontext zwei Attribute nur dann als korrespondierend, wenn sie denselben Namen besitzen und vom Typ Pointer sind.

```
1  Job J1
2    phase = PRE
3
4  Job J2
5    phase = POST
6
7  Job J3
8    phase = PRE
9
10 Job J4
11   phase = DEFAULT
12
13 Job J5
14   phase = DEFAULT
15
16 Job J6
17   phase = PRE
```

Listing 4-3: Beispielmodell, das einen Vorschlag zur Einführung einer Enumeration bedingt

Zusätzlich sollte die Anzahl möglicher Werte knapp bemessen sein. Eine feste Vorgabe, ab wann die Knappheitsbedingung nicht mehr erfüllt ist, kann allerdings nicht gegeben werden, da dies vom jeweiligen Einsatzszenario und den Vorlieben des Modellierungsexperten abhängt. Stattdessen liegt im Fokus der Analyse die wiederholte Zuweisung der gleichen Pointer-Werte an zueinander korres-

pondierende Attribute. Die Einführung einer Enumeration wird somit vorgeschlagen, wenn **mindestens zwei unterschiedliche Werte wiederholt an dieselbe Menge korrespondierender Attribute zugewiesen** werden.

Ein Beispiel für ein gültiges Szenario zeigt Listing 4-3. Es repräsentiert ein Beispielmodell mit sechs Instanzen, die alle eine Zuweisung an das imaginäre Pointer-Attribut `phase` beinhalten. Das abgeleitete Meta-Modell besteht ausschließlich aus dem Konzept `Job`, welches das besagte Attribut `phase` manifestiert. Nachdem es sich hierbei um ein Pointer-Attribut handelt und sowohl das Literal `PRE` als auch das Literal `DEFAULT` in mindestens zwei zugehörigen Zuweisungen auftaucht (nämlich in `J1`, `J3` und `J6` bzw. in `J4` und `J5`) , wird ein Vorschlag generiert, der die Einführung einer Enumeration empfiehlt.

4.5 Ableiten von Änderungen an abstrakter Syntax

Grundlegendes Prinzip beim Ableiten von Änderungen an der abstrakten Syntax ist, dass diese **Änderungen minimal** gehalten werden. Das existierende Meta-Modell wird demzufolge nur insoweit angepasst, dass modifizierte oder neu hinzugekommene Beispielmodelle Gültigkeit besitzen. Begründet ist dies mit der Möglichkeit des Benutzers, Meta-Modelle nach Belieben manuell verändern und an die jeweiligen Bedürfnisse anpassen zu können. Würde nun stets das vorhandene Meta-Modell verworfen und eine vollständige Neugenerierung erfolgen, so wären alle manuell getätigten Modifikationen am Meta-Modell verloren. Welche Modifikationen am Meta-Modell beim wiederholten Ableiten zwingend erforderlich sind und daher automatisiert vorgenommen werden, beschreibt der Unterabschnitt 4.5.1.

In Unterabschnitt 4.5.2 wird erneut die Idee aus Abschnitt 4.4 aufgegriffen, Empfehlung für Anpassungen am Meta-Modell zu unterbreiten. Diese Empfehlungen beruhen größtenteils auf evolvierten Rahmenbedingungen durch modifizierte oder ergänzte Beispielmodelle. Abschließend werden einige der beschriebenen Änderungsmöglichkeiten anhand eines Beispiels illustriert.

Die folgenden Ausführungen wurden bereits in der 7. Ausgabe des International Journal on Advances in Software publiziert [122]. Allerdings greift der kommende Text zusätzliche Aspekte auf, die nicht im besagten Artikel enthalten sind. Dies betrifft primär die Abhandlungen in Abschnitt 4.5.2. Als Ergänzung zu Abschnitt 4.3 tragen auch die kommenden Ausführungen zur Erfüllung von Anforderung 2 bei.

4.5.1 Notwendige Änderungen

Um die Konformität der Beispielmodelle hinsichtlich der abstrakten Syntax sicherzustellen, müssen in jedem Fall diejenigen Artefakte der Modelle extrahiert werden, die in **Konflikt zum Meta-Modell** stehen. Konfliktpotential tragen die um Schemalosigkeit erweiterten Komponenten des LMM (Abschnitt 4.1). Das sind primär die Typnamen von Konzepten sowie Zuweisungen im Allgemeinen.

Ist der freie Modellierungsmodus aktiviert, so darf der Benutzer **neue Instanz-Konzepte** mit einem noch nicht existenten Typ in Form eines Typnamen versehen. Für derartige Instanzen wird genauso verfahren, wie beim initialen Ableiten einer abstrakten Syntax (siehe Abschnitt 4.3 und insbesondere Abbildung 4-2).

Ändert der Benutzer den **Typnamen eines vorhandenen Instanz-Konzepts**, das bereits mit einem Meta-Konzept verknüpft ist, und passt dieser **Typname zu keinem anderen existierenden Meta-Konzept**, dann wird das betreffende Instanz-Konzept ebenfalls als neu betrachtet. Eventuell enthaltene Zuweisungen werden gleichzeitig von zugrundeliegenden Attributen gelöst. Anschließend kann wie bei einer gänzlich neuen Instanz fortgefahren werden.

Wird der **Typ eines Instanz-Konzepts auf ein anderes, bereits definiertes Meta-Konzept umgebogen**, dann kann das dazu führen, dass es zu bereits vorhandenen Zuweisungen kein passendes Attribut mehr gibt. Solche Zuweisungen werden auf die gleiche Weise behandelt, wie nachfolgend beschrieben.

Im freien Modus können beliebig **neue dynamische Zuweisungen** (d.h. Zuweisungen ohne zugehöriges Attribut) innerhalb von Instanzen erstellt werden, die schon über ein Meta-Konzept verfügen. Auch hier wird analog wie beim initialen Ableiten der abstrakten Syntax verfahren (Abbildung 4-2). Für jede Zuweisung ohne zugrundeliegendes Attribut wird ein solches Attribut erzeugt, allerdings ohne dieses einem Meta-Konzept zuzuordnen. Danach werden pro Meta-Konzept Mengen gleichnamiger Attribute bestimmt. Jede dieser Mengen wird mithilfe des in Abschnitt 4.3.1 geschilderten Verfahrens zu einem Attribut verschmolzen und schließlich dem jeweiligen Meta-Konzept hinzugefügt. Dadurch wird ein existierendes Meta-Konzept um ein zur neu erstellten Zuweisung passendes Attribut erweitert.

Darüber hinaus können **Zuweisungen mit zugrundeliegendem Attribut beliebige Werte auf der rechten Seite** aufweisen, sofern der jeweilige Tenor – also literal oder referentiell – nicht verletzt wird. Angenommen es existiert ein Integer-Attribut mit dem Namen „height", dann dürfen Zuweisungen im freien Modus auch von jedem anderen literalen Typ sein. Ein sinnvoller Wert wäre beispielsweise `3.5`, wobei dieser außerhalb des Wertebereichs von Ganzzahlen liegt. Beim Ableiten der damit verbundenen Meta-Modell-Änderung würde der literale Typ des Attributs „height" somit auf Float gesetzt werden. Auch hier kann wiederholt auf Strategien, die bereits beim initialen Ableiten einer abstrakten Syntax zum Einsatz kommen und in Abschnitt 4.3.1 erläutert sind, zurückgegriffen werden. Bei einem zugrundeliegenden literalen Attribut erfolgt eine Typkonvertierung hin zu einem größeren Wertebereich, wie in Tabelle 4-2 beispielhaft dargelegt. Bei einem referentiellen Attribut hingegen muss ein gemeinsames Basiskonzept gesetzt werden, das bei Nichtvorhandensein neu angelegt wird.

Zusätzlich unterschieden werden müssen **Attribute mit einer Enumeration als Datentyp**. Gültige Werte sind prinzipiell Pointer-Werte, die nicht ein anderes Konzept referenzieren. Treten Werte auf, zu denen es noch kein Enumerationsliteral gibt, dann wird ein entsprechendes Literal neu generiert und der Enumeration hinzugefügt. Außerdem werden auch Zeichenketten als gültig betrachtet. Wird eine Zeichenkette zugewiesen, dann wird der Attributtyp auf String geändert und sämtliche zugewiesenen Enumerationsliterale in String-Darstellung konvertiert. Dieses Vorgehen entspricht demjenigen bei der automatischen Typkonvertierung von Pointer nach String. Alle anderen Datentypen sind nicht zulässig und führen bei Zuweisung entsprechender Werte zum Abbruch des Ableitungsprozesses.

Die sechs präsentierten Fälle umspannen alle möglichen Änderungsarten innerhalb von Instanz-Modellen, die eine darauffolgende Anpassung des zugrundeliegenden Meta-Modells erfordern. Andernfalls käme es beim Wechsel in den stringenten Modellierungsmodus zu einer Invalidierung der modifizierten Instanz-Modelle.

4.5.2 Änderungsvorschläge

Die verschiedenen Arten von Änderungsvorschlägen lassen sich in **drei Kategorien** einteilen. In der ersten Kategorie befinden sich alle Empfehlungen, die im Zusammenhang mit dem Wertebereich von Attributen stehen (Abschnitt 4.5.2.1). Die zweite Kategorie umfasst Vorschläge zum Löschen von Konzepten und Enumerationen der abstrakten Syntax (Abschnitt 4.5.2.2). In Kategorie drei schließlich werden Änderungsvorschläge vorgestellt, die auf die Aufhebung von Sprachmustern hinweisen (Abschnitt 4.5.2.3). Damit repräsentieren sie den Gegenpart der Empfehlungen aus Abschnitt 4.4.

4.5.2.1 Verschärfung der Restriktionen bei Attributen

Beim Ableiten von Änderungen an Attributen werden Restriktionen ausschließlich aufgeweicht. Eine Verschärfung ist aus Gründen der manuellen Modifizierbarkeit nicht erwünscht. Stattdessen können dem Benutzer unter bestimmten Umständen jedoch Verengungen von Multiplizität bzw. Attributtyp empfohlen werden.

Zur Berechnung, ob die **Multiplizität eines Attributs verengt** werden kann, sind wiederum untere und obere Grenze separat zu betrachten. Die Einschränkung der unteren Grenze wird empfohlen, wenn sie bislang auf 0 gesetzt war und alle Instanzen, in denen eine Zuweisung an das betreffende Attribut erfolgen kann, auch tatsächlich über eine solche Zuweisung verfügen. Eine Empfehlung zur Limitierung der oberen Grenze wird hingegen vorgeschlagen, wenn sie bisher auf * gesetzt war und keine Zuweisung an das betreffende Attribut mit mehr als einem Wert existiert.

Die Behandlung des Attributtyps erfolgt erneut zweigleisig, nämlich einerseits für literale und andererseits für referentielle Attribute. Für jedes **literale Attribut** wird anhand sämtlicher zugehöriger Zuweisungen überprüft, ob ein **Typ mit geringerem Wertebereich** ausreicht (siehe dazu auch Abschnitt 4.3.1 und vorrangig Tabelle 4-2). Ist das der Fall, dann wird vorgeschlagen, den gefundenen literalen Typ anstelle des bisherigen Attributtyps zu verwenden. Ein typisches Beispiel ist die Verengung des Attributtyps von Float zu Integer, weil lediglich ganze Zahlen zugewiesen werden.

Bei **referentiellen Attributen** wird ähnlich vorgegangen. Allerdings wird hier geprüft, ob ein **generalisierter Typ durch einen spezielleren ersetzt** werden kann. Es sind also ebenfalls sämtliche Zuweisungen an das jeweilige Attribut zu untersuchen. Als Beispiel kann ein Attribut betrachtet werden, das den Typ `ProcessOrAnd` trägt, weil bislang sowohl Prozesse und AND-Gateways zugewiesen wurden. Beim nächstmaligen Ableiten von Änderungen wird jedoch festgestellt, dass dem Attribut nun ausschließlich Prozesse zugewiesen sind. Demzufolge wird dem Benutzer die Empfehlung unterbreitet, den Typ des Attributs in `Process` zu ändern.

4.5.2.2 Löschen von Meta-Konzepten

Ob Meta-Konzepte wirklich überflüssig sind und folglich gelöscht werden dürfen, kann basierend auf geänderten Instanz-Modellen nicht zweifelsfrei entschieden werden. Jedes Meta-Konzept kann beispielsweise in einem abgekoppelten Modellrepositorium referenziert oder in einem nachgeschalteten Code-Generator verwendet werden. Demgemäß wird kein Meta-Konzept automatisiert entfernt, sondern dem Modellierungsexperten lediglich die Durchführung einer entsprechenden Löschoperation vorgeschlagen. Der offensichtlichste Fall ist gegeben, wenn ein **nicht-abstraktes Meta-Konzept von keinem anderen Konzept** im vorliegenden Modellrepositorium **instanziiert** wird. Ein derartiges Meta-Konzept wird dann als Kandidat zum Löschen empfohlen.

Ebenfalls zum Löschen vorgeschlagen werden **Konzepte**, die zwar in eine Generalisierungshierarchie eingebettet sind, sonst aber **nirgends referenziert** werden und **keine Attribute** deklarieren. Exemplarisch zu nennen ist das Konzept `ProcessOrAnd`, das am Ende des vorherigen Abschnitts 4.5.2.1 nicht länger als Attributtyp verwendet wird.

4.5.2.3 Aufhebung von Sprachmustern

Die beiden folgenden Fälle können als inverse Vorschläge zu denen betrachtet werden, die die Einführung von Sprachmustern empfehlen (Abschnitt 4.4). Beide führen allerdings zwangsläufig zur Eliminierung eines Meta-Konzepts, so dass externe Referenzen auf dieses Konzept, die nur einem Modellierungsexperten bekannt sind, ungültig gemacht würden. Ein Indiz, dass ein Meta-Konzept irrelevant ist, liegt vor, wenn dieses Konzept abstrakt ist und lediglich genau ein spezialisierendes

Konzept existiert. Für jedes **abstrakte Konzept**, das diese Bedingung erfüllt, wird ein Vorschlag generiert, der empfiehlt, es **in dessen Spezialisierung zu integrieren und daraufhin ersatzlos zu eliminieren**. Angenommen, das Konzept `PaperTask` wird aus dem in Abbildung 4-5 visualisiertem Meta-Modell manuell entfernt, dann würde das zum Vorschlag führen, die Attribute `title` und `duration` des abstrakten Konzepts `Task` in das Konzept `ElectronicTask` zu verschieben und anschließend `Task` durch `ElectronicTask` zu ersetzen.

Beim erweiterten Powertyp ist ein Indiz für dessen Überflüssigkeit gegeben, wenn jede seiner Instanzen sämtliche Attribute aktiviert oder alternativ sämtliche Attribute deaktiviert und somit sein Einsatz völlig degeneriert ist. Dann würde das weitaus weniger komplexe Muster „Einfachvererbung" ausreichen. Demzufolge wird eine einfache **Spezialisierungsbeziehung als Vorschlag** unterbreitet, sobald eine der beiden degenerierten Aktivierungskonfigurationen auftritt. Im beispielhaften Meta-Modell aus Abbildung 4-6 bedeutet das, dass alle Powertyp-Instanzen (also `ElectronicTask`, `PaperTask`, `DetachedTask`, `Start`, `And` und `Exit`) die Attribute `supportsNext`, `supportsTitle` und `supportsDuration` entweder auf `true` setzen oder komplett ignorieren müssten, damit der beschriebene Vorschlag angeboten wird. Liegt zusätzlich nur eine Powertyp-Instanz vor, wird empfohlen die aktivierten Attribute in diese Instanz zu verschieben und sowohl Powertyp als auch partitionierten Typ zu löschen.

Wird eine **Enumeration in keinem Attribut als Datentyp** verwendet, dann kann sie als überflüssig betrachtet werden. In einem solchen Fall wird ein Vorschlag erzeugt, der empfiehlt die betreffende Enumeration zu entfernen.

4.5.3 Beispiel

Das Beispielmodell aus Listing 4-4 repräsentiert weitgehend denselben Prozess wie in Listing 4-2 dargestellt. Beide Modelle unterscheiden sich nur durch drei kleine, vom Benutzer getätigte Änderungen, die in Listing 4-4 rot hervorgehoben sind. Zu dem ursprünglichen Beispiel wurde bereits ein Meta-Modell generiert und vom Benutzer dahingehend angepasst, dass es der Variante aus Abbildung 4-1 entspricht. Dieses Meta-Modell sei nun die Grundlage für die Ableitung von Änderungen basierend auf den nachfolgend beschriebenen Modifikationen am Beispielmodell.

Die erste Änderung befindet sich in Zeile 12. Hier wird dem bereits vorhandenen `duration`-Attribut anstelle der Zahl `0.5` der String `"0.5h"` zugewiesen (fünfter Fall in Abschnitt 4.5.1). Beim inkrementellen Ableiten des Meta-Modells hat das zur Folge, dass der Typ des vorhandenen literalen Attributs zu einem String aufgeweitet wird (siehe Konzept `Task` in Abbildung 4-11) und sämtliche bisherigen Zuweisungen zu diesem Attribut in einen String konvertiert werden. Beispielsweise wird der in Zeile 7 zugewiesene Wert von `1` in `"1"` umgewandelt.

Zeile 13 umfasst mit der Zuweisung des booleschen Werts `true` an das noch nicht im Meta-Modell vorhandene Attribute `waitForReturn` eine weitere Modifikation des Beispielmodells (vierter Fall

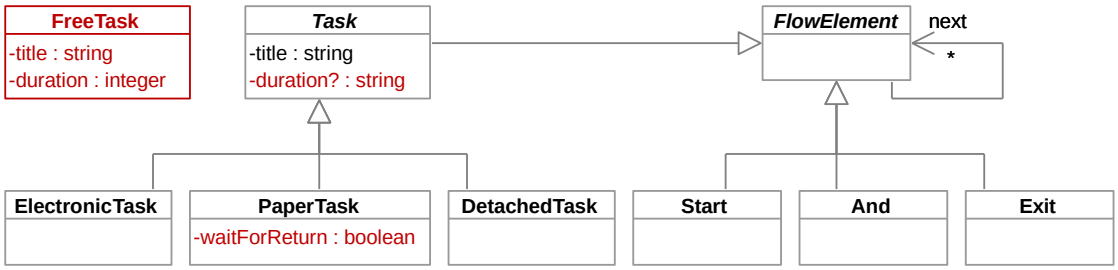

Abbildung 4-11: Automatisch angepasstes Meta-Modell für modifiziertes linguistisches Beispielmodell

in Abschnitt 4.5.1). Nachdem es die einzige Instanz von `PaperTask` betrifft, wird dieses Meta-Konzept beim Ableiten der Änderungen um ein entsprechendes boolesches Attribut ergänzt.

Die dritte und letzte Modifikation des Beispielmodells ist die Änderung des Typs von `Inform` in Zeile 35. Hierbei wird `DetachedTask` durch `FreeTask` ersetzt, wobei zu Letzterem noch kein zugehöriges Meta-Konzept existiert. Gemäß des zweiten Falls in Abschnitt 4.5.1 hat ein geänderter Typname, zu dem es noch kein Meta-Konzept gibt, dieselben Auswirkungen zur Folge, wie ein gänzlich neues Instanz-Konzept mit einem Typnamen ohne zugehöriges Konzept. Deswegen wird für `Inform` ein neues Meta-Konzept mit dem Namen `FreeTask` sowie zwei darin enthaltener Attribute `title` und `duration` generiert. Danach gibt es jedoch kein Konzept mehr, das `DetachedTask` instanziiert. Aus diesem Grund wird laut Abschnitt 4.5.2.2 dem Benutzer vorgeschlagen, das besagte Konzept zu löschen.

```
1  Start S
2    next = Search
3
4  ElectronicTask Search
5    title = "Suche nach passender Konferenz"
6    next = Request
7    duration = 1
8
9  PaperTask Request
10   title = "Einreichung eines Reisekostenantrags"
11   next = Split
12   duration = "0.5h"
13   waitForReturn = true
14
15 And Split
16   next = Register, Booking
17
18 ElectronicTask Register
19   title = "Registrierung bei Konferenz"
20   duration = 0.3
21   next = Join
22
23 ElectronicTask Booking
24   title = "Buchung eines Hotels"
25   next = Organize
26
27 ElectronicTask Organize
28   title = "Planung von An- und Abreise"
29   duration = 0.6
30   next = Join
31
32 And Join
33   next = E
34
35 FreeTask Inform
36   title = "Informieren über Inhalte"
37   duration = 1
38
39 Exit E
```

Listing 4-4: Manuell modifiziertes Beispielmodell für die Bottom-Up Anpassung einer abstrakten Syntax im linguistischen Kontext

4.6 Kompensation von lückenhaftem Domänenwissen

Das für den Rechner verfügbare Wissen über eine Domäne steckt allein in den frei erstellten Beispielmodellen und dabei insbesondere in den Instanz-Konzepten sowie deren Zuweisungen. Dieser Sachverhalt wurde schon einleitend in Kapitel 4 anhand eines Beispiels erläutert. Beim Ableiten einer abstrakten Syntax wird bereits sämtliches verfügbare Wissen ausgewertet und in das resultierende Meta-Modell gesteckt. Allerdings treten hierbei auch **Fälle** auf, an denen **kein tiefergehendes Wissen** vorhanden ist und deshalb auf vorgegebene Werte oder heuristische Strategien zurückgegriffen wird. Beispielhaft zu nennen ist die Namensgebung bei extrahierten Basiskonzepten und die automatische Typaufweitung bei literalen Attributen. Dieses lückenhafte Domänenwissen gilt es nun weitgehend zu kompensieren, so dass ein zufriedenstellendes Ergebnis erzielt werden kann.

Insgesamt sind drei atomare Lösungen denkbar. Lösung eins fordert stets direkt **Feedback vom Benutzer** ein, sofern eine Entscheidung zusätzlicher Informationen bedarf. Sie ist nur dann tragbar, wenn der Benutzer auch gleichzeitig der Modellierungsspezialist ist, da eine Person ohne detaillierte Kenntnisse im Bereich der Meta-Modellierung nicht über den für derartige Entscheidungen erforderlichen Sachverstand verfügt. Im Wesentlichen handelt es sich dabei um die zweite Methode aus Abschnitt 3.3. Als Lehrer gilt der Benutzer, der aufgefordert wird Feedback abzugeben.

Die zweite Lösung setzt eine **allgemeine Konfiguration** voraus, die für alle potentiell auftretenden Entscheidungen vordefinierte Parameter zur Verfügung stellt. Es ist Aufgabe des Modellierungsexperten die Konfiguration derart zu parametrieren, dass sie seinen Anforderungen genügt.

Die Vorgabe einer **fixen Strategie** für jede potentiell vorkommende Entscheidung stellt die dritte atomare Lösung dar. Tritt ein Entscheidungsfall ein, so wird zudem eine Meldung generiert, welche auf die von der Maßnahme betroffenen Stellen im Meta-Modell hinweist. Gleichzeitig wird – falls möglich – auf Operationen verwiesen, mit denen eine alternative Lösung umgesetzt werden kann (z.B. Umbenennung eines Konzepts). Wichtig zu erwähnen ist, dass der Benutzer dadurch nicht in seinem Arbeitsfluss behindert wird, da die Hinweise erst nachträglich angezeigt werden und keine Bestätigung vom Benutzer verlangen.

Jede der vorgestellten Lösungen besitzt ihrerseits spezifische Vorteile. Deshalb ist es ratsam eine **kombinierte Lösung** zu verfolgen. Die Basis bildet dabei stets eine allgemeine Konfiguration, die für jeden potentiell auftretenden Entscheidungsfall einen eigenen Parameter bereitstellt. Pro Parameter kann aus einer Menge alternativer Strategien ausgewählt werden, wobei immer die Option, Feedback vom Benutzer einzufordern, angeboten wird. Darüber hinaus kann für jeden Parameter festgelegt werden, ob bei einer automatischen Entscheidung eine entsprechende Meldung generiert werden soll, die später einen Modellierungsspezialisten auf eine potentielle Problemstelle hinweist. Der Spezialist kann dann selbst entscheiden, ob er das generierte Artefakt anpassen möchte oder nicht.

Es ist ratsam, **Konfigurationen in Form von Profilen zu organisieren**. Dies ermöglicht ein müheloses Umschalten zwischen verschiedenen Konfigurationen durch den Wechsel von Profilen. Damit kann jeder Modellierungsexperte für sich und sein Team (iterativ) die beste Konfiguration ermitteln und zu jeder Zeit auf eine frühere Konfiguration zurückspringen, sofern sie als Profil abgelegt wurde.

4.6.1 Konfigurationsmöglichkeiten beim Ableiten einer abstrakten Syntax

Die Konfigurationsmöglichkeiten beschränken sich beim Ableiten einer abstrakten Syntax auf den Schritt, in dem die **Verschmelzung gleichnamiger Attribute** stattfindet (Abschnitt 4.3.1). Unterschieden wird hierbei erneut zwischen den beiden Attributarten literal und referentiell.

Während der Erzeugung gleichnamiger **literaler Attribute**, die demselben Meta-Konzept zugeordnet sind, findet laut obiger Beschreibung beim zwangsläufigen Verschmelzen stets eine Konvertierung des Typs in Richtung eines größeren Wertebereichs statt. Ob dieses Verhalten jedoch wirklich vom Benutzer erwartet wird und erwünscht ist, kann nicht mit Sicherheit beantwortet werden. Insbesondere gilt das, wenn zwei Datentypen, die von ihrer ursprünglichen Intention gänzlich verschieden sind (z.B. Boolean und Float oder Integer und String). Daher ist es zweckmäßig, den Verschmelzungsprozess derart konfigurierbar zu machen, dass festgelegt werden kann, zwischen welchen literalen Typen eine Konvertierung automatisch erfolgen soll und wann z.B. der Ableitungsvorgang unter Ausgabe einer Fehlermeldung abgebrochen wird. Zusätzlich kann der Benutzer einstellen, ob im Falle einer automatischen Typaufweitung eine Meldung angezeigt werden soll.

Liegen dagegen mehrere gleichnamige **referentielle Attribute** vor, die unterschiedliche Typen aufweisen, dann muss – sofern noch nicht vorhanden – ein gemeinsames Basiskonzept erzeugt werden. Hierzu ist für dieses neue Konzept ein Name festzulegen, der allerdings nicht aus verfügbarem Wissen geschlussfolgert werden kann. Demzufolge muss eine Strategie für die Berechnung des Namens bereitgestellt werden.

In der prototypischen Implementierung zur vorliegenden Arbeit wird der Name des neuen Basiskonzepts durch Konkatenation der Namen der Ausgangskonzepte inklusive verbindendem „Or" berechnet. Ein Beispiel dafür stellt das Konzept `ElectronicTaskOrPaperTaskOrAnd` aus Abbildung 4-3 dar. Zweifellos sind auch andere Strategien denkbar, die als Option angeboten werden können (z.B. Konkatenation der Namen mit vorangestelltem „Abstract", wie in [124] vorgeschlagen). Eine weitaus bessere Strategie wäre allerdings Informationen aus anderen Meta-Modellen hinzuzuziehen. Dafür ist das Modellrepositorium zunächst nach gleichlautenden Konzepten zu durchsuchen. Werden solche Konzepte gefunden und verfügen sie über ein gemeinsames Basiskonzept, dann kann dessen Name direkt übernommen werden. Andernfalls muss für die Namensgebung eine alternative Strategie angewendet werden. Der generierte bzw. ausgewählte Namen ist damit ein typischer Kandidat, den ein Benutzer einem Review unterziehen sollte. Deshalb wird im Standardfall (d.h. wenn nicht explizit vom Benutzer deaktiviert) eine entsprechende Meldung produziert.

4.6.2 Konfigurationsmöglichkeiten für Verfeinerungsvorschläge

Als ebenfalls sinnvoll zeigt sich die Konfigurierbarkeit von Verfeinerungsvorschlägen, weil dadurch der Modellierungsexperte einstellen kann, unter welchen Gegebenheiten und für welche Konstellationen von Modellelementen Vorschläge generiert werden sollen. Nachdem diese Vorschläge immer auf korrespondierenden Attributen basieren, muss **einstellbar** sein, **wann eine Korrespondenz zwischen zwei Attributen vorliegt** und wann nicht. Im Standardfall korrespondieren Attribute bereits dann, wenn sie nur im Namen und der Attributart übereinstimmen. Multiplizität und Typ werden nicht beachtet, was jedoch für einige Anwender zu wenig restriktiv sein kann. Diesem Umstand wird anhand der im Folgenden beschriebenen Parametriermöglichkeiten genüge getan.

Da die **Multiplizität** durch eine obere und eine untere Schranke festgelegt ist, kann die Berücksichtigung beider Schranken separat reguliert werden. Demnach ist konfigurierbar, ob die Multiplizität komplett ignoriert werden soll, ob nur die unteren bzw. nur die oberen Schranken übereinstimmen müssen oder ob die Multiplizitäten identisch sein müssen, damit eine Korrespondenz vorliegt.

Bei der **Betrachtung des Typs** gilt es die Attribute wiederum nach ihrer Art zu unterscheiden. Allerdings muss in beiden Fällen die zuvor beschriebene Multiplizitätsbedingung erfüllt sein. Liegen gleichnamige literale Attribute vor, so können dieselben Einstellungen verwendet werden, die schon bei der Konfiguration für das automatische Verschmelzen literaler Attribute zum Einsatz kommen. Anstatt im nicht-übereinstimmenden Fall eine Fehlermeldung auszugeben, wird einfach die Korrespondenz der

betreffenden Attribute geleugnet. Treten hingegen gleichnamige referentielle Attribute auf, können drei Optionen differenziert werden. Die erste Option betrachtet zwei Attribute nur dann als potentiell gleichartig, wenn sie exakt denselben Typ aufweisen. Die zweite Option ist vollständig konträr dazu und vernachlässigt den Typ gänzlich. Option drei ist eine Erweiterung der ersten Option. Dabei wird zusätzlich die Generalisierungshierarchie analysiert und wenn ein gemeinsames Basiskonzept existiert, dann werden die betrachteten Attribute als korrespondierend angesehen.

Einen **Sonderfall** stellen **Vorschläge für die Einführung von Enumerationen** dar, weil sie einerseits nur für korrespondierende Pointer-Attribute gelten und andererseits zusätzlicher Parameter bedürfen. Gemäß der Beschreibung in Abschnitt 4.4.3 wird die Einführung einer Enumeration vorgeschlagen, falls mindestens zwei unterschiedliche Werte an wenigstens zwei der korrespondierenden Attribute zugewiesen werden. Besonders bei großen und komplexen Meta-Modellen kann dies zu vielen Verfeinerungsvorschlägen hinsichtlich Enumerationen führen. Deshalb ist sowohl die Anzahl mindestens auftretender unterschiedlicher Werte als auch die Anzahl der Attribute konfigurierbar, die korrespondieren müssen. Der minimal einstellbare Wert beträgt in beiden Fällen 2. Alternativ zur Angabe einer fixen Anzahl gleicher Literale kann stattdessen ein Prozentwert angegeben werden, der sich auf die Anzahl aller eindeutigen Literale bezieht. Ein Wert von 50% besagt demnach (sofern überdies der zweite Parameter auf 2 gesetzt ist), dass mindestens die Hälfte aller Literale in mindestens zwei Zuweisungen verwendet wird.

4.7 Zusammenfassung

Kernthema dieses Kapitels ist die Entwicklung von DSMLs anhand von Beispielen unter bloßer Verwendung der LML. Um dies zu ermöglichen, musste die LML um die Möglichkeit erweitert werden, Instanzen spezifizieren zu können, ohne dass ein entsprechendes (ontologisches) Meta-Konzept zugrunde liegt. Derartige Meta-Konzepte werden erst in einem späteren Schritt inkrementell induziert. Aufgrund des inkrementellen Ansatzes können Instanzen auch im Nachhinein noch frei modifiziert und notwendige Änderungen automatisiert auf das Meta-Modell übertragen werden. Beschränkt wird sich auf die abstrakte Syntax, da durch die LML bereits eine generische konkrete Syntax vorgegeben ist. Im Ergebnis handelt es sich hierbei um die **Erfüllung von Anforderung 2**, die die Möglichkeit fordert, DSMLs intuitiv und konstruktivistisch mit linguistischen Mitteln entwickeln zu können.

Auch Anforderung 3 und damit die Berücksichtigung des Qualitätsaspekts wird in diesem Kapitel **adressiert**. Dies geschieht durch die automatisierte Unterbreitung von Verfeinerungsvorschlägen für generierte Meta-Modellelemente mit Optimierungspotential. Dazu wird das Meta-Modell anhand bestimmter, auf Heuristiken beruhender Qualitätskriterien analysiert. Anschließend obliegt es dem Benutzer, ob und wie er ein vorgeschlagenes Muster auf die betreffenden Meta-Modellelemente anwendet.

Die in Kapitel 4 vorgestellte Methodik kann in ihrer Gesamtheit als eigener, **konzeptueller Beitrag** angesehen werden. Der einzige Ansatz, der eine ähnliche Lösung bereitstellt, ist MetaBUP (Abschnitt 3.4.4). Allerdings ist er parallel zur vorliegenden Arbeit entstanden, weshalb ein Rückbezug nicht möglich war.

Insgesamt liefert dieses Kapitel die Grundlage für die beispielgetriebene Entwicklung von DSMLs im ontologischen Kontext, wie sie in den drei nachfolgenden Kapiteln präsentiert wird. Ein Großteil der vorgestellten Prinzipien und Vorgehensweisen wird daher in den kommenden Ausführungen aufgegriffen und teilweise identisch übernommen. Vornehmlich betrifft dies die Strategien zur Unterbreitung von Verfeinerungsvorschlägen (Abschnitte 4.4 und 4.5.2) sowie zur Kompensation von lückenhaftem Domänenwissen (Abschnitt 4.6).

5 Grundlagen der beispielgetriebenen Entwicklung im ontologischen Kontext

Motiviert durch Anforderung 1 soll die beispielgetriebene Entwicklung von DSMLs nicht nur im linguistischen Kontext (Kapitel 4), sondern auch im ontologischen Kontext unterstützt werden. Die **ontologische Variante** kann **als logische Fortführung der linguistischen** betrachtet werden, da es neben der Ableitung der abstrakten Syntax auch um die Bestimmung einer konkreten Syntax geht. Von zentraler Bedeutung ist dabei ebenfalls die automatisierte Festlegung eines Mappings zwischen beiden Syntaxen. Der Begriff „ontologisch" meint hier, dass zur Visualisierung der Modelle nicht die LML (Abschnitt 2.2.3) verwendet wird, sondern stattdessen eine vom Benutzer definierte Darstellung zum Einsatz kommt. Nicht berücksichtigt wird dabei – wie bereits in der Einleitung begründet – die semantische Komponente von DSMLs.

Grundsätzlich existieren **zwei** voneinander zu unterscheidende **Arten für die Visualisierung von Modellen** und damit Klassen konkreter Syntaxen, nämlich grafisch und textuell (Überbegriff „visuell") [80, 82]. Andere Arten können unter Zuhilfenahme dieser beiden umgesetzt werden. Im Folgenden wird der Terminus „Dokument" als Überbegriff für Skripte (textuell) und Diagramme (grafisch) verwendet. Beide Visualisierungsklassen werden von einer separaten, Anforderung 1 untergeordneten Anforderung adressiert. Die Unterordnung ist damit begründet, dass sowohl der grafische als auch der textuelle Editor über eine weitgehend gemeinsame Basis für die integrierte Unterstützung von freier und stringenter Modellierung verfügen sollen. Dies wiederum ist schon aus rein softwaretechnischen Gründen sinnvoll, weil es die Wiederverwendbarkeit gleichartiger Artefakte gewährleistet und damit für die Verringerung von Redundanzen sorgt. Vornehmlich handelt es sich bei den besagten Artefakten um Konzepte, die in einem Basis Meta-Modell zusammengefasst sind (Abschnitt 5.1.3).

Völter prognostiziert in [139], dass die Integration von grafischen und textuellen DSMLs in Zukunft zunehmend an Bedeutung gewinnt. Dadurch ist unser **Ziel** bestätigt, die **Wiederverwendung auch auf konzeptueller Ebene** konsequent durchzuziehen, wovon letztendlich die beiden Anwendergruppen (Domänenexperten und Modellierungsspezialisten) profitieren. Sie müssen nur einmal den grundsätzlichen Ansatz verstehen und können ihn zukünftig beim Wechsel der Visualisierungsform übertragen. Lediglich die spezifischen Eigenheiten des zugehörigen Editors müssen erlernt und verstanden werden. Dieser Umstand kann jedoch als gering eingestuft werden, da die verschiedenen Benutzer im Regelfall mit entsprechenden Editoren vertraut sind. Im grafischen Umfeld sind es Werkzeuge wie Microsoft Visio oder Dia. Im textuellen Kontext, der ohnehin eher für Personen mit Programmiererfahrung gedacht ist, handelt es sich um IDEs wie IntelliJ IDEA, Eclipse oder Visual Studio. Das ursprüngliche und vertraute Eingabeverfahren wird in jedem Fall beibehalten, d.h. der grafische Editor wird überwiegend mit Maus bzw. Touch-Gesten bedient, während die Steuerung des Text-Editors primär via Tastatur erfolgt.

Um von der besagten Wiederverwendung profitieren zu können, werden in Abschnitt 5.1 gemeinsame Designprinzipien ermittelt, die sowohl für die grafische als auch die textuelle Welt zutreffen. Zugleich wird darauf Wert gelegt, dass die für spezifischen Modellierungsworkflows (Abschnitte 6.2 und 7.3)

erforderlichen Rahmenbedingungen nicht verbaut werden. Voraussetzung sind **allgemeingültige Meta-Modelle**, die zur Beschreibung der konkreten Syntax beliebiger DSMLs eingesetzt werden können (Abschnitt 5.2). Ein wichtiger Aspekt dieser Meta-Modelle ist Erweiterbarkeit, da sowohl grafische als auch textuelle Modellierungssprachen jeweils unterschiedliche Eigenheiten aufweisen. Objekte in Diagrammen können im Zuge der Modellierung beispielsweise mit einem beliebigen Styling versehen werden, während textuelle Artefakte in Dokumenten üblicherweise anhand ihres Typs (z.B. Schlüsselwort, Zahl oder Zeichenkette) automatisch eingefärbt werden. Als Novum gegenüber anderen, existierenden Modellierungssystemen liegt der konkreten Syntax und den darauf basierenden Dokumenten dasselbe Meta-Modell zugrunde. Der Hauptgrund für diese Entwurfsentscheidung ergibt sich aus dem einheitlichen Abspeicherungsformat (weitere Details folgen in Abschnitt 5.1.2).

Unter Verwendung dieses Formats können Dokumente frei modelliert werden, ohne dass irgendwelche Konstrukte einer konkreten Syntax vorgegeben sind. Sie werden bei Bedarf während der freien Modellierung erzeugt und können anschließend im selben und auch in anderen Dokumenten wiederverwendet werden. Auf Basis der in Dokumenten konstruierten visuellen Artefakte kann anschließend die noch nicht vorhandene abstrakte Syntax induziert werden. Gleiches gilt für die Domänenmodelle, die aus den einzelnen Beispieldokumenten hervorgehen. Dieser **initiale Entwurf einer DSML**, der dem **Prinzip des Konstruktivismus** folgt, ist Gegenstand der Erläuterungen in Abschnitt 5.3.

Liegt bereits eine vollständige **DSML** zugrunde, so kann diese ebenfalls **mithilfe von freier Modellierung** an geänderte Anforderungen **inkrementell angepasst** werden. Eine Behandlung, welche Restriktionen hierbei im Hinblick auf die konkrete Syntax zu beachten sind und wie erfolgreich getätigte Manipulationen auf die vorhandene abstrakte Syntax übertragen werden, erfolgt in Abschnitt 5.4.

Abschließend werden in den Abschnitten 5.5 und 5.6 Wege aufgezeigt, wie eine im ontologischen Kontext entworfene **DSML optimiert** und lückenhaftes Domänenwissen kompensiert werden kann. Erforderlich macht dies das allgemeine Prinzip der Induktion, das nie gänzlich frei von falschen oder ungenauen Schlussfolgerungen ist.

5.1 Grundlegende Designprinzipien

Zur Aufstellung grundlegender Designprinzipien ist es im Vorfeld erforderlich, die **visuellen Bausteine** der beiden Dokumentarten zu kennen und im Hinblick auf Gemeinsamkeiten zu untersuchen. Skripte bestehen zunächst aus einer Aneinanderreihung von Schriftzeichen. Im Umfeld des Compilerbaus [1] werden diese Zeichen durch einen Lexer in einer Sequenz aus Tokens zusammengefasst, die schließlich als Input für die weitere syntaktische Analyse dient. Demgemäß können Tokens als die atomaren Bausteine textueller DSMLs auf Ebene der konkreten Syntax betrachtet werden. Im darauffolgenden Schritt nimmt ein Parser die erste Interpretation der Tokens vor und organisiert sie als Baumstruktur [1, 33, 153]. Diagramme hingegen setzen sich aus Formen zusammen, die auf verschiedene Weisen zueinander in Beziehung stehen (z.B. durch Verschachtelung oder Anhaften). Hierbei kann die Verschachtelung von Formen, die stets in einer Baumstruktur resultiert, ebenfalls als primäre Strukturierung angenommen werden [41, 117]. Dass für die interne Repräsentation von textuellen und grafischen Dokumenten die Organisation als Baumstruktur sehr gut geeignet ist, zeigt die HTML5 Spezifikation [63]. Sie gibt diese Art der Strukturierung für den Aufbau von Webseiten vor. Ein gutes Beispiel für die Möglichkeiten von HTML5 liefert die *Model Workbench*, die in Kapitel 8 vorgestellt wird.

Bezogen auf **grafische Sprachen** zählen Costagliola et al. in [28] die Organisation von Diagrammen als Baum zur geometriebasierten Klasse. Allgemein formuliert werden in grafischen Sprachen, die zur geometriebasierten Klasse gehören, Beziehungen zwischen Formen durch ihre räumliche Lage zueinander ausgedrückt. Das Verschachteln von Elementen stellt demnach nur eine Möglichkeit dar,

Beziehungen visuell auszudrücken, und kann deshalb intern wie jede andere Beziehung abgelegt werden. Neben dem Enthaltensein von Elementen ist beispielsweise auch das Anhaften grafischer Objekte an anderen Objekten dieser Sprachklasse zuzuordnen. Alternativ klassifizieren Costagliola et al. Diagrammsprachen als verbindungs- oder graphbasiert, wenn Beziehungen stets in Form von Verbindungslinien dargestellt werden. Die meisten grafischen Sprachen gehören jedoch der hybriden Klasse an und unterstützen damit beide Visualisierungsarten. Aufgrund der größeren Vielfalt an Darstellungsmöglichkeiten kommt die hybride Klasse für beispielgetriebene Entwicklung grafischer DSMLs (Kapitel 6) zum Einsatz. Darüber hinaus verfügt jedes visuelle Element über bestimmte grafische Eigenschaften, wie beispielsweise Hintergrundfarbe, Rahmenfarbe, Dicke des Rahmens etc.

Auf Seite der **textuellen Sprachen** gibt es neben der Verschachtelung von Elementen ebenfalls die Möglichkeit, Beziehungen zu anderen Elementen herzustellen. Dies geschieht immer unter Angabe des voll qualifizierenden Namens des betreffenden Elements. Eine andere Form der Beziehung existiert nicht. Außer dem Text selbst und der Sichtbarkeit besitzen textuelle Elemente keine spezifischen Eigenschaften. Die Formatierung zählt nicht dazu, weil diese mittels Syntaxhervorhebung durch den Typ eines Tokens vorgegeben ist und nicht feingranular vom Benutzer beeinflusst werden kann.

Generell müssen visuelle Elemente sowie deren Beziehungen und Eigenschaften weiter differenziert werden. Alle drei Artefakte können nämlich entweder **rein visueller** oder stattdessen **semantischer** Natur sein. Allgemein gesprochen bedeutet semantisch, dass es zum betreffenden Artefakt eine Entsprechung auf Domänenseite gibt, die allerdings nicht zwangsläufig im Dokumentenmodell durch ein explizites Mapping angegeben sein muss. Es muss lediglich die Möglichkeit bestehen, diese Entsprechung im Modell festzuhalten; und zwar dann, wenn eine Abbildung auf eine abstrakte Syntax erfolgen soll. Im Folgenden bedeutet also Entsprechung auf Domänenseite nur, dass eine solche Korrespondenz angedacht ist. Rein visuelle Artefakte haben hingegen nur informellen Charakter, ohne jedwede Auswirkung auf die Anwendungsdomäne. Ihre Anwesenheit ist jedoch sinnvoll, da sie einerseits ein bequemes Erstellen eigener Konstrukte zur Modellierungszeit erlauben. Andererseits können bereits vorhandene Konstrukte global und feingranular im Nachhinein modifiziert werden.

Beispiele für semantische und rein visuelle Knoten sowie semantisches und rein visuelles Enthaltensein sind in Abbildung 5-1 veranschaulicht. Der linke Teil zeigt die konkrete grafische Darstellung, während rechts die dahinterliegende Baumstruktur abgebildet ist, die sich aus Knoten und Beziehungen zusammensetzt. Dickumrandete Knoten sowie goldfarbene Enthaltensein-Beziehungen sind semantisch, was bedeutet, dass sie ebenfalls auf Domänenseite auftreten. Jeder Knoten, von dem eine semantische Beziehung ausgeht, benötigt ein Domänenelement, dem er zugeordnet ist. Andernfalls ergibt eine semantische Enthaltensein-Beziehung wenig Sinn, da kein Domänenelement angegeben ist, das als Container für andere Domänenelemente fungiert. Domänenelemente werden stets von denjenigen visuellen Objekten repräsentiert, die direkt in einem semantischen Container liegen. Eine Ausnahme bildet die Wurzel des Dokuments und somit im vorliegenden Beispiel der `Diagram`-Knoten. Damit die zuvor aufgestellte Bedingung erfüllt ist und dadurch eine Entsprechung der Enthaltensein-Beziehung auf Domänenseite Gültigkeit besitzt, muss dieser Knoten ebenfalls einem Element der Domäne zugeordnet sein. Wegen der ausgehenden semantischen Beziehung des `Content`-Knoten, handelt es sich bei ihm um einen semantischen Container, dessen zugehöriger semantischer Knoten der `Swimlane`-Knoten ist. Visualisiert ist der Container als grauer Bereich innerhalb der grünen Swimlane. Sein einziger Kindknoten stellt eine Raute (`Diamond`-Knoten) dar, die einen Kreis (`Circle`-Knoten) beinhaltet. Nachdem die Raute jedoch ein rein visueller Container ist (graue Enthaltensein-Beziehung), gibt es zu ihrem `Circle`-Kindknoten keine Entsprechung auf Domänenseite. Die Raute mit eingeschachteltem Kreis repräsentiert demnach ein zusammengehörendes Konstrukt, das nur als Ganzes einem Domänenelement zugeordnet sein kann. Dieses anhand eines Beispiels vorgestellte Konzept lässt sich auch auf alle anderen Beziehungsarten übertragen.

Abbildung 5-1: Beispiel für semantisches und rein visuelles Enthaltensein

Bei der näheren Untersuchung möglicher visueller Eigenschaften können zwei unterschieden werden, die innerhalb beider Visualisierungsformen auftreten, nämlich die **Beschriftung** und die **Sichtbarkeit** von Elementen. Ein Schriftzug kann sowohl rein visuell sein und damit im Regelfall den gleichen Text anzeigen oder er kann semantischen Charakter aufweisen und dadurch über eine Entsprechung auf Domänenseite verfügen. Im letzteren Fall ist der Text durch seine Abhängigkeit von Domänenelementen zweifelsohne variabel. Betrachtet man die Programmiersprache Java [53], so stellen beispielsweise Klammern statische Tokens dar, während die sogenannten Modifikatoren von Methoden und Feldern (*public*, *private* und *protected*) semantisch und somit variabel sind. Diese Unterscheidung kann unverändert auf die Sichtbarkeit übertragen werden. Ist die Sichtbarkeit eines Knotens als semantisch deklariert, so wird er nur dann angezeigt, wenn die entsprechende Bedingung innerhalb der Domäne erfüllt ist. Andernfalls ist die Sichtbarkeit am Element selbst oder an dessen Konstrukt festgelegt.

5.1.1 Anforderungen an gemeinsame Meta-Modelle

Damit sind die Anforderungen an ein **gemeinsames Meta-Modell zur Festlegung der Visualisierungsstruktur** recht ähnlich. Die Anforderungen werden im Folgenden als V1 bis V8 betitelt. Ein Dokument besteht stets aus Knoten (V1), die zueinander in Beziehung stehen (V2), wobei genau ein Wurzelknoten vorhanden ist (V3). Außerdem kann jeder Knoten beliebige Eigenschaften (V4) besitzen. Nachdem verschiedene Arten von Beziehungen vorkommen können (Verschachtelung, Anhaften, ausgehende Verbindungen, textueller Bezug aufgrund gleicher Benennungen etc.) ist es wichtig, dass Beziehungen typisierbar sind (gehört zu V2). Gleichzeitig muss auch unterschieden werden, ob Beziehungen rein visuellen oder semantischen Charakter aufweisen (V5). Diese Differenzierung muss ebenfalls für Knoten (V6) und Eigenschaften (V7) gewährleistet sein. Nachdem sowohl die Beschriftung als auch die Sichtbarkeit von Elementen im Grafischen und im Textuellen vorkommt, sind beide Eigenschaften direkt vom gemeinsamen Meta-Modell bereitzustellen (V8).

Ferner soll der Modellierungsspezialist in einer späteren Review-Phase die Möglichkeit erhalten, den **Meta-Modell-Generierungsprozess aktiv** zu **beeinflussen**, indem er **Elemente annotieren** kann (V9). Dadurch kann gezielt gesteuert werden, welche Artefakte beim Ableiten der abstrakten Syntax für die jeweils annotierten visuellen Konstrukte erzeugt werden. Diese Anforderung stellt im Vergleich mit Abschnitt 4.6 eine weitere Möglichkeit dar, lückenhaftes Domänenwissen zu kompensieren und wird auch von den meisten in Abschnitt 3.4 vorgestellten Arbeiten genutzt. Sie ist nur im ontologischen Kontext nutzbar, weil sie in das visuelle Meta-Modell integriert sein muss. Im linguistischen Kontext ist dieses Meta-Modell jedoch durch das LMM fixiert.

Durch die Erfüllung dieser Anforderungen lässt sich der freie, nicht aber der stringente Modellierungsmodus umsetzen. Letzterer erfordert zusätzlich eine Möglichkeit, die visuellen Konstrukte auf Elemente der Domäne abzubilden. Die dafür notwendigen Abbildungsvorschriften sollen eindeutig in Modellform spezifizierbar sein, so dass die vorhandene Infrastruktur (*Model Workbench*, Kapitel 8) zur Verarbeitung von (Meta-)Modellen hier ebenfalls eingesetzt werden kann. Zudem bietet die Verwendung eines einheitlichen Formats den Vorteil, dass sich Benutzer nur in einen Mechanismus einarbeiten

müssen. In Anlehnung an ein gemeinsames visuelles Meta-Modell wird es demnach auch ein **gemeinsames Meta-Modell zur Definition von Abbildungsvorschriften** (sogenannte Mappings) geben. Beim Ableiten einer DSML aus einer Menge frei erstellter Dokumente muss für jedes als semantisch deklariertes visuelles Konstrukt ein zugehöriges Domänenelement inklusive Mapping generiert werden. Folglich sind für alle Anforderung des vorherigen Absatzes, die eine Unterscheidung in rein visuell und semantisch verlangen, entsprechende Abbildungsvorschriften in das allgemeine Mapping Meta-Modell aufzunehmen. Betroffen sind hiervon die Anforderungen V1 (Knoten), V5 (Beziehungen) und V8 (Schriftzüge und Sichtbarkeit). Um den Bezug nicht zu verlieren, werden die entsprechenden Anforderungen auf Mapping-Seite als M1, M5, M8a und M8b bezeichnet. M8a meint ein Mapping für die Beschriftung, wohingegen M8b ein Mapping für die Sichtbarkeit fordert.

5.1.2 Modellhierarchie

Um überhaupt ontologisch modellieren zu können, muss die Möglichkeit geschaffen werden, visuelle Konstrukte auf Konstrukte der Domäne abzubilden. Welche Modelle im Einzelnen daran beteiligt sind und wie diese zueinander in Beziehung stehen, zeigt Abbildung 5-2. Die Modelle lassen sich zunächst anhand ihrer Zugehörigkeit zu bestimmten Ebenenstacks in die beiden Kategorien *Syntax Stack* und *Domain Stack* einordnen. Unter einer Ebene wird hier ein logischer Zusammenschluss von Modellen verstanden, wobei für zwei übereinander angeordnete Ebenen gilt, dass die Konzepte der unteren Ebene stets (direkt durch Instanziierung oder indirekt mittels Instanzspezialisierung) Instanzen der Konzepte der oberen Ebene sind.

Der **Syntax Stack** umfasst diejenigen Modelle, die für die Ausprägung der konkreten Syntax sowie deren Anbindung an die Anwendungsdomäne verantwortlich sind. Sie können folglich in Modelle untergliedert werden, die rein visuellen und strukturellen Belangen genügen, und solchen, die Abbildungsvorschriften auf Elemente der Domäne enthalten. Da grafische und textuelle Sprachen mit vielen strukturellen Gemeinsamkeiten aufwarten, können beide Arten auch auf demselben visuellen Meta-Modell beruhen. Die für jede Sprachart definitiv erforderlichen Erweiterungen werden als Spezialisierung dieses Meta-Modells ausgedrückt. Gleiches gilt für das Mapping Meta-Modell, welches vorgibt, wie visuelle Konstrukte auf Konstrukte der Domäne abzubilden sind.

Von zentraler Bedeutung ist, dass der **Einstiegspunkt in Dokumente** immer **über die visuellen Modelle** erfolgt. Nur auf diese Weise können sowohl freie als auch stringente Modellierung unterstützt werden, weil im Fall der freien Variante auf die Angabe von Mappings verzichtet wird. Abstrakte Syntax

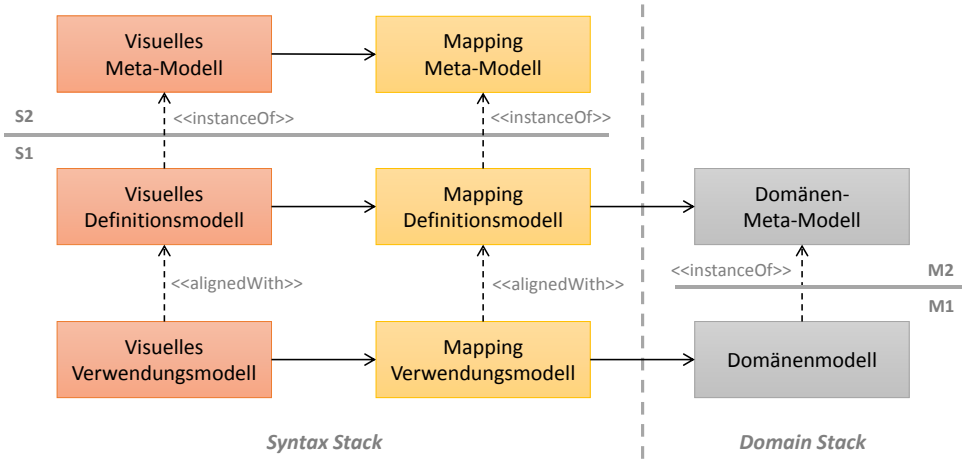

Abbildung 5-2: Modellhierarchie für ontologisches Modellieren

inklusive zugehöriger Mappings können später manuell erstellt oder – wie vom nachfolgend vorgestellten Ansatz empfohlen – weitgehend automatisch generiert werden. Der Einstieg in Dokumente über die Visualisierungsstruktur einschließlich der dreistufigen Unterteilung in Visualisierung, Mapping und Domäne findet sich auch bei BITKit [113], welches in Abschnitt 3.4.2 vorgestellt wurde.

Innerhalb des Syntax Stacks sind die zwei Definitionsmodelle Instanzen der beiden zuvor genannten Meta-Modelle. Diese *instanceOf*-Beziehung trifft gleichermaßen für die jeweils enthaltenen Konzepte zu. Visuelles Verwendungsmodell und Mapping-Verwendungsmodell befinden sich wegen der *alignedWith*-Referenz auf derselben logischen Ebene wie die Definitionsmodelle. Die Konzepte der Verwendungsmodelle **spezialisieren** jedoch die **Instanz-Facette** der Konzepte in den Definitionsmodellen (Abschnitt 2.2.1.3), weshalb erstere ebenfalls Instanzen der in den Meta-Modellen deklarierten Konzepte darstellen. Analog zu unseren Vorarbeiten in [119] wird auf diese Weise sowohl für die Syntaxdefinition als auch für die interne Darstellung von Dokumenten dasselbe Format verwendet. Kein anderes uns bekanntes Modellierungssystem verwendet einen vergleichbaren Ansatz. Typischerweise beruhen die anderen Systeme (z.B. GMF und MetaEdit+) auf separaten Meta-Modellen für die Definition der Syntax und der Dokumente. Vorteil der auf Instanzspezialisierung fußenden Vorgehensweise ist die Umsetzbarkeit einer einheitlichen API für den Zugriff auf die Syntaxdefinition sowie auf die darauf basierenden Dokumente (Verwendungen). Der Nachteil hingegen ist, dass Typen und Verwendungen über zuweisbare Attribute verfügen, die für den jeweiligen Vertreter bedeutungslos sind. Dieser Umstand ist jedoch vernachlässigbar, da die zusätzlichen Daten einfach ignoriert werden können. Ein vergleichbares Verhalten kann bei der Interpretation von CSS durch Web-Browser beobachtet werden, da diese ebenfalls ihnen unbekannte Regeln und Attribute übergehen und ausschließlich die verbleibenden Informationen auswerten [29].

Der **Domain Stack** beinhaltet alle Modelle, die zur Beschreibung der Anwendungsdomäne relevant sind. Einerseits handelt es sich hierbei um das Domänen-Meta-Modell, das die abstrakte Syntax der betreffenden DSML definiert und damit die Typen sowie deren mögliche Beziehungen zueinander deklariert. Andererseits gehören zu dem Stack auch die eigentlichen Domänenmodelle, die die Instanzen der Anwendungsdomäne inklusive zugehöriger Parametrierung widerspiegeln.

5.1.3 Definition: visuelles Konstrukt

Ein zentraler Begriff innerhalb des ontologischen Kontexts ist der des visuellen Konstrukts. Jedes Konstrukt muss in einem visuellen Definitionsmodell definiert sein, bevor es in darauf basierenden visuellen Verwendungsmodellen instanzspezialisiert und damit verwendet werden kann. Die **Intention** hinter jedem visuellen Konstrukt ist zudem, dass es eine **Entsprechung auf Domänenseite** gibt und zwar unabhängig davon, ob bereits eine abstrakte Syntax vorliegt (stringente Modellierung) oder nicht (freie Modellierung). Denn visuelle Konstrukte bilden die Vorlagen der Bauteile, aus denen die Dokumente und daraus schließlich – sobald eine abstrakte Syntax existiert – die Domänenmodelle zusammengesetzt werden. Die exakte Definition eines visuellen Konstrukts lautet wie folgt:

> *Ein visuelles Konstrukt (oder auch Konstrukt der konkreten Syntax) ist definiert als ein über rein visuelle Referenzen vernetztes Geflecht von Knoten mit genau einem Wurzelknoten als Einstiegspunkt. Der Wurzelknoten wird bereits während der Erstellung des Konstrukts dauerhaft festgelegt und hat stets semantischen Charakter. Jeder Knoten kann darüber hinaus über beliebige Eigenschaften verfügen. Dasselbe gilt auch für semantische Beziehungen, die jedoch zur Abgrenzung von anderen Konstrukten auf kein Ziel verweisen dürfen. Sämtliche Eigenschaften und Beziehungen, die ihren Ursprung in den zum Konstrukt gehörenden Knoten haben, sind Teil des Konstrukts.*

Unter einer **semantischen Beziehung** eines Konstrukts kann die Definition eines semantischen Anknüpfpunkts verstanden werden. Bei der Verwendung eines Konstrukts können dort andere Verwendungen von Konstrukten andocken. Verdeutlicht wird der Sachverhalt in Abschnitt 5.1.5 anhand eines konzeptuellen Beispiels.

Ein wichtiger Aspekt ist auch die **Festlegung des Wurzelknotens**, da dies dauerhaft gilt und deshalb nicht geändert werden darf. Detaillierte Gründe und Erläuterungen für die Notwendigkeit dieser Einschränkung folgen erst in Abschnitt 5.4.1. Dort werden die diversen Änderungsmöglichkeiten visueller Konstrukte eingehend erörtert.

5.1.4 Prinzipien von stringenter und freier Modellierung

Bei der **stringenten Modellierung** werden Modelle auf Basis einer domänenspezifischen Sprache gebaut. Im Umfeld von DSM ist dies die typische Vorgehensweise. Dabei dürfen beim Modellieren ausschließlich diejenigen visuellen Konstrukte verwendet werden, die bereits an ein Konstrukt der abstrakten Syntax gebunden sind. Zusätzlich müssen sämtliche durch diese Syntax vorgegebenen Rahmenbedingungen erfüllt werden. Primär handelt es sich hierbei um Typ und Multiplizität von Attributen.

Angenommen das Domänen-Meta-Modell umfasst die beiden Konzepte `Process` und `Task`, wobei `Process` ein optionales aber mehrwertiges Attribut `tasks` vom Typ `Task` besitzt. Diesem Attribut dürfen dann lediglich Konzepte zugewiesen werden, die `Task` instanziieren. Alle anderen Konzepte sind im stringenten Modellierungsmodus unzulässig. Wäre das Attribut darüber hinaus einwertig, dann dürfte ihm maximal eine `Task` -Instanz zugewiesen werden.

Auch die **freie Modellierung** trifft man bei der Entwicklung von Software an. Sie wird dort z.B. im Rahmen der initialen Anforderungserhebung und zur Erstellung der Dokumentation angewendet. Dabei kommen in erster Linie verbreitete Werkzeuge wie Microsoft Visio, Microsoft PowerPoint oder Dia zur Entwicklung informaler Diagramme zum Einsatz. Diese Programme bieten jedoch keine Möglichkeit die entworfenen Diagramme manuell um Informationen anzureichern, die angeben, ob die verwendeten grafischen Artefakte rein visueller oder semantischer Natur sind. Benutzer müssten also ganz auf einen Mechanismus (wie beispielsweise in [5] geschildert) vertrauen, der diese Klassifikation automatisiert vornimmt. Das kann dazu führen, dass die Diagramme falsch interpretiert werden und dadurch eine unpassende, die Anwendungsdomäne verfehlende DSML resultiert. Um dieser Problematik Einhalt zu gebieten, ist es sinnvoll, wenn der Benutzer bereits während der freien Modellierung über die Möglichkeit verfügt, jedes **visuelle Artefakte** explizit als **rein visuell oder semantisch** zu deklarieren. Das bedeutet aber nicht, dass der Benutzer diese Angabe für jedes Artefakt unbedingt machen muss und damit ständig in seinem Workflow unterbrochen wird. Die Erkennung soll primär von einem Automatismus erledigt werden. Der Benutzer kann allerdings eingreifen, sofern er mit dem Ergebnis der Klassifikation nicht zufrieden ist. Demzufolge muss direkt im Dokument und darin für jedes visuelle Artefakt hinterlegt werden können, ob es beim Ableiten der abstrakten Syntax als rein visuell oder semantisch zu betrachten ist. Derartige Angaben erfolgen stets auf Definitionsebene und gelten dann für alle Verwendungen der betreffenden Artefakte. Ein Vorteil davon ist, dass bereits definierte visuelle Konstrukte mühelos wiederverwendbar sind. Diese Wiederverwendbarkeit kann beispielsweise bei textuellen Editoren dazu verwendet werden, um Content Assist Funktionalität anzubieten. Aber auch im Grafischen ist Instanzspezialisierung ein wertvolles Muster. Hier können die visuellen Konstrukte in einer Palette abgelegt werden und vom Benutzer dem Diagramm in Form weiterer Verwendungen (z.B. via Drag & Drop) hinzugefügt werden.

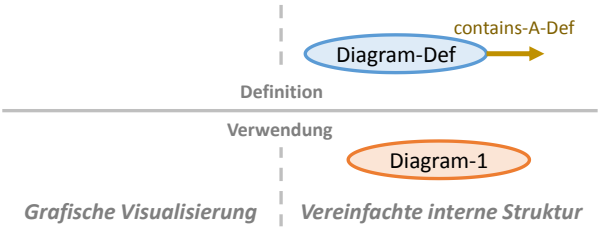

Abbildung 5-3: Beispiel für freies ontologisches Modellieren (Schritt 1, leeres Diagramm)

5.1.5 Konzeptuelles Beispiel für freie Modellierung

Im Folgenden werden anhand eines Beispiels die zuvor beschriebenen grundlegenden Designprinzipien verdeutlicht (Abbildung 5-3 bis Abbildung 5-7). In ihnen sind die unterschiedlichen Zustände eines Diagramms im Laufe seiner Entstehung und unter Einsatz der freien Modellierung visualisiert. Aus Übersichtlichkeitsgründen wird auf die Anzeige der Instanzspezialisierungsbeziehungen verzichtet und stellvertretend die Beziehung durch entsprechende Benennung angegeben. Jeweils die vorderen Teile eines Namens sind hierbei von Interesse (d.h. diejenigen Zeichen vor dem letzten Bindestrich). Wenn diese Namensbestandteile von einem Element des Definitionsmodells und einem anderen Element des Verwendungsmodells übereinstimmen, dann spezialisiert letzteres Element das Erstgenannte (z.B. `Diagram-1` als Instanzspezialisierung von `Diagram-Def` sowie `contains-B-1` als Instanzspezialisierung von `contains-B-Def`).

Den Anfang bei der freien Modellierung machen immer mindestens ein visuelles Definitions- und ein visuelles Verwendungsmodell, wobei ersteres eine leere Dokumentdefinition und letzteres eine zugehörige Verwendung enthält. Nachdem es sich um ein Beispiel handelt, das der grafischen Visualisierungsform folgt, kommen in den Modellen ausschließlich Bezeichnungen aus der grafischen Welt vor. Die Dokumentdefinition ist somit als `Diagram-Def` betitelt und verfügt standardmäßig über eine semantische Enthaltensein-Definition, die in Form des goldfarbenen `contains-A-Def`-Pfeils abgebildet ist (Abbildung 5-3). Im Übrigen werden alle semantischen Beziehungen als dicke, goldfarbene Pfeile dargestellt.

Zum gegenwärtigen Zeitpunkt gibt es noch kein grafisches Konstrukt, das im Diagramm wiederverwendet werden kann. Möchte man jetzt eine beliebige grafische Form dem Diagramm hinzufügen, so muss diese durch ein grafisches Meta-Modell zur Verfügung gestellt werden. Bei einem grafischen Meta-Modell handelt es sich um eine Erweiterung des allgemeinen visuellen Meta-Modells aus Abschnitt 5.2.1. An der Stelle wird davon ausgegangen, dass ein derartiges Meta-Modell schon existiert und dieses die Vorlagen *Raute*, *Kreis* und *Pfeil* bereitstellt.

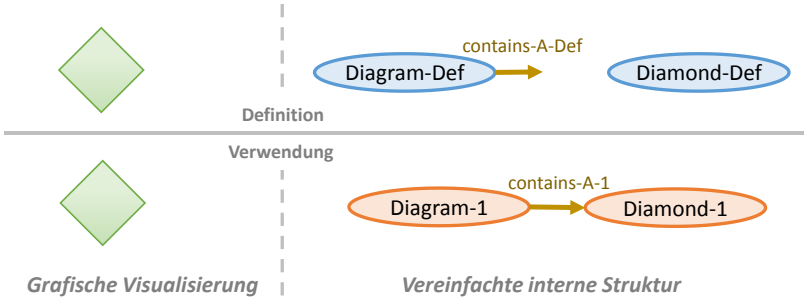

Abbildung 5-4: Beispiel für freies ontologisches Modellieren (Schritt 2, Erstellen einer Raute)

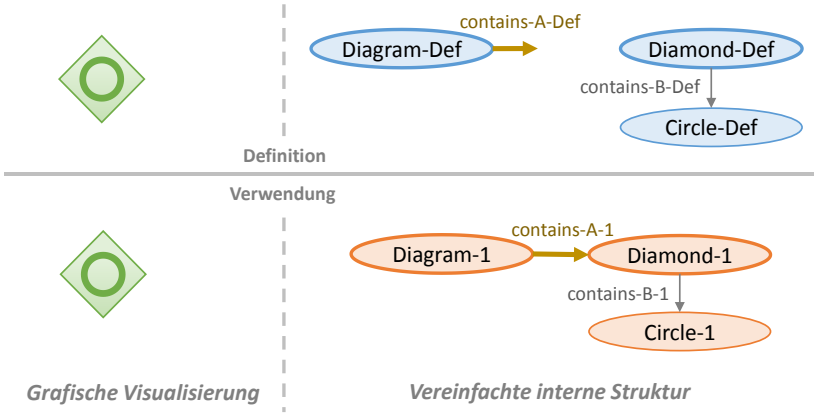

Abbildung 5-5: Beispiel für freies ontologisches Modellieren (Schritt 3, Erweiterung der Raute)

Im Beispiel wird als nächstes eine Raute (`Diamond-1`) in das Diagramm eingefügt, zu der gemäß der obigen Beschreibung gleichzeitig eine Definition (`Diamond-Def`) angelegt wird (Abbildung 5-4). Gleichsam wird im Verwendungsmodell die Enthaltensein-Beziehung `contains-A-1` erzeugt, die `contains-A-Def` spezialisiert und demzufolge semantischer Natur ist. Definitionen von Beziehungen fungieren lediglich als Platzhalter und verfügen nicht über einen Zielknoten. Der zu `contains-A-Def` gehörende Pfeil in Abbildung 5-4 zeigt somit ins Leere. Dies gilt gleichermaßen für alle Beziehungsdefinitionen in den kommenden Grafiken. Nachdem mit `Diamond-Def` ein neuer Knoten erstellt und dessen Verwendung `Diamond-1` von einer semantischen Beziehung referenziert wird, stellt er die Wurzel eines neuen grafischen Konstrukts dar. Folglich handelt es sich bei ihm um einen semantischen Knoten, der ein entsprechendes Element auf Domänenseite repräsentiert. Zur Abhebung von rein visuellen Knoten sind solche Knoten mit einer dicken Umrandung gekennzeichnet. Eine Sonderstellung nimmt hierbei der Knoten `Diagram-Def` ein, der zwar als Dokumentenwurzel semantischer Natur und damit dick umrandet ist, aber keinen Wurzelknoten eines eigenständigen Konstrukts widerspiegelt.

Im nächsten Schritt wird die soeben erstellte Raute um einen eingeschachtelten Kreis erweitert (Abbildung 5-5), was analog zur vorherigen Vorgehensweise abläuft. Es werden also gleichzeitig eine Kreis-definition `Circle-Def` sowie eine darauf basierende Verwendung `Circle-1` angelegt und über eine Enthaltensein-Beziehung an `Diamond-Def` bzw. `Diamond-1` gebunden. Die zugehörige Definition der Beziehung wird als rein visuell deklariert. Die Erläuterung, wie zwischen beiden Deklarationsarten unterschieden werden kann, ist spezifisch für den Modellierungsworkflow der jeweiligen Visualisierungsformund folgt daher erst in den betreffenden Kapiteln.

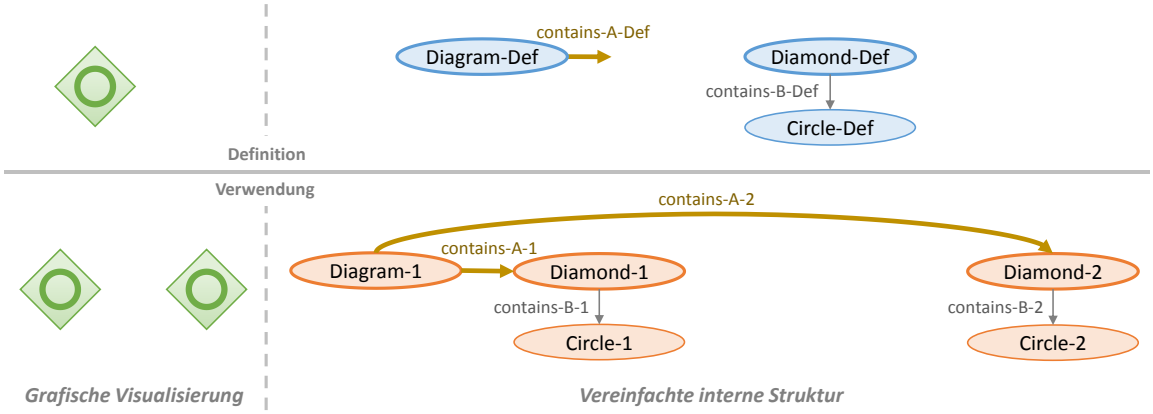

Abbildung 5-6: Beispiel für freies ontologisches Modellieren (Schritt 4, Wiederverwendung der Rautendefinition)

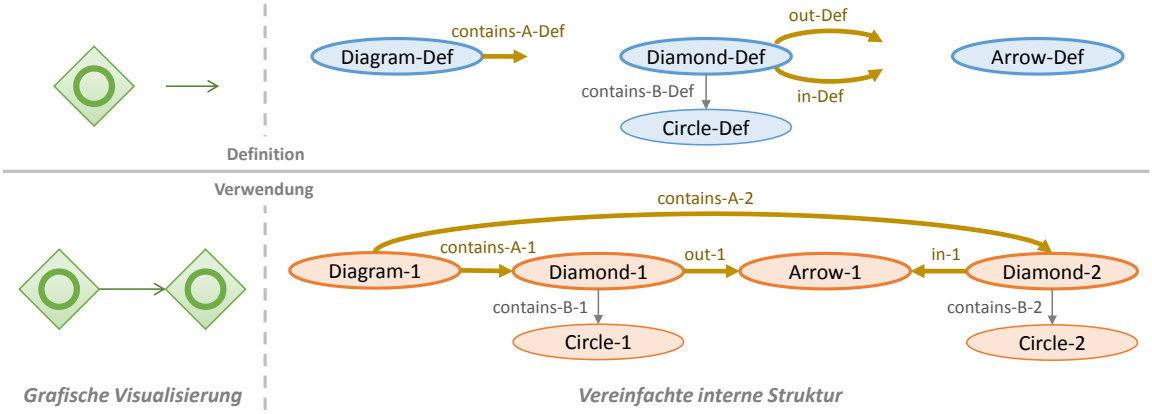

Abbildung 5-7: Beispiel für freies ontologisches Modellieren (Schritt 5, Erstellen eines Pfeils)

Anschließend wird eine zweite Raute hinzugefügt, wobei die bereits existierende Rautendefinition Wiederverwendung findet (Abbildung 5-6). Dazu sind lediglich die vier Verwendungen `contains-A-2`, `Diamond-2`, `contains-B-2` und `Circle-2` anzulegen, da sämtliche zugrundliegenden Definitionen bereits vorliegen. Der Benutzer muss hierbei explizit angeben, dass eine vorhandene Definition wiederverwendet werden soll.

Als letzter Modellierungsschritt wird ein Pfeil zwischen erster und zweiter Raute eingefügt (Abbildung 5-7). Innerhalb des Verwendungsmodells wird dies durch die Beziehungen `out-1` und `in-1` sowie durch den Knoten `Arrow-1` ausgedrückt. Da noch keine zugehörige Definition vorhanden ist, wird sie zusammen mit zwei in `Diamond-Def` entspringenden Beziehungen angelegt. Die Beziehung `out-Def` symbolisiert die Definition der Pfeilquelle, wohingegen `in-Def` die Definition der Pfeilsenke widerspiegelt. Beide sind der Rautendefinition zugeordnet, weil auf Verwendungsseite der Pfeil sowohl bei einer Raute beginnt als auch in eine Raute mündet.

5.2 Allgemeingültige Meta-Modelle für konkrete Syntax und Mapping

Gemäß Abschnitt 5.1.2 gibt es sowohl für die konkrete Syntax als auch für die Beschreibung der Mapping-Möglichkeiten ein eigenständiges Meta-Modell. Bei ersterem handelt es sich um das visuelle Meta-Modell, während letzteres als Mapping Meta-Modell bezeichnet wird. Daher ist dieser Abschnitt in zwei Unterabschnitte gegliedert, die jeweils ein Meta-Modell detailliert beleuchten. Der **Inhalt beider Meta-Modelle** beruht auf der **typischen Funktionalität existierender Werkzeuge**, die domänenspezifisches Modellieren ermöglichen (z.B. GMF, MPS und MetaEdit+). Wert gelegt wird bei der Spezifikation dieser Meta-Modelle einerseits darauf, dass die resultierenden Strukturen für grafisches textuelles Modellieren gleichermaßen geeignet sind. Andererseits müssen die Meta-Modellelemente aufgrund des tief integrierten Prinzips der Instanzspezialisierung sowohl für Konstrukte der konkreten Syntax als auch deren Ausprägungen innerhalb von Dokumenten anwendbar sein.

In den Abbildungen der Meta-Modelle ist farblich gekennzeichnet, in welchen Modellarten (Definition und/oder Verwendung) die jeweiligen Bestandteile verwendet werden dürfen. *Schwarz* ist Standard und besagt, dass ein Einsatz in beiden Modellarten zulässig ist. *Blau* schränkt die Benutzung auf das Definitionsmodell ein, während *orange* eine Limitierung auf Verwendungsmodelle bedeutet.

Als **Erweiterungsmechanismus** wird auf die **Typspezialisierung** der in den Meta-Modellen deklarierten Konzepte gesetzt, die somit bei der Konzeption der beispielgetriebenen Entwicklung von grafischen und textuellen DSMLs zum Einsatz kommt (Kapitel 6 und 7). Dieses Vorgehen hat sich in der Modellierungs- und Programmierpraxis als zweckdienlich erwiesen [106, 125].

5.2.1 Visuelles Meta-Modell

Ausgehend vom freien Modellierungsmodus liegt der Fokus zunächst auf dem visuellen Meta-Modell. Neben dem eigentlichen Inhalt dieses Meta-Modells zeigt Abbildung 5-8 auch die Verbindung zum Mapping Meta-Modell, das allerdings erst im nachfolgenden Abschnitt Gegenstand der Erläuterung ist. Im Wesentlichen handelt es sich bei den **visuellen Modellen** um sogenannte **gerichtete, attributierte Graphen** [36], deren Knoten von Instanzen des Konzepts `Node` repräsentiert werden. Dass die Graph-Datenstruktur zur kombinierten Umsetzung von freier und stringenter Modellierung gut geeignet ist, haben bereits die Autoren von DiaMeta (Abschnitt 3.4) und MLCBD (Abschnitt 3.4.3) dargelegt. Knoten entsprechen den eigentlichen visualisierten Elementen (Erfüllung von V1) und können über das `semantic`-Feld als semantisch deklariert werden (Erfüllung von V6). Dieses Feld legt auch fest, ob ein Knoten die Wurzel eines visuellen Konstrukts darstellt, wobei hiervon die Dokumentenwurzel ausgenommen ist.

Die **Kanten** des Graphen werden mithilfe des Konzepts `Reference` modelliert (Erfüllung von V2). Durch die Bildung von Typspezialisierungen können unterschiedliche Arten bidirektionaler Referenzen realisiert werden. Ein klassischer Vertreter hierfür ist die Enthaltensein-Beziehung. Aufgrund ihrer Gültigkeit im Textuellen und Grafischen ist sie bereits im allgemeinen Meta-Modell in Form des Konzepts `Containment` enthalten. Zudem kann `Reference` selbst instanziiert und damit für einfache, nicht näher spezifizierte Referenzen eingesetzt werden.

Das **Attributieren** des Graphs erfolgt über Instanzen des Konzepts `Property`. Nachdem das Konzept als abstrakt deklariert ist, können besagte Instanzen ausschließlich von Spezialisierungen gebildet werden. Spezialisierungen letztlich dazu, die Intention einer Eigenschaft festzulegen und gleichzeitig anzugeben, welche Art von Werten unterstützt und wie diese in Modellform abgelegt werden. Im Grunde können dadurch Knoten mit beliebigen Schlüssel-Wert-Paaren versehen werden (Erfüllung von V4). Bedingt durch Anforderung V8 werden die Konzepte `Label` und `Visibility` allgemein zur Verfügung gestellt. Labels können durch das Attribut `value` einen beliebigen String aufnehmen, während Instanzen von `Visibility` mithilfe von `value` durch einen booleschen Wert angeben können, ob der zugehörige Knoten angezeigt oder ausgeblendet wird.

Eigenschaften und Referenzen sind zunächst immer auf Definitionsebene zu spezifizieren, d.h. dort muss festgelegt werden, über welche Features ein Knotentyp in den Verwendungsmodellen verfügen darf. Hierbei ist zugleich das boolesche Feld `semantic` zu setzen. Es konstituiert, ob ein Feature semantischen oder rein visuellen Charakter besitzt. Dadurch, dass dieses Feld sowohl in Eigenschaften als auch in Referenzen verfügbar ist, erfüllt es die visuellen Anforderungen V5 und V7.

Der **Einstiegspunkt** des visuellen Meta-Modells ist durch das Konzept `Document` definiert, das sowohl ein Skript als auch ein Diagramm widerspiegeln kann (Einleitung zu Kapitel 5). Jedes Dokument verfügt mit root über genau einen Wurzelknoten (Erfüllung von V3). In den Abbildungen aus Abschnitt 5.1.5 repräsentieren `Diagram-Def` bzw. `Diagram-1` genau diese Wurzelknoten, und zwar einerseits auf Definitions- und andererseits auf Verwendungsebene.

Als Einstiegspunkt ist das Dokument auch dafür zuständig, die in Abschnitt 5.1.2 vorgestellte **Modellhierarchie** zu **konstituieren**. Dazu deklariert es drei Pointer-Attribute, denen je nach Modellart (visuelles Definitions- oder visuelles Verwendungsmodell) eine andere Bedeutung zugrunde liegt. Im

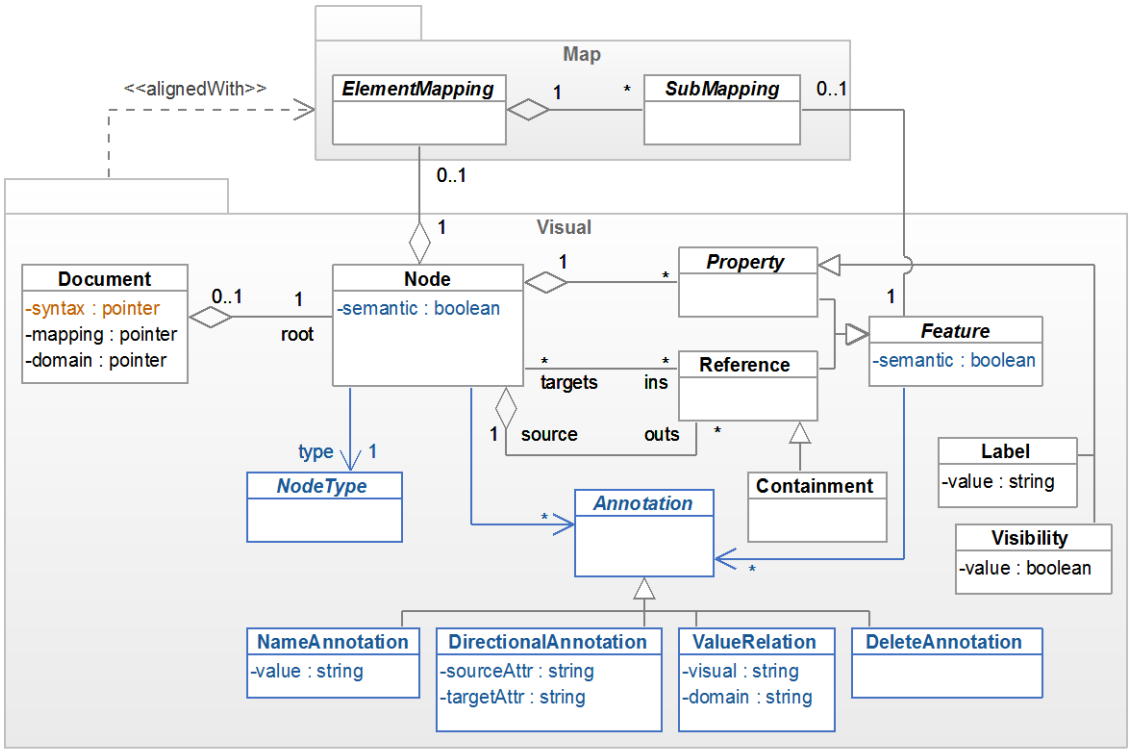

Abbildung 5-8: Allgemeines visuelles Meta-Modell der konkreten Syntax mit Anbindung des Mapping Meta-Modells

Definitionsmodell wird `syntax` ignoriert, während `mapping` auf das Mapping-Definitionsmodell und `domain` auf das Domänen-Meta-Modell zeigt. Im Verwendungsmodell dagegen referenziert `syntax` das visuelle Definitionsmodell, `mapping` das Mapping-Verwendungsmodell und `domain` das Domänenmodell.

Die von `Node`, `Reference` und `Property` gebotene Flexibilität legt bereits den Grundstein für die generische Anwendbarkeit und Erweiterbarkeit des vorgestellten Meta-Modells. Wesentlich trägt hierzu auch das Konzept `NodeType` bei. In Kombination mit `Node` implementiert es das sogenannte Type Object Pattern [71]. Es erlaubt die **dynamische Typänderung zur Laufzeit**, indem einem Knoten ein anderer Knotentyp zugewiesen wird. Dies ist insbesondere während der freien Modellierung textueller DSMLs von Bedeutung, weil dort beispielsweise ein Bezeichner problemlos in ein Schlüsselwort konvertiert werden kann. Aber auch auf grafischer Seite bietet dieses Muster seinerseits Vorteile. Zum Beispiel kann durch den Typ eines Knotens gesteuert werden, ob es sich um einen Vertex oder eine Kante handelt. Die konkrete Typisierung erfolgt also stets durch Instanzen von Spezialisierungen des `NodeType`-Konzepts. Welche Instanzen davon im jeweiligen Dokument zur Verfügung stehen, ist abhängig von der Visualisierungsform.

Zur Erfüllung von Anforderung V9 kann jeder Knoten mit beliebigen **Annotationen** versehen werden (Konzept `Annotation`), die lediglich Auswirkungen auf die Generierung der abstrakten Syntax haben und somit die Visualisierung nicht beeinflussen. Folglich werden sie nur innerhalb des visuellen Definitionsmodells ausgewertet, nicht aber in Verwendungsmodellen.

So kann mit `NameAnnotation` der Name eines Meta-Konzepts oder Attributs innerhalb der abstrakten Syntax vorgeben werden. Mithilfe von `DirectionalAnnotation` kann man festlegen, ob für eine Referenz ein unidirektionales oder zwei bidirektionale Attribute erzeugt werden. Durch eine entsprechende Parametrierung der String-Attribute `sourceAttr` und `targetAttr` kann das

Verhalten gesteuert und damit festgelegt werden, ob Attribute innerhalb des Quell- bzw. Zielkonzepts zu generieren sind und wie diese heißen.

Mithilfe der dritten Annotation `ValueRelation` kann explizit ein Wert der Visualisierung einem Wert der Domäne zugeordnet werden. Von Relevanz in der grafischen Welt ist das beispielsweise, um Wertentsprechungen für eine Menge von Farben spezifizieren zu können. Doch auch im Textuellen ist die Angabe derartiger **Wertebeziehungen** nicht ausgeschlossen (z.B. bei Abbildung von in Labels gespeichertem Text auf diskrete Zahlen in domänenseitigen Zuweisungen). Deshalb ist die `ValueRelation`-Annotation im gemeinsamen visuellen Meta-Modell deklariert.

Die Beseitigungs-Annotation (Konzept `DeleteAnnotation`) dient schließlich als **Löschindikator** für einen Knoten oder ein Feature. Das betreffende Artefakt wird dann beim nächstmaligen Durchführen des Ableitungsprozesses vollständig gelöscht. Wieso dieser auf den ersten Blick unnötige Umweg erforderlich ist, wird in Abschnitt 5.4.1 erklärt.

5.2.2 Mapping Meta-Modell

Soll nun durch Vorgabe einer abstrakten Syntax (Domänen-Meta-Modell) im stringenten Modus modelliert werden, muss eine Abbildung zwischen den in dieser Syntax enthaltenen Konstrukten und den Knoten des visuellen Meta-Modells etabliert werden. Dazu kann jeder Knoten über ein `ElementMapping` verfügen, das auf ein Konstrukt der abstrakten Syntax verweist. Um auch Mappings für einzelne Features, die ein Knoten umfasst, angeben zu können, kann jedem Feature ein `SubMapping` zugeteilt werden.

Als klassisches Beispiel für einen Knoten, der selbst einem Element der Domäne entspricht und gleichzeitig ein Feature mit Domänenbezug besitzt, gilt die Wurzel. Bei dem betreffenden Feature handelt es sich um die Enthaltensein-Beziehung. Im stringenten Modus benötigt der Wurzelknoten also zum einen ein Element-Mapping, das ihn auf ein bestimmtes Domänenelement abbildet. Zum anderen muss er über die Definition einer Enthaltensein-Beziehung (in Form einer `Containment`-Instanz) verfügen, die den Sachverhalt des Enthaltenseins mittels einer geeigneten Typspezialisierung von `SubMapping` auf die Domäne überträgt. Zu diesem Zweck existiert das `OntReferenceMapping`, dessen genaue Funktionsweise in Unterabschnitt 5.2.2.2 beschrieben wird.

Die Möglichkeiten zur Spezifikation von Mappings sind in Form eines weiteren Meta-Modells festgelegt, das in Abbildung 5-9 dargestellt ist. Dieses Meta-Modell gliedert sich in das allgemeine `Map`-Modul und in das für die ontologische Visualisierung vorgesehene `OntologicalMap`-Modul. Der Grund für die Einteilung ist, dass neben der ontologischen Visualisierung auch eine linguistische denkbar ist. Dazu müssen spezifische linguistische Mappings bereitgestellt werden, die jedoch wegen des generischen Ansatzes als Typspezialisierungen der Konzepte des `Map`-Moduls umgesetzt werden können. Auf diese Weise wird die Möglichkeit geschaffen, dieselben Editoren, die ursprünglich für die ontologische Modellierung vorgesehen waren, auch für die linguistische Modellierung zu nutzen. Ferner kann dadurch die konkrete Syntax der LML mit vergleichsweise geringem Aufwand an spezielle Bedürfnisse angepasst werden. Als Alternative zu der in Abschnitt 2.2.3 vorgestellten textuellen Repräsentation könnte dann ebenso eine grafische Visualisierung definiert werden, die beispielsweise an Klassen- und Objektdiagramme der UML angelehnt ist. Dies ist aber nicht Gegenstand der vorliegenden Arbeit, weshalb die Definition eines linguistischen Mapping Meta-Modells nicht weiter vertieft wird.

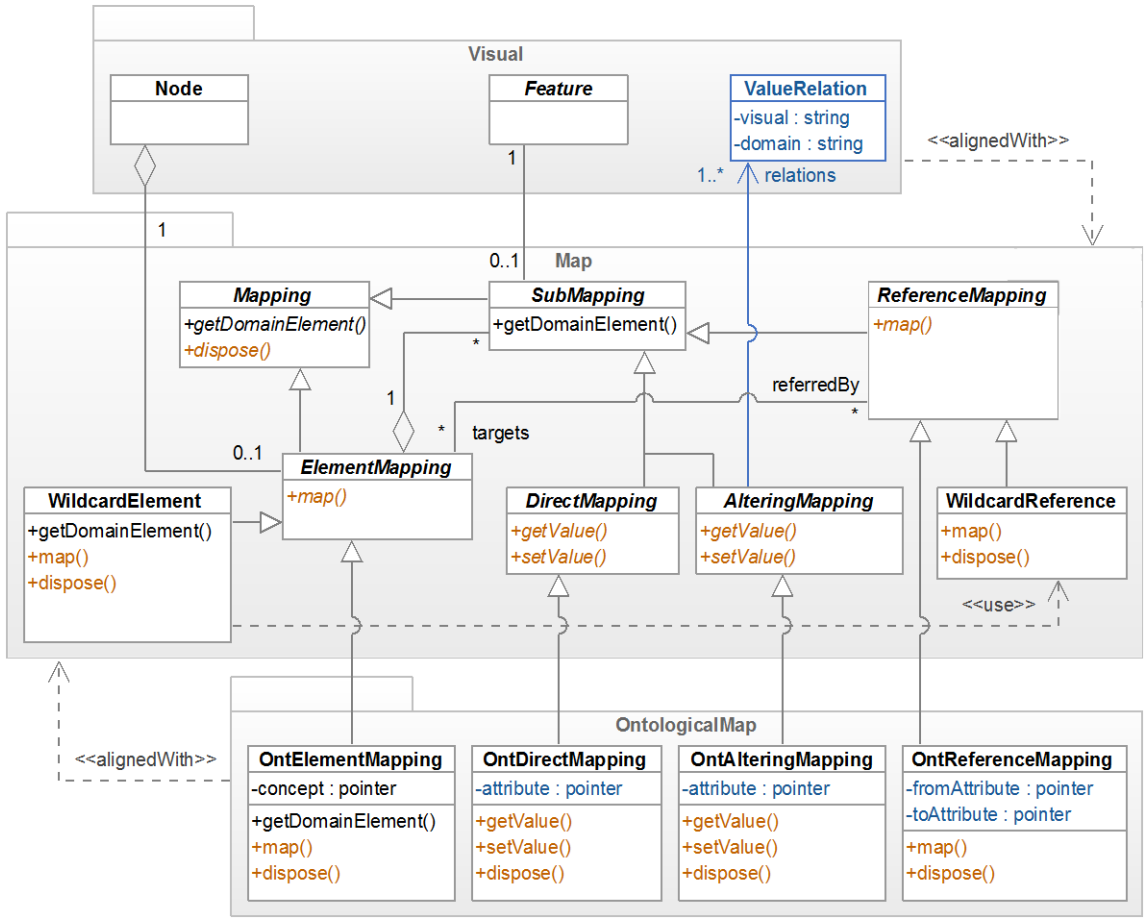

Abbildung 5-9: Allgemeines und ontologisches Mapping Meta-Modell der konkreten Syntax mit Anbindung an visuelles Meta-Modell

5.2.2.1 Das allgemeine Map-Modul

Grundsätzlich können zwei Arten von Mappings unterschieden werden, nämlich **Element-Mapping** und Sub-Mapping. Das Erstgenannte zeichnet verantwortlich für die Abbildung eines Knotens auf ein Element der Domäne (Erfüllung von M1). Unter einem Element wird im Regelfall ein Konzept verstanden, doch das muss nicht zwangsläufig der Fall sein. Soll linguistisch modelliert werden, dann ist eine Abbildung auch auf alle anderen Typen von Modellelementen (Ebenen, Pakete, Attribute etc.) zu erlauben. Die `getDomainElement()` Methode kann demnach nicht allgemein bereitgestellt werden, was die Implementierung durch spezialisierte Element-Mappings impliziert. Dasselbe trifft für die Methoden `map()` und `dispose()` zu, die als Antagonisten agieren. Während `map()` für die Etablierung des Mappings und daher für die Erzeugung des betreffenden Domänenelements zuständig ist, kümmert sich `dispose()` um die Auflösung des Mappings und damit um die Destruktion des zugeordneten Elements.

Daneben liegen die **Sub-Mappings**, die stets zu genau einem Element-Mapping gehören und diesem untergeordnet sind. Ein Sub-Mapping bezieht sich auf ein bestimmtes Artefakt, das Teil des vom zugehörigen Element-Mapping referenzierten Domänenelements ist. Üblicherweise handelt es sich dabei um Attribute oder Zuweisungen eines Domänenkonzepts. Um welches Domänenelement es sich handelt, kann auch bei Sub-Mappings mittels `getDomainElement()` abgefragt werden. Allerdings gibt es keine generische Methode, die die Etablierung des Mappings vornimmt. Der Grund dafür ist, dass sich das jeweilige Parameterprofil und somit auch der Aufrufkontext bei jeder Art von Sub-Mapping unterscheiden. Beispielsweise erwarten Schriftzug-Mappings Text, wohingegen Referenz-

Mapping das zu referenzierende Element-Mapping einfordern. Anders sieht es mit der Methode `dispose()` aus, die für die Destruktion von vorhandenen Mappings zuständig ist (d.h. ihr Aufruf beseitigt die zugehörigen Artefakte innerhalb des Domänenmodells). Sie ist in abstrakter Form in `Mapping` deklariert und muss in den jeweiligen Typspezialisierungen implementiert werden. Sie erfordert keine Parameter, da alle benötigten Informationen bereits im Modell vorliegen und dort ausgelesen werden können.

Das allgemeine `Map`-Modul enthält bereits drei spezialisierte Sub-Mappings, nämlich `DirectMapping`, `AlteringMapping` und `ReferenceMapping`. Die Zuständigkeit des `DirectMapping` liegt in der direkten Abbildung eines potentiell unendlichen Wertebereichs von Visualisierung auf Domäne und umgekehrt. Zu dem Zweck bietet es die Methoden `getValue()` und `setValue()`, deren Funktionalität von den jeweiligen Typspezialisierungen bereitzustellen ist. Ein typischer Anwendungsfall ist das Mapping eines Labels und damit von beliebigem Text (Erfüllung von M8a). Dasselbe Mapping lässt sich jedoch auch für die Abbildung semantischer Sichtbarkeit einsetzen (Erfüllung von M8b).

Aufgabe des `AlteringMapping` ist unter anderem die Unterstützung des Falls, die Sichtbarkeit invers abbilden zu können. Mithilfe assoziierter `ValueRelations` kann **angegeben** werden, **welche Werte auf Visualisierungsseite domänenseitigen Werten entsprechen**. Beim Beispiel der Sichtbarkeit kann unter Verwendung von zwei `ValueRelations` spezifiziert werden, dass, wenn das betreffende Element angezeigt wird, im Domänenmodell der Wert `false` gesetzt wird. Wenn das betreffende Element ausgeblendet ist, wird der Wert innerhalb der Domäne auf `true` eingestellt. Zur Umsetzung des konkreten Mappings existieren auch hier wieder die beiden Methoden `getValue()` und `setValue()`, welche von den jeweiligen Typspezialisierungen zu implementieren sind. Bei den von diesen Methoden zurückgegebenen bzw. entgegengenommenen Werten handelt es sich um die Werte der Visualisierung. Die Werte der Domäne sind nach außen hin verborgen, da sämtliche Mappingentitäten aus dem Visualisierungskontext angesprochen werden.

Die **Wiederverwendung des Konzepts** `ValueRelation` ist beabsichtigt. Wird eine Sprache nach dem Bottom-Up Prinzip erzeugt, so können vorhandene `ValueRelations` direkt übernommen werden, denn die erforderlichen Angaben sind sowohl im freien als auch im stringenten Modus identisch. Soll beispielsweise im freien Modus festgelegt werden, dass das Verbergen eines visuellen Elements semantischer Natur ist und auf Domänenseite dem booleschen Wert `true` entspricht, dann ist dies in Form einer `ValueRelation` anzugeben. Diese Wertebeziehung kann im Verlauf des Bottom-Up Algorithmus direkt auf das erforderliche `AlteringMapping` transferiert werden.

Als drittes Sub-Mapping ist `ReferenceMapping` für die **Abbildung von visualisierten Beziehungen auf einfache Referenzen zwischen Domänenelementen** verantwortlich (Erfüllung von M5). Das Quellelement wird dabei vom Domänenelement des übergeordneten Element-Mappings repräsentiert, wohingegen die Zielelemente durch die Domänenelemente der `targets` angegeben werden. Die Etablierung des Mappings findet in der Methode `map()` statt. Sie erwartet die Übergabe des Zielelement-Mappings, zu dem eine Referenz erstellt werden soll. Analog zu den bisher beschriebenen Mappings erfolgt die Implementierung dieser Methode innerhalb der Typspezialisierungen. Als typischer Vertreter für eine einfache Referenz auf Domänenseite, die mittels eines `ReferenceMapping` abgebildet werden kann, gilt die Enthaltensein-Beziehung. Die Quellelemente stellen hierbei die Eltern dar, während die Zielelemente von den Kindern repräsentiert werden.

Sämtliche **Methoden** – mit Ausnahme von `getDomainElement()` – sind ausschließlich **für den Einsatz auf Verwendungsebene vorgesehen** und daher in *orange* gedruckt. Sie erwarten stets eine zugrundeliegende Mapping-Definition, die auf ein Artefakt des Domänen-Meta-Modells verweist.

Anhand der Definition werden beim jeweiligen Methodenaufruf die entsprechenden Domänen-Modellelemente erzeugt, modifiziert oder gelöscht. Der Getter für das Anfordern des Domänenelements dagegen kann sowohl bei Definitionen als auch bei Verwendungen aufgerufen werden. Im ersten Fall liefert er das zugeordnete Meta-Element der abstrakten Syntax zurück, während im zweiten Fall mit der konkreten, im Domänenmodell liegenden Instanz geantwortet wird.

Einen **Spezialfall** stellen die beiden **Wildcard-Mappings** `WildcardElement` und `WildcardReference` dar. Genau genommen handelt es sich bei ihnen nicht um Mappings, da sie mit keinem Domänenelement assoziiert werden können. Als Platzhalter dienen sie stattdessen dem Zweck, vielschichtige visuelle Strukturen auf einfache Attribute und Wertzuweisungen innerhalb der Domäne abzubilden. Nähere Details und Beispiele dazu folgen im Kontext der grafischen DSMLs (Abschnitt 6.1.2), weil erst dort die notwendigen Hintergrundinformationen (primär bezogen auf Erweiterungen des visuellen Meta-Modells für grafische Sprachen) für ein tieferes Verständnis der Wildcard-Mappings vermittelt werden. Nachdem diese Platzhalter auch im Textuellen ihre Relevanz offenbaren (Abschnitt 7.7.2), sind sie Bestandteil des allgemeinen Mapping Meta-Modells. Vorneweg sei angemerkt, dass Wildcard-Mappings beim allgemeinen Ableitungsprozess der abstrakten Syntax nicht berücksichtigt werden. Dies liegt daran, dass sich deren Einsatzweisen im grafischen und im textuellen Kontext stark voneinander unterscheiden.

5.2.2.2 Das ontologische Map-Modul

Aufgrund der zahlreichen gemeinsamen Interna grafischer und textueller DSMLs, können Typspezialisierungen der zuvor vorgestellten Mapping-Konzepte definiert werden, die für beide Visualisierungsformen Gültigkeit besitzen. Diese Spezialisierungen sind im `OntologicalMap`-Modul abgelegt. Das `OntElementMapping` dient dabei der Abbildung eines beliebigen Konzepts, das mittels `concept`-Attribut in Form eines Pointers angegeben wird, auf einen Knoten im Visualisierungsmodell. Damit verkörpert das assoziierte Konzept ein Element des Domänen-Meta-Modells bzw. des Domänen-Modells (je nach Auftreten innerhalb des Definitions- bzw. Verwendungsmodells).

Des Weiteren gibt es ein spezielles Direct-Mapping namens `OntDirectMapping`. Es referenziert über das `attribute`-Feld ein Attribut des Domänenelements, welches durch das übergeordnete `OntElementMapping` festgelegt ist. Angenommen das Element-Mapping definiert eine Abbildung für ein Konzept `Task` und `Task` deklariert ein String-Attribut mit dem Namen `title`, dann kann mittels Aufruf von `getValue()` bzw. `setValue()` der Wert dieses Attributes gelesen bzw. geschrieben werden. Vornehmlich wird der Aufruf vom zugrundeliegenden visuellen Knoten getätigt, der z.B. durch einen Notifikationsmechanismus von einer Modelländerung in Kenntnis gesetzt wurde.

Das `OntAlteringMapping` basiert ebenfalls auf einem Attribut des zugrundeliegenden Domänenelements, welches wiederum durch ein Pointer-Feld namens `attribute` festgelegt ist. Die Methode `getValue()` liest den Wert aus dem angegebenen Attribut aus und übersetzt ihn durch Anwendung einer passenden `ValueRelation` in die visuelle Entsprechung. Die Methode `setValue()` funktioniert genau invers, d.h. sie nimmt einen Wert der Visualisierung entgegen, übersetzt ihn dann mithilfe einer `ValueRelation` in einen Wert der Domäne und weist diesen schließlich dem betreffenden Attribut zu.

Als vierte, allgemeingültige ontologische Spezialisierung existiert `OntReferenceMapping` für die Abbildung einfacher ontologischer Referenzen. Ausprägungen solcher Referenzen liegen im LMM als Zuweisungen von Konzepten vor. Eine Referenz kann dabei auf drei verschiedene Weisen in der abstrakten Syntax realisiert werden. Beispielhaft betrachtet wird zu dem Zweck erneut das Konzept `Task` und dabei insbesondere die Möglichkeit, einen Ablauf zu modellieren. Die dafür erforderliche

Flussbeziehung kann in Form eines Nachfolgerattributs, eines Vorgängerattributs oder auch bidirektional mit allen beiden Attributen umgesetzt werden. Deshalb bietet `OntReferenceMapping` für jede Richtung eine optionale Eigenschaft, die auf ein referentielles Attribut im Domänenelement des übergeordneten Element-Mappings verweisen kann. Die Eigenschaft `toAttribute` legt dabei das Attribut auf Quellseite fest (im Beispiel das Nachfolgerattribut), wohingegen `fromAttribute` das Attribut auf Zielseite spezifiziert (im Beispiel das Vorgängerattribut). Eines der beiden Felder muss gesetzt sein, da sonst keine Abbildung zustande kommt.

5.2.3 Beispiel für die Anwendbarkeit der Meta-Modelle

Das Beispiel soll einerseits die Entstehung der einzelnen, im ontologischen Kontext beteiligten Modelle verdeutlichen, sofern ohne vorgegebene Sprache mit der freien Modellierung begonnen wird. Dabei handelt es sich also um die Definition einer DSML nach dem Bottom-Up Prinzip. Andererseits soll dargelegt werden, in welcher Form die Konzepte der in den beiden vorherigen Abschnitten beschriebenen Meta-Modellen instanziiert werden. Eine grobe Idee wurde bereits mit dem einführenden Beispiel in Abschnitt 5.1.5 geliefert, die neben den Anforderungen als Grundlage zur Spezifikation der Meta-Modelle dient. Vorweg ist anzumerken, dass alle in diesem Abschnitt vorkommenden Objektdiagramme nicht vollständig sind, da aus Übersichtlichkeitsgründen auf die Angabe der Instanzen der Typisierungsklassen sowie Stylingangaben verzichtet wurde. Die konkreten Knotentypen und Stylingmöglichkeiten werden ohnehin erst innerhalb der jeweiligen Kapitel zu den beiden Visualisierungsformen vorgestellt und sind daher zum gegenwärtigen Zeitpunkt noch nicht bekannt.

Grundlage des Beispiels bildet das Diagramm in Abbildung 5-10. Es zeigt einen einfachen Prozess mit lediglich einem Prozessschritt. Die graue Umrandung repräsentiert die Wurzel des Diagramms und damit den Gesamtprozess. Das grüne Rechteck dagegen stellt eine Aufgabe innerhalb des Prozesses dar, das mit der zu erledigenden Tätigkeit beschriftet ist. Das kleine dunkelgrüne Männchen in der oberen linken Ecke hat nur symbolischen Charakter und bringt zum Ausdruck, dass die Aufgabe von einer Person durchzuführen ist. Erneut kommt die grafische Visualisierungsform zum Einsatz, da textuelle Beispiele von ihrer internen Strukturierung meist komplexer ausfallen. Die Art und Weise der Strukturierung würde außerdem zusätzlichen Erklärungsbedarf erfordern. Diese Erläuterungen werden jedoch erst später in Kapitel 7 gegeben, weil sie auf weiteren Entwurfsentscheidungen beruhen, die ausschließlich für die textuelle Visualisierungsform zutreffen.

Wie das Diagramm intern als visuelles Verwendungsmodell abgelegt ist, veranschaulicht Abbildung 5-11. Gemäß der grundlegenden Designprinzipien aus Abschnitt 5.1 muss zu einem Verwendungsmodell immer auch ein zugehöriges Definitionsmodell existieren. Dieses ist in Abbildung 5-12 dargestellt. Wie beim konzeptuellen Beispiel in Abschnitt 5.1.5 wird die Instanzspezialisierung durch eine entsprechende Namensgebung angezeigt und nicht durch Verbindungslinien, die die Lesbarkeit beeinträchtigen würden. Demzufolge bezeichnet `diagramDef` die zu `diagram` gehörende Definition.

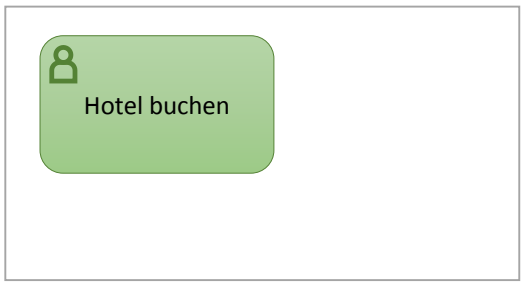

Abbildung 5-10: Diagramm eines einfachen Prozesses als Ausgangspunkt

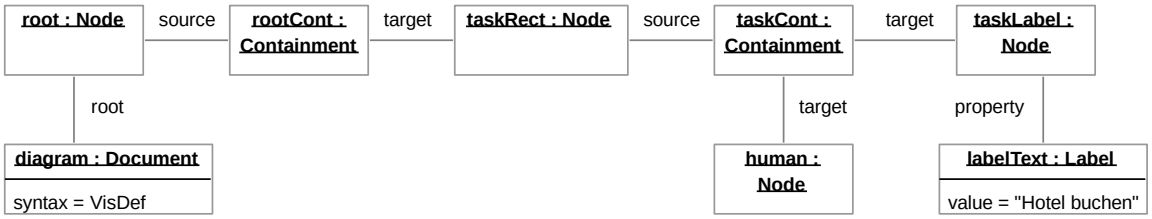

Abbildung 5-11: Beispiel für visuelles Verwendungsmodell (`VisUse`) ohne Mapping

Definitions- und Verwendungsmodell weisen weitgehend die gleiche Struktur auf. Zurückzuführen ist dies auf das sehr einfache Prozessdiagramm, welches lediglich einen Prozess umfasst. Bei mehreren Prozessen wäre das Verwendungsmodell erheblich größer, ohne dass dies jedoch Auswirkungen auf das Definitionsmodell hätte. Ersichtlich wird das durch einen Blick auf Abbildung 5-7. Darin ist die Struktur der Verwendungsebene umfangreicher als die der Definitionsebene, weil zwei Rauten im Dokument auftreten, die auf derselben Definition basieren. Im vorliegenden Beispiel hingegen fällt sogar das Definitionsmodell größer aus, weil es zusätzlich einige domänenrelevante Informationen beinhaltet.

Den Startpunkt im Verwendungsmodell bildet `diagram` als Instanz des Konzepts `Document`, indem es zum einen das Modellgeflecht konstituiert. Ohne zugrundeliegende Sprache bedeutet dies lediglich, dass es durch eine Zuweisung an das `syntax`-Attribut das zugehörige visuelle Definitionsmodell angeben muss. Zum anderen legt `diagram` via `root` den Wurzelknoten fest, der letztendlich der grau umrandeten Diagrammfläche aus Abbildung 5-10 entspricht. Die enthaltene Aufgabe (grünes Rechteck) wird vom Knoten `taskRect` repräsentiert und ist durch `rootCont` an die Wurzel gebunden. Dass es sich bei `rootCont` um eine semantische Enthaltensein-Beziehung handelt wird allerdings von der zugehörigen Definition `rootContDef` bestimmt. Analoges gilt für die Referenz `taskCont` und dessen Definition `taskContDef`, wobei es sich hierbei um rein visuelles Enthaltensein handelt. In beiden Fällen verweist die `Containment`-Instanz auf das Männchen und den Schriftzug und deklariert beide somit als Kindknoten. Jeder Knoten darf nämlich auf maximal eine Instanz derselben Feature-Spezialisierung verweisen. Sind mehrere Verweise bzw. Werte zu verwalten, dann können diese in einer gemeinsamen Instanz hinterlegt werden. Der Text des Schriftzugs wird in der Label-Eigenschaft `labelText` gespeichert. Er ist durch die zugrundeliegende Definition `labelTextDef` als semantisch deklariert.

Ein auffälliger Punkt, in dem sich Definitions- und Verwendungsmodell unterscheiden, sind die beiden Annotationen `rootName` und `taskName`. Sie dienen als Namenshinweis für die spätere Generierung einer abstrakten Syntax. So ist als Name für das Meta-Konzept, das die Diagrammwurzel repräsentiert, „Process" vorgegeben und für das Meta-Konzept, das Aufgaben widerspiegelt, der Name „Task".

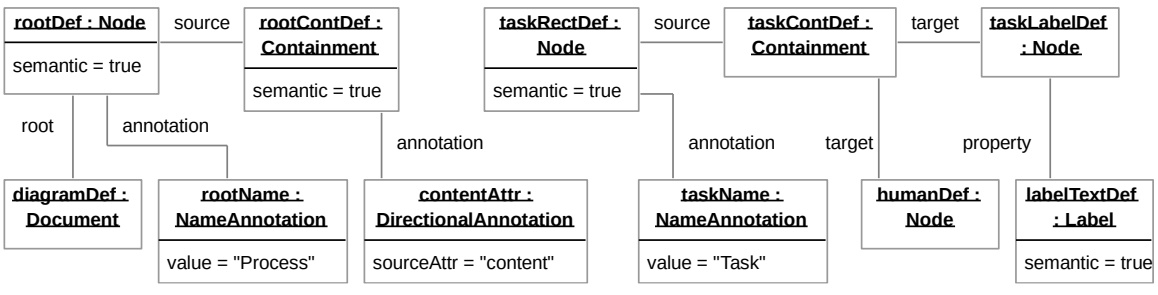

Abbildung 5-12: Beispiel für visuelles Definitionsmodell (`VisDef`) ohne Mapping-Definition

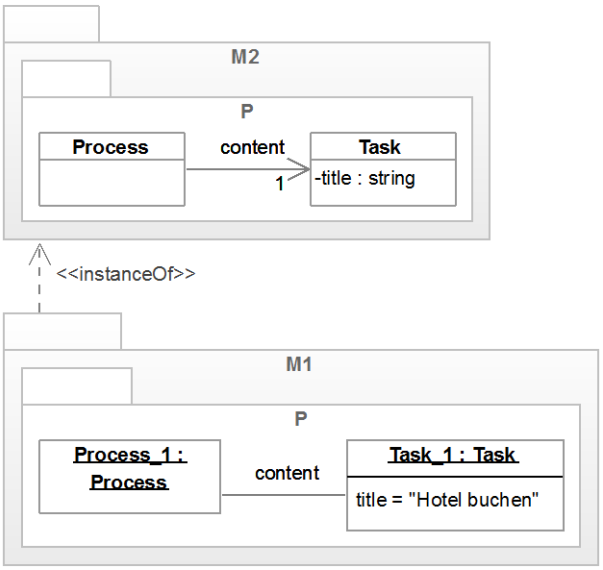

Abbildung 5-13: Beispiel für Domänen-Stack

Unter der Annahme, dass das Prozessdiagramm für unser Anwendungsszenario ausreichend ist, werden nun die noch fehlenden Komponenten einer vollständigen DSML zuzüglich Domänenmodell und Mapping-Verwendungen automatisiert abgeleitet. Abbildung 5-13 visualisiert hierfür die Modelle des Domain Stack. Für die Diagrammdefinition wird in der abstrakten Syntax `M2` das Konzept `Process` generiert, welches wegen der semantischen Referenz `rootContDef` das Attribut `content` erhält. Seine Multiplizität ist auf `1` gesetzt, weil lediglich ein beispielhaftes Prozessdiagramm mit genau einer Aufgabe vorliegt. Dass genau ein Attribut auf Quellseite mit dem Namen „content" erzeugt wird, ist der Annotation `contentAttr` geschuldet.

Für das blaue Rechteck wird ebenfalls ein Konzept auf `M2` angelegt, da die zugrundeliegende Knotendefinition `taskRectDef` von einer semantischen Beziehung referenziert wird. Andernfalls gäbe es keinen geeigneten Datentyp für `content`. Das betreffende Konzept heißt `Task`. Es enthält überdies ein String-Attribut mit dem Namen `title`, welches aus der semantischen Eigenschaft `labelTextDef` resultiert. Seine Benennung erfolgt bei Labels standardmäßig in „title", sofern keine steuernde `NameAnnotation` existiert. Innerhalb des Domänenmodells werden gleichzeitig die zum visuellen Verwendungsknoten gehörenden Instanzen abgeleitet und passend parametriert. Dabei handelt es sich um die Konzepte `Process_1` und `Task_1` sowie die Zuweisungen `content = Task_1` und `title = "Hotel buchen"`.

Um die Beziehung zwischen Visualisierung und Domäne nicht zu verlieren, muss sie sowohl auf Definitions- (Abbildung 5-14) als auch auf Verwendungsseite (Abbildung 5-15) mithilfe von Mappings festgehalten werden. Da Mappings in einem separaten Verwendungs- bzw. Definitionsmodell abgelegt werden, sind sie in den beiden Objektdiagrammen zur Unterscheidung von Visualisierungskonzepten in Gelb gedruckt. Beim Vergleich der Struktur von Mapping-Verwendungen und Mapping-Definitionen fällt wiederum auf, dass sich beide stark ähneln. Der Grund hierfür ist, dass es auch auf Mappingseite zu jeder Verwendung eine Definition geben muss.

Im Zentrum der Betrachtung steht zunächst das Mapping-Definitionsmodell. Nachdem `rootContDef` als semantisch deklariert ist, müssen sowohl Quell- als auch Zielknoten auf ein Element der Domäne abgebildet werden. Das geschieht mittels `OntElementMappings`, nämlich `processMapDef` und `taskMapDef`. Im ersten Mapping wird als `concept` das Konzept `Process` festgelegt, während beim zweiten `Task` angegeben ist.

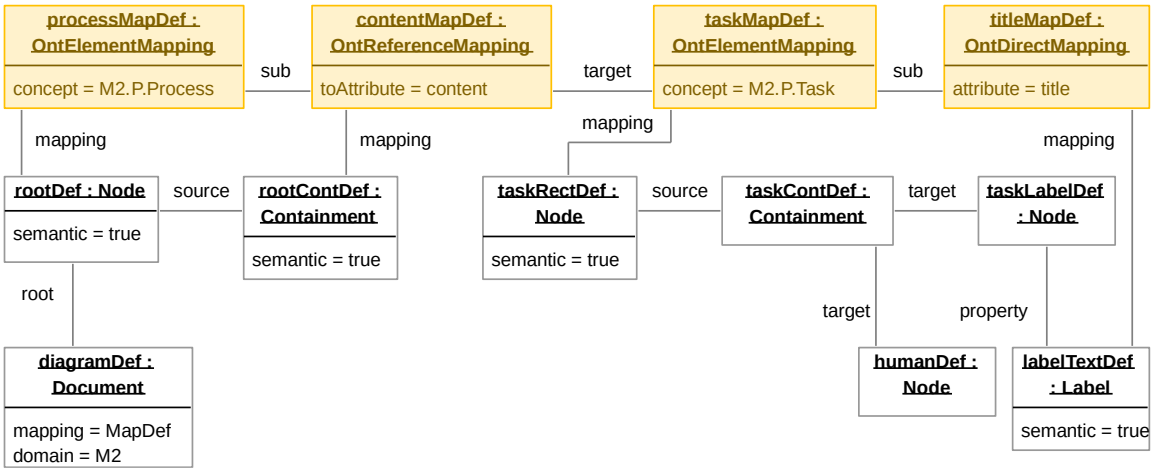

Abbildung 5-14: Beispiel für visuelles Definitionsmodell (`VisDef`) mit angebundener Mapping-Definition (`MapDef`)

Darüber hinaus muss die Enthaltensein-Beziehung als solche ebenfalls auf die abstrakte Syntax übertragen werden, was durch `contentMapDef` erledigt wird. Dieses `OntReferenceMapping` legt fest, dass die Beziehung auf Domänenseite durch eine Zuweisung von `Task`-Instanzen an das `content`-Attribut von `Process`-Instanzen abgebildet wird.

Ferner gibt es noch ein `OntDirectMapping`, das die von `taskLabelDef` repräsentierte Beschriftung an das `title`-Attribut der zugehörigen `Task`-Instanz delegiert. Wie aus dem Beispiel ersichtlich ist, können Sub-Mappings auch auf Features von Kindknoten angewendet werden. Denn `titleMapDef` bezieht sich auf die Eigenschaft `labelTextDef`, die allerdings zum Knoten `taskLabelDef` und nicht zu `taskRectDef` gehört. Wichtig ist lediglich, dass die Sub-Mappings dem (im Sinne der Domäne) korrekten Element-Mapping untergeordnet sind. Im vorliegenden Fall muss also `titleMapDef` zu `taskMapDef` gehören, da sich das Direct-Mapping auf den Titel von Aufgaben bezieht.

Beim Vergleich der beiden visuellen Definitionsmodelle fällt auf, dass in der gemappten Version keine Annotationen mehr vorhanden sind. Sie wurden beim Ableiten aus dem Definitionsmodell entfernt. Der Grund dafür ist, dass beim späteren Ableiten von Änderungen diese Annotationen erneut ausgewertet würden. Hat ein Benutzer jedoch das Domänen-Meta-Modell inzwischen manuell modifiziert, dann würden seine Änderungen unter Umständen irrtümlicherweise überschrieben werden. Weitere Details dazu finden sich in Abschnitt 5.4.

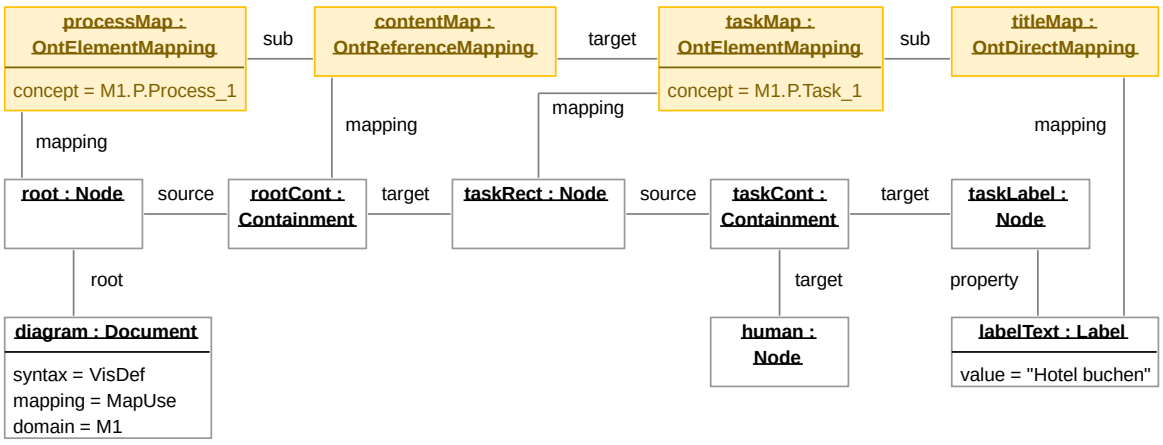

Abbildung 5-15: Beispiel für visuelles Verwendungsmodell (`VisUse`) mit angebundener Mapping-Verwendung (`MapUse`)

Damit die Zugehörigkeit der Mappings auf Verwendungsebene eindeutig ist, muss dort dieselbe Struktur nachgebildet werden. Statt Abbildungen für Elemente der abstrakten Syntax (`M2`) zu etablieren wird hier auf Elemente des Domänenmodells (`M1`) verwiesen. Demnach zeigen `processMap` auf `Process_1` und `taskMap` auf `Task_1`. Die Zuweisungen innerhalb der Sub-Mapping-Verwendungen brauchen dagegen nicht separat angegeben werden, da sie schon durch die Attribute der zugrundeliegenden Definitionen spezifiziert sind.

Ergänzend zu den Mappings müssen in den `Document`-Instanzen noch die betreffenden Modelle auf Mapping- und Domänenseite referenziert werden, so dass die komplette Modellhierarchie Gültigkeit besitzt. Dazu muss die Dokumentdefinition `diagramDef` via `mapping` auf das Mapping-Definitionsmodell `MapDef` und mittels `domain` auf das Domänen-Meta-Modell `M2` verweisen. Die zugehörige Verwendung `diagram` hingegen zeigt mithilfe derselben Attribute auf das Mapping-Verwendungsmodell `MapUse` sowie auf das Domänenmodell `M1`.

5.3 Konstruktivistischer, initialer Entwurf einer DSML

In den bisherigen Ausführungen wurden in erster Linie Gemeinsamkeiten hinsichtlich der Struktur ontologischer Dokumente ermittelt und darauf basierend Meta-Modelle hergeleitet, die als Abspeicherungsformat für diese Struktur dienen. Nachfolgend geht es um die Identifikation von Gemeinsamkeiten im ontologischen Modellierungsprozess und dabei insbesondere um die Ableitung einer DSML nach heuristischen Maßstäben. Die gesamte in diesem Abschnitt vorgestellte Methodik ist ein wesentlicher Bestandteil des eigenen, konzeptuellen Beitrags (Abschnitt 1.3).

5.3.1 Interaktion mit der konkreten Syntax

Thema dieses Abschnitts sind die diversen Interaktionsmöglichkeiten mit der konkreten Syntax, auf die der Benutzer zurückgreifen kann. Den Anfang macht die Erstellung neuer visueller Konstrukte unter Anwendung des konstruktivistischen Prinzips. Anschließend werden zwei Alternativen erörtert, wie bereits vorhandene visuelle Konstrukte wiederverwendet werden können. In Abschnitt 5.3.1.4 geht es schließlich um das Beisteuern von Domänenwissen durch das Annotieren visueller Konstrukte und deren Bestandteile.

5.3.1.1 Beispielgetriebener Entwurf visueller Konstrukte

Der Entwurf eines neuen visuellen Konstrukts beginnt damit, dass ein **neuer Knoten** im Verwendungsmodell **an das Ende einer semantischen Referenz** angehängt wird. Dies führt unweigerlich zur Erzeugung eines zugrundeliegenden, semantischen Knotens auf Definitionsseite. In der textuellen Welt ist das Hinzufügen eines nicht parametrierten Knotens zum Dokument üblich (Abschnitt 7.4). Im Grafischen hingegen ist es ratsam rudimentär vorkonfigurierte Knoten anzubieten (z.B. Knoten, die als Rechtecke, Ellipsen oder Rauten visualisiert werden).

Ein neuer Knoten kann gleichermaßen auch an einer rein visuellen Referenz befestigt werden, was in eine **Anpassung des betreffenden Konstrukts** mündet und die zugrundeliegende Knoten-Definition als rein visuell determiniert. Ein Beispiel ist das Hinzufügen einer Ellipse zur Raute in Abschnitt 5.1.5. Gibt es bereits mehrere Verwendungen eines Konstrukts, dann werden sie standardmäßig in gleicher Weise angepasst wie die ursprünglich bearbeitete Verwendung. Das trifft ebenso für rein visuelle Eigenschaften zu. Mögliche alternative Strategien sind spezifisch für die jeweilige Darstellungsform und werden erst dort aufgegriffen.

Analoges gilt auch für die **Beachtung der Intention**, die sich **hinter den jeweiligen Features** eines Konstrukts verbirgt. Festgelegt ist sie bereits durch den jeweiligen Typ des Features (im allgemeinen Fall also `Label`, `Visibility` und `Containment`). Die Intention zu berücksichtigen ist Aufgabe des konkreten Editors, da er in der Verantwortung steht, spezifische Bearbeitungsmöglichkeiten für jeden Feature-Typ bereitzustellen. Dabei muss er auf die Einhaltung des Wertebereichs des jeweiligen Features achten (z.B. nur `true` und `false` bei Instanzen von `Visibility`).

5.3.1.2 Aktives Wiederverwenden visueller Konstrukte

Bei der Wiederverwendung visueller Konstrukte lassen sich zwei Varianten unterscheiden. Die erste wird als aktive Variante betitelt, weil hier der Benutzer aktiv ein bereits im zugrundeliegenden Definitionsmodell vorhandenes Konstrukt für die Wiederverwendung im Dokument auswählt. Ein Beispiel eines erneut verwendbaren grafischen Konstrukts ist die schon mehrfach vorgekommene Raute mit eingeschachteltem Kreis. In der textuellen Welt dagegen kann es sich um die Definition eines Statements mit dem Schlüsselwort „state" und einem darauffolgenden, beliebig wählbaren Bezeichner handeln. Zwei gültige Verwendungen hierfür sind beispielsweise `state Idle` und `state Active`. Im weiteren Verlauf dieser Arbeit liegt das Hauptaugenmerk auf der ersten Variante, weil damit existierende Konstrukte ohne zusätzlichen Aufwand wiederverwendet werden können. Sie erneut nachzubauen würde zusätzlichen Aufwand ohne erkennbaren Mehrwert bedeuten.

5.3.1.3 Automatische Identifikation äquivalenter Konstrukte

Im Gegensatz zur aktiven Wiederverwendung (Abschnitt 5.3.1.2 handelt es sich bei der zweiten, in diesem Abschnitt erläuterten Variante um einen Automatismus, der auf visueller und struktureller Äquivalenz beruht. Der Automatismus identifiziert gleichartige Konstrukte, die somit potentielle Kandidaten für eine Zusammenführung darstellen. Diese zweite Variante wird nicht ignoriert, da sie ihrerseits spezifische Vorteile mit sich bringt. Fügt der Benutzer z.B. zwei Mal (versehentlich) einen Knoten in ein Diagramm ein und parametriert ihn als Rechteck (ohne sonstiger Anpassungen, wie Setzen der Hintergrundfarbe oder Verstärken der Umrandungsdicke), dann soll ein Automatismus feststellen, dass zwei gleichartige Konstrukte im Diagramm verwendet werden, denen jedoch keine gemeinsame Definition zugrunde liegt. Auf Basis dieser Erkenntnis kann dem Benutzer dann ein **Hinweis** unterbreitet werden, der empfiehlt, für beide Rechteckskonstrukte **dieselbe Definition zu setzen**.

Für grafische, graphbasierte Sprachen wurde dies bereits in [5] untersucht. Darin wird ein Algorithmus vorgestellt, der basierend auf einem Diagramm dessen „Building Blocks" ermittelt und dafür jeweils einen Typ innerhalb der abstrakten Syntax erzeugt. Dabei fordern die Autoren vom Benutzer nicht, dass Konstrukte mit demselben Domänenhintergrund auf die gleiche Weise zu visualisieren sind. Dies äußert sich dadurch, dass im Beispieldiagramm des Artikels Objekte, Prozesse und Services mit denselben Formen dargestellt sind. So gibt es beispielsweise für jeden der drei Typen mindestens eine Ausprägung, die als Kreis mit gepunkteter Umrandung visualisiert ist. Gleichzeitig existieren aber auch Prozesse und Services, die als Kreis mit doppeltem Rand gezeichnet sind. Dieses hohe Maß an Freiheit birgt das Risiko in sich, dass die Identifikation der konkreten Syntax zu einem Ratespiel degeneriert und letztendlich der Benutzer aus einer großen Anzahl grafischer Konstrukte die für seinen Anwendungsfall am besten Geeigneten auswählen muss. Deshalb betrachten wir bei unserem Ansatz nur Konstrukte als gleichartig, die auch in ihren visuellen Eigenschaften und ihrer Struktur einander gleichen. Das geschieht analog zur Äquivalenz von Typen und Attributen im linguistischen Kontext (Kapitel 4), die auf Namensgleichheit fußt. Eine vergleichbare Idee wird auch in [21] propagiert, allerdings ohne Angabe einer Strategie oder eines Algorithmus. In den Beispielen auf der Homepage des Autors [145] beschränkt sich die Analyse, ob zwei oder mehr verwendete grafische Konstrukte äquivalent sind, auf

die zugrundeliegende Form oder Bilddatei. Strukturelle und auf weiteren visuellen Eigenschaften basierende Aspekte werden offenbar nicht beachtet.

Jedes **visuelle Konstrukt** kann prinzipiell **als Teilgraph** des gerichteten, attributierten Graphs des Definitionsmodells aufgefasst werden. Folglich kann durch Anwendung von Graph Pattern Matching [47] überprüft werden, ob sich die Graphstruktur eines neuen Konstrukts im vorhandenen Definitionsmodell wiederfinden lässt. Veröffentlichungen aus den letzten Jahren zur Thematik des **Graph Pattern Matching** zeigen überdies, dass effiziente Verfahren zur Berechnung des Matching existieren [20, 38].

Im Gegensatz zur Definition „visuelles Konstrukt" werden semantische Referenzen komplett ausgeblendet, da sie keinen Beitrag zur eigentlichen Graphstruktur und Darstellungsform leisten. Dasselbe trifft für semantische Eigenschaften zu. Die Belegung semantischer Features ist ohnehin nur innerhalb der Verwendungsmodelle relevant, da in den Definitionsmodellen diese Features lediglich als Platzhalter fungieren. Deshalb bestehen **für das Matching** die Konstrukte **ausschließlich** aus **Knoten mit rein visuellen Eigenschaften und rein visuellen Referenzen**. Neben den als eigenständige Konzepte modellierten Eigenschaften zählt ebenso der Knotentyp zu den Attributen eines Knotens.

Bei komplexen Konstrukten ist es insbesondere **im textuellen Kontext sinnvoll**, wenn **Vorschläge für das Zusammenführen** von Definitionen angeboten werden. Dort geht das Erstellen solcher Konstrukte leichter von der Hand als im Grafischen, weil da Zerlegung des eingegebenen Fließtexts weitgehend automatisch im Hintergrund stattfindet (nähere Details dazu finden sich in Abschnitt 7.4.1).

Abbildung 5-16 zeigt demzufolge ein textuelles Beispielkonstrukt, an dem der Aufbau der für das Pattern Matching benötigten Graphstruktur demonstriert wird. Bei dem Konstrukt handelt es sich um ein Statement (symbolisiert durch die grau gestrichelte Umrandung im linken Bildabschnitt) zur Repräsentation eines Zustands. Gekennzeichnet wird dies durch das enthaltene Schlüsselwort `state`. Damit jeder Zustand eindeutig identifiziert werden kann, beinhaltet das Statement einen Bezeichner-Knoten, der innerhalb der möglichen Visualisierung (links) mithilfe von `<Id>` ausgedrückt ist. Zudem verfügt es über einen Knoten vom Typ `Whitespace`, der zwischen dem Schlüsselwort und dem Bezeichner liegt, so dass diese in einem gewissen Abstand voneinander platziert sind. Ein anderes Konstrukt gilt als äquivalent zu diesem Zustandsstatement, wenn es ebenfalls vier Knoten mit denselben Typen aufweist und diese Knoten auf die gleiche Weise über `Containment`-Referenzen miteinander in Beziehung stehen. Zusätzlich müssen der Schüsselwort- und der Whitespaceknoten die Eigenschaft `label` mit derselben Wertebelegung wie in der Vorlage besitzen.

Allgemein ausgedrückt gelten demnach **zwei Konstrukte** als **äquivalent**, wenn sie dieselbe Anzahl von Knoten und Kanten aufweisen, diese Knoten in der gleichen Richtung mithilfe der Kanten verbunden sind und schließlich noch über dieselben Eigenschaften verfügen. Ferner muss die Typisierung der sich aufeinander beziehenden Kanten identisch sein.

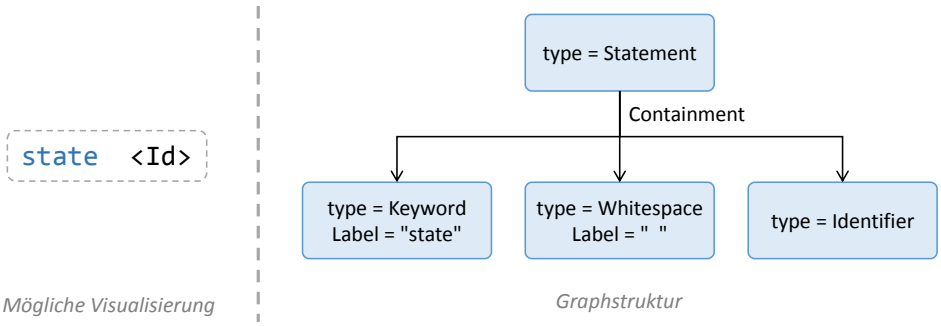

Abbildung 5-16: Attributierter Graph eines textuellen Konstrukts für die Anwendung von Pattern Matching

5.3.1.4 Annotieren visueller Konstrukte

Noch nicht behandelt wurde bisher wie die Unterscheidung erfolgt, ob **Features rein visueller oder semantischer Natur** sind. Dies stellt beim freien Modellieren und folglich während des Entwurfs der konkreten Syntax ein zentrales Thema dar. Sämtliche in Abschnitt 3.4 untersuchten Ansätze und Frameworks umgehen diese Problematik, indem sie alle visuellen Artefakte als semantisch betrachten und außerdem davon ausgehen, dass die eingesetzten visuellen Konstrukte bereits vollständig arrangiert vorliegen. Die Komposition neuer komplexer Konstrukte aus einer Reihe von Grundformen bzw. Textschnipsel liegt somit außerhalb ihrer Zuständigkeit.

Zur Bereitstellung eines integrierten Systems, welches beispielgetriebene Entwicklung von DSMLs gemäß Abschnitt 1.3 ermöglicht, ist auch das **Kreieren und Manipulieren visueller Konstrukte** eine Aufgabe, die **Unterstützungsbedarf** erfordert. Dadurch kann die Unterbrechung des Modellierungsflusses minimal gehalten werden, weil nicht auf ein externes Werkzeug zurückgegriffen werden muss. Viel wichtiger jedoch ist, dass durch den modularen Aufbau von Konstrukten, auf einfache Weise bestimmte Bereiche eines Konstrukts (in Form von Knoten) als Schnittstelle zu anderen Konstrukten deklariert werden können. So kann beispielsweise im grafischen Umfeld ein Bereich als Container gelten, während eine andere am Rand befindliche Region als Anknüpfstelle für ausgehende Verbindungen dient. Ob nun die im Container liegenden Elemente oder die ausgehenden Verbindungen nicht doch Teil des Konstrukts sind, muss vom Benutzer auf irgendeine Weise festgelegt werden.

Um den Ansätzen verwandter Arbeiten nicht zu widersprechen und weil der Großteil der Modellierungsaktionen mit hoher Wahrscheinlichkeit semantischer Natur ist, besteht die Prämisse, dass standardmäßig alle **Beziehungen** als semantisch deklariert sind. Soll eine Beziehung dennoch rein visuellen Charakter besitzen, so ist dies explizit anzugeben. Wie die konkreten Strategien hierfür aussehen, ist spezifisch für die jeweilige Visualisierungsform, weshalb sie erst später in den Abschnitten 6.2 und 7.3 vorgestellt werden.

Analog wird auch bei der Unterscheidung von literalen **Eigenschaften** vorgegangen. Regulär wird angenommen, dass verschiedene Werte für eine derartige Eigenschaft auch auf Unterschieden innerhalb der Domäne beruhen. Soll die abstrakte Syntax davon hingegen unberührt bleiben und die Eigenschaft rein visuell sein, dann ist das wiederum explizit zu spezifizieren. Auch hier sind verschiedene Strategien umsetzbar, die aufgrund ihrer Spezifität für die jeweilige Visualisierungsform erst später (ebenfalls in den Abschnitten 6.2 und 7.3) aufgezeigt werden.

Wenn ein neues Konstrukt eingeführt wird, dann verfügt es per se über einen Namen, doch der ist computergeneriert und liefert damit keinen Rückschluss auf die Intention des Konstrukts. Deshalb ist es nicht zwangsweise erforderlich, aber durchaus sinnvoll, ein neu definiertes **Konstrukt** mit einem im Kontext der Domäne **geeigneten Namen** zu versehen. Diesem Zweck dient die im visuellen Meta-Modell deklarierte `NameAnnotation`. Bei der Angabe eines Namens handelt es sich um eine zusätzliche Aufgabe, die der Benutzer erledigen sollte, sobald er ein neues Konstrukt eingeführt hat. Im ersten Moment klingt das nach dem in der Problemstellung (Abschnitt 1.1) kritisierten Überspringen der Modellebene (also von M0 direkt nach M2 unter Vernachlässigung von M1). Allerdings ist das ein Irrtum, weil erst nachdem ein visuelles Element im Dokument modelliert wurde, der Name des zugrundeliegenden Typs anzugeben ist. Es existiert somit bereits ein (imaginäres, da noch nicht abgeleitetes) Element auf Modellebene M1, dessen Typ erst anschließend auf Sprachebene M2 benannt wird. Ein weiterer bedeutender Vorteil der Benennung ist, dass eine nachgeschaltete Identifikation der abstrakten Syntax diese Namen übernehmen kann und daher nicht auf computergenerierte Namen, die ohne jeglichen Bezug zur Domäne stehen, zurückgreifen muss. Umgesetzt wird die Benennung – wie schon im Beispiel aus Abschnitt 5.2.3 geschehen – mithilfe einer Annotation, die dem Wurzelknoten des Konstrukts zugeordnet ist.

Die `NameAnnotation` dient zudem auch der **Benennung von literalen Attributen**. Dazu muss eine solche Annotation der betreffenden Eigenschaft zugeordnet werden. Dies ist jedoch eher die Aufgabe eines Modellierungsspezialisten, da der Domänenexperte üblicherweise nicht mit der abstrakten Syntax und somit irgendwelchen Attributen in Berührung kommt. Eine geeignete Möglichkeit zum Annotieren von Eigenschaften muss der jeweilige Editor bereitstellen.

Für das Annotieren von Referenzen existiert die spezielle, ebenfalls im allgemeinen visuellen Meta-Modell vorhandene `DirectionalAnnotation`. Sie gibt an, welche **Attribute für eine semantische Referenz** erzeugt werden: Soll nur ein Attribut im Quellkonzept oder nur ein Attribut im Zielkonzept oder sollen sogar beide Attribute erzeugt werden? Diese drei Optionen spiegeln genau die drei Mapping-Möglichkeiten des `OntReferenceMapping` aus Abschnitt 5.2.2.2 wider. Auch hierfür ist die Bereitstellung einer passenden Werkzeugunterstützung unerlässlich.

Die dritte, allgemeingültige Annotation `ValueRelation` erlaubt es **Beziehungen zwischen Werten der Visualisierung und Werten der Domäne** herzustellen. So können beispielsweise in grafischen Sprachen die Werte von Styling-Eigenschaften (wie Form und Farbe) auf diskrete domänenseitige Werte abgebildet werden. Eine derartige Abbildung kann auch ein Domänenexperte angeben, da er – bezogen auf das semantische Einfärben – weiß, welche Aussage sich hinter welcher Farbe verbirgt. Der jeweilige Editor muss deshalb für Eigenschaften eine komfortable Möglichkeit bieten, Wertebeziehungen spezifizieren zu können.

Ein weiterer wichtiger Punkt ist, dass sämtliche Annotationen, die **während des Ableitungsvorgangs** verarbeitet werden, **deaktiviert** werden. Dies ist deshalb von Bedeutung, weil sonst beim späteren Ableiten von Änderungen manuell getätigte Modifikationen der abstrakten Syntax unter Umständen überschrieben würden. Ersichtlich wird das schon bei der Benennungs-Annotation. Wurde beispielsweise der Name eines Konzepts während des initialen freien Modellierens auf „Task" gesetzt und später durch einen Modellierungsexperten im Domänen-Meta-Modell in „Activity" umbenannt, dann hätte die wiederholte Auswertung der besagten Annotation – wie sie typischerweise beim inkrementellen Ableiten von Änderungen (Abschnitt 5.4.2) stattfindet – ein Rückgängigmachen dieser Änderung zur Folge. Der Name des Konzepts würde also wieder „Task" lauten.

5.3.2 Ableiten des Domänen-Stacks einschließlich zugehöriger Mappings

In einigen Ausnahmefällen mag es durchaus vorkommen, dass abstrakte und konkrete Syntax völlig voneinander losgelöst sind und sich daher in ihrer Granularität und Struktur stark unterscheiden. Damit eine einfache automatisierte Verarbeitung gewährleistet werden kann, wird jedoch gemeinhin empfohlen, dass sich **beide Syntaxen** hinsichtlich der beiden erwähnten Kriterien **möglichst ähneln** [73, 114].

Dies ist auch den Prinzipien zuträglich, auf denen die Bottom-Up Vorgehensweise fußt. Denn beim Ableiten der abstrakten Syntax kann primär nur auf die visuellen Verwendungen mit den zugehörigen visuellen Definitionen zurückgegriffen werden. Es können also **lediglich Informationen** einbezogen werden, die **aus visuellen Artefakten der freien Modellierung** stammen. Der Benutzer kann zwar durch Konfiguration und Annotation zusätzliches Domänenwissen beisteuern, doch sind diese Mechanismen vorrangig für die Eliminierung potentieller Mehrdeutigkeiten gedacht und sollen wegen ihrer Spezifität nicht zur Generierung komplexer, aus Meta-Konzepten bestehenden Konstrukten führen. Sind solche Konstrukte dennoch erforderlich, dann können sie Dank des integrierten Ansatzes durch manuelles Adaptieren der abstrakten Syntax realisiert werden, ohne später auf die Erweiterung der Sprache durch freies Modellieren verzichten zu müssen.

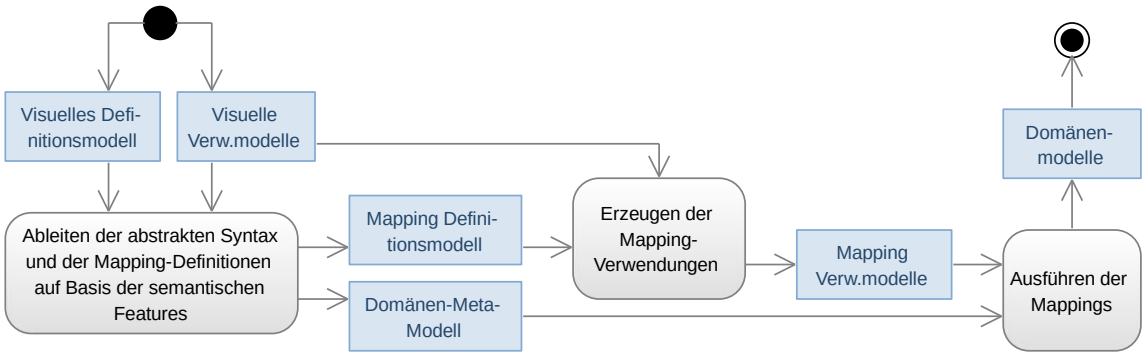

Abbildung 5-17: Algorithmus zur Generierung der noch fehlenden Modelle für semantisches Modellieren

Während im linguistischen Kontext das Ableiten der abstrakten Syntax (Abschnitt 4.3) den eigentlichen Bottom-Up Schritt darstellt, hat im ontologischen Kontext das **Bottom-Up schon beim Entwerfen und Verwenden der konkreten Syntax** stattgefunden. Ersichtlich ist das direkt durch einen Blick auf die Modellhierarchie aus Abbildung 5-2. Das visuelle Definitionsmodell steht nämlich über ein Mapping in Bezug zum Domänen-Meta-Modell (also der abstrakten Syntax). Gleiches gilt für das Verhältnis zwischen visuellem Verwendungsmodell und Domänenmodell. Bislang fehlen allerdings sowohl die Mapping-Modelle als auch die Modelle des Domänen-Stacks, weshalb sie noch erzeugt werden müssen.

Den **groben Ablauf**, in welcher Reihenfolge die noch fehlenden Modelle generiert werden, skizziert Abbildung 5-17. Zunächst werden auf Basis der visuellen Eingabemodelle das Domänen-Meta-Modell (abstrakte Syntax) und das Mapping-Definitionsmodell abgeleitet. Letzteres wird in Verbindung mit den visuellen Verwendungsmodellen zur Ableitung der Mapping-Verwendungsmodelle eingesetzt. Nachdem die einzelnen Mapping-Verwendungen über Methoden für die Ausführung der jeweiligen Abbildungsvorschrift verfügen (Abschnitt 5.2.2), können diese Methoden nun aufgerufen werden, was zur Generierung der Domänenkonzepte und damit zur Erzeugung der Domänenmodelle führt.

Die Erzeugung der Mapping-Verwendungen und deren Ausführung wurden bereits oben beschrieben. Deshalb liegt der Fokus der nachfolgenden Betrachtungen auf dem ersten Schritt. Es handelt sich dabei um einen Algorithmus, der basierend auf einer Menge visueller Modelle eine abstrakte Syntax sowie passende Mapping-Definitionen erzeugt.

5.3.2.1 Ableiten von abstrakter Syntax mit Mapping-Definitionen

Das Generieren von Mapping-Definitionsmodell und Domänen-Meta-Modell ist deswegen zu einem Schritt zusammengefasst, da in beiden Fällen dasselbe Verfahren zugrunde liegt und somit die jeweiligen Artefakte beider Modelle parallel erzeugt werden können. Dieser Schritt stellt die bisher als implizit angenommene **Verbindung zwischen dem `semantic`-Attribut des visuellen Meta-Modells** (Abschnitt 5.2.1) **und den Konzepten des ontologischen Map-Moduls** (Abschnitt 5.2.2.2) her. Ist im Folgenden von „Mappings“ die Rede, dann sind wegen der besseren Lesbarkeit stets Instanzen der ontologischen Derivate gemeint (z.B. meint Reference-Mapping eine Instanz von `OntReferenceMapping`), die sich innerhalb des Mapping-Definitionsmodells befinden.

Für das Ableiten von abstrakter Syntax und der zugehörigen Mapping-Definitionen kommen sogenannte *Inferrers* zum Einsatz. Grob gesagt sind sie für die **Generierung ontologischer Mappings und zugehöriger Domänenartefakte auf Grundlage semantischer Knoten und Features** verantwortlich. Analog zu den Mappings sind sie in Element- und Sub-Inferrer untergliedert. Wie der Name vermuten lässt, ist ein Element-Inferrer für die Erzeugung eines Element-Mappings und dem entsprechenden Meta-Konzept der abstrakten Syntax zuständig. Jeder spezifische Sub-Inferrer hingegen bezieht sich

auf Spezialisierungen von `SubMapping` und erzeugt demnach genau eine Instanz eines solchen Mappings sowie zugehörige Attribute für das betreffende Meta-Konzept.

Wann diese Inferrer eingesetzt werden, zeigt Abbildung 5-18. Sie visualisiert den **Algorithmus zum Ableiten der abstrakten Syntax inklusive zugehörigem Mapping-Definitionsmodell**. Dabei ist anzumerken, dass zugunsten der Übersichtlichkeit auf die Anzeige von Datenobjekten, die keine Auswirkungen auf den Ablauf des Algorithmus haben, verzichtet wurde. Deshalb sind weder das Domänen-Meta-Modell noch das Mapping-Definitionsmodell in diesem Aktivitätsdiagramm enthalten. Ihr Inhalt wird beim Anwenden der Element- und Sub-Inferrers erzeugt, wobei es von deren konkreter Implementierung abhängt, um welche Arten von linguistischen Modellelementen es sich handelt.

Zu Beginn des Algorithmus werden aus dem gegebenen visuellen Definitionsmodell alle Instanzen des Konzepts `Node` **extrahiert**, bei denen `semantic = true` gesetzt ist. Gemäß der Definition aus Abschnitt 5.3 repräsentiert jeder derartige Knoten nämlich die **Wurzel eines visuellen Konstrukts**. Für jedes Konstrukt der konkreten Syntax wird angenommen, dass es eine Entsprechung innerhalb der abstrakten Syntax gibt. Infolgedessen wird für jeden extrahierten Wurzelknoten anhand seines Knotentyps ein Element-Inferrer ausgewählt und angewendet. Der Inferrer erzeugt sowohl ein entsprechendes Element-Mapping als auch das zugehörige Pendant im Domänen-Meta-Modell.

Die **Möglichkeit**, einen **alternativen Element-Inferrer einsetzen** zu können, ist beispielsweise im grafischen Kontext von Bedeutung, um zwischen der Verarbeitung von Vertices und Kanten unterscheiden zu können. Während die generierten Artefakte in Bezug auf Vertices immer gleich ausfallen, gibt es bei Kanten verschiedene Resultate. Ein für Kanten spezialisierter Element-Inferrer kann sich um genau diese Differenzierung kümmern. Nähere Details dazu finden sich in Abschnitt 6.3.

Da ein neu erzeugtes Meta-Konzept mit einem **eindeutigen Namen** zu versehen ist, muss der Element-Inferrer diesen angeben. In Abschnitt 5.3.1.4 wurde zu dieser Problematik bereits eine Lösung vorgestellt, die auf **Annotationen** beruht. Als Name wird – sofern vorhanden – eine am Wurzelknoten anhaftende `NameAnnotation` ausgelesen und der darin gespeicherte Wert übernommen. Liegt keine solche Annotation vor, dann wird ein computergenerierter Name verwendet, der jedoch ohne Bezug zur Anwendungsdomäne steht. Neben dem Meta-Konzept legt der Element-Inferrer auch ein Element-Mapping an, dessen `concept`-Attribut ein Zeiger auf das eben erzeugte Meta-Konzept zugewiesen wird. Außerdem wird dieses Mapping dem Wurzelknoten des ursprünglichen visuellen Konstrukts zugeteilt.

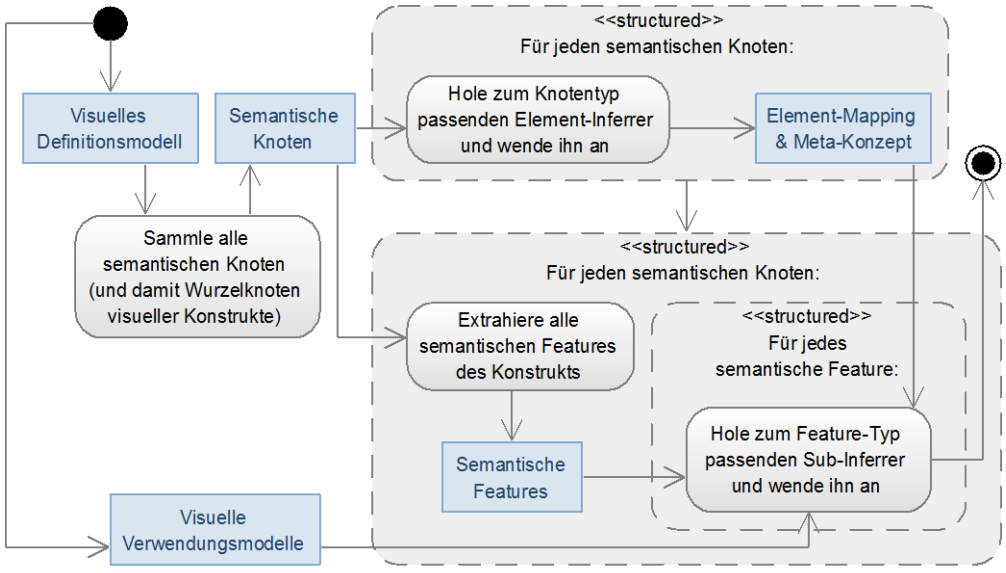

Abbildung 5-18: Algorithmus für das Ableiten von abstrakter Syntax und zugehörigen Mapping-Definitionen

Erst nachdem alle Element-Mappings mit den zugehörigen Meta-Konzepten erzeugt worden sind, erfolgt eine zweite Iteration über alle visuellen Konstrukte (gegeben durch die semantischen Knoten), die für jedes dieser Konstrukte sämtliche semantischen Features extrahiert. **Grund für das zweimalige Iterieren** sind die potentiell auftretenden Beziehungen zwischen den einzelnen Meta-Konzepten, die erst ausgebildet werden können, wenn die beteiligten Konzepte innerhalb des Domänen-Meta-Modells vorliegen. Basierend auf dem Typ der jeweiligen Feature-Spezialisierung wird anschließend ein Sub-Inferrer ausgewählt und angewendet. Ein Sub-Inferrer wirkt sich stets auf das Meta-Konzept aus, auf welches das Element-Mapping des jeweiligen visuellen Konstrukts verweist. Existiert kein solches Mapping, dann schlägt der Ableitungsvorgang fehl.

Für die bisher vorhandenen, allgemeingültigen Sub-Mappings führt dies immer zur **Erzeugung von Attributen** innerhalb des betreffenden Meta-Konzepts. Diese Attribute müssen mit einem Namen versehen werden, was auf demselben Weg wie bei der Benennung von Meta-Konzepten gelöst werden kann.

Sind keine Benennungs-Annotationen angegeben, dann müssen die Namen automatisiert bestimmt werden. Dies wiederum birgt die **Gefahr von Namenskollisionen**. Exemplarisch sei hierfür ein visuelles Konstrukt angenommen, das zwei semantische Labels umfasst, die zur Erzeugung von zwei String-Attributen innerhalb eines Meta-Konzepts führen. Zur Verhinderung von Namenskollisionen werden auf diese Weise auftretende gleichnamige Attribute durchnummeriert. Würden also beispielsweise zwei „title"-Attribute generiert, dann erhielte das zweite Attribut den Namen „title1". Hierbei sind durchaus alternative Strategien denkbar, doch würde kein im Sinne der Domäne besseres Resultat entstehen.

Darüber hinaus können die **gleichen Probleme** auftreten, die schon **im linguistischen Kontext** erörtert wurden. Sie werden deshalb in analoger Weise behandelt wie in Abschnitt 4.3. Liegt also eine Definition eines semantischen Features vor, dann werden zunächst für alle Verwendungen dieses Features Attribute generiert, ohne dass diese einem Meta-Konzept hinzugefügt werden. Stattdessen werden die Attribute im nächsten Schritt gemäß der in Abschnitt 4.3.1 beschriebenen Vorgehensweise miteinander verschmolzen. Erst das resultierende Attribut wird in das betreffende Meta-Konzept eingefügt. Abschließend wird für jedes neue Attribut getestet, ob es als optional zu deklarieren ist. Dies trifft dann zu, wenn es mindestens eine Knoten-Verwendung gibt, die auf dem zur oben erwähnten Feature-Definition gehörenden Knoten basiert und keine entsprechende Feature-Verwendung aufweist. Ist das der Fall, wird die untere Schranke der Multiplizität auf `0` gesetzt.

Beim **Anwenden eines jeden Inferrers** werden zudem die beiden letzten Schritte aus Abbildung 5-17 ausgeführt, wodurch Verwendungen der zuvor generierten Mapping-Definitionen sowie zugehörige Artefakte des Domänenmodells erzeugt werden. Hierbei wird auf die in den Mapping-Konzepten (Abschnitt 5.2.2.1) deklarierten Methoden zurückgegriffen, die zur Ausprägung des Mappings verantwortlich zeichnen.

5.3.2.2 Verarbeitung semantischer Features

Analog zu den konkreten Features (Abbildung 5-8) können im allgemeinen Fall vier Sub-Inferrers unterschieden werden. Darüber hinaus werden Wertebeziehungen als Sonderfall bei der Verarbeitung von Eigenschaften behandelt.

Label-Inferrer

Der Label-Inferrer erzeugt für jede Verwendung einer semantischen `Label`-Definition ein literales Attribut, ohne dies einem Meta-Konzept zuzuweisen. Generell ausgeschlossen sind `Label`-Verwendungen mit leerer Zuweisung (`value = ""`), weil sie sich optisch nicht von einem Nicht-Vorhandensein der selbigen unterscheiden lassen. In beiden Fällen wird im betreffenden Knoten kein Text

angezeigt. Von Bedeutung ist das für die spätere Berechnung, bei der entschieden wird, ob das resultierende Attribut als optional zu deklarieren ist.

Die Unterscheidbarkeitsproblematik kann zwar auch beim Ausblenden des besagten Knotens zutreffen, doch muss hier der Benutzer zum einen das Ausblenden aktiv veranlassen. Zum anderen muss es visuell nicht zwangsläufig der leeren Zuweisung entsprechen. Das ist dann der Fall, wenn der Knoten selbst eine visuelle Struktur aufweist (z.B. in Form einer Umrandung).

Der Attributtyp wird mithilfe eines regulären Ausdrucks ermittelt, indem dieser auf den Wert der Label-Eigenschaft angewendet wird (vergleichbar mit Abschnitt 4.3.1). Bei jedem auf diesem Weg generierten Attribut wird die die Multiplizität zunächst auf `1..1` gesetzt, wobei die obere Grenze gänzlich unveränderlich bleibt. Diese Limitierung resultiert erneut aus fehlendem Domänenwissen, da ohne Zusatzinformationen mehrere Werte nicht von einem gleichlautenden String unterscheidbar sind (z.B. kann `1, 3` die beiden Integer-Werte `1` und `3` meinen oder aber auch den String `"1, 3"`). Hierbei ist lediglich denkbar, dem Benutzer entsprechende Verfeinerungsempfehlungen zu unterbreiten, so dass er die Multiplizität nachträglich manuell ändern kann. Im Optimalfall gibt es einen speziellen Operator, der die gesamte Refaktorisierung vornimmt und dabei auch die einzelnen Wertzuweisungen folgerichtig anpasst.

Erst nach dem darauffolgenden Verschmelzen der bislang noch Eltern-losen Attribute wird das resultierende Attribut dem betreffenden Meta-Konzept hinzugefügt. Gleichzeitig kann dabei die untere Multiplizitätsgrenze herabgesetzt werden (siehe Abschnitt 5.3.2.1). Außerdem wird ein zugehöriges Direct-Mapping generiert.

Visibility-Inferrer

Die Funktionsweise des Visibility-Inferrer gestaltet sich etwas einfacher, da der Typ des zu erzeugenden Attributs – nämlich Boolean – bereits im Voraus feststeht. Deshalb und weil sowohl untere als auch obere Grenze der Multiplizität stets 1 sind, entfällt der aufwändige Prozess des Verschmelzens von Attributen. Anstelle davon kann direkt für eine semantische `Visibility`-Definition ein Boolean-Attribut abgeleitet und dem betreffenden Meta-Konzept hinzugefügt werden. Die fixe Vorgabe der Multiplizität ist vertretbar, da ein nicht gesetztes boolesches Attribut bei der Visualisierung nicht von der Zuweisung des Werts `false` unterschieden werden kann. Denn in beiden Fällen würde der betroffene Knoten nicht angezeigt werden. Standardmäßig erhält das Attribut den (konfigurierbaren) Namen „enabled“. Für die Verbindung von konkreter und abstrakter Syntax muss überdies ein entsprechendes Direct-Mapping erzeugt werden.

Eigenschaften mit Wertebeziehungen

Nicht beachtet wurde bisweilen die Möglichkeit, dass eine semantische Eigenschaft auch mit Wertebeziehungen annotiert sein kann. In einem solchen Fall wird anstelle eines Direct-Mappings ein Altering-Mapping erzeugt und die vorhandenen `ValueRelations` von der betreffenden Eigenschaft entfernt und dem Altering-Mapping zugeteilt. Treten Werte auf Visualisierungsseite auf, zu denen es keine `ValueRelation` gibt, dann werden entsprechende Wertebeziehungen ergänzt. Darüber hinaus wird auch der Attributname standardmäßig auf „kind“ (dt. Art, Sorte) gesetzt.

Ein besonderes Szenario im Zusammenhang mit Wertebeziehungen soll anhand eines Beispiels (und unter Vorgriff der erst in Abschnitt 6.1.1 eingeführten Styling-Eigenschaft `ForeColor`) in Abbildung 5-19 demonstriert werden. Die einzelnen Bereiche zeigen lediglich ein bestimmtes Fragment der beteiligten Modelle. Den Anfang macht das visuelle Definitionsmodell, das durch freies Modellieren erstellt wurde. Ausgangspunkt ist die semantische, mit einer `NameAnnotation` und drei `ValueRelations` versehene Eigenschaft `color`. Die Wertebeziehungen drücken aus, dass der Wert `ERROR` der Farbe Rot, der Wert `WARN` der Farbe Gelb und der Wert `HINT` der Farbe Weiß entspricht.

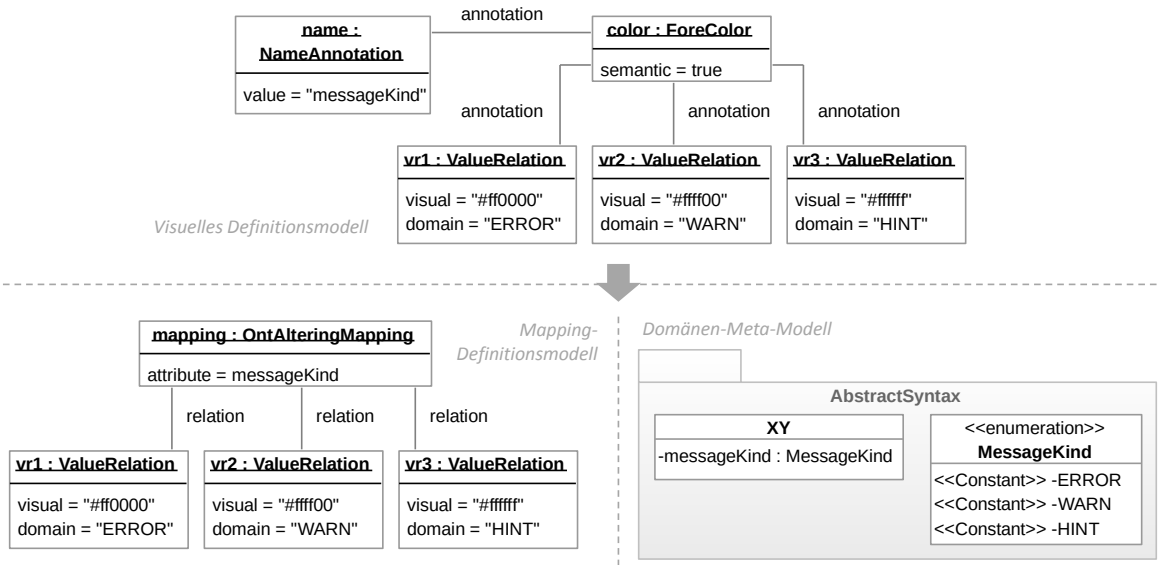

Abbildung 5-19: Beispielhaftes Szenario für den Einsatz von `ValueRelations`

Während der Durchführung des Ableitungsprozesses resultieren die unten abgebildeten Artefakte von Mapping-Definitionsmodell sowie Domänen-Meta-Modell. Auf Mapping-Seite wird ein `ValueMapping` angelegt und mit den bereits vorhandenen `ValueRelations` verbunden. Zur Vermeidung einer erneuten Verarbeitung dieser Annotationen werden sie zusammen mit `name` von der Eigenschaft `color` losgelöst. Innerhalb der abstrakten Syntax wird für die semantische Farbeigenschaft ein entsprechendes Attribut generiert und einem nicht näher spezifizierten (weil für dieses Beispiel irrelevanten) Konzepts `XY` hinzugefügt. Nachdem die einzelnen Domänenwerte der verschiedenen Wertebeziehungen dem regulären Ausdruck für Enumerationsliterale genügen, wird als Besonderheit eine Enumeration erzeugt, die diese Werte als Literale besitzt. Ihre Bezeichnung basiert auf der zu `color` gehörenden `NameAnnotation`.

Reference-Inferrer

Der Reference-Inferrer erzeugt für jede Verwendung einer semantischen `Reference`-Definition standardmäßig zwei Attribute, wobei je eines dem Quell- und eines dem Zielkonzept zugerechnet wird. Als Typ des Quell-Attributs wird das Meta-Konzept des Ziels verwendet, wohingegen beim Ziel-Attribut das Meta-Konzept der Quelle zum Einsatz kommt. Das führt in der Regel dazu, dass jedem beteiligten Meta-Konzept mehrere zusammengehörende Attribute zugesprochen werden, was letztlich der Vorgehensweise im linguistischen Kontext (Abschnitt 4.3) entspricht.

Während der anschließenden Attributverschmelzung kann es deshalb ebenso zur Ausbildung einer Generalisierungshierarchie kommen. Die Multiplizität der Attribute wird anhand der Anzahl der referenzierten Quell- bzw. Zielkonzepte bestimmt. Da die untere Grenze bereits durch den obigen Algorithmus abgedeckt ist, muss nur die obere Grenze betrachtet werden. Sie ist zunächst auf 1 gesetzt und wird nur dann auf * erweitert, sofern es mindestens eine Referenz-Verwendung mit mehr als einem `target` gibt.

Abschließend erfolgt die Benennung der resultierenden Attribute. Dazu wird der Name des Konzepts, das den Attributtyp verkörpert, übernommen, wobei der erste Buchstabe stets kleingedruckt wird. Fällt die Multiplizität größer 1 aus, so wird zum Ausdrücken des Plurals zusätzlich ein „s" an das Ende des Attributnamens angehängt.

Neben der Erweiterung des Domänen-Meta-Modells wird durch die Generierung eines Reference-Mappings eine Verbindung zwischen der semantischen Referenz und dem abgeleiteten Attribut hergestellt.

Containment-Inferrer

Der Containment-Inferrer ist weitgehend identisch mit dem Reference-Inferrer. Er kommt bei semantischen `Containment`-Verwendungen zum Einsatz. Als einziger Unterschied wird das abgeleitete Attribut im Quell-Meta-Konzept zusätzlich als komposit deklariert. Daran wird deutlich, dass Inferrer gezielt Sprach- und Entwurfsmuster anwenden können. Eine automatisierte Einführung von Entwurfsmustern – wie in [22] vorgeschlagen – wird jedoch nur so weit verfolgt, wie sie aus verfügbaren Informationen geschlussfolgert werden kann. Denn jedes derartige Muster dient einem bestimmten Zweck, der ausschließlich dem Benutzer bekannt ist. Es ist also zusätzliches Domänenwissen notwendig, das lediglich ein Mensch beisteuern kann. Durch die Typisierung einer Referenz als `Containment` ist ein solches Wissen vorhanden, weshalb die Auszeichnung eines Attributs als komposit nachvollziehbar ist.

Für Sprach- oder Entwurfsmuster, die nicht in derartiger Klarheit erkannt werden können, kann derselbe Weg wie in Abschnitt 4.4 beschritten werden. Dabei werden dem Benutzer Vorschläge für die Anwendung von jeweils passenden Mustern unterbreitet und er kann dadurch selbst entscheiden, ob er diese übernehmen möchte oder nicht.

5.3.2.3 Beispiel

Zur exemplarischen **Demonstration**, wie die **Ableitung des Domänen-Stacks** auf Basis eines frei modellierten Dokuments erfolgt, wird das Beispiel aus Abschnitt 5.2.3 aufgegriffen. Die beiden visuellen Definitionsmodelle sind bis auf die `DirectionalAnnotation`, die im hiesigen Fall fehlt, identisch. Da das Diagramm (Abbildung 5-20) anstelle von nur einem Prozessschritt zwei Schritte umfasst, fällt auch das visuelle Verwendungsmodell größer aus. Dennoch ist das visuelle Definitionsmodell erneut in Abbildung 5-21 visualisiert. Zusätzlich zu Abbildung 5-12 ist es in Sektionen

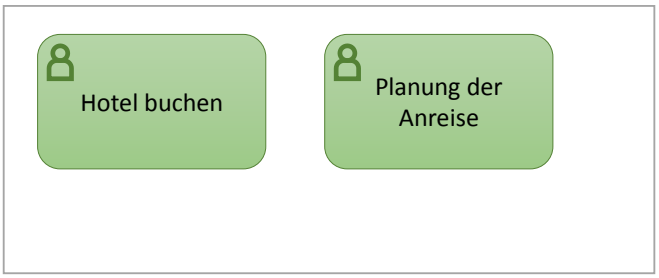

Abbildung 5-20: Frei modelliertes Diagramm mit zwei Prozessschritten

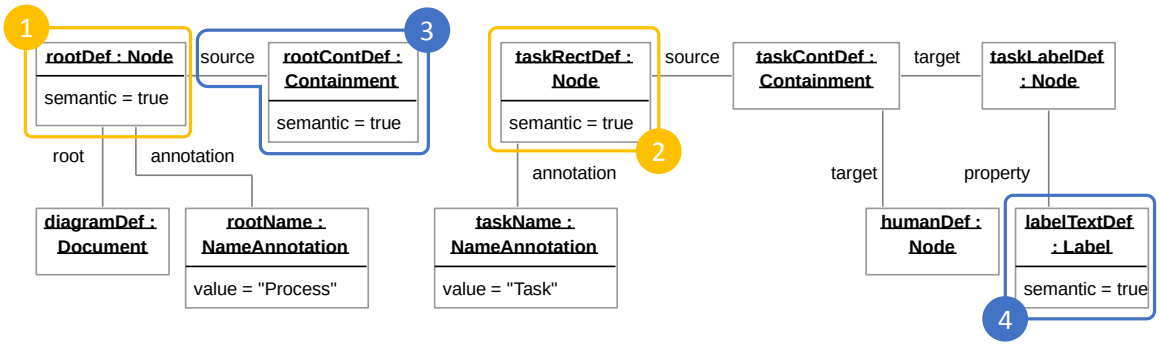

Abbildung 5-21: Anhand der anzuwendenden Inferrers in Sektionen eingeteiltes visuelles Definitionsmodell

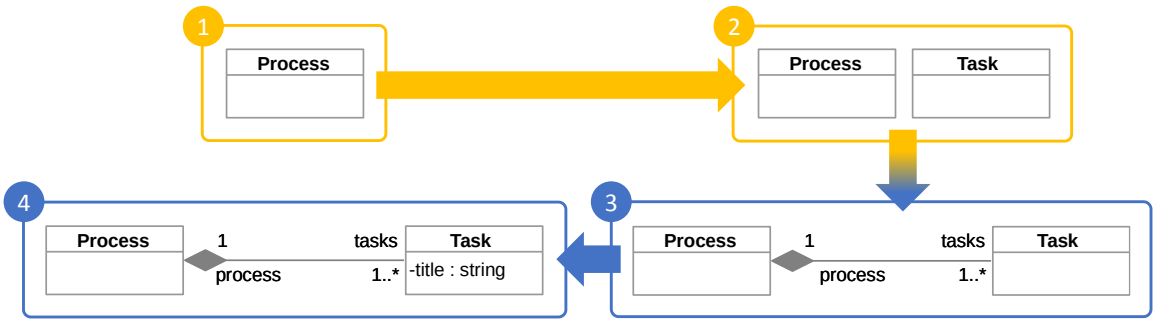

Abbildung 5-22: Zustände der abstrakten Syntax im Verlauf des Ableitungsprozesses

gegliedert, die jeweils der Anwendung eines bestimmten Inferrer angehören. Die Nummerierung gibt Auskunft, in welcher Reihenfolge die einzelnen Sektionen bzw. deren Konzepte vom jeweiligen Inferrer verarbeitet werden. Bevor jedoch ermittelt werden kann, welche Inferrers heranzuziehen sind, muss berechnet werden, worauf Inferrers überhaupt angewendet werden können.

Laut Beschreibung des vorhergehenden Abschnitts sind das zunächst die semantischen Knoten. Im Definitionsmodell sind diese durch eine gelbe Umrandung gekennzeichnet, also `rootDef` und `taskRectDef`. Für beide wird jeweils ein passender **Element-Inferrer** ermittelt und auf die Knoten angewendet, was zur Generierung der Konzepte `Process` und `Task` innerhalb der abstrakten Syntax führt. Dies wird von den ersten beiden Zuständen des Domänen-Meta-Modells in Abbildung 5-22 symbolisiert. Die Namen stammen aus den Annotationen `rootName` und `taskName`. Des Weiteren werden auch entsprechende Element-Mappings erzeugt. Wie diese zu parametrieren sind, wurde bereits in Abschnitt 5.2.3 beispielhaft verdeutlicht. Gleiches gilt auch für die Sub-Mappings, auf die nachfolgend ebenfalls nicht näher eingegangen wird.

Im nächsten Schritt werden alle semantischen Features von `rootDef` extrahiert. Im konkreten Fall ist das lediglich die Enthaltensein-Referenz `rootConfDef`. Für sie muss zunächst ein Sub-Inferrer detektiert werden, was aufgrund des Typs `Containment` in einem **Containment-Inferrer** resultiert. Er generiert im nächsten Schritt die beiden Attribute `tasks` und `process`. Das Attribut `tasks` wird wegen des Enthaltenseins als komposit deklariert und dem Konzept `Process` zugeteilt. Nachdem es – wie Abbildung 5-24 zeigt – eine Verwendung der Referenz gibt, die über mehr als ein `target` verfügt, wird die obere Grenze der Multiplizität des Attributs auf * gesetzt. Die untere Grenze bleibt bei 1, weil für das vorgestellte Beispiel ausschließlich das obige Beispieldiagramm existiert und darin der einzig vorhandene Prozess genau zwei Tasks enthält. Das ebenso generierte Attribut `process` wird dem Konzept `Task` hinzugefügt und erhält die Multiplizität 1. Jede Aufgabe ist folglich genau einem Prozess zugeordnet.

Dem gleichen Vorgang wird auch der Knoten `taskRectDef` unterzogen. Hierbei wird zunächst die Eigenschaft `labelTextDef` als semantisches Feature identifiziert und als passender Sub-Inferrer wegen des Typs `Label` der **Label-Inferrer** ausgewählt. Bei Anwendung dieses Inferrers auf das betreffende Konzept `Task` wird darin ein Attribut mit dem Namen `title` erzeugt. Als Datentyp kommt String zum Einsatz, weil allen existierenden Verwendungen von `labelTextDef` – nämlich

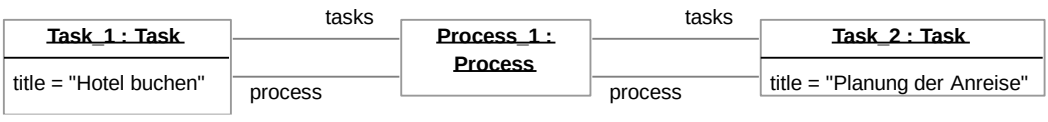

Abbildung 5-23: Generiertes Domänenmodell, das dem obigen Beispiel entspricht

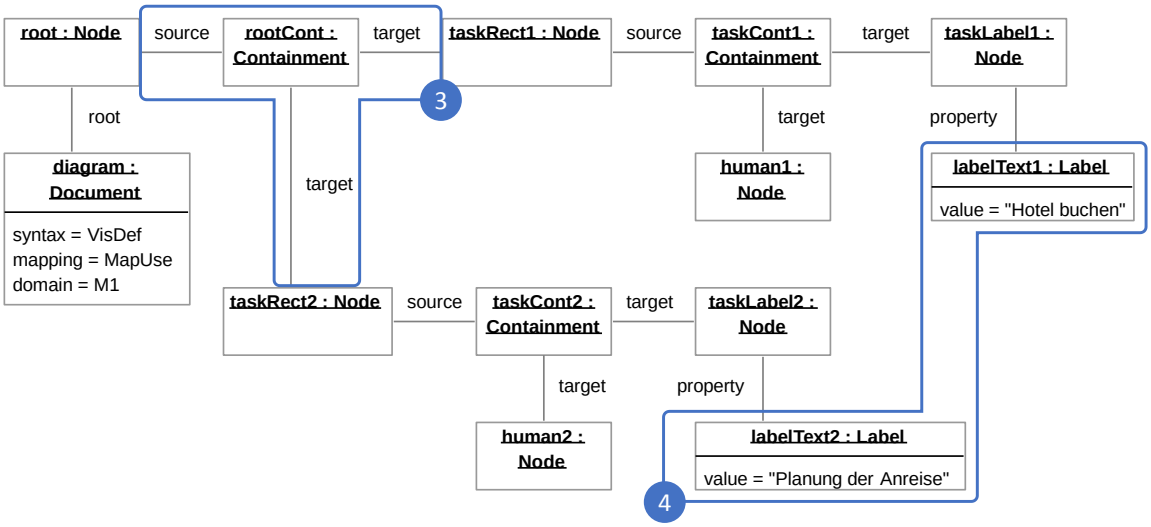

Abbildung 5-24: Anhand der angewendeten Inferrers in Sektionen eingeteiltes visuelles Verwendungsmodell

`labelText1` und `labelText2` in Abbildung 5-24 – als `value` ein Text zugewiesen ist, der lediglich als Zeichenkette abbildbar ist. Zudem bleibt die untere Grenze der Multiplizität des Attributs unangetastet, da alle Prozesskonstrukte über eine entsprechende Zuweisung in Form einer Eigenschaft verfügen.

Zum Schluss müssen noch das Mapping-Verwendungs- und das Domänenmodell befüllt werden. Nachdem sich die Parametrierung der Mapping-Verwendungen in Abschnitt 5.2.3 als durchgängig trivial herausgestellt hat, beschränken sich die Ausführungen hier auf das **Domänenmodell**. Es ist in Abbildung 5-23 dargestellt und beinhaltet die Konzepte `Process_1`, `Task_1` und `Task_2`. Die beiden Aufgaben besitzen jeweils eine Zuweisung an das `title`-Attribut, wobei der zugewiesene String dem Wert der entsprechenden Eigenschaft (`labelText1` bzw. `labelText2`) des visuellen Verwendungsmodells entspricht. Außerdem gibt jede Aufgabe das Konzept `Process_1` als ihren `process` an. Dieser wiederum verweist via `tasks` sowohl auf `Task_1` als auch auf `Task_2`.

5.4 Konstruktivistisches, inkrementelles Ändern einer DSML

Inkrementelles Ändern einer DSML erfolgt, wie auch deren initialer Entwurf, **in zwei Schritten**. Als erstes wird die konkrete Syntax modifiziert, indem der Benutzer frei modelliert. Analog zu Abschnitt 5.3.1.4 kann er ebenso visuelle Konstrukte annotieren und dadurch direkt Einfluss auf den sich anschließenden Ableitungsschritt nehmen. In diesem werden die Änderungen an der konkreten Syntax auf die abstrakte Syntax übertragen. Demgemäß beschäftigt sich der erste Unterabschnitt mit der Änderbarkeit der konkreten Syntax, während im zweiten Unterabschnitt erläutert wird, welche Auswirkungen die jeweiligen Änderungen der konkreten Syntax auf die abstrakte Syntax haben. Analog zu Abschnitt 5.3 bildet auch dieser Abschnitt einen essentiellen Bestandteil des eigenen, konzeptuellen Beitrags (Abschnitt 1.3).

Nachdem der Fokus auf der konkreten Syntax liegt, werden im Folgenden in erster Linie das visuelle Definitionsmodell sowie das via Mapping-Definitionen angebundene Domänen-Meta-Modell betrachtet. Werden Informationen aus den Verwendungsmodellen benötigt, dann ist dies explizit vermerkt.

5.4.1 Änderbarkeit der konkreten Syntax

In Abschnitt 5.3 wurde beschrieben, welche Interaktionsmöglichkeiten mit der konkreten Syntax zur Verfügung stehen, wenn noch kein Mapping-Definitionsmodell angebunden ist. Liegen aber bereits Mapping-Definitionen vor, so müssen diese berücksichtigt werden, weil dadurch bestimmte Änderungen innerhalb des Dokuments nicht länger zulässig sind. Welche Modifikationen prinzipiell möglich und unter welchen Umständen diese erlaubt sind, wird im Folgenden diskutiert.

An sämtlichen Änderungen – mit Ausnahme der Anpassung des Knotentyps – ist immer ein Feature beteiligt. Es ist also zunächst zu klären, ob eine existierende Knotendefinition um ein **neues Feature** erweitert werden darf. Im Falle von rein visuellen Features ist das immer gestattet, weil sie keine Auswirkung auf die Domäne haben und somit komplett ignoriert werden können. Neue semantische Features führen zwar zur Einführung zusätzlicher Attribute innerhalb der abstrakten Syntax, doch bleiben bereits vorhandene Attribute davon gänzlich unberührt. Deshalb dürfen auch solche Features ohne Einschränkung ergänzt werden.

Des Weiteren erlaubt ist das **Zuweisen potentiell beliebiger Werte** zu Eigenschaften, unabhängig davon, ob sie rein visuellen oder semantischen Charakter aufweisen. Analoges gilt für die Festlegung von Zielknoten für Referenzen. Hierbei ist lediglich zu beachten, dass rein visuelle Referenzen nur auf Zielknoten zeigen dürfen, die dem gleichen Konstrukt angehören, während semantische Referenzen auf Knoten verweisen müssen, die Teil eines anderen Konstrukts sind. Ein Bruch mit der abstrakten Syntax ist bei semantischen Features deshalb nicht zu erwarten, da höchstens Typ und Multiplizität der zugehörigen Attribute beeinflusst werden. Nachdem zwischenzeitlich das Meta-Modell manuell modifiziert worden sein kann, darf eine derartige Änderung ausschließlich in einer größeren Bandbreite zulässiger Typen (sprich Wertebereich) sowie einer höherwertigen Multiplizität resultieren.

Als dritte Änderungsmöglichkeit im Zusammenhang mit Features muss der **Wechsel zwischen semantisch und rein visuell** betrachtet werden. Dies ist davon abhängig, ob das Feature eine Eigenschaft oder eine Referenz verkörpert. Erstere können beliebig konvertiert werden, da kein Informationsverlust zu erwarten ist. Denn bei semantischen Eigenschaften liegt die Information im Domänenmodell, während sie sich bei rein visuellen Eigenschaften im visuellen Verwendungsmodell befindet. Im Falle von Referenzen muss erneut differenziert werden. Eine Konvertierung ist solange gestattet, wie auf Verwendungsseite keine Zielknoten angegeben sind. Andernfalls ist sie untersagt, da nicht entschieden werden kann, wie mit diesen Zielknoten zu verfahren ist. Daher muss bei mindestens einem vorhandenen Zielknoten die Referenz zunächst manuell gelöscht werden.

Anhand des Beispielprozesses aus Abbildung 5-25 wird die Problematik deutlich. Den beiden Swimlanes liegt dieselbe Definition zugrunde, die unter anderem einen Knoten beinhaltet, der als semantischer Container fungiert. Seine Verwendungen sind in den beiden visualisierten Konstrukten grau hinterlegt. Würde man diesen Container nun in einen rein visuellen Container umwandeln, müssten die Inhalte der beiden Verwendungen gelöscht werden, weil sich beide Inhalte voneinander unter-

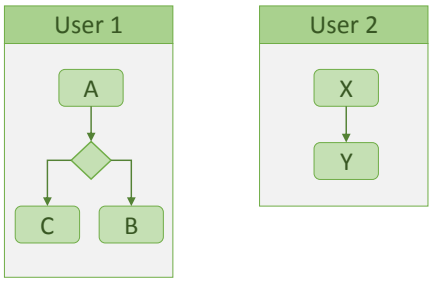

Abbildung 5-25: Konfliktfall für das Konvertieren einer semantischen Referenz in eine rein visuelle Referenz

scheiden. Denn gemäß der Vorgabe aus Abschnitt 5.1.2 muss die rein visuelle Struktur von allen Verwendungen mit der Struktur ihrer jeweiligen Definition übereinstimmen. Ein Informationsverlust wäre also vorprogrammiert, vom Benutzer an der Stelle allerdings völlig unerwartet. Mit einem derartigen Verhalten ist nur dann zu rechnen, wenn die Definition des semantischen Containers an sich gelöscht würde. Dazu muss aber der gesamte Löschprozess explizit vom Benutzer angestoßen werden und nicht durch die nebenläufige Änderung eines booleschen Wertes ausgelöst werden.

Auch die Änderung von **rein visuell zu semantisch** hätte **wenig sinnvolle Ergebnisse** zur Folge. Das Löschen der referenzierten Zielknoten als erste Möglichkeit scheidet aus dem gleichen Grund aus wie zuvor. Alternativ könnten die Ziele als neue, eigenständige visuelle Konstrukte aufgefasst werden. Bei vielen Verwendungen der ursprünglichen Container-Definition würde das allerdings zu einer Unzahl neuer semantischer Knoten führen. Ob die damit einhergehende Annahme, dass eine Entsprechung auf Domänenseite existiert, wirklich zutrifft, ist allerdings fragwürdig, da der Wandel (ebenfalls wie zuvor) durch einen impliziten Mechanismus vollzogen würde.

Der **Wechsel des Typs bei bereits gemappten Knoten** hingegen ist **nicht gestattet**. Dies würde die Intention eines gesamten Konstrukts abändern und meistens sogar dazu führen, dass die bisherige Struktur des Konstrukts invalidiert wird. Dies wiederum hätte zur Folge, dass alle vorhandenen Verwendungen ebenso zerstört würden. Ein Beispiel, in dem die besagte Invalidierung zum Tragen kommt, ist im Grafischen die Umwandlung einer Kante in einen Vertex. Denn Kanten sind stets über ihren Anfangs- und Endpunkt an unterschiedlichen Knoten befestigt, während Vertices typischerweise innerhalb einer Enthaltenseinhierarchie liegen. Es handelt sich aber in jedem Fall um eine andere Art der Einbettung in den Dokumentenkontext als bei Kanten. Die Konvertierung einer Kante in einen Vertex müsste demnach diese Einbettung automatisiert anpassen, was aufgrund der potentiellen Vielfalt an Möglichkeiten nicht umsetzbar ist.

Die letzte Änderungsmöglichkeit ist das **Löschen von Knoten und Features**. Nachdem unter Umständen vorhandene Verwendungen ebenfalls zu löschen sind, kann eine solche Modifikation weitreichende Folgen auf bereits erstellte Dokumente haben. Dennoch ist sie in sämtlichen Fällen gestattet, weil der Benutzer den Löschvorgang aktiv und damit bewusst anstoßen muss.

Verfügt ein Knoten bzw. Feature beim **Löschen** über ein Mapping, dann sind **zwei grundsätzliche Vorgehensweisen** zu unterscheiden. Die Standardvorgehensweise löscht den Knoten nicht direkt, sondern versieht ihn stattdessen mit einer sogenannten Beseitigungs-Annotation. Sie führt erst während des nächsten Ableitungsvorgangs zum Entfernen des Knotens bzw. Features. Nur durch dieses indirekte Verfahren ist es möglich, entsprechende Änderungen auf Domänenseite durchzuführen. Würde man das zum semantischen Knoten gehörende visuelle Konstrukt jedoch sofort löschen, so könnte beim späteren Ableiten diese Änderung nicht mehr rekonstruiert werden. Dies hätte zur Folge, dass die abstrakte Syntax unberührt und die betreffenden Artefakte erhalten blieben. Da das in einigen Fällen durchaus erwünscht sein kann, handelt es sich beim unverzüglichen Löschen des Knotens um die zweite Vorgehensweise, die unterstützt wird. Sollen zu einem zukünftigen Zeitpunkt die verbliebenen Artefakte der abstrakten Syntax trotzdem aufgeräumt werden, dann muss dies manuell geschehen.

Eine Übersicht über die eben vorgestellten Möglichkeiten und Restriktionen zeigt Tabelle 5-1. Sie gibt an, für welche Feature- bzw. Knotenart welche Änderungsoperationen erlaubt und welche verboten sind. Die Farbgebung reflektiert den Freiheitsgrad, wobei grün für einen hohen, gelb für einen mittleren und orange für einen niedrigen Freiheitsgrad steht. Ein grünes X bedeutet volle Unterstützung, ohne jegliche Einschränkung, während ein oranges Minus das genaue Gegenteil meint.

Tabelle 5-1: Freiheitsgrade von Änderungsmöglichkeiten in Bezug zur Feature- bzw. Knotenart

	rein visuelle Eigenschaft	semantische Eigenschaft	rein visuelle Referenz	semantische Referenz	rein visueller Knoten	semantischer Knoten
Erzeugen	X	X	X	X	X	X
Beliebige Zuweisung	unter Berücksichtigung der zulässigen literalen Typen		nur Knoten des eigenen Konstrukts	nur Knoten anderer Konstrukte	–	–
Konvertierung	X	X	nur erlaubt, wenn kein Zielknoten referenziert wird		–	–
Löschen	X	X	X	X	X	X

5.4.2 Ableiten von Änderungen der abstrakten Syntax

Modifikationen der konkreten Syntax – wie im vorherigen Abschnitt vorgestellt – müssen auf die abstrakte Syntax übertragen werden. Im Gegensatz zu Abschnitt 4.5 besitzt die Aussage, dass Änderungen an der abstrakten Syntax minimal gehalten werden sollen, nur bedingt Gültigkeit. Denn der Benutzer kann durch das Annotieren von Knotendefinitionen weitaus gezielter Einfluss auf den Ableitungsprozess nehmen als das im linguistischen Kontext möglich ist.

Dass manuelle **Änderungen nicht durch veraltete Daten überschrieben** werden, wird dadurch erreicht, indem nach der Auswertung einer Annotation diese umgehend deaktiviert wird. Diese Deaktivierung kann z.B. durch Entfernen der betreffenden Annotation umgesetzt werden. Eine andere Möglichkeit ist, Annotationen mit einem entsprechenden booleschen Flag zu versehen, das entsprechend zu setzen ist. Erst durch das explizite erneute Annotieren können Anpassungen der abstrakten Syntax im Verlauf eines weiteren Ableitungsvorgangs erreicht werden.

Hierzu wird das Beispiel aus Abschnitt 5.3.2.3 aufgegriffen und davon ausgegangen, dass zwischenzeitlich das Konzept `Task` manuell durch einen Modellierungsspezialisten in `Activity` umbenannt wurde. Ist der Domänenexperte mit dieser Modifikation nicht zufrieden, so kann er weiterhin während der freien Modellierung den Konzeptnamen beeinflussen und dadurch die Umbenennung rückgängig machen. Dafür dient erneut die schon bekannt Benennungs-Annotation. Fügt der Domänenexperte (geleitet durch eine entsprechende GUI) eine solche Annotation mit dem Wert „Task" hinzu, so wird beim nächstmaligen Ableiten dem besagten Konzept erneut der Name „Task" zugewiesen. Der Name „Activity" ist damit überschrieben.

Annotationen sind eine Sorte von Änderungen an visuellen Konstrukten, die Auswirkungen auf die Domäne hervorrufen können. Eine andere Sorte sind Änderungen, die im Zusammenhang mit semantischen oder gemappten Artefakten stehen, weil diese über eine Entsprechung innerhalb der Domäne verfügen. Grob gesprochen funktioniert die **Erkennung von Änderungen** so, dass nach Inkonsistenzen zwischen den tatsächlich vorhandenen Mapping-Definitionen und den betreffenden semantischen Artefakten des visuellen Definitionsmodells gesucht wird. Welche Inkonsistenzen dabei auftreten können und wie sie automatisiert aufgelöst werden können, wird in den kommenden Unterabschnitten erläutert.

5.4.2.1 Knoten-basierte Änderungen

Nachdem Knoten die zentralen Bausteine von visuellen Konstrukten bilden, werden **Änderungen**, die direkt an ihnen vorgenommen werden können, an erster Stelle behandelt. Dazu werden im ersten Schritt des Aktivitätsdiagramms aus Abbildung 5-26 **alle semantische Knoten** des zugrundeliegenden visuellen Definitionsmodells ausgelesen. Alle anderen Knoten können ignoriert werden, da der Wert des `semantic`-Attributs jedes Knotens bei seiner Initialisierung dauerhaft festgelegt wird.

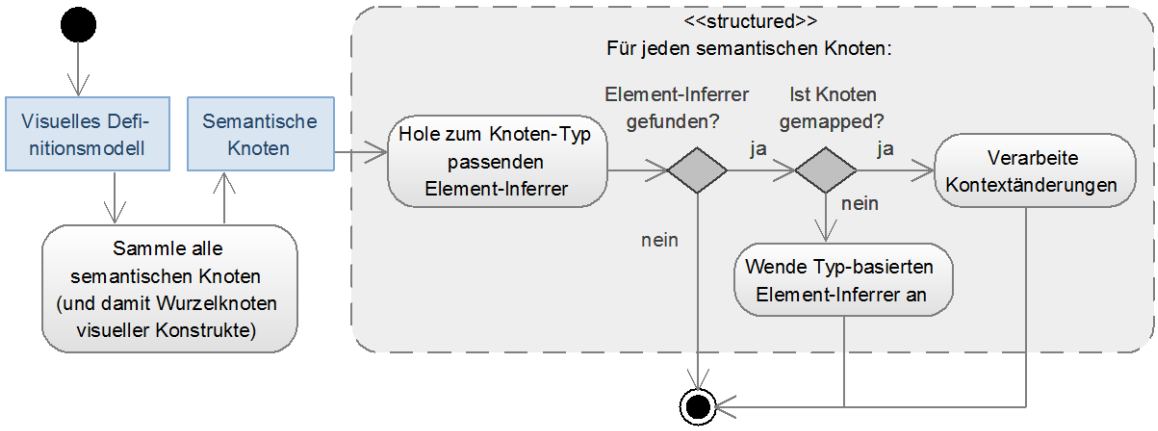

Abbildung 5-26: Ableiten von Knoten-basierten Änderungen

Als nächstes wird für den Typ jedes Knotens ein passender **Element-Inferrer** ermittelt. Ist ein solcher Inferrer vorhanden, muss unterschieden werden, ob der Knoten bereits gemappt ist. Falls nein, ist er seit dem letztmaligen Durchführen des Ableitungsprozesses neu hinzugekommen und kann genauso behandelt werden, wie beim initialen Ableiten der abstrakten Syntax. Das heißt, er wird direkt **auf den Knoten angewendet**.

Verfügt der Knoten jedoch schon über ein Mapping, dann besitzt das vorhandene Mapping weiterhin Gültigkeit. Es sind dann jedoch gegebenenfalls vorkommende **Änderungen im Kontext des Knotens zu verarbeiten**. In erster Linie sind aktive Annotationen auszuwerten, die dem Knoten zugeordnet sind.

Zu berücksichtigen ist hier einerseits die in Abschnitt 5.2.1 eingeführte **Benennungs-Annotation** `NameAnnotation`, die den Namen des Meta-Konzepts auf Domänenseite festlegt. Findet der allgemeine Element-Inferrer eine solche Annotation, dann benennt er das gemappte Meta-Konzept um, wie durch die Annotation vorgegeben (siehe dazu das Beispiel in Abschnitt 5.4.2).

Andererseits kann eine **Beseitigungs-Annotation** an einem semantischen Knoten anhaften, die diesen Knoten als gelöscht markiert. Liegt eine derartige Annotation vor, dann **wendet** sich der zugehörige **Element-Inferrer** selbst **invers an**. Dabei werden alle vom Knoten abhängigen Artefakte eliminiert. Jeder Element-Inferrer sollte also über eine entsprechende Funktionalität verfügen.

5.4.2.2 Feature-basierte Änderungen

Sobald alle semantischen Knoten abgearbeitet und die erforderlichen Element-Inferrer ordnungsgemäß angewendet worden sind, kommen die Feature-Definitionen an die Reihe. Wie der Ablauf in Abbildung 5-27 darlegt, bilden die Grundlage erneut alle semantischen Knoten des visuellen Definitionsmodells. Im Gegensatz zum initialen Ableiten werden hier jedoch **sämtliche Features** des durch den semantischen Knoten definierten visuellen Konstrukts **analysiert**. Denn jede ehemals semantische Eigenschaft mit zugeordnetem Mapping darf laut Abschnitt 5.4.1 in eine rein visuelle Eigenschaft umgewandelt werden. Für eine semantische Referenz gilt dies nur dann, wenn es keine zugehörige Verwendung gibt, die auf einen Zielknoten verweist. Bevor ein Feature allerdings verarbeitet werden kann, muss erst der zugehörige Sub-Inferrer bestimmt werden, der schließlich diese Aufgabe übernimmt.

Existiert kein passender Sub-Inferrer, dann wird zum nächsten Feature übergegangen. Andernfalls muss erneut differenziert werden, und zwar hinsichtlich der Featureeigenschaften „rein visuell" / „semantisch" in Kombination mit „ohne Mapping" / „mit Mapping".

Ist das **Feature rein visuell**, verfügt aber über ein **Mapping**, dann hat eine nachträgliche Änderung von semantisch zu rein visuell stattgefunden. Demzufolge müssen neben den zugehörigen Mappings auch die davon abhängigen Artefakte auf Domänenseite beseitigt werden. Hierzu muss jeder spezialisierte Sub-Inferrer eine entsprechende Operation anbieten, die „inverses Anwenden" genannt wird.

Als weiterer Fall kann ein **semantisches Feature** auftreten, das noch **kein Mapping** aufweist. Dort muss – wie beim initialen Ableiten – einfach der gefundene Sub-Inferrer auf das Feature angewendet werden.

Analog zu den Knoten können auch Features über Annotationen verfügen, die Änderungen an der abstrakten Syntax bewirken. Sie sind insbesondere bei **semantischen Features** relevant, die schon **mit einem Mapping** versehen sind. Folglich müssen Sub-Inferrers in die Lage versetzt werden, an Features angehängte Annotationen zu verarbeiten. Damit ist ein Aspekt möglicher Kontextänderungen abgedeckt. In den jeweiligen Spezialfällen diverser Sub-Inferrer können prinzipiell beliebige Kontextänderungen berücksichtigt werden. Dies zu unterstützen ist jedoch Aufgabe der betreffenden Sub-Inferrer.

Die **Benennungs-Annotation** stellt den trivialsten Vertreter dar, weil sie lediglich dafür sorgt, dass das betreffende Attribut – wie durch die Annotation vorgegeben – umbenannt wird. Ähnliches trifft auch für die `DirectionalAnnotation` zu. Allerdings kann sie ebenso dazu führen, dass ein neues Attribut angelegt bzw. ein bereits vorhandenes gelöscht wird (einschließlich aller assoziierten Zuweisungen).

Beim Vorfinden einer **Beseitigungs-Annotation** muss das Feature selbst sowie alle davon abhängigen Artefakte gelöscht werden. Hierzu wird der zum Typ des Features passende Sub-Inferrer ermittelt und daraufhin invers angewendet. Eine Sonderstellung nehmen gemappte Referenzen mit Beseitigungs-Annotation ein. Wird nämlich eine solche Referenz aufgeräumt, dann können Modellfragmente in den Verwendungsmodellen verbleiben, die nicht länger durch einen ontologischen Editor visualisierbar sind. Gelöscht werden können sie demnach ausschließlich mithilfe eines linguistischen Modelleditors [66]. Dies tritt auf, wenn ein verwendetes Konstrukt nicht länger Ziel einer Referenz-Verwendung ist. Derartig verwendete Konstrukte müssen deshalb während des Aufräumvorgangs ebenfalls aus dem

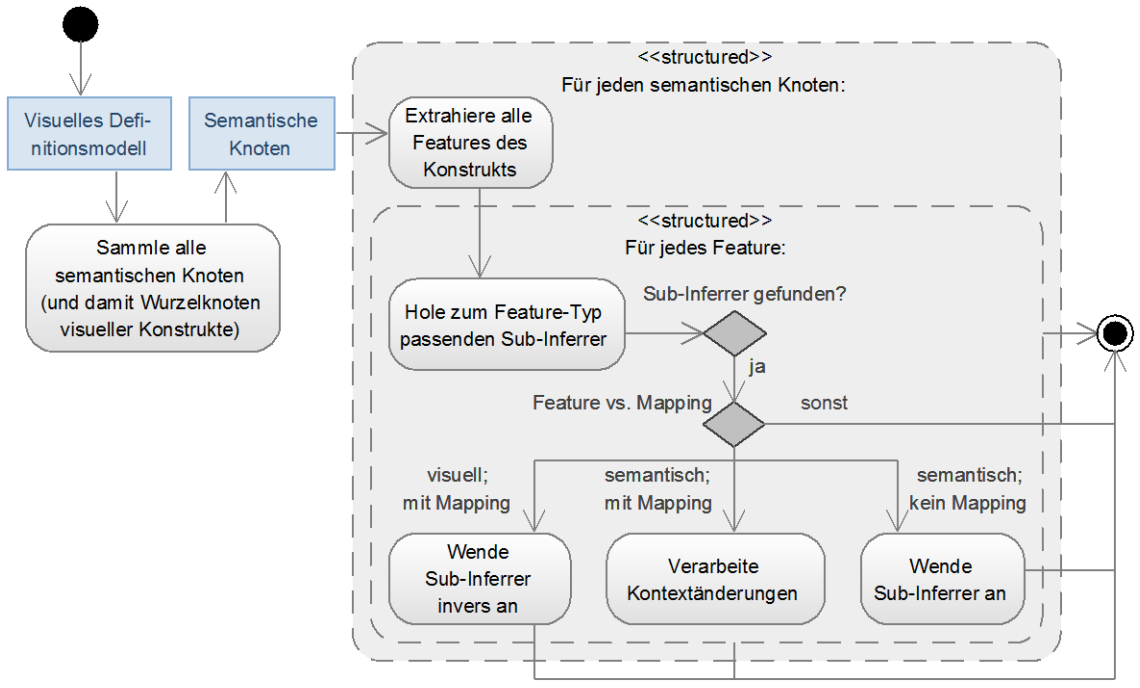

Abbildung 5-27: Ableiten von Feature-basierten Änderungen

jeweiligen visuellen Verwendungsmodell entfernt und zugehörige Artefakte (Mapping-Verwendungen sowie Elemente des Domänenmodells) gleichzeitig eliminiert werden. Bezogen auf das Beispiel aus Abschnitt 5.3.2.3 bedeutet es, dass beim Löschen der Referenz `rootContDef` (Abbildung 5-21) simultan im zugehörigen visuellen Verwendungsmodell (Abbildung 5-24) die Knoten `taskRect1` und `taskRect2` (inklusive deren Kinder) entfernt würden. Die Definition des Konstrukts `task-RectDef` innerhalb des Definitionsmodells bliebe jedoch erhalten, so dass eine Wiederverwendung an anderer Stelle möglich ist.

Zudem kann es bei semantischen Features mit Mapping vorkommen, dass **Typ und Multiplizität** des gemappten Attributs nicht mehr mit den Gegebenheiten des visuellen Definitionsmodells übereinstimmen. Ist dem so, dann müssen die abstrakte Syntax und insbesondere das betreffende Attribut entsprechend angepasst werden. Damit bereits existierende Domänenmodelle nicht invalide werden, dürfen die durch Multiplizität und Typ vorgegebenen Restriktionen lediglich aufgeweicht und in Richtung eines größeren Wertebereichs verschoben werden (analog zum linguistischen Kontext in Abschnitt 4.5.1). Ein typisches Beispiel ist, wenn ein Prozess in Form eines semantischen Containers bisher nur Aufgaben aufnehmen konnte, jetzt aber auch Start-Ereignisse enthalten darf. Der Referenztyp des betreffenden Attributs in der abstrakten Syntax muss dann auf ein Basiskonzept gesetzt werden, das den Konzepten Aufgabe und Start-Ereignis zugrunde liegt.

Als **Sonderfall** zu beachten sind Attribute, die eine **Enumeration als Datentyp** und demzufolge zu einer Eigenschaft mit entsprechenden Wertebeziehungen gehören. Sind neue Wertebeziehungen dazugekommen, dann ist die Enumeration um die betreffenden Literale zu erweitern. Im Fall, dass die neuen Domänenwerte nicht mehr dem regulären Ausdruck für Enumerationsliterale genügen, wird das Attribut in ein String-Attribut konvertiert. Sofern die Enumeration nicht anderweitig innerhalb des Modellrepositoriums Verwendung findet, wird sie obendrein zum Löschen vorgeschlagen. Die betreffende Funktionalität ist vom Label-Inferrer zur Verfügung zu stellen.

5.5 Optimierung der generierten DSML

Die abgeleitete **abstrakte Syntax** kann erneut **Optimierungspotential** besitzen, und zwar aus denselben Gründen wie in Abschnitt 4.4 geschildert. Demzufolge kann auf die im besagten Abschnitt vorgestellten Strategien zurückgegriffen werden, um Empfehlung für die Verfeinerung der abstrakten Syntax zu unterbreiten. Des Weiteren werden auch die Änderungsvorschläge berücksichtigt, die in Abschnitt 4.5.2 vorgestellt wurden. In beiden Fällen handelt es sich um eine nachgeschaltete Verarbeitung der generierten Sprache.

Neben der abstrakten Syntax wird im ontologischen Kontext auch die **konkrete Syntax** erzeugt. Diese Erzeugung erfolgt zwar manuell durch einen Menschen, doch heißt das nicht, dass sie damit völlig frei von Fehlern ist. Insbesondere bei einer Vielzahl visueller Konstrukte verliert man leicht den Überblick, was dazu führen kann, dass **gleich aussehende bzw. gleich strukturierte Konstrukte** definiert werden. Wie in Abschnitt 5.3.1.3 beschrieben, wird eine derartige Äquivalenz zumeist automatisch festgestellt und der Benutzer darauf **hingewiesen**. Er kann dann eigenständig entscheiden, ob er die äquivalenten Konstrukte auf eine gemeinsame Definition zurückführen will.

Existiert zu einem **visuellen Konstrukt keine Verwendung** mehr, dann wird der Benutzer ebenso darauf aufmerksam gemacht. Es obliegt daraufhin ihm, ob er das betreffende Konstrukt beibehält oder aus der Sprache entfernt.

5.6 Kompensation von lückenhaftem Domänenwissen

Das automatisierte Ableiten einer DSML aus einer Menge von Beispielmodellen beruht neben den Modellen oft auf zusätzlichen Annahmen und Hintergrundwissen. Hiervon nicht betroffen sind die Konstrukte der konkreten Syntax, da ihre Struktur identisch ist mit den zugehörigen Verwendungen innerhalb der Dokumente. Diese wird direkt vom Benutzer festgelegt, wodurch angenommen werden kann, dass visuelle Konstrukte in ihrer Darstellung und Struktur vollständig definiert vorliegen. Somit erfolgt im Folgenden eine **Beschränkung auf die abstrakte Syntax**. Nachdem der ontologische Kontext im Hinblick auf die abstrakte Syntax als Erweiterung des linguistischen Kontexts betrachtet werden kann, können die gleichen Fälle von Mehrdeutigkeiten auftreten, die es zu beachten gilt. Deshalb kommen hierbei dieselben Strategien zum Einsatz, die schon in Abschnitt 4.6 vorgestellt wurden.

Benennungen von Meta-Konzepten und Attributen können im linguistischen Kontext direkt aus den Beispielmodellen (Typ- bzw. Zuweisungsnamen) abgelesen werden. Diese Informationen müssen im ontologischen Kontext in einem zusätzlichen Schritt bereitgestellt werden. Bedingt ist das durch die Vorgehensweise bei der freien Modellierung und der damit einhergehenden Bestimmung der konkreten Syntax (Abschnitt 5.3.1.1). Hierbei werden neue visuelle Konstrukte stets in Form von Verwendungen deklariert, zu denen automatisch eine entsprechende Definition (das visuelle Konstrukt) erzeugt wird. Diese Definition ist die Grundlage eines entsprechenden Konstrukts innerhalb der abstrakten Syntax. Aufgrund der **automatisierten Erstellung des visuellen Konstrukts** kann lediglich ein **computergenerierter Name** für das Pendant auf Domänenseite angegeben werden.

Die jeweilige Strategie zur Namenserzeugung kann über eine externe Konfiguration global festgelegt werden. Dennoch wird sie in den seltensten Fällen passende Ergebnisse liefern, weshalb dem Benutzer die **Empfehlung** unterbreitet wird, den **generierten Namen** einem **Review** zu unterziehen und bei Bedarf zu ändern. Die manuelle Benennung eines visuellen Konstrukts erfolgt stets unter Angabe einer entsprechenden Annotation (siehe dazu auch Abschnitt 5.3.1.4). Gleiches gilt auch für semantische Features. Üblicherweise kongruieren sie mit Attributen auf Domänenseite, wobei der Name von einer entsprechenden, am Feature anhaftenden Annotation übernommen wird. Jede computergenerierte Bezeichnung sollte grundsätzlich von einem Experten überprüft werden, so dass stets für die Domäne geeignete Namen Verwendung finden.

Neben der Benennung verschiedener Artefakte des Domänen-Meta-Modells kann mithilfe von **Annotationen** auch **anderweitig** Einfluss auf den Ableitungsprozess genommen und damit **fehlendes Domänenwissen kompensiert** werden. Weitere Beispiele für Annotationen, die schon im allgemeingültigen visuellen Meta-Modell (Abschnitt 5.2.1) bereitgestellt werden, sind Wertebeziehungen und die `DirectionalAnnotation`.

5.7 Zusammenfassung

Dieses Kapitel befasst sich vornehmlich mit einer konzeptuellen Lösung für **Anforderung 1**, bei der es um die konstruktivistische Entwicklung von DSMLs im ontologischen Kontext geht. Wie schon in Kapitel 4, das die Konstruktion von DSMLs im linguistischen Kontext behandelt, wird auf die Entwicklung mithilfe von Beispielen gesetzt. Allerdings muss im ontologischen Kontext auch die konkrete Syntax berücksichtigt werden, da diese zu Beginn noch nicht vorliegt und später bei Bedarf modifiziert werden kann.

Nachdem der Benutzer stets mit visuellen Objekten interagiert, bildet die konkrete Syntax die Schnittstelle zwischen Mensch und Modellierungssprache. Es kann zwar zwischen grafischer und textueller

konkreter Syntax differenziert werden, doch lassen sich beide Welten mit weitgehend demselben visuellen Meta-Modell beschreiben. Dieses Meta-Modell bildet die Grundlage für ontologisches Modellieren, unabhängig von einer greifbaren Visualisierung. Der Vorteil davon ist, dass die Verfahren zur inkrementellen Ableitung einer abstrakten Syntax generisch für grafische und textuelle Sprachen einsetzbar sind. Diese Erkenntnis sowie die zugehörige Umsetzung repräsentieren einen zentralen Teil des eigenen, konzeptuellen Beitrags der vorliegenden Dissertation. Nicht übertragbar ist die Erkenntnis auf den Modellierungsworkflow, der spezifisch für die jeweilige Welt ist. Demzufolge wird Anforderung 1 in diesem Kapitel nicht vollständig erfüllt. Eine Behandlung der arbeitsflussspezifischen Teile erfolgt erst in den Kapiteln 6 und 7.

Der in **Anforderung 3** geforderte Qualitätsaspekt wird auch in diesem Kapitel **adressiert**. Neben der abstrakten Syntax wird ebenso die konkrete Syntax nach Stellen mit Optimierungspotential durchsucht. Werden solche Stellen gefunden, erfolgt die Ausgabe entsprechender Verfeinerungsempfehlungen.

6 Beispielgetriebene Entwicklung im grafisch ontologischen Kontext

Dieses Kapitel befasst sich mit der grafischen Spezialisierung der beispielgetriebenen Entwicklung von DSMLs im ontologischen Kontext, die von Anforderung 1a gefordert wird. Hierfür reichen die meisten der im vorherigen Kapitel eingeführten Mechanismen und Konzepte aus. Die zentrale softwaretechnische Neuerung, auch gegenüber existierenden Lösungen, bildet der grafische Modellierungsworkflow in Abschnitt 6.2. Analoges trifft für den textuellen Workflow in Kapitel 7 zu, denn er steht in Abhängigkeit zu der jeweiligen Eingabemethode, die wiederum spezifisch für die betreffende Visualisierungsform ist.

Bevor allerdings auf den Modellierungsworkflow eingegangen wird, erfolgt eine Beschreibung der Anpassung des visuellen Meta-Modells für grafische Belange. Zudem kommen im Anschluss daran ergänzende Ausführungen zum allgemeinen Mapping Meta-Modell, die spezifisch für die grafische Visualisierungsform sind. Nach der Abhandlung des Workflows wird die für Diagramme spezifische Erweiterung des Ableitungsprozesses vorgestellt. Beschränkt wird sich dabei auf die inkrementelle Ableitung von Änderungen, weil diese gleichermaßen zutreffen, auch wenn noch keine DSML zugrunde liegt. Abschließend werden Modellierungsungenauigkeiten näher betrachtet und wie diese gegebenenfalls kompensiert werden können.

6.1 Erweiterung der Meta-Modelle für grafische Syntaxen

Ergänzend zu den Grundlagen der beispielgetriebenen Entwicklung müssen die Meta-Modelle spezifische Belange grafischer Sprachen [11, 78] erfüllen. In erster Linie handelt es sich hierbei um die Möglichkeit, das Aussehen grafischer Elemente beeinflussen zu können. Typischerweise geschieht das mithilfe von grafischen Eigenschaften wie Form, Vordergrundfarbe, Hintergrundfarbe, Liniendicke und Schriftart.

Unterschiedliche Ausprägungen der einzelnen Eigenschaften können auch auf Unterschiede innerhalb der Domänenmodelle hindeuten. Als Beispiel wird erneut die Prozessdomäne betrachtet, in der Aufgaben als Rechtecke mit abgerundeten Ecken visualisiert werden. Besitzt eine solche Aufgabe eine blaue Umrandung, so kann es bedeuten, dass sie von einem Menschen zu erledigen ist. Ist die Umrandung hingegen grün, so handelt es sich um eine vom Rechner auszuführende Aktion. Um diesen Sachverhalt sinnvoll auf die Domäne abbilden zu können, ist es notwendig, dass der Benutzer angibt, welche Bedeutung sich hinter den einzelnen Farben verbirgt. Andernfalls können lediglich die im Regelfall wenig aussagekräftigen Werte der visuellen Eigenschaft übernommen werden (im Beispiel die Farbwerte der Umrandung). Mehr dazu folgt in Abschnitt 6.3.2.

Von Relevanz ist neben diesen Eigenschaften auch die räumliche Lage der einzelnen Elemente. Ein Mechanismus, der sich der Positionierungsproblematik annimmt, wird im Umfeld grafischer Editoren üblicherweise als Layouting bezeichnet [77].

Bei der Positionierung sind im Grafischen zwei Arten von Elementen zu unterscheiden. Einerseits gibt es die eigenständigen Formen (auch Vertices genannt) und andererseits verbindende Objekte (auch

Kanten genannt), die im Regelfall als Verbindungslinien visualisiert werden. Die Abbildung verbindender Objekte auf Artefakte der abstrakten Syntax nimmt ebenfalls einen Sonderfall ein, der anhand eines Beispiels in Unterabschnitt 6.1.2 beschrieben wird.

6.1.1 Grafisch-visuelles Meta-Modell

Das grafisch-visuelle Meta-Modell stellt eine Erweiterung des allgemeinen visuellen Meta-Modells (Abschnitt 5.2.1) im Hinblick auf den grafisch ontologischen Kontext dar. Aus Gründen der Einfachheit wird es auch grafisches Meta-Modell (Abbildung 6-1) genannt.

Aufgrund des Type Object Patterns [71], das für die Typisierung von Knoten zum Einsatz kommt, werden zum Zweck der **Differenzierung von eigenständigen Formen und dazwischenliegenden Verbindungen** zwei Spezialisierung von `NodeType` eingeführt. Eigenständige Formen werden folglich durch das Konzept `Vertex` repräsentiert, wohingegen Verbindungen durch das Konzept `Edge` ausgedrückt werden. Da diese Typen global gültig sind und nicht parametriert werden müssen, können sie in Form von Singleton-Instanzen global zur Verfügung gestellt werden.

Die **Befestigung von Verbindungen** an anderen Knoten erfolgt unter Verwendung der **speziellen Referenzen** `Outgoing` und `Incoming`. Die Richtung beider Referenzen geht darin stets von den Vertices (`source`) zu den Kanten (`target`). Insgesamt ist auf diese Weise auch die Richtung der Verbindung festgelegt. Sie startet bei dem Knoten, der Eigentümer der `Outgoing`-Instanz ist, und mündet in den Knoten, der über die zugehörige `Incoming`-Instanz verfügt.

Darüber hinaus sind fraglos **weitere Referenztypen denkbar** (z.B. Ankleben oder Überlappen). Sie können aber durch die vorgestellte Vorgehensweise der Typspezialisierung analog integriert werden und stellen deshalb keine zusätzliche konzeptuelle Herausforderung dar. Außerdem genügt die Graph-

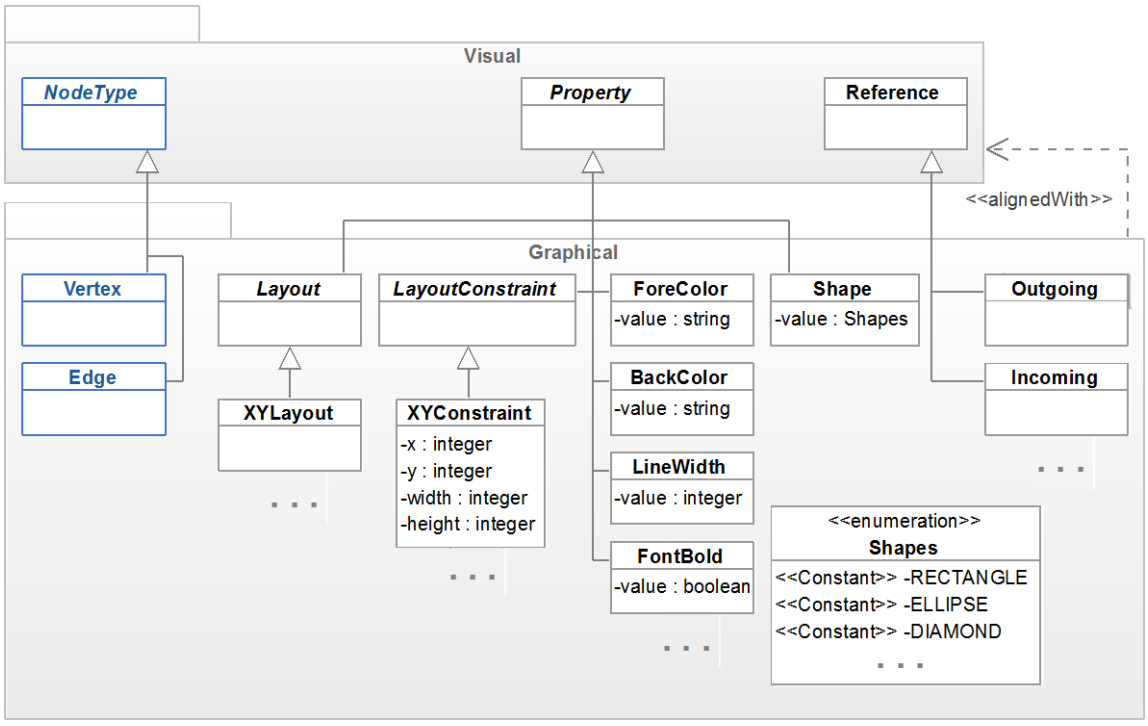

Abbildung 6-1: Erweiterung des allgemeinen visuellen Meta-Modells für konkrete grafische Syntaxen

basierte Visualisierungsklasse zusammen mit der Möglichkeit, grafische Elemente ineinander zu verschachteln, für die Umsetzung einer Vielzahl grafischer Sprachen. Wie die Autoren von [4] und [30] zeigen, beschränkt sich nämlich ein Großteil der existierenden grafischen DSMLs auf diese beiden Beziehungsarten.

Die verschiedenen **Styling-Möglichkeiten** sind in Form von separaten `Property`-Spezialisierungen realisiert. Beispielhaft abgebildet sind hierfür `Shape`, `ForeColor`, `BackColor`, `LineWidth` und `FontBold`. Zahlreiche weitere Styling-Eigenschaften werden natürlich unterstützt und können bei Bedarf ergänzt werden. Dass auch Formen über ein `value`-Attribut gesteuert werden, liefert den Vorteil, sie im Zusammenhang mit `ValueRelations` auf die gleiche Weise wie die übrigen Styling-Eigenschaften verarbeiten zu können. Um weitere Formen für grafische Syntaxen bereitzustellen, muss lediglich die Enumeration `Shapes` erweitert werden.

Zur **Umsetzung verschiedener Layouting-Strategien** gibt es überdies die beiden abstrakten Basis-Konzepte `Layout` und `LayoutConstraint`. Dabei bezieht sich das Constraint eines Knoten stets auf das Layout seines Elternknoten und liefert damit Informationen, die die Positionierung und Skalierung des Knoten beeinflussen. Als Beispiel eines konkreten Layouts ist das klassische `XYLayout` mit dem zugehörigen `XYConstraint` abgebildet. Es sorgt dafür, dass Knoten anhand ihrer `x`- und `y`-Koordinaten platziert und durch die Angabe von `width` und `height` mit einer Größe versehen werden. Darüber hinaus sind zahlreiche weitere Layouts mit passenden Constraints vorstellbar, so z.B. ein Layout, das die relative Positionierung von Knoten zueinander ermöglicht. Dieses und weitere Layouts präsentiert Vogler in [141].

6.1.2 Ergänzungen zum Mapping Meta-Modell

Wie in Abschnitt 5.2.2.1 angekündigt, stellen die **Wildcard-Mappings** des Mapping Meta-Modells einen Spezialfall dar, der am leichtesten zu verstehen ist, wenn zusätzliche Informationen aus der grafischen Welt bekannt sind. Gemeint damit ist die Abbildung von Verbindungen zwischen zwei Knoten auf die Domäne.

Motivieren lassen sich diese Mappings anhand eines Beispiels. Hierzu zeigt Abbildung 6-2 ein kleines Entity-Relationship-Diagramm und Abbildung 6-3 eine dazu passende abstrakte Syntax. Letztere besteht aus den Konzepten `Entity`, `Attribute` und `Relationship`, von denen jedes ein Attribut `name` deklariert. Zusätzlich gibt es ein Konzept `Association`, das es erlaubt Beziehungen zwischen Entitäten und Relationen zu spezifizieren. Dieser Umweg ist notwendig, um jede Assoziation mit einer Multiplizität versehen zu können. Attribute dagegen werden direkt von Entitäten referenziert. Im Diagramm sind beide Beziehungsarten jedoch in Form von Verbindungslinien und somit als eigenständige Knoten visualisiert. Demnach muss die Möglichkeit geboten werden, Verbindungen zum einen auf eigenständige Konzepte und zum anderen auf referentielle Attribute bzw. Zuweisungen abbilden zu können.

Die Konzept-bezogene Abbildung kann mithilfe der bereits bekannten Mittel umgesetzt werden. Zur Umsetzung der Attribut-bezogenen Abbildung muss hingegen auf die Konzepte `Wildcard-Element` und `WildcardReference` zurückgegriffen werden. In beiden Abbildungsfällen liegen Instanzen von `Outgoing` und `Incoming` des grafischen Meta-Modells zugrunde. Bei der Attribut-

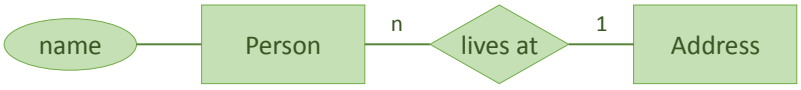

Abbildung 6-2: Beispielhaftes Entity-Relationship-Diagramm

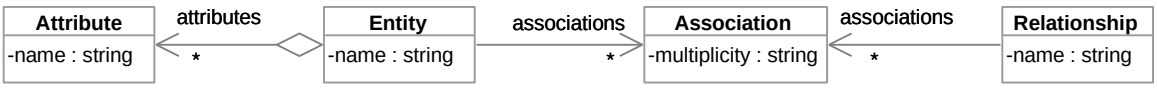

Abbildung 6-3: Rudimentäre abstrakte Syntax zur Modellierung von Entity-Relationship-Diagrammen

bezogenen Abbildung stellt jedoch nur das zur `Outgoing`-Instanz gehörende Referenz-Mapping eine entsprechende Mapping-Funktionalität bereit. Der Aufruf der `map`-Methoden von `WildcardElement` und `WildcardReference` hat folglich keine Auswirkungen auf die betreffenden Domänenmodelle. Damit das aktive Referenz-Mapping (gehört zur `Outgoing`-Instanz) korrekte Resultate liefert, muss die `getDomainElement`-Methode von `WildcardElement` das gemappte Konzept der zugehörigen eingehenden `WildcardReference` zurückliefern. Beispiele für die Umsetzung der beiden Arten von Verbindungsmappings werden in den zwei nachfolgenden Unterabschnitten demonstriert.

6.1.2.1 Beispiel für Attribut-bezogenes Mapping einer Verbindung

Das in Abbildung 6-4 veranschaulichte Modellfragment zeigt einen Ausschnitt des grafischen Definitionsmodells und des Mapping-Definitionsmodells passend zum Entity-Relationship Beispiel aus Abbildung 6-2 und Abbildung 6-3. Der Fokus liegt hierbei auf der Visualisierung der Verbindung zwischen Entitäten und Attributen. Sowohl diese Verbindung (`linkDef`) als auch die visuellen Konstrukte für Entitäten (`entityDef`) und Attribute (`attributeDef`) sind im grafischen Definitionsmodell als semantische Knoten deklariert. Hergestellt wird die Beziehung zwischen den einzelnen Knoten mithilfe der Referenzen `entityOutDef` und `attrInDef`. Durch die Instanziierung der `Reference`-Spezialisierungen `Outgoing` und `Incoming` ist gleichzeitig die Richtung der Verbindung festgelegt, nämlich von den Entitäten hin zu den Attributen.

Im Mapping-Definitionsmodell sind `entityDef` und `attributeDef` wie üblich als `OntElementMapping` definiert. Sie verweisen auf die entsprechenden Konzepte `Entity` und `Attribute` der abstrakten Syntax. Das zu `linkDef` gehörende `WildcardElement` hingegen ist keinem Artefakt der Domäne zugeordnet. Gleiches gilt auch für die zu `attrInDef` gehörende

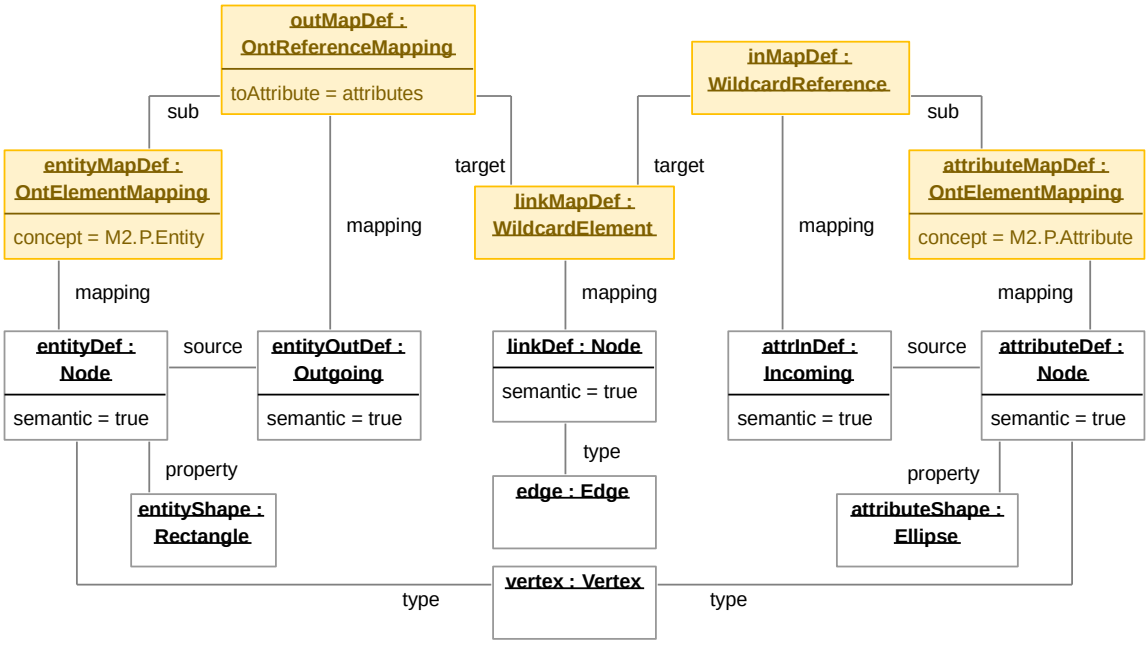

Abbildung 6-4: Beispielhafte Definition eines Attribute-bezogenen Mappings für eine grafische Verbindung

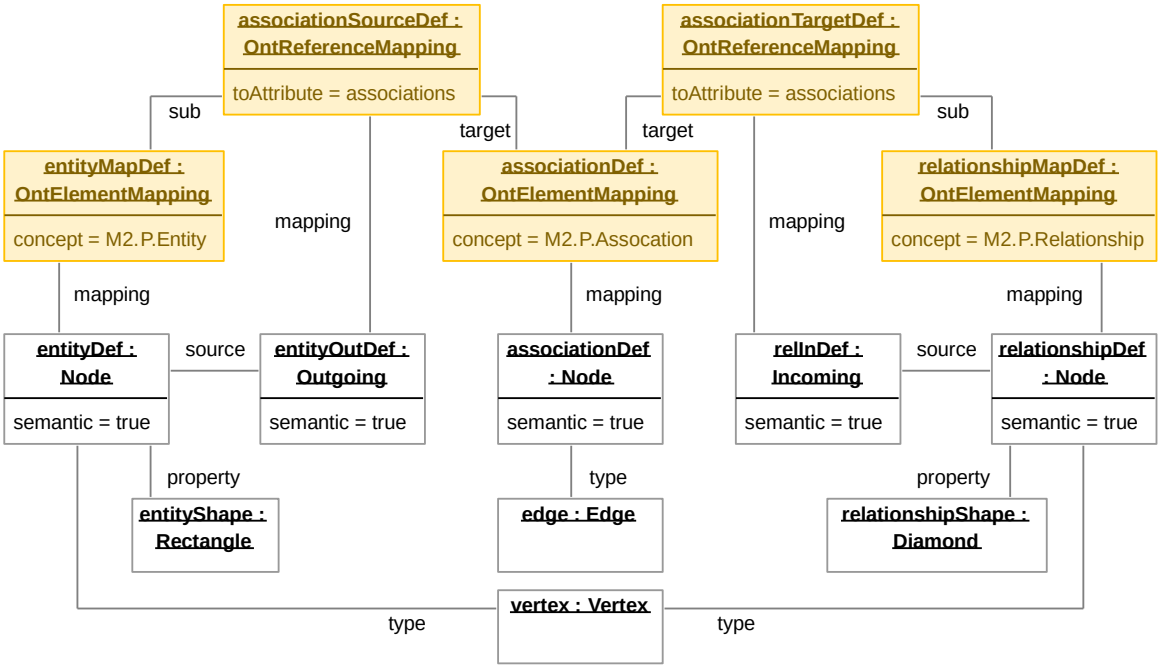

Abbildung 6-5: Beispielhafte Definition eines Konzept-bezogenen Mappings für eine grafische Verbindung

`WildcardReference`. Beide fungieren lediglich als Platzhalter, um dem vorgestellten Mapping-Mechanismus zu genügen. Die eigentliche Mapping-Funktionalität zur Abbildung der Verbindung steckt im Referenz-Mapping `outMapDef`. Es spezifiziert, dass eine `Attribute`-Instance einer `Entity`-Instanz über das Attribut `attributes` zugewiesen wird.

6.1.2.2 Beispiel für Konzept-bezogenes Mapping einer Verbindung

Das Modellfragment in Abbildung 6-5 zeigt einen anderen Ausschnitt des grafischen und des Mapping-Definitionsmodells passend zum Entity-Relationship Beispiel aus Abbildung 6-2 und Abbildung 6-3. Im Fokus liegt jetzt die Visualisierung der Verbindung von Entitäten und Relationships. Die visuelle Seite unterscheidet sich dabei in ihrer Struktur nicht vom Modellfragment aus Abbildung 6-4.

Anders verhält es sich mit den Mappings. Der Knoten `associationDef`, der die Verbindung repräsentiert, wird auf das eigenständige Konzept `Association` der abstrakten Syntax abgebildet. Des Weiteren verfügt jede Referenz über ein aktives Referenz-Mapping. Im Fall von `associationSourceDef` ist `associations` als Attribut einer Entität angegeben, dem ausgehende Assoziationen zugewiesen werden. Das Referenz-Mapping `associationTargetDef` hingegen legt das `associations`-Attribut von Relationships als Ziel für eingehende Assoziationen fest. Auf das Mapping der Multiplizität von Assoziationen wurde aus Gründen der Lesbarkeit verzichtet. Es lässt sich jedoch einfach mithilfe eines Label-Mappings in Kombination mit Wertebeziehungen realisieren.

6.2 Grafischer Modellierungsworkflow

Neben der Struktur der verschiedenen, für beispielgetriebene Entwicklung von DSMLs erforderlichen Meta-Modelle stellt der Modellierungsworkflow einen essentiellen Aspekt dar. Im Gegensatz zu klassischen Editoren, die *entweder* freies Modellieren (z.B. Microsoft Visio, Dia) *oder* stringentes Modellieren (z.B. GMF, MetaEdit+) unterstützen, liegt in der vorliegenden Arbeit die Integration beider Modi im Zentrum der Betrachtung. Ein substantieller Punkt dabei ist, dass der Arbeitsfluss weitgehend dem von freien bzw. stringenten Modellierungswerkzeugen gleichkommen soll. Diese Vermischung stellt die

eigentliche Herausforderung und Neuerung gegenüber existierenden Ansätzen dar. Sie ist demzufolge der Hauptbestandteil des eigenen, softwaretechnischen Beitrags im Kontext der grafisch ontologischen Modellierung.

Das erfordert zunächst, dass die Benutzeroberfläche an diese Werkzeuge angelehnt ist. Erfreulicherweise ähnelt sich der Aufbau von freien und stringenten Werkzeugen in großen Teilen. Im Mittelpunkt einer solchen Anwendung steht immer das Diagramm (oft auch Zeichenfläche genannt), innerhalb dessen mittels Drag & Drop grafische Konstrukte platziert werden können. Diese Konstrukte stammen aus einer sogenannten Palette, die typischerweise links- oder rechtsseitig vom Diagramm angeordnet ist. Primär erfolgt die **Eingabe** also **mithilfe einer Maus** oder – wie neuerdings zunehmend gebräuchlich – via Touch-Gesten.

Im Hinblick auf die grafischen Konstrukte können drei Phasen des Modellierungsworkflows unterschieden werden. Für jede Phase empfiehlt sich die Bereitstellung eines separaten Bereichs innerhalb der Palette, der die jeweiligen Konstrukte bereithält. Die beiden ersten Phasen gehören der **freien Modellierung** an, was bedeutet, dass die betreffenden Konstrukte noch nicht über ein Mapping auf die Domäne verfügen.

Bei den Konstrukten der **ersten Phase** handelt es sich streng genommen (Abschnitt 5.1.3) nicht um Konstrukte, sondern um vorkonfigurierte Templates für neue grafische Konstrukte. Typischerweise gibt es Templates für häufig verwendete Standardformen, wie Rechteck, Raute oder Ellipse. Durch Drag & Drop können Instanzen dieser Templates im Diagramm platziert werden, was zur Erzeugung eines entsprechenden grafischen Konstrukts sowie einer zugehörigen Verwendung führt.

Konstrukte, die mindestens einmal verwendet wurden und damit Teil der konkreten Syntax sind, gehören zur **zweiten Phase** des grafischen Modellierungsworkflows. Im Hinblick auf eine spätere stringente Modellierung sind für derartige Konstrukte einige Eigenheiten zu berücksichtigen. Diese werden in den folgenden beiden Unterabschnitten näher beleuchtet. In Abschnitt 6.2.1 geht es darum, Änderungen bezüglich ihrer Qualität und Wechselwirkung mit der Domäne zu unterscheiden. Illustriert wird dies anhand eines Beispiels. Abschnitt 6.2.2 dagegen befasst sich mit konkreten Interaktionsmöglichkeiten, die direkt grafische Konstrukte betreffen und dadurch Einfluss auf die konkrete Syntax ausüben.

Im **stringenten Modellierungsmodus (dritte Phase)** umfasst der betreffende Palettenbereich ausschließlich diejenigen Konstrukte, die bereits gemappt und somit an ein Konzept der abstrakten Syntax gebunden sind. Außerdem dürfen die Konstrukte im Diagramm nur insoweit verwendet werden, wie von der abstrakten Syntax erlaubt. Weitere Details dazu sind Abschnitt 5.1.4 zu entnehmen.

6.2.1 Differenzierung von Änderungsqualitäten

Während der freien Modellierung muss sich der Benutzer darüber im Klaren sein, ob seine Modellierungstätigkeiten semantischer oder informeller und damit rein-visueller Natur sind. Da der Großteil der Modellierungsaktivitäten einen semantischen Hintergrund hat, werden **Änderungen** innerhalb der Diagramme **standardmäßig als semantisch** betrachtet. Sollen rein visuelle Modifikationen vorgenommen werden, so muss der Benutzer das explizit kundtun.

Bei der Verwendung einer Maus liegt (bei Rechtshändern) die linke Hand gewöhnlich auf der Tastatur. Dieser Umstand wird von vielen Programmen und Computerspielen aktiv ausgenutzt, indem **Tastenkürzel** angeboten werden, um häufig benötigte Aktionen schnell und bequem durchzuführen [26, 104]. Er lässt sich analog für die Angabe einsetzen, dass von nun an durchgeführte **Aktionen rein visueller Natur** sind. Beispielsweise wird genau dann rein visuell modelliert, solange eine bestimmte Taste gedrückt ist (z.B. die Alt-Gr-Taste). Erfolgt die Eingabe mittels Touch-Gesten kann zu dem Zweck ein zusätzlicher Button auf dem Bildschirm eingeblendet werden, der vor der betreffenden Aktion

betätigt werden muss, um sie als rein visuell zu kennzeichnen. Auf diese Weise wird der Benutzer nur unwesentlich in seinem Workflow behindert und kann seine volle Konzentration auf die Modellierung richten.

Des Weiteren ist zu unterscheiden, ob **rein visuelle Änderungen** lediglich an der aktuell ausgewählten Verwendung oder an der **zugrundeliegenden Definition** vorgenommen werden sollen. In letzterem Fall wirkt sich dies auch auf alle anderen Verwendungen dieser Definition aus, sofern der betreffende Wert in der jeweiligen Verwendung nicht bereits überschrieben wurde. Die Angabe, statt der ausgewählten Verwendung die zugehörige Definition zu bearbeiten, kann auf dem gleichen Weg erfolgen, wie bei der Unterscheidung zwischen semantisch und rein visuell; und zwar durch Gedrückthalten einer bestimmten anderen Taste (z.B. der Alt-Taste) oder bei Touch-Bedienung durch die Vorabbetätigung eines weiteren auf das Display projizierten Buttons.

Ein **beispielhaftes Szenario mit allen drei Änderungsarten** zeigt Abbildung 6-6. Es handelt sich dabei um eine schematische Grafik. Konkrete Screenshots des grafischen Editors finden sich in Abschnitt 8.2. Den Rahmen des Beispiels bildet ein Diagramm mit einer rechtsseitigen Palette, die in die Bereiche „Free“ und „Syntax“ unterteilt ist. Einen dritten Bereich für bereits gemappte Konstrukte gibt es nicht, da (noch) keine derartigen Konstrukte existieren. Der Bereich „Free“ beinhaltet diverse Standardformen, die als Ausgangspunkt für eigene Konstrukte dienen. Um die Übersichtlichkeit nicht zu beeinträchtigen, werden lediglich ein Rechteck mit abgerundeten Ecken und ein Männchen zur Verfügung gestellt. Beide sind grün eingefärbt, so dass sie sich besser von den übrigen visualisierten Artefakten abheben. Es handelt sich hierbei um schnell erreichbare Vorlagen für Konstrukte. Derartige Vorlagen müssen nicht zwangsweise angeboten werden, doch Steigern sie die Produktivität in der initialen Phase beim Entwurf einer DSML erheblich, weil sie direkt eingesetzt werden können. Die daraus resultierenden Konstrukte brauchen im Optimalfall nicht oder nur geringfügig angepasst werden. Diese Anpassbarkeit der (auf Vorlagen basierenden) Konstrukte wird jedoch bis ins Detail gewährleistet.

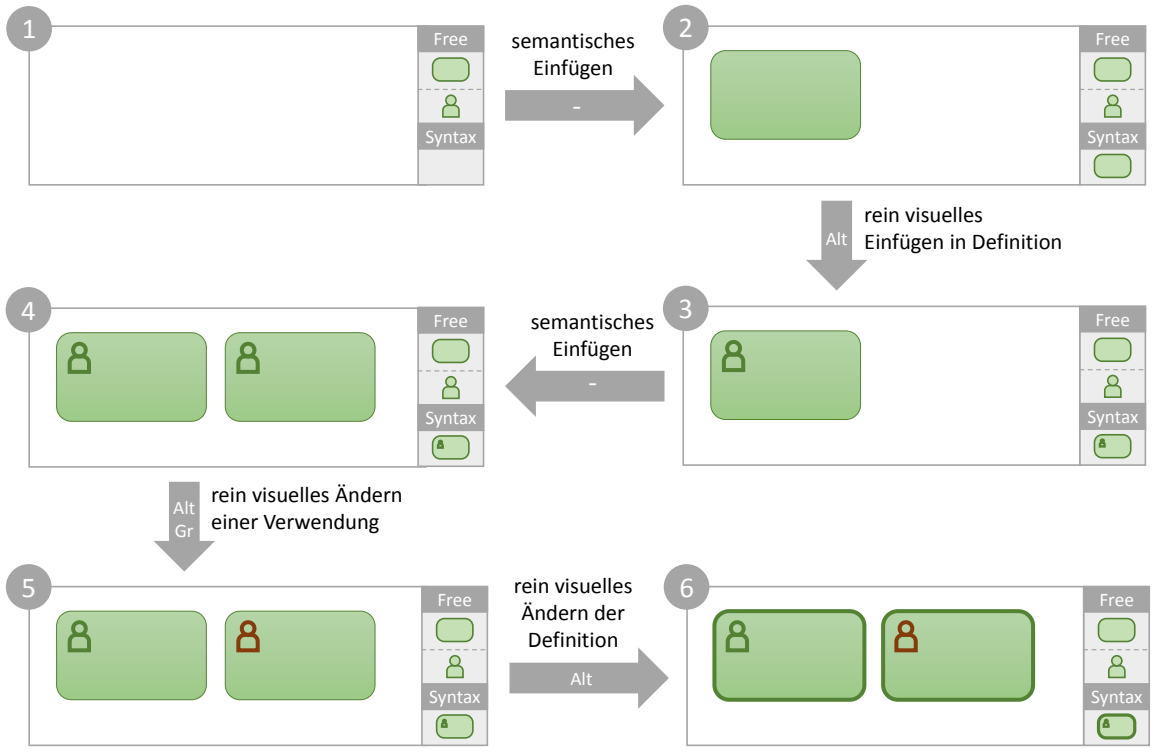

Abbildung 6-6: Grafische Änderungsarten anhand eines Beispiels

Zu Beginn (1) sind Diagramm und Syntaxbereich noch leer, da bislang keine grafischen Konstrukte definiert wurden. Dies ändert sich durch Hinzufügen eines grünen Rechtecks, wie in Schritt (2) geschehen. Nachdem der Benutzer nicht explizit spezifiziert hat, dass es sich um rein visuelles Einfügen handelt, wird die hinzugefügte Form als neues, eigenständiges Konstrukt aufgefasst und deshalb auch im Syntaxbereich der Palette hinterlegt. Aus Platzgründen wurde auf die Anzeige der Namen sämtlicher Konstrukte verzichtet. Die Vergabe eines – im Sinne der Domäne – aussagekräftigen Namens ist eine Aufgabe, die der Modellierer im Anschluss durchführen sollte. Nur dann können beim späteren Ableiten der abstrakten Syntax die jeweiligen Meta-Konzepte sinnvoll benannt werden. Nachfolgend wird davon ausgegangen, dass es sich bei dem neu erstellten Konstrukt um eine Aufgabe handelt.

Im nächsten Schritt (3) wird durch Drag & Drop die im Diagramm visualisierte Aufgabe mit einem kleinen grünen Männchen versehen. Hierbei wird angegeben (z.B. durch Gedrückthalten der Alt-Taste, wie oben vorgeschlagen), dass es sich um eine rein visuelle Änderung handelt, die obendrein die Definition und damit das visuelle Konstrukt betrifft. Dadurch wird gleichzeitig auch die Vorschau in der Palette aktualisiert.

Anschließend (4) wird das bereits definierte Konstrukt wiederverwendet und somit eine zweite Verwendung im Diagramm platziert. Dies entspricht vom Prinzip her der traditionellen, stringenten Modellierung.

Weil die neu hinzugefügte Aufgabe aus irgendeinem Grund einen besonderen Stellenwert einnimmt, soll lediglich das Männchen dieser Aufgabe rot eingefärbt werden (5). Es handelt sich somit um eine rein visuelle Änderung, die ausschließlich diese eine Verwendung betrifft. Demnach muss dies auf eine andere Weise angegeben werden (z.B. wie oben vorgeschlagen durch Gedrückthalten der Taste Alt-Gr), während die betreffende Farbeigenschaft manipuliert wird. Die eigentliche Manipulation von Eigenschaften findet bei grafischen Editoren üblicherweise in einer separaten Ansicht statt, welche alle Eigenschaften des momentan selektierten Elements anzeigt.

Gemäß Abschnitt 5.4.1 darf dieselbe Farbeigenschaft später immer noch semantisch modifiziert werden. Das kann über die gleiche Verwendung, aber auch über eine beliebig andere geschehen. Eine derartige Änderung bewirkt, dass die ehemals rein visuelle Farbanpassung nun doch semantischen Charakter hat. Denn jede Eigenschaft ist entweder rein visuell oder semantisch, nie beides zugleich. Der umgekehrte Fall ist laut dem zuvor referenzierten Abschnitt ebenso zulässig. Sind bereits in den Verwendungen der betreffenden semantischen Eigenschaft Werte zugewiesen, dann ist es bei einer solchen Modifikation ratsam, eine Rückbestätigung vom Benutzer einzuholen, um versehentliche Änderungen auf Domänenseite zu vermeiden.

Aus rein kosmetischen Gründen wird im letzten abgebildeten Schritt (6) erneut das Konstrukt „Aufgabe" geändert, indem die Dicke der Rechteckumrandung erhöht wird. Dazu kann eine der beiden Aufgaben entsprechend angepasst werden. Durch die Angabe, dass es sich um die rein visuelle Modifikation der Definition handelt (z.B. durch Betätigen der Alt-Taste), gilt diese Änderung für sämtliche Verwendungen des Konstrukts „Aufgabe". Würde man hingegen die Randfarbe des grünen Männchens innerhalb der Definition ändern, dann wäre davon lediglich die erste Aufgabe betroffen, weil die zweite Aufgabe diesen Farbwert bereits überschrieben hat.

Zur späteren Nachvollziehbarkeit oder auch bei der weiteren Bearbeitung des Diagramms durch Dritte ist es darüber hinaus hilfreich, wenn **leicht erkennbar** ist, **ob ein Artefakt semantischen oder rein visuellen Charakter** hat. Im letztgenannten Fall muss zusätzlich unterschieden werden, ob das Artefakt innerhalb der Definition oder in der aktuell betrachteten Verwendung festgelegt wurde. Anzeigen lässt sich dies z.B. durch das bedarfsweise Hervorheben der betreffenden Artefakte mit drei verschiedenen Farben, wobei lediglich Referenzen und Eigenschaften betroffen sind. Bei Referenzen können die

Anknüpfpunkte der beteiligten Knoten eingefärbt werden, wohingegen bei Eigenschaften die spezifischen Editoren (meist in einer separaten Ansicht untergebracht) mit der zutreffenden Farbe hinterlegt werden können.

Einen wichtigen Punkt stellt bei der grafischen Modellierung auch das **Selektieren der** diversen, **im Diagramm enthaltenen Elemente** dar (beispielsweise um Elemente zu verschieben oder deren Styling anzupassen). Ohne Zutun des Benutzers sind alle Änderungen semantischer Natur, weshalb im Standardfall (d.h. wenn weder Alt noch Alt-Gr gedrückt sind) bei der Auswahl eines grafischen Objekts stets der Wurzelknoten des betreffenden verwendeten Konstrukts selektiert wird. Es wäre also nicht möglich das Männchen einer Aufgabe auszuwählen, sofern zuvor nicht explizit rein visuelles Bearbeiten aktiviert wird. Nachdem auch weiterhin der Großteil aller Modellierungstätigkeiten im stringenten Modus erfolgen wird, ist diese Restriktion allerdings eher ein Vorteil. Wird somit beispielsweise auf das Männchen einer Aufgabe geklickt, dann wird das zugehörige Eltern-Rechteck selektiert.

6.2.2 Interaktionsmöglichkeiten mit grafischen Konstrukten

Um ein Wiederfinden und damit Wiederverwenden bereits definierter **Konstrukte** zu vereinfachen, ist es zweckmäßig diese **mit geeigneten Namen** zu **versehen**. Die Namen von Konstrukten werden üblicherweise auch neben dem grafischen Symbol in der Palette angezeigt (im obigen Beispiel wurde aus Platzgründen darauf verzichtet). Deshalb eignet sich der Weg über die Palette besonders, um vorhandene Konstrukte zu benennen.

Das explizite **Löschen existierender Konstrukte** ist gemäß Abschnitt 5.4.1 zu jedem Zeitpunkt erlaubt. Wegen potentiell weitreichender Folgen sollten nur solche Konstrukte gelöscht werden, die nirgends verwendet werden. Verwendungen derartiger Konstrukte tauchen somit in keinem Diagramm auf und können folglich nicht innerhalb eines grafischen Editors gefunden und entfernt werden. Das explizite Löschen wird demnach ausschließlich über die Palette ermöglicht. Existieren dennoch irgendwelche Verwendungen des zu entfernenden Konstrukts, dann ist der Benutzer darauf hinzuweisen, so dass er sich der Gefahr weitreichender Einschnitte bewusst ist.

Das **Umbenennen und Löschen von Features** gestaltet sich etwas schwieriger, weil sie nicht als eigenständige Elemente in der Visualisierung zu Tage treten. Stattdessen sind Features Merkmale von Knoten. Folglich kann in direkter Weise nur mit dem betreffenden Knoten interagiert werden. Wie bereits erwähnt, sind Diagramm-Editoren typischerweise gepaart mit einer Ansicht zur Bearbeitung von Eigenschaften des momentan selektierten grafischen Elements. Darin können auch die einzelnen Features bzw. deren Definitionen explizit ausgewählt und mit ergänzenden Informationen versehen werden (im aktuellen Kontext vor allem mit den Annotationen des in Abschnitt 5.2.1 vorgestellten, allgemeinen visuellen Meta-Modells).

Gesondert betrachtet werden muss die **Sichtbarkeitseigenschaft**. Denn ist ein Knoten einmal ausgeblendet, dann kann er im Diagramm vorerst nicht selektiert werden. Eine mögliche Lösung dafür ist, bei der Auswahl eines sichtbaren Knotens dessen unsichtbare Kinder schattenhaft darzustellen, so dass sie anschließend dennoch auswählbar und dadurch modifizierbar sind. Die geschilderte Problematik besteht nur solange, wie sich hinter der Sichtbarkeit keine Domäneneigenschaft verbirgt. Ist das nämlich der Fall, dann kann das Ein- und Ausblenden mithilfe dieser Eigenschaft gesteuert werden (für gewöhnlich in der besagten separaten Ansicht zur Bearbeitung von Eigenschaften).

6.3 Erweiterung des Ableitungsprozesses für grafische DSMLs

Im Meta-Modell aus Abschnitt 6.1.1 **neu hinzugekommen** sind die Möglichkeiten **Verbindungen zu modellieren sowie Knoten mit Styling-Informationen anzureichern**, die deren Aussehen (z.B. Form und Farbe, aber auch Dicke der Schriftart) beeinflussen. Die Mapping-Möglichkeiten, die durch das allgemeine Mapping Meta-Modell zur Verfügung stehen, reichen aus, um auch Verbindungen und Stylings auf die Domäne abzubilden. Im Zuge der Bottom-Up Modellierung müssen diese Mappings zusammen mit den zugehörigen Konstrukten der abstrakten Syntax aus solchen frei modellierten Artefakten generiert werden. Wie dabei vorgegangen wird, ist Gegenstand der kommenden Unterabschnitte. Diese Ausführungen stellen den konzeptuellen Beitrag innerhalb von Kapitel 6 dar.

6.3.1 Verarbeitung semantischer Verbindungen

Begonnen wird mit der Betrachtung von Verbindungen. Eine **Verbindung kann auf zwei unterschiedliche Weisen auf die Domäne abgebildet werden** – gemäß der beiden Beispiele aus den Abschnitten 6.1.2.1 und 6.1.2.2 für Attribut-bezogene bzw. Konzept-bezogene Mappings. Unter Berücksichtigung des in der Informatik üblichen KISS-Prinzips („Keep It Simple, Stupid") [127] wird das Attribut-bezogene Mapping bevorzugt. Es wird solange eingesetzt, bis seine Kapazitäten erschöpft sind. Das ist dann der Fall, wenn ein mit `Edge` typisierter Knoten oder eines seiner rein-visuellen Kinder über semantische Features verfügt. Auf Domänenseite entsprechen sie literalen oder referentiellen Attributen, deren Deklaration in einem Meta-Konzept erfolgen muss. Bei einem Attribut-basierten Mapping liegt allerdings kein derartiges Konzept vor. Deshalb wird bei Verbindungsknoten mit semantischen Features auf das Konzept-bezogene Mapping zurückgegriffen. Es wird also ein eigenständiges Meta-Konzept erzeugt, auf das der Verbindungsknoten abgebildet wird.

Die **Umsetzung** erfolgt **mit speziellen Inferrers** für die neu hinzugekommenen Typen `Edge`, `Incoming` und `Outgoing`. Im Zusammenspiel überprüfen sie das Kriterium, ob der mit als `Edge` typisierte Knoten (transitiv) über semantische Features verfügt. Trifft diese Bedingung zu, generiert der neue Element-Inferrer ein separates Meta-Konzept mit zugehörigen `OntElementMapping`. Außerdem werden für die Referenzen `Incoming` und `Outgoing` jeweils ein `OntReference-Mapping` inklusive erforderlicher Domänen-Attribute erzeugt. Ein konkretes Beispiel hierzu wurde bereits in Abschnitt 6.1.2.2 beschrieben. Liegt kein semantisches Feature vor, dann wird für den Verbindungsknoten ein `WildcardElement` generiert sowie ein `OntReferenceMapping` für die `Outgoing`- und eine `WildcardReference` für die `Incoming`-Instanz (siehe Beispiel in Abschnitt 6.1.2.1)

War bislang die Bedingung für ein Attribut-bezogenes Mapping erfüllt, was aber jetzt nicht mehr zutrifft, weil die Verbindung zwischenzeitlich mit einem semantischen Feature versehen wurde, dann müssen die einzelnen Inferrers diesen Umstand beheben. Am einfachsten funktioniert dies durch Wiederverwendung der vorhandenen Funktionalität. Dazu müssen sie sich zunächst invers und direkt im Anschluss regulär anwenden. Dies ist jedoch lediglich ein Implementierungsvorschlag, wie die Verarbeitung der entsprechenden Kontextänderungen umgesetzt werden kann. Die umgekehrte Richtung wird nicht automatisiert unterstützt, weil dies mit der Löschung von Konzepten einherginge. Stattdessen wird dem Benutzer ein entsprechender Hinweis unterbreitet, woraufhin er eine diesbezügliche Modellevolution manuell durchführen kann (im Optimalfall geleitet durch einen geeigneten Operator [66]).

6.3.2 Verarbeitung semantischer Styling-Eigenschaften

Semantische Styling-Eigenschaften können beim Ableiten prinzipiell auf dieselbe Weise verarbeitet werden wie semantische Labels (Abschnitt 5.3.2.2, Unterpunkt Label-Inferrer). Ohne Angabe weiterer Informationen resultiert dann ein **Attribut in Kombination mit einem Direct-Mapping**. Der Typ des Attributs ist durch den Typ des `value`-Felds der ursprünglichen Styling-Eigenschaft festgelegt (z.B. String bei `ForeColor` und Integer bei `LineWidth`). Analog zu den Labels wird die Multiplizität entweder auf `1` oder `0..1` gesetzt. Als Standardname empfiehlt sich ein generischer Begriff wie z.B. „value".

Im Regelfall werden semantische Styling-Eigenschaften nicht direkt, sondern unter Verwendung von Wertebeziehungen auf die Domäne abgebildet. Das trifft insbesondere dann zu, wenn der Datentyp der Eigenschaft nicht Boolean ist. Denn bei booleschen Werten genügt wegen des stark limitierten Wertebereichs oftmals ein direktes Mapping. Ein Beispiel dazu findet sich in Abschnitt 5.3.2.2 unter dem Punkt „Eigenschaften mit Wertebeziehungen". Dort sind die drei Farben Rot („#ff0000"), Gelb („#ffff00") und Weiß („#ffffff") in Beziehung mit den Domänenwerten `ERROR`, `WARN` und `HINT` gesetzt. Würden stattdessen die Farbwerte direkt abgebildet werden, so könnte höchstens ein Informatiker, der die Hexadezimaldarstellung der Farben kennt, erahnen, was deren Bedeutung ist. Um diesem Defizit entgegenzuwirken und in Bezugnahme auf das Qualitätskriterium „Selbstbeschreibung" (Abschnitt 2.1.3.2) wird für jede semantische Styling-Eigenschaft ohne Wertebeziehungen eine **Empfehlung** generiert, die dem Benutzer vorschlägt, passende **Wertebeziehungen bereitzustellen**.

6.4 Kompensation von Modellierungsungenauigkeiten

Modellierungsungenauigkeiten sind eine grundsätzliche **Problematik**, die beim grafischen Modellieren auftreten kann. Sie ist im Wesentlichen der **Point & Click Eingabeform** geschuldet, die zwar pixelgenaue Koordinaten zurückliefert; der Benutzer arbeitet jedoch eher flächenbezogen, d.h. er interagiert mit flächigen Objekten, die eine Vielzahl von Pixeln umfassen. Gleichgültig auf welchen Pixel einer solchen Fläche er beispielsweise klickt, es wird immer dieselbe eindeutig festgelegte Funktionalität ausgeführt. Bei der Touch-Eingabe kommt dieser Flächeneffekt noch stärker zum Tragen, weil die menschliche Fingerkuppe im Gegensatz zur Pfeilspitze eines Mauszeigers eine etwas größere Region des Bildschirms überdeckt [42].

Treten Modellierungsungenauigkeiten auf und werden als solche erkannt, so gibt es prinzipiell zwei Alternativen diesen zu begegnen. Entweder fällt der Rechner automatisiert eine Entscheidung (z.B. durch Anwenden einer Standardstrategie) oder er holt Feedback vom Benutzer ein. Diese **zwei Lösungswege** entsprechen den beiden erstgenannten Strategien zur Kompensation von lückenhaftem Domänenwissen aus Abschnitt 4.6. Die geschilderte Problematik kann sogar als Fortführung dieses Lückenausgleichs angesehen werden, da auch hier fehlende Informationen durch ergänzende Mechanismen zur Verfügung gestellt werden müssen.

Ziel dieses Abschnitts ist es, auf die bestehende Gefahr hinzuweisen und eine grundsätzliche Methode zu beleuchten, wie ihr begegnet werden kann, ohne dass der Modellierer von seiner eigentlichen Arbeit abgelenkt wird. Die erste Variante (Anwenden einer Standardstrategie) ist zu bevorzugen, weil dadurch keine zusätzlichen manuellen Eingaben erforderlich sind. Damit der Benutzer sich über die Auswirkungen seiner Modellierungsaktivitäten im Klaren ist, kommt das in grafischen Editoren und Frameworks [51, 72] übliche Konzept des **visuellen Feedbacks** zum Einsatz. Hierbei werden die besagten Auswirkungen in Form einer reaktiven Vorschau angezeigt. Will man beispielsweise ein Element einem anderen Element hinzufügen, so wird das Zielelement hervorgehoben, wenn man die Maus darüber bewegt. Beim Anlegen von Verbindungen dagegen wird typischerweise eine Vorschau

der resultierenden Verbindungslinie dargestellt. Ein ähnliches Konzept wurde bereits in Abschnitt 6.2 erläutert, bei dem durch visuelle Hervorhebung angezeigt wird, ob eine Aktion semantischen oder rein visuellen Ursprungs ist.

Allerdings sind auch Fälle vorstellbar, in denen es nicht möglich ist, präzises Feedback anzuzeigen. Dann ist es angebracht, zusätzliche Informationen vom Benutzer abzufragen. Einher geht dies mit dem Anbieten weiterer und gegebenenfalls bequemerer Bearbeitungsmöglichkeiten. Beispielhaft genannt werden kann hier das Modellierungswerkzeug Visual Paradigm for UML [138]. Vergrößert man dort ein Paket, so dass es andere Elemente überdeckt, dann wird nachgefragt, ob diese Elemente dem Paket hinzugefügt werden sollen. Generell sollte also bei der Einführung einer **neuen Bearbeitungsmöglichkeit** überprüft werden, ob das **Risiko von Mehrdeutigkeiten** besteht. Falls ja, sollte versucht werden sie mithilfe von visuellem Feedback zu umgehen. Ist dies nicht möglich, so ist es sinnvoll direkt den Benutzer in die Entscheidungsfindung einzubeziehen.

6.5 Zusammenfassung

Das Hauptaugenmerk in diesem Kapitel liegt auf dem grafischen Modellierungsworkflow. Mit seiner integrierten Unterstützung von freiem und stringentem Modellieren repräsentiert er einen Aspekt des eigenen softwaretechnischen Beitrags. Der Workflow erfolgt – wie bei grafischen Sprachen üblich – vorrangig unter Verwendung der Maus (bzw. Touch-Gesten), wodurch der Benutzer seine gewohnte Arbeitsweise aufnehmen kann. Gleiches gilt auch für den Aufbau der Benutzeroberfläche, die sich aus Diagramm-Editor, verschiedenen Palettenbereichen und einer Ansicht zur Bearbeitung der Eigenschaften des momentan selektierten Elements zusammensetzt (weitere Details dazu finden sich auch in Abschnitt 8.2). Des Weiteren werden Ergänzungen des allgemeinen, visuellen Meta-Modells eingeführt, die spezifisch für grafische DSMLs sind. Letztlich liefert der in diesem Kapitel vorgestellte Ansatz eine **Lösung für Anforderung 1a**, bei der es um die konstruktivistische Entwicklung grafischer Modellierungssprachen geht.

7 Beispielgetriebene Entwicklung im textuell ontologischen Kontext

Die beispielgetriebene Entwicklung von textuellen DSMLs (Anforderung 1b) wurde nach unserem Kenntnisstand noch nirgends untersucht. Sämtliche verwandte Arbeiten zu dieser Thematik (Abschnitt 3.3) beschäftigen sich mit der Rekonstruktion von Grammatiken aus einer möglichst großen Menge positiver und negativer Beispielsätze. In den kommenden Ausführungen wird jedoch eine Methode präsentiert, die als analog zur Bottom-Up Vorgehensweise für die Entwicklung und Verwendung grafischer DSMLs (Kapitel 6) anzusehen ist. Primär geht es hierbei um die **Unterstützung der freien Modellierung textueller DSMLs** und demnach darum, wie ein möglichst intuitiver, textueller Modellierungsworkflow aussieht und softwaretechnisch umgesetzt werden kann. Aufgrund der Neuartigkeit dieses Workflows werden die einzelnen Schritte (Abschnitte 7.3 bis 7.6) sowie die dabei verarbeiteten Datenstrukturen (Abschnitt 7.2) detailliert erläutert. Erste Ideen zu diesem Ansatz wurden von unserer Forschungsgruppe bereits in [121] veröffentlicht.

Rein textuelle, auf Zeichenketten basierende Editoren scheiden für die Umsetzung aus, weil sie die strukturellen Eigenschaften des eingegebenen Textes nicht adhoc erkennen und vorhalten können. Im Wesentlichen käme es der Verarbeitung von natürlicher Sprache [3] gleich, was jedoch ein Teilgebiet der Linguistik ist und außerhalb des Untersuchungsspektrums dieser Dissertation liegt. Stattdessen wird der **projizierende Ansatz** [33] verfolgt, bei dem die textuelle Repräsentation eine Projektion eines darunter liegenden Modells ist. Konträr zu MPS von JetBrains [94] und Domain Workbench von Intentional Software [128] wird im vorgestellten Ansatz nicht das Domänenmodell, sondern das visuelle Verwendungsmodell projiziert. Erforderlich macht das die freie Modellierung, bei der weite Teile der abstrakten Syntax und damit des Domänenmodells fehlen dürfen.

Der **Aufbau eines Dokuments** im Textuellen ist Dank des projizierenden Ansatzes identisch mit dem der grafischen Welt, was auch der Auffassung der Autoren von [60] entspricht. Jedes Dokument bildet demnach eine durch Enthaltensein-Beziehungen festgelegte Baumstruktur von Knoten. Gruppen von Knoten innerhalb eines Dokuments repräsentieren Verwendungen von Konstrukten der konkreten Syntax. Diese Knoten spannen somit einen CST auf [41] (zur Wiederholung: CST ist die Abkürzung für konkreter Syntaxbaum). Nachdem im Textuellen andersartige Beziehungen – visuell betrachtet – eine eher untergeordnete Rolle spielen, ist im Folgenden stets von Baumstruktur bzw. CST die Rede.

Zum besseren Verständnis kann ein **Konstrukt der konkreten Syntax** (siehe Definition in Abschnitt 5.1.3) beim textuell projizierenden Ansatz als **Schablone** aufgefasst werden, die an bestimmten Stellen mit Werten zu befüllen ist. Bei diesen Werten handelt es sich entweder um Text oder um Verwendungen anderer Konstrukte. Im Grunde entspricht dies der etablierten Vorgehensweise bei der grafischen Modellierung. Hier können die einzelnen grafischen Konstrukte ebenfalls als Schablonen betrachtet werden. Platzhalter dieser Schablonen sind z.B. Anknüpfpunkte für Verbindungen, Bereiche, die andere Elemente aufnehmen können, aber auch Textfelder mit prinzipiell beliebigem Freitext. Aufgrund einer höheren Prägnanz wird im Folgenden „textuell-projizierend" durch „textuell" abgekürzt.

Erklärtes **Ziel** der beispielgetriebenen Entwicklung von DSMLs ist, dem Benutzer eine Möglichkeit zu bieten, **auf intuitive Weise eigene DSMLs zu definieren**. Gemäß der Definition des Begriffs „Intuitionismus" in [100] bedeutet intuitiv in dem Zusammenhang, dass der Benutzer in die Lage versetzt werden soll, mit bisher gewohnten und vertrauten Mitteln eine Sprache zu entwerfen, die ebenfalls in ihrer Struktur dem Benutzer gewohnt und vertraut erscheint.

Zielgruppe der beispielgetriebenen Entwicklung im textuellen Kontext sind **Softwareentwickler**, da sie im Umgang mit Programmier- und gegebenenfalls auch textuellen Domänensprachen sowie den damit einhergehenden Werkzeugen geübt sind. Nimmt man beides zusammen (also Zielgruppe und Intuitionismus), kommt man zu dem Ergebnis, dass ein diesbezüglicher Editor ein sehr ähnliches Bedienverhalten zeigen muss, wie die zuvor erwähnten Werkzeuge MPS und Domain Workbench. Andernfalls ist der Benutzer gezwungen einen hohen Einarbeitungsaufwand zu betreiben, um die Funktionsweise des Editors zu erlernen. Dies stünde jedoch konträr zur intuitiven Bedienbarkeit, weil es nicht der gewohnten Arbeitsweise des Benutzers entspräche.

Vorrangig erfolgt die **Interaktion mit textuellen Dokumenten unter Verwendung einer Tastatur**, wobei das Dokument im Regelfall zeichenweise modifiziert wird. Aufgrund des projizierenden Ansatzes und der damit verknüpften direkten Manipulation des CST, ist ein Arbeitsfluss nicht umsetzbar, der gänzlich dem Schreiben von Fließtext entspricht. Akzeptabel ist dies deshalb, weil projizierende Editoren Dank ihrer zusätzlichen Visualisierungsmöglichkeiten vermehrt Einzug in die tägliche Arbeit von Modellierern und Programmierern nehmen (als prominentes Beispiel gilt das mbeddr-Projekt [140]). Dennoch wird angestrebt, sich möglichst nahe an der Freitexteingabe zu orientieren.

Des Weiteren ist die Bereitstellung **typischer Hilfsmittel moderner IDEs** (allen voran Code-Vervollständigung und Syntaxhervorhebung) von zentraler Bedeutung. Derartige Hilfsmittel sind laut Grönniger et al. [57] unverzichtbar, weil sie den Einsatz von DSMLs erleichtern und damit zur Steigerung der Produktivität beitragen. Neben den Sprachbenutzern profitieren auch die Entwickler einer DSML davon.

Der Aufbau dieses Kapitels gliedert sich folgendermaßen: Anfangs wird auf ergänzende Designprinzipien eingegangen, die spezifisch für die textuelle Welt sind (Abschnitt 7.1). Danach werden Erweiterungen des visuellen Meta-Modells vorgestellt, die zur Spezifikation textueller Konstrukte erforderlich sind (Abschnitt 7.2). Im Anschluss daran folgt die Beschreibung des Modellierungsworkflows, der die größte zu lösende Herausforderung im textuell ontologischen Kontext darstellt. Wegen seiner Komplexität ist der Workflow in die Abschnitte 7.3 bis 7.6 aufgeteilt. Erst anschließend geht es um die für textuelle DSMLs spezifische Erweiterung des Ableitungsprozesses (Abschnitt 7.7). Eine ebenfalls wichtige Stellung nimmt die Erweiterbarkeit des vorgestellten Ansatzes sowie zusätzliche Möglichkeiten zur nachträglichen Verfeinerung der generierten abstrakten Syntax ein. Beide Punkte werden abschließend in den Abschnitten 7.8 und 7.9 behandelt.

7.1 Ergänzende Designprinzipien

Zur Erfüllung der Arbeitsflussbedingung ist es unerlässlich, dass der Benutzer annähernd alle Aufgaben, die während der textuellen Modellierung anfallen, mit der Tastatur erledigen kann. Dabei muss es möglich sein, **textuelle Beispielmodelle** weitgehend (in der gewohnten Weise) **Zeichen für Zeichen eingeben** zu können. Die eingetippten Zeichen werden simultan analysiert und die zugrundeliegende Dokumentstruktur gegebenenfalls gemäß der Bedeutung des Zeichens umgebaut. Dazu ist es zunächst notwendig, diese Bedeutung zu identifizieren.

Hierbei beziehen wir uns erneut auf den Intuitionismus. Denn **bestimmte Symbole** haben im Alltag sowie in vielen verschiedenen Sprachen eine **ähnliche**, wenn nicht sogar die gleiche **Bedeutung**. Bei

der Eingabe eines entsprechenden Symbols wird diese Bedeutung somit vom Benutzer erwartet. Ausgenommen hiervon sind esoterische Sprachen, wie Brainfuck [16] oder Whitespace [146], die nicht für den Einsatz bei der produktiven Softwareentwicklung vorgesehen sind. Als zweitrangig betrachtet werden außerdem kryptische Abkürzungen (Zeiger in C, Regel- und Faktenoperatoren in Prolog, Syntax regulärer Ausdrücke im Allgemeinen etc.), die in unterschiedlicher Form in zahlreichen Programmiersprachen Einzug gehalten haben, um an einigen Stellen die Tipparbeit zu reduzieren. Primäres Ziel ist die Unterstützung von Sprachen, bei denen die Lesbarkeit im Vordergrund steht. Darauf basierende Dokumente können viel leichter als Kommunikationsmittel zwischen Modellierungsspezialist und (weniger IT-affinen) Domänenexperten eingesetzt werden.

Beispielhaft sollen nur einige Symbole genannt werden, in die man erfahrungsgemäß eine bestimmte Bedeutung hineininterpretiert. Typischerweise werden einzelne Wörter durch Leerzeichen voneinander getrennt. Die Aufzählung semantischer Elemente erfolgt häufig durch deren Aneinanderreihung, wobei als Trennsymbol das Komma zum Einsatz kommt. Zusammengehörende Elemente werden außerdem oft mithilfe von Klammern gruppiert. Anstelle von Klammern wird zur Gruppierung von Elementen in einigen Sprachen auch auf Einrückungen gesetzt. Das verringert syntaktisches Rauschen durch Weglassen der Klammern.

Für die weiteren Ausführungen ist von Relevanz, dass der **Begriff „Symbol"** nicht zwangsweise ein einzelnes Zeichen bedeuten muss. Ein Symbol darf sich auch aus mehreren Zeichen zusammensetzen, wie z.B. in Listing 7-1 bei den als „=>" dargestellten Folgepfeilen.

Der in diesem Listing dargelegte Zustandsautomat beinhaltet Vertreter für die zuvor erwähnten Strukturierungsmöglichkeiten formaler Texte (und zwar Leerzeichen, Klammern, Kommas, Zeilenumbrüche und Einrückungen). Da Softwareentwickler im Allgemeinen sowohl mit Zustandsautomaten als auch mit Programmiersprachen vertraut sind, verstehen sie das abgebildete Beispiel ohne ergänzende Informationen. Das stets erforderliche Domänenwissen ist somit vorhanden.

```
1  events (START, WAIT, STOP)
2
3  state Idle
4      START => Running
5      WAIT => Idle
6
7  state Running
8      STOP => Idle
```

Listing 7-1: Textuelles Beispiel eines Zustandsautomaten mit drei Ereignissen und zwei Zuständen

Ein Rechner kann auf diesen Erfahrungsschatz jedoch nicht zurückgreifen, weshalb er die Struktur des Dokuments nicht ohne zusätzliche Informationen schlussfolgern kann. Im textuell ontologischen Kontext ist es also zweckmäßig (sofern möglich), die Bedeutung bestimmter **eingegebener Symbole auszuwerten und auf die Baumstruktur des zugrundeliegenden Dokuments anzuwenden**. Naturgemäß handelt es sich hierbei um Interpunktionssymbole. Im Umfeld des Compilerbaus werden sie üblicherweise Begrenzer genannt, da sie Wörter oder auch ganze Wortsequenzen voneinander separieren. Wie aus dem obigen Beispiel deutlich wird, können Begrenzer nach ihrer Auswirkung auf die Dokumentstruktur in Klassen eingeteilt werden. Welche Klassen zu berücksichtigen sind und wie diese sich in das textuell-visuelle Meta-Modell integrieren, wird im nachfolgenden Abschnitt erörtert. Gleiches gilt auch für die Typisierung der im Dokument vorkommenden Wörter, die durch entsprechende Syntaxhervorhebung visualisiert ist.

7.2 Erweiterung des visuellen Meta-Modells für textuelle Syntaxen

Viele **GPLs und DSMLs** folgen in ihrem Aufbau und ihrer Struktur **immer wiederkehrenden Mustern**. Dies belegen sowohl die Spezifikationen zu verschiedenen Programmiersprachen (Java [53], C++ [134], Python [135] und Common Lisp [25]) als auch Aussagen von Fowler [43] und Friedmann [46]. Im Folgenden behalten wir die in diesen Spezifikationen verwendete sowie aus dem Compilerbauumfeld bekannte Terminologie bei. Die den Begrifflichkeiten zugrundeliegenden Konzepte werden jedoch dahingehend ergänzt und angepasst, so dass sie für die beispielgetriebene Entwicklung von textuell-projizierenden DSMLs eingesetzt werden können.

Allen voran sind die **Tokens** zu nennen, die wegen des enthaltenen Texts die sichtbaren Bestandteile eines Dokuments repräsentieren. Innerhalb des CST bilden sie die Blätter. Die inneren Knoten dagegen werden als **Container** bezeichnet und sind vornehmlich für die Strukturierung des Dokuments zuständig. Ein Beispiel eines kurzen Dokuments, das einen sehr einfachen Zustandsautomaten wiedergibt, inklusive dessen Baumstruktur ist in Abbildung 7-1 veranschaulicht. Dabei symbolisieren Kreise die Container und Rechtecke die Tokens des Dokuments. Diejenigen Rechtecke, die leer zu sein scheinen, repräsentieren Leerzeichen. Wie diese Struktur basierend auf dem gegebenen Text entsteht, wird vornehmlich in Abschnitt 7.4 erläutert.

7.2.1 Klassifizierung von Tokens

Die Syntaxhervorhebung des Beispiels in Abbildung 7-1 macht deutlich, dass sich Tokens in verschiedene Kategorien einteilen lassen. Die Einteilung erfolgt anhand der unterschiedlichen Bedeutung von Tokens, wobei grobgranular (wie bei natürlichen Sprachen auch [90]) zunächst zwischen Vokabular und Interpunktion differenziert werden kann. Darüber hinaus können formale Sprachen auch Tokens beinhalten, die komplett zu ignorieren sind. Damit gemeint sind üblicherweise Kommentare und Whitespace-Symbole. Nachdem letztere auch semantischen Ursprungs sein können, zählen sie in der vorliegenden Arbeit zur Interpunktion.

Im textuell-visuellen Meta-Modell aus Abbildung 7-2 (auch textuelles Meta-Modell genannt) wird die Kategorisierung mithilfe spezieller Knotentypen ermöglicht, die alle eine Erweiterung des Konzepts `Token` darstellen. Im Unterschied zum grafischen Kontext verfügen hier die meisten Knotentypen über Attribute (nämlich diejenigen, die `Token` spezialisieren). Innerhalb der **Typinstanzen** sind sie daher zu **parametrieren**, und zwar in Abhängigkeit von der jeweiligen DSML. Demzufolge sind die einsetzbaren Knotentypen **für jedes textuelle Definitionsmodell** spezifisch und werden erst darin

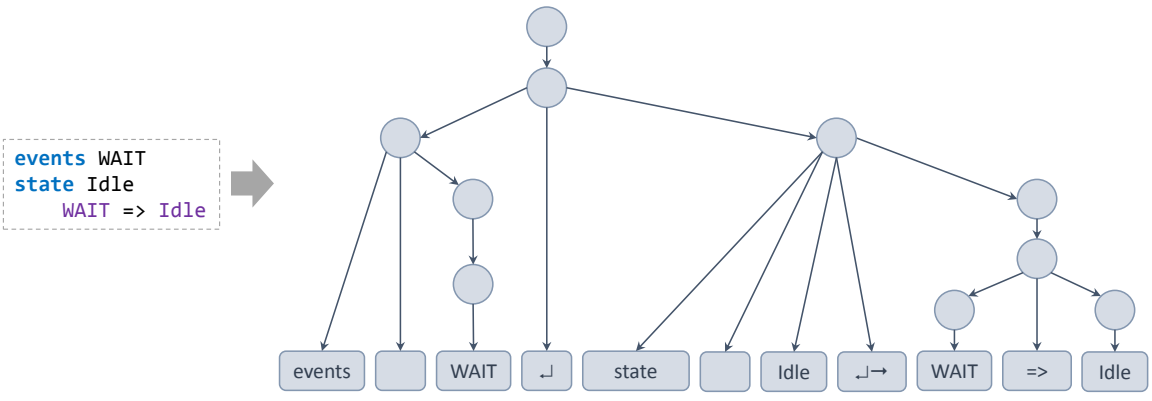

Abbildung 7-1: Baumstruktur von Dokumenten anhand eines kurzen textuellen Beispiels

festgelegt. In Abschnitt 7.4.1.2 wird eine diesbezügliche Standardkonfiguration vorgestellt, mit der jedes neu erstellte textuelle Definitionsmodell ausgestattet wird.

Zur **Unterstützung von Syntaxhervorhebung** können alle Instanzen eines Tokentyps mit einem `TextStyle` versehen werden. Dieses Konzept ist bewusst abstrakt, da die bereitzustellenden Styling-Möglichkeiten implementierungsspezifisch und nicht konzeptueller Natur sind. Welche Styling-Parameter zur Verfügung stehen, hängt somit von den Spezialisierungen des `TextStyle`-Konzepts ab. Typische Parameter sind Schriftfarbe und Schriftdicke.

Jeder spezialisierte **Tokentyp** muss eigenständig in der Lage sein zu **prüfen**, ob ein gegebener Text seinen Anforderungen entspricht. Außerdem muss er sich selbst auf einen gegebenen Knoten **anwenden** und dabei gegebenenfalls erforderliche Anpassungen an der lokalen Baumstruktur vornehmen können. Umgesetzt werden können beide Funktionalitäten beispielsweise dadurch, dass jeder Tokentyp entsprechende Methoden bereitstellt. Angelehnt an die in Abschnitt 7.2 aufgeführte Literatur werden die nachfolgend beschriebenen sechs elementaren Tokentypen unterschieden:

- *Bezeichner*: Ein Bezeichner sorgt für die eindeutige Benennung einer Konstruktverwendung innerhalb des aktuellen Namenraums. Konstruktverwendungen werden stets durch Statements repräsentiert, wohingegen jeder Block seinen eigenen Namensraum aufspannt (weitere Details dazu folgen in den Abschnitten 7.2.2 und 7.2.4.1). Die Benennung dient ausschließlich

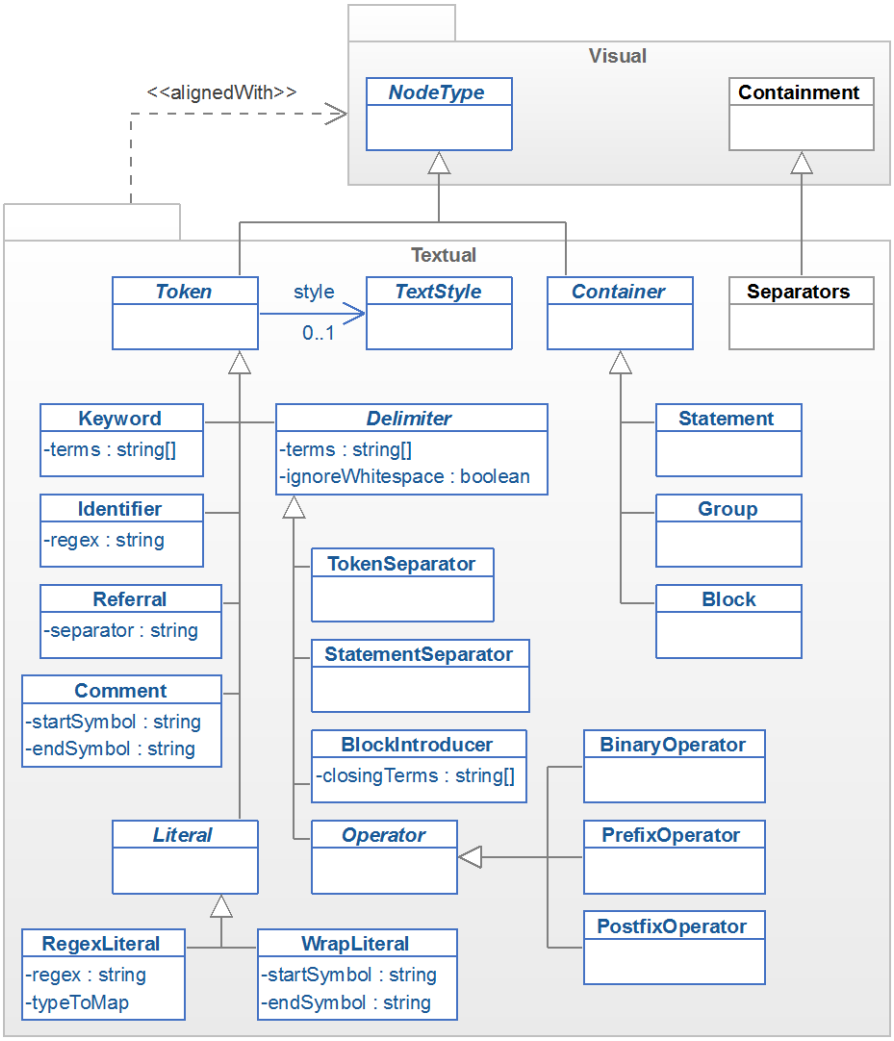

Abbildung 7-2: Erweiterung des allgemeinen visuellen Meta-Modells für konkrete textuelle Syntaxen

dem späteren Identifizieren der betreffenden Konstruktverwendung, um sie über den Namen auffinden und referenzieren zu können (siehe „Verweis"). Die eigentliche Referenz wird intern dennoch mithilfe einer Instanz des `Reference`-Konzepts abgelegt. Jeder Bezeichner ist damit rein visueller Natur und ergibt beim späteren Ableiten der abstrakten Syntax kein eigenständiges Attribut. (Konzept `Identifier`)

- *Verweis*: Ein Verweis ist das Gegenstück zum Bezeichner. Der Text eines Verweises ist immer der eindeutige Name einer Konstruktverwendung. Wie schon zuvor erwähnt, wird die eigentliche Referenz durch eine Instanz des Konzepts `Reference` ausgebildet, weshalb der Text ebenfalls nur rein visuellen Charakter hat. Nähere Details zur Auflösung von Namen folgen in Abschnitt 7.2.4.1. Er befasst sich eingehend mit der Referenzierungsthematik. (Konzept `Referral`)
- *Schlüsselwort*: Fowler zählt Schlüsselwörter zur Interpunktion [43], da es sich hierbei um von vornherein feststehende Begriffe handelt. Bei unserem Ansatz sind sie jedoch dem Vokabular zugeordnet, da sie während der Eingabe zunächst nicht von Bezeichnern zu unterscheiden sind und deshalb explizit als Schlüsselwörter deklariert werden müssen. Außerdem handelt es sich bei Schlüsselwörtern im Regelfall um domänenspezifische Termini, weshalb eine Zuordnung zum Vokabular eher gerechtfertigt ist. Pro Sprache kann eine beliebige Menge von Schlüsselwörtern definiert werden, die in Instanzen des Konzepts `Keyword` (und darin im `terms`-Attribut) abgelegt werden. Der Text von Schlüsselwörtern kann entweder rein visueller oder semantischer Natur sein. Ersteres trifft dann zu, wenn der betreffende Schlüsselwort-Knoten inklusive Text fester Bestandteil eines Konstrukts und damit im Definitionsmodell disponiert ist. Im letzteren Fall müssen Schlüsselwörter als Alternativen angegeben werden (unter Zuhilfenahme des Konzepts `ValueRelation`, Abschnitt 5.2.2.1), wodurch der konkrete Text erst in den jeweiligen Verwendungen festgelegt wird. Ein Beispiel zum besseren Verständnis folgt in Abschnitt 7.5.3.3, weil erst dort die nötigen Voraussetzungen vorgestellt sind.
- *Literal*: Ein Literal repräsentiert einen symbolgetreuen Wert, der direkt in die Domäne übernommen wird bzw. dieser entstammt. Ihr Text weist somit immer semantischen Charakter auf. Prinzipiell ließe sich jede Literalart mithilfe eines regulären Ausdrucks eindeutig identifizieren. Doch speziell für Zeichenketten bietet es sich an, stattdessen Start- und Endsymbol zu spezifizieren, auf die hin lediglich das erste bzw. letzte Symbol eines Tokens überprüft werden. Der gesamte Text zwischen diesen beiden Symbolen wird dem betreffenden Token zugerechnet, wodurch das typische Maskieren von Zeichen des Endsymbols entfällt. Deshalb erfolgt darauf basierend eine Unterscheidung in die Konzepte `RegexLiteral` und `WrapLiteral`. Konkrete Literalarten können dann als Instanzen dieser beiden Konzepte spezifiziert werden. Typische Beispiele für `RegexLiteral` sind ganze Zahlen und Fließkommazahlen, während Zeichenketten sich – wie bereits erwähnt – mithilfe von `WrapLiteral` elegant definieren lassen. Bei `RegexLiteral`-Instanzen muss mithilfe von `typeToMap` zudem spezifiziert werden, welchen literalen Datentyp das generierte Attribut erhalten soll. Damit gibt ein solches Literal immer einen Grundtenor vor.
- *Begrenzer*: Begrenzer spiegeln die Klasse der Interpunktionssymbole wider. Es handelt sich hierbei also um Symbole, die andere Arten von Tokens oder Sequenzen semantisch zusammengehörender Tokens voneinander trennen. Für den ersten Fall existiert das Konzept `TokenSeparator`, das für die Trennung zweier Tokens verantwortlich zeichnet. Ein typischer Vertreter des Token-Separators ist das Leerzeichen. Sequenzen semantisch zusammengehörender Tokens hingegen werden mithilfe von Containerknoten umgesetzt. Deshalb werden die übrigen Spezialisierungen des Konzepts `Delimiter` erst in den folgenden

Abschnitten zusammen mit den betreffenden Containern erläutert. Wegen dieses strukturgebenden Charakters wird das konkrete Begrenzersymbol stets in den Konstrukten auf Definitionsseite festgesetzt und ist somit rein visuell. Generell für alle Begrenzerinstanzen gilt, dass sie im Attribut `terms` alle zugehörigen Symbole in Form von Strings spezifizieren müssen. Zudem kann mittels `ignoreWhitespace` festgelegt werden, ob Leerraumzeichen (unter anderem Leerzeichen, Zeilenumbrüche und Tabulatoren) vor und nach den jeweiligen Symbolen zu ignorieren sind.

- *Kommentar*: Ein Kommentar ist ein Token, das ausschließlich dokumentierenden und damit rein visuellen Gesichtspunkten dient. Die Festlegung eines Kommentars erfolgt immer durch ein wohldefiniertes Anfangs- und Endsymbol. Auch einzeilige Kommentare lassen sich mit diesem Prinzip emulieren, und zwar durch die Angabe des Zeilenumbruchs als Endsymbol.

7.2.2 Klassifizierung von Containern

Bevor die einzelnen Spezialisierungen von `Delimiter` eingeführt werden, ist es wichtig die Intention der verschiedenen Containerarten zu verstehen. Sie können unterteilt werden in Statements, Blöcke und Gruppen, wobei letztere spezifisch für den hier vorgestellten projizierenden Ansatz sind.

- *Statement*: Jedes Statement repräsentiert den Wurzelknoten einer Konstruktverwendung und entspricht somit einer klassischen Instanz auf Domänenseite. Im textuellen Kontext ist ein Statement also stets ein semantischer Knoten. Aus Gründen einer einfachen Handhabung bei der Umsetzung des textuellen Modellierungsworkflows (Abschnitt 7.3), besitzen einzelne Tokens nie semantischen Charakter, sondern müssen dazu immer in ein Statement oder eine Gruppe (siehe unten) eingebettet sein. Im Fall von Containern legt der Knotentyp im Textuellen eindeutig fest, ob ein Knoten semantischer oder rein visueller Natur ist. Ein Statementknoten besitzt ausschließlich rein visuelle Kinder und darf daher keine anderen Statementknoten enthalten. Enthält ein Statement einen Bezeichner, dann spezifiziert er den Namen, über den es referenziert werden kann. Folglich darf ein Statement maximal einen Bezeichner umfassen. (Konzept `Statement`)

- *Gruppe*: Gruppen sind rein visuelle Knoten, die es ermöglichen, rein visuelle Kindknoten eines Statements zu einer Einheit zusammenzufassen. Eine Gruppe wird für gewöhnlich dann eingesetzt, wenn die Sichtbarkeit der gruppierten Knoten an einem zentralen Ort kontrollierbar sein soll. Das ist meistens dann der Fall, wenn diese Sichtbarkeit semantischen Charakter hat und deshalb über eine Entsprechung auf Domänenseite verfügt. Als Beispiel hierfür dient Listing 7-2. Es definiert eine Entität `Student` mit den drei Attributen `matriculationNumber`, `familyName` und `surname`. Lediglich das erstgenannte Attribut ist mithilfe von `as key` als Primärschlüssel gekennzeichnet. Hierbei handelt es sich um zwei eigenständige Schlüsselwörter, die nur gemeinsam auftreten dürfen (nämlich dann, wenn ein Attribut Teil des Primärschlüssels ist) und deswegen zusammen mit dem dazwischen liegenden Leerzeichen gruppiert werden müssen.

```
1  entity Student
2      matriculationNumber: number as key
3      familyName: string
4      surname: string
```

Listing 7-2: Textuelles ER-Modell mit der Definition einer Entität "Student"

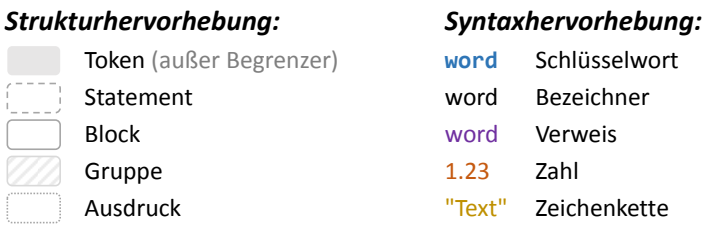

Abbildung 7-3: Legende für die Bedeutung der verschiedenen strukturellen und syntaktischen Hervorhebungen

- *Block*: Blöcke sind rein visuelle Knoten mit der Restriktion, ausschließlich semantische Knoten aufnehmen zu können. Sie besitzen also immer eine semantische Enthaltensein-Referenz, weshalb sie auch als semantische Container bezeichnet werden. (Konzept `Block`)

Analog zum Token-Separator gibt es auch spezifische Begrenzer, die im unmittelbaren Zusammenhang mit Statements und Blöcken stehen.

- *Statement-Separator*: Ein Statement-Separator trennt ganze Statements sequentiell voneinander. Wird ein derartiger Separator eingegeben, dann wird er nicht nach dem aktuellen Token platziert, sondern hinter dessen umgebenden Statementknoten. Innerhalb des Modells darf ein Statement-Separator allerdings nicht der semantischen Enthaltensein-Beziehung des betreffenden Blocks hinzugefügt werden, weil der Separator rein visuell ist und somit keine Entsprechung auf Domänenseite aufweist. Deshalb gibt es eine spezielle und stets rein visuelle Enthaltensein-Beziehung, die vom Konzept `Separators` repräsentiert wird. Im Definitionsmodell werden mit ihr die möglichen Statement-Separatoren festgelegt (für gewöhnlich nicht mehr als einer), die in den Verwendungen eingesetzt werden dürfen. Diese Beschränkung gilt nur bei der stringenten Modellierung. In Verwendungsmodellen dagegen werden über diese Beziehung die konkreten Separatorknoten referenziert, die im Dokument als eigenständige Tokens zwischen den einzelnen Statements angezeigt werden. Ein konkretes Beispiel, das den Einsatz des `Separators`-Konzepts demonstriert, wird am Ende dieses Abschnitts geliefert. Typische Vertreter von Statement-Separatoren sind Kommas, Semikolons und Zeilenumbrüche. (Konzept `StatementSeparator`)
- *Block-Einleitung*: Einfache Aneinanderreihungen von Blöcken, die nur durch ein Interpunktionssymbol voneinander getrennt sind, sind in existierenden Sprachen äußerst selten anzutreffen. Stattdessen findet man Blöcke häufig platziert zwischen Klammern oder nach einem Zeilenumbruch mit anschließender Einrückung. Demzufolge gibt es keine Block-Separatoren, sondern sogenannte Block-Einleitungen, die direkt an der jeweiligen Cursor-Position dem

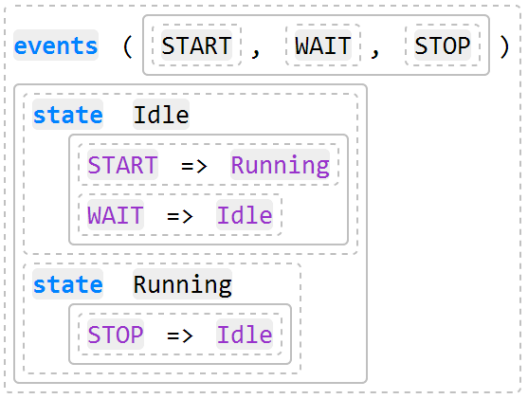

Abbildung 7-4: Textuelles Beispiel eines Zustandsautomaten mit Hervorhebung von Syntax und Struktur

aktuellen Statement einen neuen Block hinzufügen. Im Vorfeld ist für jedes Symbol im `terms`-Attribut außerdem ein Pendant im `closingTerms`-Attribut zu hinterlegen. Ist dieses kein leerer String, dann wird das angegebene Abschlusssymbol jedes Mal als weiterer Token-Separator direkt nach dem Block eingefügt. Dies entspricht dem bekannten Verhalten moderner IDEs. (Konzept `BlockIntroducer`)

Einen speziellen Separator, der typischerweise vor Gruppen steht und diese somit einleitet, gibt es allerdings nicht. Der Grund dafür ist, dass es sich bei Gruppen um ein Hilfskonstrukt handelt, das ausschließlich in projizierenden Editoren von Nutzen ist. Dort dient es primär dem oben angegebenen Zweck. Gruppen müssen folglich immer manuell erstellt werden (Abschnitt 7.4.2.9).

Vor dem Blick auf ein kleines Beispiels ist es sinnvoll, die in den Beispielen verwendete Hervorhebungen inklusive deren Semantik zu kennen. Diese sind in der Legende in Abbildung 7-3 dargestellt. Als eigener Punkt aufgelistet ist auch „Ausdruck", obwohl dafür kein entsprechender Knotentyp im textuellen Meta-Modell existiert. Diese Darstellung wird später zu Vereinfachungszwecken eingesetzt, weshalb sie hier der Vollständigkeit halber angegeben ist. Im Übrigen werden aus Übersichtlichkeitsgründen von den Tokens nur diejenigen grau hinterlegt, die keine Begrenzer sind.

Abbildung 7-4 zeigt denselben beispielhaften Zustandsautomaten wie in Listing 7-1. Zusätzlich sind jedoch sämtliche Container explizit hervorgehoben, so dass die interne Baumstruktur direkt abgelesen werden kann. Das Vorhandensein dieser internen Strukturierung ist für die spätere Interpretation während der automatisierten Ableitung eines Domänen-Meta-Modells zwingend notwendig. Durch die explizite Visualisierung der Container kann der Benutzer bereits im Zuge der freien Modellierung evaluieren, ob die erfasste Struktur seinen Vorstellungen entspricht. Ist dem nicht so, kann er sie gezielt

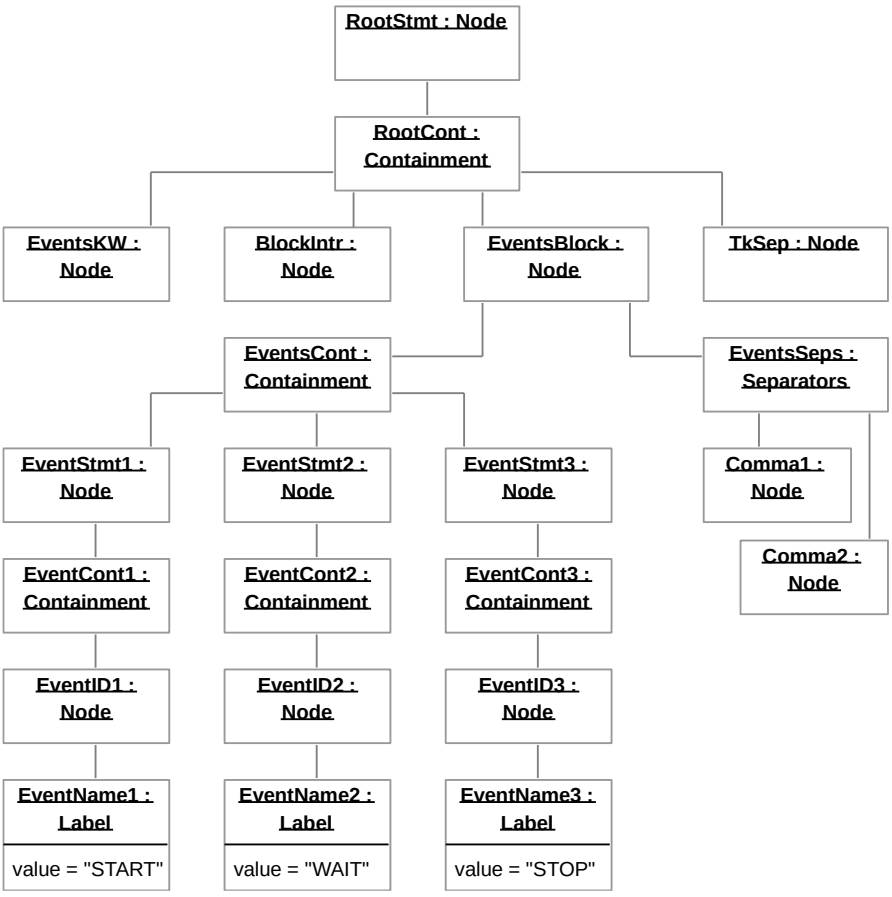

Abbildung 7-5: Ausschnitt des visuellen Verwendungsmodells für den beispielhaften Zustandsautomaten

anpassen und sieht gleichzeitig das Ergebnis seiner Arbeit. Wie dies genau geschieht, wird bei der Erläuterung des textuellen Modellierungsworkflows in den Abschnitten 7.4.1.3 und 7.4.2.9 geschildert.

Das Beispiel des Zustandsautomaten in Abbildung 7-4 ist wie folgt aufgebaut: Die dargestellte Wurzel ist ein Statement, welches den eigentlichen Zustandsautomaten repräsentiert. An erster Stelle enthält es das Schlüsselwort `events` gefolgt von einer öffnenden Klammer, die eine Block-Einleitung repräsentiert. Deshalb befindet sich im Anschluss daran ein Block, der in Form von eigenständigen Statements die drei Ereignisse `START`, `WAIT` und `STOP` deklariert. Wegen des jeweils beinhalteten Bezeichners sind alle Ereignisse über den angegebenen Text referenzierbar. Nach der darauffolgenden schließenden Klammer kommt ein Block, innerhalb dessen die beiden Zustände `Idle` und `Running` spezifiziert werden. Jeder Zustand setzt sich aus dem Schlüsselwort `state`, einem Bezeichner und einem eingerückten Block mit möglichen Zustandsübergängen zusammen. Diese Übergänge sind selbst wieder Statements. Beim ersten Token handelt es sich immer um einen Verweis auf ein Ereignis. Danach folgt ein Folgepfeil (=>), dem sich wiederum ein Verweis auf einen Zielzustand anschließt.

Einen Ausschnitt des zu diesem Beispiel gehörenden visuellen Verwendungsmodells zeigt Abbildung 7-5. Beschränkt wird sich dabei aus Platzgründen auf die erste, mit dem Schlüsselwort `events` eingeleitete Zeile. Der entsprechende Ausschnitt des zugrundeliegenden visuellen Definitionsmodells hingegen ist in Abbildung 7-6 illustriert, wobei dieselben Namenskonventionen wie in Abschnitt 5.2.3 zum Tragen kommen. Welchen Typ die einzelnen Knoten besitzen, ist ebenfalls anhand von Fragmenten des jeweiligen Namens ersichtlich. „Stmt" steht hierbei für `Statement`, „KW" für `Keyword`, „BlockIntr" für `BlockIntroducer`, „TkSep" für `TokenSeparator` und „ID" für `Identifier`.

Wie schon im allgemeinen Beispiel deutlich wurde, werden alle statischen Teile auf Definitionsebene festgelegt. Das gilt somit für das Schlüsselwort `events`, für die beiden Klammern und außerdem für

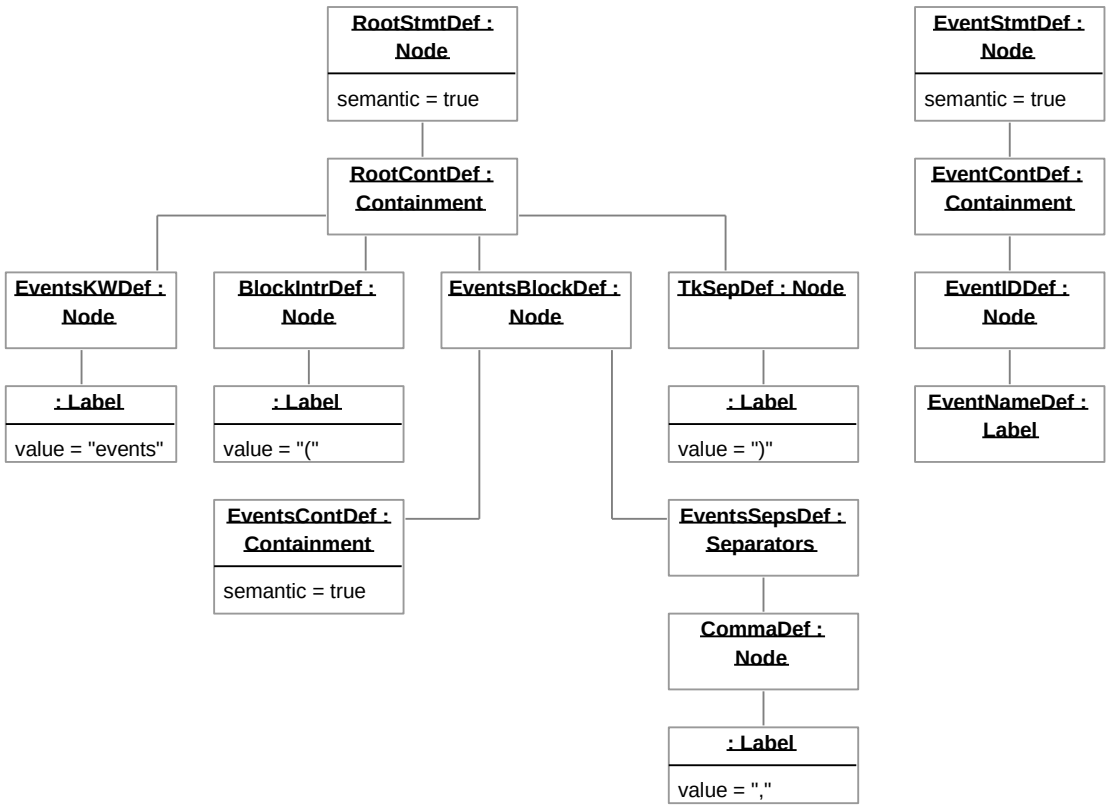

Abbildung 7-6: Ausschnitt des visuellen Definitionsmodells für beispielhaften Zustandsautomaten

das Komma als Separatorsymbol. Letzteres ist als einzig zulässiger Statement-Separator für die Aufzählung von Ereignissen deklariert, da `CommaDef` als alleiniger Zielknoten von `EventsSepsDef` angegeben ist. Im Verwendungsmodell gibt es von `CommaDef` zwei Verwendungen, die die konkreten Kommas `Comma1` und `Comma2` zwischen den drei Ereignis-Statements `EventStmt1`, `EventStmt2` und `EventStmt3` widerspiegeln.

7.2.3 Berechnungsvorschriften als Ausdrücke

Eine Vielzahl von GPLs und DSMLs bietet die Möglichkeit Berechnungsvorschriften in Form mathematischer Ausdrücke hinzuschreiben (z.B. eine einfache Berechnung wie `size * 8 + offset`). Sie **adäquat zu unterstützen** ist daher abermals **dem Intuitionismus zuträglich**. Adäquat bedeutet, dass bei der Erkennung solcher Ausdrücke eine Baumstruktur angelegt wird, die einem äquivalenten abstrakten Syntaxbaum in GPLs entspricht. Damit erhält der Benutzer eine passgenaue Vorschau auf die Struktur, die beim späteren Ableiten der abstrakten Syntax resultiert und kann sie bei Bedarf seinen Vorstellungen anpassen. Die Äquivalenz von abstrakter Syntax und Baumstruktur des Dokuments (d.h. eins-zu-eins Abbildung) ist dem Bottom-Up Entwurf geschuldet. Andere Arten von Mappings sind zwar möglich, doch müssen sie weiterhin händisch spezifiziert werden, da dem Induktionsalgorithmus die hierfür erforderlichen Zusatzinformationen fehlen.

Nachdem einige DSMLs auf existierenden GPLs (oftmals als Basissprache betitelt) aufsetzen, wird dort auf die mitgelieferte Unterstützung für Ausdrücke gesetzt. Dies ist somit eine durchwegs gängige **Alternative**, die mit „Piggyback" [131] sogar als spezielles Entwurfsmuster für DSMLs bezeichnet wird. Beim **Einsatz einer Basissprache** handelt es sich allerdings lediglich um eine Verschränkung von stringenter und freier Modellierung, wie sie später in Abschnitt 7.6 vorgestellt wird. Es wird hier also nicht gesondert darauf eingegangen.

Konzeptuell betrachtet stellt die **Unterstützung von Ausdrücken** keine zusätzliche Herausforderung dar, sondern ist eher als eine von zahlreichen denkbaren **Erweiterungen der beispielgetriebenen Entwicklung** von DSMLs anzusehen. Aufgrund ihrer weiten Verbreitung ist der überwiegende Teil aller Softwareentwickler im Umgang mit Ausdrücken vertraut. Im weiteren Verlauf dieses Kapitels dienen sie deshalb an vielen Stellen als Anwendungsfall und zeigen dabei zugleich die generische Anwendbarkeit des propagierten Ansatzes. Trotz der besagten Vertrautheit ist deren interne Repräsentation gepaart mit den potentiellen Interaktionsmöglichkeiten nicht zwangsläufig selbstverständlich. Deswegen werden Ausdrücke im Folgenden immer wieder aufgegriffen und bei Bedarf näher beleuchtet. Den Anfang machen die kommenden drei Unterabschnitte, die sich mit der Umsetzung auf Modellseite befassen und binäre, Klammer- sowie unäre Ausdrücke behandeln.

Neben unären Ausdrücken finden sich in zahlreichen Programmiersprachen auch ternäre Ausdrücke der Form `value ? 100 : 50`. Sie sind jedoch spezifisch für GPLs und kommen nach unserer Erfahrung nur in solchen DSMLs vor, die auf einer GPL mit Unterstützung für ternäre Ausdrücke basieren. Demzufolge gibt es im textuellen Meta-Modell keine speziellen Konzepte für die Erkennung derartiger Ausdrücke.

7.2.3.1 Binäre Ausdrücke

Auch Ausdrücke lassen sich mit den bereits bekannten Mitteln Statement und Block verwirklichen, wenngleich die resultierende Struktur etwas komplexer ausfällt als bei gewöhnlichen Parse-Bäumen. Geschuldet ist dies der allgemeingültigen Einsatzmöglichkeiten von Statements und Blöcken. Denn jeder Operand ist semantischer Natur und liegt deswegen immer in einem Block. Zudem ist jedes Token (gemäß der Festlegung in Abschnitt 7.2.2 unter dem Punkt „Statement") in ein Statement eingebettet.

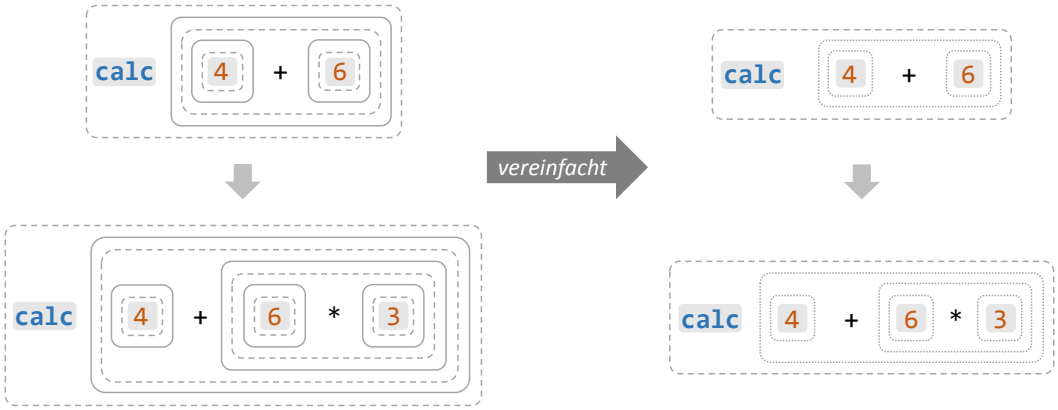

Abbildung 7-8: Gegenüberstellung von tatsächlicher und vereinfachter Visualisierung der Struktur binärer Ausdrücke

Um den Benutzer nicht unnötig zu überfordern, ist es deshalb sinnvoll, die Struktur vereinfacht zu visualisieren. Abbildung 7-8 zeigt hierzu die tatsächliche (links) und eine mögliche vereinfachte (rechts) Darstellung der Baumstruktur für die Ausdrücke `4 + 6` sowie `4 + 6 * 3`. Bei der vereinfachten Darstellung werden Blöcke ausgeblendet und Statements mit einer gepunkteten Umrandung notiert. Diese Umrandung steht gemäß der Legende aus Abbildung 7-3 für Ausdrücke. Um das Zusammenspiel mit einem etwaig vorhandenen Kontext erkennen zu können, sind alle Beispielausdrücke in ein weiteres Statement eingebettet, das mit dem Schlüsselwort `calc` eingeleitet wird.

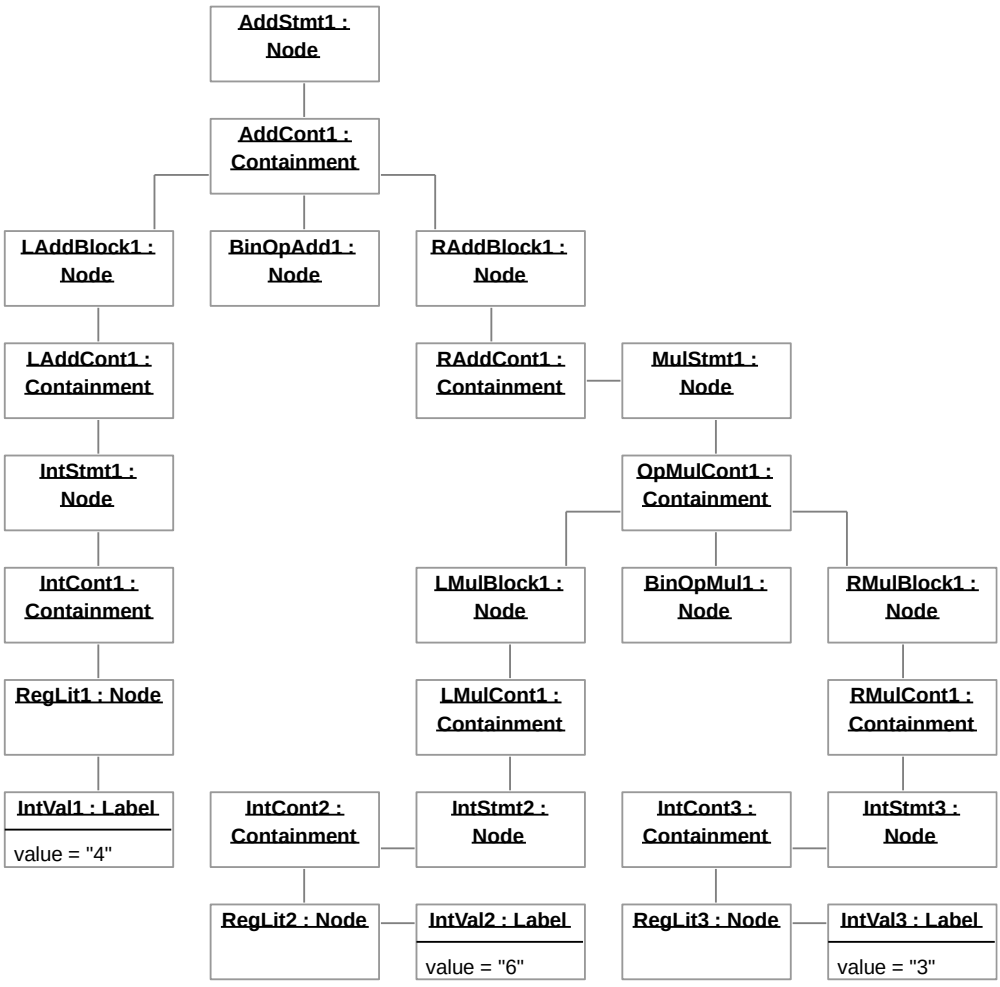

Abbildung 7-7: Ausschnitt aus visuellem Verwendungsmodell für den binären Ausdruck 4 + 6 * 3

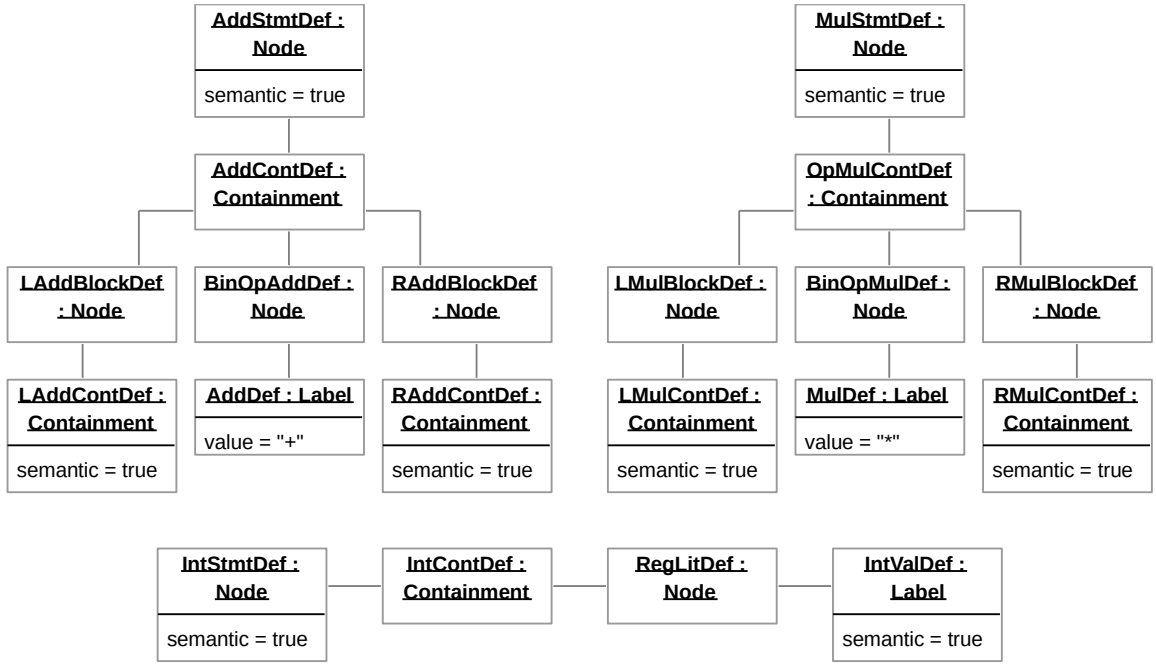

Abbildung 7-9: Ausschnitt aus visuellem Definitionsmodell, der zum binären Ausdruck 4 + 6 * 3 passt

Im Folgenden beschränken wir uns vorerst auf binäre Ausdrücke. Jeder derartige Ausdruck basiert auf einem binären Operator, der zwischen zwei Operanden liegt und von einem als `BinaryOperator` typisierten Knoten repräsentiert wird. Die hierbei vorgestellten Prinzipien sind allgemeingültig und können – wie später erklärt wird – auch auf andere Arten von Ausdrücken übertragen werden.

Der zum Ausdruck `4 + 6 * 3` passende Ausschnitt eines visuellen Verwendungsmodells ist in Abbildung 7-7 illustriert, wohingegen Abbildung 7-9 das Fragment des zugehörigen Definitionsmodells veranschaulicht. Die Namen der verschiedenen Knoten geben darin erneut Auskunft über den jeweiligen Knotentyp. So meint „Stmt" den Typ `Statement`, „Block" den Typ `Block`, „BinOp" den Typ `BinaryOperation` und „RegLit" den Typ `RegexLiteral`.

Damit wird deutlich, dass literale Werte von einem Statement gekapselt werden müssen, um anstelle von anderen Ausdrücken referenziert werden zu können. Auf Definitionsebene existiert zur Kapselung das Statement `IntStmtDef`, das mit `RegLitDef` über ein Token verfügt, welches eine ganze Zahl repräsentiert. Für die eigentliche Aufnahme dieser Zahl dient die semantische Label-Definition `IntValDef`. Die darauf beruhenden Verwendungen legen die einzelnen literalen Werte fest, und zwar im Beispiel die Zahlen `4`, `6` und `3`.

Etwas anders verhält es sich mit den Symbolen von Operatoren. Sie werden bereits im Definitionsmodell durch rein visuelle Labels festgelegt (siehe `AddDef` und `MulDef`) und sind deshalb fixer Bestandteil der betreffenden Ausdrucksdefinition. Für jedes unterscheidbare Operatorsymbol gibt es somit eine separate Ausdrucksdefinition (im Beispiel `AddStmtDef` und `MulStmtDef`).

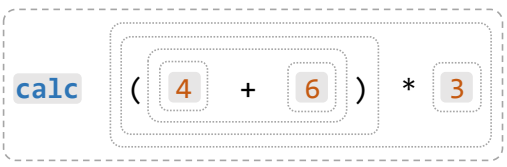

Abbildung 7-10: Exemplarischer Klammerausdrucks unter Anzeige der vereinfachten Struktur

Ein im Hinblick auf den Intuitionismus wichtiger Aspekt ist die Präzedenz von Operatoren. Sie wurde im obigen Beispiel durch Erfüllung der Regel „Punkt vor Strich" bereits berücksichtigt, wodurch der in Abbildung 7-7 visualisierte Baum rechtsschwer ausfällt. Wäre die Regel hingegen ignoriert worden, so hätte dies zu einem linksschweren Resultat geführt, was der inkorrekten Auswertungsreihenfolge `(4 + 6) * 3` entspräche. Die Operatorpräzedenz wird im Definitionsmodell durch eine `Operator`-Instanz pro Präzedenzstufe festgelegt. Diese Instanzen müssen zusätzlich eine entsprechende Sortierung aufweisen (z.B. durch die Reihenfolge, in der sie im Definitionsmodell liegen).

7.2.3.2 Klammerausdrücke

Um dem Mechanismus der Operatorpräzedenz aktiv entgegenzuwirken, werden Berechnungsvorschriften üblicherweise mit einer geeigneten Klammerung versehen. Soll beispielsweise beim Ausdruck `4 + 6 * 3` die Addition vor der Multiplikation greifen, dann muss die Addition in Klammern eingefasst werden. Der resultierende Ausdruck lautet danach `(4 + 6) * 3`.

Eingeleitet werden Klammerausdrücke mit einem Token, das durch eine Instanz des bereits vorgestellten Konzepts `BlockIntroducer` typisiert ist. Jeder Klammerausdruck ist eine separate Statement-Verwendung, die sich demnach aus einer öffnenden Klammer, einem Block für weitere (eingebettete) Ausdrücke und einer zugehörigen schließenden Klammer zusammensetzt. Exemplarisch zeigt Abbildung 7-10 die vereinfachte Struktur des oben aufgeführten Klammerausdrucks, an der dieser Aufbau hervorgeht (zur Erinnerung: eine gepunktete Umrandung repräsentiert einen Ausdruck und damit einen Block mit eingeschachteltem Statement).

Dieselbe vereinfachte Struktur findet sich auch bei Jetbrains MPS wieder. Zu dem Zweck ist in Abbildung 7-11 ein Screenshot von MPS für die interne Baumstruktur des Ausdrucks `(4 + 6) * 3` dargestellt. Darin sind diejenigen Knoten des abgebildeten Baums gelb hinterlegt, die den einzelnen Knoten der vereinfachten Darstellung in Abbildung 7-10 entsprechen. Insbesondere von Relevanz sind Containerknoten, da sie der Strukturierung dienen. Im MPS-Beispiel handelt es sich hierbei um die drei Knoten `* {int}`, `(expr) {int}` und `+ {int}`.

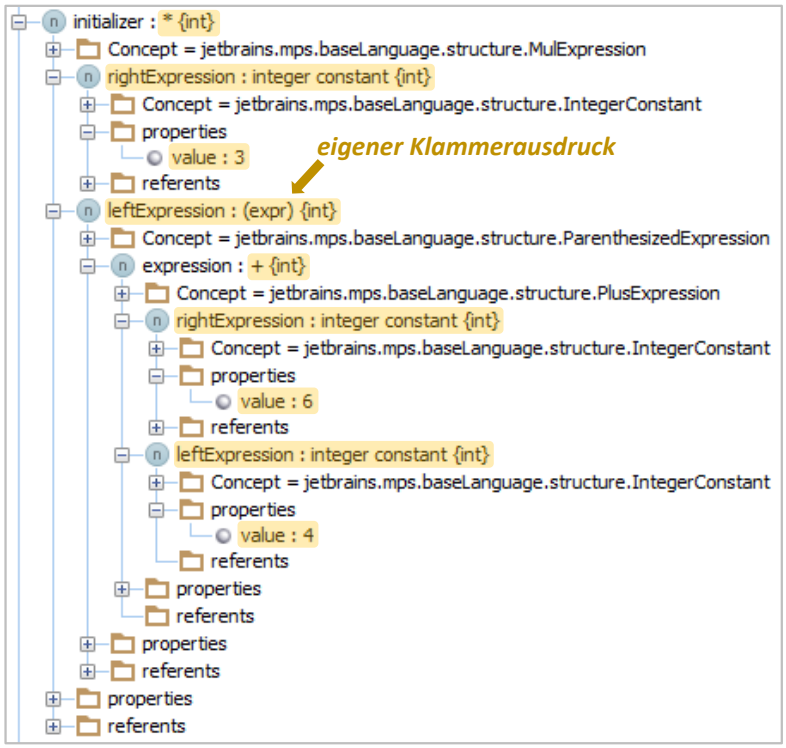

Abbildung 7-11: Interne Baumstruktur bei MPS für den Ausdruck `(4 + 6) * 3`

Jetbrains Ansatz hat sich in der Praxis inzwischen bewährt [140], weshalb wir ihn weitgehend übernehmen. Um jedoch im Rahmen der freien Modellierung generisch anwendbar zu bleiben, fällt die interne Baumstruktur mit dem Vorhandensein von Statements und Blöcken etwas komplexer aus als bei MPS. Doch Dank der vereinfachten Visualisierung der Baumstruktur ist dies für den Benutzer nicht weiter hinderlich (sofern er die Anzeige der Struktur überhaupt wünscht).

7.2.3.3 Unäre Ausdrücke

Unäre Ausdrücke sind ebenfalls häufige Begleiter bei der täglichen Arbeit mit formalen, textuellen Sprachen. Je nachdem ob der betreffende Operator vor oder hinter dem Operanden platziert wird, werden sie als Präfix- oder Postfixausdrücke bezeichnet. Analoges gilt für die Benennung des jeweiligen Operators. Typische Vertreter von Präfixoperatoren sind − zur Negierung und ! als logisches „Nicht". Aus an C++ [134] angelehnten Programmiersprachen bekannte Postfixoperatoren dagegen sind ++ und −−. Sie dienen üblicherweise dazu, den Wert einer Variablen an Ort und Stelle um 1 zu in- bzw. dekrementieren, und können auch als Präfixoperatoren eingesetzt werden. Die Handhabung von solchen potentiellen Mehrdeutigkeiten wird als Teil des Modellierungsworkflows erst in Abschnitt 7.4.2.8 behandelt.

Präfixausdrücke werden durch ein als Präfixoperator deklariertes Symbol initiiert. Ein solches Symbol muss dazu im `terms`-Attribut einer `PrefixOperator`-Instanz im Definitionsmodell hinterlegt sein. Abbildung 7-12 zeigt hierzu einen Ausschnitt aus einem Verwendungsmodell (links) inklusive zugehöriger Definitionen (rechts) für den Präfixausdruck `++count`. Die Vernetzung der einzelnen Konzepte ist dabei konform zu der von binären und Klammerausdrücken. Beim Verwendungsmodell wurde bewusst auf die Darstellung der semantischen Referenz verzichtet, weil dies an der Stelle nebensächlich ist.

Analog dazu existiert für die Unterstützung von Postfixausdrücken das Konzept `Postfix-Operator`.

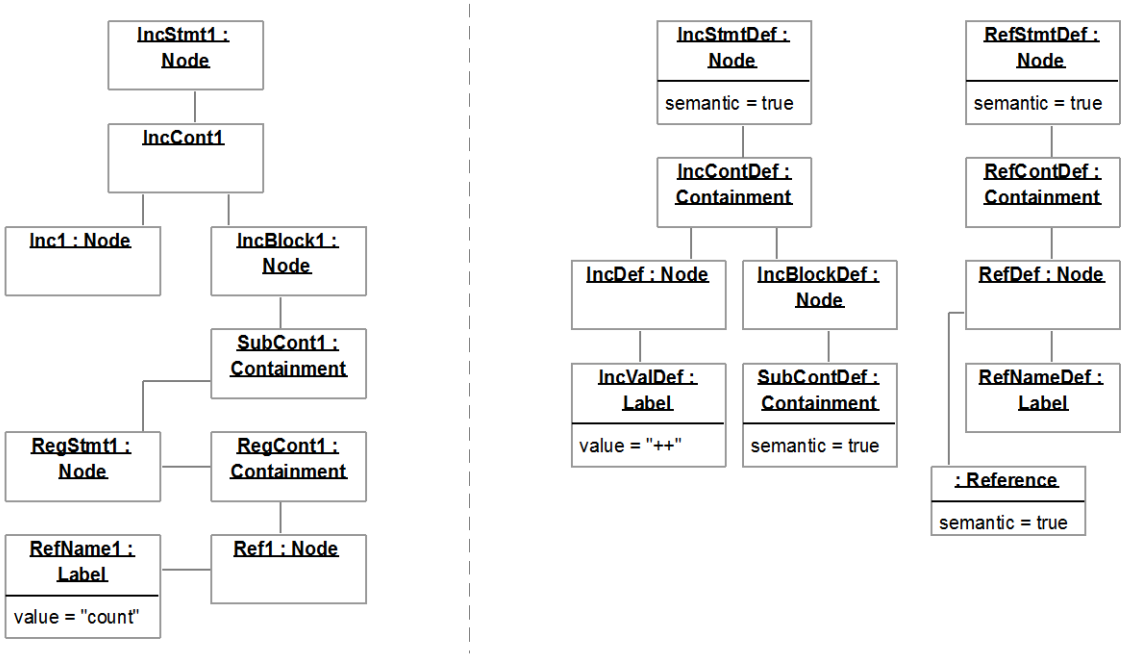

Abbildung 7-12: Verwendung und zugehörige Definition für den beispielhaften Präfixausdruck `++count`

7.2.4 Komfortable Unterstützung für Verweise

Um in textuellen Dokumenten die enthaltenen Entitäten referenzieren zu können, benötigen diese Namen. Eine **Referenz auf** eine solche **Entität** kann dann durch die einfache **Nennung ihres Namens** etabliert werden. In strukturarmen Sprachen (d.h. in Sprachen mit geringer Schachtelungstiefe wie dem Zustandsautomatenbeispiel aus Abbildung 7-4) ist diese Methode völlig ausreichend.

Anders kann es aber bereits mit Sprachen aussehen, deren Schachtelungstiefe größer ausfällt. Das linksseitige Beispiel in Abbildung 7-13 ist in einer Sprache formuliert, die drei Hierarchieebenen gestattet. Es zeigt den Stammbaum von Familie Maier, in dem die beiden Personen Josef und Peter aufgeführt sind, wobei Josef der Vater von Peter ist. Zur Vermeidung von Redundanz soll darin Peter den Nachnamen von Josef übernehmen, was jedoch nicht ohne weiteres Zutun möglich ist. Durch die erneute Angabe von `name` innerhalb der `name`-Zuweisung wird diese selbst referenziert. Das resultiert in einer endlosen, rekursiven Spezifikation von Peters Nachnamen.

Geschuldet ist die Selbstreferenzierung der sogenannten **Überdeckung** (engl. shadowing) [139]. Erwartungsgemäß werden bei gleichnamigen Statements nämlich diejenigen bevorzugt, die innerhalb der Hierarchie näher am aktuellen Verweis-Token liegen. Demnach wird erst im aktuellen Block gesucht, wodurch der Mechanismus bereits mit der zu Peter gehörenden Namensdeklaration fündig wird und sich somit auf dieses Statement bezieht. Erst wenn im lokalen Kontext kein passendes Statement gefunden wird, wird die Suche auf der nächst höheren Hierarchiestufe fortgesetzt. Das wird rekursiv solange fortgeführt, bis entweder ein passendes Statement entdeckt oder die Wurzel des Dokuments erreicht ist.

7.2.4.1 Qualifizierende Namen

Durch die bloße Angabe von „name" kann die Namensdeklaration von Josef nicht aufgelöst werden, weil stets die Namensdeklaration von Peter im Weg steht. **Im Rahmen hierarchischer Strukturen formaler, textueller Sprachen** haben sich deshalb sogenannte qualifizierende Namen **etabliert**, die dieser Problematik begegnen. Sie basieren auf dem Konzept der vollqualifizierenden Namen, müssen ihnen gegenüber jedoch nur im jeweiligen Scope (Namensraum) eindeutig auflösbar sein. Gemäß der Definition in [144] setzen sich vollqualifizierende Namen immer aus den in einer Sequenz angeordneten einzelnen Namen der Elternelemente sowie dem Namen des betreffenden Elements selbst zusammen. Zusätzlich sind die einzelnen Namen durch ein spezielles Trennzeichen voneinander separiert. In Programmiersprachen wird hierfür häufig ein Punkt eingesetzt, was bei Anwendung auf die Namensfestlegung von Josef den vollqualifizierenden Namen `Maier.Josef.name` ergibt.

Insbesondere bei tiefen hierarchischen Strukturen ist die stete Angabe vollqualifizierender Namen mit viel Schreibarbeit verbunden. Deswegen unterstützt ein Großteil der modernen Sprachen auch qualifizierende Namen, die **lediglich im aktuellen Namensraum eindeutig** sein müssen und sich typischerweise aus erheblich weniger Einzelnamen der Gesamthierarchie aufbauen.

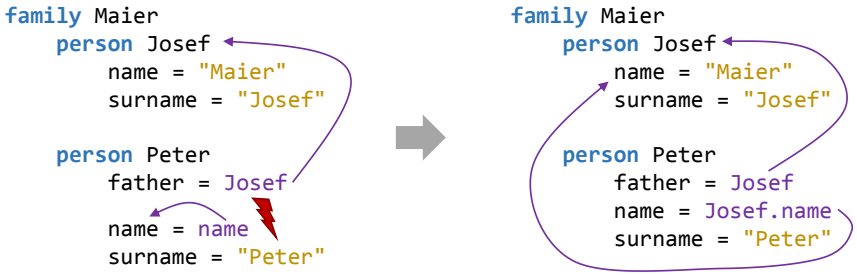

Abbildung 7-13: Textueller Familienstammbaum ohne (links) und mit (rechts) Einsatz eines qualifizierenden Namens

Auch der vorgestellte Ansatz bietet tief integrierte Unterstützung dieses Prinzips, wobei jeder Block seinen eigenen Namensraum aufspannt. Die tiefe Integration sieht so aus, dass qualifizierende Namen direkt von Verweisen unterstützt werden. Es handelt sich dabei um ein komfortables und einfaches Mittel, verschiedene, aber auf den ersten Blick gleichnamige Statements voneinander unterscheiden zu können. Die **Anzeige dieser Namen** ist ohnehin **nur für den Benutzer von Bedeutung**, da wegen des projizierenden Ansatzes Referenzen auf andere Knoten als solche im Modell abgespeichert werden. Im obigen Beispiel genügt daher bei der Namensfestlegung für Peter das Aufführen von `Josef.name`. Wie aus Abbildung 7-13 (rechte Seite) ersichtlich ist, wird dafür ein einziges Verweis-Token eingesetzt, das direkt das `name`-Statement von Maier referenziert. Für das spätere Ableiten einer abstrakten Syntax ist nur diese Referenz relevant, während die Angabe des qualifizierenden Namens ausschließlich dem Benutzer dient. So kann er leicht erkennen, auf welches `name`-Statement sich der Verweis bezieht.

Um das **Trennsymbol für qualifizierende Namen** flexibel pro Sprache festlegen zu können, bietet `Referral` das `separator`-Attribut. Durch mehrere Instanzen von `Referral` können sogar verschiedene Arten von Verweisen umgesetzt werden, die sich anhand des Trennsymbols auch optisch voneinander unterscheiden.

7.2.4.2 Typsystem-abhängige Verweise

Das rechte Beispiel in Abbildung 7-13 ist mit der doppelten Referenz auf die Person Josef immer noch nicht frei von Redundanz. Denn einerseits ist mit `father = Josef` angegeben, dass Josef der Vater von Peter ist. Andererseits wird bei der Festlegung des Nachnamens von Peter durch `name = Josef.name` erneut (mittelbar) auf Josef verwiesen. Eine alternative Lösung dazu veranschaulicht Abbildung 7-14. Darin wird sich bei der Namensdeklaration auf den Vater bezogen, und zwar mittels `father.name`. Es handelt sich dabei jedoch nicht länger um einen qualifizierenden Namen, da das `father`-Statement kein Kind-Statement mit dem Bezeichner „name" besitzt und somit die – gemäß der Definition – notwendige **Enthaltensein-Bedingung verletzt** ist.

Bei vielen existierenden Sprachen werden solche Verweise intern als spezielle binäre Ausdrücke abgelegt. Für deren Verarbeitung ist allerdings zusätzliches Domänenwissen erforderlich, das bei textuellen DSMLs in Form eines **Typsystems** vorliegt. Seine Aufgabe ist grob gesagt die Bereitstellung von Typinformationen für jedes Artefakt des Domänenmodells. Darauf, dass dieses Modell ganz oder auch nur in Teilen vorhanden ist, kann beim vorgestellten Ansatz jedoch nicht vertraut werden.

Die in einem Typsystem enthaltenen **Regeln** weisen oftmals eine **hohe Komplexität** auf. Ein prominentes Beispiel sind die erforderlichen Regeln zur Umsetzung von impliziten Konvertierungen, wie sie von der Programmiersprache Scala [2] unterstützt werden. Mithilfe dieses Musters ist es möglich, eine existierende Klasse implizit um zusätzliche Methoden zu erweitern, ohne dass hierfür die Klasse selbst

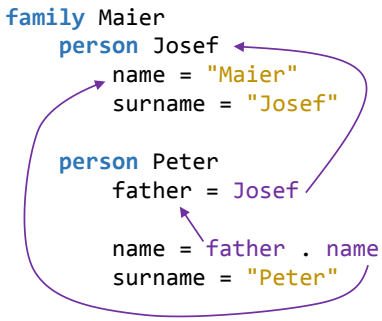

Abbildung 7-14: Beispiel für einen Typsystem-abhängigen Verweis

manipuliert werden müsste. Die entsprechenden Methoden werden extern definiert, wobei der erste Parameter von dem Typ sein muss, auf dem die zusätzliche Methode aufrufbar sein soll. Ein weiteres, ebenfalls in Scala unterstütztes Feature ist die Typinferenz. Hierbei sind Typen nicht mehr (oder nur äußerst selten) explizit zu spezifizieren, sondern können vom Typsystem aus Kontextinformationen berechnet werden. Aufgrund der enormen Vielfalt an Möglichkeiten, die zu der besagten hohen Komplexität führen, sind die dazu benötigten Berechnungsvorschriften auch weiterhin **durch den Sprachdesigner zur Verfügung zu stellen** und im Typsystem zu hinterlegen. Nachdem der Fokus dieser Dissertation auf den strukturellen Komponenten von DSMLs liegt, wird die Betrachtung von Typsystemen nicht weiter vertieft.

7.3 Textueller Modellierungsworkflow

Der textuelle Modellierungsworkflow lässt sich anhand der Interaktionsmöglichkeiten des Benutzers in drei Phasen untergliedern. Nachdem insbesondere die ersten beiden Phasen wegen ihrer Neuartigkeit sehr groß ausfallen und im Gegensatz zum grafischen Workflow deutlich voneinander zu unterscheiden sind, wurde jedem dieser Phasen ein eigener Abschnitt direkt innerhalb des 7. Kapitels gewidmet. Die Grafik in Abbildung 7-15 vermittelt einen Überblick über den gesamten Workflow und zeigt gleichzeitig, welchen Zustand die involvierten textuellen Konstrukte in der jeweiligen Phase innehaben.

Die erste und zweite Phase beschäftigen sich mit verschiedenen Stadien der freien Modellierung und repräsentieren einen Großteil des eigenen Beitrags innerhalb von Kapitel 7. Dieser Beitrag ist keine rein wissenschaftliche Leistung, sondern zudem die Bereitstellung einer Lösung für eine anspruchsvolle softwaretechnische Problemstellung. Insbesondere wegen ihrer Neuartigkeit war die Lösungsfindung eine komplexe implementierungslastige Herausforderung.

In der **ersten Phase** (Abschnitt 7.4) geht es um den Workflow bei der initialen Erstellung textueller Konstrukte. Diese Konstrukte befinden sich hier noch im initialen Entwurf, d.h. für jedes derartige Konstrukt existiert maximal eine Verwendung, die nach Belieben und ohne einschlägige Restriktionen editiert werden darf.

Phase 2 (Abschnitt 7.5) hingegen handelt von den Interaktionsmöglichkeiten mit der konkreten Syntax, bei der bereits vorhandene Konstrukte wiederverwendet und bei Bedarf manipuliert werden können. Solche Konstrukte werden als vollständig definiert bezeichnet, da es bereits mindestens eine weitere Verwendung gibt und die initiale Entwurfsphase somit abgeschlossen ist.

Die **dritte Phase** (Abschnitt 7.6) verfolgt schließlich die Integration von stringenter und freier Modellierung. Darin beachtet werden ausschließlich textuelle Konstrukte, die bereits über ein Mapping auf ein Element der abstrakten Syntax verfügen. Die Editiermöglichkeiten können hier durch die abstrakte Syntax stark begrenzt sein.

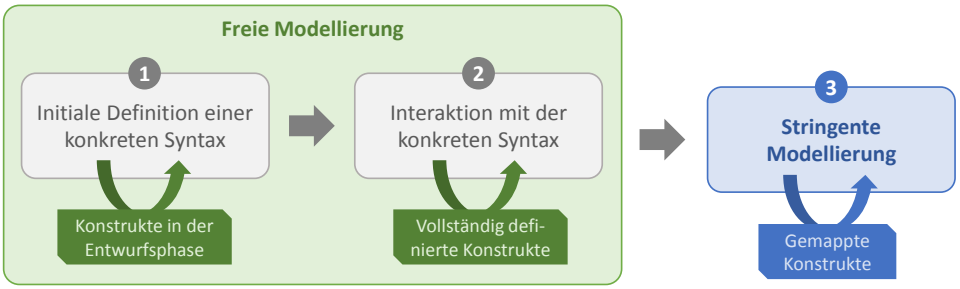

Abbildung 7-15: Übersicht des textuellen Modellierungsworkflows

Alle drei Phasen bauen inhaltlich aufeinander auf, und zwar in der Reihenfolge, wie in der Übersichtsgrafik durch die grauen Pfeile angegeben. Die Pfeile bezeugen jedoch keinen strikten Ablauf, in welcher Reihenfolge die einzelnen Phasen zu durchlaufen sind. Stattdessen drücken die Pfeile aus, wie sich der Zustand textueller Konstrukte über den Verlauf der Zeit ändert, nämlich von „initial" über „vollständig definiert" hin zu „gemappt".

7.4 Initiale Definition der konkreten Syntax

Durch die in Abschnitt 7.2 vorgestellten Erweiterungen des visuellen Meta-Modells ist eine Art konfigurierbare und damit flexibel änderbare **Super-Grammatik** definiert, die **im Zuge der freien Modellierung angewendet** wird. Gewährleistet ist die Konfigurierbarkeit durch die verschiedenen, beliebig parametrierbaren Tokentypen. Daneben stehen die drei Containertypen, die – auf Knoten angewendet – zur hierarchischen Strukturierung des Dokuments und somit zur Ausbildung des CST dienen.

Doch bevor der CST aufgebaut werden kann, muss der Benutzer Text eingegeben haben. Dies geschieht – wie bei Text-Editoren üblich – zeichenweise mithilfe einer Tastatur. Der eingetippte Zeichenstrom wird unter Rückbezug auf die konfigurierbare Super-Grammatik unmittelbar während der Eingabe in Tokens zerlegt, was der klassischen lexikalischen Analyse [1] entspricht (nähere Details folgen in Unterabschnitt 7.4.1). Die identifizierten Tokens werden gleichzeitig typisiert, was im Fall von Begrenzern dazu führt, dass eine Art syntaktische Analyse [1] nachgeschaltet wird (siehe dazu Unterabschnitt 7.4.1.3). Sie sorgt je nach erkanntem Begrenzer dafür, dass die lokale Baumstruktur intentionsgemäß angepasst wird. Demnach arbeitet der Modellierer auf dem CST und nicht auf dem Token-Stream oder gar dem Freitext. Die klassischen Varianten von lexikalischer und syntaktischer Analyse wurden daher für ein direktes, freies Modellieren auf Basis eines CST adaptiert.

Wiederholt sei an der Stelle noch einmal, dass es **zu jedem Konzept in den Verwendungsmodellen ein Pendant auf Definitionsseite** gibt. Dies gilt gleichermaßen für alle Arten von Tokens (ausgenommen Kommentare) und Container. Ist im Folgenden also davon die Rede, dass eine neue Verwendung angelegt wird, dann bedeutet es, dass gleichzeitig eine zugrundeliegende Definition erzeugt wird; es sei denn, eine Anmerkung besagt etwas Gegenteiliges. Statt „Verwendung" wird aus Gründen der Prägnanz oftmals auch direkt der jeweilige Typ benannt (z.B. Token statt Token-Verwendung und Statement statt Statement-Verwendung).

Getätigte **Änderungen an einer Verwendung werden** ebenso **direkt an das zugrundeliegende Konstrukt** der konkreten Syntax **delegiert**. Eine Ausnahme bilden lediglich die semantischen Eigenschaften von Knoten (im Regelfall das semantische Label von Tokens) sowie Kommentare. Aufgrund ihres informellen (und damit rein visuellen) Charakters kommen letztere ausschließlich auf Verwendungsseite vor.

Das Delegieren gestaltet sich oftmals schwierig, falls bereits weitere Verwendungen eines Konstrukts existieren. In den folgenden Unterabschnitten gilt jedoch die Prämisse, dass **jedes Konstrukt** genau einmal verwendet wird. Alle betrachteten Konstrukte befinden sich somit noch **in der initialen Entwurfsphase**, d.h. ein explizites Unterscheiden in Anpassung der Definition und Anpassung der Verwendung ist in diesem Abschnitt nicht erforderlich. Im Ausnahmefall übernimmt dies ohnehin der jeweilige Tokentyp. Näheres zum Umgang mit mehrfach verwendeten Konstrukten folgt in Abschnitt 7.5.

Analoges gilt auch für die **Differenzierung**, ob eine **Aktion semantischer oder rein visueller Natur** ist. Denn auch das wird durch die Knotentypen festgelegt. Lediglich in Bezug auf die Sichtbarkeit muss der Benutzer sich explizit für rein visuell oder semantisch entscheiden.

7.4.1 Lexikalische Analyse

Aufgabe der lexikalischen Analyse ist einerseits die **Zerlegung des eingegebenen Texts in Tokens** und andererseits die **Typisierung dieser Tokens**. Letztere findet immer direkt nach der Eingabe eines Zeichens statt. Dabei wird das geänderte Token mit sämtlichen im zugrundeliegenden visuellen Definitionsmodell vorhandenen Instanzen des Konzepts `Token` abgeglichen.

Standardmäßig ist jedes Token als **Token-Separator** deklariert, weil Token-Separatoren die Struktur des CST nur geringfügig beeinflussen, indem sie maximal zu zwei neuen Tokens führen (siehe dazu das Beispiel in Abbildung 7-16). Demzufolge gibt es keine Tokens ohne Typisierung, weshalb drei Fälle differenziert werden können, sobald der Text eines Tokens geändert wurde.

- *Tokentyp bleibt unverändert*: Keine weitere Aktion erforderlich
- *Tokentyp angepasst*: Der jeweilige Tokentyp wird darüber informiert und kann daraufhin weitere Maßnahmen ergreifen (z.B. Einfluss auf die Baumstruktur nehmen)
- *Kein passender Tokentyp vorhanden*: Zerlegung des Tokens an der Stelle, an der das Zeichen eingefügt wurde

Die in Frage kommenden Zerlegungsmöglichkeiten werden zunächst anhand der beiden typischen Eingabebeispiele aus Abbildung 7-16 verdeutlicht. Dabei gilt, dass das Wort „family" als Schlüsselwort registriert wurde, Bezeichner der Java-Spezifikation entsprechen und Leerzeichen als Token-Separatoren angesehen werden. Um auf den ersten Blick erkennen zu können, auf welches Token sich die momentane Eingabe auswirkt, ist das betreffende Token hellgrün hinterlegt. Dies gilt auch für Begrenzer, die sonst keine Färbung des Hintergrunds aufweisen (siehe Legende in Abbildung 7-3).

Linksseitig wird direkt nach `family` ein Leerzeichen eingefügt, was zur Folge hat, dass kein anderer Tokentyp zur Zeichenkette dieses Tokens passt. Die Zeichenkette muss somit in zwei Tokens zerlegt werden, nämlich in eines für `family` und ein weiteres für das eben eingetippte Leerzeichen. Nachdem sich das erste Token nicht geändert hat, bleibt auch dessen Typ erhalten. Das Leerzeichen muss allerdings typisiert werden, wodurch es als Token-Separator deklariert wird.

Auf der rechten Seite dagegen liegt ein Bezeichner-Token mit dem Wort „familyMaier" vor, in das zwischen „family" und „Maier" ebenfalls ein Leerzeichen eingefügt wird. Zu „family Maier" existiert kein passender Tokentyp, weshalb eine Zerlegung in die drei Tokens „family", Leerzeichen und „Maier" erfolgt. Das ursprüngliche Token kann als gelöscht betrachtet werden, da dessen Text nirgends mehr vorkommt. Demzufolge muss der Typ aller drei Tokens neu berechnet werden, was aus dem `family`-Token ein Schlüsselwort, aus dem Leerzeichen einen Token-Separator und aus „Maier" einen Bezeichner macht.

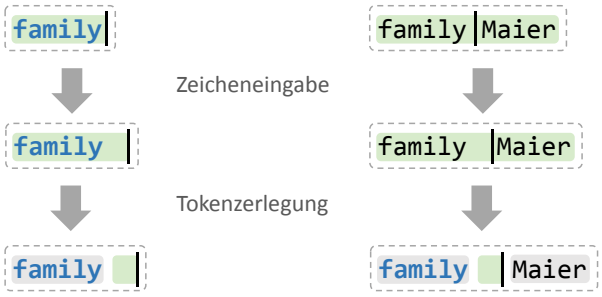

Abbildung 7-16: Zwei typische Beispiele für die lexikalische Tokenzerlegung

Ein eher selten auftretender, aber nichtsdestotrotz denkbarer Fall ist die Eingabe eines Zeichens innerhalb eines Tokens, bei dem lediglich **zwei Tokens während der Zerlegung** anfallen. Ein Beispiel hierfür ist das Token `value12`. Zählt nun das Minuszeichen zum regulären Ausdruck von ganzen Zahlen und wird es zwischen „value" und „12" eingefügt, dann resultieren daraus die beiden Tokens `value` und `-12`, weil Bezeichner in Java kein Minus enthalten dürfen.

Das **Anfügen eines Zeichens am Anfang eines Tokens** ist **nur auf indirektem Weg** möglich. Üblicherweise wirkt sich die Eingabe nämlich auf das jeweils vordere (erste) Token aus, sofern der Text-Cursor genau zwischen zwei Tokens liegt. Erst wenn ein eingegebenes Zeichen dazu führt, dass kein passender Typ zum geänderten Token gefunden werden kann, wird versucht dieses Zeichen an den Anfang des Nachfolgertokens zu hängen. Misslingt dabei die Typerkennung ebenso, dann wird ein neues Token für das eingegebene Zeichen erzeugt und direkt hinter dem ersten Token abgelegt.

Das in Abbildung 7-17 visualisierte Beispiel veranschaulicht diese drei Möglichkeiten, indem nach dem Leerzeichen (ein Token-Separator) jeweils ein anderes Zeichen eingegeben wird. Im oberen Fall wird ein Doppelpunkt eingetippt, der einfach an das Leerzeichen-Token angehängt wird, weil er in keinem anderen Tokentyp Verwendung findet. Im zweiten Fall wird der Buchstabe „v" eingegeben. Es gibt jedoch keinen Tokentyp, der mit einem Leerzeichen beginnen und einem „v" enden darf. Der Buchstabe wird anschließend dem Nachfolgertoken mit dem Text „Maier" vorangestellt, wodurch es weiterhin einen gültigen Bezeichner darstellt. Bei Eingabe der Zahl 3 im letzten Fall hingegen kann das Zeichen weder dem Vorgängertoken hinten noch dem Nachfolgertoken vorne hinzugefügt werden. Folglich wird dafür ein separates Zahl-Token angelegt und nach dem Leerzeichen-Token eingefügt.

Diese **Sonderbehandlung** kommt nur dann zum Tragen, wenn sich **Vorgänger- und Nachfolger-Token im selben Container** befinden. In allen anderen Fällen, bei denen das nachfolgende Token beeinflusst werden soll, ist dies explizit vom Benutzer anzugeben. Eingehend behandelt wird dieses Thema in Abschnitt 7.4.3.

Eine **Ausnahme** bildet jeweils das im gesamten Dokument **erste und letzte Token**. Bei Zeichen, die direkt vor dem ersten Token eingegeben werden, wird zunächst versucht, sie an den Anfang des Tokens zu stellen. Bezogen auf das vorherige Beispiel sind lediglich die beiden unteren Fälle zu beachten. Wird ein Zeichen direkt nach dem letzten Token eingetippt, so sind nur der oberste und der unterste Fall zu berücksichtigen.

Der **gesamte Ablauf der lexikalischen Analyse** bei der Eingabe eines Zeichens ist in Form eines informalen Aktivitätsdiagramms in Abbildung 7-18 veranschaulicht. Zunächst wird nur das jeweilige Token an der aktuellen Cursor-Position berücksichtigt (bei Mehrdeutigkeit ist es immer das vorhergehende Token). Nachdem das Zeichen an der richtigen Position in den Text des Tokens eingesetzt worden ist, wird basierend auf dem Text der Token-Typ bestimmt. Hat er sich geändert, muss das Token entsprechend angepasst werden und dabei sowohl Text als auch Typ übernommen werden. Letzteres kann bei Begrenzern zu Modifikationen an der Baumstruktur führen (Abschnitt 7.4.1.3). Dieser Vorgang wird auch als Anwenden bezeichnet.

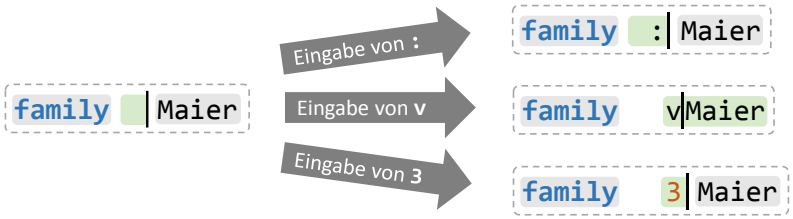

Abbildung 7-17: Mögliche Resultate bei der Eingabe eines Zeichens zwischen zwei Tokens

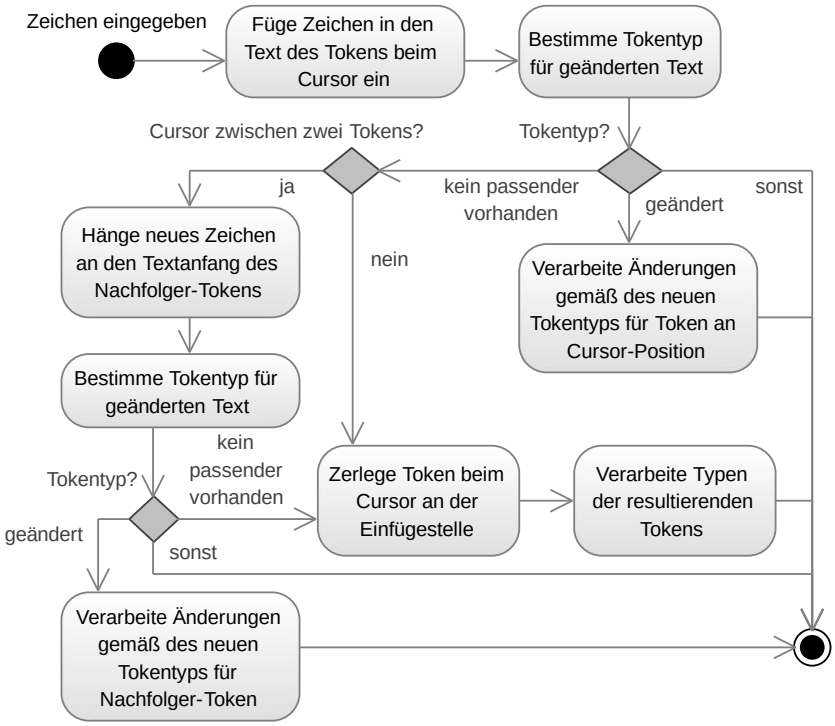

Abbildung 7-18: Ablauf der lexikalischen Analyse

Wird kein passender Tokentyp gefunden, dann wird geprüft, ob sich der Text-Cursor zwischen zwei angrenzenden Tokens befindet. Ist das *nicht* der Fall, wird das Token an der Einfügestelle aufgespaltet. Die einzelnen Möglichkeiten sind oben anhand von Beispielen erläutert (Abbildung 7-16). Im Anschluss muss der Typ eines jeden resultierenden Tokens bestimmt und entsprechend angewendet werden.

Liegt der Cursor jedoch zwischen zwei benachbarten Tokens, dann wird das eingegebene Zeichen an den Textanfang des Nachfolger-Tokens angehängt und daraufhin dessen Token-Typ bestimmt. Kann ein Typ identifiziert werden, so wird er (falls geändert) übernommen und entsprechend angewendet. Gibt es keinen passenden Tokentyp, wird mit dem ursprünglichen Token (dasjenige direkt vor dem Text-Cursor) auf dieselbe Weise verfahren, als läge der Cursor innerhalb dieses Tokens.

7.4.1.1 Selektieren, Löschen und Überschreiben von Text

Das Selektieren von Text steht in engem Zusammenhang mit dem Löschen und Überschreiben desselbigen. Bei projizierenden Editoren gilt es hier wiederum zu beachten, dass nicht auf dem Text selbst, sondern auf dem zugrundeliegenden CST gearbeitet wird.

Sofern **Text zeichenweise entfernt** wird (z.B. durch Betätigen der Rücktaste, ohne dass ein Textfragment ausgewählt ist), kann noch auf dieselbe Vorgehensweise zurückgegriffen werden wie beim zeichenweise Einfügen. Auch wenn Zeichen in lediglich einem Token selektiert sind, können diese einfach gelöscht oder überschrieben werden. Beim reinen Löschen wird als Eingabezeichen einfach ein leeres Zeichen angenommen, also ein Zeichen, dessen Eingabe den Text unverändert lässt. Es ist dadurch vergleichbar mit dem leeren Wort (meistens symbolisiert durch ε) bei kontextfreien Grammatiken [17]. Werden alle Zeichen eines Tokens gelöscht, führt das gleichzeitig zum Entfernen des gesamten Tokens. Soll das Token stattdessen nur ausgeblendet werden, so ist dies explizit vom Benutzer zu spezifizieren (z.B. mithilfe eines geeigneten Tastenkürzels wie Strg+Entf).

Verläuft die **Textauswahl** jedoch **über Tokengrenzen hinweg**, dann müssen **immer ganze Teilbäume selektiert** sein (vergleichbar mit dem Editorverhalten bei MPS). Visuell betrachtet heißt das, dass

Abbildung 7-19: Beispiel für Möglichkeiten bei Container-übergreifendem Löschen

Container-Grenzen nicht einseitig überschritten werden dürfen. Innerhalb der Auswahl dürfen jedoch Containerknoten (vollständig enthalten) liegen. Diese werden zusammen mit den vollständig enthaltenen Tokens beim Überschreiben oder Löschen komplett entfernt. Dürfte die Selektion in einem Container beginnen oder enden und über seine Begrenzung hinausgehen, dann wäre dies oft mit einem Aufbrechen und gegebenenfalls neu Anlegen der Baumstruktur verbunden. Diese weitreichenden Folgen sind allerdings vom Benutzer nicht oder nur schwer abschätzbar.

Das in Abbildung 7-19 gezeigte Beispiel soll die Problematik demonstrieren. Dabei stellt der bläulich überdeckte Bereich die momentane Textauswahl dar, die im nächsten Schritt gelöscht wird. Wie ersichtlich ist, überspannt sie die Grenzen des inneren Blocks sowie des darin enthaltenen Statements. Die Frage, die sich nun stellt, ist, wie mit den beiden Containern verfahren werden soll. Drei mögliche Lösungen sind auf der rechten Seite abgebildet. In allen werden die beiden Tokens `Josef` und `name` zu dem Token `Josme` verschmolzen. Die erste Variante löst darüber hinaus die zwei überschnittenen Container auf und verschiebt die beinhalteten Tokens in das äußere Statement. Lösung 2 und 3 hingegen ähneln einander, da in beiden Fällen die Container erhalten bleiben. Allerdings wird einmal das `Josme`-Token dem äußeren Statement und beim anderen Mal dem inneren Statement zugeteilt.

Lediglich die erste Variante scheint semantisch sinnvoll zu sein, was allerdings am gewählten Beispiel liegt. Dies ist aber stark abhängig vom jeweiligen Anwendungsfall. Nachdem im Beispiel lediglich zwei Containergrenzen überschritten werden, sind die Lösungsmöglichkeiten noch überschaubar. Mit steigender Anzahl dieser Grenzüberschreitungen wachsen auch die Zahl der Möglichkeiten sowie deren Komplexität. Deshalb ist es – wie oben bereits erwähnt – wenig sinnvoll solche Löschoperationen zu unterstützen, sondern ausschließlich das Löschen ganzer Teilbäume zu gestatten. So kann beispielsweise das durch das `name`-Token identifizierte Statement ausgewählt und entfernt werden. Analoges gilt auch für den Block, der dieses Statement enthält.

Eine **Ausnahme** bilden **Tokens** die sich **an Anfang und Ende der Auswahl** befinden. Sie brauchen nicht vollständig enthalten sein. Stattdessen darf die Selektion im jeweiligen Text an beliebiger Stelle starten bzw. enden. Beim Löschen oder Überschreiben werden dann die beiden äußeren Tokens zu einem einzigen Token verschmolzen und alle dazwischenliegenden Knoten entfernt. Das Verschmelzen geschieht nach dem im übergeordneten Abschnitt 7.4.1 vorgestellten Prinzip.

7.4.1.2 Standardkonfiguration der Tokentypen

Standardkonfiguration meint die Parametrierung der einzelnen Instanzen des Konzepts Token, die standardmäßig in jedem neu angelegten Definitionsmodell zur Verfügung stehen. Es handelt sich hierbei um eine Art Super-Grammatik, die auf Best Practices und unseren Erfahrungen im Zusammenhang mit textuellen DSMLs beruht. Den Anfang machen alle Tokentypen, die keinen Begrenzer darstellen.

- *Bezeichner*: Hierzu gibt es genau eine Instanz des Konzepts `Identifier`, deren regulärer Ausdruck demjenigen von Bezeichnern der Java Spezifikation [53] entspricht.

- *Verweise*: Hierzu gibt es genau eine Instanz des Konzepts `Referral`, wobei als Trennsymbol der Punkt angegeben ist. Die Verwaltung aller referenzierbaren und damit über einen Bezeichner verfügenden Statements geschieht extern. Gleiches trifft auch für die Berechnung und Auflösung der qualifizierenden Namen zur Vermeidung von Namensüberdeckungen (Abschnitt 7.2.4.1). Das gegebenenfalls notwendige Importieren anderer Modelle erfolgt mit Mitteln des LMM (Abschnitt 2.2.2).
- *Schlüsselwörter*: Bei einer neuen Sprache ist noch kein Schlüsselwort registriert. Sie müssen explizit vom Benutzer spezifiziert werden, wobei es sich wegen der Steuerung mittels Tastatur empfiehlt auf Tastenkürzel zurückzugreifen (z.B. Strg+K). Dabei handelt es sich schon um eine Beeinflussung des Standardverhaltens, wie sie in Abschnitt 7.4.1.3 näher beschrieben wird. Gespeichert werden alle Schlüsselwörter in einer Instanz des Konzepts `Keyword`, von der es pro Sprache im Regelfall genau eine gibt. Mehrere sind aber durchaus zulässig, sofern dies ein Szenario erfordert.
- *Strings*: Hierfür gibt es zwei Instanzen des Konzepts `WrapLiteral`, wobei eine für `startSymbol` und `endSymbol` das Zeichen `"` und die andere das Zeichen `'` festlegt. Sowohl Start- als auch Endsymbol gehören immer vollständig zum jeweiligen String-Token. Wird Text zwischen diesen Symbolen eingegeben, dann ist er Teil des Tokens. Andernfalls wird der Text in ein anderes Token verlagert. Wird ein gültiges Startsymbol eingegeben, dann wird automatisch das zugehörige Endsymbol ergänzt und der Text-Cursor zwischen beiden Symbolen positioniert.
- *Zahlen*: Hierfür gibt es zwei Instanzen des Konzepts `RegexLiteral`. Eine Instanz ist für ganze Zahlen (mit `typeToMap = Integer`) und die andere für Fließkommazahlen (mit `typeToMap = Float`) zuständig. Wichtig dabei ist, dass die Instanz für ganze Zahlen stets vor der für Fließkommazahlen ausgewertet wird, weil der reguläre Ausdruck für letztere auch auf ganze Zahlen passt. Schließlich kann jede ganze Zahl auch als Fließkommazahl gelten. Aus Gründen der Lesbarkeit werden entsprechend typisierte Tokens nachfolgend auch als Zahl-Tokens bezeichnet.
- *Kommentare*: Standardmäßig werden die beiden Kommentararten von Java und JavaScript unterstützt. Dazu gibt es zwei Instanzen des Konzepts Comment. Die erste davon hat als `startSymbol` den Wert `/*` und als `endSymbol` den Wert `*/` (mehrzeiliger Kommentar). Die zweite hingegen hat als `startSymbol` den Wert `//` und als `endSymbol` das Zeichen für einen Zeilenumbruch (einzeiliger Kommentar). Mit Text, der vor, zwischen oder hinter den Symbolen eingetippt wird, wird analog zu den Strings (siehe oben) verfahren. Gleiches trifft auch für die automatische Ergänzung des Endsymbols in Verbindung mit der Positionierung des Text-Cursors zu.

Aufgrund ihres engen Bezugs zu den verschiedenen Containertypen erfahren Begrenzer bereits in Abschnitt 7.2.1 eine Sonderbehandlung. Sie werden deshalb auch hier separat gelistet. Um Wiederholungen zu vermeiden wird hier schon erwähnt, dass jede Begrenzer-Spezialisierung genaue einmal pro Definitionsmodell instanziiert wird (ausgenommen Operatoren) und auch stets `ignoreWhitespaces` auf `true` gesetzt ist.

- *Token-Separatoren*: Sie nehmen eine Sonderstellung ein, da jedes Symbol, das nicht explizit von einem anderen Tokentyp reserviert wird, standardmäßig als Token-Separator typisiert wird. Symbole, die an das `terms`-Attribut von `TokenSeparator` zugewiesen werden,

stellen deshalb echte Alternativen dar, sofern sie auch dem `terms`-Attribut eines anderen Begrenzer-Subtyps zugewiesen sind. Im Falle einer Mehrdeutigkeit muss der Benutzer somit jedes Mal eine der möglichen Optionen wählen.

- *Statement-Separatoren*: Hierzu zählen die Symbole Komma, Semikolon und Zeilenumbruch.
- *Block-Einleitung*: Hierzu zählen alle öffnenden Klammern, also `(`, `[` und `{`. Eine Sonderstellung nimmt der Zeilenumbruch mit direkt nachgeschaltetem Tabulatorzeichen ein. Im Gegensatz zu den Klammern gibt es dazu nämlich kein Abschlusssymbol (umsetzbar als leere Zeichenkette, die nicht zur Erzeugung eines abschließenden Tokens führt). Diese Zeichenkombination zählt aber ebenso zu den Block-Einleitungen.
- *Binäre Operatoren*: Um das vom Benutzer erwartete Verhalten zu erreichen, ist es an der Stelle wichtig, die übliche Operatorpräzedenz (unter anderem Punkt vor Strich) zu beachten. Deshalb gibt es für jede Präzedenzklasse eine eigene Instanz von `BinaryOperator`. Nach absteigender Präzedenz geordnet gibt es standardmäßig folgende, durch Klammern gekennzeichnete Instanzen: (`*` `/` `%`), (`+` `-`), (`<` `<=` `>` `>=`) sowie (`==` `!=`).
- *Unäre Operatoren*: Sie weisen eine höhere Präzedenz als alle binären Operatoren auf. Differenziert wird hierbei zwischen Präfix- und Postfixoperatoren. Für jede dieser Klassen existiert eine separate Instanz der Konzepte `PrefixOperator` bzw. `PostfixOperator`. Als klassische Postfixoperatoren sind die Symbole `++` und `--` deklariert. Gleichzeitig ist `ignoreWhitespaces` auf `false` gesetzt, weil diese beiden Symbole in allen bekannten Fällen direkt am jeweiligen Operanden anhaften. Als Präfixoperatoren kommen dieselben Symbole sowie `!"` und `-` zum Einsatz. Letzteres ist ebenfalls Teil des regulären Ausdrucks für Zahlen, weshalb das Minuszeichen vor einer Zahl dieser zugerechnet und nicht als Präfixoperator eingestuft wird. Mit welchem der beiden Operatortypen ein `++`- oder `--`-Token typisiert wird, hängt von dessen Position innerhalb des umgebenden Statements ab. Befindet es sich am Anfang, handelt es sich um einen Präfix-, andernfalls um einen Postfix-Operator.

7.4.1.3 Abweichungen vom Standardverhalten

Die im vorherigen Abschnitt vorgestellte Standardkonfiguration besitzt zwar den Zweck, in möglichst vielen Fällen direkt während der Eingabe die einzelnen Tokens intentionsgemäß zu typisieren. Doch das ist nicht immer möglich, weshalb der Benutzer den **Tokentyp** auch **explizit beeinflussen** können muss. Als geläufiges Beispiel gilt hier die Re-Typisierung eines Tokens als Schlüsselwort.

Daraus lässt sich eine andere Qualität der Einflussnahme ableiten, die positive Auswirkungen auf die Benutzererfahrung mit sich bringt. Hierbei handelt es sich um die **spontane Beeinflussung des Standardverhaltens**. Angewendet auf das vorherige Beispiel heißt dies, dass es bei einer expliziten Typisierung als Schlüsselwort zweckmäßig ist, den betreffenden Text im `terms`-Attribut der `Keyword`-Instanz zu hinterlegen. Bei einer späteren Eingabe desselben Texts für ein anderes Token, wird dieses Token automatisch als Schlüsselwort erkannt.

Letztendlich kann diese **Anpassbarkeit** für alle **Tokentypen** gewährleistet werden, die über ein `terms`-Attribut verfügen. Das trifft neben dem Schlüsselwort-Konzept auf sämtliche Begrenzertypen zu. Ist ein bestimmtes Symbol bereits einem Tokentyp zugeordnet und soll erneut einem anderen Tokentyp zugeteilt werden, so sollte der Benutzer zwischen Ersetzen und zusätzlich Vergeben auswählen können. Ist ein Symbol mehrfach vergeben, dann muss der Benutzer später manuell entscheiden, welchen Typ er im Einzelfall anwenden möchte.

Des Weiteren ist es hilfreich, die **automatisierte Typisierung** (und damit automatische Aufspaltung) **von Tokens kurzzeitig deaktivieren** zu können. Dadurch kann beliebiger Text innerhalb des aktuellen Tokens eingegeben werden und dieses im Anschluss manuell typisiert werden. Ausschließlich hilfreich ist dies wiederum für Tokentypen, die auf einer Menge festgelegter Symbole bzw. Begriffe basieren. Denn durch die anschließende Typisierung kann der Text des aktuellen Tokens im betreffenden `terms`-Attribut gespeichert werden. Ein Beispiel hierfür ist die Deklaration von += als Token-Separator zur Laufzeit. Standardmäßig würde nämlich + als binärer Operator und lediglich = für sich genommen als Token-Separator typisiert werden.

In manchen Fällen kann außerdem die Notwendigkeit vorherrschen, lediglich **lokal den Typ des jeweiligen Tokens explizit zu setzen**. Da es sich bei der Typisierung um eine Ausnahme handelt, soll die Parametrierung des Tokentyps davon allerdings unberührt bleiben. Der Typ des betreffenden Tokens wäre damit fix und könnte nur durch erneutes, aktives Festlegen geändert werden (d.h. die automatisierte Tokentyp-Erkennung wäre für das Token deaktiviert). Auf diese Weise kann z.B. ein Token, das standardmäßig als binärer Operator deklariert wurde, in einen Token-Separator umgewandelt werden, wodurch das umgebende Statement nicht länger als Teilausdruck verstanden wird. Dies bedingt gleichzeitig, dass die Operatorpräzedenz an der Stelle nicht mehr greift wie gewohnt (was aber je nach Anwendungsfall durchaus beabsichtigt sein kann).

Interessant ist dieses Feature auch im Rahmen von Blöcken und dabei hauptsächlich im Zusammenhang mit dem `Separators`-Konzept. Denn hierbei kann ein beliebiges Token als Statement-Separator typisiert werden, was dazu führt, dass seine Definition als eine Verwendungsmöglichkeit der zugehörigen `Separators`-Definition zugeordnet wird. So lässt sich beispielsweise das Leerzeichen als Statement-Separator innerhalb von Blöcken einer bestimmten Definition nutzen.

Nachdem die Arbeit mit textuellen DSMLs primär unter **Verwendung der Tastatur** stattfindet, ist es **für** den Benutzer von Vorteil, wenn er **sämtliche Anpassungen und Beeinflussungen** ebenfalls damit durchführen kann. Ein in der Praxis bewährtes Mittel sind sogenannte Tastenkürzel, die möglichst einprägsam sein sollten. Für die explizite Typisierung eines Tokens als Schlüsselwort empfiehlt es sich beispielsweise, dass das zugehörige Tastenkürzel die Taste K für „Keyword" enthält. Wie in modernen IDEs üblich, können die Tastenkürzel frei vom Benutzer konfiguriert und ebenso eigene erstellt werden.

Als letzte Token-bezogene Abweichung gilt das aktive **Erstellen eines neuen Tokens**. Standardmäßig wird ein auf diesem Weg erstelltes Token mit einem Leerzeichen als Text versehen, weil dessen Typisierung als Token-Separator allenfalls geringfügige Auswirkungen auf die lokale Baumstruktur hat und daher unproblematisch ist. Wird der Erstellvorgang am Ende eines Tokens initiiert, dann wird das neue Token direkt hinter jenem hinzugefügt. Befindet sich der Cursor mittig in einem Token, dann wird dieses an der Cursorposition aufgespalten und das neue Token dort platziert. Jedes der auf die Weise neu entstandenen Tokens wird im Anschluss automatisch typisiert, was unter Umständen zu Fehlern führen kann, die als solche angezeigt werden. Ein zu inkorrekten Ergebnissen führendes Beispiel ist die Aufspaltung eines String-Tokens. Der Benutzer steht in der Verantwortung, etwaig auftretende Fehler zu beheben (z.B. durch explizite Typisierung oder durch Editieren des Tokentexts). Die aktive Erstellung eines Tokens unterscheidet sich von der direkten Eingabe eines Leerzeichens also dadurch, dass die Erzeugung eines neuen Tokens erzwungen wird.

7.4.2 Syntaktische Analyse

Aufgabe der syntaktischen Analyse ist – analog zum Compilerbau – der **Auf- und Umbau der Baumstruktur des Dokuments**. Diese Struktur wird beim vorgestellten Ansatz während der Eingabe im Hintergrund weitgehend automatisiert generiert und modifiziert. Ermöglicht wird dies durch die in

Abschnitt 7.4.1 erwähnte Super-Grammatik, die für jedes visuelle Definitionsmodell gleichermaßen vorgegeben, aber dennoch konfigurierbar ist. Sie beruht auf der vertrauten Erwartung, die man in bestimmte Symbol- und Wortklassen hat.

Für die konkrete Durchführung der jeweiligen **Restrukturierungsmaßnahmen** sind die einzelnen **Tokentypen selbst verantwortlich**. Jeder Tokentyp darf dabei die lokale Baumstruktur nach Belieben (gemäß seiner Intention) manipulieren.

Zunächst erfolgt in den nachfolgenden Unterabschnitten die Diskussion der einzelnen Tokentypen hinsichtlich ihrer Auswirkungen auf den CST. Um jedoch den vorgestellten Ansatz auf ein noch größeres Spektrum potentieller Zielsprachen auszuweiten und somit eine Verallgemeinerung zu erreichen, folgen im Anschluss zwei weitere Abschnitte, die sich dieser Thematik widmen. Zum einen geht es dabei um eine Verzögerung der automatisiert durchgeführten Restrukturierungen. Zum anderen wird die Notwendigkeit für die Abweichung vom Standardverhalten durch die explizite Erstellung bestimmter Container erklärt.

7.4.2.1 Nicht-Begrenzer

Das Verhalten von Tokentypen, die keine Begrenzer darstellen (sogenannte **Nicht-Begrenzer**), ist im Hinblick auf den Umbau der lokalen Baumstruktur in weiten Teilen identisch. Grundsätzlich gilt nämlich, dass jedes derartige Token **innerhalb eines Statements oder einer Gruppe** liegen muss, um somit auf Definitionsseite Bestandteil eines visuellen Konstrukts zu sein. Wird ein Nicht-Begrenzer als direktes Kind in einem Block erkannt, dann wird er automatisch in ein eigenes Statement eingebettet, wodurch die besagte Bedingung erfüllt ist.

Ein Beispiel hierzu zeigt Abbildung 7-20. Darin wird direkt nach einem Statement-Separator (ein Komma), das demnach für die Trennung von Statements verantwortlich ist und innerhalb eines Blocks liegt, der Buchstabe „W" eingetippt. Dieser gehört nicht zum vorherigen Statement-Separator, weshalb ein neuer Bezeichner mit dem Text „W" angelegt wird. Gleichzeitig wird er in ein neues Statement eingebettet.

Einen **Sonderfall** nehmen **Kommentare** ein, da sie keinerlei Auswirkungen auf die einem Statement zugrundeliegenden Definition haben. Sie dürfen also zwischen beliebigen Tokens innerhalb von Statements und Gruppen liegen, ohne die Struktur des zugehörigen Konstrukts zu beeinflussen. Die Eingabe eines Kommentars direkt innerhalb eines Blocks wird infolgedessen ignoriert.

7.4.2.2 Token-Separator

Die Funktionsweise von Token-Separatoren entspricht derjenigen von Nicht-Begrenzern. Denn auch sie kommen nur in Statements und Gruppen vor, um andere Tokens direkt voneinander abzugrenzen. Wird ein Token-Separator innerhalb eines Blocks identifiziert, dann wird er gleichermaßen in ein eigenes Statement eingebettet.

7.4.2.3 Statement-Separator

Nachdem ein **Statement-Separator** stets zwei Statements voneinander abgrenzt und somit zwischen diesen platziert ist, liegt er immer innerhalb eines Blocks. Wird ein solcher Separator ganz **am Ende**

Abbildung 7-20: Beispiel für die Eingabe eines Bezeichner-Zeichens innerhalb eines Blocks

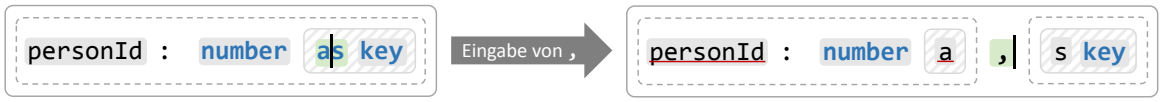

Abbildung 7-21: Beispiel für die Eingabe eines Statement-Separators innerhalb einer Gruppe

eines Statements erkannt, dann wird er in den nächsthöheren Block direkt hinter das aktuelle Statement verschoben. Dasselbe trifft ebenso zu, wenn die Eingabe am Ende einer Gruppe erfolgt und diese Gruppe am rechten äußeren Rand des Statements liegt.

Wird ein **Statement-Separator** allerdings an anderer Stelle **innerhalb eines Statements** oder einer Gruppe eingegeben, dann werden alle Container die Baumhierarchie hinauf bis zum nächsten Block an dieser Stelle aufgetrennt. Die Aufspaltung eines Statements (nämlich dasjenige, welches direktes Kind des besagten Blocks ist) in zwei Teile ist damit unumgänglich. Dazwischen wird der eingetippte Separator platziert. Das jeweils linke Statement bleibt dabei weiterhin mit der ursprünglichen Definition verknüpft, während für das rechte Statement ein neues Konstrukt erzeugt wird. Die ursprüngliche Definition wird folglich an die neue (reduzierte) Struktur des Statements angepasst.

Ein diesbezügliches Beispiel zeigt Abbildung 7-21. Darin wird zwischen den beiden Buchstaben des Schlüsselworts `as` ein Komma eingegeben, was zur Aufspaltung dieses Tokens, der umgebenden Gruppe und schließlich des äußeren Statements führt. Wie das Beispiel demonstriert, kann dies leicht zu fehlerhaften Statements führen. Denn durch das Zerlegen umfasst das erste Statement nun fälschlicherweise zwei Bezeichner, was laut Abschnitt 7.2.2 untersagt ist. Derartige Fehler sind nicht vermeidbar, weshalb sie der Benutzer manuell beheben muss.

Bei Statement-Separatoren handelt es sich jedoch nicht um semantische Entitäten, weshalb sie von einer speziellen Enthaltensein-Beziehung (Konzept `Separators`) referenziert werden müssen. Details dazu wurden bereits in Abschnitt 7.2.2 erläutert. Damit können die für textuelle Sprachen typischen Aufzählungen realisiert werden, bei denen verschiedene Elemente aufgelistet und meist durch ein bestimmtes Symbol (z.B. Komma, Zeilenumbruch, Semikolon) voneinander getrennt sind.

Bei der Anzeige im Dokument erfolgt eine Mischung aus semantischer Enthaltensein-Beziehung und der speziellen Beziehung für Separatoren, wobei sich die jeweils referenzierten Knoten stets abwechseln. Zusätzlich gilt, dass sich am Anfang und am Ende der Auflistung immer ein semantischer Knoten befindet. Demzufolge ist es **nicht möglich, zwei Separatoren direkt hintereinander anzuzeigen** (was inhaltlich betrachtet ohnehin keinen Sinn ergibt). Wird trotzdem direkt nach einem Statement-Separator ein weiterer derartiger Separator erkannt, dann wird letzterer einfach ignoriert und das entsprechende Token verworfen. Würde also im Beispiel aus Abbildung 7-21 direkt nach dem Komma ein weiteres Komma eingegeben werden, so hätte dies keinerlei Auswirkungen auf das Dokument.

7.4.2.4 Block-Einleitung

Block-Einleitungen werden ausschließlich innerhalb von Statements und Gruppen akzeptiert, weil nur darin Blöcke zulässig sind. An anderer Stelle eingegeben werden sie in ein neues (oder falls vorhanden in das direkt nachfolgende) Statement verfrachtet, bevor weitere Umbaumaßnahmen stattfinden. Handelt es sich bei einer Block-Einleitung um eine öffnende Klammer, dann wird nach dem Block außerdem die **zugehörige schließende Klammer angehängt**. Dies entspricht dem bekannten Verhalten moderner IDEs sowie für die Programmierung konzipierter Texteditoren.

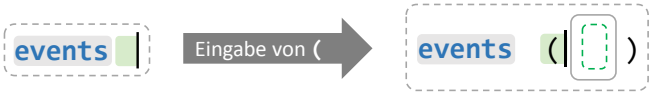

Abbildung 7-22: Beispiel für Eingabe einer Block-Einleitung

Abbildung 7-23: Beispiel für Eingabe einer Block-Einleitung bei selektiertem Token

Ein diesbezügliches Beispiel veranschaulicht Abbildung 7-22. Die Eingabe der öffnenden Klammer führt dabei zur Erzeugung eines Blocks mit eingeschachteltem, leeren Statement sowie einer darauffolgenden schließenden Klammer. Das Statement erhält zudem den Fokus (symbolisiert durch die grüne Linienfarbe), um für kommende Eingaben als Ziel zu dienen. Beim Eintippen eines weiteren Zeichens landet dieses somit nicht direkt nach der öffnenden Klammer im äußeren Statement, sondern stattdessen in einem neuen Token innerhalb des grün umrandeten Containers. Eine Erläuterung, wie Container aktiv selektiert werden können, findet sich in Abschnitt 7.4.3 zum Thema Navigation.

Darüber hinaus ist es im Hinblick auf eine komfortable Bedienbarkeit hilfreich, wenn die Unterstützung für Block-Einleitungen noch etwas tiefer geht. So werden alle **selektierten Knoten** in einen neuen Block und darin in ein neues Statement **eingebettet**, sofern eine Block-Einleitung eingetippt wird. Das Symbol an sich wird dann direkt davor (und falls vorhanden der im `closingTerms`-Attribut angegebene, zugehörige Gegenpart direkt dahinter) platziert. Zu beachten ist hierbei lediglich, dass die Auswahl stets alle überspannten Knoten in ihrer Gesamtheit enthalten muss. Ein Beispiel dazu zeigt Abbildung 7-23. Dort wurde fälschlicherweise der Bezeichner START direkt in das Statement für sämtliche Ereignisse aufgenommen. Durch dessen Selektion und anschließendes Eintippen einer Block-Einleitung (hier die runde öffnende Klammer) wird das START-Token in ein eigenes Statement mit umgebenden Block verfrachtet. Zusätzlich wird vor bzw. nach dem Block eine öffnende bzw. eine schließende Klammer eingefügt.

7.4.2.5 Binärer Operator

Binäre Operatoren sind die Begrenzer, welche die lokale Baumstruktur am stärksten beeinflussen. Geschuldet ist dies dem semantischen Hintergrund, der bei solchen Operatoren schon während der freien Modellierung berücksichtigt wird. Vorrangig zählt dazu die Operatorpräzedenz, die recht spezifisch ist und in einem Großteil der existierenden Sprachen auf die gleiche Weise umgesetzt wird. Die üblicherweise zum Einsatz kommenden Strategie stellt Bornat in [15] vor. Eine Übertragung auf projizierende Editoren findet sich in JetBrains MPS mit den sogenannte „side transforms" [139].

Zusätzlich dazu müssen **bei der freien Modellierung auch unvollständige Ausdrücke** verarbeitet werden können, die bei der Eingabe binärer Operatoren unweigerlich auftreten. Dazu zeigt Abbildung 7-24 die einzelnen Phasen während der Eingabe des Ausdrucks `4 + 6 * 3`. Den Ausgangspunkt bildet bereits ein Statement mit der Zahl `4` als separates Token, wobei das Statement in einem eigenen Block sitzt. Aus Verständlichkeitsgründen ist in allen Beispielgrafiken dieses Abschnitts die tatsächliche und nicht die vereinfachte Baumstruktur abgebildet

Wird nun der +-Operator direkt im Anschluss an die Zahl `4` eingetippt, so liegt ein unvollständiger Ausdruck vor, weil der zweite Operand fehlt. Dennoch wird die lokale Baumstruktur für die Darstellung eines Ausdrucks vorbereitet, wodurch alle Tokens des bisherigen Statements (hier nur die Zahl `4`) in einen neuen Block und darin in ein neues Statement eingefügt werden. Es ist wichtig, dass die Zahl `4` zuvor in einem eigenen Statement gelegen hat, da andernfalls diese Zahl zusammen mit dem `calc`-Token als erster Operand aufgefasst worden wäre. Bewerkstelligt werden kann dies durch die explizite Erstellung eines Containers (Abschnitt 7.4.2.9).

Nach jedem binären Operator sollte intentionsgemäß ein zweiter Operand folgen. Auch er ist als Block mit eingeschachteltem Statement zu realisieren. Unterschieden werden können hierbei drei Fälle.

- Dem binären Operator folgt im aktuellen Statement bereits ein solcher Operand, der von einem Block mit enthaltenem Statement repräsentiert wird. Ein Token, das direkt nach dem Operator erkannt wird, wird dann am Anfang dieses Statements hinzugefügt.
- Dem binären Operator folgen im aktuellen Statement beliebige andere Knoten. Dann werden diese Knoten in ein separates Statement verfrachtet und das Statement in einen Block eingebettet, der jetzt als zweiter Operand anstelle der besagten Knoten liegt.
- Der binäre Operator steht am Ende eines Statements. Jedes direkt im Anschluss an den Operator eingegebene Token wird in einen neuen Block und darin in ein neues Statement eingefügt. Dies entspricht den Schritten 2 und 4 in Abbildung 7-24, bei denen die Zahlen `6` bzw. `3` eingetippt werden.

Als im dritten Schritt der binäre Operator `*` als solcher erkannt wird, greift zum ersten Mal sichtbar die Operatorpräzedenz. Der Operator bezieht sich nämlich lediglich auf die Zahl `6` und nicht auf den kompletten vorherigen Ausdruck `4 + 6`. Bei der **Berücksichtigung der Operatorpräzedenz** handelt es sich um ein **Standardproblem** der Informatik, zu dem auch eine Standardlösung existiert. Demnach wird sie hier nicht im Detail vorgestellt, sondern stattdessen mit [15] erneut darauf verwiesen.

Zu ergänzen ist lediglich, woran dieses Verfahren die **Grenzen eines Ausdrucks erkennt**. Bei diesen Überlegungen sind neben den binären auch die unären Operatoren zu beachten, weil sie ebenfalls mit der Operatorpräzedenz wechselwirken. Das Verfahren trifft hinsichtlich möglicher **Teilausdrücke**, die immer von einem Statement repräsentiert werden, folgende Annahmen.

- Ein *binärer Teilausdruck* besteht aus mindestens einem linksseitigen Operanden in Form eines Blocks mit eingeschachteltem Statement und einem binären Operator. Zusätzlich darf lediglich ein weiterer rechtsseitiger Operand vorkommen, der ebenfalls von einem Block repräsentiert wird.
- Ein *Präfix-Teilausdruck* muss an erster Stelle lediglich einen Präfix-Operator enthalten. Um vollständig zu sein, sollte diesem Operator jedoch ein Operand in Form eines Blocks mit eingeschachteltem Statement folgen.
- Ein *Postfix-Teilausdruck* besteht aus einem Operanden in Form eines Blocks mit eingeschachteltem Statement gefolgt von einem Postfix-Operator.

Das obige Beispiel umfasst ausschließlich binäre Operatoren, weshalb lediglich binäre Teilausdrücke vorkommen. Jedes Statement in Abbildung 7-24, das sowohl ein Additions- als auch ein Multiplikationssymbol enthält, stellt einen gültigen binären Teilausdruck dar. Wird nun ein Token als binärer Operator identifiziert, dann muss die Baumstruktur in beide Richtungen rekursiv solange abgelaufen

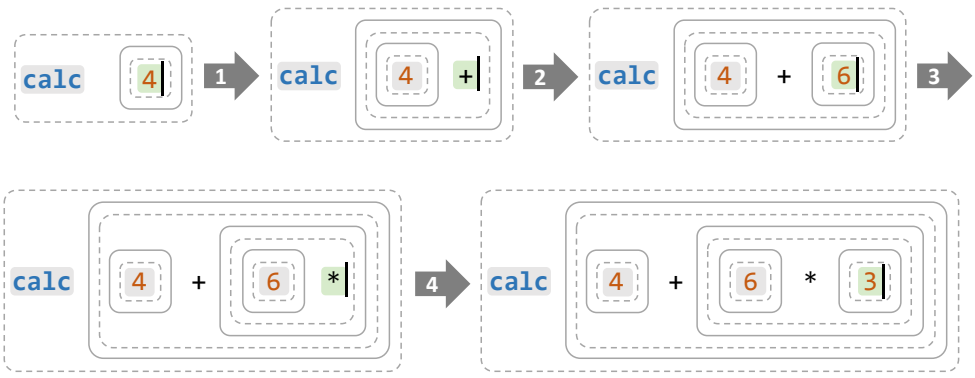

Abbildung 7-24: Beispiel für die Eingabe eines Ausdrucks mit zwei binären Operatoren

werden, bis ein Statement entdeckt wird, das die Teilausdruck-Bedingung nicht erfüllt. Die dabei durchwanderten Knoten ergeben schließlich einen im Sinne des Verfahrens gültigen Ausdruck, auf den die Präzedenzanalyse anzuwenden ist. Die originale Baumstruktur ist anschließend durch die gegebenenfalls umgebaute Struktur zu ersetzen.

Im Beispiel verdeutlicht werden soll dies an Schritt 3, bei dem das `*`-Symbol eingetippt und als binärer Operator erkannt wird. Dies führt direkt zu der dargestellten Struktur. Ob diese gemäß der Operatorpräzedenz korrekt ist, muss allerdings erst geprüft werden. Dazu wird der linke Teilbaum des Statements mit dem `*`-Token untersucht, wobei festgestellt wird, dass kein weiterer Unterausdruck vorliegt. Einen rechten Teilbaum gibt es nicht, weshalb direkt das nächsthöhere Eltern-Statement analysiert wird. Hier wird wegen dem +-Operator und dem davon links liegenden Block ein übergeordneter Teilausdruck erkannt. Nachdem das nächsthöhere Eltern-Statement (eingeleitet mit dem `calc`-Schlüsselwort) keinen Ausdruck repräsentiert, terminiert das Verfahren mit einem Teilbaum, dessen Wurzel das Statement mit +-Operator ist. Die Blätter sind die beiden Operator-Tokens sowie die Blöcke, die die drei in Statements eingefassten Zahlen umgeben.

Die Ersetzung des Multiplikationssymboles durch ein Minuszeichen hätte beispielsweise einen Umbau zur Folge. Damit wären beide Operatoren gleichwertig und der Ausdruck müsste von links nach rechts ausgewertet werden. Die resultierende Baumstruktur ist in Abbildung 7-25 veranschaulicht. Gleichzeitig sind die für die Beachtung der Präzedenz relevanten Knoten grün hervorgehoben.

Als **Sonderfall** zu betrachten ist die Eingabe eines **binären Operators direkt am Anfang eines Statements**. Nachdem damit seine Intention verletzt ist, wird er einfach als Token-Separator aufgefasst, was keine weiteren Restrukturierungsmaßnahmen bewirkt. Er wird erst dann zu einem binären Operator, wenn davor mindestens ein anderer Knoten platziert wird.

Wird ein binärer Operator direkt nach einem Statement-Separator eingetippt, dann wird er – wie auch Nicht-Begrenzer (Abschnitt 7.4.2.1) – in einem neuen bzw. (falls vorhanden) im nachfolgenden Statement abgelegt. Gemäß dem zuvor beschriebenen Sonderfall wird das betreffende Token in einen Token-Separator konvertiert.

Die hinter **Operatoren** stehende Intention liefert bereits ein **gewisses Maß an Domänenwissen**. So werden die beiden Operanden von binären Operatoren üblicherweise als linker und rechter Operand bezeichnet. Dieses Wissen wird mithilfe von Benennungs-Annotationen an die betreffenden Enthaltensein-Beziehungen übertragen, wodurch die später abgeleiteten Attribute entsprechend benannt werden (z.B. als „leftOperand“ und „rightOperand“).

Dasselbe Prinzip lässt sich potentiell auch für die Namen von Konstrukten einsetzen, die Ausdrücke repräsentieren. Allerdings ergibt sich hier der Name aus dem jeweiligen Operatorsymbol. Diese Symbole können jedoch bei Bedarf beliebig gesetzt werden, weshalb für jedes Symbol im visuellen Definitionsmodell anzugeben wäre, mit welchem Ausdrucksnamen es assoziiert ist. Eine mögliche Lösung wäre es, ein zusätzliches Attribut im Operator-Konzept zu deklarieren, über das die Namen für die jeweiligen Operatorsymbole festgelegt werden können. Aufgrund der Spezifität wurde darauf aber verzichtet.

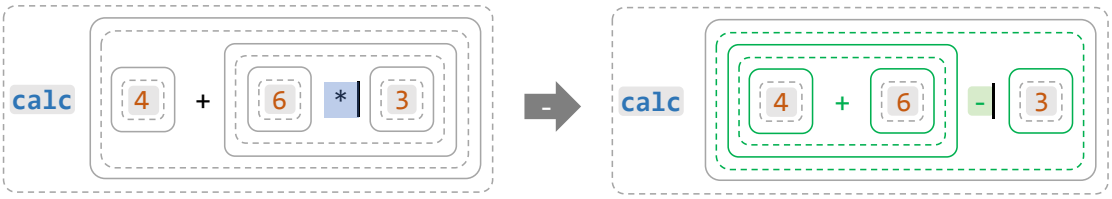

Abbildung 7-25: Auswirkungen der Operatorpräzedenz anhand eines Beispiels

7.4.2.6 Präfix-Operator

Damit ein Statement einen validen Präfix-Teilausdruck darstellt, muss es als erstes Token einen Präfix-Operator beinhalten. Demgemäß wird ein Präfix-Operator nur dann als solcher erkannt, wenn er am Anfang eines Statements steht. Ein sich anschließender Block mit eingeschachteltem Statement als Operand ist zwar optional, im Sinne der Intention eines Präfix-Ausdrucks allerdings erstrebenswert. Die Erkennung und Festlegung des Operanden erfolgt auf die gleiche Weise, wie die des zweiten Operanden bei binären Ausdrücken (Abschnitt 7.4.2.5). Es können somit auch dieselben drei Fälle differenziert werden. Für die automatisierte Benennung des Operanden empfiehlt sich „operand" zu verwenden (mittels Benennungs-Annotation, Abschnitt 7.4.2.5).

7.4.2.7 Postfix-Operator

Postfix-Teilausdrücke müssen stets vollständig vorliegen. Demnach muss ein entsprechendes Statement an erster Stelle einen Operanden in Form eines Blocks mit eingeschachteltem Statement und an zweiter Stelle einen Postfix-Operator enthalten. Ist kein solcher Block vorhanden, dann wird er auf die gleiche Weise wie bei der Eingabe eines binären Operators (Abschnitt 7.4.2.5) erzeugt. Alle direkt nach einem Postfix-Operator eingefügten Tokens führen dazu, dass der Postfix-Operator in einen Token-Separator transformiert wird. Weitere Tokens müssen folglich explizit außerhalb des Statements, das den Postfix-Ausdruck repräsentiert, platziert werden. Auch hier empfiehlt sich für die automatisierte Benennung des Operanden „operand" zu verwenden (mittels Benennungs-Annotation, siehe ebenfalls Abschnitt 7.4.2.5).

7.4.2.8 Verzögerung der Restrukturierung des CST

Weil bestimmte Zeichen in verschiedenen Tokentypen mehrfach vergeben werden dürfen, kann ein **sofortiger Umbau der lokalen Baumstruktur** direkt nach der Eingabe **unerwünschte Ergebnisse** zur Folge haben. Dazu sei beispielhaft angenommen, dass das Symbol += als Token-Separator und + als binärer Operator deklariert sind. Würde nun sofort nach der Eingabe von + der automatische Umbau der Baumstruktur angestoßen werden, dann hätte dies zur Folge, dass ein direkt im Anschluss eingegebenes = als Bestandteil des zweiten Operanden interpretiert würde (wie das im Einzelnen geschieht, wird in Unterabschnitt 7.4.2.4 erklärt). Die Erkennung von += als Token-Separator wäre auf die Weise unmöglich.

Deshalb muss die **Restrukturierung verzögert** durchgeführt werden. Betroffen hiervon sind sämtliche Begrenzertypen, weil ausschließlich sie aufwändige Umbaumaßnahmen bedingen (können). Eine Lösung zur Umsetzung der Verzögerung ist es, so lange mit der Restrukturierung zu warten, bis genau ein Begrenzertyp übrig bleibt, von dessen Symbolen mindestens eines mit dem aktuell eingegebenen Text beginnt. Auf diese Weise kann somit += automatisiert als Token-Separator erkannt werden (vorausgesetzt dieses Symbol ist als solcher deklariert, wie im vorherigen Absatz angenommen).

Diese verzögerte Restrukturierung ist **eng verwandt mit der kurzzeitigen Deaktivierung der automatischen Tokentypisierung** aus Abschnitt 7.4.1.3. Denn erst während der Typisierung wird festgestellt, welcher Tokentyp zum soeben eintippten Text passt, was letztlich zur Restrukturierung der lokalen Baumstruktur führen kann. Ohne sie bleibt auch der CST unangetastet.

7.4.2.9 Explizites Erstellen von Containern

Im Gegensatz zu Token-bezogenen Abweichungen vom Standardverhalten geht es bei Containern ausschließlich um deren explizite Erstellung. Die potentiellen Strukturierungsmöglichkeiten sind derart vielfältig, dass der Benutzer die **Struktur fein granular und explizit beeinflussen** können muss.

Für Gruppen ist dies sogar **zwingend erforderlich,** weil es hierzu keine einleitenden Begrenzer gibt (Abschnitt 7.2.2) und sie somit nie automatisiert entstehen könnten. Eine neue Gruppe kann einerseits an der aktuellen Cursorposition eingefügt werden. Sind jedoch einer oder mehrere Knoten innerhalb eines Statements oder einer bereits existierenden Gruppe selektiert, so kann andererseits eine neue Gruppe an der Stelle hinzugefügt werden, wobei alle zuvor ausgewählten Knoten ihr untergeordnet werden.

Auf ähnliche Weise funktioniert es auch für **Blöcke**. Zusätzlich wird allerdings ein Statement erzeugt, das in den Block eingebettet ist. Im Fall von zuvor selektierten Knoten, werden diese alle dem Statement und nicht dem Block zugeteilt. Beispiele hierzu zeigen Abbildung 7-22 und Abbildung 7-23, wobei die umgebenden Klammern zu ignorieren sind.

Separieren von Statements ohne dazwischen liegenden Begrenzer ist wenig sinnvoll, da sie sonst bei deaktivierter Anzeige der Baumstruktur nicht voneinander unterschieden werden können. Demzufolge wird keine Möglichkeit geboten, explizit ein neues **Statement** einzufügen. Dies ist **nur auf indirektem Weg** durch Eingabe eines Statement-Separators möglich. Alternativ kann ein anders typisiertes Token explizit als ein solcher Separator deklariert werden, was denselben Effekt erzielt.

7.4.3 Navigation im Dokument

Während der Arbeit an einem Dokument wird bei projizierenden Editoren stets die zugrundeliegende Baumstruktur sowie deren Inhalt manipuliert und nicht – wie bei Freitexteditoren üblich – eine Zeichenkette, die den Text des Dokuments widerspiegelt. Deshalb ist es vor allem bei Dokumenten mit tiefen Hierarchien wichtig, angeben zu können, auf **welches Statement** sich **kommende Änderungen** beziehen.

Abbildung 7-26 zeigt hierzu ein Beispiel. Der Cursor befindet sich darin direkt hinter dem letzten Token des `name`-Statements von `Peter`. Wird dort nun ein Zeilenumbruch (und somit gemäß Abschnitt 7.4.1.2 ein Statement-Separator) eingegeben, dann wird das entsprechende neue Token im nächst höheren Block hinter diesem Statement hinzugefügt. Will der Benutzer stattdessen allerdings eine weitere Person spezifizieren, dann müsste das Zeilenumbruch-Token im Wurzelblock hinter dem `Peter`-Statement eingefügt werden.

Daher ist es neben der reinen Positionierung des Text-Cursors in einem die freie Modellierung unterstützenden, projizierenden Editor wichtig, dass auch das **Eltern-Statement explizit angegeben** werden kann, auf das sich die nächste Eingabe beziehen soll. Sinnvoll ist das unter anderem für Statements, innerhalb derer der Cursor liegt, sei es direkt oder innerhalb weiterer Statements. Bezieht er sich auf ein anderes als das direkt umgebende Statement, dann empfiehlt es sich jenes visuell hervorzuheben.

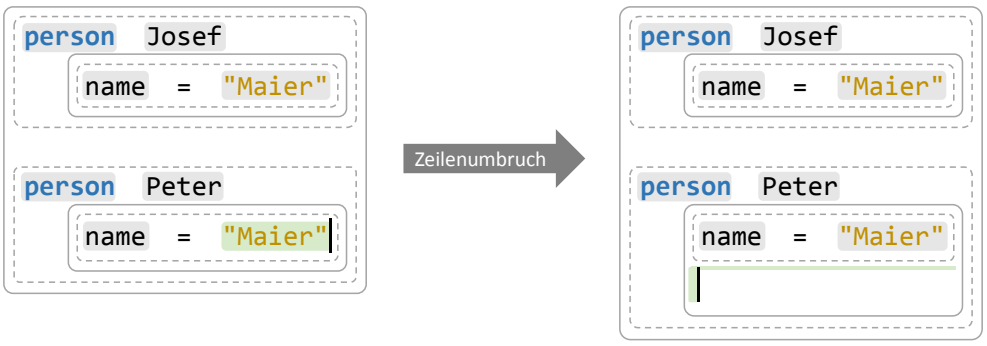

Abbildung 7-26: Eingabe eines Zeilenumbruchs im aktuellem Statement

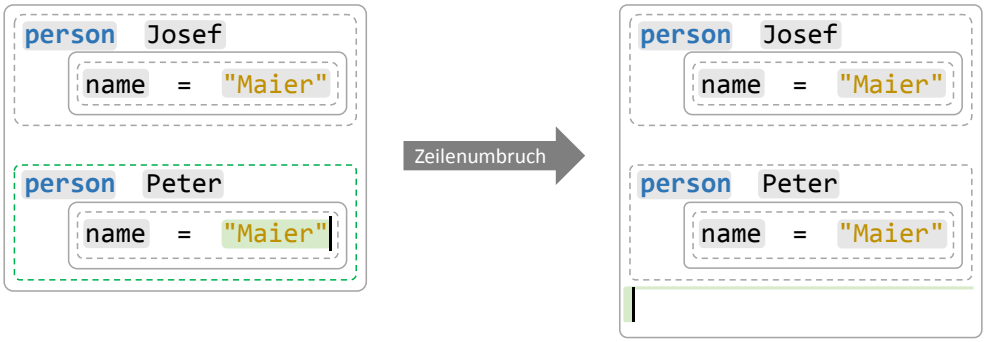

Abbildung 7-27: Eingabe eines Zeilenumbruchs im übergeordneten Statement

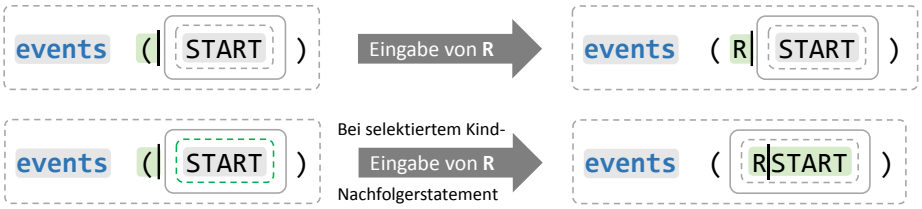

Abbildung 7-28: Reguläre Eingabe im Vergleich zur Beeinflussung des Kind-Statements im nachfolgenden Block

Wie dies aussehen kann ist beispielhaft in Abbildung 7-27 dargestellt, in der die Umrandung des `Peter`-Statements grün eingefärbt ist. Der eingegebene Zeilenumbruch wirkt sich nun auf den Wurzelblock aus, da er das besagte Statement direkt umgibt.

Ebenfalls nützlich ist es, die nächste **Eingabe** bei Bedarf explizit **an das nachfolgende, aber in einem anderen Statement liegenden Token anhängen** zu können. Das gilt unter anderem dann, wenn dieses Statement in einem Block liegt, der sich direkt an das momentane Token anschließt. Beispielhaft erkannt werden kann die Problematik an Abbildung 7-28. Die reguläre Eingabe von „R" führt nämlich (wie im oberen Bereich abgebildet) zur Erzeugung eines neuen Bezeichners innerhalb des `events`-Statements. Wird zuvor allerdings das sich im direkt anschließenden Block befindliche `START`-Statement als Ziel ausgewählt, dann wird „R" an den Anfang des `START`-Tokens hinzugefügt (unterer Teil der Abbildung).

7.5 Interaktion mit der konkreten Syntax

Die Interaktionsmöglichkeiten des Benutzers mit der konkreten Syntax sind – wie schon in Abschnitt 7.3 geäußert – ebenfalls dem textuellen Modellierungsworkflow zuzurechnen. Denn während der freien Modellierung ergeben das Anlegen, Manipulieren, Verwenden und gegebenenfalls auch das Löschen von Konstrukten die typischen Arbeitsschritte. Wie eine angemessene Unterstützung des Benutzers zur Erfüllung dieser Aufgaben aussehen kann, wird in den folgenden Unterabschnitten dargelegt.

In den vorausgegangenen Ausführungen waren mit Konstrukten stets solche gemeint, die sich noch in der Entwurfsphase befinden. Sie waren weder gemappt noch gab es zu ihnen jeweils mehr als eine Verwendung. Wird eine derartige Verwendung modifiziert, dann können die Änderungen faktisch eins zu eins auf die zugehörige Definition übertragen werden, ohne dass irgendwelche Restriktionen zu berücksichtigen sind (mit Ausnahme derjenigen aus Abschnitt 5.4.1). Die Synchronisation von Verwendung und Definition fällt somit trivial aus.

Anders verhält es sich mit **vollständig definierten textuellen Konstrukten**, auf deren Augenmerk sich dieser Abschnitt richtet. Ein Konstrukt gilt als vollständig definiert, wenn mindestens eine der folgenden Bedingungen erfüllt ist.

- Es gibt mindestens zwei Verwendungen des Konstrukts.
- Das Konstrukt verfügt über ein Mapping auf ein Element der abstrakten Syntax.

7.5.1 Benennung syntaktischer Artefakte

Im Verlauf der freien Modellierung kommen immer wieder neue Konstrukte hinzu. Um sie einerseits leicht von anderen Konstrukten unterscheiden zu können und andererseits ergänzendes Domänenwissen für das spätere Ableiten der abstrakten Syntax bereitzustellen, sollten diese Konstrukte möglichst zeitnah mit einem aussagekräftigen Namen versehen werden. Eine **Übersicht über alle bereits vorhandenen Konstrukte** inklusive Interaktionsmöglichkeiten ist deshalb **unerlässlich**. Beim grafischen Modellierungsworkflow kann hierfür auf die Palette zurückgegriffen werden (Abschnitt 6.2.2). Ein derartiges Hilfsmittel bietet sich somit auch zur Unterstützung des textuellen Modellierungsworkflows an. Um die Brücke zwischen beiden Welten zu schlagen, ist es zweckmäßig, wenn die textuelle Palette auf die gleiche Weise bedient werden kann.

Im Gegensatz zum grafischen Kontext ist in der textuellen Welt naturgemäß sehr viel **textuelles Wissen gegeben**. Es ist also zweckdienlich dieses Wissen bei der automatisierten Generierung einer initialen Konstruktbezeichnung einzubeziehen. Wie schon Heidenreich et al. [60] erkannt haben, eignen sich hierfür primär Schlüsselwörter, die innerhalb des Konstrukts Einsatz finden. Wichtig dabei ist, dass es sich um Schlüsselwörter handelt, die fest im Konstrukt verankert sind und somit in jeder zugehörigen Verwendung auf die gleiche Weise angezeigt werden. Als **Positivbeispiel** gilt hier Abbildung 7-27 mit den enthaltenen `person`-Statements. Diese Statements sind Verwendungen desselben Konstrukts und verfügen stets an erster Stelle über ein Schlüsselwort mit dem Text „person". Es liegt demnach nahe, das zugrundeliegende Konstrukt ebenfalls als „person" zu betiteln.

Ein **Negativbeispiel** hingegen sind Methodendeklarationen in der Programmiersprache Java. Listing 7-3 zeigt dazu eine einfache Klasse mit zwei derartigen Deklarationen in den Zeilen 2 und 3. Die erste Methode wird darin mit dem Schlüsselwort „public" und die zweite mit dem Schlüsselwort „protected" eingeleitet. Es ergibt also wenig Sinn eines der beiden Schlüsselwörter für die Bezeichnung des betreffenden Konstrukts heranzuziehen. Auch sonst existiert kein anderes, für die Benennung geeignetes Schlüsselwort. In einem solchen Fall kann deshalb dem Konstrukt auch weiterhin lediglich ein vollständig computergenerierter Name zugeordnet werden, da keine weiteren Domäneninformationen verfügbar sind. Ein späteres Ändern des Namens durch den Benutzer ist hier also sehr wahrscheinlich.

```
1 class User {
2     public String getName() { ... }
3     protected String getPassword() { ... }
4 }
```

Listing 7-3: Negativbeispiel für die automatisierte Benennung von Konstrukten

Enthält ein Konstrukt **mehrere fest verankerte** und somit **unveränderliche Schlüsselwörter**, dann sind verschiedene Strategien denkbar. Beispielsweise kann immer nur das erste derartige Schlüsselwort zur Benennung herangezogen werden. Eine andere Möglichkeit wäre es, alle Schlüsselwörter zu konkatenieren und das Resultat als Konstruktbezeichnung zu verwenden. Es empfiehlt sich daher, die Benennungsstrategie konfigurierbar zu machen.

7.5.2 Vervollständigungsvorschläge zur Verwendung existierender Konstrukte

Ein Feature, ohne das moderne IDEs nicht mehr vorstellbar wären, ist die **Unterbreitung von Vorschlägen zur Vervollständigung** oder Ergänzung **der momentanen Eingabe**. Hierbei kann der Benutzer aus einer Liste von Vorschlägen auswählen, woraufhin der ausgewählte Eintrag an der aktuellen Cursorposition eingefügt wird. Die Auswahl aus einer Menge von Vervollständigungsvorschlägen ist demnach gängige Praxis bei der täglichen Arbeit von Softwareentwicklern mit GPLs (und in zunehmendem Umfang auch mit DSMLs, wie unter anderem Xtext, Spoofax und MPS zeigen).

Es bietet sich also an, auf dieses Feature zu setzen, um bereits definierte Konstrukte im Dokument wiederzuverwenden. Ein Zugriff via Maus auf die im vorherigen Abschnitt erwähnte Palette ist damit nicht zwangsweise notwendig. Vervollständigungsvorschläge werden nur dann unterbreitet, **solange** der Benutzer noch kein anderes Konstrukt als Definition ausgewählt hat und sich somit das zugrundeliegende **Konstrukt** unweigerlich **in der Entwurfsphase** befindet.

Als Vervollständigungsvorschlag angeboten werden diejenigen Konstrukte, deren linksseitig enthaltene Knoten vollständig mit allen Knoten des soeben eingegebenen Konstrukts in Typ und Inhalt übereinstimmen. Da der Vergleich auf Definitionsebene stattfindet, ist nur der Inhalt rein visueller und damit statischer Artefakte von Relevanz. Weist ein Knoten im Vorschlagskonstrukt die Sichtbarkeitseigenschaft auf, dann darf der Knoten im eingetippten Konstrukt fehlen.

Beim **Übernehmen eines Vervollständigungsvorschlags** wird zunächst das zugehörige, im Entwurf befindliche Konstrukt gelöscht und alle Knoten-Verwendungen des Statements auf die Knoten des Vorschlagskonstrukts umgebogen. Daraufhin werden alle im Statement noch fehlenden Knoten als Verwendungen der betreffenden Definitionen des Vorschlagskonstrukts ergänzt. Sämtliche Knoten, deren Inhalt auf Verwendungsseite festzulegen ist (Bezeichner, Verweise, Literale, Blöcke und manche Schlüsselwort-Tokens), können als Platzhalter betrachtet werden, die mit Werten zu befüllen sind. Nicht verpflichtend ist dies für Blöcke sowie Knoten mit Sichtbarkeitseigenschaft.

Zum leichteren Verständnis zeigt Abbildung 7-29 ein kleines Beispiel, bei dem ein im Entwurf befindliches Konstrukt durch ein anderes, vollständig definiertes Konstrukt ersetzt wird. Die gestrichelten blauen Pfeile symbolisieren die einzelnen Typ-Verwendungsbeziehungen zwischen den Verwendungen (unten) und den Definitionen (oben). Im linken Bereich der Grafik ist der Ausgangszustand illustriert, bei dem zwei vollständig definierte Konstrukte (Zahl-Literal und Additionsausdruck) sowie ein im Entwurf befindliches Konstrukt (unvollständiger Additionsausdruck) vorliegen. Nachdem die linksseitige Struktur des Additionsausdrucks deckungsgleich mit dem des unvollständigen Additionsausdruck ist, wird vorgeschlagen letzteren durch den erstgenannten zu ersetzen. Diese Empfehlung wird akzeptiert, was sich im Dokument dadurch ausdrückt, dass nach dem Plus-Operator ein weiterer Block hinzugefügt wird. Für den Benutzer wirkt dies wie eine Vervollständigung seines Statements, weshalb der Begriff Vervollständigungsvorschlag gerechtfertigt ist, auch wenn tatsächlich im Hintergrund eine

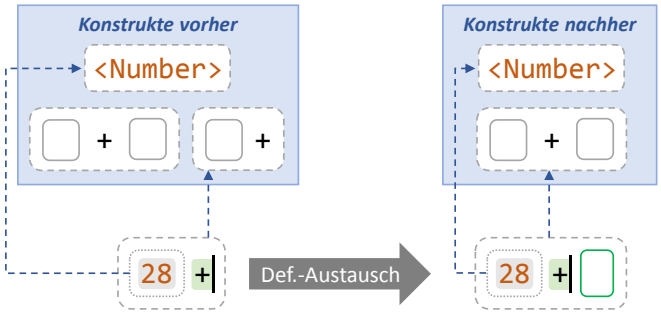

Abbildung 7-29: Beispiel für die Vervollständigung einer Konstruktverwendung

Ersetzung stattfindet. Außerdem wird der von der ursprünglichen Cursor-Position aus nächste Platzhalterknoten (hier der grün umrandete Block) für die kommende Eingabe selektiert. Abschließend erfolgt noch die Entfernung des unvollständigen Additionsausdrucks, der kurzfristig als separates Konstrukt Bestandteil der konkreten Syntax war.

Dem Intuitionismus ebenfalls zuträglich ist, wenn die vorgeschlagenen Statement-Definitionen eine **sinnvolle Sortierung** aufweisen. Deshalb sollten Konstrukte, die im selben Block zum Einsatz kommen, an der Spitze der Vorschlagsliste platziert werden. Danach sollten solche Statement-Definitionen folgen, die in Blöcken mit derselben zugrundeliegenden Block-Definition verwendet werden. Im Anschluss kommen alle übrigen Konstrukte. Die hierfür erforderlichen Informationen können Dank der Graphstruktur der Modelle mit geeigneten Anfragen direkt aus dem Modellrepositoriums (Abschnitt 2.2.2) ausgelesen werden.

Alternativ zur expliziten Verwendung eines existierenden Konstrukts kann der Benutzer auch einfach mit dem **Schreiben fortfahren** und dadurch die bereits vorhandene, sich noch in der Entwurfsphase befindende **Statement-Definition beibehalten**. Darüber hinaus bietet sich das Vervollständigungsfeature auch innerhalb von Tokens an, um Schlüsselwörter oder Verweise auf andere Statements zu ergänzen.

7.5.3 Manipulation textueller Konstrukte

Im Verlauf dieses Abschnitts werden diejenigen Änderungsoptionen erörtert, die die Gefahr von Konflikten zwischen den einzelnen Verwendungen und im Hinblick auf die zugrundliegende Definition (ein vollständig definiertes Konstrukt) in sich bergen. Für die kommenden Ausführungen ebenfalls von Bedeutung ist, dass **Modifikationen** nie an der Definition selbst stattfinden, sondern **immer erst an einer Verwendung durchgeführt** und daraufhin – sofern keine Unregelmäßigkeiten vorherrschen – an die zugrundeliegende Definition delegiert werden. Dadurch kann bei etwaig auftretenden Konflikten oder Mehrdeutigkeiten der Benutzer im Vorfeld eingreifen, was das Risiko versehentlicher, mit Informationsverlusten behafteter Umbauten senkt.

7.5.3.1 Hinzufügen weiterer Knoten

Das Hinzufügen eines neuen, rein visuellen Knotens zu einem vollständig definierten Konstrukt stellt eine der diversen Grundoperationen dar. Auf Tokenseite werden in dem Abschnitt jedoch nur Nicht-Begrenzer (ausgenommen Kommentare, weil die ohnehin nur lokal in der jeweiligen Verwendung existieren) und Token-Separatoren betrachtet, da diese keinen Einfluss auf die umgebende Baumstruktur haben. Die verbleibenden Begrenzertypen werden erst in Abschnitt 7.5.3.4 behandelt.

Prinzipiell darf jeder Nicht-Begrenzer und Token-Separator an beliebiger Stelle innerhalb eines Konstrukts eingefügt werden. Gleiches gilt für das Einfügen von Gruppen und Blöcken. Zusätzlich muss bei vollständig definierten Konstrukten beachtet werden, dass das **Hinzufügen rein visueller Knoten** sich ebenso **auf alle zugehörigen Verwendungen auswirkt**. Hierbei sind die einzelnen Tokentypen gemäß ihres Einflusses auf die anderen Verwendungen voneinander zu unterscheiden.

- *Schlüsselwörter* und *Token-Separatoren* werden auf Definitionsebene festgelegt, d.h. ihr Text wird zu den Verwendungen durchgereicht, sofern das Token nicht als optional deklariert wird. Bei Schlüsselwörtern kann es noch weitere Ausnahmen geben, doch die sind beim Hinzufügen nicht relevant.
- *Literale* werden erst innerhalb der Verwendungen mit Werten versehen. Für alle implizit neu generierten Token-Verwendungen sollte deshalb ein Standardwert gesetzt oder stattdessen

das Token als optional deklariert werden. Andernfalls führt die Änderung zu so vielen Fehler, wie es entsprechende Verwendungen gibt.

- Ähnliches trifft für *Verweise* zu, die ebenso erst in den Verwendungsmodellen konkretisiert werden. Allerdings ist es wenig sinnvoll einen Standardwert festzulegen, da es hierfür zusätzlich ein global referenzierbares Statement geben müsste, auf das dann alle implizit neu erzeugten Token-Verwendungen zeigen könnten.
- Auch *Bezeichner* werden erst auf Verwendungsebene mit einem Wert belegt, der schließlich das umgebende Statement benennt. Deshalb gilt es hierbei zusätzlich zu berücksichtigen, dass pro Statement maximal einer vorhanden sein darf. Treten trotzdem mehrere Bezeichner auf, so werden sie als fehlerhaft gekennzeichnet. Weder die Angabe eines Standardwertes noch die Deklaration als optional ist bei Bezeichnern möglich. Demzufolge ruft die Einführung eines neuen Bezeichners immer manuell zu behebende Fehler bei sämtlichen neu generierten Token-Verwendungen hervor.

Für das mehrfach angesprochene **Deklarieren eines hinzugefügten Tokens als optional** können verschiedene Strategien angewendet werden. Typischerweise bietet es sich hierfür an, auf die Sichtbarkeitseigenschaft zurückzugreifen. Je nach Tokentyp kann sie rein visueller oder semantischer Natur sein. Ersteres ist nur bei **Schlüsselwörtern und Begrenzern** möglich, da sie keinen direkten Bezug zur Domäne aufweisen. Der semantische Fall ist immer erlaubt. Stehen beide Varianten zur Verfügung, dann muss der Benutzer sich bewusst für eine entscheiden. Um im Nachhinein zwischen rein visueller und semantischer Sichtbarkeit differenzieren zu können, ist es ratsam die betreffenden Tokens entsprechend zu kennzeichnen (z.B. durch Hinterlegen mit unterschiedlichen Farben, wie bereits in Abschnitt 6.2.1 empfohlen).

Wegen des semantischen Charakters von **Literalen und Verweisen** muss auch die **Deklaration als optional** einen semantischen Hintergrund besitzen. Eine semantische Sichtbarkeitseigenschaft hätte beim späteren Ableiten jedoch zur Folge, dass für die betreffende Token-Definition zwei Attribute generiert würden, da neben der Sichtbarkeit ein weiteres semantisches Feature existiert. Um das Erzeugen zweier Attribute zu verhindern, wird statt der Verbergung von Knoten den anderen semantischen Feature-Verwendungen ein Null-Wert zugewiesen. Im Fall von Referenzen werden alle Zielangaben entfernt. Damit resultiert schließlich genau ein Attribut mit unterer Multiplizitätsgrenze gleich 0. Diese Lösung gilt daher für alle Knoten, die Eigentümer eines semantischen Features sind.

Allgemein wird die **Möglichkeit** geboten, auch im Nachhinein **Knoten auf Definitionsseite** und somit **standardmäßig** für alle darauf basierenden Verwendungen (sofern der Wert nicht bereits explizit überschrieben wurde) zu **verbergen**. Ausgenommen hiervon sind Knoten mit semantischen Features, da hier aktiv Werte überschrieben würden, was einen Verlust von Informationen nach sich zöge.

In projizierenden Editoren ist es wichtig, dass **ausgeblendete und leere Knoten bei Bedarf angezeigt** werden können. Andernfalls ist die Modifikation von deren Zustand und Inhalt nicht möglich. Eine schon in Abschnitt 6.2.2 erwähnte Strategie ist es, bei der Selektion eines Knotens (z.B. indem sich der Text-Cursor innerhalb seiner Grenzen befindet) seine rein visuellen, aber ausgeblendeten Kindknoten leicht transparent sichtbar zu machen.

Üblicherweise entstehen neue Tokens im Zuge der lexikalischen Analyse (Abschnitt 7.4.1), die nach jeder Zeicheneingabe lokal durchgeführt wird. Dies trifft auch weiterhin für vollständig definierte Konstrukte zu, was bedeutet, dass auch das **Modifizieren der Sprache durch zeichenweises Eintippen** von Buchstaben, Zahlen und sonstigen Symbolen erfolgt. Lediglich die Auswirkungen fallen unter Umständen etwas anders aus, und zwar in der Gestalt, wie in den kommenden Abschnitten beschrieben.

7.5.3.2 Automatisierte Anpassung von Tokens und deren Grenzen

Die Fülle möglicher, automatisiert durchgeführter Anpassungen, die aus einer zeichenweisen Eingabe resultieren, ist im Vergleich zu Konstrukten in der Entwurfsphase etwas limitiert. In erster Linie soll damit der Verlust von Informationen, die andere Verwendungen des betreffenden Konstrukts beinhalten, vermieden werden. Die Einschränkung drückt sich darin aus, dass der Typ eines bereits definierten Tokens nicht automatisch neu gesetzt wird.

Ein Beispiel hierzu liefert Abbildung 7-30. Linksseitig listet es zwei Verwendungen desselben Konstrukts auf, wobei in der oberen der Text des letzten, als String deklarierten Tokens selektiert ist. Er wird nun durch Eingabe der Zahl `5` überschrieben, wodurch sich der Typ in ein Zahlliteral ändert. Eine automatisierte Übertragung dieses Typwechsels auf die andere Verwendung wäre unweigerlich mit Informationsverlusten verbunden, denn die Zeichenkette `"16 Jahre"` lässt sich aufgrund ihrer linguistischen Merkmale nur bedingt in eine Zahl konvertieren (nämlich beschränkt auf die ersten beiden Zeichen). **Anstelle einer Re-Typisierung** erfolgt bei vollständig definierten Konstrukten deshalb die **Markierung des geänderten Tokens als fehlerhaft** (in der Abbildung durch die rote Unterstreichung des in der Mitte befindlichen `5`-Tokens wiedergegeben).

Wird der Text eines Tokens geändert, dessen Text innerhalb der Definition festgelegt ist (vor allem zutreffend für Begrenzer), und bleibt dabei der Typ erhalten, dann wirkt sich das direkt auf die betreffende Definition aus. In bestimmten Fällen ist dieses Verhalten allerdings unerwünscht oder es gibt gleichwertige Alternativstrategien. Hierbei muss der Benutzer eine fallbezogene Entscheidung treffen. Unter anderem existieren solche Alternativen im Zusammenhang mit Schlüsselwörtern (Abschnitt 7.5.3.3) und beim Vorhandensein strukturell ähnlicher Konstrukte (Abschnitt 7.5.4).

7.5.3.3 Manuelle Re-Typisierung

Alternativ zur lexikalischen Analyse (Abschnitt 7.4.1) kann der Benutzer die Re-Typisierung von Tokens jedoch manuell und damit explizit anstoßen. Mit Ausnahme der auf Symbolen (`terms`-Attribut) basierenden **Tokentypen** müssen für die betreffende Token-Verwendung die **spezifischen Rahmenbedingungen erfüllt** sein (d.h. der reguläre Ausdruck muss passen oder die vorgegebenen Symbole für Start und Ende müssen übereinstimmen). Sonst ist die Re-Typisierung nicht zulässig und wird verweigert.

Diese Restriktion gilt indes nicht für die übrigen Token-Verwendungen, die auf derselben Definition beruhen wie das soeben geänderte Token, da andernfalls eine Re-Typisierung im Großteil der Fälle unmöglich wäre. Ein Typwechsel bedingt daher oft eine Anpassung des von den betroffenen Tokens gespeicherten Texts, was unter Umständen zu Informationsverlusten führen kann. Mithilfe adäquater **Migrationsstrategien** lässt sich dieser Verlust reduzieren und in wenigen Fällen sogar ganz vermeiden. Diese Fälle ergeben sich aus einer Gegenüberstellung der verschiedenen Tokentypen, wobei sich auf diejenigen beschränkt werden kann, deren Text in den Verwendungsmodellen abgelegt wird.

- Die höchste Qualität lässt sich bei der Migration in Richtung von `WrapLiteral`-Instanzen erreichen, da deren einzige Vorgabe im Vorhandensein eines bestimmten Start- und Endsymbols besteht. Das Risiko Informationen zu verlieren ist hier somit eliminiert. Ist der

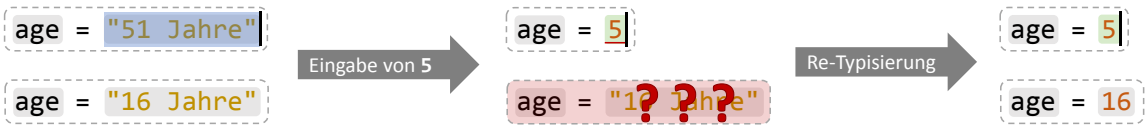

Abbildung 7-30: Beispiel für Re-Typisierung eines Nicht-Begrenzers mit einhergehendem Informationsverlust

Ausgangstyp eine andere Instanz von `WrapLiteral`, dann sind lediglich Start- und Endsymbol zu ersetzen (z.B. von `"some text"` nach `'some text'`). In allen anderen Fällen ist der Originaltext zu nehmen und zwischen die beiden Symbole einzubetten (z.B. von `7.89` nach `"7.89"` oder von `Josef.name` nach `"Josef.name"`).

- Soll in Richtung von `RegexLiteral`-Instanzen migriert werden, dann kann zur Geringhaltung des Informationsverlusts versucht werden, im ursprünglichen Text nach einem zum regulären Ausdruck passenden Teilstring zu suchen. Wird ein solcher gefunden, so kann er als neuer Wert übernommen werden. Ein Beispiel dazu zeigt ebenfalls Abbildung 7-30. Darin stößt der Benutzer im zweiten Schritt explizit eine Re-Typisierung an, wodurch aus dem String `"16 Jahre"` die Zahl `16` extrahiert und als neuer Text zugewiesen wird. Darüber hinaus ist es sinnvoll einen Standardwert anzugeben, so dass dieser gesetzt werden kann, falls kein passender Teilstring existiert.
- Bei der Migration in Richtung von `Referral`-Instanzen kann im Wesentlichen auf dieselbe Strategie zurückgegriffen werden wie bei `RegexLiteral`-Instanzen. Zusätzlich sollte bei erfolgreicher Extraktion eines Teilstrings dessen Auflösbarkeit überprüft werden. Nur wenn diese gegeben ist, wird der Teilstring übernommen. Die Angabe eines Standardwerts ergibt jedoch wenig Sinn, weil dessen Auflösbarkeit kontextabhängig ist und somit nicht überall gewährleistet werden kann. Stattdessen wird bei nicht vorhandenem, auflösbarem Teilstring der ursprüngliche Text beibehalten und die damit verbundene Kennzeichnung als fehlerhaft in Kauf genommen.
- Auch bei der Migration in Richtung von `Identifier`-Instanzen kann auf der Strategie bei `RegexLiteral`-Instanzen aufgesetzt werden. Gemäß der Intention von Bezeichnern gilt hier allerdings die Nicht-Auflösbarkeit eines extrahierten Teilstrings als zusätzliches Kriterium. Außerdem darf der Teilstring auch kein reserviertes Wort oder Symbol darstellen. Denn nur wenn der vom Teilstring repräsentierte Name im jeweiligen Kontext noch nicht vergeben wurde, ist ein Setzen dieses Namens zweckdienlich. Andernfalls bleibt – analog zu `Referral`-Instanzen – der ursprüngliche Text erhalten, wodurch die betreffenden Tokens als fehlerhaft markiert werden.
- Eine Migration in Richtung von `Comment`-Instanzen ist bei der Re-Typisierung ebenfalls möglich. Sie funktioniert auf dieselbe Weise wie die bei `WrapLiteral`-Instanzen. Obendrein wird aufgrund der informellen Natur von Kommentaren die zugrundeliegende Token-Definition jedoch gelöscht (siehe dazu Abschnitt 7.5.3.6).

Schlüsselwörter nehmen eine Sonderstellung ein, da sie je nach Bedarf sowohl auf Definitionsseite als auch in den Verwendungsmodellen festgelegt werden können. Relevant ist diese Unterscheidung erst hier, weil frühestens im Zuge der Manipulation eines Schlüsselwort-Tokens der zweite Fall eintreten kann. Bei ihrer Initialisierung wird der Text von Schlüsselwort-Tokens also zunächst auf Definitionsebene festgesetzt. Änderungen an Tokens sind – wie schon mehrfach erwähnt – immer innerhalb von Verwendungen durchzuführen.

So geschieht es auch im Beispiel aus Abbildung 7-31. Darin wird im ersten Schritt das unterste, als Schlüsselwort typisierte `human`-Token mit dem Text „script" überschrieben. Nachdem „script" (noch) nicht als Schlüsselwort eingetragen ist, wird es als Bezeichner interpretiert. Dies wiederum resultiert zwangsläufig in einem Fehler, weil das betreffende Statement bereits mit `Validate` einen anderen Bezeichner besitzt.

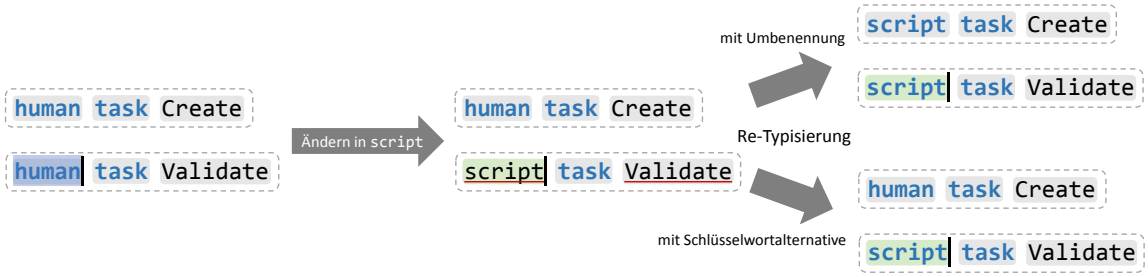

Abbildung 7-31: Beispiel für die Re-Typisierung als Schlüsselwort

Im nächsten Schritt wird nun vom Benutzer eine explizite Typisierung als Schlüsselwort eingeleitet. Hierbei werden zwei Alternativen unterschieden, von denen manuell eine auszuwählen ist. **Einerseits** kann die Wahl auf die **Umbenennung des bisherigen Schlüsselworts** fallen, was im Beispiel (oberer Pfad) dazu führt, dass der Text der zugrundeliegenden Token-Definition in „script" geändert wird. Wegen des Typ-Verwendungskonzepts (Abschnitt 2.2.1.3) zeigen daraufhin alle zugehörigen Token-Verwendungen diesen Text an.

Andererseits kann die Entscheidung darauf fallen, dass der zuvor neu eingegebene Text als **alternatives Schlüsselwort** behandelt wird. Im Beispiel (unterer Pfad) sind ab dem Zeitpunkt für alle Verwendungen der zugrundeliegenden Token-Definition die beiden Schlüsselwörter `human` und `script` erlaubt. Auf Modellseite hat dies zur Folge, dass der Schlüsselworttext nicht länger auf Definitionsebene, sondern in den jeweiligen Verwendungen hinterlegt ist. Die einzelnen Schlüsselwortalternativen werden dabei von der entsprechenden Token-Definition unter Zuhilfenahme von Instanzen des Konzepts `ValueRelation` verwaltet.

Ist diese Umstellung schon zuvor erfolgt und der Text eines solchen Schlüsselwort-Tokens wird modifiziert, kann erneut zwischen Umbenennen der einen Alternative und Hinzufügen einer weiteren Alternative ausgewählt werden. Im ersten Fall sind neben der betreffenden `ValueRelation`-Instanz auch sämtliche Token-Verwendungen mit diesem Text anzupassen, da hier das Typ-Verwendungskonzept nicht mehr greift. Im zweiten Fall hingegen wird lediglich eine neue `ValueRelation`-Instanz auf Definitionsseite ergänzt.

7.5.3.4 Automatisierte Strukturänderungen

Automatisierte Umbauten an der lokalen Baumstruktur finden auch bei vollständig definierten Konstrukten statt, sobald entsprechende Begrenzer in einer darauf basierenden Verwendung eingegeben und als solche identifiziert werden. Begrenzer, die ausschließlich **neue Knoten hinzufügen**, sind dabei gänzlich **unbedenklich**, weil die damit verbundenen Änderungen am Konstrukt direkt übernommen werden können, ohne Informationsverluste befürchten zu müssen.

Anders verhält es sich jedoch mit Restrukturierungsmaßnahmen, die ein **Löschen von Knoten** bedingen. Ein Delegieren der Löschoperation an die zugrundeliegende Definition und damit an alle zugehörigen Verwendungen geht in vielen Fällen mit einem Informationsverlust einher. Nachdem dieser allerdings zu vermeiden ist, werden in solchen Fällen **neue Statements inklusive neuer Konstrukte** angelegt.

Um ein Explodieren der Anzahl an Konstrukten zu vermeiden, ist es zweckmäßig, ein **Wiederverwenden der ursprünglichen Definition** zu gewährleisten, soweit dies möglich ist. Deswegen wird vor der Einführung neuer Konstrukte für ein neues Statement überprüft, ob die Struktur dieses Statements der Struktur des ursprünglichen Konstrukts gleicht. Ist dem so, dann wird dieses Konstrukt als Definition des neuen Statements eingesetzt und folglich kein neues Konstrukt generiert.

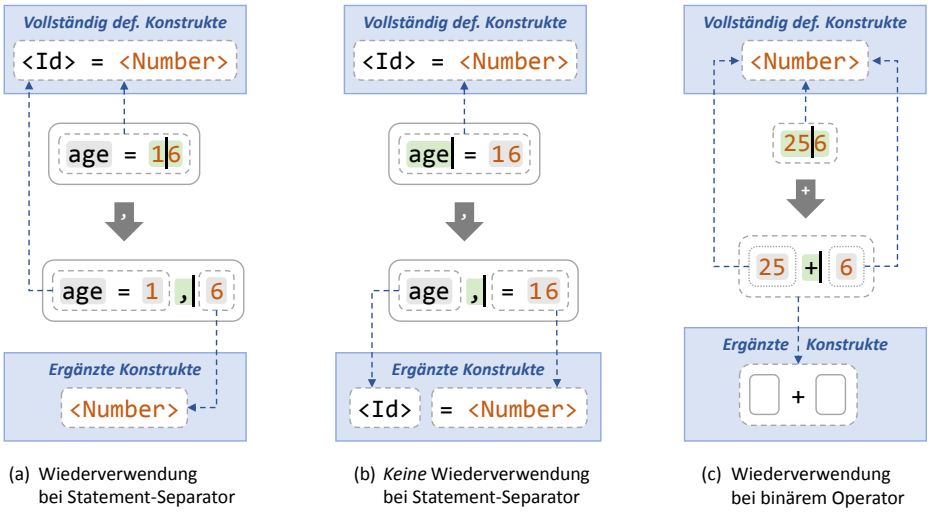

Abbildung 7-32: Drei Beispiele für automatisierte Restrukturierungen am CST

Verdeutlicht wird dieser Sachverhalt anhand der drei Beispiele in Abbildung 7-32. Ausgangspunkt von (a) und (b) ist dasselbe Statement mit dem gleichen zugrundeliegenden, vollständig definierten Konstrukt. Lediglich die Position des Text-Cursors unterscheidet sich. In (a) befindet er sich zwischen den beiden Ziffern der Zahl `16`, was bei der Eingabe eines Kommas (ein Statement-Separator) zur Aufspaltung in zwei separate Statements genau an dieser Stelle führt. Da allerdings die Struktur des linken Statements mit der des ursprünglichen Statements übereinstimmt, wird dessen Konstrukt als Definition des linken Statements eingesetzt. Die Struktur des rechten Statements hingegen passt nicht zum ursprünglichen Konstrukt, weshalb dafür ein neues Konstrukt angelegt wird.

Im Fall von (b) befindet sich der Cursor zwischen dem `age`- und dem =-Token. Die erneute Eingabe eines Kommas resultiert ebenfalls in zwei Statements, die durch Aufspaltung des ursprünglichen Statements genau an der Stelle entstehen. Nachdem jedoch weder das linke noch das rechte Statement dem Original im Hinblick auf die Struktur gleichen, wird für jedes eine eigene Definition erzeugt.

Das beschriebene Verhalten ist unabhängig vom Begrenzertyp und somit universell anwendbar. Demonstriert wird dies mit Beispiel (c) durch Eingabe eines Plus als binärer Operator. Er spaltet die Zahl `256` in zwei Operanden auf, die ebenfalls wieder Statements mit genau einem Zahl-Token umfassen. Demzufolge findet die ursprüngliche Definition des `256`-Statements in beiden Fällen Wiederverwendung. Darüber hinaus entsteht ein neues Konstrukt zur Repräsentation von Additionsausdrücken.

7.5.3.5 Manuelle Strukturänderungen

Die einfachste Form struktureller Änderungen ist das Einbetten von Tokens und **Gruppen in eine neue Gruppe** innerhalb eines Statements. Analoges gilt für das Entfernen einer Gruppe und dem damit verbundenen **Ausbetten** der bislang enthaltenen Knoten. Möglich ist beides deshalb problemlos, weil es sich um rein visuelle Modifikationen ohne jeglichen Informationsverlust handelt.

Des Weiteren kann es unter bestimmten Umständen sinnvoll sein, eine **Teilmenge von Knoten eines Statements in ein neues Statement einzubetten**, wobei letzteres einem neuen Block hinzugefügt wird, der an der Stelle der besagten Knoten innerhalb des ursprünglichen Statements platziert wird. Ein typischer Anwendungsfall hierfür ist das Beispiel aus Abbildung 7-30, für das damit eine alternative Konfliktbeseitigungsstrategie zur Verfügung steht.

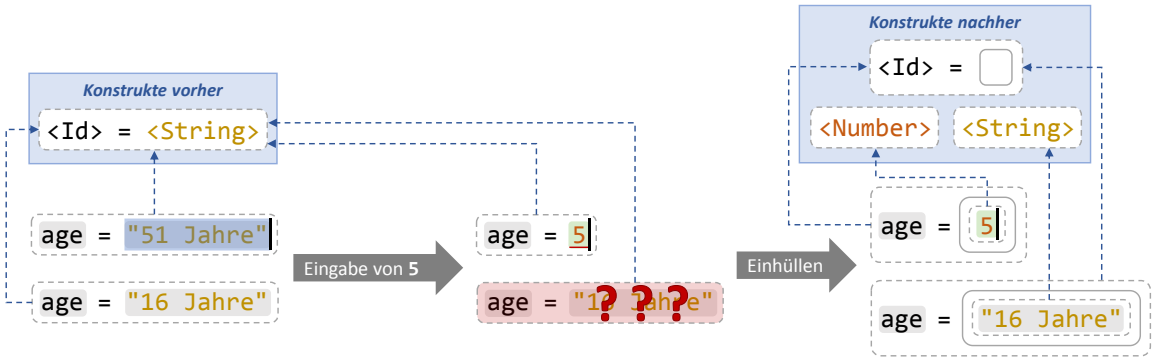

Abbildung 7-33: Konfliktbehebung durch Einhüllen von Tokens in neues Statement

Wie das im konkreten Fall aussieht, ist in Abbildung 7-33 veranschaulicht. Dort wird für das fehlerhafte `5`-Token angegeben, dass es von einem eigenen Statement repräsentiert werden soll, in das es schließlich eingehüllt wird. Um Konformität mit den anderen Statements sicherzustellen, die auf demselben Konstrukt beruhen, werden die betroffenen Tokens (hier nur `"16 Jahre"`) ebenfalls in ein separates Statement verschoben. Aufgrund der unterschiedlichen Typen kann beiden, neu erzeugten Statements nicht die gleiche Definition zugrunde liegen. Daher resultieren zwei neue Konstrukte, nämlich eines für Zahlen und eines für Zeichenketten. Diese Einführung zweier Konstrukte entspricht der eigentlichen Konfliktbehebung. Außerdem muss die ursprüngliche Statement-Definition an die geänderten Gegebenheiten angepasst werden, was in eine Ersetzung der bisherigen String-Token-Definition durch eine Block-Definition mündet.

Doch auch im fehlerfreien Fall können (direkt hintereinander folgende) Knoten eines Statements in ein neues Statement eingehüllt werden. Erwartungsgemäß führt das nicht zu zwei neuen Konstrukten, sondern nur zu einem. Würde man den vorgestellten Mechanismus bereits auf das anfängliche `"51 Jahre"`-Token anwenden, so entstünde lediglich ein neues Statement mit eingeschachteltem String-Token. Die Modifikation der ursprünglichen Statement-Definition würde aber ebenfalls auf dieselbe Weise durchgeführt werden. Ebenfalls möglich wäre es auf diesem Weg mehrere Knoten in ein neues Kind-Statement einzubetten (so z.B. durch gemeinschaftliche Auswahl von `=-` und `"51 Jahre"`-Token im ersten oberen Statement).

7.5.3.6 Selektions- und Löschmöglichkeiten

Das Selektieren und Löschen von Text und damit von Tokens innerhalb eines Statements ist eng verknüpft. Denn Text wird meistens dann ausgewählt, um ihn entweder zu Entfernen oder zu Überschreiben. Auch letzteres bedingt zunächst ein Löschen der selektierten Zeichen.

Deshalb ist es zweckmäßig die **Selektionsmöglichkeiten** derart zu **limitieren**, dass ein Überschreiben oder explizites Löschen immer zu sinnhaften Ergebnissen führt. Die Textauswahl muss enthaltene Knoten immer ganz oder gar nicht enthalten. Ausgenommen hiervon ist lediglich die Selektion von Text innerhalb eines einzigen Tokens. Andernfalls könnte nicht zweifelsfrei entschieden werden, welche Auswirkungen beispielsweise ein Löschen des ausgewählten Textes auf andere Verwendungen hätte.

Verdeutlicht wird dies mithilfe von Abbildung 7-34. Dort wird aus dem oberen `person`-Statement der ausgewählte Text gelöscht, wodurch letztlich ein Token mit dem Text „persef“ übrig bleibt. Eine eindeutige Lösung, wie diese Änderung auf das andere `person`-Statement zu transferieren ist, kann nicht angegeben werden, weil die Textauswahl auf Zeichenebene nicht übertragbar ist.

Abbildung 7-34: Problematik des Token-übergreifenden Löschens bei vollständig definiertem Konstrukt

Werden jedoch **Knoten als Ganzes** selektiert und daraufhin **gelöscht**, so geschieht dasselbe mit den Pendants der zugrundeliegenden Definition sowie sämtlicher Verwendungen. Enthält die Auswahl ein Bezeichner-Token, das bereits an anderer Stelle namentlich referenziert wird, dann führt dies zu einer Invalidierung des Textes der betreffenden Verweise. Dank des projizierenden Ansatzes, besitzen alle Modelle jedoch weiterhin Gültigkeit. Denn die Referenz ist nicht nur textuell über den Namen gespeichert, sondern auch im CST als eigenständiges Konzept hinterlegt. Um die besagte Invalidierung nicht durch ein Versehen auszulösen, ist es empfehlenswert, vor dem Durchführen einer Aktion, die zu einer Invalidierung führt, eine zusätzliche Bestätigung vom Benutzer einzuholen. Derselbe Sachverhalt gilt, wenn anstatt die Auswahl zu löschen ein neues Zeichen eingegeben und infolgedessen diese überschrieben wird. Im Falle einer erfolgreichen Eingabe, wird für das eingetippte Zeichen ein neues Token angelegt.

Im engen Zusammenhang mit dem Löschen von **Knoten** steht das **Ausblenden** der selbigen. Dieses Thema wurde bereits in Abschnitt 7.5.3.1 adressiert, da neu hinzugefügte Knoten innerhalb eines vollständig definierten Konstrukts unter Umständen nur in der tatsächlich modifizierten Verwendung angezeigt werden sollen. Statt Knoten wirklich gänzlich aus der zugrundeliegenden Definition und damit aus allen darauf basierenden Verwendungen zu entfernen, kann der Benutzer auch die Absicht verfolgen, die betreffenden Knoten nur lokal auszublenden. Doch wie bereits im obigen Fall muss er das auch hier explizit spezifizieren. Eine mögliche Umsetzung ist es, den Benutzer beim Anstoßen eines Löschvorgangs zu fragen, ob er die Definition wirklich manipulieren will oder alternativ die ausgewählten Knoten in der aktuellen Verwendung lediglich ausblenden möchte. Alternativ könnte ein spezielles Tastenkürzel (z.B. Strg+Entf) angeboten werden. Bei semantischen Features würde als Alternative zur Ausblendung (ebenfalls analog zu oben) nur die Wert- bzw. Zielzuweisung entfernt werden.

Soll ein bereits verborgener **Knoten wieder eingeblendet** werden, so kann auf die gleiche Lösungsstrategie zurückgegriffen werden, wie schon im grafischen Kontext (Abschnitt 6.2.2). Befindet sich demnach der Text-Cursor in einem Statement mit ausgeblendeten Kindknoten, dann werden diese trotzdem angezeigt, jedoch mit verringerter Deckkraft.

7.5.4 Ersetzen des zugrundeliegenden Konstrukts

Um den Arbeitsfluss möglichst nah an dem von Freitexteditoren zu halten, ist die Unterstützung einer weiteren Interaktionsmöglichkeit mit der konkreten Syntax von positiver Bedeutung. So soll die Definition, die einer Verwendung zugrunde liegt, unter bestimmten Umständen ausgetauscht werden können. Diese Umstände beruhen primär auf dem **Prinzip der strukturellen Gleichheit** (Abschnitt 5.3.1.3) und sind vergleichbar mit dem Finden passender Vervollständigungsvorschläge in Abschnitt 7.5.2.

Anhand des Beispiels in Abbildung 7-35 wird der Nutzen eines solchen Features untermauert. Vorneweg angemerkt sei, dass das Konstrukt für die Darstellung von Zahlen zwar abgebildet ist, aus Lesbarkeitsgründen auf die Visualisierung der zugehörigen Instanzspezialisierungsbeziehungen jedoch verzichtet wurde. Ausgangspunkt ist ein Additionsausdruck, der auf einem entsprechenden Konstrukt beruht. Der Plus-Operator wird nun durch einen Minus-Operator ersetzt, was einen Konflikt mit dem zugrundeliegenden Konstrukt hervorruft. Gäbe es kein strukturgleiches Konstrukt mit Minus-Operator, dann würde der Text der betreffenden Token-Definition einfach überschrieben werden und der Konflikt wäre sofort behoben. Nachdem aber ein solches Konstrukt bereits existiert, stellt der

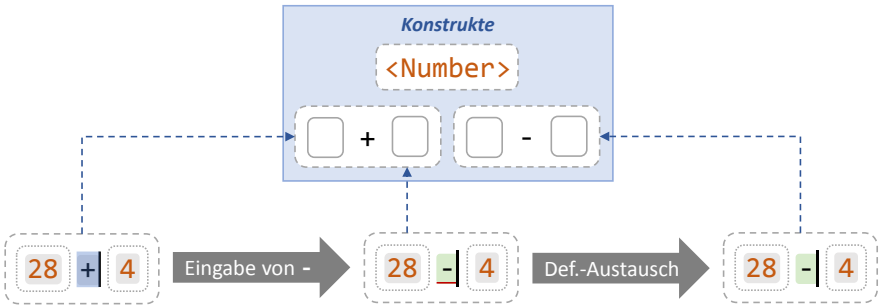

Abbildung 7-35: Beispiel für den Austausch der Definition bei einer Verwendung

Austausch der zugrundeliegenden Definition eine potentielle Alternativlösung dar. Der Benutzer steht unter diesen Umständen somit in der Pflicht sich für eine der beiden Strategien zu entscheiden. Im Beispiel fällt die Wahl auf die Ersetzung des Konstrukts, wodurch der Ausdruck schließlich wieder Gültigkeit erlangt.

Verallgemeinert ausgedrückt ist der **Austausch eines Konstrukts eine mögliche Alternative**, wenn zum einen die Struktur der geänderten Verwendung mit der Struktur einer (oder gegebenenfalls auch mehrerer) existierenden Definition(en) übereinstimmt. Zum anderen müssen in der Verwendung auch die übrigen Rahmenbedingungen dieser Definition(en) eingehalten werden. Als wichtigstes Merkmal sei hier die Textgleichheit der aufeinander beziehenden Token-Knoten (einerseits Verwendung, andererseits Definition) genannt, deren Text auf Definitionsseite spezifiziert ist. Im obigen Beispiel ist damit das Minuszeichen gemeint. Es wurde im zweiten Schritt vorerst in der Verwendung überschrieben, wodurch strukturelle Gleichheit mit dem Subtraktionskonstrukt hergestellt ist.

Aufgrund dieser Verallgemeinerung können Ersetzungsvorschläge auch bei allen anderen Arten von Modifikationen unterbreitet werden, die zu einer Manipulation des zugehörigen Konstrukts führen würden. Beispielhaft angenommen sei hierfür eine Sprache, die über Konstrukte verfügt, welche in ihrer Struktur den Feld- und Methodendeklarationen bei Java entsprechen. Zudem sei angemerkt, dass nach der Eingabe einer öffnenden Klammer automatisch die zugehörige schließende Klammer ergänzt wird. Sie wird direkt nach dem ebenfalls automatisch angehängten Block (öffnende Klammern gelten standardmäßig als Block-Einleitung) platziert. Fügt man jetzt der Verwendung einer Felddeklaration am Ende eine runde Klammer und direkt im Anschluss eine geschweifte Klammer hinzu, dann wird empfohlen, das Felddeklarationskonstrukt durch ein Methodendeklarationskonstrukt auszutauschen.

7.5.5 Entfernen textueller Konstrukte

Das explizite Löschen existierender Konstrukte ist gemäß Abschnitt 5.4.1 zu jedem Zeitpunkt erlaubt. Wegen potentiell weitreichender Folgen sollten nur solche Konstrukte gelöscht werden, die nirgends verwendet werden. Verwendungen derartiger Konstrukte tauchen somit in keinem Dokument auf und können folglich nicht innerhalb eines Text-Editors gefunden und entfernt werden. Das explizite Löschen wird demnach auch im textuellen Kontext ausschließlich über die Palette ermöglicht. Existieren dennoch irgendwelche Verwendungen des zu entfernenden Konstrukts, dann ist der Benutzer darauf hinzuweisen, so dass er sich der Gefahr weitreichender Einschnitte bewusst ist.

7.6 Integration mit stringenter Modellierung

Ist der stringente Modellierungsmodus aktiv, so sind die **Bearbeitungsmöglichkeiten gemäß** der zugrundeliegenden **abstrakten Syntax eingeschränkt**. Manipulationen am Dokument sind nur insoweit zulässig, wie sie keine Änderungen an der abstrakten und konkreten Syntax hervorrufen. Dies ist

von den einzelnen Tokentypen zu berücksichtigen. Um deren Komplexität jedoch nicht zusätzlich zu beeinträchtigen, ist es bei einer Implementierung zweckmäßig, diesen Aspekt extern zu behandeln. Hierfür bietet es sich beispielsweise an, das von vielen modernen Programmiersprachen bereitgestellte Konzept der Ausnahmebehandlung einzusetzen.

Gearbeitet wird stets mit **textuellen Konstrukten**, die über ein **Mapping** auf ein Konstrukt der abstrakten Syntax verfügen. Folglich kann der stringente Modellierungsmodus nur dann betreten werden, wenn sämtliche im Dokument vorhandenen Konstruktverwendungen auf die Domänen abgebildet sind. Ist das nicht der Fall, so müssen zunächst die automatische Induktion des Domänen-Stacks inklusive zugehöriger Mappings initiiert oder die entsprechenden Artefakte händisch angelegt werden.

Abgesehen von Statement-Separatoren wird Text auch bei der stringenten Modellierung stets in einem Token eingegeben, das in einem Statement (auch indirekt, falls innerhalb einer Gruppe befindlich) liegt. Nachdem dieses **Statement** einem Domänenelement zugeordnet sein muss, ist von **vornherein anzugeben, auf welche Statement-Definition es sich bezieht**. Denn jede Statement-Definition (als Konstrukt der konkreten Syntax) verfügt im stringenten Modus über ein Mapping auf ein Konstrukt der abstrakten Syntax. Vorzugsweise geschieht die Angabe einer Definition durch Auswahl aus einer Liste möglicher Vervollständigungsvorschläge. Gegenüber den Vorschlägen bei der Arbeit mit vollständig definierten Konstrukten im freien Modus (Abschnitt 7.5.2), werden ausschließlich solche Definitionen angezeigt, die im aktuellen Kontext verwendet werden dürfen. Andere würden spätestens bei der Abbildung auf das Domänenmodell ohnehin abgelehnt werden.

Bei der stringenten Modellierung ist der **Template-Gedanke** demgemäß weitaus **stärker ausgeprägt**. Die Grenzen der als Platzhalter dienenden Tokens und Blöcke sind fix, d.h. wenn Text innerhalb eines Tokens eingetippt wird, dann gehört er immer komplett zu diesem Token. Wie unter anderem Jetbrains MPS zeigt, können die Bearbeitungsmöglichkeiten hierfür noch weiter optimiert werden, so dass die Eingabe stark an diejenige von Freitext erinnert. Allerdings liegt dies außerhalb des Untersuchungsspektrums dieser Arbeit, weshalb hier nicht weiter darauf eingegangen wird.

Im Gegensatz zur freien Modellierung sind auch die **Selektionsmöglichkeiten** stärker **limitiert**. So kann entweder der Text innerhalb eines Tokens selektiert werden oder alternativ nur ganze Statements, die jedoch alle im gleichen Block liegen müssen. Geschuldet ist dies der zugrundeliegenden abstrakten Syntax, deren Regeln stets einzuhalten sind. Dasselbe Verhalten kann beispielsweise auch bei MPS beobachtet werden, da hier ausschließlich mit dem AST interagiert wird.

7.7 Erweiterung des Ableitungsprozesses für textuelle DSMLs

Dank der generischen Anwendbarkeit des allgemeinen Mapping Meta-Modells (Abschnitt 5.2.2), ist für textuelle DSMLs keine diesbezügliche Erweiterung erforderlich. Auch beim Ableiten genügen somit die Konzepte dieses Meta-Modells. Lediglich der Ableitungsprozess als solches muss an einigen Stellen ergänzt werden, was nachfolgend geschieht. Der erste Unterabschnitt befasst sich dabei mit der Verarbeitung der einzelnen Knotentypen, während es im zweiten Unterabschnitt um eine vereinfachte Abbildung von Strukturen des CST auf die abstrakte Syntax geht, so dass Multiplizitäten mit einer Obergrenze größer 1 generiert werden können.

7.7.1 Differenzierung anhand der Knotentypen

Welche Artefakte auf Domänenseite abgeleitet werden, hängt nicht zuletzt von den Typen der zugrundeliegenden Knoten ab. Bevor irgendwelche Features berücksichtigt werden können, müssen daher **zunächst** deren zugehörige **semantische Knoten ausgewertet** werden. Im Textuellen handelt es sich

dabei stets um Statements. Anders typisierte semantische Knoten gibt es nicht. Gruppen und Blöcke als rein visuelle Knoten können hierbei ignoriert werden.

Bei der Auswertung von **Statements** wird das **Vorhandensein eines Bezeichners** ergänzend berücksichtigt. Denn der dem rein visuellen Bezeichner-Label zugewiesene Text kann zur Benennung des Instanz-Konzepts im Domänenmodell oftmals identisch übernommen werden. Je nach regulärem Ausdruck des Bezeichners ist das nicht immer möglich, weil dieser Ausdruck auch Zeichen erlauben kann, die für Konzeptnamen verboten sind. In einem solchen Fall werden ungültige Zeichen durch ein beliebig gültiges, aber im Vorfeld festgelegtes Zeichen (z.B. ein Unterstrich) ersetzt. Des Weiteren kann es zu Namenskollisionen kommen, die jedoch (beispielsweise) durch Anhängen einer Zufallszahl auflösbar sind. Durch Einführen eines speziellen Element-Mappings, das sich zusätzlich um den Abgleich von Bezeichner und Konzeptname kümmert, kann beides synchron gehalten werden. Andernfalls wird die Aktualisierung des Konzeptnamens erst bei der nächstmaligen Durchführung des Ableitvorgangs vollzogen.

Bei **Blöcken und Gruppen** gilt es lediglich eine etwaig auftretende semantische Sichtbarkeitseigenschaft zu beachten. Diese Eigenschaft darf mit Ausnahme von Bezeichnern und Kommentaren auch bei jedem Token zum Einsatz kommen. Darüber hinaus verfügen Blöcke immer über eine semantische Enthaltensein-Beziehung. Verarbeitet werden die zwei Features jedoch ganz regulär, wie bei den Grundlagen der beispielgetriebenen Entwicklung in Kapitel 5 besprochen. Eine Erweiterung existiert lediglich im Zusammenhang mit Schlüsselwörtern, worauf später in diesem Abschnitt noch eingegangen wird.

Die Beschriftung und damit die **Label-Eigenschaft** erfährt eine spezielle Behandlung. Wie ein Label auszuwerten ist, steht nämlich zusätzlich **in Abhängigkeit zum Typ des rein visuellen Eigentümerknotens** (ein Token) dieses Labels. Zutreffend ist dies nicht für alle Tokentypen, sondern ausschließlich für Literale und unter bestimmten Umständen auch Schlüsselwörter. Der in Kapitel 5 eingeführte Label-Inferrer ist folglich um entsprechende Funktionalitäten zu erweitern (einfach umzusetzen beispielsweise mittels Typspezialisierung). Auch weiterhin wird immer ein Attribut erzeugt und dessen Multiplizität auf `1..1` oder `0..1` gesetzt. Wie eine Flexibilisierung der oberen Multiplizitätsgrenze erreicht werden kann, wird erst in Abschnitt 7.7.2 erläutert.

- *Literal mit regulärem Ausdruck*: Den Typ des resultierenden Attributs legt bei `RegexLiteral` schon die jeweilige Instanz mithilfe von `typeToMap` fest. Dies ist deshalb sinnvoll, weil mit dieser Typangabe jedes Literal eine gewisse Grundintention innehat (nämlich ganze Zahl, Fließkommazahl, Wahrheitswert oder Text). Wird als Zieltyp nicht String gewählt, dann muss dafür gesorgt werden, dass der reguläre Ausdruck nicht mehr Werte zulässt als der Zieltyp gestattet.
- *Literal mit Start- und Endsymbol*: Als Attributtyp wird bei der Verarbeitung von `WrapLiteral`-Instanzen stets String gesetzt. Im Zuge einer Wertzuweisung werden das jeweilige Start- und Endsymbol abgeschnitten und nur die dazwischen liegende Zeichenkette übernommen.
- *Schlüsselwort*: Das Label eines Schlüsselwort-Tokens kann verschiedene, mithilfe von `ValueRelations` spezifizierte Schlüsselwörter aufnehmen. Sind derartige Alternativen gegeben und besitzt das Label daher semantischen Charakter (siehe auch Abschnitt 7.5.3.3), dann wird zusätzlich eine entsprechende Enumeration erzeugt und diese als Attributtyp gesetzt. Ist eine derartige Enumeration schon vorhanden, dann wird sie lediglich den neuen Umständen entsprechend angepasst. Weitere Details hierzu finden sich in den Abschnitten 5.3.2.2 und 5.4.2.2.

Durch die Erweiterung des allgemeinen **Visibility-Inferrer** wird **bei optionalen Schlüsselwörtern** erreicht, dass der Name des Schlüsselworts als Name für das zu generierende Attribut übernommen wird. Möglich ist dies nur für Schlüsselwörter, deren Text auf Definitionsebene festgesetzt ist. Bei mehreren alternativen Wörtern kann ein passender Name nicht eindeutig ermittelt werden.

Abschließend gilt es noch **Verweise** zu betrachten. Nachdem das ihnen zugeordnete Label rein visueller Natur ist, wird es gänzlich ignoriert. Ausgewertet hingegen wird die semantische Referenz, die auf ein beliebiges Statement zeigen darf. Aufgrund ihres informalen und damit rein visuellen Charakters sind Kommentare für den Ableitungsprozesses völlig belanglos.

7.7.2 Automatische Simplifizierung

Abgesehen von Attributen, die aus Enthaltensein-Beziehungen hervorgehen, kann als **Obergrenze für die Multiplizität nie** ein Wert **größer 1** resultieren. Das hängt damit zusammen, dass ausschließlich Aufzählungen von Statements, aber nicht von einzelnen Tokens als eigentliche Träger von Werten unterstützt werden. Diese Statements müssen obendrein immer in einem Block liegen. Geschuldet ist dies der generischen Anwendbarkeit des vorgestellten Ansatzes, der deshalb größtmögliche Flexibilität gewährleistet.

Um diesem Defizit entgegenzuwirken, müssen **Statements als einzelne Werte identifiziert** werden können. Das kann allerdings nur für solche Statements geschehen, die über genau ein Token als Wertträger verfügen und dieses Token seinen Wert im Verwendungsmodell speichert (d.h. Literale, Verweise sowie Tokens, die eine Reihe alternativer Schlüsselwörter aufnehmen können). Im Statement sind zwar weitere Knoten erlaubt, doch müssen diese rein visuell auf Definitionsebene festgelegt sein (vornehmlich Gruppen und Token-Separatoren ohne ein semantisches Feature). Bezeichner scheiden aus, da sie ein Statement auf Verwendungsseite textuell referenzierbar machen. Darüber hinaus müssen auch die betreffenden Blöcke, die als Schnittstelle zwischen den Werte-Statements und den eigentlichen Adressaten dieser Werte fungieren, gewisse Rahmenbedingungen befolgen. Denn sie dürfen ausschließlich Statements umfassen, denen dieselbe Definition zugrunde liegt. Andernfalls ist eine Typverträglichkeit nicht sichergestellt.

Das Beispiel aus Abbildung 7-36 soll die gesamte Thematik noch einmal illustrieren. Linksseitig abgebildet ist eine Sprache mit zwei Konstrukten, wobei nur das obere für Simplifizierungsmaßnahmen beim Ableitungsvorgang in Frage kommt. Ohne solche Maßnahmen würde standardmäßig das Domänen-Meta-Modell resultieren, wie es rechts oben dargestellt ist. Im Zuge einer Vereinfachung kann das Zahl-Konstrukt jedoch als Integer-Attribut in das `Ratings`-Konzept eingegliedert werden, was rechts unten geschehen ist. Gäbe es noch ein weiteres Konstrukt, das beispielsweise als einzigen Knoten ein String-Token enthält, und würde es innerhalb eines `ratings`-Statement Verwendung finden (z.B. anstelle des `63`-Statements), dann wäre dem vorherigen Absatz zufolge eine Vereinfachung unzulässig.

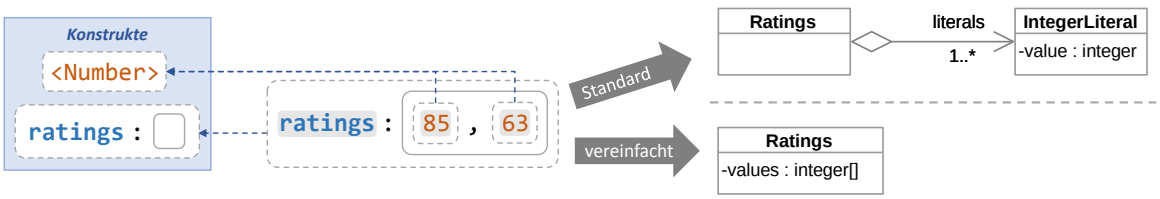

Abbildung 7-36: Exemplarisches Textfragment mit Simplifizierungspotential auf Seite der abstrakten Syntax

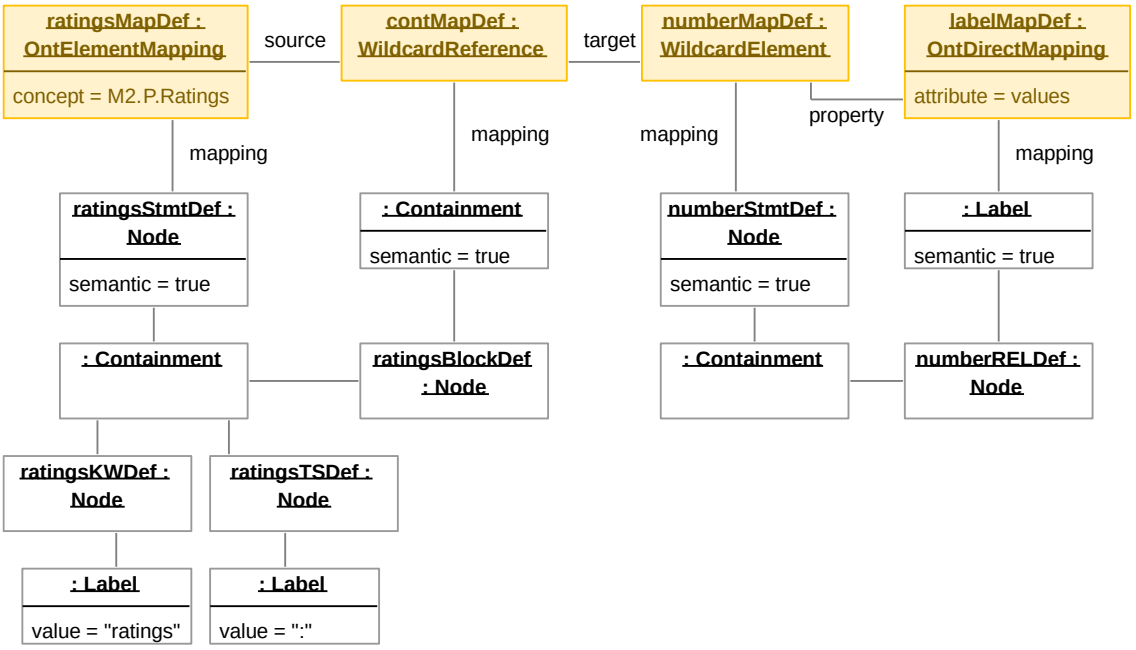

Abbildung 7-37: Beispielhafte Definition eines simplifizierenden Mappings

Intern auf Definitionsebene umgesetzt wird das Beispiel wie in Abbildung 7-37 veranschaulicht. Zur Verhinderung, dass die semantische Enthaltensein-Beziehung sowie die Zahl-Statement-Definition auf eigene Sprachartefakte abgebildet werden, kommen die beiden Platzhalter-Mappings `WildcardReference` und `WildcardElement` zum Einsatz. Dadurch ist sichergestellt, dass das Label-Mapping `labelMapDef` sich beim Domänenelement nicht auf `numberMapDef`, sondern mit `ratingsMapDef` auf das `ratings`-Statement bezieht.

Dies entspricht in ihren Grundzügen der Umsetzung des Attribut-bezogenen Links aus Abschnitt 6.1.2. In analoger Weise muss daher auch der **Ableitungsprozess erweitert** werden, was mit einer Ergänzung von Element- und Containment-Inferrer erreicht wird. Beide müssen dazu in der Lage sein, Statements als Werteträger ermitteln zu können. Unter welchen Umständen dies der Fall ist, wurde ein paar Absätze zuvor bereits erläutert. Trifft die notwendige Bedingung für alle Statements einer zugrundeliegenden Definition zu, dann werden anstelle der regulären Mappings Instanzen von `WildcardReference` und `WildcardElement` eingesetzt. Hierbei wird erneut auf das **KISS-Prinzip** [127] gesetzt, weil das abgeleitete Meta-Modell weniger komplex ausfällt, sofern dies die Umstände erlauben. Am Beispiel aus Abbildung 7-36 ist das durch Vergleichen der beiden resultierenden abstrakten Syntaxen direkt ersichtlich.

7.8 Erweiterbarkeit

Wie in Abschnitt 7.1 beschrieben, liegt der Fokus des vorgestellten Standardverhaltens auf Sprachen, die möglichst auch von Domänenexperten mit informatischer Grundbildung gelesen und verstanden werden können. Es gibt aber auch Sprachklassen, für die das nicht oder nur entfernt zutrifft. Ein Beispiel hierfür sind sogenannte Template-Sprachen. Dennoch ermöglicht der präsentierte Ansatz auch diese Sprachklasse zu unterstützen, indem das textuelle Meta-Modell entsprechend erweitert wird (unter anderem um einen zusätzlichen Tokentyp, der die fixen Teile eines Templates typisiert). Dies gilt auch für alle anderen Klassen, denen die standardmäßig zur Verfügung gestellten Konzepte nicht genügen. Folglich ist Erweiterbarkeit ein zentrales Thema, das gesondert betrachtet wird.

Der übliche Erweiterungsfall ist das **Hinzufügen eines neuen Tokentyps**. Jedem Tokentyp liegt eine bestimmte Intention zugrunde, die oft mit Auswirkungen auf die lokale Baumstruktur verknüpft ist. Der Typ muss somit für jede Containerart wissen, wie er sich ihr gegenüber zu verhalten hat. Dies kann von einem Unterdrücken der Eingabe, über ein einfaches Einfügen des zugehörigen Symbols bis hin zu komplexen Umbaumaßnahmen der Baumstruktur führen. Für letztgenannten Fall stellen binäre Operatoren einen typischen Vertreter dar.

Nicht empfohlen dagegen ist die **Ergänzung weiterer Containertypen**, da zum einen mit den vorhandenen Konzepten bereits sämtliche Konstellationen textuellen Enthaltenseins abgebildet werden können. Andererseits müsste beim Hinzufügen eines neuen Containertyps jeder Tokentyp festlegen, wie er sich verhält, wenn ein entsprechend typisiertes Token innerhalb eines derartigen Containers erkannt wird. Dies wäre also mit einem enorm hohen Aufwand verbunden.

7.9 Verfeinerung der generierten abstrakten Syntax

Der generische Typisierungsansatz gestattet es, neue Knotentypen bei Bedarf zu entwickeln und dem textuellen Meta-Modell hinzuzufügen. Von Relevanz ist dies insbesondere für Begrenzertypen, die nach ihrer Eingabe und Identifikation die lokale Baumstruktur manipulieren (können) – gemäß einer stets zugrundeliegenden Erwartungshaltung. Das Resultat der **Restrukturierung beruht** dabei auch immer **auf der Struktur anderer Meta-Modelle**, die einem sehr ähnlichen (wenn nicht sogar identischen) Problem begegnen mussten.

Als **typischer Vertreter** gelten im textuellen Meta-Modell die **Operatortypen**. So erwartet ein binärer Operator immer einen linken und einen rechten Operanden, was in ihrer Gesamtheit einen binären Ausdruck ergibt. Ein Großteil aller existierenden Programmiersprachen unterstützen binäre Ausdrücke und bilden sie im AST auf eine stets vergleichbare Struktur ab. Auf diesem Erfahrungsschatz basieren auch unsere Umbaumaßnahmen, wie sie in Abschnitt 7.4.2.4 vorgestellt wurden.

Demzufolge kann die abstrakte Syntax **anderer Sprachen als ein Indikator für die Einführung zusätzlicher Begrenzertypen** betrachtet werden. Nachdem diese Sprachen aber schon mit einer geeigneten Lösung aufwarten, kann diese in einigen Fällen (direkt oder in leicht abgewandelter Form) auf die abgeleitete Syntax übertragen werden. Das hätte den Vorteil, dass nicht alle denkbaren Konstellationen in Form von Beispielmodellen ausmodelliert werden müssten, um ein entsprechend flexibles Meta-Modell zu erhalten. Stattdessen wird die erwartete Flexibilität durch die Übertragung der besagten Lösung aus einem vorhandenen Meta-Modell erreicht.

Was unter Flexibilität zu verstehen ist, wird im Folgenden anhand eines Beispiels verdeutlicht. Den Startpunkt bildet der in Abbildung 7-38 illustrierte Ausdruck `4 + 6 * 3`, der sich intern aus der Multiplikation `6 * 3` und der Addition `4 +` *Multiplikationsergebnis* zusammensetzt. Angenommen dieser Ausdruck stellt das einzige Beispielmodell dar, dann resultiert daraus im Verlauf des Ableitungsprozesses ein Meta-Modell, das demjenigen unten links entspricht.

Um nun ebenfalls Zahlen als rechten Operanden einer Addition angeben zu können, müsste dies explizit in Form eines weiteren Beispiels modelliert werden. Vor allem bei vielen verschiedenen binären Ausdrücken bedeutet dies allerdings einen recht hohen Aufwand. Als Alternative wird der Benutzer deshalb auf solche Fälle hingewiesen und kann daraufhin das Meta-Modell anpassen. Dabei kann er sich auf eine andere abstrakte Syntax beziehen und eine passende Lösung händisch transferieren.

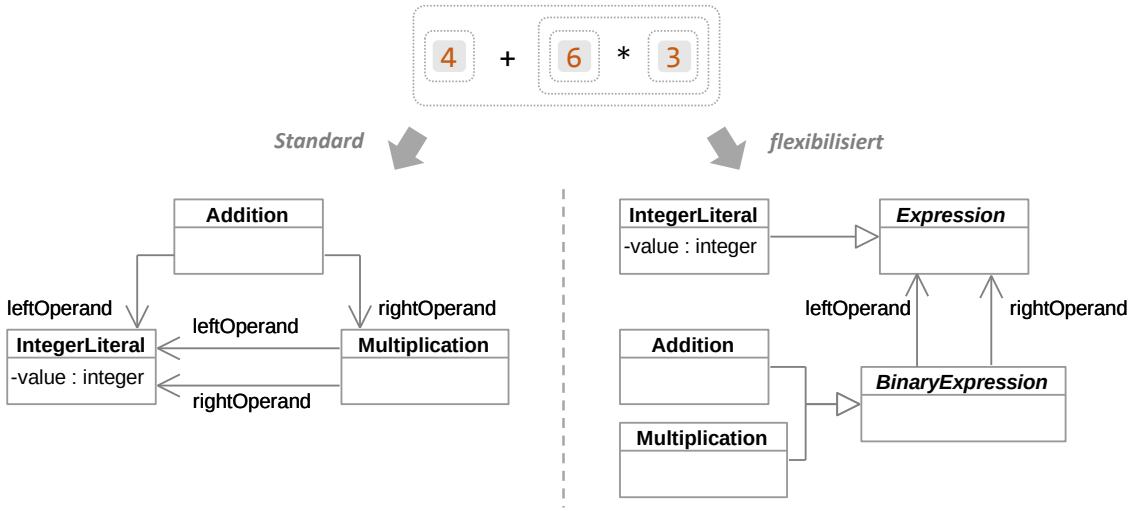

Abbildung 7-38: Gegenüberstellung von standardmäßiger und flexibilisierter abstrakter Syntax für binäre Ausdrücke

Erfreulicherweise handelt es sich bei derartigen Wiederverwendungen aber um einen klassischen Anwendungsfall im Umfeld der Meta-Modellierung, für den mit der **Komposition von Meta-Modellen** [84] bereits diverse Lösungsstrategien zur Verfügung stehen. Eine geeignete Strategie ist die in [37] eingeführte Template-Instanziierungsmethode, mit der sich Fragmente aus einem gegebenen Meta-Modell in der Gestalt von Templates auf andere Meta-Modelle übertragen lassen. Bezogen auf das obige Operatorbeispiel würde ein passendes Template die Basisstruktur für Ausdrücke vorgeben, woraus sich durch Template-Instanziierung das Klassendiagramm in Abbildung 7-38 gewinnen ließe.

Zur Unterbreitung entsprechender Verfeinerungsvorschläge für die abstrakte Syntax empfiehlt es sich, zusätzlich **Informationen der konkreten Syntax mit einzubeziehen**. Denn die betreffenden Strukturen sind verlässlicher, da sie im Regelfall durch die Erkennung und Anwendung eines Begrenzertyps entstanden sind. Dadurch kann beispielsweise auf einfachem Weg bestimmt werden, ob ein Statement einen binären Ausdruck repräsentiert. Dies trifft genau dann zu, wenn es drei Kinder aufweist, von denen das erste und das letzte ein Block sind und das zweite ein binärer Ausdruck (in Form eines Statements). Bei einer Limitierung auf die abstrakte Syntax könnten lediglich die Namen der einzelnen Artefakte in Kombination mit ihrer Struktur analysiert werden. Sobald jedoch einzelne Artefakte umbenannt würden, hätte dies zur Folge, dass die Analyse misslingt.

Als gänzlich andere Möglichkeit, die vor allem im Kontext textueller Sprachen von Bedeutung ist [32], gilt die Entwicklung einer **DSML basierend auf einer existierenden Sprache** (meist eine GPL). In einem solchen Fall können sämtliche Konstrukte direkt verwendet werden (z.B. binäre Ausdrücke, sofern vorhanden) und brauchen nicht neu definiert werden. Hinzugefügte Konstrukte beeinflussen die Basissprache dabei nicht, sondern stellen einen separaten Aufsatz dar, der bedarfsweise Artefakte dieser Sprache spezialisiert oder anderweitig referenziert. Ergänzend beachtet werden muss lediglich, dass Konstrukte der Basissprache nicht manipuliert werden dürfen. Für deren Verwendung gelten somit die Restriktionen der stringenten Modellierung. Der vorgestellte Ansatz kann daher ebenso für (neue) DSMLs verwendet werden, denen eine Basissprache zugrunde liegt.

7.10 Zusammenfassung

Mit der beispielgetriebenen Entwicklung textueller DSMLs wird **Anforderung 1b** (konstruktivistische Entwicklung von DSMLs im textuell ontologischen Kontext) **gänzlich erfüllt**. In erster Linie wird dies durch die Unterstützung eines Modellierungsworkflows erreicht, der dem Benutzer eine bequeme Eingabe von Modellen mithilfe der Tastatur erlaubt. Dies ist die gängige Eingabeform für textuelle Modelle. Durch den automatisierten Umbau der Dokumentstruktur, der basierend auf der Eingabe bestimmter Symbole erfolgt, können komplexe Dokumente schnell und mühelos frei erstellt werden. In welcher Weise die Restrukturierung stattfindet, hängt vom jeweiligen Symbol und dessen Intention ab. Das Standardverhalten ist jedoch konfigurierbar, weshalb der präsentierte Ansatz ein hohes Maß an Flexibilität aufweist. Dennoch hat der Benutzer zu jeder Zeit die Möglichkeit, die Baumstruktur gemäß seiner Bedürfnisse anzupassen. Die Unterstützung des besagten Workflows stellt den ersten Teil des eigenen Beitrags in diesem Kapitel dar. Aufgrund der umsetzungsnahen Problemstellung, die es durch die Bereitstellung eines entsprechenden Werkzeugs zu lösen gilt, zählt dieser Beitrag zur Kategorie „Software Engineering“ aus Abschnitt 1.3.

Anders verhält es sich mit dem zweiten Beitrag dieses Kapitels, der eher konzeptuellen Charakter besitzt. Bei ihm handelt es sich um die Erweiterung des allgemeinen, visuellen Meta-Modells für die Erstellung textueller Modelle. Diese Erweiterungen stellen die Basis für verschiedenen Symbole dar, deren Eingabe zu eine automatisierten Restrukturierung der Baumstruktur führen kann. Außerdem wird auf Eigenheiten des Ableitungsprozesses eingegangen, die der textuelle ontologische Kontext bedingt. Dies betrifft ebenfalls die automatische Unterbreitung von Verfeinerungsvorschlägen, wodurch auch **Anforderung 3 adressiert** wird.

Ein konkretes Anwendungsszenario für das beispielgetriebene Entwickeln einer textuellen Modellierungssprache findet sich in Abschnitt 8.3. Der Anwendungsfall ist dort bewusst in die *Model Workbench* eingebettet, um ihn innerhalb des zu dieser Arbeit gehörenden Prototyps exemplarisch nachvollziehen zu können.

8 Model Workbench als Implementierungsgrundlage

Die *Model Workbench* ist eine mit HTML5 und JavaEE umgesetzte Modellierungsplattform, die als Grundlage für die im Kontext der vorliegenden Dissertation entstandene Implementierung dient. Sie wird im Rahmen des Projekts „Kompetenzzentrum für praktisches Prozess- und Qualitätsmanagement" (KpPQ) entwickelt und dort primär zur Modellierung von Geschäftsprozessen sowie zur Definition und Anpassung domänenspezifischer Prozessmodellierungssprachen eingesetzt. Gefördert wird dieses Projekt durch den Europäischen Fonds für regionale Entwicklung (EFRE, Fördernummer 1502/89304-01/2012). Die Entscheidung auf Webtechnologien aufzusetzen wurde vorrangig von den Industriepartnern des KpPQ-Projekts vorangetrieben, die die resultierende Anwendung auf möglichst vielen Endgeräten einsetzen wollen. Zudem erleichtert diese Entscheidung den Einsatz der *Model Workbench* innerhalb einer Cloud-Umgebung.

Im Folgenden wird zunächst die auf Erweiterbarkeit fokussierte Architektur der *Model Workbench* beschrieben. Anschließend werden die beiden, als Erweiterungen realisierten Editoren vorgestellt, die für unterschiedliche Visualisierungsformen konzipiert sind. Den Anfang macht der grafisch ontologische Editor (Abschnitt 8.2), gefolgt vom textuell ontologischen Editor (Abschnitt 8.3). Auf den linguistischen Editor wird nicht separat eingegangen, da er für die Unterstützung der freien Modellierung nur geringfügig angepasst werden musste. Die erforderlichen Anpassungen wurden bereits in Abschnitt 4.1 geschildert. Alle drei Editoren erfüllen den Zweck einer Machbarkeitsstudie, die die in dieser Arbeit vorgestellten Konzepte und Lösungsstrategien validieren.

8.1 Architektur der Model Workbench

Die Architektur der *Model Workbench* (Abbildung 8-2) ist eine Umsetzung des typischen Client-Server Modells, wobei der Server in drei weitere Schichten unterteilt ist. Als Basis zur Persistierung der (Meta-)Modelle und damit als **Modellrepositorium** findet die Graph-Datenbank Neo4J Verwendung. Neben umfangreichen Anfragemöglichkeiten stellt sie auch eine performante Unterstützung des ACID-Prinzips bereit.

Darauf aufsetzend liegt die **Model API**, die einen komfortablen und uniformen Zugriff auf die Knoten und Kanten der zugrundeliegenden Graph-Datenbank ermöglicht. Alle elementaren Änderungen an der Datenbank werden von atomaren Kommandos gekapselt, wobei letztere innerhalb von Operatoren konsolidiert und dadurch komplexe Änderungsoperationen realisiert werden können. Jedes Kommando verfügt über eine passende Undo/Redo-Funktionalität, wovon wegen der Konsolidierung auch sämtliche Operatoren profitieren.

Die Elemente des LMM (Ebene, Konzept, Attribut etc.) sind selbst wieder Operatoren. Alle Änderungen, die an Modellelementen durchgeführt werden, werden auf atomare Kommandos heruntergebrochen und können deshalb stets rückgängig gemacht bzw. widerrufen werden. Neue, durch Erweiterungen hinzugefügte Operatoren können dann komfortabel auf Basis der Elemente implementiert werden. Als eigenständige Operatoren realisiert sind unter anderem sämtliche Algorithmen zur inkrementellen

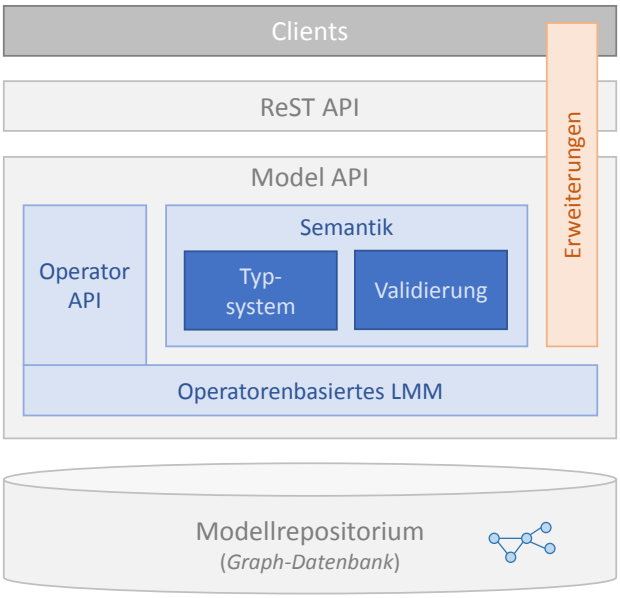

Abbildung 8-2: Architektur der Model Workbench

Ableitung einer abstrakten Syntax, die in den Kapiteln 4 und 5 vorgestellt werden. Nähere Informationen zur Entwurfsentscheidung, **Modellelemente als Operatoren** umzusetzen, sind der Dissertation von Jahn zu entnehmen [66].

Ein weiterer Bestandteil der Model API Schicht ist die Semantikkomponente, von der allerdings nur das **Validierungsmodul** für die späteren Erläuterungen von Bedeutung ist. Dieses Modul zeichnet verantwortlich für die automatisierte Generierung von Empfehlungshinweisen, wie sie unter anderem in den Abschnitten 4.4 und 4.5.2 zur Verfeinerung eines induzierten Meta-Modells beschrieben wurden.

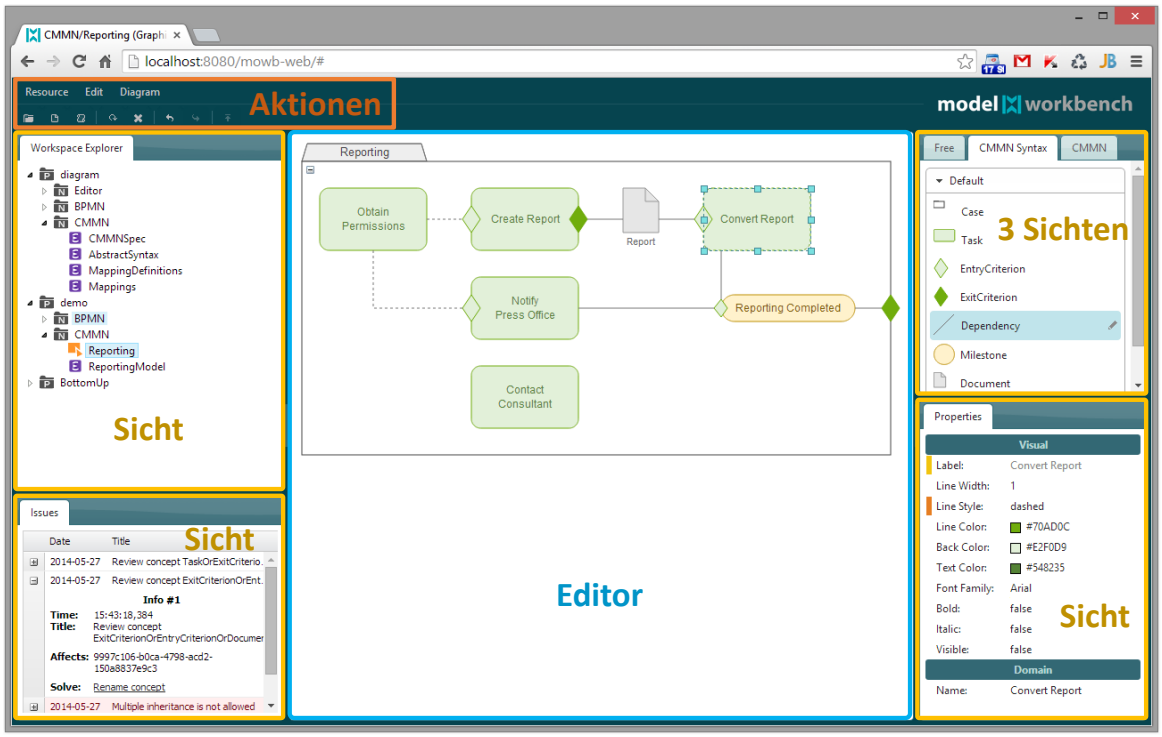

Abbildung 8-1: Oberflächenaufbau der Model Workbench

Die **ReST API** bildet die externe Schnittstelle des Servers, über die **Clients** mit den Modellen interagieren können. Typischerweise gibt es pro Editor einen spezialisierten ReST-Service, der einen vereinfachten Zugriff auf die darunterliegende Model API gewährleistet und gleichzeitig den Informationsfluss auf ein akzeptables Maß reduziert.

An der Stelle kommt auch die **Erweiterbarkeit** der *Model Workbench* zum Tragen, die sich über die obersten drei Schichten der Architektur erstreckt. So können bei Bedarf neue Editoren inklusive einem zugehörigen ReST-Service ergänzt werden. Da die Weboberfläche der *Model Workbench* stark der grafischen Benutzerschnittstelle moderner IDEs (z.B. Eclipse, IntelliJ IDEA und Visual Studio) ähnelt, ist es gleichermaßen möglich, weitere Sichten und Aktionen beizusteuern. Wo die einzelnen Komponenten innerhalb der Oberfläche zu finden sind illustriert Abbildung 8-1 durch Markierung der entsprechenden Bereiche. Ebenso erweitert werden kann die Model API, indem neue, komplexe Operatoren oder Validierungsregeln hinzugefügt werden.

8.2 Unterstützung der grafisch ontologischen Modellierung

Als Ausgangspunkt für die nachfolgenden Ausführungen dient Abbildung 8-1, die die *Model Workbench* mit geöffnetem Diagramm-Editor einschließlich zugehöriger Sichten zeigt. Die verschiedenen Markierungen sowie die großen farbigen Schriftzüge können ignoriert werden, da sie nachträglich zur Veranschaulichung des Oberflächenaufbaus ergänzt wurden. Exemplarisch ist ein Modell geladen, das sich an der Case Management Model and Notation (CMMN) [109] orientiert. Dieses Modell wurde zunächst frei modelliert und daraus eine vollständige DSML induziert.

Dass eine Sprache zugrunde liegt, ist primär daraus ersichtlich, dass insgesamt drei **Paletten** zur Verfügung stehen (Abbildung 8-3). Die erste Palette (Reiter *Free*) stellt ein breites Spektrum vorkonfigurierter Formen und Verbindungslinien bereit, die auf der Zeichenfläche zu eigenen, grafischen Konstrukten komponiert und in ihrer Konfiguration (z.B. Hintergrundarbe und Umrandungsdicke) angepasst werden können. Derartige Konstrukte finden sich dann als Einträge in der zweiten Palette (Reiter *CMMN Syntax*), die im Beispiel aktiv ist. Jedes Konstrukt kann über diese Palette auch direkt umbenannt werden. Dazu muss lediglich auf den kleinen Stift am rechten Rand des zugehörigen Paletteneintrags geklickt werden. Der Stift erscheint, sobald die Maus über einem solchen Eintrag platziert wird. Ist eine

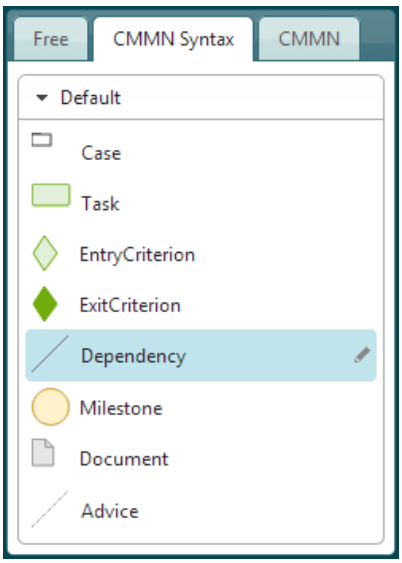

Abbildung 8-3: Paletten des Diagramm-Editors

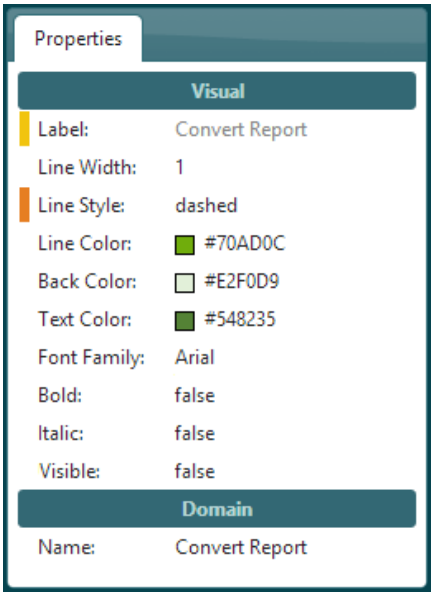

Abbildung 8-4: Sicht für Eigenschaften des momentan selektierten Elements

der zwei ersten Paletten aktiv, so befindet sich der Editor im freien Modellierungsmodus. Anders verhält es sich, wenn die dritte Palette (Reiter *CMMN*) ausgewählt ist. Dann werden beim Modellieren die Rahmenbedingungen konsequent beachtet, die von einer abstrakten Syntax vorgegeben sind. Diese dritte Palette ist nur dann vorhanden, wenn eine vollständige DSML vorliegt. Inhaltlich umfasst die Palette ausschließlich solche grafischen Konstrukte, die über ein Mapping auf ein Konstrukt der abstrakten Syntax abgebildet sind.

Die **Sicht zur Bearbeitung der literalen Eigenschaften** des aktuell im Diagramm-Editor ausgewählten Elements ist eine weitere wichtige Komponente (Abbildung 8-4). Sie gliedert sich in einen visuellen Teil (*Visual*) und einen Domänenteil (*Domain*). Letztgenannter ist nur dann verfügbar, wenn eine abstrakte Syntax und demnach ein Domänenmodell zugrunde liegt. Er zeigt sämtliche literalen Attribute des gemappten Domänenelements, denen ein Wert zugewiesen werden darf. Im Beispiel ist die Aufgabe „*Convert Report*" selektiert, weshalb auch deren Eigenschaften aufgelistet sind. Ihr zugeordnetes Domänenelement besitzt nur das Attribut *Name*, das somit als einziger Eintrag im Domänenteil dargestellt ist. Dieses Attribut ging aus der semantischen *Label*-Eigenschaft hervor, die als erster Eintrag im visuellen Teil aufgeführt ist. Dass sie semantischer Natur ist, wird durch den gelben Balken symbolisiert. Nachdem bereits ein entsprechendes Mapping existiert, kann der Text der *Label*-Eigenschaft nicht mehr editiert werden und ist deshalb grau gedruckt. Stattdessen ist das besagte *Name*-Attribut zu setzen. Der orangefarbene Balken neben der „*Line Style*"-Eigenschaft dagegen gibt an, dass diese auf Verwendungsseite rein visuell überschrieben wurde. Alle übrigen Eigenschaften ohne spezielle Kennzeichnung entstammen der Definition.

Bei der ontologischen Modellierung ebenso von Bedeutung ist die in Abbildung 8-5 illustrierte **Issues-Sicht** (für „Issues" eignet sich im Deutschen in diesem Kontext am besten die Übersetzung „Belange"). Die darin aufgelisteten Hinweise und Fehlermeldungen wurden beim vorherigen Ableiten der Domänen-(Meta-)Modellelemente mit den dazugehörigen Mappings ausgegeben. Die beiden in Abbildung 8-5 gezeigten Fehlermeldungen (rot hinterlegt) resultieren daraus, dass zwei Meta-Konzepte mehr als ein Meta-Konzept spezialisieren und dadurch Mehrfachvererbung auftritt. Gemäß der aktuellen Konfiguration ist diese jedoch verboten, was zu einer Invalidierung des induzierten Domänen-Meta-Modells durch das in Abschnitt 8.1 erwähnte Validierungsmodul führt. Um dem Substitutionsprinzip gerecht zu werden (Abschnitt 4.3.1), hat der Ableitungsalgorithmus (Abschnitt 5.3.2) zwei

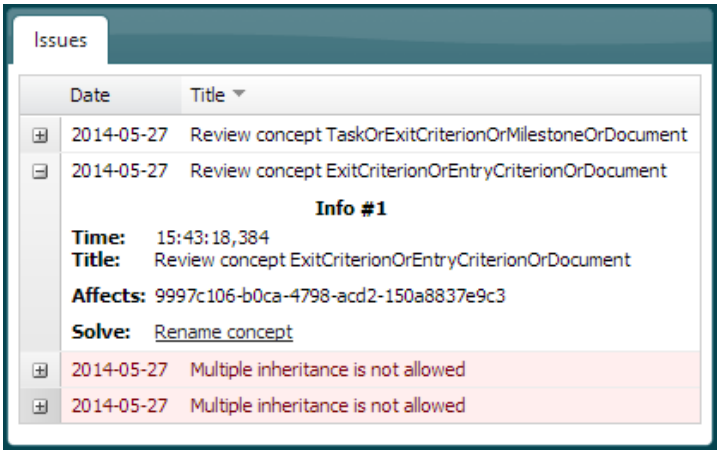

Abbildung 8-5: Issues-Sicht mit zwei Hinweisen und zwei Fehlermeldungen

abstrakte Basis-Konzepte erzeugt und sie mit einem computergenerierten Namen versehen. Beide Namen genügen aber mit sehr hoher Wahrscheinlichkeit nicht den Anforderungen des Modellierers, weshalb entsprechende Review-Hinweise ausgegeben werden. Durch Ausklappen eines Eintrags in der Issues-Sicht können weitere Informationen dazu abgerufen werden. Insbesondere bedeutsam ist, dass jede Meldung üblicherweise mindestens einen Operator bereitstellt, mit dessen Hilfe der adressierte Belang behandelt werden kann. So bieten Review-Hinweise unter dem Punkt *Solve* einen Operator an, der es erlaubt das betreffende Konzept umzubenennen.

8.3 Unterstützung der textuell ontologischen Modellierung

Der Großteil der Modellierungstätigkeiten des textuellen Workflows findet direkt innerhalb des Texteditors statt. Nur ein kleiner Teil der Nutzerinteraktion erfordert einen Zugriff auf die Palette. Aus Platzgründen beschränkt sich daher der in Abbildung 8-6 dargestellte Screenshot auf die Anzeige dieser beiden Komponenten. Workspace-Explorer und Issues-Sicht sind ausgeblendet.

Beim **frei erstellen** Dokument handelt es sich um ein **textuelles Prozessmodell**, dessen konkrete Syntax an die der Declarative Process Intermediate Language (DPIL) [156] angelehnt ist. DPIL ist eine Sprache zur Definition regelbasierter Prozesse, mit deren Hilfe ein hohes Maß an Agilität bei der Ausführung erreicht werden kann. Regeln lassen sich unterscheiden in harte Regeln (Pflicht) und weiche Regeln (Empfehlungen bzw. Best Practices). Inhaltlich betrachtet ist der Beispielprozess recht ähnlich zu der grafischen Variante aus Abbildung 8-1, jedoch nicht mit ihr identisch. Dies liegt unter anderem daran, dass das textuelle Beispiel noch nicht vollständig spezifiziert ist (siehe letzte Zeile, in der aktuell eine Eingabe erfolgt). Im Gegensatz zum grafischen Beispiel liegt noch keine komplette DSML vor, da Mappings und Domänenseite fehlen. Dies ist sofort ersichtlich, da es keine zweite Palette gibt, die ausschließlich Konstrukte mit Domänenbezug umfasst.

Alle **Hervorhebungen** innerhalb des Texteditors entsprechen weitgehend der Legende in Abbildung 7-3. Lediglich die gepunktete Umrandung wird weitläufiger eingesetzt als bei den Abbildungen des 7. Kapitels. Jeder Block, der genau ein Statement enthält, wird als ein Rechteck mit gepunktetem Rand visualisiert, wobei auf die Darstellung der Statement-Umrandung verzichtet wird. Nachfolgend wird daher nicht näher auf den Aufbau des Dokuments eingegangen, sondern es werden nur relevante und sichtbare Features des Editors erklärt.

Eines dieser Features ist die **Auto-Vervollständigung von Statements**, wie sie unten links in Abbildung 8-6 gezeigt wird. Sie setzt sich aus einer Liste mit Vorschlägen und einer Vorschau des derzeit selektierten Vorschlags zusammen. Für die exemplarische Eingabe von „en“ werden die

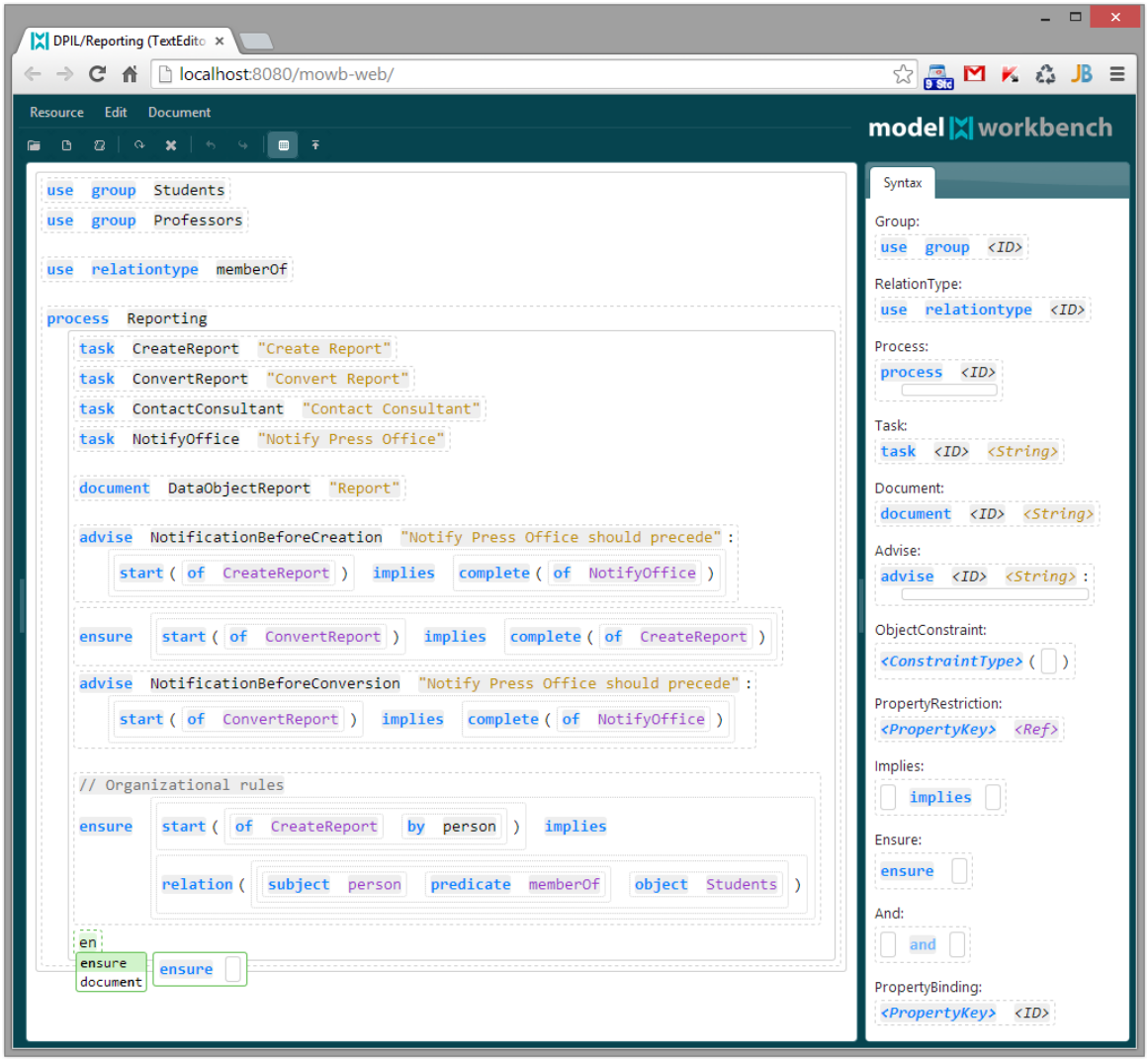

Abbildung 8-6: Screenshot der Model Workbench mit geöffnetem Text-Editor

visuellen Konstrukte `Ensure` und `Document` angeboten, da beide den besagten Teilstring beinhalten. Durch Bestätigen der Auswahl wird das noch unvollständige `en`-Statement durch eine neue Verwendung des `Ensure`-Konstrukts ersetzt.

Ein weiteres wichtiges Feature ist die Möglichkeit, die **Visualisierung der Baumstruktur deaktivieren** zu können. Zu dem Zweck existiert der momentan aktivierte Button in der Werkzeugleiste. Ist diese Funktion ausgeschaltet, dann sieht das Dokument aus wie frei geschriebener Text mit Syntaxhervorhebung.

Die rechts in Abbildung 8-6 befindliche **Palette** ist ebenfalls ein **zentrales Hilfsmittel**, vor allem im Rahmen der freien Modellierung. Denn über sie können – wie auch im grafisch ontologischen Kontext (Abschnitt 8.2) – die einzelnen Konstrukte umbenannt werden. Zugleich zeigt sie eine originalgetreue Vorschau des jeweiligen Konstrukts, was eine Einflussnahme auf die einzelnen Bestandteile ermöglicht. Kursiv gedruckte Tokens repräsentieren Platzhalter, die es im Dokument mit Werten zu befüllen gilt (z.B. `<ID>`, `<String>` und `<PropertyKey>`). Des Weiteren können Knoten leicht transparent erscheinen, was ausdrückt, dass diese optional und somit im Dokument nicht zwingend darzustellen sind. Im Beispiel trifft dies für das `and`-Schlüsselwort des `And`-Konstrukts zu. Eine Verwendung dieses Konstrukts findet sich im letzten `ensure`-Statement zwischen den Klammern des `start`-Statements. Dort ist das `and`-Schlüsselwort bewusst rein visuell ausgeblendet, so dass dies keine

Auswirkungen auf ein späteres Domänenmodell hat. Semantische Sichtbarkeit hätte bei der Vorschau zusätzlich zur Folge, dass der betreffende Knoten gelb hinterlegt bzw. umrandet ist. Gesetzt werden kann diese Eigenschaft direkt im Texteditor an einer entsprechenden Verwendung oder über ein Kontextmenu, das auf jedem Knoten innerhalb der Vorschau aufgerufen werden kann. Gleiches gilt für das Anbringen von Annotationen zur aktiven Beeinflussung der Algorithmen, die für das Ableiten einer abstrakten Syntax inklusive Mappings verantwortlich sind.

8.3.1 Initiale abstrakte Syntax

Wird aus dem Beispielprozess aus Abbildung 8-6 eine abstrakte Syntax abgeleitet und wird hierbei auf ein automatisches Eliminieren von Mehrfachvererbung verzichtet, dann sieht das resultierende Meta-Modell wie das in Abbildung 8-7 aus. Gleichzeitig wird eine Reihe von Verfeinerungsvorschlägen generiert, mit deren Hilfe ein Modellierer effizienter die Qualität des Meta-Modells steigern kann.

Am offensichtlichsten sind die **wenig aussagekräftigen Namen** der fünf abstrakten Konzepte (unter anderem `GroupOrRelationTypeOrProcess` und `PropertyRestrictionOrAnd`). Sie sollten deshalb von einem Modellierungsspezialisten überprüft und gegebenenfalls umbenannt werden.

Ferner wird ein **Verfeinerungsvorschlag** für die beiden Attribute von `And` erzeugt, weil sie einerseits aus einem **binären Ausdruck** entstanden sind (das Schlüsselwort `and` wurde als binärer Operator deklariert) und andererseits verschiedene Typen aufweisen. Gemäß Abschnitt 7.9 ist es bei einer solchen Konstellation oftmals ratsam die Konzepthierarchie für Ausdrücke aus einer anderen Sprache (z.B. mittels Meta-Modell Komposition) zu übernehmen.

Schließlich werden noch drei **Empfehlungen für Einfachvererbung** generiert. Die erste betrifft die `label`-Attribute der Konzepte `Task`, `Document` und `Advise`, die zweite betrifft die `implies`-Attribute von `Advise` und `Ensure` und die dritte die `key`-Attribute von `PropertyBinding` und `PropertyRestriction`.

Geleitet durch diese Vorschläge wurde das Meta-Modell derart umgebaut, dass es nun die Struktur wie in Abbildung 8-8 besitzt. Durch die **Reduktion redundanter Informationen** und die **präzisere Namensgebung** wirkt es bereits auf den ersten Blick aufgeräumter als die ursprüngliche Fassung.

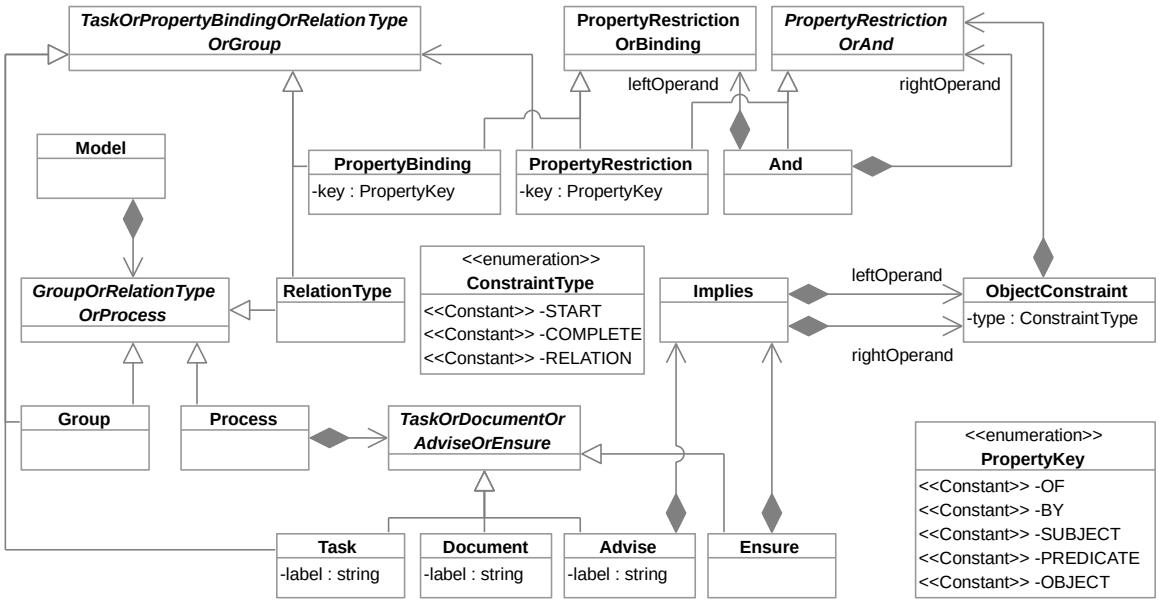

Abbildung 8-7: Vereinfachte abstrakte Syntax von DPIL direkt nach dem initialen Ableiten

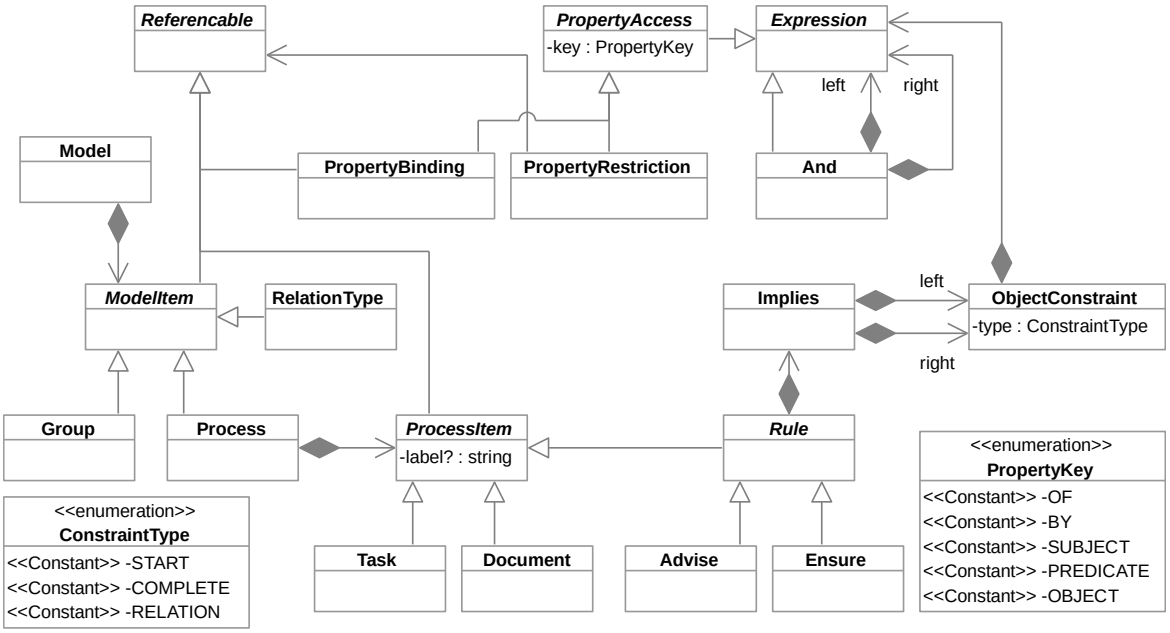

Abbildung 8-8: Verfeinerte Variante der initiale abgeleiteten abstrakten Syntax von DPIL

Auch bei Anwendung der beispielgetriebenen Entwicklung von DSMLs kommt ein Modellierungsexperte nicht um die händische Anpassung der abstrakten Syntax herum. Die beiden großen Vorteile dieses Ansatzes sind jedoch, dass zum einen durch das automatisierte Ableiten bereits eine Grundstruktur vorhanden ist. Zum anderen erhält der Experte mit den Verfeinerungsempfehlungen Hinweise auf Stellen im Meta-Modell, die mit hoher Wahrscheinlichkeit Verbesserungsbedarf aufweisen und daher einem Review unterzogen werden sollten. Im Beispiel wurden alle Problempunkte korrekt erkannt und anschließend von einer Person manuell überarbeitet. Im Optimalfall kann sie auf Operatoren zurückgreifen, die das Meta-Modell in der gewünschten Weise evolvieren (z.B. wie in Abbildung 8-5 der Operator zum Umbenennen von Konzepten).

8.3.2 Modifizierte abstrakte Syntax

Zwischenzeitlich wird der **Beispielprozess** aus Abbildung 8-6 um die Regeln aus Abbildung 8-9 **ergänzt**. Als große Neuerung gilt das hinzugekommene `Milestone`-Konstrukt, dessen Aufbau dem des Vorschlagskonstrukts gleicht. Im Unterschied dazu enthält der eine Meilenstein allerdings einen `And`-Ausdruck als Kindelement. Darüber hinaus gibt es mit `write` einen weiteren Typus von Objekt-Constraints. Diese Constraints können nun außerdem unmittelbarer Bestandteil von `And`-Ausdrücken sein.

```
// Data rules
ensure   complete ( of  CreateReport )   implies   write ( DataObjectReport )
ensure   start ( of  ConvertReport )   implies   write ( of  DataObjectReport )
ensure   complete ( of  ConvertReport )   implies   write ( of  DataObjectReport )

milestone  Done  "Reporting completed" :
    complete ( of  CreateReport )   and   complete ( of  ConvertReport )   and   complete ( of  NotifyOffice )
```

Abbildung 8-9: Zusätzliche Regeln für den textuellen Reporting-Prozess

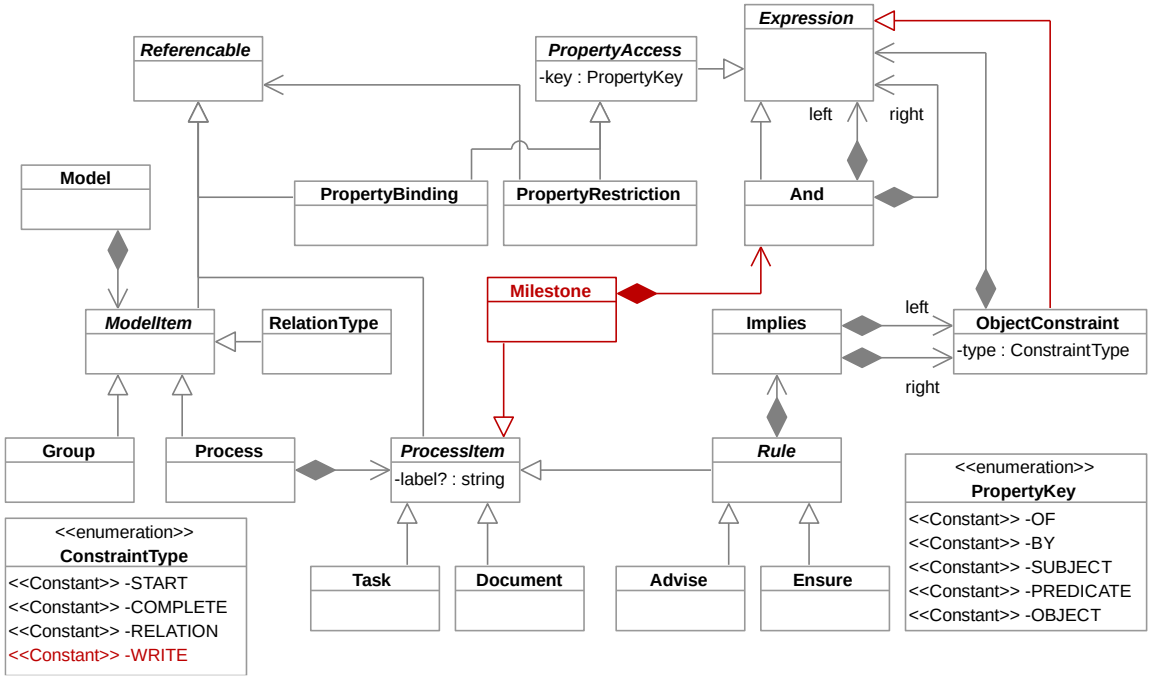

Abbildung 8-10: Automatisiert geänderte abstrakte Syntax des vereinfachten DPIL-Derivats

Beim **Ableiten notwendiger Änderungen** an der verfeinerten Variante der abstrakten Syntax resultiert das Meta-Modell, wie es Abbildung 8-10 zeigt. Die angepassten bzw. hinzugekommenen Artefakte sind rot gekennzeichnet. So wurde mit `Milestone` ein neues Konzept für Meilensteine generiert. Es spezialisiert das Konzept `ProcessItem`, da dieses der Typ der möglichen Kindkonzepte von Prozessen ist und auch Meilensteine als Kinder von Prozessen modelliert werden. Dadurch erbt `Milestone` auch das erforderliche `label`-Attribut. Die Kompositionsbeziehung zwischen `Milestone` und `And` existiert, weil der einzelne Meilenstein eine Verwendung eines `And`-Konstrukts umfasst. Das Enumerationsliteral `WRITE` hat seinen Ursprung in `write`-Schlüsselwörtern der Datenregeln. Schließlich wurde noch eine Generalisierungsbeziehung zwischen `Expression` und `ObjectConstraint` geschaffen. Sie ist notwendig, da Constraints nun gleichermaßen als linker und rechter Operand von `And`-Ausdrücken eingesetzt werden dürfen.

Auch nach dem automatisierten Ableiten von **Änderungen am Meta-Modell** muss ein Modellierungsspezialist diese sichten, **beurteilen und gegebenenfalls refaktorisieren**. Im aktuellen Szenario können Meilensteine ebenfalls als Regeln betrachtet werden, weshalb es sich anbietet, das entsprechende Konzept als Spezialisierung von `Rule` festzulegen (anstelle von `ProcessItem`) und die Kompositionsbeziehung zu entfernen. Dies bedingt, dass die Kompositionsbeziehung zwischen `Rule` und `Implies` auf `Expression` umgebogen wird. Zugleich muss `Implies` `Expression` spezialisieren, wodurch die beiden Assoziationen zwischen `Implies` und `ObjectConstraint` überflüssig werden. Dieses Verfeinerungspotential wird ebenso automatisiert erkannt, und zwar anhand der Attribute `left` und `right`, die sowohl in `Implies` als auch in `And` vorkommen. Denn beide Attribute weisen einen Typ auf, der sich Dank der ergänzten Generalisierungsbeziehung zwischen `ObjectConstraint` und `Expression` innerhalb derselben Typhierarchie befindet.

9 Resümee mit Ausblick

Abrundend erfolgt zunächst ein Rückbezug auf die einleitend in Abschnitt 1.2 hergeleiteten Anforderungen. Es wird darauf eingegangen, in welchen Teilen der vorliegenden Arbeit sie adressiert und wie sie dabei gelöst wurden. Zudem wird ein Überblick über potentiell zukünftige Forschungsthemen gegeben, die im Kontext der beispielgetriebenen Entwicklung domänenspezifischer Modellierungssprachen stehen oder daran anknüpfen.

9.1 Bezugnahme auf die ursprünglichen Anforderungen

Den Grundstein für die Erfüllung von **Anforderung 1** (konstruktivistische Entwicklung von DSMLs im ontologischen Kontext) legt die in Kapitel 5 eingeführte Modellhierarchie und demnach das visuelle Meta-Modell in Verbindung mit dem Mapping Meta-Modell. Denn diese Strukturen ermöglichen sowohl freies als auch stringentes Modellieren auf einer gemeinsamen Modellbasis und damit einer gemeinsamen Technologie. Bei letzterer handelt es sich um die in Kapitel 8 vorgestellte *Model Workbench*, die als generische Datenstruktur das in Abschnitt 2.2 vorgestellte LMM bietet. Zusätzlich wird in den übrigen Teilen von Kapitel 5 eine Methode beschrieben, mit deren Hilfe - basierend auf Beispielen und dadurch auf intuitive Weise – vollständige DSMLs konstruiert werden können.

Anforderung 1 ist anhand der beiden möglichen Visualisierungsformen in zwei **Teilanforderungen** untergliedert, was primär dem **Modellierungsworkflow** geschuldet ist, der sich beim Umgang mit grafischen und textuellen Sprachen unterscheidet. Der Workflow für grafische DSMLs konzentriert sich auf eine Bedienung mit Maus bzw. Touch-Gesten, während beim textuellen Modellieren der Fokus auf der Eingabe von Text mithilfe einer Tastatur liegt. Adäquate Lösungen für einen intuitiven Arbeitsfluss innerhalb der jeweiligen Visualisierungsform werden in Kapitel 6 respektive Kapitel 7 präsentiert.

Bei **Anforderung 2** geht es um eine konstruktivistische Entwicklung von DSMLs im linguistischen Kontext. Im Mittelpunkt steht die abstrakte Syntax der zu entwerfenden DSML, da mit der LML bereits eine konkrete Syntax vorgegeben ist. Adressiert wird diese Anforderung in Kapitel 4, indem einerseits Algorithmen zur direkten, inkrementellen Ableitung eines Meta-Modells aus einer gegebenen Menge von Beispielmodellen eingeführt werden. Andererseits werden Verfahren vorgestellt, mit deren Hilfe lückenhaftes Domänenwissen kompensiert werden kann, das nicht aus den Beispielmodellen extrahierbar ist.

Nachdem bei den Lösungen der beiden vorherigen Anforderungen Meta-Modelle automatisiert erzeugt und angepasst werden, sollten auch automatisierte Mechanismen bereitstehen, die die Qualität dieser Meta-Modelle berücksichtigen. Dieser Thematik widmet sich **Anforderung 3**. Diesbezügliche Lösungen haben die Gestalt, dass sie geänderte Meta-Modelle im Hintergrund analysieren und dabei nach Modellfragmenten mit Optimierungsbedarf suchen. Werden derartige Fragmente identifiziert, dann erfolgt die Ausgabe möglicher Verfeinerungsoptionen in Form von Empfehlungen. Die Durchführung einer spezifischen Verfeinerung obliegt aber weiterhin dem Benutzer, da nur er über den Sachverstand und das notwendige Domänenwissen verfügt. Begegnet wird Anforderung 3 in den Abschnitten, 4.4, 4.5.2, 5.5 und 7.9.

Als Ergebnis steht fest, dass alle in der Einleitung aufgestellten Anforderungen in vollem Umfang erfüllt werden. Die Umsetzbarkeit des erarbeiteten Konzepts wurde außerdem durch die auf der *Model Workbench* basierenden Implementierung (Kapitel 8) belegt.

9.2 Ausblick

Im Verlauf der Forschungstätigkeiten zur beispielgetriebenen Entwicklung von DSMLs sind diverse Fragestellungen aufgetreten, die außerhalb des Untersuchungsspektrums dieser Arbeit liegen. Dennoch sind die im Raume stehenden Fragen es wert, näher betrachtet zu werden, da sowohl konzeptuelle als auch technische Verbesserungen zu erwarten sind. Demgemäß wird nachfolgend auf potentiell zukünftige Arbeitsfelder im Rahmen der Entwicklung von Modellierungssprachen eingegangen, wobei sowohl konzeptuelle als auch technische Anknüpfungspunkte berücksichtigt werden. Letztere betreffen die in Kapitel 8 vorgestellte prototypische Implementierung, deren Basis die *Model Workbench* bildet.

- Zunehmend von Bedeutung ist die **Integration grafischer und textueller Sprachen** [139]. Dies ist mitunter daran erkennbar, dass Tool-Hersteller wie JetBrains die Unterstützung grafischer Modellierung in die Roadmap von MPS – einem Werkzeug, das ursprünglich für textuell-projizierendes Modellieren konzipiert war – aufgenommen haben [102]. Durch die gemeinsame Meta-Modellbasis (Abschnitt 5.2) ist unser Ansatz bereits für ein derartiges Vorhaben vorbereitet. Allerdings ist zukünftig sowohl ein **kombinierter Modellierungsworkflow** zu unterstützen als auch ein Editor zu entwickeln, der beide Welten miteinander vereint.
- Derzeit existieren zwei Vorgehensweisen zur Abbildung einer konkreten Syntax auf eine abstrakte Syntax. Die erste läuft beim Ableiten der abstrakten Syntax gemäß Abschnitt 5.3.2 vollständig automatisiert ab, was aufgrund des Induktionsprinzips immer in 1-zu-1-Mappings resultiert. Die zweite Variante erfordert ein komplett händisches Festlegen der Abbildungen durch den Benutzer, was zum einen zeitaufwändig ist und zum anderen die Gefahr von Fehlern in sich birgt. Deshalb ist es insbesondere für **komplexe Mappings** (die es obendrein noch zu spezifizieren gilt) zweckdienlich, eine angemessene **Werkzeugunterstützung** bereitzustellen, so dass das Fehlerrisiko und der zeitliche Faktor reduziert werden.
- Die Umsetzung der freien Modellierung im textuell ontologischen Kontext auf die in Kapitel 7 präsentierte Weise ist gänzlich neuartig. Wegen der feingranularen Eingriffsmöglichkeiten funktioniert sie prinzipiell für sämtliche Modellierungssprachen. Allerdings sollte die Anzahl manueller Umbaumaßnahmen so gering wie möglich ausfallen, damit der Schreibfluss des Benutzers nicht ausgebremst wird. Dies hängt allerdings stark von der jeweiligen Sprachart und den darin verwendeten Symbolen ab. Mit der Standardkonfiguration aus Abschnitt 7.4.1.2 funktioniert es gut für DSMLs, die anhand ihrer syntaktischen Struktur grob dem Stammbaum der auf C basierenden Sprachen zugerechnet werden können. Anders verhält es sich aber beispielsweise mit Template-Sprachen, wie schon in Abschnitt 7.8 erwähnt. Für eine umfassende, intuitive Unterstützung der **beispielgetriebenen Entwicklung textueller DSMLs** empfiehlt es sich daher, **weitere Sprachklassen** sowie deren spezifische Tokentypen zu identifizieren. Nachdem spezifische Features nur in bestimmten Sprachklassen benötigt werden, bietet es sich an, die Erweiterungen in Form von Profilen zu organisieren. Daraus kann der Benutzer schließlich das für seinen Anwendungsfall am besten geeignete Profil auswählen und bekommt dabei nur die Funktionalitäten zur Verfügung gestellt, die er auch wirklich benötigt.

- Da eine textuelle DSML typischerweise über ein Typsystem verfügt, muss dieses bei der Schaffung einer neuen DSML ebenfalls spezifiziert werden. Das ist eine aufwändige Aufgabe, die mit viel Programmierarbeit verbunden ist. In Bezug auf die beispielgetriebene Entwicklung von Modellierungssprachen wirft das die Frage auf, inwieweit auch **Typsysteme mithilfe von Beispielen konstruiert** werden können. Ist dies nicht möglich, sollte zumindest versucht werden, den Anteil an Programmierarbeit auf ein Minimum zu reduzieren, um mit möglichst geringem Aufwand zu einer vollständigen DSML zu gelangen.
- Ein gleichermaßen wichtiger Aspekt im textuell ontologischen Kontext ist es, ein **komfortables Konfigurieren** der Super-Grammatiken zu ermöglichen. Hilfreich ist das z.B. bei der Festlegung eigener binärer Operatoren inklusive deren Präzedenz. Aktuell müssen dazu die betreffenden Definitionsmodelle händisch vom Benutzer angepasst werden. Analoges trifft für die Konfigurationsmöglichkeiten der Ableitungsalgorithmen zu.
- Zusätzlich sollte durch externe Studien die **Parametrierung der Konfigurationen** verbessert werden, die näher die Gegebenheiten von realen Szenarien und Anwendungsfällen widerspiegelt. Die derzeitige Parametrierung fußt auf den Erfahrungen von Studenten und Mitarbeitern des Lehrstuhls für Angewandte Informatik IV der Universität Bayreuth.
- Zum **Ermitteln von Constraints** existieren bereits erste Ansätze [87, 155], die auf Beispielmodellen beruhen und bei den verwandten Arbeiten (Abschnitte 3.4.1 und 3.4.4) evaluiert wurden. Sie in die *Model Workbench* zu integrieren ist einer unserer nächsten Schritte. Doch auch hier gibt es noch weiteren Forschungsbedarf, dem nachgegangen werden sollte.
- **Natural Language Processing** (NLP, dt. „Verarbeitung natürlicher Sprache") ist ein Forschungsgebiet, das verstärkt Einzug in die domänenspezifische Modellierung nimmt [130]. Ziel hierbei ist es, formale Modelle aus natürlich sprachlichen Texten zu generieren, wodurch im Optimalfall das manuelle Modellieren überflüssig wird oder zumindest weniger Zeit in Anspruch nimmt. Wichtig dabei ist allerdings, dass bereits eine Modellierungssprache zugrunde liegt, in der die Modelle formuliert werden können. Ein zweistufiges Ableiten einer DSML anhand gegebener, natürlich sprachlicher Texte ist daher nicht ohne weiteres möglich. Dennoch existieren im Rahmen von NLP verschiedene Techniken, deren Anwendung zu einer **qualitativen Verbesserung der automatisiert generierten Meta-Modellartefakte** führen kann. Welche das im Einzelnen sind, sollte in anknüpfenden Forschungsarbeiten eruiert werden. Ein Beispiel ist die inhaltliche Auswertung von Textfragmenten unter Zuhilfenahme eines lexikalisch-semantischen Netzes wie WordNet [148], so dass auf diesem Weg geeignetere Bezeichnungen für Attribute bestimmt werden können.

Die Liste der hier aufgeführten Punkte erhebt keinen Anspruch auf Vollständigkeit. Sie soll jedoch verdeutlichen, dass es zahlreiche Herausforderungen gibt, die es im Kontext der beispielgetriebenen Entwicklung von DSMLs noch zu lösen gilt. Integriert in ein Werkzeug (z.B. die *Model Workbench*) stellen diesbezügliche Lösungen zusammen mit dem vorgestellten Ansatz einen alternativen, ganzheitlichen Modellierungsprozess bereit, der neben der Verwendung einer Modellierungssprache auch deren Definition und Evolution unterstützt.

Abbildungsverzeichnis

Listings

Tabellenverzeichnis

Literaturverzeichnis

[1] Aho, A. V., Lam, M.S., Sethi, R. and Ullman, J.D. 2008. *Compiler*. Pearson Studium.

[2] Alexander, A. 2013. *Scala Cookbook: Recipes for Object-Oriented and Functional Programming*. O'Reilly Media.

[3] Allen, J.F. 2003. Natural language processing. *Encyclopedia of Computer Science*. John Wiley and Sons Ltd. 1218–1222.

[4] Anaby-Tavor, A., Amid, D. and Fisher, A. 2009. An Empirical Study of Enterprise Conceptual Modeling. *Proceedings of the 28th International Conference on Conceptual Modeling* (Gramado, Brazil, 2009), 55–69.

[5] Anaby-Tavor, A., Amid, D., Fisher, A., Ossher, H., Bellamy, R., Callery, M., Desmond, M., Krasikov, S., Roth, T., Simmonds, I. and de Vries, J. 2009. An Algorithm for Identifying the Abstract Syntax of Graph-Based Diagrams. *IEEE Symposium on Visual Languages and Human-Centric Computing* (Corvallis, OR, USA, 2009), 193–196.

[6] Angluin, D. 1987. Learning Regular Sets from Queries and Counterexamples. *Information and Computation*. 75, 2 (1987), 87–106.

[7] Antkiewicz, M., Bąk, K., Zayan, D., Czarnecki, K., Wąsowski, A. and Diskin, Z. 2013. Example-Driven Modeling Using Clafer. *Proceedings of the 1st International Workshop on Model-driven Engineering By Example* (Miami, FL, USA, 2013), 32–41.

[8] Atkinson, C. 1997. Meta-Modeling for Distributed Object Environments. *Proceedings of the Enterprise Distributed Object Computing Workshop (EDOC '97)* (Gold Coast, Queensland, Australia, 1997), 90–101.

[9] Atkinson, C. and Kühne, T. 2005. Concepts for Comparing Modeling Tool Architectures. *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems* (Montego Bay, Jamaica, 2005), 398–413.

[10] Atkinson, C. and Kühne, T. 2000. Meta-level Independent Modelling. *Proceedings of the International Workshop on Model Engineering at 14th European Conference on Object-Oriented Programming* (Cannes, France, 2000), 12–16.

[11] Baar, T. 2006. Correctly Defined Concrete Syntax for Visual Modeling Languages. *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems* (Genova, Italy, 2006), 111–125.

[12] Backus, J.W. 1959. The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference. *Proceedings of the International Conference on Information Processing* (Paris, France, 1959), 125–132.

[13] Bertoa, M.F. and Vallecillo, A. 2010. Quality attributes for software metamodels. *Proceedings of the 13th TOOLS Workshop on Quantitative Approaches in Object-Oriented Software Engineering* (Malaga, Spain, 2010).

[14] Bloch, J. 2008. *Effective Java*. Addison-Wesley Longman.

[15] Bornat, R. 1979. *Understanding and Writing Compilers:: A do-it-yourself guide*. Macmillan Press Ltd.

[16] Brainfuck: 1993. *http://www.muppetlabs.com/~breadbox/bf/*. Accessed: 2014-06-16.

[17] Bundy, A. and Wallen, L. 1984. Context-Free Grammar. *Catalogue of Artificial Intelligence Tools*. Springer Berlin Heidelberg. 22–23.

[18] Checkland, P. and Scholes, J. 1999. *Soft Systems Methodology in Action*. John Wiley and Sons, Inc.

[19] Chen, P.P.S. 1976. The entity-relationship model - toward a unified view of data. *ACM Transactions on Database Systems (TODS)*. 1, 1 (1976), 9–36.

[20] Cheng, J., Yu, J.X., Ding, B., Yu, P.S. and Wang, H. 2008. Fast Graph Pattern Matching. *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*. IEEE.

[21] Cho, H. 2013. *A Demonstration-Based Approach for Domain-Specific Modeling Language Creation*. PhD Thesis, Department of Computer Science, University of Alabama.

[22] Cho, H. and Gray, J. 2011. Design Patterns for Metamodels. *Proceedings of the SPLASH '11 Workshops* (Portland, OR, USA, 2011), 25–32.

[23] Cho, H., Gray, J., Sun, Y. and White, J. 2011. Key Challenges for Modeling Language Creation By Demonstration. *ICSE 2011 Workshop on Flexible Modeling* (Waikiki, Honolulu, Hawaii, 2011).

[24] Clark, T., Sammut, P. and Willans, J. 2008. *Applied Metamodelling: A Foundation for Language Driven Development*. CETEVA.

[25] Common Lisp the Language, 2nd Edition: 1994. *http://www-prod-gif.supelec.fr/docs/cltl/clm/clm.html*. Accessed: 2014-06-16.

[26] Cornett, S. 2004. The Usability of Massively Multiplayer Online Roleplaying Games : Designing for New Users. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Vienna, Austria, 2004), 703–710.

[27] Correia, F.F. and Aguiar, A. 2013. Patterns of Flexible Modeling Tools. *Proceedings of the 20th Conference on Pattern Languages of Programs* (Monticello, IL, USA, 2013), 18.

[28] Costagliola, G., Delucia, A., Orefice, S. and Polese, G. 2002. A Classification Framework to Support the Design of Visual Languages. *Visual Languages & Computing*. 13, 6 (2002), 573–600.

[29] CSS 2.1 - Syntax and basic data types: 2011. *http://www.w3.org/TR/CSS2/syndata.html*. Accessed: 2014-06-16.

[30] Davies, I., Green, P., Rosemann, M., Indulska, M. and Gallo, S. 2006. How do practitioners use conceptual modeling in practice? *Data & Knowledge Engineering* (2006), 358–380.

[31] Desmond, M., Ossher, H., Simmonds, I., Amid, D., Anaby-Tavor, A., Callery, M. and Krasikov, S. 2010. Towards smart office tools. *SPLASH 2010 Workshop on Flexible Modeling Tools* (Reno, Nevada, 2010).

[32] Van Deursen, A., Klint, P. and Visser, J. 2000. Domain-Specific Languages : An Annotated Bibliography. *ACM SIGPLAN Notices*. 35, 6 (2000), 26–36.

[33] Dmitriev, S. 2004. Language oriented programming: The next programming paradigm. *JetBrains onBoard*. (2004).

[34] DuPont, A.J.C. 1969. *Strategy and Structure: Chapters in the History of the American Industrial Enterprise*. MIT Press.

[35] Eclipse Modeling Framework Project (EMF): 2014. *http://www.eclipse.org/modeling/emf/*. Accessed: 2014-06-16.

[36] Ehrig, H., Ehrig, K., Prange, U. and Taentzer, G. 2006. *Fundamentals of Algebraic Graph Transformation*. Springer.

[37] Emerson, M. and Sztipanovits, J. 2006. Techniques for Metamodel Composition. *Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling* (Portland, OR, USA, 2006), 123–139.

[38] Fan, W., Li, J., Ma, S., Tang, N., Wu, Y. and Wu, Y. 2010. Graph Pattern Matching: From Intractable to Polynomial Time. *Proceedings of the VLDB Endowment*. 3, 1 (2010), 264–275.

[39] Flanagan, D. 2011. *JavaScript: The Definitive Guide*. O'Reilly Media, Inc.

[40] Floridi, L. and Sanders, J.W. 2004. *Levellism and the Method of Abstraction*. Research Report, Information Ethics Group.

[41] Fondement, F. 2007. *Concrete Syntax Definition for Modeling Languages*. PhD Thesis, Faculté Informatique et Communications, École Polytechnique Fédérale de Lausanne.

[42] Forlines, C., Wigdor, D., Shen, C. and Balakrishnan, R. 2007. Direct-touch vs. mouse input for tabletop displays. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (San Jose, CA, USA, 2007), 647–656.

[43] Fowler, M. 2011. *Domain-Specific Languages*. Addison-Wesley Professional.

[44] Fowler, M. 2002. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional.

[45] Frank, U. 2011. Some Guidelines for the Conception of Domain-Specific Modelling Languages. *EMISA*. 190, GI (2011), 93–106.

[46] Friedman, D.P. and Wand, M. 2008. *Essentials of Programming Languages*. MIT Press.

[47] Fu, J. 1995. Pattern Matching in Directed Graphs. *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching* (Espoo, Finland, 1995), 64–77.

[48] Fuentes-Fernández, L. and Vallecillo-Moreno, A. 2004. An introduction to UML profiles. *UML and Model Engineering*. 5, 2 (2004), 6–13.

[49] Gabrysiak, G., Giese, H., Lüders, A. and Seibel, A. 2011. How Can Metamodels Be Used Flexibly? *ICSE 2011 Workshop on Flexible Modeling* (Waikiki, Honolulu, Hawaii, 2011).

[50] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

[51] GEF (Graphical Editing Framework): 2014. *http://www.eclipse.org/gef/*. Accessed: 2014-06-16.

[52] Gold, E.M. 1967. Language Identification in the Limit. *Information and control*. 10, 5 (1967), 447–474.

[53] Gosling, J., Joy, B., Steele, G. and Bracha, G. 2005. *The Java Language Specification*. Addison-Wesley.

[54] Göttler, H. 1987. Graph Grammars and Diagram Editing. *Proceedings of the 3rd International Workshop on Graph Grammars and their Application to Computer Science* (Warrenton, VA, USA, 1987), 216–231.

[55] Graphical Modeling Project: 2014. *http://www.eclipse.org/modeling/gmp/*. Accessed: 2014-06-16.

[56] Greenfield, J. 2001. *UML Profile For EJB*. Public Draft, Rational Software Corporation.

[57] Grönniger, H., Krahn, H., Rumpe, B., Schindler, M. and Völkel, S. 2008. MontiCore : A Framework for the Development of Textual Domain Specific Languages. *Companion of the 30th International Conference on Software Engineering* (Leipzig, Germany, 2008), 925–926.

[58] Halbert, D.C. 1984. *Programming by Example*. PhD Thesis, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley.

[59] Han, C. 2008. *The Building of A Micro Network Modeling Toolkit by Marama*. Project Report, Mcgill University, Montreal, Canada.

[60] Heidenreich, F., Johannes, J., Karol, S., Seifert, M. and Wende, C. 2009. Derivation and Refinement of Textual Syntax for Models. *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications* (Enschede, The Netherlands, 2009), 114–129.

[61] Henderson-Sellers, B. and Gonzalez-Perez, C. 2005. The Rationale of Powertype-based Metamodelling to Underpin Software Development Methodologies. *Proceedings of the 2nd Asia-Pacific conference on Conceptual modelling* (Newcastle, Australia, 2005), 7–16.

[62] Herrmannsdoerfer, M., Vermolen, S.D. and Wachsmuth, G. 2010. An Extensive Catalog of Operators for the Coupled Evolution of Metamodels and Models. *Proceedings of the 3rd International Conference on Software Lanugage Engineering* (Eindhoven, Netherlands, 2010), 163–182.

[63] HTML5: 2013. *http://www.w3.org/TR/html5/*. Accessed: 2014-06-16.

[64] Jablonski, S. and Bussler, C. 1996. *Workflow Management: Modeling Concepts, Architecture and Implementation*. International Thomson Computer Press.

[65] Jablonski, S., Volz, B. and Dornstauder, S. 2009. On the Implementation of Tools for Domain Specific Process Modelling. *Proceedings of the 4th International Conference on Evaluation of Novel Approaches to Software Engineering* (Milan, Italy, 2009), 109–120.

[66] Jahn, M. 2014. *Evolution von Meta-Modellen mit sprachbasierten Mustern*. PhD Thesis, Fakultät für Mathematik, Physik und Informatik, Universität Bayreuth.

[67] Jahn, M., Roth, B. and Jablonski, S. 2014. Instance Specialization – a Pattern for Multi-level Meta Modelling. *Proceedings of the 1st International Workshop on Multi-Level Modelling (MULTI 2014, in conjunction with MODELS 2014)* (Valencia, Spain, 2014), 23–32.

[68] Jardine, D.A. 1977. *The ANSI/SPARC DBMS model*. Elsevier Science Inc.

[69] Javed, F., Bryant, B.R. and Sprague, A. 2005. Extracting Grammar from Programs : Evolutionary Approach. *ACM SIGPLAN Notices*. 40, 4 (2005), 39–46.

[70] Javed, F., Mernik, M., Gray, J. and Bryant, B.R. 2008. MARS: A metamodel recovery system using grammar inference. *Information and Software Technology*. 50, 9-10 (2008), 948–968.

[71] Johnson, R. and Woolf, B. 1997. The Type Object Pattern. *Pattern Languages of Program Design 3*. Addison-Wesley Longman. 47–65.

[72] JointJS - the HTML 5 JavaScript diagramming library: 2014. *http://www.jointjs.com/*. Accessed: 2014-06-16.

[73] Karsai, G., Rumpe, B., Schindler, M. and Völkel, S. 2009. Design Guidelines for Domain Specific Languages. *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling* (Orlando, FL, USA, 2009), 7–13.

[74] Kats, L.C.L. and Visser, E. 2010. The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Reno, NV, USA, 2010), 444–463.

[75] Kelly, S. and Pohjonen, R. 2013. Dynamic Symbol Templates and Ports in MetaEdit+. *Proceedings of the 2013 ACM Workshop on Domain-Specific Modeling* (Indianapolis, IN, USA, 2013), 19–20.

[76] Kelly, S. and Tolvanen, J.-P. 2008. *Domain-Specific Modeling: Enabling Full Code Generation*. John Wiley & Sons.

[77] Ķikusts, P. and Ručevskis, P. 1996. Layout Algorithms of Graph-Like Diagrams for GRADE Windows Graphic Editors. *Graph Drawing*. 1027, (1996), 361–364.

[78] Kouhen, A. El, Dumouliin, C., Gérard, S. and Boulet, P. 2013. A Component-Based Approach for Specifying DSML's Concrete Syntax. *Proceedings of the 2nd Workshop on Graphical Modeling Language Development* (Montpellier, France, 2013), 3–11.

[79] Krahn, H. 2009. *MontiCore: Agile Entwicklung von domäanenspezifischen Sprachen im Software-Engineering*. PhD Thesis, Fakultät für Mathematik, Informatik und Naturwissenschaften, RWTH Aachen.

[80] Krahn, H., Rumpe, B. and Steven, V. 2007. Integrated Definition of Abstract and Concrete Syntax for Textual Languages. *Proceedings of the ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems* (Nashville, TN, USA, 2007), 286–300.

[81] Kurtev, I., Bézivin, J., Jouault, F., Valduriez, P. and Inria, A. 2006. Model-based DSL Frameworks. *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages and Applications* (Portland, OR, USA, 2006), 602–615.

[82] Langlois, B., Jitia, C.-E. and Jouenne, E. 2007. DSL Classification. *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling* (Montréal, Canada, 2007).

[83] De Lara, J. and Vangheluwe, H. 2002. AToM3: A Tool for Multi-formalism and Meta-modelling. *Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering* (Grenoble, France, 2002), 174–188.

[84] Ledeczi, a., Nordstrom, G., Karsai, G., Volgyesi, P. and Maroti, M. 2001. On Metamodel Composition. *Proceedings of the 2001 IEEE International Conference on Control Applications* (Mexico City, Mexico, 2001), 756–760.

[85] Liskov, B. 1988. Data Abstraction and Hierarchy. *ACM SIGPLAN Notices*. 23, 5 (1988), 17–34.

[86] Liu, Q., Bryant, B.R. and Mernik, M. 2010. Metamodel Recovery from Multi-Tiered Domains Using Extended MARS. *Proceedings of the 34th IEEE Annual Computer Software and Applications Conference* (Seoul, South Korea, 2010), 279–288.

[87] López-Fernández, J.J., Cuadrado, J.S., Guerra, E. and de Lara, J. 2013. Example-driven meta-model development. *Software & Systems Modeling*. (2013), 1–25.

[88] Lorenzen, P. 1987. *Lehrbuch der konstruktiven Wissenschaftstheorie*. Bibliographisches Institut.

[89] Ma, Z., He, X. and Liu, C. 2013. Assessing the quality of metamodels. *Frontiers of Computer Science*. 7, 4 (2013), 558–570.

[90] Manning, C.D. and Schütze, H. 1999. *Foundations of Statistical Natural Language Processing*. MIT Press.

[91] Marama Meta-Tools: *https://wiki.auckland.ac.nz/display/csidst/Welcome*. Accessed: 2014-06-16.

[92] Mernik, M., Heering, J. and Sloane, A.M. 2005. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*. 37, 4 (2005), 316–344.

[93] Mernik, M., Hrncic, D., Bryant, B.R., Sprague, A.P., Gray, J., Liu, Q. and Javed, F. 2009. Grammar Inference Algorithms and Applications in Software Engineering. *22th International Symposium on Information, Communication and Automation Technologies* (Sarajevo, Bosnia, 2009), 1–7.

[94] Meta Programming System: 2014. *http://www.jetbrains.com/mps/*. Accessed: 2014-06-16.

[95] metaBUP: 2013. *http://jesusjlopezf.wordpress.com/2013/06/13/metabup/*. Accessed: 2014-06-16.

[96] MetaEdit+ Domain-Specific Modeling (DSM) environment: 2014. *http://www.metacase.com/products.html*. Accessed: 2014-06-16.

[97] Minas, M. 2002. Concepts and Realization of a Diagram Editor Generator Based on Hypergraph Transformation. *Science of Computer Programming*. 44, 2 (2002), 157–180.

[98] Minas, M. 2006. Generating Meta-Model-Based Freehand Editors. *Proceedings of the 3rd International Workshop on Graph Based Tools* (Natal, Brazil, 2006), 1–13.

[99] Minas, M. 2006. Syntax Definition with Graphs. *Electronic Notes in Theoretical Computer Science*. 148, 1 (Feb. 2006), 19–40.

[100] Mittelstraß, J. 2010. Intuitionismus. *Enzyklopädie Philosophie und Wissenschaftstheorie*. J.B. Metzler.

[101] Modeling SDK for Visual Studio - Domain Specific Languages: 2013. *http://msdn.microsoft.com/en-us/library/bb126259.aspx*. Accessed: 2014-06-16.

[102] MPS public roadmap: 2014. *http://confluence.jetbrains.com/display/MPS/MPS+public+roadmap*. Accessed: 2014-06-16.

[103] Neumayr, B., Schrefl, M. and Thalheim, B. 2011. Modeling Techniques for Multi-Level Abstraction. *The evolution of conceptual modeling*. 6520, (2011), 68–92.

[104] Nielsen, J. 1994. *Usability Engineering*. Morgan Kaufmann.

[105] Noble, J., Taivalsaari, A. and Moore, I. 1999. *Prototype-Based Programming: Concepts, Languages and Applications*. Springer.

[106] Nystrom, N., Chong, S. and Myers, A.C. 2004. Scalable Extensibility via Nested Inheritance. *ACM SIGPLAN Notices*. 39, 10 (Oct. 2004), 99–115.

[107] Ober, I. and Prinz, A. 2006. What do we need metamodels for? *Proceedings of 4th Nordic Workshop on UML and Software Modelling* (Grimstad, Norway, 2006), 8–23.

[108] Odell, J.J. 1994. Power Types. *Journal of Object-Oriented Programming*. 7, 2 (1994), 8–12.

[109] OMG 2014. Case Management Model and Notation. 1.0.

[110] OMG 2013. OMG Meta Object Facility (MOF) Core Specification. 2.4.1.

[111] OMG 2011. OMG Unified Modeling Language (OMG UML), Infrastructure. 2.4.1.

[112] Ortner, E. 2005. *Sprachbasierte Informatik: wie man mit Wörtern die Cyber-Welt bewegt*. Edition am Gutenbergplatz.

[113] Ossher, H., Bellamy, R., Simmonds, I., Amid, D., Anaby-Tavor, A., Callery, M., Desmond, M., de Vries, J., Fisher, A. and Krasikov, S. 2010. Flexible Modeling Tools for Pre-Requirements Analysis: Conceptual Architecture and Research Challenges. *ACM SIGPLAN Notices*. 45, 10 (2010), 848–864.

[114] Paige, R.F., Ostroff, J.S. and Brooke, P.J. 2000. Principles for Modeling Language Design. *Information and Software Technology*. 42, 10 (2000), 665–675.

[115] Parr, T. 2010. *Language Implementation Patterns*. The Pragmatic Bookshelf.

[116] Patterson, D.A. and Hennessy, J.L. 2008. *Computer Organization and Design*. Morgan Kaufmann.

[117] Ráth, I., Ökrös, A. and Varró, D. 2009. Synchronization of abstract and concrete syntax in domain-specific modeling languages. *Software and Systems Modeling*. 9, 4 (2009), 453–471.

[118] Rekers, J. and Schürr, A. 1996. A graph based framework for the implementation of visual environments. *Proceedings of the IEEE Symposium on Visual Languages* (Boulder, CO, USA, 1996), 148–155.

[119] Roth, B. 2010. *Konzeption und Implementierung eines generischen Modellierungswerkzeugs zur Unterstützung der domänenspezifischen Prozessmodellierung*. Master Thesis, Fakultät für Mathematik, Phsyik und Informatik, Universität Bayreuth.

[120] Roth, B., Jahn, M. and Jablonski, S. 2013. A Method for Directly Deriving a Concise Meta Model from Example Models. *Proceedings of the 5th International Conferences on Pervasive Patterns and Applications* (Valencia, Spain, 2013), 52–58.

[121] Roth, B., Jahn, M. and Jablonski, S. 2013. On the Way of Bottom-Up Designing Textual Domain-Specific Modelling Languages. *Proceedings of the 2013 ACM Workshop on Domain-Specific Modeling* (Indianapolis, IN, USA, 2013), 51–55.

[122] Roth, B., Jahn, M. and Jablonski, S. 2014. Rapid Design of Meta Models. *International Journal on Advances in Software*. 7, 1&2 (2014), 31–43.

[123] Rumbaugh, J. and Eddy, F. 1991. *Object-Oriented Modeling and Design*. Prentice Hall.

[124] Sánchez-Cuadrado, J., Lara, J. De and Guerra, E. 2012. Bottom-Up Meta-Modelling: An Interactive Approach. *Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems* (Innsbruck, Austria, 2012), 3–19.

[125] Schneider, T. 2013. *Domänenspezifische Evaluation und Optimierung von Datenstandards und Infrastrukturen*. PhD Thesis, Fakultät für Mathematik, Physik und Informatik, Universität Bayreuth.

[126] Seidewitz, E. 2003. What models mean. *IEEE Software*. 20, 5 (2003), 26–32.

[127] Seven Principles Of Software Development: 1996. *http://c2.com/cgi/wiki?SevenPrinciplesOfSoftwareDevelopment*. Accessed: 2014-06-16.

[128] Simonyi, C. and Clifford, S. 2006. Intentional Software. *Proceedings of the 21st annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications* (Portland, OR, USA, 2006), 451–464.

[129] Singh, G. 1994. Single Versus Multiple Inheritance in Object Oriented Programming. *ACM SIGPLAN OOPS Messenger*. 6, 1 (1994), 30–39.

[130] Soderland, S., Roof, B., Qin, B., Xu, S., Mausam and Etzioni, O. 2010. Adapting Open Information Extraction to Domain-Specific Relations. *AI Magazine*. 31, 3 (2010), 93–102.

[131] Spinellis, D. 2001. Notable design patterns for domain-specific languages. *Journal of Systems and Software*. 56, 1 (2001), 91–99.

[132] Stahl, T., Völter, M., Efftinge, S. and Haase, A. 2007. *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. Dpunkt Verlag.

[133] Stevenson, A. and Cordy, J.R. 2012. Grammatical Inference in Software Engineering: an Overview of the State of the Art. *Proceedings of the Fifth International Conference on Software Language Engineering* (Dresden, Germany, 2012), 204–223.

[134] Stroustrup, B. 1997. *The C++ Programming Language*. Addison-Wesley.

[135] The Python Language Reference: 2014. *http://docs.python.org/3/reference/index.html*. Accessed: 2014-06-16.

[136] Tolvanen, J. and Kelly, S. 2009. MetaEdit+: Defining and Using Integrated Domain-Specific Modeling Languages. *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications* (Orlando, FL, USA, 2009), 819–820.

[137] Valiant, L.G. 1984. A Theory of the Learnable. *Communications of the ACM*. 27, 11 (1984), 1134–1142.

[138] Visual Paradigm for UML 11: 2014. *http://www.visual-paradigm.com/product/vpuml/*. Accessed: 2014-06-16.

[139] Voelter, M. 2013. *DSL Engineering*. CreateSpace Independent Publishing Platform.

[140] Voelter, M., Ratiu, D., Schaetz, D. and Kolb, B. 2012. mbeddr: an Extensible C-based Programming Language and IDE for Embedded Systems. *Proceedings of the 3rd Annual conference on Systems, Programming, and Applications: Software for Humanity* (Tucson, AZ, USA, 2012), 121–140.

[141] Vogler, C. 2013. *Werkzeugunterstützung für die Definition grafischer Modellierungskonstrukte*. Bachelor Thesis, Fakultät für Mathematik, Phsyik und Informatik, Universität Bayreuth.

[142] Volz, B. 2011. *Werkzeugunterstützung für methodenneutrale Metamodellierung*. University of Bayreuth.

[143] Wedekind, H., Ortner, E. and Inhetveen, R. 2004. Informatik als Grundbildung. Teil IV: Objektsprache/Metasprache. *Informatik Spektrum*. Springer.

[144] Weik, M. 2001. *Computer Science and Communications Dictionary*. Kluwer Academic Publishers.

[145] Welcome to Hyun's Homepage: 2012. *http://hcho7.students.cs.ua.edu/*. Accessed: 2014-06-16.

[146] Whitespace: 2004. *http://compsoc.dur.ac.uk/whitespace/*. Accessed: 2014-06-16.

[147] Wile, D.S. 1997. Abstract Syntax from Concrete Syntax. *Proceedings of the 19th International Conference on Software Engineering* (Boston, MA, USA, 1997), 472–480.

[148] WordNet: 2013. *http://wordnet.princeton.edu/*. Accessed: 2014-06-16.

[149] Working with the Symbol Editor: 2014. *http://www.metacase.com/support/50/manuals/mwb/Mw-3_1.html*. Accessed: 2014-06-16.

[150] Wüest, D., Seyff, N. and Glinz, M. 2013. Semi-automatic Generation of Metamodels from Model Sketches. *Proceedings of the IEEE/ACM 28th International Conference on Automated Software Engineering* (Silicon Valley, CA, USA, 2013), 664–669.

[151] Wüest, D., Seyff, N. and Glinz, M. 2013. Von der Idee zum Anforderungsmodell ohne Medienbruch. *Softwaretechnik-Trends*. 33, 1 (May. 2013), 21–22.

[152] WYSIWYG: definition of WYSIWYG in Oxford dictionary (British & World English): *http://www.oxforddictionaries.com/definition/english/WYSIWYG*. Accessed: 2014-06-16.

[153] Xtext - Language Development Made Easy!: 2013. *http://www.eclipse.org/Xtext*. Accessed: 2014-06-16.

[154] Xtext Documentation: 2013. *http://www.eclipse.org/Xtext/documentation.html#metamodelInference*. Accessed: 2014-06-16.

[155] Zayan, D., Czarnecki, K. and Rayside, D. 2013. Example-Driven Modeling. *Proceedings of the 35th International Conference on Software Engineering* (San Francisco, CA, USA, 2013), 1273–1276.

[156] Zeising, M., Schönig, S. and Jablonski, S. 2014. Towards a Common Platform for the Support of Routine and Agile Business Processes. *Proceedings of the 10th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing* (Miami, FL, USA, 2014), 10.

[157] Zhu, N., Grundy, J. and Hosking, J. 2005. Constructing domain-specific design tools with a visual language meta-tool. *Proceedings of the 17th Conference on Advanced Information Systems Engineering* (Porto, Portugal, 2005), 139–144.