Integration von Modellen in einen codebasierten Softwareentwicklungsprozess

Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler
Technische Universität Braunschweig
Institut für Software Systems Engineering
38106 Braunschweig, Germany
http://www.sse.cs.tu-bs.de

Abstract: Dieses Artikel beschreibt Konstellationen, unter denen Modelle und Quellcode innerhalb eines Projekts integrativ genutzt werden können. Ähnlich wie die in den Frühzeiten des Compilerbaus notwendige Integration von Hochsprachen und Assembler stellt ein solches Vorgehen einen wichtigen Zwischenschritt zu einer im Kern modellbasierten Softwareentwicklung dar. Anhand einer Fallstudie zu Statecharts wird beschrieben, wie das an der TU Braunschweig in Entwicklung befindliche Modellierungswerkzeug MontiCore genutzt werden kann, um Modelle effizient in einen agilen Softwareentwicklungsprozess zu integrieren.

1 Einführung

In der Softwaretechnik nehmen Komplexität und Umfang von Softwareprojekten seit vielen Jahren stetig zu. Deshalb wird in der Softwareentwicklung das Mittel der sprachlichen Abstraktion als wesentliches Konzept zur Beherrschung der steigenden Komplexität eingesetzt. Dazu gehört zum Beispiel der Wechsel von der maschinennahen und damit hardwareabhängigen Programmierung hin zu höheren Programmiersprachen. Zusätzliche Bibliotheken mit vorgefertigten Funktionen und Gruppierung von Funktionseinheiten in Schichtenarchitekturen wie Betriebssystemen brachten durch die verbesserte Wiederverwendung und Kapselung eine weitere Produktivitätssteigerung. Ein nächster naheliegender Schritt in dieser Entwicklung ist die Verwendung von Modellierungssprachen, um von Details der Implementierung zu abstrahieren. Dabei hat sich die UML [OMG05b] in den letzten Jahren als Standard etabliert.

Um den Nutzen der Modelle zu erhöhen, wurde von der OMG die Model Driven Architecture (MDA) [OMG05a] eingeführt. In der MDA bilden verschiedene Abstraktionsstufen in Form von Computation Independent Model (CIM), Platform Independent Model (PIM), Platform Specific Model (PSM) und Quellcode sowie Transformationen, die technische Informationen zu den Modellen hinzufügen, die Grundlage für die effiziente Erzeugung von Software [KWB03]. Dieses Vorgehen beinhaltet jedoch einige Nachteile:

1. Bei weniger komplexen Projekten kann durch den hohen Abstraktionsgrad die Definition der verschiedenen Modelle und die Implementierung der zugehörigen Trans-

[GKR+06] H. Grönniger, H. Krahn, B. Rumpe, M. Schindler.
Integration von Modellen in einen codebasierten Softwareentwicklungsprozess..
In: Proceedings der Modellierung 2006. 22.-24. März 2006, Innsbruck.
GI-Edition - Lecture Notes in Informatics, LNI P-82,
ISBN 3-88579-176-5, 2006.
www.se-rwth.de/publications

formationen den Aufwand der direkten Programmierung deutlich übersteigen, ohne einen wesentlichen Vorteil bei Qualität und Wiederverwendung zu bringen.

2. Manuelle Änderungen durch einen Entwickler an den aus Transformationen entstehenden Modellen oder Quellcode haben zur Folge, dass die Modelltransformationen entweder nur einmalig möglich sind („one shot“) oder ein technisch schwieriges und heute im Wesentlichen nur für struktur-orientierte Diagramme beherrschtes Round-Trip-Engineering verwendet werden muss.

3. Wird die Konsistenz zwischen CIM/PIM/PSM-Modellen nicht automatisch gesichert, so besteht die Notwendigkeit zur Verfolgung von Abhängigkeiten zwischen Modellen mit relativ hohem Aufwand, um Änderungen beherrschen zu können.

Aufgrund dieser Nachteile wird in diesem Artikel ein Ansatz vorgestellt, Modelle direkt als Programm einzusetzen, um den daraus generierten eigentlichen Quellcode für den Entwickler unsichtbar zu halten. Für den Entwickler ergänzen sich Modell und der für ihn sichtbare Teil des Codes, sind aber ohne Redundanz. Dieser Ansatz ist vom Prinzip her ähnlich wie bei frühen Compilern, die zunächst Assembler-Quellcode erzeugt haben, den die Entwickler aber nicht mehr betrachten sollten.

Grundsätzlich wird in dem hier vorgestellten Konzept die modellbasierte Softwareentwicklung (MBSE) [Rum04a, Rum04b] nicht ausschließlich auf UML-Modelle beschränkt, sondern je nach Domänen- oder Projektanforderung spezifische Modelle (DSLs) eingesetzt [CE00, Tol04]. Ziel ist die Bereitstellung einer Infrastruktur, die es ermöglicht, alle sinnvollen Informationen bereits auf Modellebene zu definieren, so dass eine Entwicklung im Wesentlichen durch Nutzung von Modellen möglich ist. Solche Modelle definieren dann in Verbindung mit den von ihnen genutzten Codegeneratoren und vorgefertigten Implementierungsbausteinen vollständig Struktur und Verhalten der entstehenden Software und erlauben so die Generierung eines lauffähigen Systems. Modelle werden somit die „first class artifacts“ der Softwareentwicklung [MB02, Rum02] und dienen nicht mehr nur der Diskussion und Dokumentation oder der Generierung von manuell zu modifizierenden Code-Rümpfen.

Spätestens seit dem Scheitern der visuellen Programmiersprachen [Sch98] ist klar, dass nicht alles durch graphische Modelle beschrieben werden sollte. Für ein effektives Arbeiten mit Modellen nutzt der hier vorgestellte Ansatz daher eine Einbettung von textuellen Blöcken beispielsweise für Ausdrücke und Anweisungen einer Programmiersprache in Modelle, wie das heute bereits in vielen Werkzeugen der Fall ist. Dadurch wird außerdem eine zusätzliche Definition einer eigenen Aktionssprache wie etwa [OMG01] für die jeweiligen Modelle überflüssig.

Jedoch ist es darüber hinaus ebenfalls notwendig, eigenständige Codeeinheiten (etwa Klassen) in Modellen zu nutzen und umgekehrt die in Modellen beschriebenen Softwareteile in Code einzubinden. Dazu ist die Kenntnis der Schnittstellen des generierten Codes notwendig. Für eine Integration von Modellen in eine codebasierte Softwareentwicklung ist daher die Bekanntmachung von Schnittstellen jenseits des bisher meist üblichen Nachschauens im generierten Quellcode notwendig.

Da ein direkter und vollständiger Übergang von rein quellcodebasierter zu rein modellbasierter Softwareentwicklung unserer Ansicht nach technisch oft noch nicht machbar und mit Blick auf bereits existierenden Code zu anspruchsvoll ist, ist hier ein Zwischenschritt erforderlich, der beide Ansätze kombiniert. In der in diesem Artikel beschriebene Machbarkeitsstudie erfolgt daher als erster Schritt eine Integration von UML-Modellen und Java-Code, wobei zwei verschiedene Integrationsmöglichkeiten kombiniert werden. Zum einen werden Quellcodeelemente wie Java-Statements und -Expressions als Aktionssprache in Modelle eingebettet [Rum04a, Rum04b]. Dieses ist vergleichbar mit der Situation bei der früher möglichen Einbettung von Assembler-Code in Pascal-Programme. Zum anderen werden Modelle in quellcodebasierte Projekte integriert, wobei Modelle und Quellcode miteinander über definierte Schnittstellen interagieren.

Dementsprechend manifestiert sich in diesem Ansatz ein Modell als eine oder mehrere eigenständige Quellcode-Einheiten mit fest definierten und vom Modell (in Kombination mit dem Generator) festgelegten Schnittstellen. Die hier eingesetzten Arten von Modellen können daher als eigenständige Komponenten im Softwareprodukt verstanden werden [Bez05].

Die Integration kann nur dann erfolgreich sein, wenn

- die Schnittstellen und natürlich auch die Semantik der Modelle für einen Entwickler klar nachvollziehbar sind, ohne die Interna des generierten Codes kennen zu müssen,
- die Entwicklungsinfrastruktur einen Komfort bietet, der mit heutigen integrierten Entwicklungsumgebungen für Quellcode vergleichbar ist, und
- der Nutzen etwa durch Produktivitätssteigerung erkennbar ist.

Im Folgenden wird beschrieben, wie diese Ziele für ausführbare Modelle umgesetzt werden können. Der Rest dieses Artikels ist wie folgt gegliedert. In Abschnitt 2 werden zunächst die Voraussetzungen für eine erfolgreiche Integration von Quellcode und Modellen konzipiert. In Abschnitt 3 wird das in Entwicklung befindliche Werkzeug MontiCore beschrieben, das es ermöglicht, auf effektive Weise eine umfangreiche Infrastruktur für Modellierungssprachen zu erzeugen und daher die Grundlage für die Studie bildet. In Abschnitt 4 wird beispielhaft die Integration von Statecharts in Eclipse unter Nutzung von MontiCore gezeigt. Abschnitt 5 gibt einen zusammenfassenden Ausblick.

2 Integration ausführbarer offener Modelle

Unter einem *ausführbaren Modell* verstehen wir ein Modell, das eine wohldefinierte Ausführungssemantik hat und für das eine Codegenerierung existiert, die diese widerspiegelt. Potentielle Unterspezifikationen des Modells sollen vermieden oder durch die Codegenerierung in determinierter Form aufgelöst werden. Nicht-deterministische Spezifikation des Modells kann im Allgemeinen wie Unterspezifikation verstanden und behandelt werden, die der Generierung Freiheitsgrade lässt [Rum96, BS01].

Ein System wird als *offen* bezeichnet, wenn es „explizite Schnittstellen an die Systemumgebung hat“ [Rum04a, S. 264]. Beispiele für ausführbare und je nach Form der Code-

generierung auch offene Modelle sind Statecharts, Sequenzdiagramme, Automaten und Aktivitätsdiagramme.

Die *Schnittstelle* eines Quell- bzw. Objektcodemoduls beschreibt in der objektorientierten Form die nach außen sichtbaren Möglichkeiten Objekte zu erzeugen oder zu löschen, Methoden aufzurufen, aber auch Attribute zu nutzen und zu verändern, Threads zu manipulieren oder im verteilen System, Nachrichten und Signale zu senden.

Zum Beispiel bedeutet dies für einen Java-Entwickler, dass er eine Klasse `A` in eine Textdatei `A.java` schreiben muss, und dann aus einem anderen Programmteil heraus diese mit `new A()` instanzieren kann. Diese Verwendung ist in Programmiersprachen selbstverständlich, da nur so eine kompositionale Softwareentwicklung möglich wird, fehlt aber für Modelle fast komplett und führt daher notwendigerweise zu der heute in Werkzeugen üblichen zentralisierten Modellvorstellung.

Eine agile Vorgehensweise bei gleichzeitigem Einsatz von Modellen und Quellcode ist nur möglich, wenn die Entwickler die Schnittstellen und damit die Interaktion zwischen beiden intuitiv verstehen und eine kompositionale Softwareentwicklung auf Modellbasis ermöglicht wird. Daraus ergeben sich einige Anforderungen, die bei der Generierung von Quellcode aus einem Modell beachtet werden sollten:

- Ein Modell wird auf eine Klasse abgebildet. Komplexere Codegenerierungen können das Fassade-Muster [GHJV95] nutzen, um einen einzigen Interaktionspunkt zur Verfügung zu stellen.
- Die Eingabeschnittstellen des Modells werden als öffentliche Methoden dieser Klasse zur Verfügung gestellt. Alternativ kann es eine (oder wenige) vordefinierte Methoden geben, die Eingaben als Parameter verarbeiten.
- Die Ausgaben des Modells werden bei einer synchronen Modellsemantik als Rückgabewerte der Methoden realisiert. Bei einer asynchronen Modellsemantik kann beispielsweise ein Callback/Listener-Konzept verwendet werden.
- Sprachkonstrukte der Zielsprache, wie Anweisungen (statements) und Ausdrücke (expressions), werden in das Modell eingebettet. Somit ist eine eigene „Aktionssprache“ des Modells überflüssig.
- Die öffentlich zur Verfügung gestellten Methoden und Konstruktoren sind als Schnittstellen einfach und intuitiv aus dem Modell ableitbar, ohne den generierten Quellcode zu bemühen.

Obige Bedingungen sind hinreichend für eine kompositionale und modulare Nutzung von Modellen. Erstellt ein Entwickler beispielsweise ein Statechart `A`, ist dieses mit `new A()` instanzierbar. Verarbeitet das Statechart ein Event `b`, existiert damit eine öffentliche Methode `b()`, die aufgerufen werden kann, um das Event auszulösen. Für das Verständnis dieses Konzepts sind keine Kenntnisse der internen Realisierung der Codegenerierung notwendig. Aus Sicht des Entwicklers bedeuten demnach die Schnittstellen eine Zugriffsmöglichkeit auf das Modell und nur indirekt auf den generierten Code.

Die gestellten Bedingungen mögen zunächst relativ trivial erscheinen, werden aber in der Praxis oft vergessen. Insbesondere werden oft generierte Hilfsmethoden des Codes benutzt, deren Existenz nicht publiziert war und die in der nächsten Iteration des Produkts nicht mehr existieren.

Für eine nahtlose Integration in eine vorhandene Quellcodeentwicklung sind des Weiteren die folgenden Punkte von großem Nutzen:

Entwicklungsumgebung: Die Modelle und ihre Codegenerierung sind in eine offene Entwicklungsumgebung zu integrieren, so dass kein Wechsel der Anwendung nötig ist, um Modell und Quellcode zu bearbeiten.

Komfortables Editieren: Moderne IDEs bieten visuelle Hilfen an, die eine Entwicklung deutlich erleichtern, wie Syntaxhervorhebung und eine Übersicht zur Navigation durch hierarchische Teile des Modells. Des Weiteren sind Auto-Vervollständigung, Warnungen, Fehlermeldungen und Vorschläge für die Fehlerbehebung hilfreich.

Refactoring: Analog zu den Refactorings [Fow99] für den Quellcode, müssen Refactorings auf Modellebene zur Verfügung stehen. Insbesondere Refactorings, die schnittstellenübergreifend zwischen Code und Modellen zum Einsatz kommen, wie etwa die konzeptionell an sich einfache Umbenennung müssen sich sowohl auf das Modell als auch auf den das Modell nutzenden Quellcode auswirken.

Tests: Die Nutzung von Modellen ist ein wesentlicher Teil des Entwicklungsprozesses und muss daher geeigneten Qualitätssicherungsmaßnahmen unterliegen. Daher ist beispielsweise die Einbindung von Modellen in ein Unit-Testframework wichtig. Dieses Test-Framework kann selbst Modelle als Testfallbeschreibungen nutzen und so eine analoge Integration zwischen handgeschriebenen Tests und modellbasierten Tests erreichen.

Fehlersuche: Der Entwickler sollte im selben Debugger Modelle und Quellcode behandeln können. Dabei sollte die Fehlersuche insbesondere direkt im Modell möglich sein und nicht nur im generierten Quellcode.

Versionskontrolle: Für eine effektive Arbeit sind eine kohärente und integrierte Versionskontrolle und analoge Mechanismen zum Qualitäts- und Projektmanagement notwendig.

3 Das Modellierungswerkzeug MontiCore

Als ein notwendiger Zwischenschritt in Richtung einer vollständig modellbasierten Softwareentwicklung wurde in Abschnitt 2 die parallele Entwicklung von ausführbaren Modellen und damit zusammenarbeitendem Quellcode identifiziert. Für eine technische aber auch konzeptuell einfach nutzbare integrierte Entwicklungsumgebung wird am Institut für Software Systems Engineering der TU Braunschweig das Modellierungswerkzeug MontiCore als Framework entwickelt und seine Oberfläche in Eclipse integriert.

In der vorliegenden Fassung von MontiCore wird dabei bewusst auf eine eigenständige graphische Oberfläche verzichtet, um so schneller Modellierungssprachen, darauf aufbauende Analyse-, Transformations- und Generierungstechniken entwickeln zu können.

MontiCore erlaubt die Definition von Modellierungssprachen durch die Definition einer Grammatik der Sprache. Durch eine relativ kompakte Erweiterung dieser Grammatik lassen sich weitere wichtige Sprachattribute für die integrierte Entwicklung festlegen. MontiCore erzeugt für eine gegebene Grammatik ein zugehöriges Lexer/Parser-Paar, um konkrete Modellinstanzen einlesen zu können, wobei intern der weit verbreitete Parsergenerator ANTLR [PQ95] verwendet wird. MontiCore erlaubt es, Grammatiken und damit Sprachen auf verschiedene Weisen zu kombinieren, wobei voneinander unabhängige und damit modulare Definitionen der einzelnen Sprachen zu unabhängig generierten Parsern/Lexern führen. Diese können dann zu einem späteren Konfigurationszeitpunkt in verschiedenen Konstellationen miteinander verbunden werden. Des Weiteren unterstützt MontiCore eine zu ANTLR ähnliche Art der Grammatikvererbung und damit die Wiederverwendung von Grammatikregeln.

Die Kompositionalität kann insbesondere dazu genutzt werden, bereits entwickelte Modellkomponenten in neuen Modellierungssprachen zu verwenden. MontiCore beinhaltet bereits eine Java-Grammatik, so dass Modellierungssprachen gezielt Java-Statements oder -Expressions einbetten können.

Die beim Parsen einer Modellinstanz entstehenden abstrakten Syntaxbäume (AST) sind jedoch stark getypt und unterscheiden sich stark von denen, die von ANTLR standardmäßig erzeugt werden, um möglichst flache Hierarchien zu erzeugen und so die Programmierung von Modelltransformationen zu erleichtern. Die generierten AST-Klassen unterstützen ein im Vergleich zu [GHJV95] erweitertes Visitorenkonzept zur Traversierung von Objektstrukturen, wobei insbesondere die Visitoren bei Modifikation der Klassenhierarchie nicht angepasst werden müssen und ebenso kompositional wie die Parser bezüglich eingebetteter Sprachen sind. Diese Technik kann insbesondere zur Definition von Modelltransformationen, Codegenerierung, Analysen und semantischen Überprüfungen verwendet werden.

Die Codegenerierung innerhalb von MontiCore ist derzeit mit dem Ziel Java ausgeführt, wobei der generierte Code dann wiederum von einem Java-Compiler übersetzt wird. Dabei können der vorhandene Java-PrettyPrinter und die MontiCore-Template-Engine [KR05], deren Templates kompilierbare Klassen sind, genutzt werden.

In MontiCore entwickelte Modellierungssprachen lassen sich als Eclipse-Plugins verfügbar machen. Der aus IDEs für Quellcode bekannte Komfort lässt sich dabei durch automatische Generierung typischer Unterstützungsfunktionen (ebenfalls aus der oben genannten Grammatik) auch für Modelle erreichen und kann teilweise sogar sprachübergreifend für Quellcode und Modelle verfügbar sein. Dabei werden die bekannten Funktionen wie Syntaxhervorhebung, Fehlermeldungen oder Übersichten vom MontiCore-Framework zur Verfügung gestellt. Weitere IDE-typische Funktionen wie Auto-Vervollständigung oder das Anbieten von Vorschlägen für die Fehlerbehebung sind derzeit in der Entwicklung. Es besteht außerdem technische und methodische Unterstützung, einfache generische Modell-Refactorings in Eclipse zu integrieren. In Zukunft soll besonders die sprachübergreifende Anwendung der Refactorings erweitert werden.

Als Proof-of-Concept für den MontiCore-Ansatz ist zu sehen, dass MontiCore selbst mit den oben genannten Techniken entwickelt wird und auch ein teilweises Bootstrapping von Parser- und AST-Generatoren durchgeführt wird.

4 UML/P-Statecharts als Programmiersprache

Zur Demonstration der Ausführungen früherer Abschnitte wird in diesem Abschnitt die Entwicklung einer ausführbaren Modellierungssprache anhand von UML/P-Statecharts [Rum04b] gezeigt. Dabei wurden bewusst Statecharts gewählt, weil diese in vielen Werkzeugen bereits umgesetzt sind und die Umsetzung der Statecharts selbst daher keine Neuerung darstellt. Die UML/P-Statecharts beinhalten dabei Hierarchie ohne parallele Zustände (nur OR-States) und keine History. Dies ist aber für objektorientierte Modellierung im Allgemeinen ausreichend.

Der erste Schritt beim Entwurf einer Modellierungssprache mit MontiCore besteht darin, eine Grammatik geeignet zu definieren, aus der Lexer, Parser und die AST-Struktur automatisch erzeugt werden. Abbildung 1 beinhaltet eine Grammatik für UML/P-Statecharts in der MontiCore-Notation. Die Angabe der Grammatik führt zur Definition von Statecharts mit darin eingebetteten Java-Anweisungen in Zuständen und Transitionen sowie Java-Definitionen von Methoden und Attributen außerhalb des Statechart, die vom Parser ebenfalls verarbeitet und syntaktisch geprüft werden.

Sprachdefinition In Zeile 1 wird der Name („statechartDSL“) für die Modellierungssprache festgelegt. In den Zeilen 2 und 3 können Generierungsparameter (das zu erzeugende Java-Package für die AST-Struktur und der Pfad) sowie weitere, hier nicht vertiefte, Lexer- und Parser-Optionen angegeben werden.

In den Zeilen 4 bis 34 folgt die Angabe der Grammatikregeln, die sowohl die konkrete und als auch die abstrakte Syntax von Statecharts beschreiben. Die erste Regel in Zeile 5 legt fest, dass ein Statechart mit dem Schlüsselwort „statechart“ beginnt, wobei Schlüsselwörter des Modells in der Grammatik durch das vorangehende Ausrufezeichen markiert werden. Darauf folgt ein Identifier (IDENT), der den Namen des Statechart-Modells festlegt. In geschweiften Klammern folgt eine Definition einer Alternative (mit |) (Menge von Zuständen (State), Transitionen (Transition) oder eingebetteter Quellcode (Code)) mit beliebiger Wiederholung markiert durch den Kleene-Stern (*).

Zustände des Statecharts (Zeilen 10-18) beginnen mit dem Schlüsselwort „state“, gefolgt von einem Identifier, mit dem der Name des Zustands angegeben wird. Es folgt eine optionale Angabe (markiert durch ein Fragezeichen), ob es sich um einen initialen oder finalen Zustand handelt, wobei beliebige Kombinationen der Wörter „initial“ und „final“ zulässig sind - in der UML üblichen Angabe als Stereotyp in doppelten spitzen Klammern. Um auf diese Attribute eindeutig zugreifen zu können, werden diese zusätzlich noch mit einem Namen versehen. Im Rumpf eines Zustands (eingeschlossen in geschweiften Klammern, Zeilen 14/15) können „entry“ oder „exit“ Aktionen sowie hierarchisch eingebettete Transitionen und Zustände existieren.

Entry-Aktionen (Zeile 20) beginnen mit dem Schlüsselwort „entry“. Nach einem Doppel-

```
grammar "statechartDSL"
    package "mc.statechart.ast"
    path "gen"
{
  Statechart :
    !"statechart" name:IDENT "{"
      (states:State | transitions:Transition | userCode:Code)*
    "}" EOF;

  State :
    !"state" name:IDENT
    ("<<" (initial:[!"initial"] | final:[!"final"])* ">>")?
    (
      ( "{" (entryAction:EntryAction)? (exitAction:ExitAction)?
            (states:State | transitions:Transition)* "}" )
    |
        ";"
    );

  EntryAction : !"entry" ":" block:$BlockStatement;

  ExitAction : !"exit" ":" block:$BlockStatement;

  Transition :
    from:IDENT "->" to:IDENT
    (":" (event:IDENT
      ( "(" arguments:Argument ( "," arguments:Argument )* ")" )? )?
    ("[" guard: $Expression "]")?
    ("/" action: $BlockStatement)? )? ";";

  Argument : paramType:IDENT paramName:IDENT;

  Code : !"code" code: $Classbody ;
}
```

Abbildung 1: Statechart-Grammatik in MontiCore-Syntax

punkt folgt eingebettet ein Block von Java-Statements. Für jede auf der rechten Seite mit $ beginnenden Regel wird der generierte Statechart-Parser so vorbereitet, dass an dieser Stelle der Parser gewechselt und ein passender innerer Parser aufgerufen wird, der Block-Statements verarbeiten kann. Hat dieser erfolgreich ein BlockStatement geparst, gibt er die Kontrolle wieder an den Statechart-Parser zurück. Die Kombinierbarkeit solcher Parser ist zwar im Recursive-Descent-Verfahren grundsätzlich machbar, aufgrund der Verzahnung der lexikalischen Parser und des zur Optimierung des Laufzeitverhaltens linearisierten Lookaheads nicht einfach und in der aktuell verfügbaren Fassung auch nicht vollständig universell einsetzbar. Auch die Markierung von Schlüsselwörtern über ein Ausrufezeichen ist rein technischer Natur und wird voraussichtlich in kommenden Fassungen nicht mehr nötig sein.

Damit sind alle hier eingesetzten Elemente der MontiCore-Grammatik beschrieben. Die Elemente der Statechart-Sprache sollten weitgehend selbsterklärend sein. Die Einbettung eines Klassenrumpfes bestehend aus weiteren Methoden und lokalen Attributen, die zur

generierten Klasse ergänzt werden, ist durch die Regel in Zeile 33 möglich.

Die Einbettung von Java-Sprachelementen erlaubt eine frühzeitige Fehlermeldung auch bei Java-Syntaxfehlern, die direkt an den Benutzer gemeldet werden können. Diese Fehler werden nicht erst bei der Kompilierung des generierten Codes festgestellt wie es oftmals in Case-Tools üblich ist, sondern direkt bei Erstellung des Quellcodes, wie in der Eclipse-IDE für Java üblich.

Die Statechart-Grammatik bleibt unverändert und wird auch nicht neu zur Generierung benötigt, wenn Java z.B. gegen C++ ausgetauscht werden soll, da keine statische Bindung zwischen den kompositionalen Parsern besteht.

Abstrakte Syntax Eine Kernidee von MontiCore ist es, aus einer Grammatikbeschreibung nicht nur ein Parser/Lexer-Paar zu erzeugen, sondern auch heterogen getypte AST-Klassen. Die aus der Statechart-Grammatik entstehenden AST-Klassen lassen sich im Klassendiagramm in Abbildung 2 erkennen. Dieses Wissen ist allerdings für den Nutzer von Statecharts irrelevant und betrifft nur Werkzeug-Entwickler und -Adaptoren, die weitere Analysen oder spezielle Generierungstechniken für einen bereits vorhandenen abstrakten Syntaxbaum („Metamodell“) entwerfen wollen.

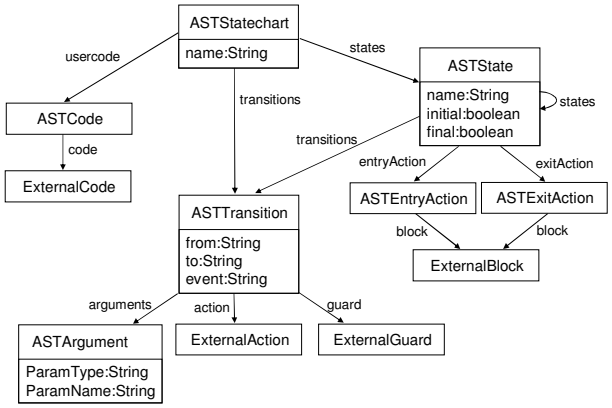

Abbildung 2: UML-Klassendiagramm der aus der Statechart-Grammatik generierten AST-Klassen (vgl. Abbildung 1)

Wie bei der Codegenerierung, bei der auf eine einfache und nachvollziehbare Übersetzung des Modells in Code Wert gelegt wird, liegt auch der Beziehung von Grammatik-Regeln zur generierten AST-Struktur eine schematische Abbildung zu Grunde, die dem Werkzeug-Entwickler hilft, die Schnittstellen zum abstrakten Syntaxbaum zu nutzen.

Jede Regel der Grammatik wird auf eine AST-Klasse abgebildet, wobei die linke Seite einer Regel den Klassennamen und die rechte Seite die Attribute der Klasse bestimmt. Aus Zeile 5 wird deshalb die AST-Klasse `ASTStatechart` generiert, die u.a. das Attribut `states` enthält (das Wort vor dem Doppelpunkt legt den Namen des Attributs fest). Der Typ des Attributs in den Klassen ist eine Liste von States im Sinne der Java Collections

List mit generiertem Typ `ASTState`.

Abbildung 3 zeigt exemplarisch die aus der Regel „Statechart“ entstehende AST-Klasse `ASTStatechart` und die bereits erwähnte Assoziation zur AST-Klasse `ASTState`. Angegeben sind ebenfalls zwei wichtige Methoden der AST-Klassen. Zum einen handelt es sich um die `traverse`-Methode, die benötigt wird, um die Traversierung durch einen Visitor zu steuern. Zum anderen wird für jede AST-Klasse eine `deepClone`-Methode generiert, mit der ein beliebiger Teil-AST rekursiv ab dem angegeben AST-Knoten kopiert werden kann.

Die genannten Attribute sind allerdings in den Klassen gekapselt und nur über entsprechende Methoden bzw. den Visitor-Mechanismus zugänglich. Zusätzlich zur Navigation entlang der Baumhierarchie nach unten ist es auch möglich nach oben zu navigieren. Die dazu notwendige Infrastruktur wird automatisch generiert, bleibt aber dem Werkzeug-Entwickler verborgen.

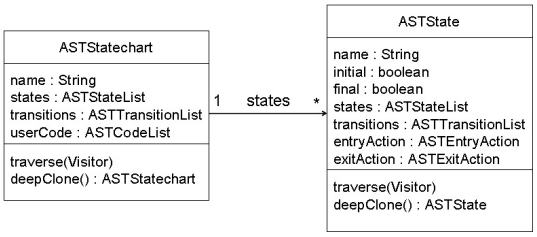

Abbildung 3: Ausschnitt aus der AST-Klassenstruktur für die Regel „Statechart“

Generierung von Code Auf Basis des generierten AST und mit Hilfe der Parser für Statecharts und Java und weiterer durch MontiCore zur Verfügung gestellter Infrastruktur (Visitorkonzept, Template-Engine) lässt sich nun eine Codegenerierung für Statecharts entwickeln, die die Ausführungssemantik widerspiegelt.

Bei der Codegenerierung für Statecharts wird wie gefordert ein Mapping von Elementen der Modellierungssprache auf Java-Code mit eindeutig definierter Schnittstelle vorgenommen. Ein Statechart wird auf eine Java-Klasse mit demselben Namen abgebildet. Die Events werden als Methodenaufrufe mit entsprechenden Parametern als Schnittstelle nach außen zur Verfügung gestellt. Über einen Callback auf übergebene Parameter ist ein Informationsfluss aus dem Statechart heraus zur restlichen Anwendung möglich. Weitere generierte Methoden und Klassen bleiben intern und haben keinen Effekt auf das nach außen sichtbare Verhalten. Es sei aber hier verraten, dass wahlweise das State-Muster oder eine doppelte Case-Anweisung zum Einsatz kommen kann [Rum04a].

IDE-Einbettung in Eclipse Wie angedeutet wird auch ein großer Teil der Erzeugung eines Eclipse-Plugins für die neue Sprache zur Verfügung gestellt. Automatisch generiert werden können Syntaxhervorhebung für Keywords, Strings etc., eine Übersicht und die Anzeige von Fehlern, die durch den Parser oder manuell erstellte semantische Checks

auf Basis der AST-Struktur gefunden werden. Manuell zu erstellen ist derzeit auch die Integration als Eclipse „Nature“, mit der ein inkrementelles Build des Statechart möglich ist.

Aus Platzgründen werden Details der Codegenerierung mittels Template-Engine, semantische Überprüfungen mit dem Visitorkonzept und Einzelheiten der Eclipse-Pluginerstellung nicht näher behandelt.

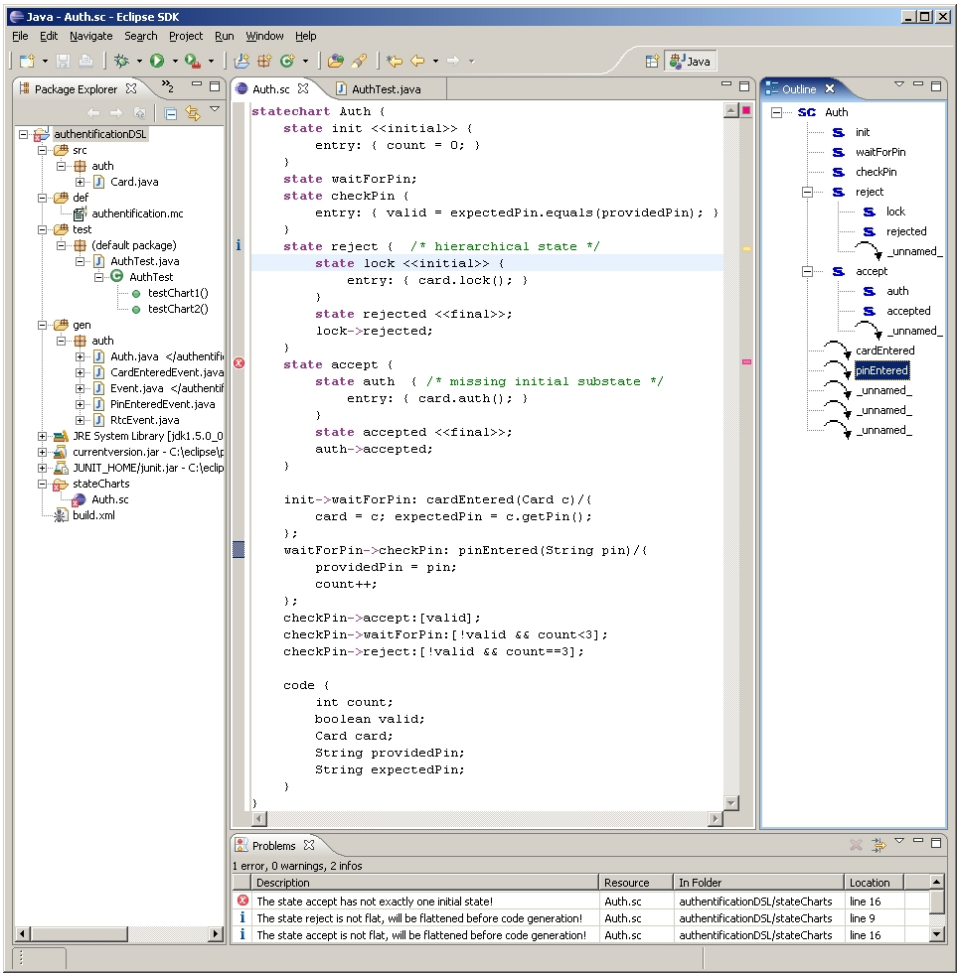

Abbildung 4: Eclipse-integrierte Statechart Modellierung

Abbildung 4 zeigt die vollständige Integration von Statecharts in ein Eclipse Java-Projekt. Das modellierte Beispiel ist zum Vergleich als UML/P-Statechart in graphischer Notation in Abbildung 5 zu sehen. Modelliert wird ein einfacher Authentifizierungsmechanismus. Nachdem zu Beginn eine Karte eingegeben wurde, kann der Benutzer bis zu drei-

mal eine PIN-Eingabe vornehmen. Bei der dritten falschen PIN wird die Karte gesperrt, ansonsten erfolgt die Authentifizierung der Karte. Die Übersicht auf der rechten Bildseite zeigt die Struktur des hierarchischen Automaten. Im Editor-Fenster ist der Quelltext des Statecharts zu sehen. Entsprechende Marker weisen auf Probleme hin, die auch im „Problems“-Fenster in der unteren Bildhälfte angezeigt werden. Im Beispiel wird die Information angezeigt, dass für die Codegenerierung ein nicht-hierarchisches Statechart erzeugt werden muss (ein entsprechender Algorithmus hierfür wurde ebenfalls mit Hilfe der MontiCore Infrastruktur erstellt). Eine weitere semantische Überprüfung weist darauf hin, dass für den hierarchischen Zustand „accept“ kein Startzustand für die Unterzustände definiert wurde. Nach Behebung des Fehlers wird automatisch der entsprechende Java-Code generiert (sichtbar in der linken Bildhälfte im Package Explorer; es entsteht die Schnittstellenklasse `Auth.java` und Hilfsklassen, die Events und deren Parameter kapseln).

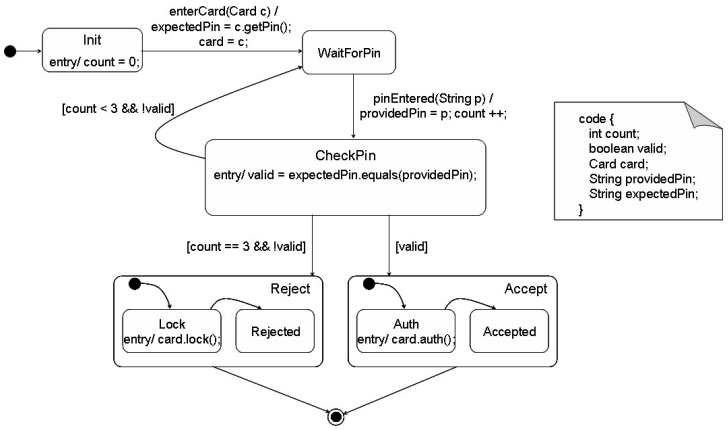

Abbildung 5: UML/P Statechart in graphischer Notation

Testen von Modellen Für eine weitgehende Integration von Modellen in die Quellcodeentwicklung ist in Abschnitt 2 die Testbarkeit auf Modellebene identifiziert worden. Wird wie in diesem Fall ein Modell zur Konstruktion von Code eingesetzt, so sind alternative Beschreibungen notwendig, um die Korrektheit des definierten Modells zu prüfen. Grundsätzlich könnte sonst ein Statechart, wie bei modellbasierten Tests üblich [Pre03], zur Testfallgenerierung eingesetzt werden.

Einfache Tests eines Statecharts können, wie in der Quellcodeentwicklung sonst auch üblich, mit Hilfe von Unit-Tests erfolgen. Abbildung 6 zeigt, wie sich so ein bestimmtes Verhalten des Statechart überprüfen lässt, indem hier indirekt über die Schnittstelle der Klasse `Card` getestet wird, die die generierte Klasse einsetzt. Darüber hinaus wird hier ebenfalls der Zugriff auf die einzelnen Elemente der Statecharts illustriert.

Eine noch offene grundsätzliche Überlegung betrifft die Frage, ob zum Testen von Modellen zusätzliche Funktionen generiert werden sollen, die es erlauben, weitere Interna eines Modells zu lesen bzw. zu modifizieren. So können für ein Statechart zum Beispiel Me-

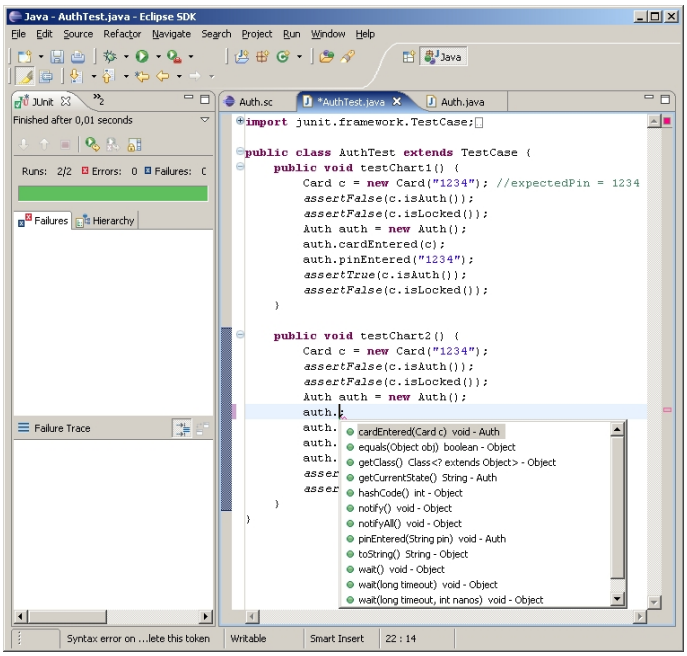

Abbildung 6: Unit Test des Statechart

thoden zum Auslesen und zur Modifikation des aktuellen Zustands definiert werden, die im Produktivsystem nicht zugänglich sind. Darüber hinaus kann im Statechart der aktuelle Trace gespeichert und bei einem automatisierten Test auf Richtigkeit geprüft werden. Solche Inspektionen der Internas sind allerdings in automatischen Tests vorsichtig einzusetzen, da sie die Modifikation des Statechart-Modells aufwändiger machen.

Die in diesem Abschnitt vorgestellte Modellierungssprache und die zugehörigen Technologien sind ein konkretes Beispiel für eine effiziente Integration von Modellen in eine quellcodebasierte Entwicklung. Das Hauptziel bei der Entwicklung von MontiCore ist es, dafür effiziente Unterstützung so generisch wie möglich zur Verfügung zu stellen. Es wird daher ein hoher Automatisierungsgrad bei der Erstellung der Modellierungssprache, bei der Erzeugung von Codegenerierung, IDE-Integration und später auch Analysen, Testfalldefinition und Transformationstechniken für beliebige Modellarten in der Softwareentwicklung angestrebt.

Viele Mechanismen können wie beschrieben bereits automatisch generiert werden. Zurzeit ist eine Erweiterung der Unterstützungsfunktionen geplant, so dass auch komplexere Refactorings oder ein Debugging von beliebigen ausführbaren Modellen möglich wird.

Weitere verwandte Arbeiten Für den methodischen Einsatz von Statecharts in der Softwareentwicklung gibt es eine Vielzahl von Konzepten und Werkzeugen, die hier nur exem-

plarisch beschrieben werden können. So beschreibt beispielsweise [GZ02] Fujaba-Statecharts, die aber den Integrationsgrad dieses Ansatzes nicht erreichen.

In [Dmi04] ist eine Erweiterung einer integrierten Entwicklungsumgebung dargestellt, die einen einfachen Entwurf von textuellen domänenspezifischen Sprachen erlaubt. Im Gegensatz zu unserer Arbeit verwenden die erzeugten Editoren schablonenartige Masken.

Executable UML [MB02] beschreibt eine Teilmenge der UML, die zur Ausführung geeignet ist, nutzt aber eine eigene, nicht mehr aktuelle Aktionssprache. Wir ziehen die Verwendung vorhandener Programmiersprachen vor.

Viele Entwicklungswerkzeuge wie Poseidon [Gen], Rhapsody [Rha], Statemate [Sta], erlauben verschiedene Varianten der Codegenerierung aus Statecharts. Keines dieser Werkzeuge erlaubt jedoch eine nahtlose Integration der Modelle mit Quellcode, sondern es müssen stets Kompilierungsprozesse angestoßen werden. Die generierten Klassen sind teilweise kompliziert und von Laufzeitumgebungen abhängig. Außerdem sind die Schnittstellen häufig nicht explizit ausgewiesen.

5 Zusammenfassung und Ausblick

In diesem Artikel haben wir Anforderungen dargestellt und in einer Studie teilweise umgesetzt, die eine nahtlose Integration von Quellcode und Modellen in einer agilen Entwicklung erlauben. Dieses erscheint uns ein wichtiger Schritt zur modellbasierten Softwareentwicklung, die in den nächsten Jahren weiter an Bedeutung gewinnen wird. Der Zwischenschritt einer solchen Integration kann unserer Ansicht nach nur gelingen und auf breite Akzeptanz stoßen, wenn sie den Entwicklern den intuitiven Umgang mit Modellen in einer quellcodebasierten Softwareentwicklung ermöglicht. Die dazu notwendigen Integrationsschritte haben wir exemplarisch anhand einer Fallstudie für Statecharts gezeigt und dabei auch die Möglichkeiten zur Automatisierung der Erstellung von Entwicklungswerkzeugen für beliebige Modelle durch das Modellierungswerkzeug MontiCore beschrieben.

Geplant sind Arbeiten an der Realisierung weiterer Module für MontiCore, um die Integration von Modellen und Quellcode weiter zu erleichtern. MontiCore spielt dabei in zweierlei Hinsicht eine zentrale Rolle: zur Erleichterung der Integration von beliebigen Modellen in andere Projekte und als Testprojekt für konkrete Modelle im Zuge des fortschreitenden Bootstrapping-Prozesses.

Literatur

[Bez05] J. Bezivin. On the Unification Power of Models. *Software and System Modeling (SoSym)*, 4(2):171–188, 2005.

[BS01] M. Broy und K. Stølen. *Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement*. Springer Verlag Heidelberg, 2001.

[CE00] K. Czarnecki und U. Eisenecker. *Generative Programming*. Addison-Wesley Boston, 2000.

[Dmi04] Sergey Dmitriev. Language Oriented Programming: The Next Programming Paradigm, Nov 2004. http://www.onboard.jetbrains.com/is1/articles/04/10/lop/mps.pdf.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley, 1999.

[Gen] Gentleware Poseidon. http://gentleware.com/.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GZ02] Leif Geiger und Albert Zündorf. Statechart Modeling with Fujaba. In *International Conference on Graph Transformations, Barcelona, Spain*, 2002.

[KR05] Holger Krahn und Bernhard Rumpe. Techniques Enabling Generator Refactoring. Bericht TR-CCTC/DI-36, Centro de Ciencias e Tecnologias de Computacao, Departamento de Informatica Universidade do Minho, Braga, Portugal, 2005.

[KWB03] Anneke Kleppe, Jos Warmer und Wim Bast. *MDA Explained. The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.

[MB02] Stephen J. Mellor und Marc Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. Foreword By-Ivar Jacoboson.

[OMG01] OMG. Action Semantics for the UML. Response ot OMG RFP ad/98-11-01. Bericht OMG Document ad/2001-08-04, Object Management Group, 2001.

[OMG05a] Object Management Group. MDA (Model Driven Architecture), 2005.

[OMG05b] Object Management Group. UML Superstructure Specification, v2.0, 2005.

[PQ95] T. J. Parr und R. W. Quong. ANTLR: A Predicated-LL(k) Parser Generator. *Software – Practice and Experience*, 25(7):789–810, 1995.

[Pre03] A. Pretschner. *Zum modellbasierten funktionalen Test reaktiver Systeme*. Dissertation, Technische Universität München, 2003.

[Rha] I-Logix Rhapsody. http://www.ilogix.com/sublevel.aspx?id=53.

[Rum96] B. Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Herbert Utz Verlag Wissenschaft, 1996.

[Rum02] B. Rumpe. Executable Modeling with UML. In *Issues & Trends of Information Technology Management in Contemporary Associations, Seattle*, Seiten 697–701, Hershey, London, 2002. Idea Group Publishing.

[Rum04a] Bernhard Rumpe. *Agile Modellierung mit der UML*. Springer, Berlin, 2004.

[Rum04b] Bernhard Rumpe. *Modellierung mit der UML*. Springer, Berlin, 2004.

[Sch98] S. Schiffer. *Visuelle Programmierung. Grundlagen und Einsatzmöglichkeiten*. Addison-Wesley, 1998.

[Sta] I-Logix Statemate. http://www.ilogix.com/sublevel.aspx?id=74.

[Tol04] J. Tolvanen. Making model-based code generation work. *Journal on Embedded Systems Europe*, Aug/Sept, 2004.