

FOOSE — Eine integrierte formale Softwareentwicklungsmethode

Alexander Knapp

Ludwig-Maximilians-Universität München
knapp@informatik.uni-muenchen.de

Die Methode “FOOSE” (formal Object-Oriented Software Engineering) exemplifiziert eine auf der “Unified Modeling Language” (UML) basierende, integrierte formale Softwareentwicklungsmethode: Jacobsons “Object-Oriented Software Engineering”-Methode (OOSE) wird um diagrammgestützte, formale Spezifikationen, Beweisverpflichtungen und Verfeinerungen erweitert. Die diagrammatischen UML-Modelle werden um Anmerkungen, etwa Invarianten, in einer formalen, algebraischen Sprache optional ergänzt. Aus der Semantik der Diagramme und der Anmerkungen kann halbautomatisch eine ablauffähige Spezifikation in der objektorientierten, algebraischen Spezifikationssprache Maude abgeleitet werden: Ein automatisch erstelltes Spezifikationsgerüst muß vom Benutzer unter Ausnutzung der formalen Annotationen vervollständigt werden. Diese Spezifikationen erlauben insbesondere die Verifikation von Verfeinerungsbeziehungen zwischen Modellen. Schließlich bildet die semi-automatisch erstellte Spezifikation aus einem gegebenen, annotierten Softwaremodell auch die Grundlage für eine korrekte Implementierung dieses Modells in der objektorientierten, nebenläufigen Programmiersprache Java.

1 Einleitung

Integrierte formale Softwareentwicklungsmethoden [Kem90] verbinden bekannte Softwareprozesse und wohletablierte diagrammatische Entwurfstechniken mit formalen Methoden, um die Entwicklung zuverlässiger, korrekter Software durch die Verwendung von praxistauglichen Ingenieursstandards einerseits und formalen Spezifikationen, Verfeinerungen und Beweisverpflichtungen andererseits zu unterstützen. So ergänzt etwa Colemans “Fusion” [CAB⁺94] die informellen Methoden und Techniken der objektorientierten Entwicklungsansätze von Rumbaugh, Booch und Wirfs-Brock um Modellannotierungen in strukturiertem Englisch und formale Lebenszyklenausdrücken. Der “Syntropy”-Ansatz von Cook und Daniels [CD94] kombiniert die Notationen aus Rumbaughs “Object Modeling Technique” (OMT) mit der modellbasierten Spezifikationsprache Z. In Araujos “Metamorphosis” [Ara96] wird eine Fusion-ähnliche Notation halbautomatisch in die objektorientierte Z-Variante Object-Z übersetzt; der Benutzer muß Spezifikationsrahmen manuell ergänzen. Nakajima und Futatsugi [NF97] integrieren eine szenarienbasierte, an Jacobsons “Object-Oriented Software Engineering” (OOSE) angelehnte Methode mit der hybriden formalen Methode GILO-2, die auf algebraischen Spezifikationen und einer Z-Variante beruht. Lanos “Formal Object-Oriented Development” (OOFD [Lan95]) übersetzt statische und dynamische OMT-Modelle in die objektorientierten Z- und VDM-Erweiterungen Z++ und VDM++, und überführt diese Spezifikationen in Code. Die “Fox”-Methode von Achatz und Schulte [AS97] basiert auf Fusion, ergänzt aber die semiformalen Fusion-Notationen durch formale Annotationen in Object-Z und verbindet Fusion-

Schritte mit der Erzeugung und Erledigung von Beweisverpflichtungen und dem Nachweis von Verfeinerungsbeziehungen.

In den letzten Jahren hat sich die “Unified Modeling Language” (UML [BRJ98]) als die Standard-Softwaremodellierungssprache für objektorientierte Systeme durchgesetzt. Die UML baut wesentlich auf Notationen von Rumbaugh’s OMT, der Booch-Methode und Jacobson’s OOSE auf, definiert aber einerseits, in Ergänzung der Vorgängersprachen, eine präzise abstrakte Syntax und eine zumindest informelle Semantik ihrer Modelle, und sieht andererseits bereits die Anfügung von formalen Anmerkungen an Modelle vor. Die UML bietet damit eine allgemein akzeptierte, geeignete Grundlage für die Entwicklung einer integrierten formalen Entwicklungsmethode.

Wir stellen im folgenden, aufbauend auf den Ergebnissen der Dissertation [Kna01], die fOOSE-Methode (“formal Object-Oriented Software Engineering”) vor, eine auf der UML basierende, integrierte formale Erweiterung von OOSE. Einerseits übertragen wir die Ergebnisse bereits bestehender integrierter formaler Methoden auf die neuen semantischen Anforderungen der UML, andererseits ergänzen wir Auslassungen der verfügbaren Ansätze zu einer tatsächlich weitgehend automatisierten, bewiesenermaßen korrekten Methodik. In [Kna01] wird dazu zunächst eine formale Semantik für den in fOOSE benutzten UML-Teil, annotierte statische Strukturen und Kollaborationen, in linearer Temporallogik vorgestellt. Es wird eine Übersetzung der annotierten UML-Teilsprache in die objektorientierte, algebraische Spezifikationsprache Maude [CDE⁺99], spezieller sogenannte Maude-Objekttheorien, sowie in Beweisverpflichtungen angegeben und, unter Verwendung einer operationalen Maude-Semantik, gezeigt, daß die Übersetzung bezüglich der Temporallogiksemantik korrekt ist. Auf Maude-Objekttheorien wird weiters ein UML-adäquater Verfeinerungsbegriff eingeführt. Schließlich wird die UML-Teilsprache in die nebenläufige, objektorientierte Programmiersprache Java [GJS96] übersetzt und diese Übersetzung, aufbauend auf einer strukturell-operationalen Semantik der Programmiersprache Java, als korrekt bezüglich der Maude-Übersetzung bewiesen.

2 fOOSE

Jacobson’s “Object-Oriented Software Engineering”-Methode (OOSE [JCJÖ93]) besteht aus vier Hauptprozessen: *Analyse*, *Konstruktion*, *Komponenten* und *Test*, die in Teilprozesse aufgespalten, iteriert und verwoben werden. Die Systementwicklung geht von einer gegebenen Anforderungsspezifikation aus. Die erste Analysephase, die *Anforderungsanalyse*, begrenzt das System und definiert seine Funktionalität; identifiziert Geschäftsobjekte; und beschreibt möglicherweise die Mensch-Maschine-Schnittstelle. Das daraus resultierende *Anforderungsmodell* wird in der zweiten Analysephase, der *Robustheitsanalyse* strukturiert; die Funktionalität wird auf Schnittstellen-, Daten- und Kontrollobjekte verteilt. Es ergibt sich ein *Analysemodell*. Die Konstruktionsphase ist in zwei Teilprozesse aufgespalten: *Entwurf* und *Implementierung*. Im Entwurf wird das Analysemodell einem gegebenen Implementierungsumfeld angepaßt; die Interaktionen der Entwurfsobjekte und

ihr internes Verhalten werden definiert und zu einem *Entwurfsmodell* zusammengefaßt. Die Implementierung setzt das Entwurfsmodell in ein *Implementierungsmodell*, bestehend aus annotiertem Quellcode in einer Programmiersprache, um. Der Konstruktionsprozeß wird vom Komponentenprozeß, der Wiederverwendung von bestehenden Entwurfsmodellen und Bibliothekscode, begleitet. Schließlich validiert der Test das Entwurfs- und das Implementierungsmodell bezüglich des Anforderungsmodells und erzeugt ein *Testmodell*. Die “formal Object-Oriented Software Engineering” (fOOSE)-Methode behält die Struktur und die Schritte von OOSE bei, ergänzt aber, aufbauend auf einer präzisen OOSE-Modellsemantik, die Anforderungs-, Analyse- und Entwurfsmodelle um *funktionale Spezifikationen* der verwendeten Datentypen und *formale Annotationen*; leitet daraus formale, *ausführbare Spezifikationen* ab; erzeugt aus den Modellen *Beweisverpflichtungen*; fügt Entwicklungsschritte zur Erledigung von Beweisverpflichtungen und zum Beweis von *Verfeinerungsbeziehungen* zwischen Modellen ein; und kombiniert die Implementierung mit *Codeerzeugung*. Abb. 1 zeigt einen Überblick über die fOOSE-Erweiterungen (grau; ohne Iterationen). Auf der technischen Seite [Kna01] verwendet fOOSE die “Unified Modeling Language” (UML [BRJ98]) als Modellierungsnotation, bedingte Gleichungslogik als Anschriftensprache, die ausführbare objektorientierte, algebraische Spezifikationsprache Maude [CDE⁺99] als Spezifikationsprache und die objektorientierte, nebenläufige Programmiersprache Java [GJS96] als Implementierungssprache. Die entscheidende Eigenschaft der fOOSE-Methode ist die simultane Verwendung traditioneller, diagrammatischer Notationen, Datentypspezifikationen und formaler Anno-

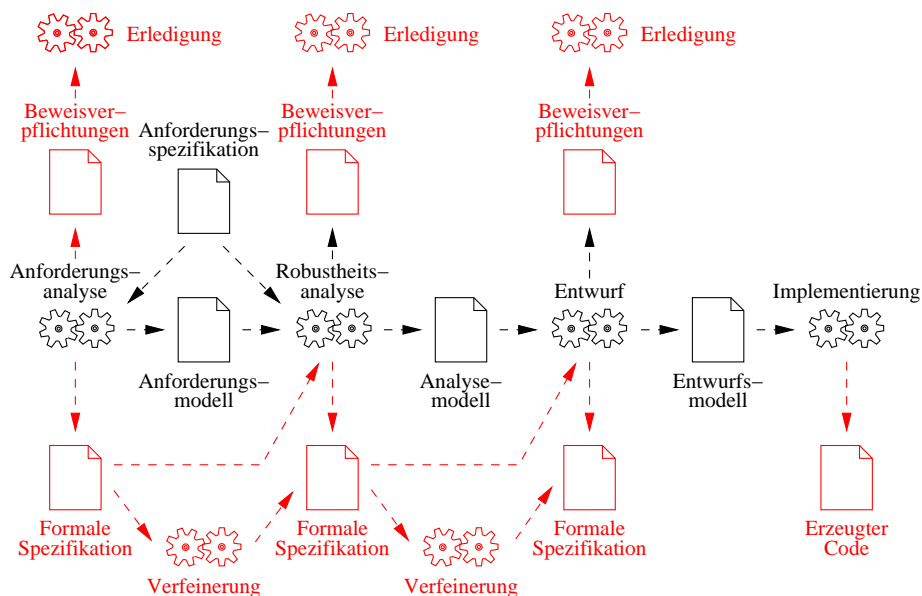


Abbildung 1: fOOSE-Erweiterungen von OOSE

tationen auf der Basis eines verbindenden formalen Modells. Die aus solch heterogenen fOOSE-Modellen erzeugten Spezifikationen und Implementierungen integrieren diese scheinbar unterschiedlichen Formalitätsebenen. Annotierungen erlauben die Aufstellung von Systemeigenschaften, die nicht einfach oder einfach nicht diagrammatisch darstellbar sind, etwa Invarianten, Nebenläufigkeitseinschränkungen, Fairnessannahmen &c. Die Generierung formaler Spezifikationen integriert die Semantik der annotierten Diagramme, die resultierenden Spezifikationen erlauben eine formale Überprüfung, ob die gewünschten Systemeigenschaften automatisch gegeben sind oder manuell gewährleistet werden müssen. Diese Analyse beeinflusst die folgenden Entwicklungsschritte und bildet die Basis formaler Verfeinerungsbeweise. Automatische generierte Spezifikationen und Implementierungen können allerdings unvollständig sein und müssen durch den Entwickler vervollständigt werden. Die Verbindung automatischer Generierung mit der Erzeugung von Beweisverpflichtungen, die bei der Vervollständigung von Spezifikationen oder Implementierungen erledigt werden müssen, garantiert aber die Korrektheit der komplettierten formalen Systemrepräsentation zumindest partiell.

Zusätzlich sind die fOOSE-Erweiterungen von OOSE konservativ: Die Entwicklung kritischer Systemteile kann alle oder ausgewählte fOOSE-Schritte und Modellerweiterungen beinhalten, für andere Teile aber der konventionelle OOSE-Entwicklungsprozeß benutzt werden. Schließlich wird das Testen einerseits durch frühe Simulation unterstützt, andererseits wird die Validierung durch formale Verifikation komplementiert.

3 Fallstudie: Eine Leergutannahmemaschine

Wir illustrieren die fOOSE-Methode mit einer kleinen Fallstudie, der Entwicklung einer Leergutannahmemaschine, die auch als durchgehendes Fallbeispiel in Jacobsons Darstellung der OOSE-Methode dient [JCJÖ93]: Eine Leergutannahmemaschine nimmt Leergut, etwa Flaschen oder Dosen, von einem Kunden an; das System registriert wie viele Gebinde eines Typs zurückgegeben werden und druckt eine Quittung über die retournierten Güter aus, ein Systemadministrator kann eine Gesamtübersicht über die an einem Tag zurückgegebenen Gebinde anfordern. Wir beschränken uns allerdings auf die Darstellung der Anforderungsanalysephase, zeigen aber dennoch die fOOSE-Codeerzeugungsmöglichkeiten auch in dieser frühen Phase. Dieser grobe Überblick stützt sich auf ein lediglich intuitives Verständnis der UML-Anwendungsfall-, Klassen- und Kollaborationsdiagramme sowie der Maude-Spezifikationsprache und der Programmiersprache Java.

3.1 Anforderungsanalyse

Die fOOSE-Anforderungsanalysephase dient der Aufstellung eines Anforderungsmodells, der Abgrenzung des Systems und der Spezifikation seines Verhaltens. Das Anforderungsmodell besteht aus einem Anwendungsfallmodell: Aktoren legen fest, was außerhalb des

Systems existiert, Anwendungsfälle definieren, was das System leisten soll; einem Geschäftsobjektmodell zur Festlegung der logischen Geschäftsobjekte; und, in Erweiterung von OOSE, einem Interaktionsmodell, das zeigt, wie Aktoren mit den Geschäftsobjekten des Systems interagieren und wie diese Objekte Nachrichten austauschen.

Das Anwendungsfallmodell wird als UML-Anwendungsfalldiagramm mit Anwendungsfallbeschreibungen notiert, das Geschäftsobjektmodell als UML-Statistisches Strukturdiagramm, das Interaktionsmodell als UML-Sequenz- oder Kollaborationsdiagramm. Statische Strukturdiagramme definieren Klassen von Objekten mit ihren Attributen und Operationen, sowie Assoziationen zwischen den Klassen. Sequenzdiagramme bilden die zeitlich geordneten, eventuell nebenläufigen Interaktionen mehrerer Objekte ab, Kollaborationsdiagramme zeigen zusätzlich die Assoziationen zwischen den zusammenarbeitenden Objekten, ersetzen aber die graphische Repräsentation der partiellen Interaktionsordnung durch Sequenznummern.

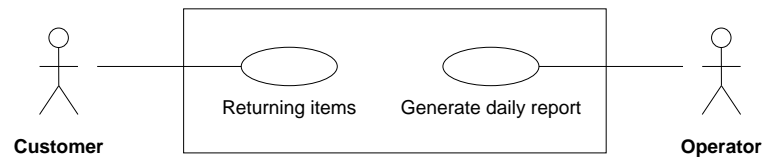
In zusätzlicher Erweiterung des OOSE-Anforderungsmodells werden in fOOSE die Basisdatentypen des Geschäftsobjektmodells durch funktionale Spezifikationen der gewählten Spezifikationsprache Maude formal beschrieben. Das Geschäftsobjektmodell wird um formale Annotationen in bedingter Gleichungslogik angereichert, Invarianten der Systemobjektstruktur, die während der Ausführung immer gelten müssen. Formale Annotationen des Interaktionsmodells drücken Nachrichtenabhängigkeiten aus. Aus diesen erweiterten Modellen wird, unter Verwendung der funktionalen Datentypspezifikationen eine formale Spezifikation des Anforderungsmodells halbautomatisch abgeleitet, d. h., es wird eine Maude-Rahmenspezifikation generiert, die einige von Hand auszufüllende Lücken enthält. Die Vervollständigungsprozedur wird dabei durch die Objektinvarianten unterstützt, die Beweisverpflichtungen erzeugen. Die ausführbare, vervollständigte Maude-Spezifikation kann durch die Bereitstellung geeigneter Startkonfigurationen getestet werden.

Anwendungsfälle. Es gibt zwei Aktoren für die Leergutannahmemaschine: den Kunden und den Betreuer. Das System stellt die Funktionalität zur Rückgabe von Leergut durch den Kunden und zur Abfrage der an einem Tag zurückgegebenen Leergüter durch den Betreuer bereit. Wir leiten daraus das Anwendungsfalldiagramm in Abb. 2(a) und die folgenden Anwendungsfallbeschreibungen ab:

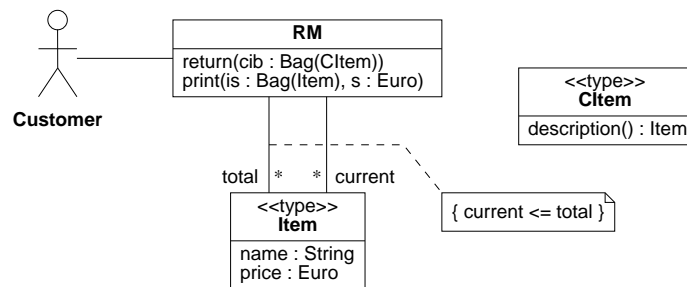
Returning items Ein Kunde gibt Leergut, z. B. Dosen oder Flaschen, an die Leergutannahmemaschine zurück. Es werden Beschreibungen des Leerguts gespeichert und die Tagessumme an retourniertem Leergut über alle Kunde erhöht. Der Kunde erhält eine Quittung für alle von ihm retournierten Gebinde. Die Quittung enthält eine Beschreibung der zurückgegebenen Gebinde und die Gesamtabrechnungssumme.

Generate daily report Der Betreuer erhält einen Ausdruck, wie viele Leergüter eines Gebindetyps zurückgegeben wurden und wie viele Leergüter an diesem Tag insgesamt retourniert wurden. Die zurückgegebene Gesamtanzahl wird zurückgesetzt.

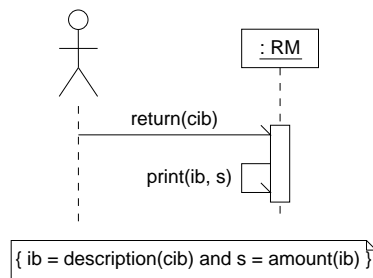
Wir behandeln im folgenden lediglich den ersten Anwendungsfall.



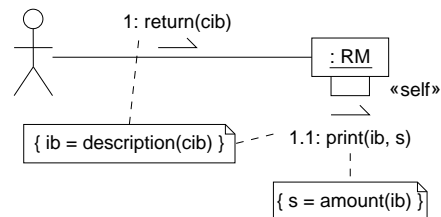
(a) Anwendungsfallmodell



(b) Geschäftsobjektmodell



(c) Sequenzdiagramm



(d) Kollaborationsdiagramm

Abbildung 2: Anforderungsmodell der Leergutannahemaschine

UML-Modell. Das System wird, in erster abstrakter Repräsentation des Anwendungsfalls “Returning items”, als einzelne Klasse RM modelliert, die die Leergutannahemaschine repräsentiert; s. Abb. 2(b).

Der Klasse RM ist eine Klasse Item assoziiert; die multiple Assoziation current enthält alle von einem Kunden zurückgegebenen Leergüter, die multiple Assoziation total speichert alle Leergüter, die am selben Tag zurückgegeben wurden. Der Datentyp Item wird

zur Systemrepräsentation konkreter durch den Datentyp `CItem` dargestellter Gebinde, wie Flaschen oder Dosen, verwendet; aus jedem konkreten Gebinde kann vermittels der Funktion `description` ein systeminternes Gebinde errechnet werden. Jedes Item hat einen Preis (in Euro). Die Klasse `RM` definiert zwei Methoden, `return` und `print`, die die gewünschte Funktionalität der Leergutannahemaschine bereitstellen.

Das Sequenzdiagramm in Abb. 2(c) zeigt die Interaktion zwischen dem Kunden und der Leergutannahemaschine. Der Kunde sendet eine asynchrone `return`-Nachricht mit mehreren zurückzugebenden Leergütern an die Leergutannahemaschine, d. h. einen Aufruf dieser Methode, ohne auf ein Ergebnis zu warten. Die Maschine druckt dann eine Quittung mit Beschreibungen der zurückgegebenen Gebinde und deren Gesamtwert durch Aufruf ihrer eigenen Methode `print` aus. Das Diagramm läßt insbesondere offen, ob durch diese Methodenaufrufe Attribute von `RM` verändert werden. Alternativ kann das Sequenzdiagramm auch als Kollaborationsdiagramm dargestellt werden, s. Abb. 2(d) (hier werden die formalen Annotationen auf lediglich zwei Nachrichten bezogen aufgespalten.)

Die Invariante des Geschäftsobjektmodells sagt aus, daß die Menge der aktuell zurückgegebenen Gebinde immer eine Teilmenge der insgesamt zurückgegebenen Gebinde sein muß. Die Abhängigkeit zwischen der `return`- und der `print`-Nachricht im Kollaborationsdiagramm verlangt, daß die Gebindebeschreibungen, die auf der ausgedruckten Quittung aufscheint, tatsächlich den Beschreibungen der zurückgegebenen konkreten Gebinde entsprechen müssen. Die Annotation der `print`-Nachricht bedeutet, daß der Gesamtwert der zurückgegebenen Gebinde auf der Quittung aufscheinen muß.

Funktionale Spezifikationen. Die durch funktionale Spezifikationen zu definierenden Datentypen des Geschäftsobjektmodells sind: `CItem`, `Item` und `Euro`, sowie Multimengen aus `CItem` und `Item`. Insbesondere müssen die Funktionen auf diesen Datentypen, die in den formalen Annotationen vorkommen, berücksichtigt werden. Andererseits können vordefinierte Bibliotheksspezifikationen, etwa für die ganzen Zahlen, Listen, &c., wiederverwendet werden.

In Maude lassen sich diese Spezifikationen wie folgt formulieren:

```
fmod EURO is
  sort Euro .
  op 0 : -> Euro .
  op _+_ : Euro Euro -> Euro
                                     [comm assoc id: 0] .
endfm

fth ITEM is
  protecting STRING . -Zeichenketten
  protecting EURO .
endft

sort Item .
op name : Item -> String .
op price : Item -> Euro .
endft

fth CITEM is
  protecting ITEM .
  sort CItem .
  op description : -> Item .
endft
```

Für die funktionalen Spezifikationen von Multimengen aus `Item` und `CItem` sehen wir eine generische Multimengenspezifikation als gegeben an, die wir passend instanziiieren:

```
fmod RDATA is
  protecting EURO ITEM CITEM .
  protecting BAG[Item] BAG[CItem] .
```

```

op _<=_ : Bag[Item] Bag[Item] -> Bool .
op description : Bag[CItem] -> Bag[Item] .
op amount : Bag[Item] -> Euro .

var I : Item . var C : CItem .
var Ib : Bag[Item] . var Cb : Bag[CItem] .
eq (empty <= Ib) = true .
eq (I Ib) <= Ib' = if (I in Ib') then (Ib <= Ib') else false fi .
eq description(empty) = empty .
eq description(C Cb) = description(C) description(Cb) .
eq amount(empty) = 0 .
eq amount(I Ib) = price(I)+amount(Ib) .
endfm

```

Spezifikationserzeugung. Unter Verwendung der funktionalen Datentypspezifikationen kann eine formale Spezifikation des Anforderungsmodells für den Anwendungsfall “Returning items” halbautomatisch erzeugt werden.

Das Geschäftsobjektmodell generiert die folgenden Maude-Klassen und -Nachrichten:

```

class Customer | rm : Oid .
class RM | current : Bag[Item], total : Bag[Item] .
msg return : Oid Bag[CItem] Oid -> Msg .
msg print : Oid Bag[Item] Euro Oid -> Msg .

```

Die Assoziation zwischen den UML-Klassen Customer und RM wird durch ein Maude-Attribut `rm` der Klasse Customer repräsentiert; die weiteren Assoziationen von RM durch die Maude-Attribute `current` und `total`. Die UML-Operationen `return` und `print` werden als Maude-Nachrichten modelliert, die zusätzliche Sender- und Empfängerparameter der Sorte `Oid` tragen; `Oid` repräsentiert Maude-Objektidentifikatoren.

Das Interaktionsmodell erzeugt unvollständige Ersetzungsregeln auf den Objekten des Geschäftsobjektmodells; die Ersetzungsregeln spiegeln die Struktur der Kollaboration direkt wider: kommt eine `return`-Nachricht bei der Leergutannahmemaschine an, so müssen ihre internen Attribute `current` und `total` (in einer noch nicht festgelegten, durch Fragezeichen markierten Weise) eventuell geändert und eine `print`-Nachricht gesendet werden; die Kollaboration legt kein Verhalten für den Empfang einer `print`-Nachricht fest. Die Nachrichtenabhängigkeiten des Interaktionsmodells definieren Nebenbedingungen für die Ersetzungsregeln:

```

rl [return] :
  return(P, CIb, O)
  < O : RM | current : Cb, total : Tb > =>
  < O : RM | current : ?, total : ? >
  print(O, Ib, S, O)
  where Ib = description(CIb) and S = amount(Ib) .

rl [print] :
  print(P, Ib, S, O)
  < O : RM | current : Cb, total : Tb > =>
  < O : RM | current : ?, total : ? > .

```

Zur Vervollständigung der Regeln müssen die Fragezeichen durch geeignete Werte ersetzt werden. Die Wertewahl wurde partiell durch die Invariante von RM eingeschränkt: $\text{current} \leq \text{total}$, d. h., die Multimenge current muß eine Teilmultimenge von total sein. Insbesondere muß, ersetzen wir das erste Fragezeichen durch einen Ausdruck X und das zweite durch Y die folgende Beweisverpflichtung erfüllt sein:

$$Ib = \text{description}(CIb) \text{ and } S = \text{amount}(Ib) \text{ and } Cb \leq Tb \text{ implies } X \leq Y$$

Eine mögliche Vervollständigung ist also $X = Ib$ und $Y = Tb$; die Invariante ist offensichtlich erfüllt. Allerdings wären auch $X = Cb$ und $Y = Tb$ oder $X = Cb$ und $Y = Tb$ möglich, wenn auch nicht erwünscht.

Ebenso erhält ein Ersetzen des ersten Fragezeichens der `print`-Regel mit Cb , des zweiten mit Tb die Invariante. Wir führen eine zusätzliche Ergebnismeldung ein, um die Ausgabe einer Spezifikationsausführung überprüfen zu können:

```
msg output : Bag[Item] Euro -> Msg .
rl [print] :
  print(P, Ib, S, O)
  < O : RM | current : Cb, total : Tb > =>
  < O : RM | current : Cb, total : Tb >
  output(Ib, S) .
```

Anfangskonfiguration. Initiale Konfigurationen stellen Testfälle für die vervollständigte Spezifikation bereit. Jede Anfangskonfiguration muß Maude-Objekte für alle an der Kollaboration beteiligten Klassen aufweisen; Interaktionen des Aktors mit dem System werden durch zusätzliche Nachrichten ausgedrückt.

Für die Leergutannahmemaschine ist folgende Testkonfiguration nützlich:

```
ops c r : -> Oid . op init : -> Configuration .
eq init = < c : Customer | rm : r >
  < r : RM | current : empty, total : empty >
  return(c, ci1 ... cin, r) .
```

3.2 Implementierung

Die fOOSE-unterstützte Java-Codeerzeugung schließt sich an die, hier nicht weiter ausgeführten, Analyse- und Entwurfsphasen an; sie erlaubt die Erzeugung korrekter nebenläufiger Java-Programme aus UML-Statistischen Strukturen und -Kollaborationen. Die Codeerzeugung setzt dabei voraus, daß das entwickelte UML-Modell vollständig ist, es müssen also alle Nebenbedingungen und alle unterspezifizierten Nachrichten aufgelöst sein. Wir demonstrieren die Codeerzeugung hier lediglich für das ergänzte Anforderungsanalyse-Kollaborationsdiagramm in Abb. 3, in dem wir, der Einfachheit halber, die asynchronen Nachrichten durch synchrone ersetzt haben.

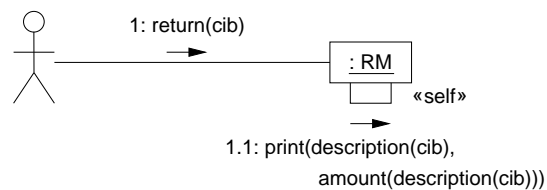


Abbildung 3: Vervollständigte Kollaboration für die Leergutannahmemaschine

Datentypen. Analog zur Bereitstellung der funktionalen Spezifikationen für die Datentypen des Anforderungsanalysemodells müssen alle in Java nicht vordefinierten Datentypen eingeführt werden. Auch hier kann auf bestehende Bibliotheksfunktionalitäten, etwa auf Multimengen, zurückgegriffen werden.

```

class Euro {
    int val;
    public static Euro
        add(Euro e1, Euro e2) {
            return new Euro(e1.val+e2.val);
        }
}
class Item {
    String name;
    Euro price;
    public Euro price() {
        return price;
    }
}
class CItem {
    Item description;
    public Item description() {
        return description;
    }
}
  
```

Codeerzeugung. Aufbauend auf diesen Datentypen folgt die Übersetzung demselben allgemeinen Schema wie die Maude-Spezifikationserzeugung. Allerdings müssen, da in Java Nebenläufigkeit nur durch Threads darstellbar ist, nebenläufige und auch asynchrone Nachrichten in eigene Threads eingepackt werden. Die Übersetzung synchroner Nachrichten ist dagegen, wegen der Semantik von Java-Methoden und der Blockstrukturierung, einfach:

```

class RM {
    public void mreturn(CItemBag cib) { // wegen Java-Schlüsselwort return
        print(description(cib), amount(description(cib)));
    }
}
  
```

4 Zusammenfassung

Im Gegensatz zu formalen Softwareentwicklungsansätzen, die informelle, diagrammatische Methoden formal ergänzen und Beweisverpflichtungen und Verfeinerungen einsetzen, um die Entwicklung beweisbar korrekter Software zu unterstützen, aber entweder die traditionellen, bildlichen Darstellungen als reine Illustrationen formaler Spezifikati-

on verwenden, oder, obwohl sie auf einer mathematisch wohlfundierten Semantik aufsetzen, während der Entwicklung jegliche Referenz auf Spezifikationen und formale Termini vermeiden; im Gegensatz dazu propagieren wir einen integrierten Ansatz, der bekannte Modellierungstechniken und formale Spezifikationen durch formal annotierte diagrammatische Modelle verbindet, diese erweiterten Modelle in Spezifikationen und Beweisverpflichtungen übersetzt und Eingriffe des Softwareentwicklers sowohl auf der diagrammatischen wie auch der Spezifikationsebene erlaubt.

Wir haben, in Anschluß an [Kna01], diese Prinzipien anhand einer formalen Erweiterung von Jacobsons OOSE-Methode, der fOOSE-Methode, erläutert und annotierte UML-Klassen- und Kollaborationsdiagramme in formale Maude-Spezifikationen und Java-Code übersetzt. Die fOOSE-Methode bedarf allerdings noch erheblicher technischer Ergänzungen, insbesondere die Einbeziehung zustandsorientierter Modellierungstechniken bleibt ein Desiderat.

Literaturverzeichnis

- [Ara96] Araújo, J.: *Methamorphosis: An Integrated Object-Oriented Requirements Analysis and Specification Method*. Dissertation, Lancaster University, 1996.
- [AS97] Achatz, K.; Schulte, W.: *A Formal OO Method Inspired by Fusion and Object-Z*. In Proc. 10th Int. Conf. Z Users (Bowen, J. P.; Hinchey, M. G.; Till, D., Hg.). Springer, Berlin, 1997, Bd. 1212 von Lect. Notes Comp. Sci., S. 92–111.
- [BRJ98] Booch, G.; Rumbaugh, J.; Jacobson, I.: *The Unified Modeling Language User Guide*. Addison–Wesley, Reading, Mass., &c., 1998.
- [CAB⁺94] Coleman, D.; Arnold, P.; Bodoff, S.; Dollin, C.; Gilchrist, H.; Hayes, F.; Jeremaes, P.: *Object-Oriented Development — The Fusion Method*. Prentice Hall, Englewood Cliffs, New Jersey, &c., 1994.
- [CD94] Cook, S.; Daniels, J.: *Designing Object Systems — Object-Oriented Modelling With Syntropy*. Prentice Hall, New York, &c., 1994.
- [CDE⁺99] Clavel, M.; Durán, F.; Eker, S.; Lincoln, P.; Martí-Oliet, N.; Meseguer, J.; Quesada, J.: *Maude: Specification and Programming in Rewriting Logic*. Manual, Computer Science Laboratory, SRI, 1999. <http://maude.csl.sri.com/manual>.
- [GJS96] Gosling, J.; Joy, B.; Steele, G.: *The Java Language Specification*. Addison–Wesley, Reading, Mass., 1996.
- [JCJÖ93] Jacobson, I.; Christerson, M.; Jonsson, P.; Övergaard, G.: *Object-Oriented Software Engineering — A Use Case Driven Approach*. Addison–Wesley, Wokingham, England, 4th Aufl., 1993.
- [Kem90] Kemmerer, R. A.: *Integrating Formal Methods into the Development Process*. In *IEEE Software*, Bd. 7 (5):(1990), S. 37–50.
- [Kna01] Knapp, A.: *A Formal Approach to Object-Oriented Software Engineering*. Dissertation, Ludwig-Maximilians-Universität München, 2001.

- [Lan95] Lano, K.: Formal Object-Oriented Development. Formal Approaches to Computing and Information Technology. Springer, London, 1995.
- [NF97] Nakajima, S.; Futatsugi, K.: An Object-Oriented Modeling Method for Algebraic Specifications in CafeOBJ. In Proc. 19th Int. Conf. Softw. Eng. (Adrion, W. R., Hg.), 1997, S. 34–44.

Alexander Knapp wurde 1971 in Wien, Österreich geboren. Von 1990 bis 1995 studierte er Informatik mit dem Nebenfach Mathematik an der Universität Passau. Seit 1995 ist er wissenschaftlicher Mitarbeiter am Institut für Informatik der Ludwig-Maximilians-Universität München. Während dieser Zeit verbrachte er einen halbjährigen Forschungsaufenthalt am SRI International, Kalifornien. Im Juli 2000 promovierte er mit einer Arbeit über integrierte formale Softwareentwicklungsmethoden.