Einsatz von UML zur Software-Prozeßmodellierung

Mario Beyer, Wolfgang Hesse

FB Mathematik und Informatik
Philipps-Universität Marburg
(pdf-Version, Layout nicht identisch zu eingereichter Fassung)

Zusammenfassung

Bei der hier vorgestellten Arbeit geht es darum, die Eignung der Unified Modeling Language (UML) zur Modellierung von Software-Entwicklungsprozessen zu erproben. Als Demonstrationsbeispiel wurde das Vorgehensmodell EOS (für Evolutionäre, Objektorientierte Software-Entwicklung) gewählt. Zur Modellierung wurden primär UML-Klassendiagramme, Aktivitätsdiagramme und Zustandsdiagramme herangezogen; für spezielle Prozeßdetails wurden ferner Sequenz- und Kollaborationsdiagramme eingesetzt.

Das Ergebnis zeigt, daß UML prinzipiell für diese Art von Aufgabenstellung geeignet ist und eine gut lesbare, vielfältig weiterverwendbare Spezifikation des EOS-Prozesses liefert. Allerdings lassen sich einige Eigenschaften und komplexe Zusammenhänge nicht unmittelbar in UML abbilden, sondern nur mit einer dementsprechend erweiterten Modellierungssprache oder durch ausgiebigen Gebrauch der UML-Erweiterungsmechanismen. Die Vor- und Nachteile beider Lösungen werden diskutiert.

1 Einleitung

Seit dem Aufkommen der objektorientierten (OO-) Analyse- und Entwurfsmethoden Ende der 1980er Jahre hat der Sektor der OO-Modellierung in der Forschung und Entwicklung einen außerordentlichen Aufschwung genommen. Dieser hat einen (vorläufigen) Höhepunkt mit der Definition der *Unified Modeling Language* [UML99] gefunden, mit der eine Vereinheitlichung der vielfältigen vorher existierenden, miteinander konkurrierenden Notationen gelang.

Auf dem Felde der sog. *statischen* Modellierung, die sich i. w. auf die Datenbestände eines betrachteten Anwendungssystems konzentriert, hat die UML mittlerweile ihre Bewährungsproben bestanden und wird heute bereits vielerorts eingesetzt. Dagegen befindet man sich bei der *dynamischen* Modellierung noch weit mehr im Experimentierstadium. Das liegt einerseits an den hier auftretenden grundsätzlich größeren Schwierigkeiten, zum anderen daran, daß es nicht immer leicht zu entscheiden ist, welche Diagrammart aus der Fülle des UML-Angebots (Anwendungsfalldiagramme, Sequenz- und Zustandsdiagramme, Aktivitäts- und Kollaborationsdiagramme) am besten geeignet für einen bestimmten Aspekt ist und wie man komplementäre Darstellungen miteinander konsistent und möglichst redundanzfrei halten kann.

Ein Arbeitsfeld, auf dem noch relativ wenige Erfahrungen mit der Unified Modeling Language vorliegen, ist die Modellierung des Software-Entwicklungsprozesses selbst. Da hier nicht ein konkreter Anwenderprozeß, sondern der unterstützende Software-Entwicklungsprozeß zum Gegenstand der Modellierung wird, spricht man zuweilen (und nicht immer berechtigt) auch von *Meta-Modellierung*.

In der Vergangenheit sind Software-Entwicklungsmodelle vorwiegend informell und dabei in erster Linie durch Text – zum Teil angereichert durch veranschaulichende Diagramme – beschrieben worden. Beispiele dafür findet man z. B. bei B. Boehm (Wasserfall-, Spiral- oder V-förmige Darstellungen), bei T. DeMarco, J. Rumbaugh (Datenflußdiagramme oder dazu ähnliche Bubble Diagrams) und bei vielen firmenspezifischen Vorgehensmodellen (vgl. [Rum91, HN99]).

B. Boehm, W. Humphrey, W. Royce und viele andere Autoren haben grundlegende Vorarbeiten zur Struktur und Zusammensetzung von Software-Entwicklungsprozessen geleistet [Boe81, Hum89, Roy98]). Für präzisere Darstellungen braucht man jedoch eine geeignete (Prozeß-)Modellierungssprache. Bisher vorhandene Sprachen erweisen sich oft als zu technisch orientiert (wie z. B. STL oder ROOM) oder zu detailliert (wie etwa Petri-Netze) und damit für unsere Anwendung und ihre Zielgruppen ungeeignet.

UML wurde als universelle Modellierungssprache konzipiert und ist mittlerweile weit verbreitet - also lag es nahe, diese Sprache einem Eignungstest für die Software-Prozeßmodellierung zu unterziehen. In der hier vorgestellten Arbeit sollte das an einem verallgemeinerten Prozeßmodell oder „Musterprozeß“ geschehen. Dazu wurde das vom zweiten Autor entwickelte und an anderer Stelle veröffentlichte *EOS-Modell* ausgewählt (vgl. [Hes95, Hes96, Hes97]).

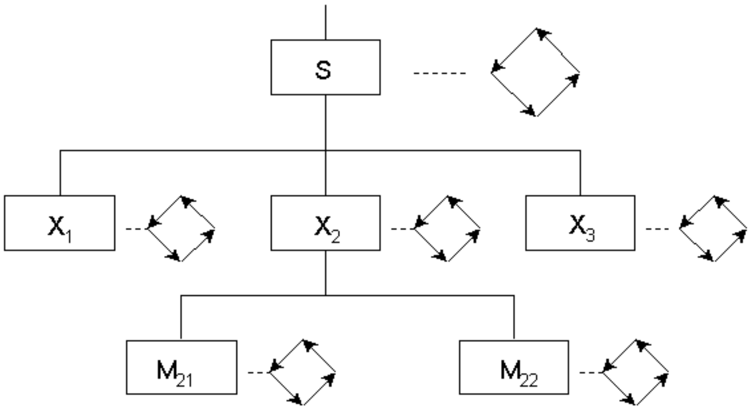

Abbildung 1: Baustein-Hierarchie mit Entwicklungszyklen im EOS-Modell

Für die Wahl dieses Beispielmodells sprachen verschiedene Gründe:

- Es existierte vorher noch keine ausgearbeitete Formalisierung oder Modellierung,
- die Systematik und Orthogonalität des Modells macht es geeignet für ein solches Unterfangen,
- statische und dynamische Elemente sind in ausgewogener Weise vertreten und ließen einen umfassenden und in bezug auf EOS nahezu flächendeckenden Einsatz von UML erwarten,
- das Ergebnis kann nicht nur für eine Evaluation von UML, sondern auch als Grundlage für weitere Werkzeuge und Hilfsmittel zur Prozeßunterstützung (wie Handbücher, Projektassistenten, Navigatoren und Hilfssysteme) verwendet werden.

2 Das Prozeßmodell EOS

Die Vorgehensmodelle für Software-Projekte haben sich im Laufe der Jahre von ursprünglich sehr intuitiven, einfachen Ablaufschemata zu immer differenzierteren und umfassenderen Modellen entwickelt. Zu den einfachen sequentiellen Strukturen der frühen Phasenmodelle kamen zunehmend Quer- und Rückbezüge (z. B. in den V-Modellen) bis hin zu zyklischen Strukturen (z. B. im Spiralmodell) hinzu. Damit ließ sich der Prozeß dynamischer gestalten und eine größere Realitätsnähe erreichen. In den 1990er Jahren erfolgte der Übergang zu objektorientierten Prozeßmodellen (z. B. Boochs Mikro- und Makro-Zyklen [Boo94], Jacobsons Objectory Process [Jac93], RUP [JBR99, Kru99]), die den Softwareprozeß in seiner Gänze zu erfassen versuchen. Diese Modelle stellen den bislang umfassendsten Ansatz für die Modellierung des Software-Entwicklungsprozesses dar.

Von seiten des Managements wurden die neuen Erkenntnisse über die Auswirkungen objektorientierter oder komponentenbasierter Techniken auf den Software-Prozeß zunächst nur ansatzweise oder halbherzig umgesetzt, da damit i. a. komplexere Prozeßstrukturen sowie ein höherer Planungs- und Projektverfolgungsaufwand verbunden sind. Die konkreten Projektsituationen in der Praxis erfordern aber gerade eine solche differenzierte Betrachtung und Flexibilität. Das hier näher betrachtete EOS-Modell wurde konzipiert, um den genannten Forderungen möglichst weitgehend gerecht zu werden.

Das EOS-Modell ist selbst objektorientiert aufgebaut. „Objekte“ im Sinne des Prozeßmodells sind die *Bausteine* der Systementwicklung – angefangen vom Gesamtsystem über Komponenten, Subkomponenten bis hin zu Modulen bzw. Klassen. Diese Hierarchie bildet das Rückgrat der System-Architektur und macht EOS zu einem (im Gegensatz zu konkurrierenden Ansätzen wie etwa RUP) tatsächlich architekturzentrierten Modell (vgl. Abb. 1 und für Einzelheiten [Hes00]).

Jeder EOS-Baustein durchläuft (je nach Bedarf beliebig oft) einen eigenen Entwicklungszyklus durch vier sog. *Baustein-Entwicklungsphasen*. Dieser einfache, aber systematisch durchgehaltene orthogonale Aufbau ermöglicht es, Bausteine auf verschiedenen Hierarchieebenen unabhängig voneinander zu planen, zu entwickeln und zu nutzen. Damit werden Projektabläufe flexibler und orientieren sich mehr an den Projekt-Erfordernissen als an einem starren Phasenschema. Dies gehört neben der Klammerung von Entwicklungs-, Einsatz- und Erprobungsphasen und der Verankerung von Wiederverwendungsmechanismen zu den wichtigsten Voraussetzungen für eine evolutionäre Software-Entwicklung.

In einer kürzlich abgeschlossenen Arbeit [Bey01] bestand die Aufgabe darin, dieses Prozeßmodell mit Hilfe der Unified Modeling Language zu beschreiben. Vorrangige Ziele waren dabei, den Prozeß leichter verständlich darzustellen, Prozeßdetails zu konkretisieren und eine Grundlage für eine maschinelle Prozeßunterstützung zu schaffen.

3 Klassendiagramme

Die Modellierung des EOS-Prozesses beginnt mit der Erstellung eines statischen Modells in Form eines UML-Klassendiagramms.[1] Abbildung 2 zeigt das (in [Bey01] schrittweise erstellte) Klassendiagramm, das alle in EOS enthaltenen Elemente bzgl. der Entwicklung beinhaltet. Im unteren Teil des Diagramms sind eher allgemeine, auch für andere Prozeßmodelle relevante Klassen angesiedelt, im oberen Teil EOS-spezifische Klassen.

Zentrale Elemente des Diagramms sind die Klassen *Baustein* und *Aktivität*. Mit Generalisierungen und Kompositionen lassen sich die Beziehungen zwischen System, Komponente und Modul sowie dem Sammelbegriff Baustein leicht darstellen. Einschränkungen („Constraints“, in geschweiften Klammern) erlauben genauere Bestimmungen der Verwendbarkeit. Auch das zur Bausteinhierarchie (d. h. der Zerlegung von Systemen in Komponenten und Module bzw. Klassen) orthogonale Konzept der „Subsysteme“ ist einfach mittels einer Aggregation darstellbar.[2] Die Beziehungen zwischen den zentralen Aktivitäten und den mit ihnen verbundenen Klassen (Akteur als allgemeine beteiligte Person, Werkzeug, Ergebnis, ...) können durch einfache bidirektionale Assoziationen mit entsprechenden Kardinalitäten dargestellt werden.

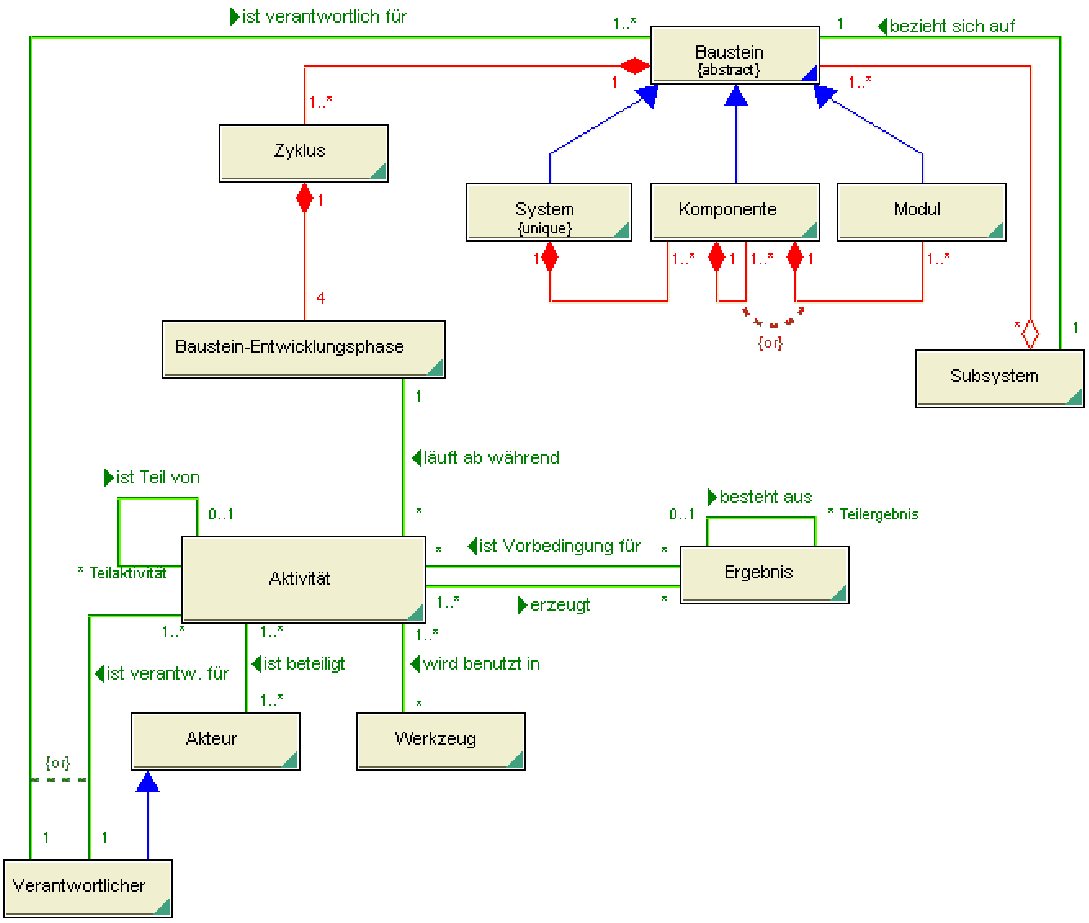

Abbildung 2: Das statische Metamodell als Klassendiagramm

Die Modellierung von Details wie die zu den Klassen gehörenden Attribute und Operationen könnte auch innerhalb der Diagrammdarstellung vorgenommen werden. Aus Gründen der Übersichtlichkeit empfiehlt sich aber eine getrennte Darstellung, eventuell auch in Textform. Hieraus kann auch die Notwendigkeit neuer Klassen

[1] Eine vorherige Anwendungsfallanalyse erwies sich für die vorliegende „generische“ Aufgabenstellung als wenig ergiebig und wurde daher nicht weiter verfolgt, siehe [Bey01].

[2] Ein Subsystem ist nach Definition die von einem Baustein induzierte Menge aller derjenigen Bausteine, die für seinen Ablauf zu Test- und Integrationszwecken benötigt werden.

entstehen; so ist z. B. im EOS-Modell eine Assoziationsklasse für die Verknüpfung von Akteuren mit Aktivitäten angebracht.

Die statische Struktur des EOS-Modells läßt sich also problemlos mit den von UML zur Verfügung gestellten Mitteln darstellen. Dies verwundert nicht, da die Klassendiagramme bzw. „statischen Strukturdiagramme“ die umfangreichste und damit mächtigste Diagrammart der UML sind. Mit den UML-Erweiterungsmechanismen wie z. B. den Einschränkungen („Constraints“) lassen sich detailliertere Angaben in Textform machen, die über das hinausgehen, was sich in UML graphisch darstellen läßt. So läßt sich ein einheitliches Erscheinungsbild wahren, ohne einen Wildwuchs von selbst kreierten Elementen zu erzeugen.

4 Aktivitätsdiagramme

Zur dynamischen Modellierung wird von UML nicht eine einzelne, umfassende Diagrammart zur Verfügung gestellt, sondern gleich mehrere spezielle Diagrammtypen, die eine Modellierung unter verschiedenen Aspekten gestatten. Für die Modellierung eines Prozeßmodells, in dem die Aktivitäten der Beteiligten eine zentrale Rolle spielen, bietet sich die Modellierung durch Aktivitätsdiagramme an. Mit diesen läßt sich der Ablauf jeder einzelnen Bausteinphase mit den zugehörigen Aktivitäten und Ergebnissen übersichtlich modellieren. Die genaue Beschreibung von Aktivitäten mit detaillierten Informationen ihres Gegenstands und der beteiligten Hilfsmittel und Personen können jeweils in Textform angegeben werden, um die Übersichtlichkeit des Diagramms nicht zu stören.

Auch parallel durchzuführende Aktivitäten sind prinzipiell unproblematisch. So ist das laufende Führen und Pflegen einer Dokumentation als einzelne, parallel zum Entwicklungsprozeß verlaufende Aktivität darstellbar. Dennoch treten bei der Modellierung von EOS einige Schwierigkeiten auf. So gibt es z. B. den Mechanismus der Subbausteine[3], die jederzeit und unabhängig abgegrenzt und abgespalten werden können (d. h. für einen Subbaustein wird während einer laufenden Entwicklung ein eigener Entwicklungszyklus gestartet). In UML läßt sich zwar eine dynamische Mehrfachausführung von Aktivitäten darstellen, allerdings insofern eingeschränkt, als zu einem bestimmten Zeitpunkt die Anzahl von gleichzeitig zu startenden, nebenläufigen Aktivitäten feststehen muß. EOS verlangt eine größere Dynamik: Hier muß kenntlich gemacht werden, daß von einer bestimmten Aktivität mehrere Exemplare parallel ablaufen können, die aber auch unabhängig voneinander zu beliebigen Zeitpunkten gestartet werden können.

Verschiedene Ansätze könnten zum Ausdruck der gewünschten Semantik führen, von einer entsprechenden textuellen Bezeichnung der Aktivität bis hin zu einer eigenen Definition eines entsprechenden Diagrammelements. Die meisten Lösungsmöglichkeiten machen den Sachverhalt aber entweder graphisch nicht deutlich oder entfernen sich von dem UML-Grundsatz der Einheitlichkeit der Sprache. Als die am besten geeignete Möglichkeit erweist sich die Definition eines bestimmten Transitions-Stereotyps, durch den die Transition, die zu einem Aktivitätsstrang mit den oben beschriebenen Eigenschaften führt, mit der gewünschten Semantik versehen werden kann. (hier *«multiple-transition-on-demand»* genannt, s. Abb. 3, rechter Strang)

Eine weitere Schwierigkeit der Modellierung ergab sich aus dem Wunsch nach einem Mechanismus, mit dem Verbindungen zwischen verschiedenen Aktivitätsdiagrammen ermöglicht werden. Beispielsweise für den Fall, daß ein Ergebnis einer Entwicklungsphase in eine Aktivität einer anderen Phase einfließt. Eine solche Möglichkeit besteht nur, wenn die beiden beteiligten Aktivitäten im gleichen Diagramm vorkommen; es kann aber oft nützlich sein, ein Diagramm in mehrere kleinere aufzuteilen, so wie es hier anhand der Phasengrenzen vorgenommen wurde.

Zur Skalierung großer Diagramme bietet UML nur die Möglichkeit, mehrere Aktivitäten zu einer „Super-Aktivität“ zusammenzufassen, deren Inhalt sich mittels eines eigenen (Sub-)Aktivitätsdiagramms getrennt darstellen läßt. Die daraus entstehenden gröberen Diagramme führen aber oft nicht zu dem gewünschten Ergebnis. Im vorliegenden Beispiel könnten die Phasen-Aktivitätsdiagramme zu jeweils einem groben Zustand zusammengefaßt und damit in ein Diagramm zusammengeführt werden, das einen gesamten Zyklus umfaßt. Dadurch entfällt aber ebenfalls die Möglichkeit, Verbindungen zwischen einzelnen Aktivitäten der Teildiagramme darzustellen.

Hier wäre die Möglichkeit der Angabe von gekennzeichneten Konnektoren wünschenswert, mit denen ein Kontroll- oder Objektfluß an einer Stelle im Diagramm suspendiert und an anderer Stelle wiederaufgenommen werden könnte. Solche Konnektoren sind zum Beispiel aus den Schaltplänen der Elektrotechnik bekannt. Leider entspricht die Semantik solcher Konnektoren einer Art *goto*-Funktionalität, die aus guten Gründen in modernen ablauforientierten Sprachen nicht mehr vorkommt. Eine dringende Notwendigkeit für ein solches Element in UML liegt auch nicht vor, da durch die Verwendung genügend großer Diagramme die gewünschte Modellierung

[3] Subbausteine werden gebildet, um Teile der Funktionalität eines gegebenen Bausteins herauszulösen und als eigenständigen Baustein behandeln zu können. Sie sind nicht mit den oben erwähnten Subsystemen zu verwechseln.

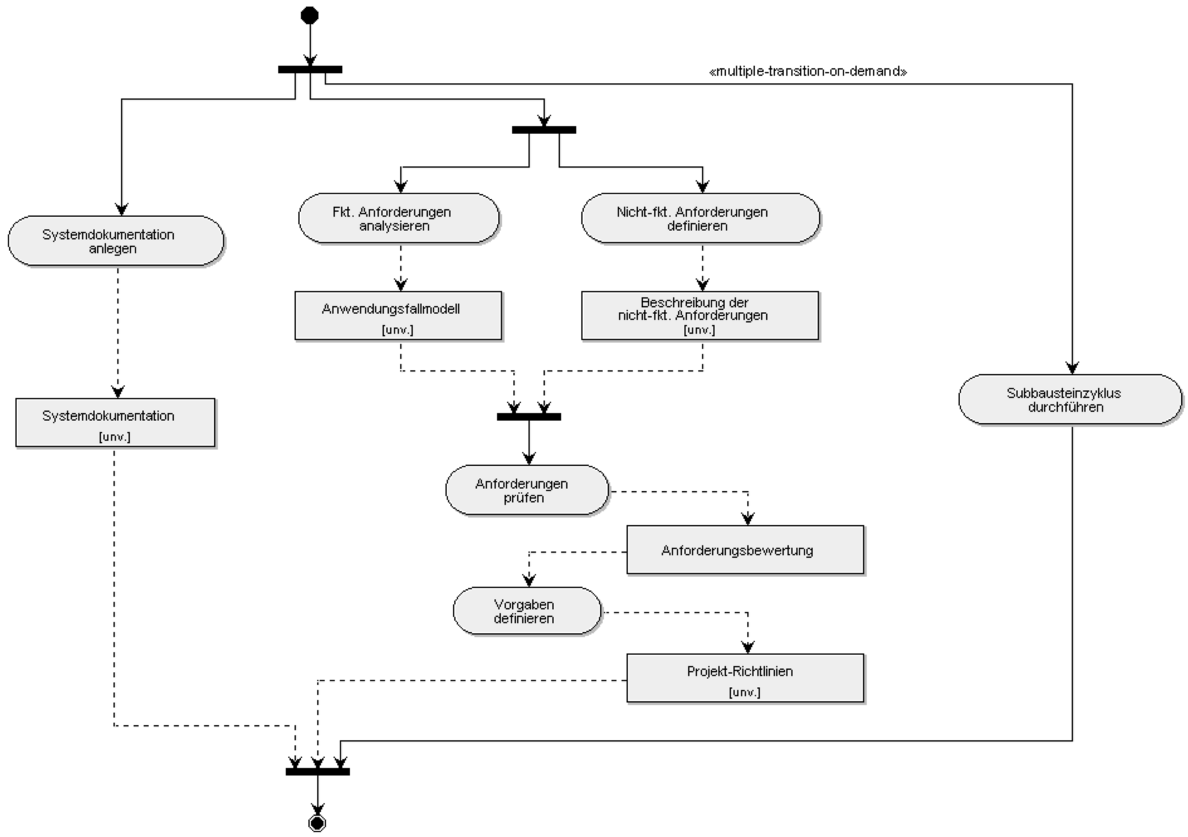

Abbildung 3: Aktivitätsdiagramm der Phase System.Analyse

möglich ist; bei sinnvollem und sparsamem Einsatz könnten solche Konnektoren aber durchaus eine Bereicherung darstellen.

5 Zustandsdiagramme

Für die Modellierung von Software-Prozessen ist natürlich die Betrachtung von Zuständen und Zustandsübergängen besonders interessant. Im Falle von EOS, wo an Bausteine beliebiger Granularität eigene Entwicklungszyklen gebunden sind, bilden diese mit ihren Phasen die gröbsten Zustände der Bausteine. Als weitere Zustände könnte man das Durchführen einzelner Aktivitäten identifizieren. Diese Aspekte wurden aber bereits mit den Aktivitätsdiagrammen modelliert.

Darüber hinaus lassen sich aber weitere Zustände identifizieren, wie sie Abbildung 4 zeigt. Zum Beispiel ist ein Baustein der Systemebene ständig in dem Zustand, in dem die Abspaltung und das Anstoßen eines Komponenten-Entwicklungszyklus möglich ist. Weiter läßt sich eine grobe Einteilung in Zustände der Analyse, der eigentlichen Entwicklung und der Beobachtung eines Bausteins vornehmen, die nicht mehr der Unterteilung in die vier Entwicklungsphasen entspricht. So befindet sich ein Baustein beispielsweise während eines internen Funktionstests noch klar in der Implementierungsphase, während des Echttests in der Zielumgebung aber bereits in der Phase Operationeller Einsatz.

Ein wichtiger Zustand eines Bausteins besteht darin, daß sich seine Entwicklung zu diesem Zeitpunkt im wesentlichen auf die o. g. Abspaltung von Subbausteinen und das Durchlaufen der ihnen zugeordneten Zyklen konzentriert. Auf der Systemebene erstreckt sich dieser Zustand z. B. vom Beginn des Aufteilens in Komponenten bis zur Beendigung der dementsprechenden Integrationsschritte. Im Diagramm wird dieser Zustand mit „Komponentenentwicklung angebracht“ bezeichnet. Auch er erstreckt sich über zwei Entwicklungsphasen. Hier zeigt sich der Nutzen einer Zustandsmodellierung unabhängig von der Aktivitätenstruktur. Weiter wird in dem Diagramm aber auch unterstrichen, daß generell eine Abspaltung von Komponentenzyklen möglich ist, indem sämtliche Zustände einem entsprechend bezeichneten globalen Zustand untergeordnet sind.

Die angeführten Beispiele zeigen eine typische Situation für komplexe Prozeßmodelle: Es gibt verschiedenartig ineinander geschachtelte Zustände, die sich für einen EOS-Baustein definieren lassen. Diese können mit

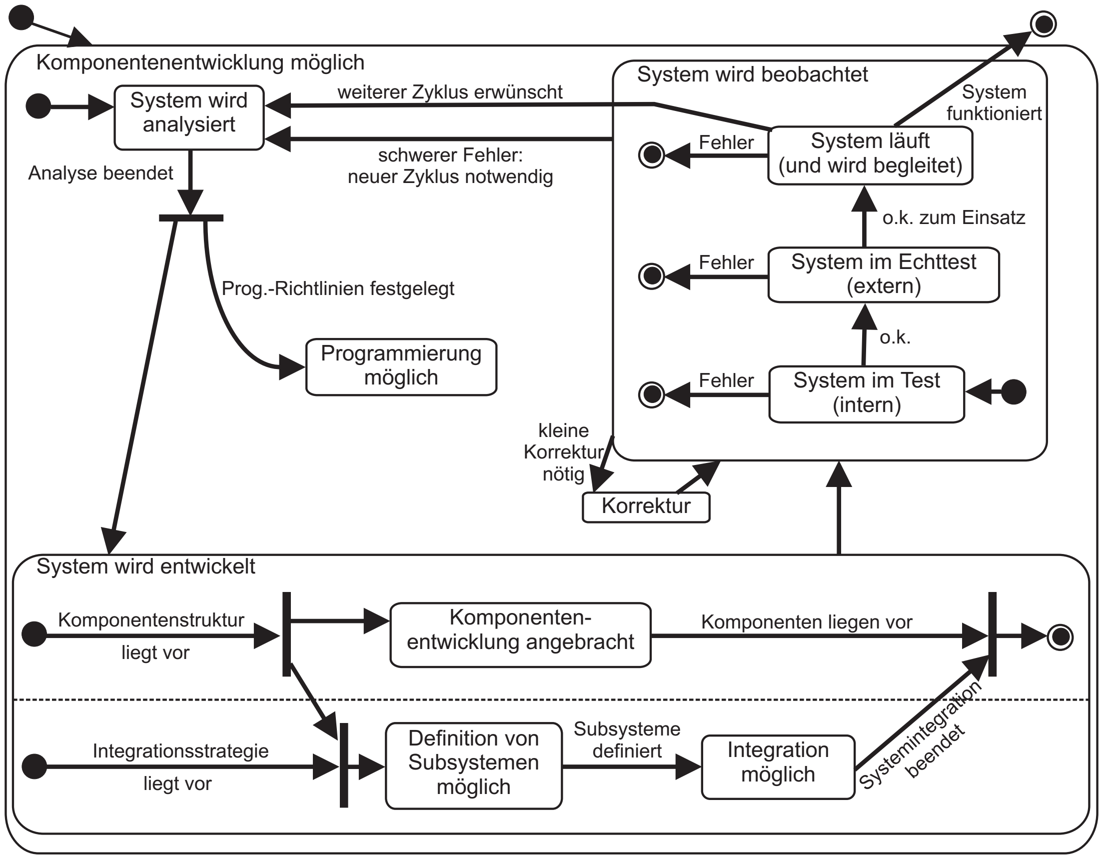

Abbildung 4: Das Zustandsdiagramm der Systemebene

Zustandsdiagrammen und den darin enthaltenen Mechanismen der inneren Zustände einfach und übersichtlich umgesetzt werden. Dazu trägt auch die UML-Erweiterung der State Charts um Konstruktionselemente für parallele, nebenläufige Zustände und Zustandsübergänge bei. In [Bey01] wurde für jede Bausteinart ein eigenes Zustandsdiagramm modelliert. Dadurch wird hauptsächlich der Subbaustein-Mechanismus von EOS näher und aus einer anderen Sicht beschrieben.

Nicht ganz unproblematisch gestalten sich allerdings auch hier Auswirkungen eines Zustands auf Elemente einer anderen Bausteinebene und damit auf andere, fremde Zustandsdiagramme. So kann beispielsweise auf Modulebene erst mit der Programmierung begonnen werden, wenn global auf der Systemebene die Richtlinien hierfür festgelegt wurden, also der Zustand „Programmierung möglich“ erreicht wurde. Eine direkte Verknüpfung zwischen den beiden Diagrammen ist nicht möglich. Zwar kann man diese Abhängigkeit als Bedingung textuell angeben, der direkte Bezug zu dem entsprechenden Zustand ist aber in den Diagrammen nicht explizit graphisch darstellbar.

6 Sonstige Diagramme

Um weitere Aspekte des Software-Prozesses modellieren zu können, wurden Sequenzdiagramme und Kollaborationsdiagramme auf ihre Anwendbarkeit hin untersucht. Die zeitlich gegliederten Sequenzdiagramme eignen sich für die Modellierung von linearen Botschaftsabfolgen zwischen bestimmten Objekten. Beispielsweise kann hierdurch der Vorgang der Abgrenzung und Entwicklung eines Subbausteins mit Blick auf die beteiligten Objekte (Verantwortliche, Akteure und Bausteine) modelliert werden. Auch hier fällt das Fehlen von Ausdrucksmöglichkeiten für bedarfsweise mehrfache Ausführung von parallelen Aktionen auf.

Kollaborationsdiagramme sind semantisch verwandt, konzentrieren sich aber auf die Zusammenarbeit von Objekten, die graphisch gegenüber den Sequenzdiagrammen in den Vordergrund gerückt werden, indem die Botschaften als Beschriftung der Objektverbindungen dargestellt werden. Entsprechend lassen sich auch mit diesem Ausdrucksmittel bestimmte Abläufe aus einem anderen Blickwinkel modellieren. Für die Modellierung eines Ablaufs selbst eignen sich diese Diagramme aber nicht, da die Botschaftsfolge durch Sequenznummern dargestellt wird, was sich als eine ziemlich unpraktische Lösung erweist. Als Analogon aus dem Bereich der Programmiersprachen drängen sich hier Zeilennummern mit all ihren Nachteilen auf. Die Übersichtlichkeit der Numerierung wird zusätzlich dadurch erschwert, daß Unterteilungen ($1 \rightarrow 2.1 \rightarrow 2.2 \rightarrow 3\ldots$) gestattet sind, wodurch in großen Kollaborationsdiagrammen leicht Botschaften übersehen werden können. Ansonsten bieten Kollaborationsdiagramme auch nicht sehr viele Modellierungselemente, so daß sich ihr Einsatz oft eher auf eine Möglichkeit einer Alternativdarstellung reduziert. Die verbleibenden Diagrammtypen – Komponenten- und Einsatzdiagramme – sind vornehmlich für die Strukturierung einer Implementierung gedacht und waren damit für die hier angestrebte Form der Prozeßmodellierung nicht relevant.

Über die bislang behandelten entwicklungsspezifischen Aspekte von EOS hinaus wurde in [Bey01] auch der Gesamtablauf eines „EOS-Prozesses“ modelliert, d. h. das Zusammenspiel der verschiedenen parallel ablaufenden Subprozesse Projektmanagement, Entwicklung, Qualitätssicherung, Konfigurationsmanagement und Nutzung/Bewertung (vgl. [Hes97]). Dies ist mit einer erweiterten Fassung des Klassendiagramms darstellbar. Ebenso konnte (durch den Einsatz von Aktivitätsdiagrammen) ein weiterer wichtiger Aspekt für den Gesamtablauf umgesetzt werden, das Konzept der „Revisionspunkte“, die die bekannten Meilensteine ablösen.

7 Ergebnis

Insgesamt hat die durchgeführte exemplarische Umsetzung des EOS-Modells gezeigt, daß sich UML-Diagramme gut zur Darstellung von Software-Prozeßmodellen eignen. Die gebotenen Erweiterungsmechanismen sind vielseitig und werden auch benötigt. Zum Teil wären aber erweiterte Möglichkeiten wünschenswert. Anforderungen dafür ergeben sich häufig dort, wo nicht-hierarchische Strukturen auftreten (Zusammenhänge zwischen verschiedenen Entwicklungsebenen) bzw. wo eine komplexe Situation zu modellieren ist, die z. B. dynamische Entscheidungen zu beliebigen Zeitpunkten zuläßt. Zwar ist eine solche flexible Dynamik schwierig übersichtlich darzustellen, die Integration eines entsprechenden Konzepts in die UML sollte aber in Zeiten, in denen Multi-Threading in Programmen eine große Bedeutung gewinnt, erwogen werden.

Darstellerische Probleme ergeben sich für den UML-Anwender immer dann, wenn die angebotenen Sprachkonstrukte nicht für die Darstellung komplexer Sachverhalte ausreichen. Dann muß er sich entscheiden, ob er die oft wenig intuitiven Erweiterungsmechanismen nutzen will oder den Sprachumfang von UML durchbrechen soll.

Aus UML-Sicht stehen hier zum einen die Einfachheit und Verständlichkeit der Sprache, zum zweiten die Universalität und breitbandige Einsetzbarkeit sowie eine hohe Ausdrucksfähigkeit auch für komplexe oder spezielle Sachverhalte miteinander in Konflikt. Die UML-Erweiterungsmechanismen (Einschränkungen, Stereotypen etc.) erlauben zwar geeignete Erweiterungen der Semantik ohne diese Konzepte zu verletzen, engen allerdings den Spielraum für graphische Erweiterungen ein.

Mit den verschiedenen Diagrammarten lassen sich viele unterschiedliche Aspekte modellieren. Allerdings sind nicht alle Arten gleichermaßen für die Prozeßmodellierung geeignet. Im Vordergrund stehen Klassen-, Aktivitäts- und Zustandsdiagramme, die eine detaillierte und weitgehend übersichtliche Beschreibung der statischen und dynamischen Eigenschaften des Prozeßmodells ermöglichen. Andere Diagrammtypen können zum Teil für Alternativdarstellungen eingesetzt werden oder sind weniger brauchbar, weil sie einen für die Prozeßmodellierung weniger maßgeblichen Ansatz verfolgen. Die Notwendigkeit von Erweiterungen bei der Prozeßmodellierung hängt auch von der Diagrammart ab. So besteht zum Beispiel bei den Klassendiagrammen kaum Bedarf an einer Erweiterung, während dies bei anderen, stärker eingeschränkten Diagrammarten eher notwendig werden kann.

Die Vielfalt der Diagrammtypen in UML bietet aber einen weiteren Vorteil: Dadurch, daß Modellierungen aus verschiedenen Perspektiven ermöglicht werden, besteht eine starke Kopplung der Diagramme untereinander. Hieraus resultiert eine große Sicherheit der Modellierung. Die Koppelung verschiedenartiger dynamischer Diagramme hilft dabei, frühzeitig Fehler zu erkennen. Darüber hinaus besteht eine Rückkopplung zum statischen Klassenmodell, das z. B. durch das Zusammenspiel von Objekten in Sequenz- und Kollaborationsdiagrammen einer fortlaufenden Revision unterzogen werden kann.

In der Gesamtsicht fällt die Bewertung also positiv aus. UML wird dem Anspruch der vielseitigen Anwendbarkeit gerecht. Nötige Erweiterungen können vorgenommen werden, notfalls immer auf Textbasis, wobei Abstriche in der Übersichtlichkeit in Kauf genommen werden müssen. Ein großer Vorteil der Prozeßmodellierung mit UML ist die einheitliche Darstellung, die ein Vergleichen verschiedener Prozeßmodelle erleichtert.

Mit der Modellierung von EOS wurde der Einsatz von UML an einem modernen Prozeßmodell mit vielen nichtlinearen Eigenschaften ausprobiert. Im Anschluß daran wäre es interessant zu überprüfen, ob und inwieweit sich die Ergebnisse auch auf andere Prozeßmodelle übertragen lassen, die möglicherweise anderen Ansätzen folgen. Weiterhin könnte die Eignung von UML für die Detailmodellierung tiefergehend untersucht werden, indem man versucht, ein Prozeßmodell in voller Tiefe ausschließlich mit UML-Mitteln zu beschreiben. Bei einem solchen Unterfangen wären auch die weiteren Formalisierungsmöglichkeiten zu prüfen, die UML zusätzlich vorsieht (z. B. formale Angabe von Einschränkungen, Bedingungen, Querbezügen etc., etwa durch den Einsatz von OCL). UML wird laufend weiterentwickelt, der Sprung auf Version 2.0 steht bevor. Sobald diese Version einsatzbereit ist, wäre im einzelnen zu untersuchen, welche neuen Erweiterungen sie bietet und inwiefern diese die hier aufgezeigten Probleme zu lösen erlauben.

Literatur

[Bey01] M. Beyer: *Verwendung von UML zur Modellierung von Software-Entwicklungsprozessen*. Diplomarbeit, Philipps-Universität Marburg, 2001

[Boe81] B. Boehm: *Software Engineering Economics*. Prentice Hall, 1981

[Boo94] G. Booch: *Object-Oriented Analysis and Design with Applications*. 2nd Edition, Benjamin/Cummings Publ. Comp., 1994

[Hes95] W. Hesse: *Evolutionäre Softwareentwicklung und Projektmanagement*. In: Huber-Wäschle, Schauer, Widmeyer (ed.): GISI '95 – Herausforderungen eines globalen Informationsverbundes für die Informatik. Informatik aktuell, Springer-Verlag, 1995

[Hes96] W. Hesse: *Theory and Practice of the Software Process - A Field Study and its Implications for Project Management*. In: Montangero (ed.): Software Process Technology, 5th European Workshop, EWSPT 96. Springer LNCS 1149, pp. 241–256, 1996

[Hes97] W. Hesse: *Improving the Software Process guided by the EOS Model*. In: Proc. SPI '97 European Conference on Software Process Improvement. Barcelona, 1997

[Hes00] W. Hesse: *Software-Projektmanagement braucht klare Strukturen – Kritische Anmerkungen zum „Rational Unified Process"*. In: Ebert, Frank (ed.): Modelle und Modellierungssprachen in Informatik und Wirtschaftsinformatik. Proc. „Modellierung 2000", pp. 143–150, Fölbach-Verlag, 2000

[HN99] W. Hesse, J. Noack: *A Multi-Variant Approach to Software Process Modelling*. In: Jarke, Oberweis (ed.): Advanced Information Systems Engineering. 11th Int. Conf. CAiSE '99, LNCS 1626, pp. 210–224, 1999

[Hum89] W. Humphrey: *Managing the software process*. Addison-Wesley, 1989

[Jac93] I. Jacobson: *Object-Oriented Software Engineering – A Use Case Driven Approach*. Addison-Wesley, 1993

[JBR99] I. Jacobson, G. Booch, J. Rumbaugh: *The Unified Software Development Process*. Addison-Wesley, 1999

[Kru99] P. B. Kruchten: *The Rational Unified Process (An Introduction)*. Addison-Wesley, 1999

[Roy98] W. Royce: *Software Project Management – A Unified Framework*. Addison-Wesley, 1998

[Rum91] J. Rumbaugh et al.: *Object-oriented Modelling and Design*. Prentice Hall, 1991

[UML99] The Object Management Group: *OMG Unified Modeling Language Specification*. Version 1.3, June 1999.
`http://www.rational.com/uml/resources/documentation`