Formale Methodik des Entwurfs verteilter objektorientierter Systeme

Bernhard Rumpe

Institut für Informatik,
Technische Universität München
80333 München
rumpe@in.tum.de, http://www.in.tum.de/

Dieser Artikel faßt die wesentlichen Ergebnisse, sowie einige Höhepunkte der Dissertation [10] zusammen. Es wird eine methodische Grundlage für die Spezifikation von Struktur und Verhalten verteilter, objektorientierter Systeme skizziert. Es werden Beschreibungstechniken für Klassen- und Zustandsdiagramme definiert und ein Verfeinerungskalkül für diese angegeben. Insbesondere wird in diesem Artikel der Verfeinerungskalkül für Zustandsdiagramme vorgestellt und an einem ausführlichen Beispiel motiviert. Die sich ergebenden Folgerungen für den objektorientierten Softwareentwicklungsprozeß werden diskutiert.

1 Einführung

Die Entwicklung von Softwareprodukten besitzt in der heutigen, stark auf Informationstechnik ausgerichteten Industrie eine außerordentliche Bedeutung. Die Erstellung industrieller Softwareprodukte ist daher ähnlich systematischen und strukturierten Richtlinien zu unterwerfen, wie es bei anderen industriellen Produkten üblich ist. Nur dadurch lassen sich Entwurfs- und Produktionsprozeß und damit die Produktqualität verbessern. In den klassischen Ingenieursdisziplinen werden deshalb ausgereifte, genormte Techniken eingesetzt, die die verschiedenen Phasen der Erstellung eines Produkts, von der Entwicklung bis zur Wartung, unterstützen. Solche Techniken bieten Produktsichten, in denen einzelne Aspekte explizit herausgearbeitet werden können. Beispiele hierzu sind etwa Schalt- oder Baupläne. Diese Techniken werden zumeist durch Werkzeuge unterstützt, die Routinearbeiten übernehmen sowie überprüfbare Entwicklungsfehler verhindern.
Obwohl die inhärente Komplexität von Softwaresystemen von anderen Produkten kaum erreicht wird, ist ein systematischer und strukturierter Konstruktionsprozeß (in der Informatik auch Entwicklungsprozeß genannt) noch nicht

[Rum97] B. Rumpe
Formale Methodik des Entwurfs verteilter objektorientierter Systeme.
In: Ausgezeichnete Informatikdissertationen 1997.
B. G. Teubner Stuttgart.
www.se-rwth.de/publications

gefunden. Die hier dargestellten Ergebnisse aus [10] stellen einen Beitrag auf dem Weg zu einem solchen Entwicklungsprozeß dar. Darüberhinaus wird ein Beitrag zur wissenschaftlichen Fundierung der Softwareentwicklung bereitgestellt, der eine Brücke zwischen Theorie und Praxis bildet.
In der Softwaretechnik werden heute verstärkt objektorientierte Methoden eingesetzt, die auf dem neu entwickelten Standard der Unified Modeling Language (UML, [12]) basieren. Die UML bietet als Kombination mehrerer älterer objektorientierter Methoden eine reichhaltige Menge von Konzepten, die unter anderem in Klassen-, Zustands-, oder Sequenzdiagrammen zusammengefaßt sind und verschiedene Sichten und Abstraktionen eines Systems darstellen. Für die UML existiert derzeit eine präzise Festlegung der Syntax und eine informelle, an vielen Stellen noch unpräzise und mißverständliche Semantik. Techniken zur Transformation von UML Modellen zum Zweck der Verfeinerung oder Komposition werden derzeit in einigen Gruppen diskutiert (siehe etwa Proceedings der UML'98 [6]), sind aber in der Praxis noch nicht etabliert. Genau diese Techniken aber bilden das Bindeglied zwischen den Methoden im Großen und den Beschreibungstechniken. Dadurch wird etwa der Projektfortschritt meßbar und besser beherrschbar. Auch können Anforderungen und Entwurfsentscheidungen besser bis zur Implementierung verfolgt werden (Traceability).
Die hier dargestellten Ergebnisse bilden einen Weg solche Bindeglieder zu etablieren. Die Dissertation [10] enthält folgende Ergebnisse:

- Ein auf Dokumenten basierendes generisches Vorgehensmodell wurde eingeführt, das sich insbesondere zur Verfolgung von Verfeinerungs- und Revisionsschritten eignet.
- Klassendiagramme und Zustandsdiagramme wurden als graphische Notationen, Klassensignaturen und Datentypdokumente als textuelle Ergänzungen eingeführt. Bei beiden Diagrammarten wurde eine Teilmenge der UML übernommen und adaptiert.
- Ein Systembegriff wurde definiert, der präzise beschreibt welche Art von Systemen implementiert wird. Hier sind systemimmanente Eigenschaften wie die Art der Kommunikation oder die Form der Nebenläufigkeit beschrieben. Ein solcher Systembegriff erlaubt die Definition einer integrierten Semantik für heterogene Beschreibungstechniken und stellt damit eine wichtige Basis für rigorose Konsistenzprüfungen und Transformationen dar.
- Klassendiagramme und Zustandsdiagramme haben eine formale Semantik durch eine Abbildung auf den Systembegriff erhalten.
- Für beide Notationen wurde ein intuitiver Verfeinerungsbegriff definiert, der auf dem Systembegriff beruht. Es wurde insbesondere die Korrektheit des Verfeinerungskalküls bewiesen.

Ein wesentliches Ergebnis aus [10] ist der entstandene Verfeinerungskalkül für Zustandsdiagramme. Er soll im Rest dieses Artikels vorgestellt werden. Weitere Ergebnisse sowie eine genauere Untersuchung der Eigenschaften des vorgestellten Kalküls können in [10] nachgelesen werden.
Der zweite Abschnitt beschäftigt sich mit der Fundierung des Softwareentwicklungsprozesses und gibt einen Einblick in den gewählten Systembegriff. Der dritte Abschnitt führt Zustandsdiagramme ein und skizziert für diese eine präzise Semantik. Im vierten Abschnitt wird zunächst der Verfeinerungskalkül für Zustandsdiagramme an einem Beispiel vorgeführt und anschließend komplett vorgestellt. Der fünfte Abschnitt enthält eine Diskussion verwandter Arbeiten, und der sechste Abschnitt gibt einen Ausblick auf weitere Aktivitäten.

2 Entwicklungsprozeß und Systemmodell

Die Entwicklung eines Softwaresystems ist aufgrund der Komplexität des entstehenden Systems notwendigerweise selbst ein komplexer Vorgang. Deshalb ist es wichtig, den Vorgang der Systementwicklung zu strukturieren und damit zu systematisieren.
Je nach Art der benutzten Beschreibungstechniken entsteht während der Systementwicklung eine große Menge auf komplexe Weise miteinander in Verbindung stehender Dokumente unterschiedlicher Arten. Für diese Dokumente ist zunächst eine eigenständige Semantik wichtig. Um das Zusammenspiel verschiedener Arten von Dokumenten zu verstehen, ist jedoch eine Integration der Semantiken notwendig. Dazu wurde ein Systembegriff entwickelt, der alle wesentlichen Eigenschaften eines Systems, wie Struktur, Dynamik, Verhalten einzelner Komponenten und Interaktionen beschreibt. Abbildung 1 zeigt das Verfahren einer Semantikdefinition: Jede Notation wird auf das Systemmodell abgebildet. Dadurch können die Entwickler der Notationen auf Basis des Systemmodells über Kontextbedingungen und Transformationen, wie zum Beispiel einen Verfeinerungskalkül nachdenken. Wichtig ist, daß der Anwender der Notationen mit dem (formal beschriebenen) Systemmodell nicht in Berührung kommt, sondern die gefundenen Kontextbedingungen und Transformationen in Werkzeuge implementiert werden und so dem Anwender implizit zur Verfügung stehen.

2.1 Das Systemmodell

Das Systemmodell dient zur semantischen Fundierung von Techniken zur Beschreibung verteilter, objektorientierter Software. Es besitzt für objektorientierte Systeme typische Eigenschaften, die hier kurz skizziert werden.

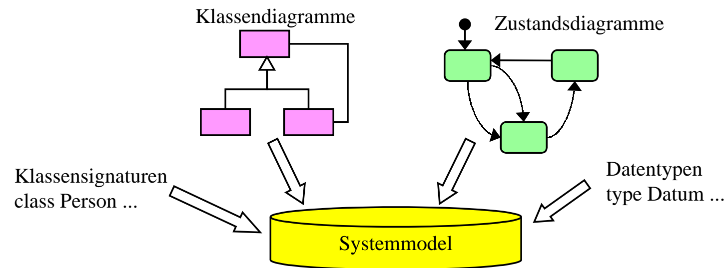


Abbildung 1: Formalisierte Dokumentarten

Ein objektorientiertes System besteht aus *Objekten*, die zumindest konzeptuell *parallel* interagieren und damit *verteilt* realisiert werden können. Ein Objekt verfügt über einen eindeutigen *Identifikator*, einen gekapselten *Zustand* und eine explizite *Schnittstelle*. Objekte interagieren über den *Austausch von Nachrichten*. Die Kommunikation ist *asynchron*, das heißt der Empfänger kann den Sender nicht blockieren, aber gegebenenfalls wird die Nachricht zunächst in einem Puffer gespeichert. Nachrichten werden mittels Objektidentifikatoren adressiert. Objekte können dynamisch erzeugt und gelöscht werden.

Die prinzipiell unbeschränkte Menge von Objekten wird durch *Klassen* in eine endliche Menge von zu beschreibenden Einheiten gruppiert. Klassen beschreiben ein Typsystem für Objekte sowie eines für Objektidentifikatoren. Sie charakterisieren das Verhalten ihrer Instanzen und geben deren Implementierung an. Klassen stehen ihrerseits in Vererbungsbeziehungen, die damit eine ähnlich vielfältige Rolle spielen wie die Klassen selbst.

All diese Aspekte wurden unter Benutzung der Theorie der Ströme [1] im Systemmodell präzise festgelegt. Die Stromtheorie basiert auf reiner Mathematik und bildet aufgrund ihrer Mächtigkeit und Flexibilität eine ideale Technik, das skizzierte Systemmodell kompakt und präzise festzulegen. Insbesondere kann aus einer gegebenen Glass-Box-Sicht für ein Objekt eine Black-Box-Sicht des Objekts berechnet werden. Dies ist eine wesentliche Basis des nachfolgend angegebenen Verfeinerungskalküls. Die Glass-Box-Sicht beschreibt den nur für den Entwickler einer Klasse sichtbaren Zustand und das Verhalten eines Objekts. Die Glass-Box-Sicht wird durch ein Zustandsdiagramm adäquat beschrieben. Für die Umgebung ist nach Definition des Systemmodells jedoch nur das Verhalten sichtbar, denn der Zustand ist gekapselt. Entsprechend ist nur das für die Umgebung zugesicherte Verhalten bei einer Verfeinerung zu bewahren. Der Zustandsraum darf sich frei ändern. Die Stromtheorie bietet dafür die Technik der Black-Box-Sicht der Umgebung und die darauf definierte Verfeinerungsrelation.

2.2 Dokumentorientierte Softwareentwicklung

Während der Entwicklung und Implementierung eines Softwareprodukts entstehen immer neue Dokumente, die Informationen über das System in unterschiedlichen Formen und Abstraktionsgraden enthalten. Dazu zählen die in dieser Arbeit untersuchten Diagrammarten genauso wie informelle Texte (Pflichtenheft, Dokumentation) und der implementierte Code.

Zwischen diesen Dokumenten besteht ein komplexes Beziehungsgeflecht, das von Benutzungsabhängigkeiten über Generierung von Dokumenten bis hin zur Versionsverwaltung reicht. Leider wird die Verwaltung solcher Dokumente und deren vielfältiger Beziehungen heute noch nicht ausreichend verstanden und beherrscht, weshalb heute oft der modellorientierte Ansatz benutzt wird. Dieser stellt einen jeweils aktuellen Modellierungszustand dar und bietet daher in seiner reinen Form z.B. keine Verfolgung von Anforderungen (Traceability) oder Versionsverwaltung.

In einem dokumentorientierten Softwareentwicklungsprozeß entsteht ein Dokumentgeflecht mit verschiedensten Beziehungen zwischen den Dokumenten. Eine von uns untersuchte, an der Semantik orientierte Beziehung ist die *Verfeinerung.*

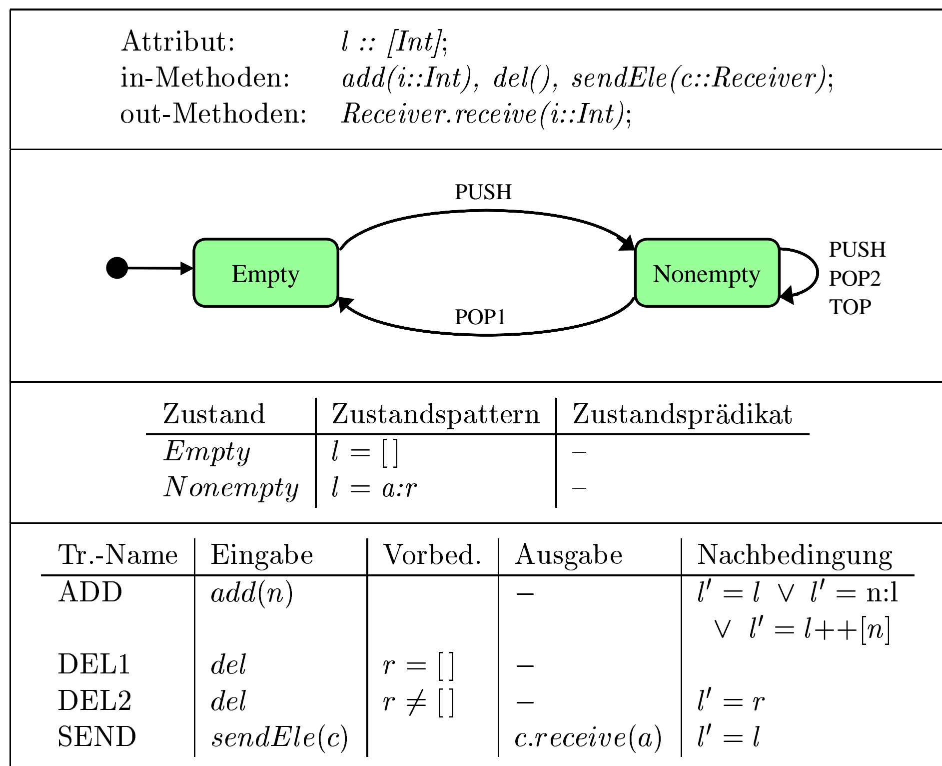
Generell ist ein Dokument D' Verfeinerung eines Dokuments D, wenn die im neuen Dokument D' enthaltenen Informationen alle Informationen aus D umfassen. D wird damit redundant und braucht im weiteren Verlauf der Softwareentwicklung nicht mehr beachtet zu werden. Wenn die Semantik der Dokumente, wie bei den hier vorgestellten Diagrammen, präzise definiert ist, so kann auch die Verfeinerungsrelation präzise angegeben werden. Informell bedeutet das: Jedes System das D' implementiert ist auch eine Implementierung von D.

Es gibt Gründe D weiterhin zu behalten und zu nutzen. Zum Beispiel kann D als eine Schnittstellenbeschreibung einer Klasse für die Umgebung zur Verfügung stehen, während Implementierungsdetails aus D' verborgen bleiben. Damit bleibt eine Änderung der Implementierung D' lokal, wenn die Schnittstelle D nicht betroffen ist. Genau dieses bei Klassensignaturen längst übliche Prinzip kann auch auf Verhaltensbeschreibungen angewendet werden. Insbesondere kann D als abstrakte Beschreibung des Verhaltens einer Oberklasse in Subklassen auf verschiedene Weise spezialisiert werden.

Im Rest dieses Artikels werden wir uns auf die Vorstellung der Zustandsdiagramme und deren Transformationen einschränken. Der theoretische Unterbau, sowie eine intensivere Untersuchung der Eigenschaften der hier eingeführten Zustandsdiagramme können in [10] nachgelesen werden.

3 Zustandsdiagramme

Zustandsdiagramme werden in der Softwareentwicklung genutzt, um das Verhalten von Objekten einer Klasse zu beschreiben. Ein Zustandsdiagramm wird einer Klasse zugeordnet und beschreibt, welche Nachrichten (Methodenaufrufe) ein Objekt verarbeitet (Eingabe) und welche es selbst produziert (Ausgabe). Es beschreibt den Zustandsraum eines Objekts und die Verarbeitung einer ankommenden Nachricht in Abhängigkeit des aktuellen Zustands. Zustandsdiagramme sind daher das Bindeglied zwischen Zustand und Verhalten eines Objekts.
Die Form eines Zustandsdiagramms soll hier nur anhand des Beispiels in Abbildung 2 skizziert werden. Wir beschreiben darin einen Speicher natürlicher Zahlen, zum einen weil er wohl bekannt und verstanden ist, zum zweiten weil er es erlaubt alle wesentlichen Eigenschaften von Zustandsdiagrammen zu demonstrieren, und zum dritten, weil in jedem Softwaresystem heute die von ihm modellierten Container-Klassen eine wesentliche Rolle spielen.



Attribut: *l :: [Int]*;
in-Methoden: *add(i::Int), del(), sendEle(c::Receiver)*;
out-Methoden: *Receiver.receive(i::Int)*;

Zustand	Zustandspattern	Zustandsprädikat
Empty	$l = []$	–
Nonempty	$l = a{:}r$	–

Tr.-Name	Eingabe	Vorbed.	Ausgabe	Nachbedingung
ADD	$add(n)$		–	$l' = l \ \vee \ l' = \text{n:l}$ $\vee \ l' = l{+}{+}[n]$
DEL1	*del*	$r = []$	–	
DEL2	*del*	$r \neq []$	–	$l' = r$
SEND	$sendEle(c)$		$c.receive(a)$	$l' = l$

Abbildung 2: Zustandsdiagramm für einen Datenspeicher

Neben den Attributen, hier bestehend aus einer Liste l von Zahlen, wird angegeben, welche Methodenaufrufe akzeptiert und welche emittiert werden. Das Diagramm beschreibt, welche Zustände und Transitionen existieren, und definiert einen Anfangszustand. Sowohl Zustände als auch Transitionen sind mit Bezeichnern versehen, um in den nachfolgenden Tabellen deren genauere Eigenschaften festzulegen. Zur Spezifikation dieser Eigenschaften wird die um prädikatenlogische Ausdrücke erweiterte funktionale Sprache Gofer verwendet. Andere Sprachen, wie etwa die Object Constraint Language, wie sie in der UML definiert wurde, sind aber ebenfalls geeignet. $[Int]$ bezeichnet eine Liste von Zahlen, $[\,]$ die leere Liste, $++$ die Konkatenation zweier Listen, $a{:}r$ ein Pattern mit der Zahl a und der Liste von Zahlen (Rest) r.

Die Methoden *add* und *del* modifizieren den Speicher. Die Methode *sendEle* führt zu einer Weiterleitung des ersten Elements der Liste.

Jedem Zustand wird ein Pattern und ein Prädikat zugeordnet. Das Pattern $l = a{:}r$ beschreibt, daß das Attribut l in der Form $a{:}r$ für eine passende Zahl a und eine Rest-Liste r ist. Damit wird zum einen gesichert, daß l nicht leer ist, und zum anderen werden die Werte a und r besetzt, die bei den Transitionen, die vom Zustand $Nonempty$ ausgehen, benutzt werden können. Ein weiteres Prädikat zur Charakterisierung des Zustands $Nonempty$ erübrigt sich.

Transitionen bestehen aus einem Eingabepattern, das die Form der bearbeiteten Nachricht festlegt und die darin enthaltenen Variablen für spätere Verwendung bindet, und einem Ausgabeausdruck, der beschreibt, welche Nachricht das Objekt als Reaktion emittiert. Eine zusätzliche Vorbedingung schränkt den Schaltbereich weiter ein. Die Nachbedingung erlaubt die detailliertere Beschreibung des neuen Zustands (l') unter Benutzung des alten Zustands. Die vollständige Nachbedingung einer Transition ergibt sich durch Einbeziehung des Zielzustandsprädikats, so daß etwa bei der Transition DEL1 bereits durch den Zielzustand festgelegt wird, daß $l = [\,]$.

Die Verwendung von Pattern und Prädikaten erlaubt die endliche Darstellung von meist unendlichen Zustandsräumen und Transitionsmengen. Die Benutzung von Pattern (die auch als Prädikate dargestellt werden könnten) erhöht einerseits den Lesekomfort, da so elegant zusätzliche Variablen gebunden werden können, und erlaubt eine automatische Überprüfung vieler Eigenschaften, die ansonsten bei formaler Verwendung des Verfeinerungskalküls zu verifizieren wären. Darüberhinaus ist durch Benutzung von Pattern auch Rapid Prototyping möglich.

Zustandsdiagramme sind primär nichtdeterministisch. Dies kann sich durch mehrere Transitionen mit überlappendem Pattern und überlappender Vorbedingung zeigen. Ein Fehlen einer Transition zu einer Nachricht, wie etwa *del* im Zustand *Empty* stellt völlige Unterspezifikation dar, die bei Weiterentwicklun-

gen durch eine robuste Implementierung verfeinert werden kann. Eine dritte Möglichkeit ist gegeben, wenn die Nachbedingungen mehrere Verhaltensvarianten offen läßt. So erlaubt die Transition ADD dem Datenspeicher sich wie ein Keller ($l' = $ n:l) oder eine Warteschlange ($l' = l{+}{+}[n]$) zu verhalten. Es ist auch möglich die Kapazität zu beschränken ($l' = l$) oder verschiedene Kombinationen daraus zu implementieren. Hier kann vom Entwickler eine weitere Verfeinerung angegeben werden, die zum Beispiel eine Mindestkapazität sichert.
Mit der Definition eines Automaten ist eine intuitive Vorstellung verbunden, die der angegebene Automat vermittelt. Es ist wichtig, daß diese mit der Semantik übereinstimmt. Sonst wäre eine graphische Darstellung weniger eine Unterstützung bei der Softwareentwicklung als vielmehr ein Hindernis. Um die Übereinstimmung von Intuition und Semantik zu sichern, müssen daher geeignete Kontextbedingungen eingehalten werden:

- Automatenzustände bilden nichtleere Äquivalenzklassen von Datenzuständen (Erfüllbarkeit und Disjunktheit der Zustandsprädikate).
- Die Schaltbereitschaft einer Transition hängt nur vom Quellzustand und der Vorbedingung ab (Enabledness von Transitionen).

Die erste Bedingung sichert, daß ein Objekt sich nicht in „mehreren Automatenzuständen gleichzeitig“ befindet. Die zweite Bedingung sichert, daß eine Transition, die einmal begonnen wurde nicht abgebrochen und ungeschehen gemacht werden muß, weil festgestellt wird, daß sie nicht ausführbar ist. Bei Verwendung von Prädikaten führen die Kontextbedingungen zu Beweisverpflichtungen. Zum einen sind die so entstehenden Beweisverpflichtungen sehr klein gegenüber rein in Logik formulierten Verhaltens- und Strukturbeschreibungen und können damit einer zumindest teilautomatisierten Verifikation zugänglich sein. Zum zweiten ist bereits die explizite Ausgabe von solchen Beweisverpflichtungen und deren informelle Durchsicht auf Plausibilität eine sehr große Hilfe für die Entwicklung stabiler Software. Aufgrund der Benutzung von Pattern können außerdem viele Beweisverpflichtungen bereits automatisiert überprüft werden. Würden aus den Zustandsbeschreibungen Beweisverpflichtungen erzeugt, so würden diese wie folgt aussehen[1]:

$$\exists l.\ l = [\,], \quad \exists l, a, r.\ l = a{:}r, \quad \neg(l = [\,] \wedge (\exists a, r.\ l = a{:}r))$$

Weitere Beweisverpflichtungen entstehen aus den beiden Transitionen, um die Enabledness-Bedingung zu sichern. Beispielsweise generiert sich aus DEL1 folgende Verpflichtung:

[1] Freie Variablen der angegebenen Axiome sind außen ∀-quantifiziert. Die Eingabe ist mit *in* und die Ausgabe mit *out* benannt.

$$l = a{:}r \land in = del \land r = [\,] \Rightarrow \exists l', out.\ l' = [\,] \land l' = [\,] \land out = \epsilon$$

Unter den Beweisverpflichtungen des Zustandsdiagramms ist keine, die nicht von einem Theorembeweiser automatisiert bewiesen werden könnte.
In vielen Anwendungen, vor allem im Bereich von Steuersystemen mit nur endlichem Zustandsraum, reicht ein vereinfachter Ansatz ohne Prädikate und Pattern bereits aus. Wir werden im nächsten Abschnitt ein solches, vereinfachtes Beispiel für die Vorstellung des Verfeinerungskalküls nutzen.

4 Verfeinerungskalkül für Zustandsdiagramme

Die in Abschnitt 2.2 beschriebene allgemeine Verfeinerungsbeziehung zwischen Dokumenten kann für Zustandsdiagramme ganz spezifisch zugeschnitten werden. Während die Verfeinerungsbeziehung auf der Semantikseite durch Mengeninklusion dargestellt werden kann, ist für eine zielgerichtete Anwendung durch den Benutzer ein Satz von konstruktiven Regeln wichtig. Diese Regeln manipulieren Elemente der Dokumente und sichern durch Kontextbedingungen, daß diese Transformationen korrekte Verfeinerungen darstellen. Ein Verfeinerungskalkül besteht also aus Regeln, die den in Abbildung 3 wiedergegebenen Sachverhalt erfüllen. Ein Zustandsdiagramm Z' ist Verfeinerung eines Zustandsdiagramms Z, wenn das durch Z erlaubte Verhalten eines Objekts durch Z' genauer festgelegt wird.

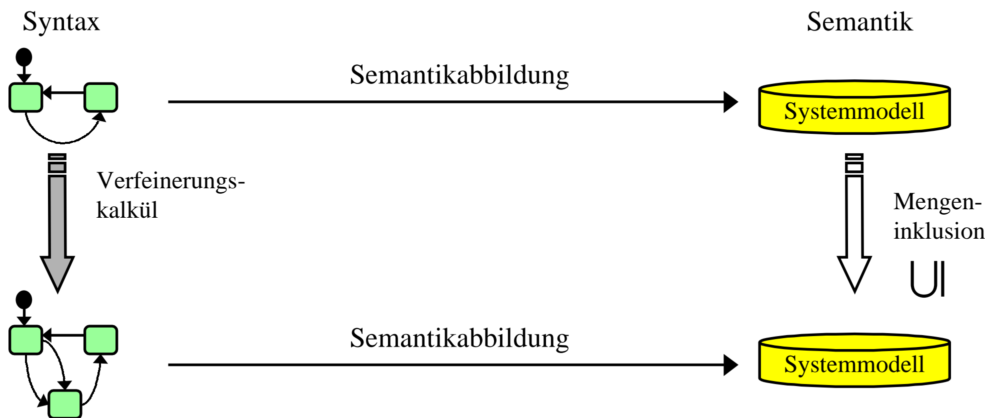


Abbildung 3: Verfeinerung und Semantik

4.1 Anwendungsbeispiel *Figure*

Vor der Definition der Verfeinerungsschritte wird an der Klasse *Figure* und ihrer Subklasse *2D-Figure* demonstriert, wie diese Schritte angewendet werden

und damit das Verhalten von Komponenten während einer Softwareentwicklung verfeinert und vererbt wird. Dazu erhalte ein *Figure*-Objekt Nachrichten vom Terminal, das Benutzereingaben darstellt, und gebe seinerseits Nachrichten an die Bildschirmanzeige weiter.

Die nachfolgend dargestellte Entwicklung spiegelt einen oft typischen Entwurfsprozeß wider, der zunächst bestimmte Festlegungen trifft (hier etwa die Einführung eines Fehlerzustands) und diese durch spätere Verfeinerungen wieder überflüssig macht. Diese etwas erratischen Verfeinerungen geben uns die Möglichkeit, alle wesentlichen Verfeinerungsschritte an diesem Beispiel zu demonstrieren. Die nachfolgende Entwicklung hätte auch in drei Schritten durchgeführt werden können.

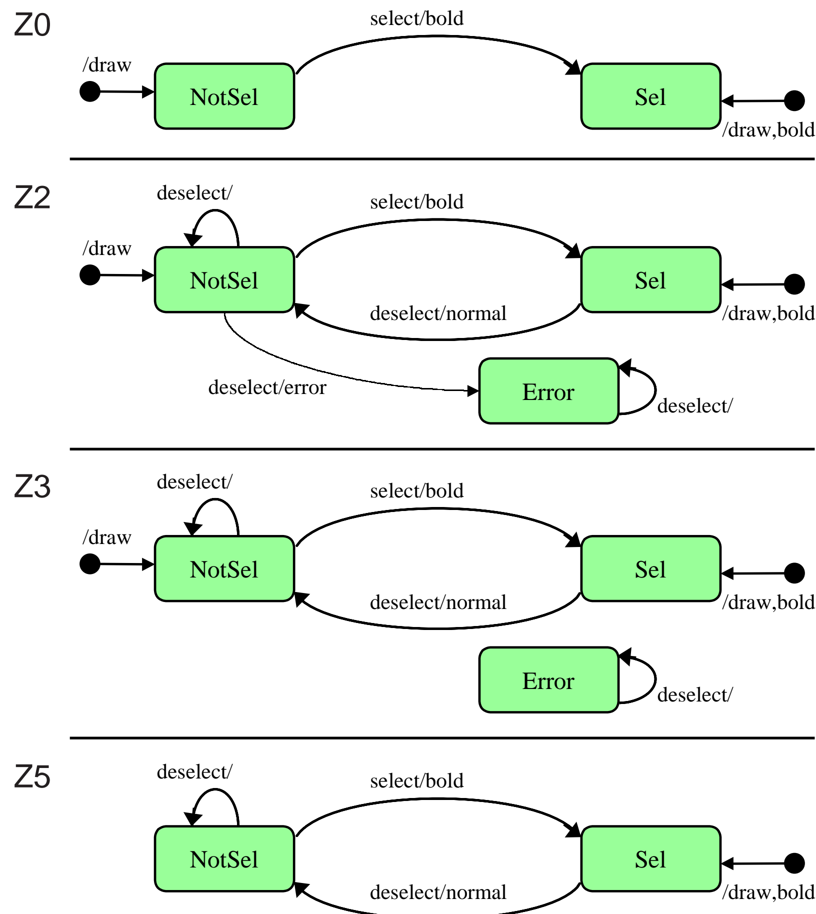


Abbildung 4: Verfeinerungen für Klasse *Figure*

0. Anfängliche Verhaltensbeschreibung: Wir starten unsere Entwicklung mit dem Zustandsdiagramm $Z0$ in Abbildung 4, das unser Wissen über das Verhalten von $Figure$-Objekten zum gegenwärtigen Zeitpunkt widerspiegelt. Die beiden Zustände *Sel* und *NotSel* reflektieren den Selektionszustand einer Figur. Weil wir im Moment nicht festlegen wollen oder können, in welchem Zustand ein Objekt anfangs ist, markieren wir beide Zustände als Initialzustände. Bei der Erzeugung der Figur ist diese zu zeichnen ($draw$). Im selektierten Zustand ist der entsprechende Fensterausschnitt hervorgehoben darzustellen (*bold*). Die einzige Transition dieses Automaten beschreibt, daß bei einer Selektion im unselektierten Zustand der Fensterausschnitt hervorgehoben wird, und ein Zustandsübergang stattfindet. Durch das Fehlen einer Transition zur Verarbeitung von *select* im Zustand *Sel* wird das Verhalten hier offen gelassen.

1. Verfeinerungsschritt: Neben der Selektion von Figuren ist auch deren Deselektion wichtig. Bevor wir entsprechende Transitionen einfügen, führen wir einen neuen Zustand *Error* ein, der eingenommen werden kann, wenn ein Bedienungsfehler, wie etwa Deselektion im $NotSel$-Zustand, auftritt. Der neue Zustand ist zunächst nicht erreichbar und trägt damit nicht zum Verhalten des Objekts bei (ohne Bild).

2. Verfeinerungsschritt: Wir führen nun die Nachricht *deselect* für die Deselektion einer Figur ein, und erweitern damit die Eingabemenge. Dann fügen wir einige Transitionen hinzu, die das Verhalten einer Figur bei Erhalt einer *deselect*-Nachricht beschreiben. Wir wissen nicht genau, welches Verhalten für *deselect* im *Sel*-Zustand gewünscht ist und geben daher zwei Alternativen an. Eine Transition ignoriert die Nachricht, eine andere Transition führt in den Fehlerzustand. Dies läßt einer Implementierung und jeder Subklasse beide Varianten offen (siehe $Z2$).

3. Verfeinerungsschritt: Der Anwender wünscht eine robuste Implementierung. Wir streichen die Möglichkeit, daß *deselect* in den Fehlerzustand führt. Dies ist erlaubt, weil mit der *deselect*-Schlinge im Zustand $NotSel$ eine Alternative zur Verfügung steht. Es handelt sich damit um eine genauere Festlegung des für $Figure$-Objekte möglichen Verhaltens im Zustand *Sel* (siehe $Z3$).

4. Verfeinerungsschritt: Der Fehlerzustand $Error$ ist im Diagramm unerreichbar. Wir dürfen ihn daher streichen. Er hätte also gar nicht eingeführt werden müssen (ohne Bild).

5. Verfeinerungsschritt: Als letzten Verfeinerungsschritt für das Verhalten der Objekte der Klasse $Figure$ beschließt der Anwender, daß eine Figur, wenn sie erzeugt wird, immer selektiert sein soll. Wir entfernen daher $NotSel$ aus der Menge der Initialzustände und schließen die Entwicklung für Klasse $Figure$ hier ab (siehe $Z5$).

Alle entwickelten Automaten beschreiben das den Klienten der Klasse *Figure* zugesicherte Verhalten. Jeder Automat verfeinert seinen Vorgänger und enthält so eine detailliertere Beschreibung des Verhaltens von Objekten der Klasse *Figure*. Klienten der Klasse *Figure* können jede dieser Abstraktionsstufen nutzen. Je weniger detailliert das Wissen eines Klienten über die *Figure*-Objekte ist, desto leichter kann die *Figure*-Implementierung geändert werden, ohne den Klienten zu modifizieren.
Das Substitutionsprinzip für Objekte fordert, daß jedes Objekt aus der Subklasse *2D-Figure* ebenfalls dieses Verhalten besitzt. Wir vererben daher die Verhaltensbeschreibung, die durch den letzten Automaten angegeben wird, an die Klasse *2D-Figure* und spezialisieren hier das Verhalten weiter.

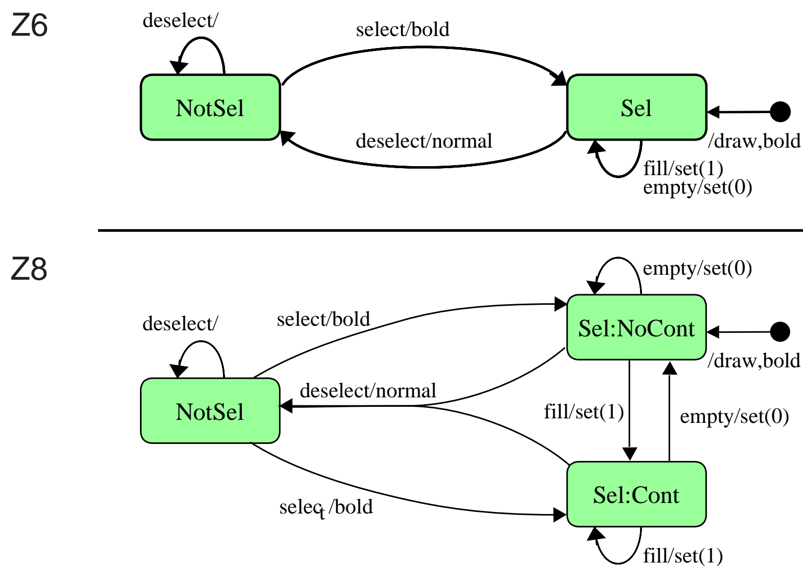


Abbildung 5: Verfeinerungen für Subklasse *2D-Figure*

<u>Vererbung und 6. Verfeinerungsschritt</u>: Durch die Vererbung des Automaten an die Subklasse *2D-Figure* wird dessen Eingabemenge um die Nachrichten *fill* und *empty* erweitert. *2D*-Figuren besitzen eine Fläche, deren Inhalt gefüllt und geleert werden kann. Dazu werden die Nachrichten *fill* und *empty* im selektierten Zustand akzeptiert und deren Verarbeitung verläßt diesen Zustand nicht. Die Methode *set* wird an die Bildschirmanzeige weitergeleitet und dient zum Füllen bzw. Leeren des Inhalts einer 2D-Figur (siehe *Z*6 in Abbildung 5).
<u>7. Verfeinerungsschritt</u>: Die Reaktion auf *fill* und *empty* kann mit dem momentanen Automatenzustandsraum nicht detailliert wiedergegeben werden. Deshalb führen wir eine Zustandsverfeinerung des Zustands *Sel* in zwei neue Zustände *Sel:NoCont* und *Sel:Cont* durch, die neben der Selektion vermer-

ken, ob der Inhalt der zweidimensionalen Figur gefüllt (*Sel:Cont*) oder leer (*Sel:NoCont*) ist. Dabei werden ankommende und ausgehende Transitionen vervielfacht. Zum Beispiel entstehen aus der einen *fill*-Transition vier neue Transitionen. Weil der alte Zustand Initialzustand war, werden beide neuen Zustände ebenfalls Initialzustände (ohne Bild).
8. Verfeinerungsschritt: Durch die Teilung der Transitionen ist neuer Nichtdeterminismus im Zustandsdiagramm entstanden, der nun zur Präzisierung des Verhaltens genutzt werden kann. Es werden einige Transitionen und ein Initialelement entfernt. Dadurch wird festgelegt, daß neu erzeugte zweidimensionale Figuren zunächst nicht gefüllt aber selektiert sind, und daß *fill* und *empty* tatsächlich adäquate Zustandsänderungen verursachen (siehe $Z8$).
Die hier beschriebenen Entwicklungsschritte reichen aus, um die Flexibilität und die Mächtigkeit des in diesem Kapitel definierten Verfeinerungskalküls zu demonstrieren. Das Beispiel zeigt, daß damit echte Softwareentwicklungen betrieben werden können, es zeigt aber auch, daß für größere Entwicklungen Werkzeugunterstützung erforderlich ist.

4.2 Der Regelsatz

In diesem Abschnitt wird der Satz von Regeln kurz vorgestellt, der zur Verfeinerung von Zustandsdiagrammen verwendet werden kann. Für jede dieser Regeln existiert relativ zur skizzierten Beobachtungssemantik ein Korrektheitsbeweis. Der Kalkül wurde nach Kriterien der praktischen Anwendbarkeit entworfen. Er ist deshalb zwar nicht im theoretischen Sinne vollständig, dürfte aber für praktische Belange ausreichend sein. Insbesondere die zielgerichtete Kombination der Regeln führt zu einer mächtigen Verfeinerungstechnik.

Entfernung von Initialzuständen ist erlaubt, solange wenigstens ein Initialzustand existiert. Damit wird das Anfangsverhalten genauer festgelegt.

Entfernung von Transitionen ist möglich, wenn zu den entfernten Transitionen alternative Transitionen existieren, die dieselbe Eingabe in dem selben Quellzustand verarbeiten. Auf diese Weise werden Auswahlmöglichkeiten entfernt, und so das Verhalten genauer festgelegt.

Hinzufügen von Transitionen ist nur erlaubt, wenn bisher noch keine Transitionen existiert haben, die dieselbe Eingabe im selben Zustand verarbeitet haben, denn dann hat bisher völlige Unterspezifikation (Chaos) gegolten.

Entfernung unerreichbarer Zustände Eine Gruppe von außen nicht erreichbarer Zustände trägt nicht zum Verhalten bei und kann entfernt werden.

Hinzufügen neuer Zustände ist immer möglich, da diese zunächst unerreichbar sind. Zu beachten ist allerdings, daß neue Automatenzustände auch neuen Äquivalenzklassen von Objektzuständen entsprechen.

Teilen von Zuständen wirkt wie das Teilen von Äquivalenzklassen von Objektzuständen. Es erlaubt die anschließende detailliertere Verhaltensbeschreibung durch Manipulation der Transitionen.

Erweiterung der Eingabe durch neue Nachrichten ist zum Beispiel für die Vererbung oder die Erweiterung einer Schnittstelle sinnvoll.

Weitere in [10] definierte Regeln beschäftigen sich mit der Frage unendlicher Ausgaben sowie komplexerer Transformationen, die aus diesen Basistransformationen gebildet werden können. Einige davon sind:

Einengung eines Zustandsprädikats wird benutzt, um bisher in dem Zustandsprädikat enthaltene Datenzustände zu entfernen. Dies ist nur möglich, wenn gesichert ist, daß keine ankommende Transition einen solchen Datenzustand einnehmen will.

Verschärfung der Nachbedingung einer Transition wird benutzt, um eine Verhaltensbeschreibung detaillierter und deterministischer zu machen.

Teilung von Transitionen eignet sich, um eine anschließende Bearbeitung der entstandenen Transitionen vorzubereiten.

Vervollständigung des Zustandsraums bietet die Möglichkeit, alle noch nicht in explizit aufgenommenen Datenzustände in einem eigenen Automatenzustand darzustellen, und bildet die Grundlage für eine robuste Verhaltensvervollständigung.

4.3 Anwendung von Zustandsdiagrammen

Jedes Zustandsdiagramm ist einer Klasse zugeordnet. Es beschreibt das Verhalten der zu dieser Klasse gehörenden Objekte. Jedoch gibt es verschiedene Einsatzmöglichkeiten für Zustandsdiagramme.
Ein Zustandsdiagramm kann von einer Klasse an deren Subklassen vererbt werden. Dann müssen die Objekte der Subklassen das in der Oberklasse festgelegte Verhalten ebenso besitzen. Diese können allerdings unterschiedliche Spezialisierungen nutzen und erlauben so eine Vielfalt möglicher Implementierungen mit einem gemeinsamen abstrakten Verhalten.
In einem Framework bilden sogenannte „Hot Spots“ die Punkte, in denen selbstdefinierte Methoden bzw. Objekte eingesetzt werden können. Mangels einer adäquaten Beschreibungsform für das Verständnis des möglichen Verhaltens solcher Hot Spots ist bis jetzt entweder eine Inspektion des Quellcodes des

Frameworks oder ein Try-And-Error-Verfahren notwendig. Zustandsdiagramme und die hier vorgestellte Technik der Verfeinerung bieten ein adäquates Mittel zur Definition von Soll-Vorgaben für solche Hot Spots.
Der Einsatz von Zustandsdiagrammen bietet sich insbesondere zur Definition von Verhaltens-Schnittstellen an. So können mehrere verschiedene Schnittstellen definiert und diese unterschiedlichen Klienten zur Verfügung gestellt werden. Eine Implementierung ist dann eine gemeinsame Verfeinerung dieser Schnittstellen.
Automobilfirmen und andere Produzenten technischer Produkte können mit Hilfe solcher Schnittstellenbeschreibungen den Zulieferern nicht nur strukturelle Vorgaben, sondern auch Vorgaben im Verhaltensbereich einer Steuerkomponente machen und von den Zulieferern durch Vorlage des Verfeinerungswegs auch deren Korrektheit demonstrieren lassen. Hierzu ist eine die Verwendung der gezeiteten Variante des Kalküls aus [10] sinnvoll.
In [11] werden ähnliche Automaten nicht zur Verhaltensspezifikation, sondern zur Implementierungsbeschreibung eingesetzt. Dazu werden statt Prädikaten konkrete Anweisungen einer Implementierungssprache verwendet. Bei größeren Systemen führt dies aber gerne zu großen und unübersichtlichen Zustandsdiagrammen. Die umgekehrte Anwendung des Verfeinerungskalküls führt in diesem Fall zur *Extraktion* einer Abstraktion des Automaten, der zwar nicht alle Verhaltensdetails wiedergibt, stattdessen aber einen guten Überblick über das Verhalten bietet. Damit kann der Verfeinerungskalkül auch zum Reengineering eingesetzt werden.

5 Verwandte Arbeiten

In vielen objektorientierten Methoden [12] werden Automaten als Verhaltensbeschreibung einzelner Objekte eingesetzt, jedoch werden wegen der informellen Semantik der in diesen Softwareentwicklungsmethoden benutzten Beschreibungstechniken keine Anhaltspunkte über die Beziehung zwischen Automaten von in Vererbungsbeziehung stehenden Klassen gemacht. Ein zu dem hier vorgestellten konzeptionell ähnlicher Ansatz wird in Troll [3] verfolgt. Dort wird der Lebenszyklus einer Klasse nicht als Zusicherung eines bestimmten, garantierten Verhaltens, sondern hauptsächlich als Restriktion des möglichen Verhaltens, verstanden. Deshalb unterscheiden sich auch die Bedingungen bei der Vererbung eines Lebenszyklus-Diagramms deutlich. Ein interessanter Ansatz wird von Nierstrasz in [7] verfolgt. Dort werden endliche Automaten als Typisierung verwendet, um zu beschreiben, in welchem Zustand ein Objekt welche Nachrichten akzeptiert. In [8] wurde eine Vorarbeit dieses Kalküls veröffentlicht, die statt Ausgaben den Zustand als Beobachtungsbegriff nutzt.

6 Zusammenfassung und Ausblick

In [10] wurde für den Entwurf verteilter, objektorientierter Systeme eine formale Methodik entwickelt, die sich an in der Praxis eingesetzten Techniken orientiert. Dazu wurden Zustandsdiagramme, Klassendiagramme, Klassensignaturen und Datentypdokumente mit konkreter Darstellungsform, abstrakter Syntax und präziser Semantik definiert. Ein Datentypdokument definiert die in einem System benutzten Basisdatentypen und deren Funktionen. In einer Klassensignatur wird die Signatur einer Klasse in Form einer Menge von Methoden und Attributen festgelegt. Ein Klassendiagramm beschreibt die Struktur des Systems durch Daten- und Vererbungsbeziehungen zwischen Klassen. Ein Zustandsdiagramm beschreibt schließlich das Verhalten von Objekten einer Klasse auf unterschiedlichen Abstraktionsstufen.

Die Beschreibungstechniken besitzen eine präzise Semantik auf Basis des hier skizzierten Systemmodells. Für Zustands- und Klassendiagramm wurden Entwicklungsschritte als syntaktische Transformationen definiert, deren semantische Korrektheit durch formale Beweise gezeigt wurde. Diese Technik geht weit über bisherige Ansätze zur Formalisierung graphischer Beschreibungstechniken hinaus. Die in [10] definierte Methodik demonstriert, daß in der Praxis eingesetzte graphische Beschreibungstechniken mit formalen Ansätzen kombiniert und so die Vorteile beider Welten vereint werden können.

Die Definition der Zustandsdiagramme wurde in zwei Schritten vorgenommen. Deshalb ist als selbständiger Teil der Arbeit die *Theorie buchstabierender Automaten* entwickelt worden. Sie bildet das Bindeglied zwischen der im Systemmodell benutzten Verhaltensbeschreibung durch Ströme und der konkreten Syntax der Zustandsdiagramme. Buchstabierende Automaten bilden einen zu den I/O-Automaten [5] alternativen Ansatz. Während I/O-Automaten pro Transition genau eine Ausgabe oder eine Eingabe besitzen, können die Transitionen buchstabierender Automaten mit einer Eingabe und einer ganzen Sequenz von Ausgaben attributiert werden. Das führt zu einem wesentlich kompakteren Zustandsbegriff (keine Zwischenzustände notwendig) und erlaubt es auf Fairneß-Mengen zu verzichten. Buchstabierende Automaten sind daher für die Beschreibung objektorientierter Systeme wesentlich adäquater als I/O-Automaten.

Seit dem Abschluß der Dissertation wurde eine Implementierung des Verfeinerungskalküls im Rahmen eines am Lehrstuhl Broy entwickelten CASE-Tool-Prototyps durchgeführt [2]. Eine Anwendung des Verfeinerungskalküls auf in der Telekommunikation auftretende Probleme der Feature-Interaction [4] zeigt, daß auch hier Anwendungsgebiete existieren. Mittlerweile wurde die in [10] entwickelte Idee des Verfeinerungskalküls auch auf Datenflußdiagramme übertra-

gen [9] und im Kontext mit der Unified Modeling Language diskutiert [6].

Danksagung

Für interessante Gespräche und Unterstützung für die Dissertation danke ich meinen Kollegen und insbesondere meinem Doktorvater Prof. Dr. Manfred Broy und meinem Zweitgutachter Prof. Dr. Manfred Paul. Diese Arbeit ist Teil des durch den Leibniz Preis der DFG und Siemens geförderten SysLab-Projekts.

Literatur

[1] M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, and R. Weber. The Design of Distributed Systems — An Introduction to FOCUS. SFB-Bericht 342/2-2/92 A, Technische Universität München, January 1993.

[2] M. Fahrmair and B. Rumpe. Frisco STDA - Ein Werkzeug zur methodischen Bearbeitung von Automaten. Technical Report TUM-I9815, Technische Universität München, 1998.

[3] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. Object-Oriented Specification of Information Systems: The Troll Language (Version 0.01). Informatik-Berichte 91-04, Technische Universität Braunschweig, Dec. 1991.

[4] C. Klein, C. Prehofer, and B. Rumpe. Feature specification and refinement with state transition diagrams. In P. Dini, editor, *Fourth IEEE Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems*. IOS-Press, 1997.

[5] N. Lynch and E. Stark. A Proof of the Kahn Principle for Input/Output Automata. *Information and Computation*, 82:81–92, 1989.

[6] P. Muller and J. Bezivin. *Proceedings of UML'98*. Springer-Verlag, LNCS, to appear, 1998.

[7] O. Nierstrasz. Regular Types for Active Objects. In A. Paepke, editor, *OOPSLA '93*. ACM Press, Oct. 1993.

[8] B. Paech and B. Rumpe. A new Concept of Refinement used for Behaviour Modelling with Automata. In *FME'94, Formal Methods Europe, Symposium '94*, LNCS 873. Springer-Verlag, Berlin, Oct. 1994.

[9] J. Philipps and B. Rumpe. Refinement of information flow architectures. In M. Hinchey, editor, *ICFEM'97 Proceedings, Japan*. IEEE CS Press, 1997.

[10] B. Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert-Utz Verlag Wissenschaft, München, 1996.

[11] B. Selic, G. Gulkeson, and P. Ward. *Real-Time Object-Oriented Modeling*. John Wiley and Sons, 1994.

[12] UML Group. Unified Modeling Language. Version 1.2, Rational Software Corporation, Santa Clara, CA-95051, USA, June 1998.