# UML-A oder warum die Wissenschaft ihre eigene einheitliche Modellierungssprache haben sollte

Friedrich Steimann

Institut für Informationssysteme
Universität Hannover
Appelstraße 4
D-30167 Hannover
steimann@acm.org

**Abstract:** Mit der Standardisierung der Unified Modeling Language (UML) wurde der Versuch unternommen, für die Softwareentwicklung eine normierte graphische Spezifikationsform einzuführen, die für alle Projektbeteiligten verständlich ist und somit eine Überprüfung von Vollständigkeit und Korrektheit der darin festgehaltenen Produktanforderungen sowie deren Umsetzung erlaubt. So begrüßenswert diese Initiative ist, so ernüchternd ist das Ergebnis: Obwohl das Bemühen, Syntax und Semantik von UML unmißverständlich festzuschreiben, klar erkennbar ist, ist der Standard voller Ungenauigkeiten, Ungereimtheiten und sogar Widersprüche. Erklärungen dafür gibt es viele — hingenommen werden kann es hingegen nicht. Da von Seiten der Autoren des Standards aufgrund vielfacher Interessenkonflikte nur bedingt Besserung zu erwarten ist, muß es, um Gottlob Frege frei zu zitieren, der Wissenschaft erlaubt sein, sich für einen bestimmten Zweck ihre eigene Sprache zurechtzufeilen, wenn sie eine wenig geeignete vorfindet. Dieser Beitrag soll zu einem konzertierten ersten Schritt in diese Richtung aufrufen.

## 1 Einleitung

Als Gottlob Frege gerade den zweiten Band seines Werkes „Die Grundgesetze der Arithmetik“, einer Grundlegung der Mathematik in Logik und Mengenlehre, aus dem Druck abholte, bekam er (am 16. Juni 1902) einen Brief von Bertrand Russell, der ihn freundlich, aber schonungslos auf einen darin enthaltenen Fehler hinwies: die nach Russell benannte Antinomie von der Menge der Mengen, die sich nicht selbst enthalten. Frege sah den Fehler sofort ein, wollte aber keine der von Russell vorgeschlagenen Lösungen akzeptieren und fand auch keine eigene, die ihn zufriedengestellt hätte. Er starb, ohne einen konsistenten Satz von Axiomen vorgelegt und damit sein Lebenswerk vollendet zu haben. Dennoch gelten seine Grundgesetze der Arithmetik heute als die erste vollständig formalisierte Fassung der Prädikatenlogik erster Ordnung und Frege selbst als der Vater der modernen Logik. [vK89]

Im Jahr 2003 erhielt die internationale Softwaremodellierungsgemeinde von der Object Management Group (OMG) eine Vorabversion des UML-Standards in der Version 2.0

gemeinsam mit einer Frist (7.11.2003), innerhalb der sog. „issues“ gemeldet werden konnten. Wir durften uns also alle angesprochen fühlen, die Rolle des Russell zu übernehmen und uns auf die Fehlersuche zu begeben, wenn wir uns denn die Mühe machen wollten, uns durch das gut 800 Seiten (ohne OCL 2.0 und XMI) starke Werk durchzuarbeiten. Daß dies keine ganz einfache Arbeit ist, liegt u. a. daran, daß der neue Standard das Produkt eines jahrelangen Einigungsprozesses verschiedenster Interessenvertreter ist und somit alle Eigenschaften sowohl einer synthetisierten als auch einer Gremiensprache in sich vereinigt. Daß er auch noch die Versionsnummer 2 trägt, erinnert zudem in geradezu fataler Weise an Fred Brooks' Second System Effect.

Die OMG als normierendes Gremium ist ein nichtakademisches Konsortium, das sich die Standardisierung und Verbreitung objektorientierter Technologie zum Ziel gesetzt hat. Naturgemäß besteht ein Interesse der daran Beteiligten darin, mit dem Standard Geld zu verdienen, sei es durch frühe Einbringung in eigene Produkte oder Dienstleistungen, durch Beratung oder durch den Verkauf von Werkzeugen. Das Geschäft kann aber nicht von allen gemacht werden – das öffentliche Interesse wird somit allenfalls indirekt (z. B. durch eine tatsächliche Verwendbarkeit des Standards und den damit verbunden allgemeinen Nutzen) bedient.

Es wäre naiv, zu behaupten, daß ein akademisches UML nicht auch zunächst vor allem private Interessen bedienen würde: Forscher leben vom Wettbewerb der Ideen, und wessen Ideen sich durchsetzen, der bekommt in der Gemeinde einen Namen. Dies u. a. erklärt, warum sich die akademischen Welt bislang noch nicht auf eine Modellierungssprache für die Softwareentwicklung einigen konnte: Zu sehr sind die meisten von uns mit der Verfolgung eigener Interessen beschäftigt.[1]

Die bedauerliche Folge dieser Situation ist, daß wir heute mehr oder weniger ohnmächtige Zeugen einer Entwicklung sind, in der ein evidenter Bedarf, dessen Stillung ein öffentliches Interesse darstellt — nämlich die *Definition einer einheitlichen, unmißverständlichen und brauchbaren Sprache für die Softwaremodellierung* —, durch einige wenige Vertreter vorwiegend privater Interessen bedient wird, die schon durch ihren Einfluß dafür sorgen können, daß das Ergebnis auch akzeptiert werden wird. Paradoxerweise sind mit dem Resultat nicht einmal die Verantwortlichen selbst zufrieden – daran zumindest ließen die öffentlichen Äußerungen der anwesenden Vertreter der OMG auf der letztjährigen UML-Konferenz (UML 2003) keinen Zweifel. Wie zielführend nimmt sich dagegen die Arbeit von Frege und Russell aus.

Ziel des vorliegenden Beitrags ist, aufbauend auf einer Darstellung des derzeitigen Stands der UML-Definition und der vorgesehenen Verwendung der Sprache (Abschnitt 2) einen Katalog von Eigenschaften zu formulieren, denen eine formale Basis der Softwaremodellierung in Wirtschaft und Forschung genügen sollte (Abschnitt 3). Diese Eigenschaften werden hier zugunsten einer etwas ausführlicheren Problemdarstellung nur skizziert; es ist jedoch zu erwarten, daß schon die vage Andeutung eines solchen Katalogs Anlaß zu kontroverser Diskussion bieten wird.

[1] Ein anderer Grund liegt natürlich darin, daß dies insgesamt ein nichttriviales Unterfangen ist.

# 2 Ausgangslage: UML heute

## 2.1 UML als Sprachfamilie und Einordnung von UML in die Familie der Sprachen

Das Besondere an Freges Prädikatenlogik ist, daß sie eine Basis darstellt, zu der es kaum Alternativen gibt. Die ontologische Zweiteilung in Objekte und Prädikate, die Aussagen über Objekte treffen, ist nicht nur kaum noch weiter zu reduzieren, sondern sie findet sich auch in der Struktur unserer (wenn vielleicht auch nicht aller) natürlichen Sprache wieder. Die Prädikatenlogik ist sparsam im besten Sinne: Mit einer kleinen Anzahl von Konzepten kann ungeheuer viel ausgedrückt werden.

Am anderen Ende der Liste von Sprachen, die für die Softwareentwicklung von Bedeutung sind, steht die Sprache der Turing-Maschine: Mit einer wiederum kaum zu reduzierenden Anzahl von Konzepten kann alles ausgedrückt werden, was sich mit einem Computerprogramm ausdrücken läßt. Der größte Nachteil der Turing-Maschine ist ihre technische Unpraktikabilität; ihr nächster Verwandter, der RISC-Mikroprozessor, kann jedoch (zumindest was den Befehlssatz angeht) immer noch als so primitiv angesehen werden, daß es sich nicht lohnt, über eine weitere Reduktion nachzudenken.

Zwischen Prädikatenlogik als deklarativer Sprache der Problemdefinition und dem Turing-Formalismus als imperativer Sprache der Problemlösung tut sich ein kaum zu übersehender Wust von nichtminimalen Sprachen auf, die für den einen oder anderen Zweck entworfen wurden und mehr oder weniger gut dazu geeignet sind. Wenn man nun die Modellierungssprachen den Programmiersprachen gegenüberstellt, so drängt sich die Annahme auf, daß erstere, was die Nähe zur deklarativer Problemdefinition und Abstraktion vom imperativ formulierten Lösungsweg angeht, in Abbildung 1 über den Programmiersprachen stehen. Die Programmiersprachen der 4. und 5. Generation haben ihren Bereich jedoch stark in Richtung Abstraktion ausgedehnt, während (z. B. mit UML 2.0) die Modellierung immer weiter in das Gebiet der Programmierung eindringt. Eine strikte Trennung scheint also weder möglich noch sinnvoll.

Das besondere an UML ist, daß es im zweidimensionalen Raum zwischen abstrakten /konkreten und reichen/armen (was die Anzahl der Konzepte betrifft) Sprachen ein Fläche einnimmt: UML soll nämlich nicht als eine einzelne, sondern als eine ganze Familie von Sprachen verstanden werden. Dies ist zum Teil dem universellen Anspruch der UML geschuldet, der sie eine beträchtliche Anzahl historisch kaum miteinander verwandter Teilmodellierungssprachen unter einen Hut zu bringen zwingt, zum Teil aber auch der Tatsache, daß verschiedene Nutzer der UML nur Teile der Sprache gebrauchen und sich ihr eigenes UML konfigurieren können wollen. Und nicht zuletzt muß es nach Auffassung der OMG in einer universellen Modellierungssprache möglich sein, bei Bedarf neue Sprachelemente hinzuzufügen (zur Zeit über die sog. Profiles). Eine Einordnung von UML als Sprachfamilie in die Familie der Sprachen vor diesem Hintergrund ist in Abbildung 1 skizziert.

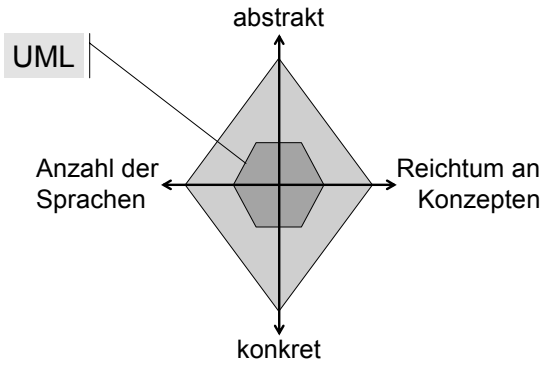

Abbildung 1: UML (schattierte Fläche) im Spektrum der Sprachen

### 2.2 MDA oder des Kaisers neue Kleider

Von Anbeginn der objektorientierten Modellierung war eins der zentralen Versprechen, daß sie den nahtlosen Übergang von Analyse zu Design zu Implementierung erlaube. Man ging davon aus, daß ein Modell über verschiedene Ebenen der Abstraktion bis hin zu einem fertigen Programm entwickelt werde könne. Dazu sei es lediglich notwenig, zu den Objekten, die die Elemente der abgebildeten Realität repräsentieren, zunächst Designobjekte (Objekte, die durch den Entwurf bedingt werden) und später Implementationsobjekte (solche, die eine Umsetzung des Designs in Code realisieren) hinzuzufügen. Die Betonung liegt hierbei auf hinzufügen, denn die ursprünglichen Analyse- und Designartefakte wären im fertigen Code alle wiederzufinden. Diese These hat sich wohl bewahrheitet, auch wenn man vielleicht die Zahl der Objekte (Klassen), die auf diese Weise bei einer kommerziellen Softwareentwicklung zusammenkommen, und die damit verbundenen strukturellen Probleme (hauptsächlich verursacht durch das Fehlen eines Teile-und-herrsche-Konzepts in der Organisation von Objektstrukturen) unterschätzt hat [SGM03].

Vor diesem Hintergrund mag es erstaunen, daß die OMG heute die sog. Model-Driven Architecture (MDA) als den notwendigen nächsten Schritt propagiert.[2] Dies mag als Reaktion darauf verstanden werden, daß Modelle in der Praxis vielerorts — vielleicht aus einer gefühlten Verpflichtung heraus — zwar zunächst erstellt werden, sich die weitere Softwareentwicklung aber schnell von den Modellen löst, da die eigentliche Arbeit und die damit verbundenen Probleme nicht auf konzeptueller Seite liegen, sondern in der Beherrschung der und Anpassung an die extrem komplexen Rahmentechnologien (wie z. B. die sog. Middleware oder auch — und immer noch — die Konstruktion geeigneter Benutzungsschnittstellen). Es ist aber nichtsdestoweniger ein deutlicher Indikator dafür, daß es mit der nahtlosen Umsetzung bislang nicht so recht geklappt hat, denn sonst müßten sich die Analyse- und Designmodelle ja ohne größeren Aufwand — wenn nicht so-

[2] Die Idee hinter MDA ist, Code als primäres Artefakt der Softwareentwicklung durch Modelle abzulösen, die (zusammen mit geeigneten Transformationsdirektiven) genug Informationen enthalten, um den Code daraus generieren zu können, diesen also zum abgeleiteten Artefakt zu machen. Dabei soll insbesondere (und wieder einmal) das Problem der Plattform(un)abhängigkeit gelöst werden.

gar automatisch — an eine geänderte Implementierung anpassen lassen. Die Bedeutung der Modellierung für die Softwareentwicklungspraxis hat also offensichtlich insgesamt nicht das Ausmaß erreicht, das man erwartet hatte (und mit dem man sich bei der OMG zufriedengeben könnte).

Um die objektorientierte Modellierung zu stärken, ist es notwendig, den Nutzen der Modelle für die Praxis zu steigern. Dies kann am wirksamsten erreicht werden, indem der Übergang von Modellen zu Code systematisiert, am besten sogar automatisiert wird. Transformation ist hier das Stichwort[3], die systematische (d. h. wiederholbare) Überführung von (plattformunabhängigen) Modellen in (plattformabhängigen) produktionsreifen Code das ehrgeizige Ziel. Sieht man einmal von der angestrebten Automatisierung ab, handelt es sich hierbei jedoch lediglich um eine Wiederholung des ursprünglichen Versprechens vom nahtlosen Übergang von der Problembeschreibung durch ein Modell zu dessen Umsetzung durch ein Programm.

Es ist aber gerade die Automatisierung, die Anlaß zur begründeten Skepsis gibt. Für eine automatische Transformation im Sinne einer Codegenerierung gibt es nur zwei Möglichkeiten: Entweder sie ist semantikerhaltend, d. h., sie stellt lediglich eine Abbildung von einem Formalismus in einen anderen dar, ohne jedoch an der Bedeutung des Dargestellten etwas zu verändern, oder sie ist kreativ, fügt also Dinge hinzu.[4] Wenn sie aber automatisch (also ohne Eingriff von außen) Dinge hinzufügen soll, dann kann sie das nur zufällig tun, denn jeder systematische Ansatz müßte auf Information gründen, die im Modell enthalten ist, und wäre damit wiederum lediglich semantikerhaltend.[5]

Wie also kann die von den Proponenten der MDA als „semantic gap“ bezeichnete Lücke zwischen Modell(ierungssprache) und Programm(iersprache) geschlossen werden? Doch nur, indem man Modelle mit den Informationen anreichert, die zur automatischen Codegenerierung gebraucht werden. Damit jedoch degeneriert (oder entwickelt sich, je nach Sichtweise) die Modellierung zur (grafischen) Programmierung; folgerichtig spricht man bei den Generierungswerkzeugen auch von Modellcompilern.

An dieser Stelle sind wir jedoch bei einem anderen Versprechen angelangt, daß nur zum Teil in Erfüllung gegangen ist, nämlich dem der Programmiersprachen der vierten Generation: Durch die Einführung mächtiger Sprachkonstrukte sollte die Programmierung komplexer Systeme um Größenordnungen kompakter werden, z. B. indem atomare Anweisungen ganze Algorithmen ersetzen. Das Problem dieser Sprachen ist jedoch, daß ihre Mächtigkeit den Preis der Flexibilität hat: Allzuoft macht ein vorgesehenes Sprachkonstrukt nur ungefähr das, was man eigentlich braucht — die notwendige Modifikation ist dann entweder nicht möglich oder so aufwendig, daß man schnell zu der Einsicht gelangt, daß man mit einer Sprache der dritten Generation genauso schnell ans Ziel gelangt wäre.

---

[3] und Codegenerierung der Vorläufer, der aber nicht gern erwähnt wird, vielleicht weil es sich inzwischen herumgesprochen hat, daß es damit in der Praxis nicht weit her ist

[4] Daß sie Sachen unterschlägt, ist eine theoretische dritte Möglichkeit, die aber in diesem Kontext keinen Sinn ergibt.

[5] Daß die Transformation intelligent ist in dem Sinne, daß sie das modellierte Problem (er)kennt und Aufgrund dessen Kenntnis und ihres Domänenwissens die richtigen Ergänzungen von selbst findet, sei hier ausgeschlossen.

Es ist vom heutigen Standpunkt aus nicht zu erkennen, wie dieses Problem mit MDA gelöst werden soll. Vielmehr ist zu erwarten, daß in Modelle Codesegmente (in einer plattformunabhängigen Programmiersprache wie z. B. einer sog. Action Language [MB02]) eingeflickt werden müssen, die die zur Ausführung fehlende Information enthalten. Die Modellierungssprache wird damit zu einer Programmiersprache mit grafischen Elementen.[6] Dies ist an sich nichts Schlechtes — die Frage ist nur, warum sich die softwareentwickelnde Welt mit *einer* solchen Programmiersprache (nämlich UML) zufrieden geben sollte, wenn sie in der Vergangenheit Hunderte hervorgebracht und bis heute keinen Konsens darüber erzielt hat, welche denn die beste sei und dementsprechend verwendet werden solle.

### 2.3 UML als Papiertiger

Kennzeichnend für das Schaffen der OMG ist, daß sie Spezifikationen regelmäßig vor der Entwicklung eines ersten funktionierenden Werkzeugs, das die Spezifikation umsetzt, veröffentlicht. Dies mag für die Definition von Standards die übliche Vorgehensweise sein, setzt aber voraus, daß man das Problem und seine Lösung zur Gänze verstanden hat. Gerade dies ist jedoch bei Softwareprojekten (und als solches darf die Definition eines Modellierungsstandards durchaus gesehen werden) selten der Fall. Statt dessen fördert jeder Versuch der Umsetzung des Standards neue Aspekte zutage, die diesen als über-, unter- oder schlichtweg als falsch spezifiziert erscheinen lassen.

Anders als bei der Normierung von Programmiersprachen, die immer eindeutig sein muß, führt die von der OMG durchaus gewollte Unterspezifikation der UML dazu, daß verschiedene Werkzeughersteller verschiedene Varianten des Standards definieren. Dadurch ist jedoch eine Rückübertragung der von den Werkzeugherstellern als sinnvoll befundenen Festlegungen auf den Standard (die Sprachspezifikation), wenn nicht unmöglich, so doch extrem heikel, da sie praktisch immer mit der Begünstigung eines und der Benachteiligung anderer Werkzeughersteller verbunden ist. Ein Ausweg aus dieser vertrackten Situation ist nur schwer zu finden. Am ehesten wäre noch die Einrichtung eines Open-Source-Projekts vorstellbar, in dem ein UML-Werkzeug synchron mit dem Standard entwickelt wird. Kommerzielle Werkzeughersteller stünden dann jedoch vor der Aufgabe, einen Mehrwert zu finden, der nicht Gegenstand der Standardisierung ist; die zu erwartenden Margen würden wohl entsprechend schrumpfen, das wirtschaftliche Interesse ist entsprechend gering.

---

[6] Tatsächlich scheint die einzig wirklich notwendige Ergänzung die Zuweisung zu sein, für die es keine grafische Entsprechung gibt (warum nicht?); allerdings ist der Ausdruck anderer gängiger Elemente von Programmiersprachen (wie z. B. der üblichen Kontrollstrukturen) grafisch ungleich viel platzaufwendiger darzustellen als ihr textuelles Äquivalent. Insofern ist auch die Ergänzung von UML 2.0 um Struktogrammen entsprechende grafische Darstellungen von Kontrollfluß in Sequenzdiagrammen eine zweifelhafte Errungenschaft.

## 3 Ein Programm für die Zukunft

Was also kann die akademische Gemeinde an dieser unbefriedigenden Situation ändern? Sie kann weiter versuchen, durch Einflußnahme auf die Mitglieder der OMG in Richtung einer Reparatur der Sprache hinzuwirken, oder sie kann beginnen, an einer Alternative zu arbeiten. Beide Möglichkeiten sind nicht ohne Probleme: Zum einen sieht sich die OMG durch das wenig koordinierte Vorgehen der Wissenschaft einer Flut von individuellen Verbesserungsvorschlägen ausgesetzt, die sich, zumal sie in der Regel auch nur Ausschnitte der UML betreffen, nur schwerlich berücksichtigen und in ein sinnvolles Ganzes integrieren lassen, zum anderen ist jede Segregation, bei der die Abtrünnigen nicht einen gemeinsamen Kurs verfolgen, auf lange Sicht zum Scheitern verurteilt, insbesondere dann, wenn sie für potentielle Anhänger mit erheblichem Anpassungsaufwand verbunden ist.

Wie also könnte ein Ausweg aus diesem Dilemma aussehen? In der aktuellen März-Ausgabe der Communications of the ACM schrieb Dennis de Champeaux in einem Leserbrief zu einer zuvor erschienene Reihe von Beiträgen zu UML 2.0:

»*The entirety of UML is not required for dealing with problem understanding. A subset of UML—call it UML-A (for UML OO Analysis)—should be used to "rewrite" the unformalized requirements document into a formal version, without committing to how the system would operate. UML-A should be simple enough that the sponsor helps validate that the model captures the intent expressed by the requirements; for example, the sponsor should be able to confirm that use cases formulated in plain English (and optionally captured in diagrams) are faithfully represented in interaction diagrams and scenario diagrams.*

*Being able to extend a UML-A output model to address design issues and committing to how the target system will operate is another story.*

*The executability of a UML model is very handy. High-level executability opens the door for iterated smart compilation in conjunction with meaning-preserving transformations, ultimately leading to (semi)automatic coding. Thus, the code would be guaranteed to correspond to the high-level executable model.*

[…]

*UML-A needs to be demarcated as a subset of UML that captures what a target system is supposed to do and that can be appreciated by nontechnical stakeholders.*« [dC03]

Die Forderung de Champeauxs faßt das derzeitige Dilemma in der Softwaremodellierung gut zusammen und weist zugleich auf einen möglichen Ausweg hin: auf Basis von UML einen Sprachkern definieren, der mit einer ausführbaren Semantik ausgestattet ist und somit die Erfassung der Anforderungen mit der Modellierung der Lösung zu verbinden erlaubt, wobei das Modell gleichzeitig ein Prototyp zur Validierung der Anforderungen ist und über die notwendigen formalen Eigenschaften verfügt, um die Erfüllung bestimmter Anforderungen an das Modell und deren korrekte Umsetzung in ein Programm beweisen zu können. Auch wenn de Champeaux dafür eine deklarative Semantik

(im Sinne seiner eigenen und Hoares Arbeiten) im Sinn hat (persönliche Kommunikation), so würde insbesondere die Modellvalidierung vor allem von einer ausführbaren, d. h. operationalen Semantik profitieren.

Ziel könnte also sein, mit UML-A eine Teilsprache aus UML zu isolieren, die Syntax und intendierte Semantik des Standards weitgehend erhält und die mit Werkzeugen versehen ist, die ihre Anwendung in Softwareentwicklungsprojekten mit formaler Strenge erlauben. Besonderer Wert sollte dabei auf die hohe Verwendbarkeit von Modellen gelegt werden, da sie gemeinsam mit der Übernahme der Diagrammsyntax bereits getätigte Investitionen in Modelle schützt und zukünftige lukrativer macht. Zugleich sollte mit UML-A eine einheitliche Grundlage geschaffen werden, von der ausgehend die Softwaremodellierung in beliebige Richtungen weiterentwickelt werden kann. Kurzgefaßt: Ein UML-A sollte die Rolle für die Softwaremodellierung einnehmen, die die Prädikatenlogik für die Wissensrepräsentation innehat. Abbildung 2 stellt daher UML-A als Punkt an die Spitze der UML-Sprachfamilie (wobei die Ableitung von UML aus UML-A bewußt nicht als Teil der UML-A-Definition dargestellt wurde).

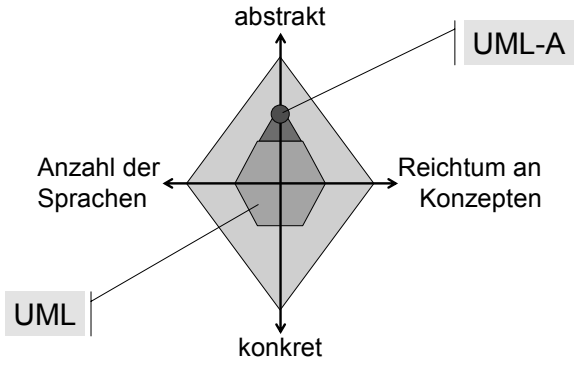


Abbildung 2: UML-A an der Spitze der UML-Sprachfamilie

Um ein solches UML-A etablieren zu können, muß es einer ganzen Reihe von Bedingungen genügen. Diese Bedingungen sollen im folgenden skizziert werden, ohne jedoch hier schon den Anspruch auf Vollständigkeit zu erheben. Insbesondere sollen an dieser Stelle noch keine konkreten Festlegungen (wie etwa auf einen speziellen Semantikformalismus) getroffen werden.

### 3.1 Sparsamkeit

Genau wie am untersten Ende der Abstraktionsskala von Softwarespezifikationen immer ein Programm für eine von-Neumann-Rechnerarchitektur steht, soll auf der höchsten Abstraktionsebene mit UML-A ein kleiner Formalismus zur Verfügung stehen. Eine Beschränkung auf die definierenden Konzepte des objekt- bzw. komponentenorientierten Paradigmas (im wesentlichen Objekte, Verknüpfungen, Narichtenaustausch und Methodendefinitionen sowie Komposition als strukturierendes Sprachelement; s. u.) ist für diesen Formalismus angezeigt. Während die Semantik durch die Referenzimplementie-

rung eines Werkzeugs standardisiert sein soll (s. u.), soll die Sprachdefinition selbst für verschiedene semantische Zielformalismen offenbleiben, so daß für mögliche Erweiterungen verschiedene Zielsprachen (nebst der dazugehörigen Werkzeuge) zum Einsatz kommen können. Spezielle Eigenheiten bestimmter (Ziel-)Programmiersprachen (z. B. die Art des Dispatching von gesendeten Nachrichten) sollen nicht übernommen werden; statt dessen ist, wo immer Alternativen bestehen, die „reine Lehre“ (also z. B. Multidispatch) zur Anwendung zu bringen.

Die Sprachfamilie, für die UML heute steht, soll sich aus diesem UML-A ableiten oder zumindest auf dessen Basis neu definieren lassen (Abbildung 2).

### 3.2 Vollständige Integration

UML ist ein aus verschiedenen Diagrammtypen und der Object Constraint Language (OCL) zusammengesetztes heterogenes Sprachgemisch mit vielen Brüchen und offenen Fragen. Alle in UML-A enthaltenen Sprachelemente sollen dagegen derart miteinander in Beziehung gesetzt sein, daß die in einzelnen Diagrammen (desselben oder verschiedenen Typs) ausgedrückten Sichten eines Softwareprodukts zu einer konsistenten, ausführbaren Gesamtspezifikation dessen zusammengesetzt werden können.[7] Diagrammtypen oder Diagrammelemente, die sich einer Integration entziehen, gehören nicht in die Sprachdefinition von UML-A.

### 3.3 Skalierbarkeit

Auch UML-A-Modelle können erheblichen Umfang erreichen, so daß mächtige Mechanismen des Modellmanagements vorgesehen werden müssen. Während UML bislang (in den Versionen 1.x) zur Beherrschung großer Systeme lediglich paketähnliche Konstrukte vorsah, deren strukturierender Effekt auf ein Modell nicht viel größer ist als der einer Schere auf ein Blatt Papier, finden sich in UML 2.0 mit den sog. Structured Classes komponentenähnliche Konstrukte wieder, die ein rekursives Teilen und Beherrschen von bestimmten Aspekten eines Modells erlauben.

Ziel muß es sein, den Kompositionsbegriff auf alle Modellelemente auszudehnen, um einen einheitlichen Strukturierungsmechanismus zur Verfügung zu haben, mit dem sich, ähnlich wie in SDL, ein System beliebig „zoomen“ läßt. Die ursprünglich dafür vorgesehene Generalisierung, die mit großem Aufwand auf verschiedenste Sprachelemente der UML übertragen wurde, hat sich dafür leider nicht bewährt [SGM03].[8]

---

[7] Dazu müssen in UML-A nicht nur die Elemente eines Diagramms, sondern auch verschiedene Diagramme untereinander systematisch in Verbindung gesetzt werden können, eine Spezifikationsform, die in UML bislang weitgehend fehlt.

[8] Mangelnde Skalierbarkeit ist übrigens auch einer der Gründe, warum viele existierende Formalismen in der Praxis kaum eine Chance haben: Algebraische Spezifikationen beispielsweise eignen sich zwar hervorragend, um eine Schule zu begründen und dazu passende Klausuraufgaben zu formulieren, große Systeme mit über eintausend Datentypen lassen sich so jedoch nur schwer beherrschen. Die gleiche Kritik gilt auch für Ansätze wie ALLOY, dessen Autor das Problem selbst einräumt [Ja02].

### 3.4 Beibehaltung der Syntax

UML-A-Diagramme sollen, soweit möglich, syntaktisch korrekte UML-Diagramme sein und UML-Diagramme gültige UML-A-Diagramme (wobei von UML-A nicht unterstützte Diagrammelemente uninterpretiert bleiben). Auch wenn dies gewissen technischen Einschränkungen unterliegt, so ist es doch für die Verwendbarkeit von UML-A Voraussetzung, zum einen, weil die bereits getätigten Investitionen in bestehende Modelle und in das Erlernen der Notation (Ausbildung) gesichert, zum anderen, weil die problemlose Weiterentwicklung von UML-A-Modellen über beliebige konventionelle UML-Modelle bis hin zu Implementierungen ermöglicht werden muß.

### 3.5 Validierbarkeit von Modellen

Um einem der Hauptrisiken der Softwareentwicklung, der Entwicklung der falschen Anforderungen [Bo91], entgegenzuwirken, muß bereits in einer frühen Phase des Softwareprojekts eine Validierung stattfinden. Sinnvollerweise geschieht dies durch eine Simulation des Modells („Modellanimation"), das dafür eine ausführbare Semantik haben muß. Im Vordergrund steht dabei nicht wie bei der klassischen Berechenbarkeitstheorie die Berechnung von Funktionen, sondern die Spezifikation von Objektinteraktionen und die aus den Interaktionen resultierenden Änderungen der Objektkonfigurationen, soweit sie durch das Modell vorgegeben sind.

Wenn die Modellanimation als Basis der Validierung dienen soll, müssen Möglichkeiten zur Interaktion mit den validierenden Personen bestehen. Dies umfaßt u. a. die Unterstützung der Formulierung von Start- und Endkonfigurationen sowie von Nebenbedingungen, die zur Laufzeit eingehalten werden sollen, aber auch Dinge wie die Anmeldung eines Benutzers in einer bestimmten Rolle (als Akteur), die Auswahl eines auszuführenden Anwendungsfalls sowie der dazugehörigen Parameterobjekte, die Abfrage von Eingaben für die mögliche Verzweigung von Ausführungsflüssen etc. All diese Möglichkeiten müssen durch ein entsprechendes UML-A-Werkzeug umsetzbar sein (s. u.).

### 3.6 Verifizierbarkeit von Modellen

Bei der Definition von UML-A muß darauf geachtet werden, daß die Sprache bekannte berechenbarkeitstheoretische Eigenschaften aufweist bzw. sich in Formalismen abbilden läßt, die solche Eigenschaften besitzen. Ähnlich wie beim heute schon üblichen Model Checking ist z. B. zu klären, in welchen Konstellationen nur endliche Objektstrukturen entstehen können, so daß damit verbundene Modelleigenschaften wie die Terminierung von Abläufen oder die Äquivalenz zweier Modelle effizient bewiesen werden können [SV04]. Auch der Nachweis der Korrektheit von UML-A-Modellen gegenüber einer gegebenen Spezifikation (Verifikation) wird auf diese Weise mit bekannten Mitteln ermöglicht.[9]

[9] Um welche Eigenschaften es sich dabei konkret handeln soll, wird an dieser Stelle bewußt offengelassen, da dieser Beitrag wie gesagt einen Aufruf zur Diskussion darstellt und keine Fakten zu schaffen versuchen will.

### 3.7 Volle Werkzeugunterstützung

UML in seiner jetzigen Fassung ist unter anderem deswegen so fehlerbehaftet, weil in der Definitionsphase kaum geeignete Werkzeuge zum Ausprobieren und Anwenden der Spezifikation zur Verfügung standen. Für UML-A muß der Akt der Sprachdefinition von Anfang an mit Werkzeugen unterstützt werden, die die tentativen Definitionen umsetzen, um Inkonsistenzen und das auch für die Definition von UML-A nicht auszuschließende Entwickeln der falschen Anforderungen schon zu einem frühen Zeitpunkt entdecken zu können. Der Entwicklungsprozeß von ALLOY [Ja02] am MIT darf in dieser Hinsicht als vorbildlich angesehen werden.

### 3.8 Tauglichkeit für die universitäre Lehre

Ein weiteres wichtiges Ziel für die Definition von UML-A leitet sich aus der Tatsache ab, daß UML bereits heute in vielen universitären Curricula eine zentrale Position im Softwareengineering eingeräumt wird: Mancherorts werden sogar ganze Softwaretechnikvorlesungen an UML aufgezogen. Ohne dies kritisieren zu wollen, sollte ein Formalismus, der eine derart wichtige Rolle in der akademischen Ausbildung spielt, gewissen Mindestanforderungen genügen. Ein mit einer eindeutigen Syntax und Semantik versehenes UML-A („A“ in diesem Fall für akademisch) sollte das mit vielen Fragwürdigkeiten behaftete UML zumindest in der Grundlagenausbildung weitgehend ersetzen können.

## 4 Diskussion

Als Diskussionsbeitrag, den diese Arbeit darstellt, sollte sie die Diskussion nicht vorwegzunehmen versuchen. So soll an dieser Stelle nur auf zwei häufige Mißverständnisse hingewiesen werden, um die Diskussion vor Sackgassen zu bewahren.

Es gibt natürlich bereits ein ganze Reihe von Ansätzen mit dem Versuch, aus (Teilen von) UML eine Sprache zu machen, die den hier skizzierten Anforderungen genügt. Viele dieser Ansätze entstammen jedoch einer bestimmten Anwendungsdomäne, nämlich der der eingebetteten Systeme. Diese Systeme besitzen aufgrund ihrer Nähe zur Hardware Eigenschaften, die eine Spezifikation mit den in der Entwicklung von Hardware üblichen Mitteln ermöglicht. Insbesondere kommen dort vor allem Zustands- und Sequenzspezifikationen zum Einsatz, die das Verhalten einzelner Systemkomponenten und deren Interaktion beschreiben. Die dazugehörigen Objektstrukturen sind dabei fast immer statisch: Es werden i. d. R. weder neue Objekte erzeugt noch verschwinden existierende, auch ändert sich der Verknüpfung der Objekte untereinander i. d. R. nicht. Gerade dies macht jedoch das Wesen von Informationssystemen aus, und hierfür wird UML mindestens ebenso benötigt wie für eingebettete Systeme, für die ja auch bereits andere Sprachen (wie z. B. SDL) existieren und im erfolgreichen Einsatz sind.

Bei der ganzen Diskussion um die fehlende Präzision in der Definition (insbesondere die fehlende Semantik) von UML darf ferner nicht vergessen werden, daß UML eine Sprache der Softwareentwickler ist und dies auch bleiben muß. Dabei ist zu bedenken, daß

für UML das gilt, was für alle Sprachen gilt: Damit mit man sich darin richtig ausdrükken kann, muß man ihre Semantik kennen. Dies gilt um so mehr, als man, wie in UML erlaubt, die Sprache (und damit auch ihre Semantik!) für einen bestimmten Verwendungszweck anpassen können soll. Alle bisher vorgebrachten Ansätze, deren Verwendung ein tiefes Verständnis formaler Spezifikationssprachen wie z. B. Z erfordert, werden wohl schon daran gescheitert sein, daß sie innerhalb der OMG nicht von ausreichend vielen verstanden wurden (so z. B. auch die Vorschläge der 2UP-Gruppe zur UML 2.0); sie werden vermutlich auch in Zukunft nicht durchsetzbar sein. Was nützt die schönste Sprache, wenn sie nur wenige verstehen?

Von Softwareentwicklern sehr gut verstanden werden hingegen Programmiersprachen. Was also läge näher, als UML in Anlehnung an eine Programmiersprache zu definieren? Es darf dabei nur nicht der Fehler gemacht werden, Modellierung mit Programmierung zu verwechseln und UML als eine grafische Programmiersprache zu definieren: Die Möglichkeiten zur grafischen Darstellung von Programmabläufen sind gut untersucht, meistens umständlich und tragen längst nicht immer zum besseren Verstehen bei (was übrigens auch das Scheitern grafischer Programmiersprachen[10] erklärt). Der Fokus der Modellierung liegt aber eben nicht auf einer möglichst vollständigen Spezifikation eines Systems, sondern auf der (exemplarischen und verallgemeinerten) Darstellung von Strukturen, der Interaktion und den resultierenden Strukturveränderungen, und zwar in dem Ausmaß, in dem sich die wesentlichen Systemmerkmale daran festmachen lassen. Daß dabei zum Teil dieselben Ausdrucksmittel wie bei der Programmierung zum Einsatz kommen, ist unbestritten, nur sollte allen klar sein, daß eine weitere Programmiersprache – und sei ihre Syntax UML noch so ähnlich – nicht das Ergebnis einer Definition von UML-A sein kann, insbesondere dann nicht, wenn sie (wie z. B. JAVA, das ja gern als semantischer Zielformalismus für UML in Erwägung gezogen wird) voller Eigenheiten ist.

## 5 Schluß

Wer sich heute dazu bekennt, sich mit der Softwaremodellierung — UML insbesondere — zu befassen, der läuft Gefahr, sich wissenschaftlich ins Abseits zu begeben: Zuviel ist bereits geschrieben worden, das den praktischen Anforderungen nicht gerecht wird, zu gering ist die Chance, noch gehört zu werden. Angesichts der dringenden Notwendigkeit einer funktionierenden Modellierungssprache für die Wirtschaft, die ganz offensichtlich selbst nicht in der Lage ist, für eine solche zu sorgen, und daher auf Unterstützung durch die Wissenschaft angewiesen ist, ist dies keine glückliche Situation. In dem vorliegenden Beitrag soll für eine Wiederbelebung der wissenschaftlichen Diskussion um UML geworben werden sowie für einen Zusammenschluß der Kräfte für eine tragfähige Grundlegung.

[10] Scheitern in dem Sinne, daß sie keine breite Verwendung erfahren haben

## Literaturverzeichnis


[Bo91] B Boehm „Software risk management: principles and practices“ *IEEE Software* 8:1 (1991) 32–41.

[dC03] D de Champeaux „Forum: Extending and shrinking UML“ *Communications of the ACM* 46:3 (2003) 11–12.

[Ja02] D Jackson „Alloy: a lightweight object modelling notation“ *ACM Transactions on Software Engineering and Methodology* 11:2 (2002) 256–290.

[MB 02] SJ Mellor, MJ Balcer *Executable UML* (Addison Wesley, 2002).

[SGM03] F Steimann, J Gößner, T Mück „On the key role of composition in object-oriented modelling“ in: *UML 2003: Proceedings of the 6th International Conference* (Springer, 2003) 106–120.

[SV04] F Steimann, H Vollmer „Exploiting the practical limitations of diagrammatic specifications for model validation and execution“ in Vorbereitung für: *Journal on Software & System Modeling*.

[vK89] F von Kutschera *Gottlob Frege: Eine Einführung in sein Werk* (de Gruyter, Berlin 1989).