Zustandstypen als Schlüssel zur Integration statischer und dynamischer Sichten der objektorientierten Modellierung

Friedrich Steimann

Lehrgebiet Programmiersysteme
Fernuniversität in Hagen, 58084 Hagen
steimann@acm.org

Thomas Kühne

Fachgebiet Metamodellierung
TU Darmstadt, 64289 Darmstadt
kuehne@informatik.tu-darmstadt.de

Abstract: Die objektorientierte Modellierung ist als eine Sammlung bewährter, aber nicht unbedingt zur kombinierten Verwendung bestimmter Notationen und Formalismen entstanden. So besteht bis heute ein Bedarf an Ansätzen zur Integration der verschiedenen, durch die Einzelformalismen gewährten Sichten auf ein modelliertes System. In dieser Arbeit präsentieren wir Zustandstypen als Bindeglied zwischen (statischen) Strukturspezifikationen und (dynamischen) Verhaltensspezifikationen. Wir zeigen, daß sich diese Zustandstypen auf natürliche Weise als Subtypen von Klassen ergeben, die ihr Verhalten über Zustandsautomaten spezifizieren und so erlauben, Assoziations- und Methodendeklarationen zustandsabhängig zu überschreiben. Daraus ergeben sich einige interessante Integrationsansätze, die wir skizzenhaft beschreiben.

1 Einleitung

Entwicklungsgeschichtlich bedingt bedient sich die objektorientierte Modellierung[1] in der Praxis keines Formalismus' „wie aus einem Guß", sondern verwendet ein Konglomerat verschiedenster, aus mehr oder weniger verwandten Disziplinen zusammengesuchter Einzelnotationen. Da diese Notationen nicht für die gemeinsame Verwendung entworfen wurden, sollte es nicht verwundern, daß sie nur bedingt zueinander passen.

Um die Komplexität zu reduzieren und ein fokussierte Betrachtung zu erlauben, möchte man Modellierung aus mehreren, möglichst unabhängigen Perspektiven oder Sichten betreiben. Zwar ist die Unabhängigkeit der Sichten eine wünschenswerte Eigenschaft einer Modellierungssprache, damit diese – ähnlich den Modulen eines Programms – getrennt voneinander entwickelt werden können. Das enthebt sie aber nicht von der Notwendigkeit, – wiederum wie die Module eines Programms – integrierbar zu sein, da sonst die Gesamtbedeutung aller Sichten undefiniert ist. Bei Programmen erfolgt die Integration durch gegenseitige Verwendung der Modulschnittstellen – in den Sichten eines Modells sind solche Schnittstellen aber nicht vorgesehen, weswegen sie „zusammengewoben" werden müssen. Dieses „Zusammenweben" erfordert gemeinsame Bezugspunkte und damit Notationselemente, die in mehreren Sichten vorkommen können.

[1] Wenn wir hier von objektorientierter Modellierung sprechen, dann meinen wir Modellierung nach der Fasson der UML und riskieren dabei, der einen oder anderen sauber definierten (akademischen) Modellierungssprache unrecht zu tun. Wir bitten, dies zu entschuldigen.

In dieser Arbeit schlagen wir vor, Zustandstypen also solche Notationselemente einzuführen, die spezielle, über Automaten definierte Subtypen von Klassen sind und so verschiedene Sichten miteinander verwebbar machen. Dazu wird zunächst (in Abschnitt 2) auf die grundsätzliche Unterscheidung verschiedener Arten von Typen eingegangen und Zustandstypen als eine solche Art motiviert. Abschnitt 3 definiert dann Zustandstypen und klärt deren Verhältnis zu Klassen. Abschnitt 4 beleuchtet den Zusammenhang von Zustandstypen und Automaten. Abschnitt 5 zeigt auf, wie sich unter Zuhilfenahme von Zustandstypen auf einfache Weise Modelle spezifizieren lassen, deren statische und dynamische Sicht automatisch verwoben werden können. In Abschnitt 6 schließlich diskutieren wir verwandte Arbeiten und offenen Fragen.

2 Verschiedene Arten von Typen für die Modellierung

In [St00a, St00b, St02] wurde das Konzept der Rolle in Modellierung und Programmierung ausführlich diskutiert und, im wesentlichen [Gu92] folgend, von sogenannten natürlichen Typen (Klassen) abgegrenzt. Kurz gefaßt lassen sich Rolle und Klasse mittels zweier primitiver, ontologischer Klassifikationseigenschaften unterscheiden: 1.) Fundierung und 2.) Rigidität, oder, etwas selbsterklärender ausgedrückt: 1.) Abhängigkeit vom Bestehen einer Beziehung und 2.) Unveränderlichkeit einer Klassifizierung. Abbildung 1 zeigt beide Eigenschaften mit ihren möglichen Werten als Vierfeldertabelle; Klassen sind demnach rigide, aber nicht fundiert, Rollen sind fundiert, aber nicht rigide. Im Klartext bedeutet dies: Ein Objekt kann Instanz einer Klasse sein, ohne dazu zu einem anderen in Beziehung stehen zu müssen (Anti-Fundierung; Teil-Ganzes-Beziehungen und Attributierungen sind jedoch zulässig), und kann nicht aufhören, Instanz dieser Klasse zu sein, ohne seine Identität zu verlieren (Rigidität). Das Objekt ist hingegen Instanz einer Rolle, wenn es dazu zu einem anderen (in einer Gegenrolle) in Beziehung stehen muß (Fundierung) und wenn es die Rolle jederzeit annehmen und wieder ablegen kann, ohne dabei seine Identität zu verlieren (Anti-Rigidität).

Klassifizierungseigenschaften		fundiert (beziehungsabhängig)	
		ja	nein
rigide (unveränderlich)	ja	(*Qualität*)	*Klasse*
	nein	*Rolle*	*Zustand*

Abbildung 1: Verschiedene Arten von Typen mit ihren Klassifizierungseigenschaften (s. Text).

Es stellt sich natürlich sofort die Frage, ob und wie die beiden verbleibenden Felder sinnvoll ausgefüllt werden können.

Diese Arbeit befaßt sich mit Typen, deren Klassifizierung die Eigenschaften „vorübergehend“ (anti-rigide) und „beziehungsunabhängig“ (anti-fundiert) aufweist. Ein Objekt kann also auch nur zeitweise unter eine solche Klassifikation fallen (von einem solchen Typ sein), ohne dadurch seine Identität zu verlieren. Gleichzeitig ist dazu keine Beziehung zu einem anderen Objekt notwendig – es kann die Klassifizierungsmerkmale ganz auf sich selbst gestellt erfüllen. *Jugendlicher* ist so ein Typ: Eine Person kann jugendlich werden und wieder aufhören zu sein, ohne dadurch ihre Identität zu verlieren, und jugendlich zu sein verlangt kein Verhältnis zur irgendeiner anderen Person oder Sache.

Wir werden solche Typen im folgenden Zustandstypen nennen; auf die vierte Art, die man als Qualität bezeichnen könnte, gehen wir in dieser Arbeit nicht ein (auch wenn der Zusammenhang von Attributierung eines Objektes mittels Qualitäten und seinem Zustand naheliegt).

3 Zustandstypen

Typen werden in der objektorientierten Modellierung dazu verwendet, von Aussagen, die konkrete Objekte betreffen, zu abstrahieren: Man spezifiziert gerne auf Typebene. Dabei bestehen Spezifikationen auf Typebene zum Gutteil aus Deklarationen. Im allgemeinen wird unterstellt, daß diese Deklarationen allgemeingültige Aussagen (im Sinne von Allquantifikationen) darstellen: Die Deklaration

$$\textit{drucken}: \textit{Drucker} \times \textit{Dokument} \tag{1}$$

beispielsweise suggeriert, daß alle Drucker mit allen Dokumenten gepaart werden können.[2] Dies ist jedoch schon in relativ einfachen Fällen nicht der Fall: Im gegebenen Beispiel könnte es z. B. sein, daß Texte nur auf Zeilendruckern ausgedruckt werden können und Diagramme nur auf Plottern. Dies würde in den zusätzlichen Deklarationen

$$\begin{aligned} &\textit{drucken}: \textit{Zeilendrucker} \times \textit{Text} \\ &\textit{drucken}: \textit{Plotter} \times \textit{Diagramm} \end{aligned} \tag{2}$$

ausgedrückt, die die vorausgehende ergänzen (überschreiben); man spricht hier auch von einer kovarianten Redefinition, die die ursprüngliche Deklaration in ihrer Allgemeingültigkeit einschränkt.[3] Dabei wird unterstellt, daß *Plotter* und *Zeilendrucker* Sub*klassen* von *Drucker* sind, genauso wie *Text* und *Diagramm* von *Dokument*. Entscheidend hierbei ist, daß das Drucken für andere als die angegebenen Paarungen von Druckern und Dokumenten nicht definiert ist. Die Situation ist in Abbildung 2 dargestellt.

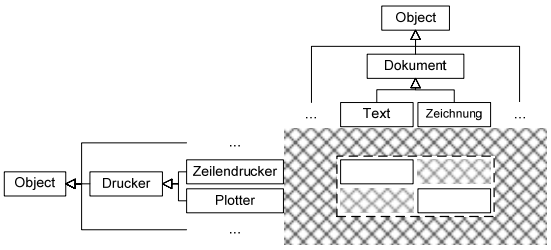


Abbildung 2: Darstellung der Relation *drucken* als kartesisches Produkt *Drucker* × *Dokument* (gestricheltes Rechteck) der statischen Extensionen der Klassentypen gemäß Deklaration (1) und als abschnittsweise Definition (*Zeilendrucker* × *Text*) ∪ (*Plotter* × *Zeichnung*) (weiße Rechtecke) gemäß Deklaration (2). Es gilt, alle schraffierten Bereiche vom Definitionsbereich auszuschließen.

---

[2] Wir deklarieren hier zunächst nur ganz allgemein Relationen. In der objektorientierten Modellierung würde man ggf. statt *drucken*: *Drucker* × *Dokument* auch *Drucker*.*drucken*(*Dokument*) für eine Methode oder *Drucker*.*drucken*: *Dokument* für ein Attribut schreiben. Wir bleiben hier jedoch vorerst bei der allgemeineren Relationenschreibweise, da sie Assoziationen, Methoden und Attribute subsumiert [SK02].

[3] Man könnte sagen, daß die Relation *drucken* durch die beiden Deklarationen in (2) abschnittsweise definiert ist; mehr dazu unten.

Ein auf den ersten Blick ähnlich gelagertes Beispiel ist das folgende: Selbst wenn sich grundsätzlich alle Dokumente auf allen Druckern drucken lassen, so ist ein Druck im konkreten Einzelfall u. U. nicht möglich, z. B. wenn der Drucker gerade leer ist. Dies führt dann entweder zu einem gegenüber der allgemeinen Erwartung veränderten Verhalten (statt eines Ausdrucks erhält man wilde Piepsignale o. ä.) oder ist einfach undefiniert (als unerlaubte Aktion vorgesehen). Wenn auf der anderen Seite das zu druckende Dokument leer ist (keine zu druckenden Zeichen enthält), kann der Druckvorgang ebenfalls ganz anders ablaufen: Man kann entweder ein leeres Blatt auswerfen oder einfach nichts tun. Je nachdem, was man ausdrücken möchte, käme man beispielsweise zu den folgenden, Deklaration (1) „relativierenden“ Deklarationen:

$$\begin{aligned} &\textit{drucken}: \textit{NichtLeererDrucker} \times \textit{NichtLeeresDokument} \\ &\textit{drucken}: \textit{NichtLeererDrucker} \times \textit{LeeresDokument} \end{aligned} \tag{3}$$

wobei das Fehlen von Deklarationen für leere Drucker wie oben bedeuten soll, daß die Relation an diesen Stellen undefiniert ist (im gegeben Fall für *LeererDrucker*).

Dieser Sachverhalt unterscheidet sich jedoch insofern substantiell von dem vorigen, als *NichtLeererDrucker* und *NichtLeeresDokument* keine Sub*klassen* von *Drucker* bzw. *Dokument* sind: *NichtLeererDrucker* und *NichtLeeresDokument* sind vielmehr anti-rigide und somit, bei gleichzeitiger Anti-Fundierung, Zustandstypen. Zugleich ergeben sich unterschiedliche Gründe für die Undefiniertheit: Im ersten Fall ist eine ungültige Objektkombination die Ursache, im anderen der ungültige Zustand eines oder mehrerer Objekte. Für das erstere Problem gibt es Lösungen, die sich auf Deklarationsebene formulieren und durch eine spezielle statische Typprüfung implementieren lassen (z. B. die sog. Dependent types [Sh95]); für das zweite liegen diese, aufgrund der den Zustandstypen inhärenten Dynamik der Klassifikation, zumindest nicht auf der Hand.

Im Zuge dieser Arbeit konzentrieren wir uns dennoch, da wir ja an der Integration von Statik und Dynamik interessiert sind, auf die zweite Ursache; erst am Ende (Abschnitt 6) werden die beiden Formen der Undefiniertheit wieder zusammengeführt.

3.1 Verhältnis von Klassentypen zu Zustandstypen

Obiges Beispiel deutet an, daß es günstig sein könnte, Zustandstypen als Subtypen von Klassentypen aufzufassen. Um zu klären, ob dies legitim ist, muß zunächst klar sein, welche Eigenschaften von der Subtypbeziehung in der Modellierung erwartet werden.

In der Modellierung werden Subtypen als Spezialisierungen von anderen, allgemeineren Typen, Generalisierungen genannt, aufgefaßt. Generalisierung ist eine Abstraktion auf Typebene: alle unter einer Generalisierung subsumierten Subtypen werden darin zusammenfaßt. Es folgt, daß alle Aussagen, die auf Elementen (Variablen) einer Generalisierung (als Supertyp) getroffen werden, auch für Elemente (Objekte) jedes Subtyps zutreffen. Das Prinzip der Substituierbarkeit [LW94] gilt auch (und aufgrund ihres deklarativen Charakters – man postuliert einfach, daß die Subtypen verhaltenskonform sind – gerade) für die Modellierung.

Begründet werden kann die Substituierbarkeit durch eine Betrachtung von Intensionen und Extensionen. Die Intension eines Typs ist seine Definition (das Typprädikat oder seine charakteristische Funktion), seine Extension ist die Menge der Objekte, die unter dieses Prädikat fallen. Bei der Extension muß man zusätzlich noch zwischen statischer und dynamischer Extension unterscheiden: Die statische Extension umfaßt alle Objekte, die jemals unter die Intension fallen, die dynamische genau die, die dies zu einem bestimmten Zeitpunkt tun [SK02]. Es folgt, daß die dynamische Extension immer (d. h. zu jedem Zeitpunkt) eine (möglicherweise unechte) Teilmenge der statischen Extension ist.

Ein Subtyp erweitert nun die Intension seiner Supertypen um zusätzliche Anforderungen. Das Typprädikat wird dadurch spezifischer, die Menge der darunter fallenden Objekte (die Extension) kleiner. Es folgt, daß die statische Extension eines Typs immer eine (möglicherweise unechte) Teilmenge der statischen Extensionen seiner Supertypen ist. Gleiches gilt auch für die dynamischen Extensionen, zu jedem beliebigen Zeitpunkt. Aus diesem Zusammenhang folgt wiederum das Prinzip der Substituierbarkeit: Überall dort, wo eine Instanz eines Typs verlangt wird, können auch Instanzen seiner Subtypen erscheinen – sie erfüllen per Definition der Subtypenbeziehungen dieselben Anforderungen. *Text* und *Diagramm* bzw. *Zeilendrucker* und *Plotter* sind solche Subtypen. Da sie zudem rigide und anti-fundierte Klassifikationen darstellen, sind es genauer Subklassen.[4]

Ein Zustandstyp wie beispielsweise *LeererDrucker* ist nun genau dann ein Subtyp von *Drucker*, wenn er dessen Intension erweitert (und damit verstärkt) und dessen Extension verringert. Dies ist aber im gegeben Beispiel genau der Fall: Die Intension von *Drucker* wird erweitert um die Bedingung, daß die Druckerinstanzen leer sein müssen (was auch immer das heißen mag), und durch diese verschärfte Bedingung ist die Zahl der leeren Drucker niemals größer als die Zahl der Drucker. Allerdings kann sie auch allenfalls bei Betrachtung der dynamischen Extension kleiner sein, denn im Prinzip (also statisch gesehen) können *alle* Drucker leer sein. Tatsächlich ist die statische Extension eines Zustandstyps immer gleich der statischen Extension der Klasse, von der er ein (direkter) Subtyp ist. Man beachte allerdings, daß anders als bei der Sub*klassen*beziehung die Zustandssubtypen abstrakt sind, also selbst keine Instanzen erzeugen können – diese müssen von ihren konkreten Supertypen (den Klassen) stammen und können dann zwischen den Zustandstypen herum wandern (ein Fall von Objektmigration, wie sie für dynamische Klassifikationen typisch ist). Die Abstraktheit eines Typs bedeutet hier also – anders als in der objektorientierten Programmierung – nicht automatisch, daß es sich auch um einen Supertyp handelt.

Nun sind Drucker nicht die einzigen Objekte, die leer sein können – wie das obige Beispiel bereits andeutete, können es z. B. auch Dokumente oder ganz andere Objekte sein. Man könnte genaugenommen auch einen Zustandstyp *Leer* definieren, dessen Intension praktisch nichts mit der von Drucker zu tun hat und dessen statische Extension u. a. alle Drucker enthält (da sie ja alle irgendwann einmal leer sein können). Die Betonung liegt

---

[4] Man könnte bemerken, daß im Kontext der Deklaration (1) die Typen *Text* und *Diagramm* eigentlich keine Subtypen von *Dokument* sein können, da sonst, nach dem Prinzip der Substituierbarkeit, auch die „illegalen" Parameterkombination *Text* × *Plotter* zulässig sein müßten. Der Fehler liegt hier allerdings in der falschen Interpretation der Deklaration (1), die eben nicht als „für alle …" gelesen werden darf [Sh95].

hier allerdings auf statisch, denn die dynamische Extension von *Drucker*, die alle Drukker umfaßt, die zu einem bestimmten Zeitpunkt existieren, wird nicht immer eine Teilmenge der dynamischen Extension von *Leer* sein (oder alle Drucker wären immer leer und *Leer* wäre als Zustand für Drucker sinnlos), weswegen *Leer* ganz klar kein Supertyp von *Drucker* sein kann. Das ändert jedoch nichts daran, daß *Leer* ein Zustandstyp sein kann, nur ist eben *Drucker* kein Subtyp davon.[5] Bleibt noch zu klären, ob *LeererDrukker* ein Subtyp von *Leer* ist.

Diese Frage ist schnell beantwortet: *LeererDrucker* ergänzt die Intension von *Leer* um druckerspezifische Merkmale und die Menge aller leeren Drucker ist immer eine Teilmenge aller leeren Dinge (statisch wie dynamisch), womit *LeererDrucker* ein Subtyp von *Leer* ist. Allerdings sind und bleiben beide Zustandstypen abstrakt; die Elemente der Extension von *Leer* (die Menge seiner sog. indirekten Instanzen) rekrutiert sich über den Umweg seiner Sub(zustands)typen aus deren konkreten Supertypen (Klassen), im gegeben Beispiel aus Druckern und Dokumenten. Genau wie Rollen werden Zustandstypen immer über Klassen mit Leben gefüllt (Abbildung 3).

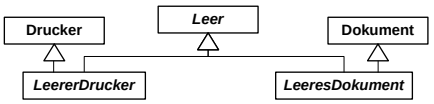


Abbildung 3: Die Subtypenbeziehungen zwischen Zustandstypen (kursiv) und Klassen. Zustandstypen sind immer abstrakt, haben also keine eigenen Instanzen. Zwischen den Extension von Typen und ihren Supertypen gilt immer die (ggf. unechte) Teilmengenbeziehung.

3.2 Zustandstypen und dynamisches Binden

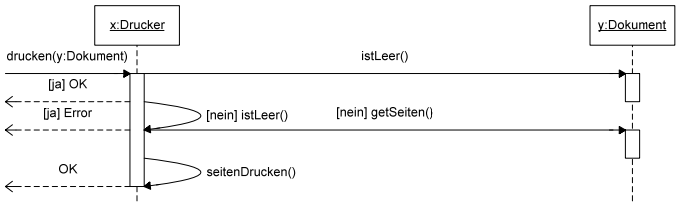
Eine der größten Errungenschaften der Objektorientierung ist das dynamische Binden: Bei einem Methodenaufruf wird anhand des (dynamisch ermittelten) Typs des Empfängerobjektes (und je nach Sprache auch der Parameter) entschieden, welche Implementierung ausgewählt wird. Es findet also eine Fallunterscheidung statt mit der Besonderheit, daß diese implizit, also in der Prozedur des Methodenaufrufs versteckt ist. Der Typ eines Objekts ist dabei immer der, von dem das Objekt eine direkte Instanz ist, und da nur Klassen instanziierbar und diese rigide sind, ist dieser Typ unveränderlich. Damit bezieht sich das dynamische Binden lediglich auf den Umstand, daß der tatsächliche Typ eines Objekts nicht mit dem deklarierten seines Platzhalters übereinstimmen muß; der tatsächliche Typ selbst ist jedoch unveränderlich.

Wenn man jedoch Zustände als Typen auffaßt und noch dazu als Subtypen von Klassen, dann ergibt sich ein ganz anderes Bild. Es ist dann nämlich auch der tatsächliche Typ eines Objektes dynamisch, d. h., über die Zeit veränderlich. Hierdurch wird nahegelegt, daß, entsprechende Deklarationen vorausgesetzt, zur Auswahl einer Methode bei einem Methodenaufruf nicht nur die Klassen der Objekte, von denen sie Instanzen sind, sondern auch ihre Zustände herangezogen werden können (ähnlich dem sog. Predicate

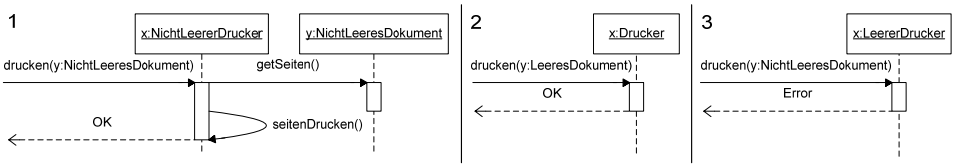
[5] Allgemein können, nach obiger Argumentation, Zustandstypen keine Supertypen von Klassen sein. Siehe hierzu auch [GW+04].

dispatch; s. Abschnitt 7.2). Man erspart sich dadurch die (insbesondere für die Modellierung) lästigen Fallunterscheidungen, die den Zustand eines Objektes betreffen – eine erweiterte Anwendung des „Replace conditional with polymorphism“ Refactorings [Fo99].

Was ist damit gewonnen? Im Beispiel des Druckers könnte das Verhalten der Operation *drucken* mit zwei Parametern vom Typ *Drucker* und *Dokument* anstelle von



durch die drei jeweils wesentlich einfacheren Diagramme



spezifiziert werden. Im allgemeinen Fall erspart man sich durch die Verwendung von Zustandstypen in Verhaltensspezifikationen die In-situ-Behandlung von Sonderfällen wie *FileNotOpen*, *NullObject* [Kü96] genauso wie problemspezifische Fallunterscheidungen, wie z. B. ob eine zu druckende Seite des Dokuments im Hoch- oder im Querformat vorliegt. Gleichzeitig können durch die deklarative Verwendung von Zustandstypen in Diagrammen der statischen Modellierung (z. B. in Klassendiagrammen) bereits Bedingungen ausgedrückt werden, die sonst in die dynamischen Diagramme verfrachtet werden müßten (z. B. wann eine Methode aufgerufen werden kann oder wann eine Beziehung besteht).

4 Zustandstypen und Zustandsautomaten

Ein einzelner Zustandstyp ist wenig aussagekräftig – Zustände beziehen ihren Wert aus der Möglichkeit des Zustandswechsels, wozu jedoch mindestens zwei notwendig sind. Zustände treten also in Gruppen auf. Diese Gruppen sind häufig strukturiert, z. B. als Sequenz (wie das beispielsweise beim eingangs erwähnten Zustandstyp *Jugendlicher* und seinen Nachbarn der Fall ist). Für den allgemeinen Fall können beliebige (also auch nichtlineare) Zustandsfolgen durch endliche Automaten ausgedrückt werden. Entscheidend ist, daß die Zustände zusammenhängen und es nicht sinnvoll ist, sie einzeln mit Klassen zu verknüpfen: entweder alle, oder keiner. Das Interessante an Automaten in diesem Zusammenhang ist, daß sie neben den möglichen Zustandswechseln auch noch deren Bedingungen zu spezifizieren in der Lage sind.

In der objektorientierten Modellierung werden endliche Automaten (oder Statecharts) vor allem eingesetzt, um das Verhalten (von Instanzen) einer Klasse zu spezifizieren. Nun sind solche Automaten per se keine Typen, jedoch läßt sich aus ihnen Typinformation ableiten, wenn man die Menge der Ereignisse eines Automaten (sein Eingabealphabet) als eine Menge von Methoden auffaßt. Wenn beispielsweise eine Klasse deklariert, in ihrem Verhalten mit dem endlichen Automaten aus Abbildung 4 konform zu sein, dann läßt sich daraus folgern, daß sie die Methoden mit den Signaturen *drukken*(*LeeresDokument*) und *drucken*(*NichtLeeresDokument*) besitzen muß. Dabei dürfen die Methoden nur in den Reihenfolgen aufgerufen werden, die durch den Automaten vorgegeben sind. Hält man sich nicht daran, ist das Verhalten nicht definiert.

Des weiteren können aus den Automatenzuständen auch die Zustands(sub-)typen einer Klasse abgeleitet werden. Indem eine Klasse angibt, daß ihr Verhalten durch den Automaten spezifiziert ist, wird die Menge der Objekte dieser Klasse dynamisch in Teilmengen aufgeteilt. Jede solche Teilmenge kann als die Extension eines Zustandssubtypen der Klasse aufgefaßt werden, dessen Intension (Definition) aus den dazugehörigen Methodendeklarationen und deren Implementierung besteht. Wenn die Implementierung einer Methode einen Zustandswechsel eines oder mehrerer Objekte zur Folge hat, wechseln die Objekte eben ihren Zustandstyp, wodurch weitere Methodenaufrufe per dynamische Bindung entsprechend anderen Implementierungen zugeordnet werden. Im Extremfall bestehen die Implementierungen nur aus Zustandswechseln; in der Regel wird jedoch mehr erforderlich sein, was schon daraus ersichtlich ist, daß Automaten und Statecharts in praktischen Anwendungen massiv mit Code-Fragmenten annotiert werden.

Wir fassen also die Ereignisse eines Automaten als Methodenaufrufe auf Objekten auf, deren Klassen deklarieren, mit dem Automaten verhaltenskonform zu sein. (Die Relationsdeklarationen aus den obigen Beispielen werden also ab hier als Methodendeklarationen interpretiert und entsprechend notiert; s. Fußnote 2.) Dabei lassen wir parametrisierte Ereignisse zu, die entsprechend auf parametrisierte Nachrichten bzw. Methoden abgebildet werden. Die Parameter sind typisiert und dürfen insbesondere auch Zustandstypen zum Typ haben. Zwei gleiche Ereignisse mit unterschiedlichen Parametertypen führen dann zu zwei verschiedenen Implementierungen ein und derselben Methode. Da der Automat als Folge von Ereignissen Zustandswechsel spezifiziert, kann die Ausführung einer Methode zu einer Veränderung des Zustandstyps des Empfängerobjekts führen. Ein Zustandswechsel von Parameterobjekten ist nicht vorgesehen; dafür müssen deren Methoden aufgerufen (Automaten angestoßen) werden.

Abbildung 4 definiert einen Automaten mit parametrisierten Ereignissen. Man beachte, daß die Parameter der Ereignisse lediglich Typinformation vorsehen, also Variablen sind und keine konkreten Objekte. Daß es sich bei den Typen um Zustandstypen (in diesem Fall Subtypen von *Dokument*) handelt, liegt am Beispiel; es könnten prinzipiell auch andere Typen sein. So aber besteht eine Abhängigkeit zu einem (den Zuständen eines) zweiten Automaten (der hier erst in Abschnitt 5 eingeführt wird).

Der Automat enthält übrigens einen Indeterminismus: Das Drucken von nicht leeren Dokumenten im Zustand *NichtLeererDrucker* kann, muß aber nicht zu einem Zustandswechsel führen. Hier offenbart sich bereits die in Fußnote 6 bemerkte Ausdruckschwä-

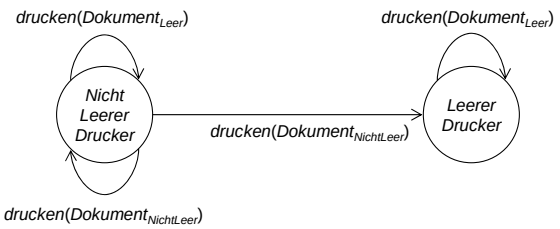


Abbildung 4: Zustandsübergangsdiagramm des Automaten *Druckerverhalten*

che von endlichen Automaten, die aber durch allerlei Erweiterungen (insbesondere durch Statecharts) aufgehoben wird.

Wenn nun die Klasse *Drucker* deklariert, zur Verhaltensspezifikation des Automaten konform zu sein, ergeben sich daraus die folgenden Methodendeklarationen:

NichtLeererDrucker.drucken(*Dokument*$_{Leer}$)
NichtLeererDrucker.drucken(*Dokument*$_{NichtLeer}$)
LeererDrucker.drucken(*Dokument*$_{Leer}$)

Dabei sind die Namen der Zustände des Automaten *Druckerverhalten* absichtlich so gewählt worden, daß sie mit den Zustandssubtypen von *Drucker* aus (3) übereinstimmen. Daß das i. a. nicht der Fall sein muß (und dabei der Sache keinen Abbruch tut), wird im nächsten Abschnitt klar.

5 Modellierung mit Zustandstypen

Die in den vorigen Abschnitten eingeführten Zustandstypen lassen eine weitere Steigerung der schon bestehenden Komplexität der Modellierung befürchten. Insbesondere dann, wenn der Modellierer anfängt, allerlei Zustandstypen zu ersinnen und seine Modelle damit aufzuschlüsseln, besteht die Gefahr, im Detail zu ertrinken. Glücklicherweise ist dies gar nicht notwendig, denn die Zustandstypen und die auf ihnen basierenden Deklarationen ergeben sich automatisch, wie die folgenden Überlegungen zeigen.

Eine Modellspezifikation mit Klassen- und Zustandstypen bestehe fortan aus:

- einer Menge von *Klassentypen* $\mathbf{C} = \{C_1, \ldots\}$,
- einer Menge $\leq$ von *Subklassendeklarationen* mit $\leq \subseteq \mathbf{C} \times \mathbf{C}$, wobei $\leq$ eine Halbordnung auf C definiert (die Klassenhierarchie),
- einer Menge von *Zustandstypen* $\mathbf{Q} = \{Q_1, \ldots\}$,
- einer Menge von *Automaten* (oder Statecharts[6]) $\mathbf{A} = \{A_1, \ldots\}$, wobei für jedes $A \in \mathbf{A}$ dessen Zustandsmenge $Q_A \subseteq \mathbf{Q}$ ist und sich die Q_A paarweise nicht überlappen,

[6] Wichtig ist hier nur, daß die Ausdrucksstärke nicht auf reguläre Sprachen beschränkt ist, denn das wäre schon für das einfache Beispiel aus Abbildung 4 zu wenig.

- einer Menge $\angle$ von *Verhaltenskonformitätsdeklarationen* mit $\angle \subseteq \mathbf{C} \times \mathbf{A}$, die angibt, welche Klassen im Verhalten (ihrer Instanzen) welchen Automaten folgen.

Die in Abschnitt 3.1 erwähnte Subtypenbeziehung zwischen Zuständen ergibt sich z. T. automatisch (s. u./Abbildung); explizite Deklarationen wollen wir zumindest so lange nicht zulassen, solange wir dafür noch keinen Gebrauch haben.

Wenn wir das laufende Beispiel noch um einen Automaten *Füllbar* mit den Zuständen *Leer* und *NichtLeer* sowie der Zustandsübergangsfunktion

$$\delta(\textit{Leer}, \textit{füllen}()) = \textit{NichtLeer}, \delta(\textit{NichtLeer}, \textit{füllen}()) = \textit{NichtLeer}.$$

ergänzen, sähe eine konkrete Modellspezifikation wie folgt aus:

- **C** = {*Drucker*, *Dokument*, *Zeilendrucker*, *Plotter*, *Text*, *Zeichnung*}
- **A** = {*Druckerverhalten*, *Füllbar*}
- **Q** = {*Leer*, *NichtLeer*, *LeererDrucker*, *NichtLeererDrucker*}
- $\leq$ wie in Abbildung 2
- $\angle$ = {*Drucker* $\angle$ *Druckerverhalten*, *Dokument* $\angle$ *Füllbar*}[7]

Für jede Verhaltenskonformitätsdeklaration $C \angle A$ mit Q_A als der Menge von Zuständen (Zustandstypen) von A ergibt sich automatisch eine Familie von Zustandssubtypen von C,

$$\left(C_Q\right)_{Q \in Q_A},$$

die jede dynamische Extension von C partitionieren. Im gegebenen Beispiel ergeben sich so aus *Dokument* $\angle$ *Füllbar* $\textit{Dokument}_{\textit{Leer}}$ und $\textit{Dokument}_{\textit{NichtLeer}}$ als Zustandssubtypen von *Dokument* sowie $\textit{Drucker}_{\textit{LeererDrucker}}$ und $\textit{Drucker}_{\textit{NichtLeererDrucker}}$ aus *Drucker* $\angle$ *Druckerverhalten*. Man muß also die Zustandssubtypen für Klassen nicht explizit deklarieren – sie ergeben sich aus der Verknüpfung einer Klasse mit einem Automaten per Konformitätsdeklaration.

Auf diesen Zustandstypen ergeben sich dann unmittelbar die folgenden Methodendeklarationen:

$\textit{Drucker}_{\textit{NichtLeererDrucker}}.\textit{drucken}(\textit{Dokument}_{\textit{Leer}})$ $\quad$ $\textit{Dokument}_{\textit{Leer}}.\textit{füllen}()$
$\textit{Drucker}_{\textit{NichtLeererDrucker}}.\textit{drucken}(\textit{Dokument}_{\textit{NichtLeer}})$ $\quad$ $\textit{Dokument}_{\textit{NichtLeer}}.\textit{füllen}()$
$\textit{Drucker}_{\textit{LeererDrucker}}.\textit{drucken}(\textit{Dokument}_{\textit{Leer}})$

Für andere Typkombinationen sind die Methoden zunächst nicht definiert. Gleichwohl wird man die Deklaration der Methoden auf Klassenebene (also auf die Ebene von

---

[7] Man kann sich fragen, warum hier nicht nur ein Automat *Behälter* spezifiziert wurde, der *Füllbar* und *Druckerverhalten* bis auf Umbenennung der Ereignisse bzw. Methoden subsumiert. Das Problem ist, daß *füllen* dann nicht mehr (wie jetzt in *Füllbar*) parameterlos sein dürfte, und daß ein geeigneter Parametertyp (der ja nicht *Dokument* sein kann, da sonst Dokumente nur durch sich selbst füllbar wären) im Rahmen dieses Beispiels nicht eingeführt werden sollte. Mehr dazu auch in Abschnitt 7.1.

Drucker bzw. *Dokument*) hochziehen wollen (das Pull-up-method-Refactoring [Fo99]), muß sich aber im klaren darüber sein, daß diese Methoden nicht für alle Konstellationen definiert sind (wie das schon bei den Deklarationen (1) und (3) der Fall war).

Aufgrund der Subklassenbeziehungen von *Zeilendrucker*, *Plotter*, *Text* und *Zeichnung* zu *Drucker* bzw. *Dokument* ergeben sich weiterhin $Zeilendrucker_{LeererDrucke}$, $Zeilendrucker_{NichtLeererDrucke}$ etc. als Zustandssubtypen von *Zeilendrucker* usw. Gleichzeitig werden auch die Methodendeklarationen übertragen, jedoch mit parallel verfeinerten Parametertypen: $Zeilendrucker_{NichtLeererDrucker}.drucken(Text_{Leer})$ usw. Die vollständige Typhierarchie für das Beispiel umfaßt 22 Typen und 28 Subtypenbeziehungen. Man beachte, daß sie nicht vom Modellierer spezifiziert werden muß, sondern daß sie durch das Verweben von Automaten (dynamische Sicht) und Klassen (statische Sicht) automatisch entsteht. Der Modellierer kann die Typen aber jederzeit verwenden (in Deklarationen einsetzen), wenn er zustandsbezogene Aussagen treffen möchte.

6 Statische Projektion: Modellierung mit Ausnahmen

Durch den gegenwärtigen Trend zur modellgetriebenen Softwareentwicklung und aufgrund der Tatsache, daß in der objektorientierten Programmierung die statische Typisierung (und damit verbunden die statische Typprüfung) vorherrschend sind, stellt sich die Frage, wie sich Zustandstypen in eine statische Modellierungs- oder Programmiersprache übertragen lassen.

Zunächst kann man Deklarationen wie (3) als Überladungen auffassen, die eine Relation wie *drucken* abschnittsweise definieren. So wird ausgedrückt, daß das Drucken von leeren Dokumenten auf nicht leeren Druckern anders definiert ist als das Drucken von nicht leeren Dokumenten auf nicht leeren Druckern, und daß das Drucken auf leeren Druckern (außer für leere Dokumente), genauso wie das Drucken auf irgendwelchen anderen Objekten (die keine Drucker sind) überhaupt nicht definiert ist (vgl. a. [Ca95]). Entsprechendes gilt für die Deklarationen in (2): Sie drücken aus, daß das Drucken nur für Paare von Texten und Zeilendruckern bzw. von Diagrammen und Plottern definiert ist.

Zwar wird der Umstand, daß es sich bei den Deklarationen in (2) und (3) um (die Signaturen zu) abschnittsweise(n) Definitionen derselben Relation handelt, durch die Gleichheit des Relationsnamens ausgedrückt, doch wird man im Zuge der Generalisierung versucht sein, Deklarationen (und Definitionen) allgemeiner („generisch“) zu formulieren, um kompaktere Spezifikationen zu erhalten. So kommt man zu Deklarationen wie (1), die aber eine Überverallgemeinerung darstellen, da sie ohne weitere Zusätze auch Paarungen zulassen, für die das Drucken gar nicht definiert ist.

Eine wie in (1) deklarierte Relation hat also Definitionslücken. Um diese zu schließen, bleibt nichts weiter, als die Deklarationen mit expliziten Einschränkungen zu versehen, die sie auf den Umfang des Definitionsbereichs zurechtstutzen. Formal ist diese Einschränkung als eine Relation (oder Menge von Tupeln) zu verstehen, die von dem durch die Deklaration suggerierten kartesischen Produkt als Definitionsbereich abgezogen werden muß. Dies ist für den in (2) beschriebenen Fall relativ leicht möglich: Das Delta

ist einfach (*Plotter* × *Text*) ∪ (*Zeilendrucker* × *Zeichnung*), also eine statisch umrissene (unveränderliche) Menge von Objektpaaren. Man könnte (2) also relativieren und per

drucken: *Drucker* × *Dokument* \ *Plotter* × *Text* \ *Zeilendrucker* × *Zeichnung*

deklarieren. Für (3) läßt sich das Delta zunächst genauso angeben, doch da die statischen Extensionen von Zustandsubtypen immer gleich den statischen Extension ihrer Super(klassen)typen sind (s. Abschnitt 3.1), bleibt bei statischer Betrachtung nach Abzug nichts übrig. In statisch typisierten Sprachen ist auf diese Weise also nichts zu machen.

Nun sehen manche objektorientierte Programmiersprachen eine Möglichkeit vor, mit Beschränkungen der Definiertheit von Methoden (obigen Deltas) umzugehen, die nicht durch das statische Typsystem abgefangen werden können, nämlich die Deklaration von Ausnahmen (Exceptions). Diese können zumindest eine Methodendeklaration (als eine mögliche Umsetzung einer Relation; s. [SK02]) um den Hinweis erweitern, daß die Methode nicht für alle Fälle (Parameterkombinationen und Zustände) definiert ist. Wenn man die Ausnahmen zudem als Erweiterung des Rückgabewertbereichs (in dieser Arbeit nicht behandelt) interpretiert, lassen sie sich einheitlich dazu einsetzen, die Löcher in abschnittsweisen Definitionen zu schließen, und zwar unabhängig davon, ob die Löcher statischen oder dynamischen Ursprungs sind.[8] Dies erlaubt dem Programmierer, seine Deklarationen so weit nach oben zu ziehen, wie es ihm aus Abstraktionsgründen sinnvoll erscheint, und das Delta, die Definitionslücken, in Ausnahmen zu kumulieren. Dies kann man für die Modellierung genauso handhaben.

7 Diskussion

Die zahlreichen Möglichkeiten, die sich aus der Einführung von Zustandstypen und deren Verknüpfung mit Automaten ergeben, konnten hier nur angerissen werden. So wäre es im verwendeten Beispiel beispielsweise naheliegend gewesen, *Drucker* ebenfalls als zu *Füllbar* verhaltenskonform zu deklarieren, was jedoch wiederum die Frage aufgeworfen hätte, wie die beiden Zustände $Drucker_{LeererDrucker}$ und $Drucker_{Leer}$ zusammenhängen, usw. All dies müssen wir uns für spätere Arbeiten aufheben. Hier wollen wir nur noch kurz auf den Zusammenhang von Rollen und Zuständen eingehen, weil sich hieraus unserer Ansicht nach sehr interessante Denkanstöße ergeben, die unsere Einführung von Zustandstypen auch nachträglich rechtfertigen.

7.1 Rollen und Zustände

Man ist vielleicht versucht, Automaten selbst als Typen aufzufassen. So möchte man beispielsweise einen Platzhalter als *Füllbar* typisieren, um auszudrücken, daß an seiner Stelle beliebige Objekte stehen können, die das in *Füllbar* spezifizierte Verhalten aufweisen (das sog. Behavioural subtyping [LW94], nicht weniger als der heilige Gral der Objektorientierung). Diese Objekte könnten dann alle mit dem Automaten verbundenen

[8] Die oben getrennten Fälle der Deklarationen (2) und (3) wären also hier wieder zusammengeführt.

Zustände einnehmen, was nach unserer Sicht allerdings bedeuten würde, daß sie auch die entsprechenden Zustandssubtypen annehmen würden. Das wiederum würde bedeuten, daß diese Zustandstypen Subtypen der Automaten sein müßten, was wiederum implizieren würde, daß sie die Intension des Automaten erben, also insbesondere selbst wieder Automaten sind, was nicht sinnvoll ist.

Man könnte statt dessen eine abstrakte Klasse einführen, die mit dem Automaten verhaltenskonform ist, und alle anderen Klassen davon ableiten. Diese abstrakte Klasse wäre dann der geeignete Typ für obigen Platzhalter. Sie wäre aber, zumindest im allgemeinen Fall, nicht wirklich eine Klasse, denn sie spezifiziert in der Regel nur eine partielle Sicht auf ihre Subklassen, nur eines von mehreren möglichen Verhalten (die bereits oben erwähnte Möglichkeit, daß eine Klasse mit mehreren Automaten konform zu sein deklarieren kann). Als rigide Typen können Klassen aber nicht mehrere direkte Superklassen haben, da ihre Instanzen dann auch mehrere Identitäten haben müßten (für die ausführliche Begründung s. [GW+04]).

Die Lösung liegt auf der Hand: Rollen. Rollen definieren Verhalten, das von den Objekten, die die Rolle spielen, im Kontext der Rolle erwartet wird. Was läge also näher, als zunächst eine Rolle als mit einem Automaten verhaltenskonform zu deklarieren und dann die Klassen deklarieren zu lassen, daß sie die Rolle füllen? Statisch gesehen ist die Extension einer Rolle immer die Vereinigung der Extensionen der rollenspielenden Klassen und damit einer Obermenge jeder einzelnen; dynamisch ergibt sich allerdings das Problem, daß die Extension einer Rolle durch die Extensionen der Relationen, in denen sie eine Stelle repräsentiert, eingeschränkt ist, also in der Regel nicht alle (dynamisch existierenden) Instanzen der Rollenspielerklassen umfaßt [St00a, St02]. Hier sind noch genauere Untersuchungen vonnöten; es deutet sich jedoch an, daß das Rollespielen selbst ein Zustand ist, also entsprechende „Rollenzustandstypen“ induziert, zwischen denen die Instanzen migrieren können.

7.2 Verwandte Arbeiten

Beugnard hat auf die Unverzichtbarkeit klarerer Regeln zum dynamischen Binden in der Modellierung hingewiesen [Be02]; durch die Einführung von Zustandstypen als Subtypen gewinnt das dynamische Binden für die Modellierung zusätzlich an Bedeutung. Millstein erweitert Java, aufbauend auf einer Arbeit von Ernst et al. [EK+98], um „Predicate dispatch“, also die Möglichkeit die Methodenauswahl zusätzlich durch den Typ von Argumenten oder den Wert von (Parameter- und Objekt-) Feldern zu beeinflussen [Mi04]. Hierbei liegt der Schwerpunkt auf der Konstruktion eines modularen, statischen Typsystems und dem Erkennen von mehrdeutigen Methodendefinitionen. Die Methodenauswahl ist auch im Fall der Auswertung von Parameterwerten auf die Überprüfung bereits existierende Klassentypen und darüber hinaus auf Integer- und Boole’sche Werte beschränkt und unterscheidet sich damit substantiell von dem hier vorgestellten Dispatch auf Zustandsuntertypen.

Nierstrasz definiert reguläre Typen („regular types“) als Typen, deren Protokoll durch nicht-deterministische endliche Automaten spezifiziert (oder zumindest approximiert) wird [Ni93]. Aufgrund der Tatsache, daß sich für solche Automaten eine Äquivalenz

leicht entscheiden läßt, kann er ein Verfahren angeben, das entscheidet, ob ein regulärer Typ Subtyp eines anderen ist (im Sinne einer protokollbezogenen Substituierbarkeit). Bei uns stehen Automaten nicht direkt in Beziehung zueinander; sobald aber Klassen, die zu unterschiedlichen Automaten verhaltenskonform sind, in Subtypbeziehung zueinander stehen, können sich u. U. Konflikte ergeben, nämlich dann, wenn die Ereignismengen der Automaten nicht disjunkt sind (zur Erinnerung: Bei uns kann eine Klasse prinzipiell zu mehreren Automaten gleichzeitig konform sein). Die Klassen selbst erben das Verhalten von Automaten, können damit also direkt nicht in Widerspruch stehen.

Paech und Rumpe gehen über die Arbeit von Nierstrasz hinaus, indem sie die Zustände eines Automaten mit Einschränkungen der Attributwerte der Objekte der Klasse verbinden, zu der der Automat gehört [PR94]. Dies entspricht in etwa der kovarianten Redefinition von Attributen in Zustandstypen, die wir in dieser Arbeit aus Platzgründen nicht behandelt haben. Ähnlich wie Nierstrasz spezifizieren die beiden Autoren Regeln zur Verhaltensspezialisierung und so zur Subtypbeziehung, die sie allerdings in erster Linie als Möglichkeit der Verfeinerung im Sinne einer inkrementellen Modellentwicklung verstanden wissen wollen. Ganz interessant ist, daß auch Paech und Rumpe Rollen mit (partiellen) Verhaltensspezifikationen in Verbindung bringen und verlangen, daß Rollenspieler diese einhalten (vgl. Abschnitt 7.1). Ansonsten gilt aber das zu Nierstrasz gesagte.

Wieringa et al. definieren sog. dynamische Subklassen und Rollenklassen in Ergänzung zu gewöhnlichen (statischen) Klassen, um verschiedene Arten der Objektmigration zu modellieren [WJ+95]. Dynamische Klassen entsprechen in etwa unseren Zustandstypen und unterscheiden sich von Rollen vor allem dadurch, daß ein Objekt und ein Objekt in einer Rolle unterschiedliche Identitäten haben. Auf allgemeine Fragen der Modellierung, insbesondere die Integration statischer und dynamischer Sichten, wird in der Arbeit nicht eingegangen.

DeLine und Fähndrich führen Typzustände in die objektorientierte Programmierung ein, um mit Hilfe eines statischen, modularen Typsystems exaktere Aussagen über Objektzustände machen zu können [DF04]. Zustandstypen werden in Vor- und Nachbedingungen verwendet, um so u. U. die Ausführbarkeit von Methoden mit Voraussetzungen an den Objektzustand zu garantieren. Damit sind die möglichen Zustandsübergänge auf alle Methoden verteilt und der Zustandsüberganggraph, im Gegensatz zu unseren expliziten Automaten, nur implizit definiert. DeLine und Fähndrich beschäftigen sich intensiv mit der Behandlung von Typzuständen bei Vererbungsbeziehungen zwischen Klassen, nutzen aber nicht die Möglichkeit, das dynamische Binden auch auf den Einbezug des Typzustands hin zu erweitern.

8 Schluß

Die Integration statischer und dynamischer Sichten, insbesondere von typbasierten und von zustandsbasierten Spezifikationen, ist bis heute ein Problem. Wir haben Zustandstypen als Bindeglieder zwischen beiden Welten ontologisch motiviert und ihre Beziehungen zu etablierten Modellierungselementen, insbesondere Klassen, Relationen (Methoden) und Automaten, definiert. Es zeigt sich, daß sich mit Hilfe von Zustandstypen or-

thogonale Sichten zu einem Modell verweben lassen, das um ein vielfaches komplexer ist als die jeweiligen Ausgangsmodelle. Während die dadurch gewonnenen Möglichkeiten der spezifischeren, weil zustandsbezogenen, Formulierung per Deklaration sowie der Vereinfachungen von Verhaltensspezifikationen bereits auf der Hand liegen, ist auch klar, daß noch einiges an Arbeit nötig ist, um die Möglichkeiten und Grenzen des Ansatzes weiter auszuloten.

Literaturverzeichnis

[Be02] A Beugnard “Is MDA achievable without a proper definition of late binding?” *UML Workshop in Software Model Engineering (WiSME)* (2002).

[Ca95] G Castagna “Covariance and contravariance: conflict without a cause” *ACM TOPLAS* 17:3 (1995) 431–447.

[DF04] R DeLine, M Fähnrich “Typestates for Objects” in: *Proceedings of the European Conference on Object-oriented Programming* (ECOOP '04) LNCS 3086 (2004) 465–490

[EK+98] M Ernst, C Kaplan, C Chambers “Predicate Dispatching: A Unified Theory of Dispatch” in: *Proceedings of the European Conference on Object-oriented Programming* (ECOOP '98) LNCS 1445 (1998) 186–211

[Fo99] M Fowler *Refactoring: Improving the Design of Existing Code* (Addison-Wesley 1999).

[GW+04] G Guizzardi, G Wagner, N Guarino, M van Sinderen “An Ontologically Well-Founded Profile for UML Conceptual Models” in: *CAiSE 2004* LNCS 3084 (2004) 112–126.

[Gu92] N Guarino “Concepts, attributes and arbitrary relations” *Data & Knowledge Engineering* 8 (1992) 249–261.

[Kü96] Thomas Kühne “Nil and None considered Null and Void” *Proceedings of WOON'96* (1996).

[LW94] BH Liskov, JM Wing “A behavioral notion of subtyping” *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16:6 (1994) 1811–1841.

[Mi04] T Milstein “Practical Predicate Dispatch” in: *Conference on Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA) (2004) 345–364.

[Ni93] O Nierstrasz “Regular types for active objects” in: *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (1993) 1–15.

[PR94] B Paech, B Rumpe “A new concept of refinement used for behaviour modelling with automata” in: *2nd Int. Symp. of Formal Methods Europe* Springer LNCS 873 (1994) 154–174.

[Sh95] DL Shang “Covariant deep subtyping reconsidered” *SIGPLAN Notices* 30:5 (1995) 21–28.

[St20a] F Steimann “On the representation of roles in object-oriented and conceptual modelling” *Data & Knowledge Engineering* 35:1 (2000) 83–106.

[St20b] F Steimann “A radical revision of UML’s role concept” in: *UML 2000, Proceedings of the 3rd International Conference* (Springer 2000) 194–209.

[St02] F Steimann *Ein natürlicher Rollenbegriff für die Softwaremodellierung* (Shaker-Verlag 2002).

[SK02] F Steimann, T Kühne “A radical reduction of UML’s core semantics” in: *UML 2002: Proceedings of the 5th International Conference* (Springer, 2002) 34–48.

[WJ+95] R Wieringa, W de Jonge, P Spruit “Using dynamic classes and role classes to model object migration” *Theory and Practice of Object Systems* 1:1 61–83.