Seminar

Graph-Grammatiken

Lehrstuhl für Informatik III
RWTH Aachen
Sommersemester 2004

Ein Ansatz zur Systemmodellierung mit Offenen Graphtransformationssystemen

Mark Lehmacher

Inhaltsverzeichnis

1 Motivation

Spezifikation und Modellierung von Systemen und Beziehungen zwischen Elementen eben dieser Systeme sind ein komplexes Thema. Dank umfangreicher Forschungen auf diesem Gebiet gibt es viele Ansätze zur Modellierung komplizierter Zusammenhänge, welche in ihren jeweiligen Ausführungen für spezielle Problemfälle mehr oder weniger geeignet sind.

Konkret eingegangen werden soll im Folgenden auf die sich in der Softwareentwicklung von großen Projekten stellenden Probleme und Ansätze zur Bewältigung dieser Schwierigkeiten. Vorgestellt wird dafür ein Sicht-basierter Ansatz der Systemmodellierung, basierend auf Offenen Graphtransformationssystemen.

1.1 Probleme der Softwareentwicklung

Das Hauptproblem der Softwareentwicklung ist sicherlich die steigende Komplexität der in Software umzusetzenden Anforderungen und der zu modellierenden Real-World-Systeme. Ziel muss es deswegen sein, effiziente Mittel der Komplexitätsreduktion zu finden.

Eine bereits etablierte Methode ist die Wiederverwendung von bereits bekannten und vorhandenen Fragmenten eines Softwareprojekts, wie sie etwa aus einem vorherigen Projekt vorhanden sein mögen. Spezifikation, Dokumentation und Quellcode eignen sich hierfür besonders. Gerade beim Quellcode von Software ist es mittlerweile gebräuchlich, in-sich- abgeschlossene Funktionalitäten zu kapseln. In den Anfangstagen der Softwareentwicklung war dies noch beschränkt auf einzelne Funktionen und Codefragmente, aber dies hat sich mit der Zeit von Klassen, über Komponenten, auf komplette Frameworks ausgeweitet. Patterns, anders als Klassen und Komponenten, sind generelle Verhaltensmuster und Ideen für bestimmte, häufig auftretende Problemklassen und damit gleichzeitig eine andere Möglichkeit, die Komplexität eines Projektes zu reduzieren. Viele der hier vorgestellten Wiederverwendungsideen setzen erst spät im Softwareentwicklungsprozess an (z.B. Codewiederverwendung auf Implementierungsebene). Wünschenswert wäre daher ein Ansatz, um schon auf Projektspezifikationsebene der Komplexität umfangreicher Systeme Herr werden zu können.

Dafür sieht ein möglicher Modularisierungsansatz vor, Teilaspekte eines Systems von verschiedenen Teilgruppen des Projektstabes separat und parallel bearbeiten zu lassen. Als Ergebnis einer solcherart ausgeführten Spezifikationsphase erhält man so z.B. verschiedene Teilspezifikationen (auch *Designsichten* genannt), die dann zu einem konsistenten Ganzen zusammengefügt werden müssen, dem so genannten *Systemmodell*. Diese Konstruktion des Systemmodells aus möglicherweise überlappenden Designsichten kann durch *Sichtintegration* geschehen.

1.2 Relevanz für Datenbanksysteme

Sichten spielen nicht nur bei der Softwareentwicklung eine große Rolle, sondern sind auch in der Welt der Datenbanken wieder zu finden. Üblicherweise unterscheidet man bei Datenbanken zwischen einem konzeptuellen Modell (das vollständige Schema einer Datenbank) und eingeschränkten *Benutzersichten*. Verschiedene Benutzer sehen bei Letzterem immer nur den für sie relevanten Ausschnitt des konzeptuellen Modells.

Grundsätzlich gibt es also zwei Arten von Sichten. Zum einen Benutzersichten, die Einschränkungen eines bereits vorhanden und definierten Datenbank Schemas darstellen und zum anderen Designsichten, über deren (oftmals mühselige) Integration

ein Datenbank Schema, welches allen integrierten Sichten genügt, konstruiert wird. Nach abgeschlossener erfolgreicher Integration werden Designsichten dann zu Benutzersichten, weil sie nicht mehr nur Mittel zum Zweck eines konzeptuellen Modells, sondern eben Teilaspekte des durch Integration gewonnenen Schemas sind.

2 Begriffsbildung

Nachdem in der Einleitung bereits das Konzept der Sichten angerissen wurde, sollen hier einige neue Begriffe eingeführt und bereits erwähnte konkretisiert werden. Im Folgenden werden die Begriffe Sicht und Spezifikation etwas verschwimmen, aber da jede Sicht genau eine Spezifikation bestimmt, mag uns das nicht weiter stören.

2.1 Sicht

Eine Sicht ist die unvollständige Spezifikation eines Systems, basierend auf einem Ausschnitt dieses Systems. Naturgemäß kann eine einzelne Sicht den Zustand eines Systems nur teilweise beschreiben, ebenso wie Effekte von Operationen der Sicht das komplette System möglicherweise nur teilweise beeinflussen. So kann es durchaus sein, dass auf Systemebene einzelne Operationen einer Sicht erst gekoppelt mit den Operationen anderer Sichten zu einem sinnvollen und konsistenten Systemzustand führen. Folgerichtig definieren Sichten für ein System also nur das, was in einem bestimmten Systemzustand mindestens passieren muss.

2.2 Referenzmodell

Das Referenzmodell ist der formale Ausgangspunkt aller Sichten. Im Referenzmodell werden die grundlegenden Objekte, Eigenschaften und Beziehungen zwischen Gegenständen des zu modellierenden Systems festgelegt. Somit bestimmt das Referenzmodell das Vokabular für die Designer der Sichten und erleichtert durch Schaffung einer gemeinsamen Basis eine spätere Integration verschiedener Sichten. Wohlgemerkt werden im Referenzmodell nur die grundlegenden Begriffe geschaffen, für Konkretisierungen und Erweiterungen sind spezielle Designsichten zuständig.

2.3 Systemmodell

Ziel des Designprozesses und der Sichtintegration ist die Spezifikation des finalen Systemmodells. Das Systemmodell ist Ergebnis der Integration der vom Referenzmodell abgeleiteten Sichten und stellt somit eine Erweiterung des Referenzmodells um die in den Sichten eingeführten Elemente sowie deren Beziehungen untereinander dar.

3 Ablauf eines Sicht-basierten Entwicklungsprozesses

Ausgehend von dem formal zu spezifizierenden System wird das Referenzmodell aufgebaut, indem dort globale Begrifflichkeiten und Operationen registriert werden. Vom Referenzmodell aus konstruiert jede Gruppe von Entwicklern Sichten, welche einen Ausschnitt des Gesamtsystems beschreiben und konkretisieren. Operationen könnten z.B. hinzugefügt oder erweitert werden. Oftmals sind diese Designsichten dadurch gekennzeichnet, dass sie Systemausschnitte aus der Sicht verschiedener Partizipanten dieses Systems darstellen. Es werden also Rollen im System identifiziert und als Ausgangspunkt zur Entwicklung von speziellen Sichten genutzt. Schließlich werden die Designsichten zu einem Systemmodell integriert. Um die Integration der Sichten kümmert sich ein Modell-Manager, ein ausgesuchter Entwickler, der für die Konsistenz zwischen Sichten verantwortlich zeichnet. Seine Aufgabe wird dadurch vereinfacht, dass globale Begriffe und Operationen bereits durch das Referenzmodell

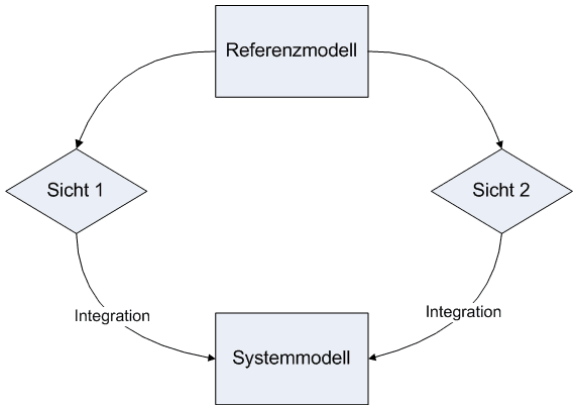

Abbildung 1: Ablauf des Designprozesses mit zwei Sichten

vorgegeben sind und er daher in den einzelnen Sichten in der Regel auf problemspezifische Begriffe, die nicht notwendigerweise für andere Sichten relevant sind, stoßen wird. In diesem Falle werden diese Begriffe oder Operationen in das Referenzmodell aufgenommen, um so schließlich, nach Integration aller Sichten, das Systemmodell zu bilden. Im Falle von Abhängigkeiten von Bezeichnern oder Operationen zwischen Sichten, müssen Konflikte und Inkonsistenzen beigelegt werden. Beispielhaft betrachten wir hier zwei Fälle, welche die Behandlung von Namen betreffen.

(i) Das selbe Konzept (etwa eine Operation) wird in zwei Sichten mit zwei unterschiedlichen Namen bezeichnet.
(ii) Die selben Namen werden in zwei Sichten für semantisch verschiedene Konzepte benutzt.

Im ersten Fall (i) ist es Aufgabe des Modell-Managers, sich für einen der beiden Namen zu entscheiden und das definierte semantische Konzept unter diesem Namen in das Referenzmodell aufzunehmen. Für (ii) sind zwei Lösungen vorstellbar: Zwei Namen werden in das Referenzmodell übernommen (jeweils der Name qualifiziert durch den Bezeichner der jeweiligen Sicht, also etwa SichtA.Name oder SichtB.Name), oder der Modell-Manager lässt das Referenzmodell unverändert.

In Abbildung 1 wird der Ablauf des Designprozesses im Falle von zwei vom Referenzmodell abgeleiteten Sichten skizziert. In der späteren konkreten Ausführung der Integration der Sichten zum Systemmodell wird auch speziell die Integration von genau zwei Sichten betrachtet werden.

Illustriert wird dieser Entwicklungsprozess im Folgenden anhand des Beispiels eines Banksystems. Das Referenzmodell wird anfangs die grundlegenden Begriffe der Bank, wie etwa Konto und Transaktion beinhalten und die typischen Zusammenhänge zwischen diesen Begriffen durch Operationen modellieren. Zwei Designsichten werden entwickelt, eine, die das System aus der Sicht eines Kunden darstellt, die andere aus der Sicht eines Bankangestellten. Während der Spezifikation der beiden Sichten wird das Referenzmodell laufend durch den Modell-Manager angepasst und erweitert. Beispielhafte Operationen wie die Eröffnung eines Kontos oder die Tätigung einer Überweisung werden betrachtet.

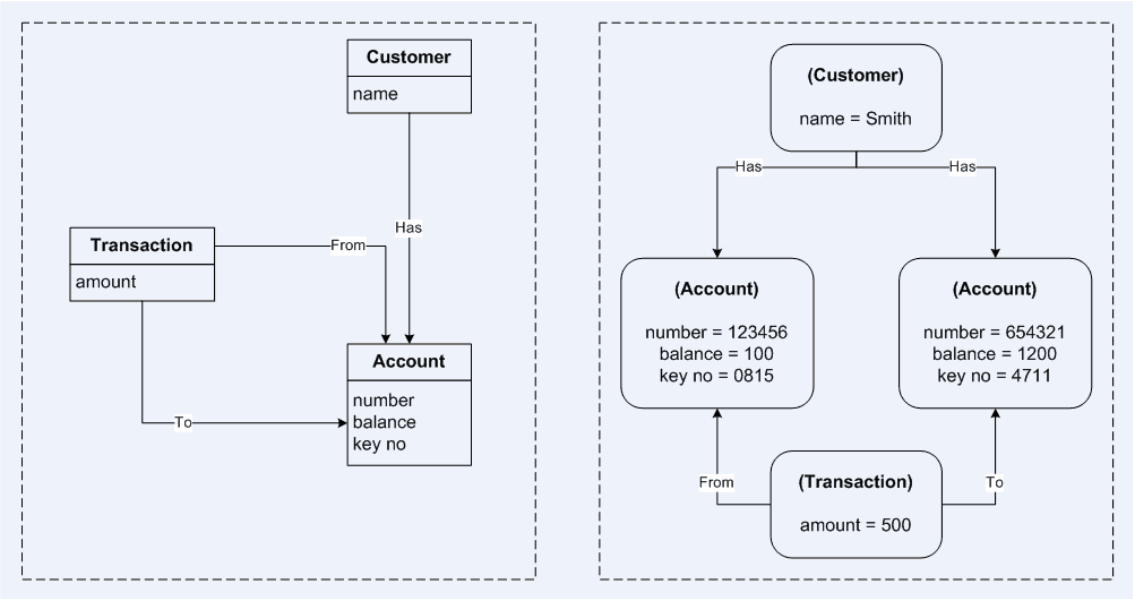

Abbildung 2: Beispiel eines Typgraphen mit einem dazugehören Instanzgraphen

4 Graphentheoretische Hilfsmittel

Um den Systemmodellierungsprozess formal darstellen zu können, müssen erst entsprechende Grundlagen geschaffen werden.

4.1 Graphtransformationssysteme

Ein einfacher Graph ist definiert durch seine Knotenmenge V und seine Kantenmenge E. Außerdem muss jede Kante e aus E eine Startknoten $s(e)$ aus V, und Zielknoten $t(e)$ in V haben. Graphen mit Beschriftungen (Attributnamen) an Knoten oder Kanten und möglichen, mit den Beschriftungen assoziierten (Attribut-)Werten, heißen *attributierte Graphen*.

Weiterhin unterscheiden wir zwischen *Typgraphen* und *Instanzgraphen*. Analog zum Verhältnis zwischen einer Klasse und Objekten (Instanzen) dieser Klasse in der objektorientierten Programmierung, ist ein Typgraph eine Schablone für einen Instanzgraphen. Typgraphen geben *Typen* mit möglichen asoziierten Attributnamen vor und Instanzgraphen instanziieren Attribute von einzelnen Typen dann mit konkreten Werten. Im Typgraphen werden Typen fürn Knoten und Kanten defniert, wobei Knoten *Objekte* und die Kanten die *Beziehungen* zwischen diesen Objekten repräsentieren.

Ein Typgraph inklusive einem zugehörigem Instanzgraphen wird in Abbildung 2 dargestellt. Der Typgraph gibt die Objekt- und Beziehungstypen vor. Objekttypen sind hier das Konto, der Kunde und die Transaktion, während Beziehungen beispielsweise in der Form einer *Hat* Verbindung von Kunde zu Konto auftreten. Mit den Objekten sind verschiedene Attribute verknüpft, so hat der Kunde einen Namen und das Konto besitzt die Attribute Nummer, Stand und Geheimzahl. Der dargestellte Instanzgraph repräsentiert einen Systemzustand, in dem der Kunde Smith zwei Konten besitzt und gerade eine Transaktion zwischen diesen Konten durchgeführt wird.

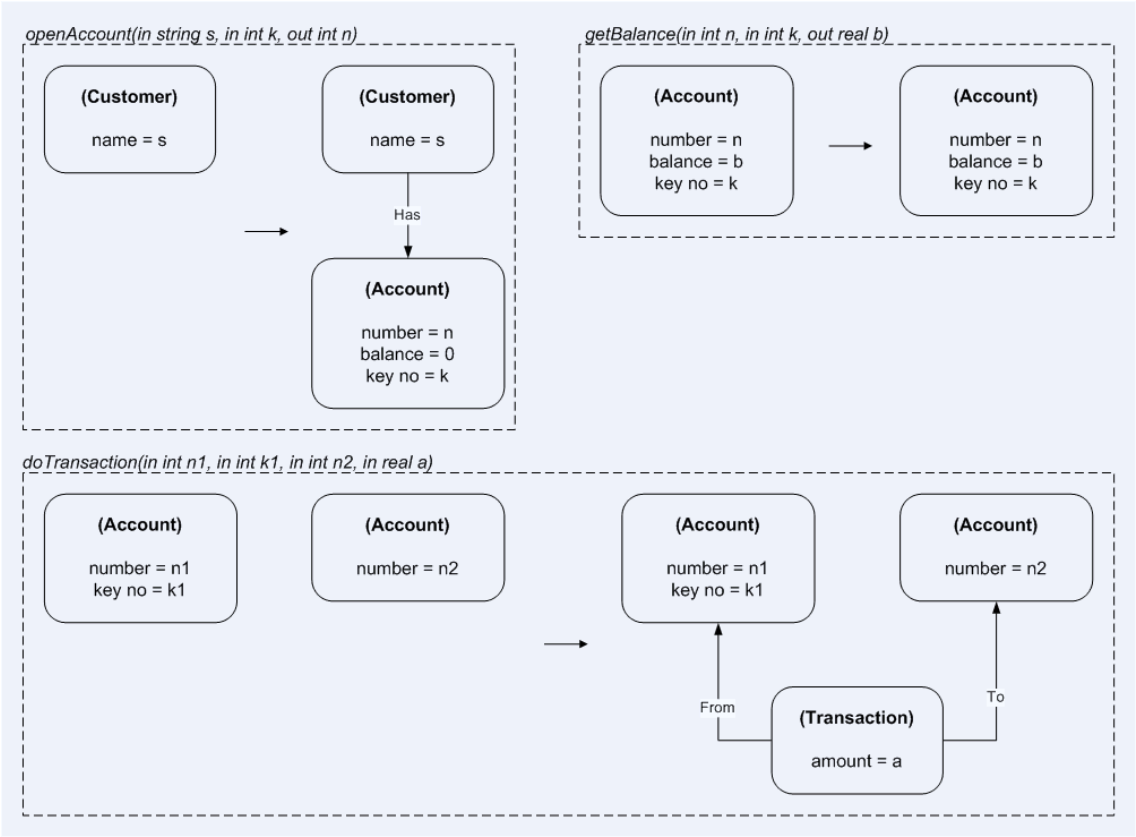

Abbildung 3: Graphtransformationsregeln zum Eröffnen eines Kontos, Abfragen des Kontostandes und Durchführen einer Transaktion (aus der Sicht eines Bankkunden)

4.2 Regelbasiertes Graphtransformationssystem

Zustandsänderungen bei Graphen werden modelliert durch Transformationen, welche durch Transformationsregeln (Produktionen) $p : L \rightarrow R$ spezifiziert werden. Eine Regel besteht aus einem Namen p und zwei Instanzgraphen L und R, welche jeweils einen Ausschnitt eines Systemzustands vor und nach der Anwendung der Regel repräsentieren. Wir nennen L die Vorbedingung und R die Nachbedingung der Produktion. Den Schnitt $L \cap R$ bezeichnen wir als das Interface von p. Er beinhaltet Elemente, auf die zwar lesend zugegriffen wird, welche aber nicht gelöscht werden (so gesehen der Kontext einer Regel).

Abbildung 3 und 4 sind graphische Darstellungen von Produktionen. Die Produktion links oben in Abbildung 3 beschreibt die Eröffnung eines neuen Kontos. Dabei erhält ein Kunde durch Eingabe von Name und Geheimzahl ein neues leeres Konto, ausgedrückt durch eine *Hat* Beziehung zwischen Kunde und Konto. Ausgabe dieses Vorgangs ist die Kontonummer des angelegten Kontos. Die Regel *getBalance* liest den Kontostand eines Kontos (modelliert durch eine identische Ersetzung), während *doTransaction* eine Transaktion anstößt, welche allerdings erst später von einem Angestellten wirklich ausgeführt wird, siehe die dazugehörige Regel *doTransaction* in Abbildung 4.

Ein *Ableitungsschritt* $G \overset{p}{\Longrightarrow} H$ von G nach H über p mit $p : L \rightarrow R$ setzt voraus, dass L (oder ein zu L isomorpher Graph) Teilgraph von G ist. $L \setminus R$ wird aus G

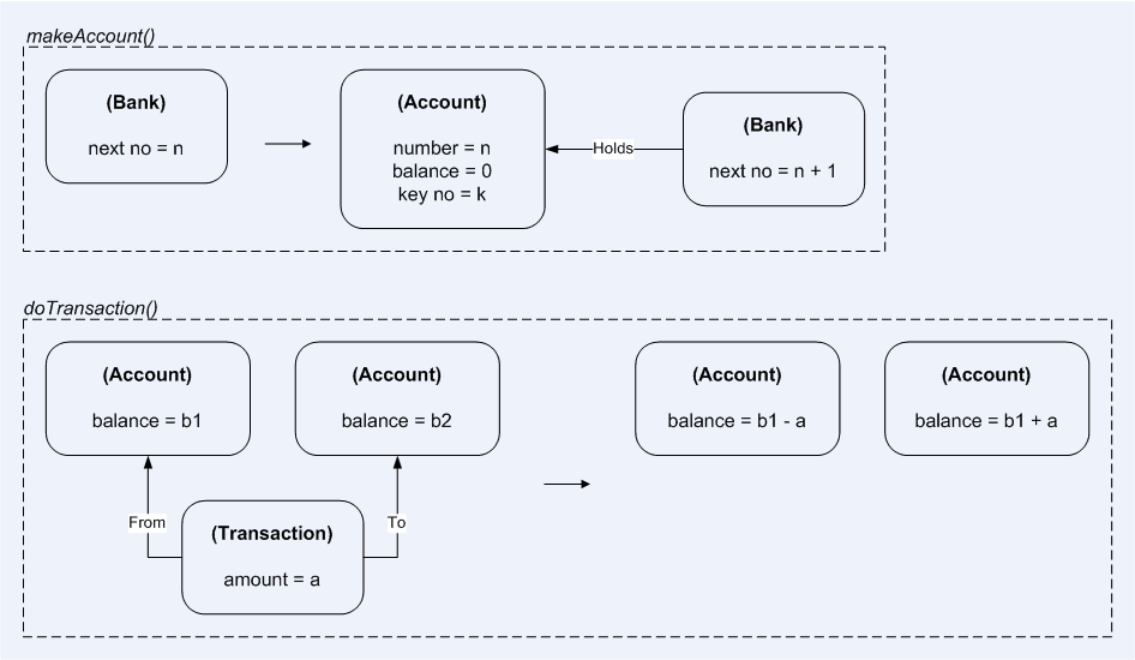

Abbildung 4: Graphtransformationsregeln zum Anlegen eines Kontos und Durchführen einer Transaktion (aus der Sicht eines Bankangestellten)

entfernt und dafür an seiner Position $R \setminus L$ eingefügt. Wir erhalten dann den Graphen H, welcher einen zu R isomorphen Teilgraphen beinhaltet. Den Graphen $L \setminus R$ bezeichnen wir mit $delete(p)$ und analog $R \setminus L$ mit $add(p)$. Eine Produktion p darf nur angewendet werden, wenn nach Anwendung von p wieder ein gültiger Graph entsteht. Gemeinhin impliziert man eine Rahmenbedingung, welche besagt, dass eine Produktion p genau den Graphen $delete(p)$ löschen und $add(p)$ hinzufügen darf. Ohne diese Bedingung würde eine Regel p mindestens $delete(p)$ löschen und mindestens $add(p)$ hinzufügen dürfen, also beispielsweise insbesondere auch lose Kanten entfernen dürfen, ohne dass dies in der Regel explizit angegeben wurde.

Für die Modellierung von Systemen durch Graphen sind uns allerdings beide dieser Bedingungen zu streng. Daher benötigen wir eine Bedingung, die sowohl unspezifizierte Effekte zulässt, es uns aber auch ermöglicht, gewisse Typen vor dieser losen Semantik zu schützen. Das geschieht durch Angabe einer so genannten *expliziten Rahmenbedingung* für Typen, die sich sowohl negativ als auch positiv ausdrücken lässt. Drückt man sie positiv aus, gibt man eine explizite Liste aller Typen eines Graphen an, welche offen sind gegenüber losen Effekten, während die negative Spezifikation alle geschützten Typen aufzählt. Beide Arten diese Beschreibung sind äquivalent, da ein Typ, der nicht offen ist, automatisch geschützt ist und umgekehrt. Wir werden uns hier auf die positive Spezifikation beschränken, welche wiederum aufgeteilt ist in eine Menge der zum Löschen offenen Typen und derer, die offen zum Hinzufügen sind. Formal sieht eine (positiv formulierte) explizite Rahmenbedingung dann wie folgt aus: $O = \langle O^-, O^+ \rangle$ wobei O^- die Menge der zum Löschen offenen Typen ist und analog O^+ die Menge der zum Hinzufügen offenen.

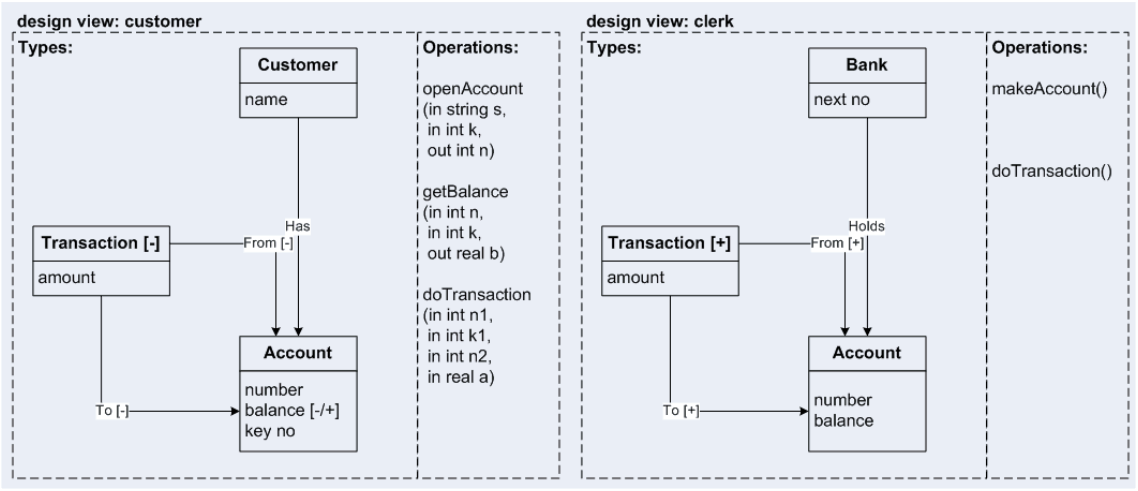

Abbildung 5: Zwei Designsichten, die eine aus der Sicht eines Bankkunden, die andere aus Sicht eines Bankangestellten

4.3 Offene Graphtransformationssysteme

Wir sind nun in der Lage, ein Offenes Graphtransformationssystem (*OGTS*) formal zu definieren: Ein OGTS hat die Form $\mathcal{O} = \langle TG, P, \pi, O \rangle$, wobei TG ein Typgraph, P eine Liste von Produktionsnamen und π eine Zuordnung von Produktionsnamen auf eine konkrete Regel $L \to R$ und L, R Instanzgraphen von TG sind. O ist eine explizite Rahmenbedingung. Regeln können sequentiell oder parallel ausgeführt werden, für $p_1 \cdots p_n$ und $n \geq 0$ hat die parallele Ausführung die Form $p_1 + \cdots + p_n : L_1 + \cdots + L_n \to R_1 + \cdots + R_n$ als paarweise disjunkte Vereinigung der gegebenen Regeln. Zwei Vorkommen von L_j und L_i mit $i \neq j$, dürfen sich im gegebenen Graphen überlappen, allerdings nur an Stellen die nicht gelöscht werden. Als Spezialfall der parallelen Ausführung für $n = 0$ erhalten wir durch Anwendung die leere Regel $\epsilon : \emptyset \to \emptyset$, welche benutzt werden kann, um eine leere Transition darzustellen, also eine Zustandsänderung im Graphen, die ausschließlich durch externe Einflüsse verursacht wird (z.B. das Buchen einer Transaktion durch den Bankangestellten, was aus der Sicht des Kunden einen externen Einfluss darstellt). Offensichtlich ist $add(\epsilon) = delete(\epsilon) = \emptyset$.

Die beiden Graphen in Abbildung 5 repräsentieren zwei Designsichten: Zum einen das System der Bank aus der Sicht eines Kunden und zum anderen aus der Sicht eines Angestellten. Sie sind nichts anderes als Typgraphen, erweitert um die Definition der offenen Typen. Diese werden in eckigen Klammen an den entsprechenden Typ- und Attributnamen durch $-$ und $+$ Symbole für geschützte, respektive offene Typen angegeben. Attribute dürfen zusammen mit ihren Trägerobjekten angelegt und gelöscht werden, eine Löscherlaubnis wie z.B. beim *Transaction* Knoten der Kundensicht impliziert, dass man auch die zugehörigen Attribute löschen darf, also hier konkret *amount*. Anders verhält es sich, wenn man ausschließlich die Attribute eines Knotens ändern möchte (durch Löschen und Hinzufügen des betroffenen Attributes), wie etwa beim *balance* Attribut beim *Account* Knoten in der Kundensicht.

Abbildung 6 illustriert eine Sequenz von Produktionen der Kundensicht und ihre Auswirkungen auf Zustände des zugrunde liegenden repräsentierenden Graphen.

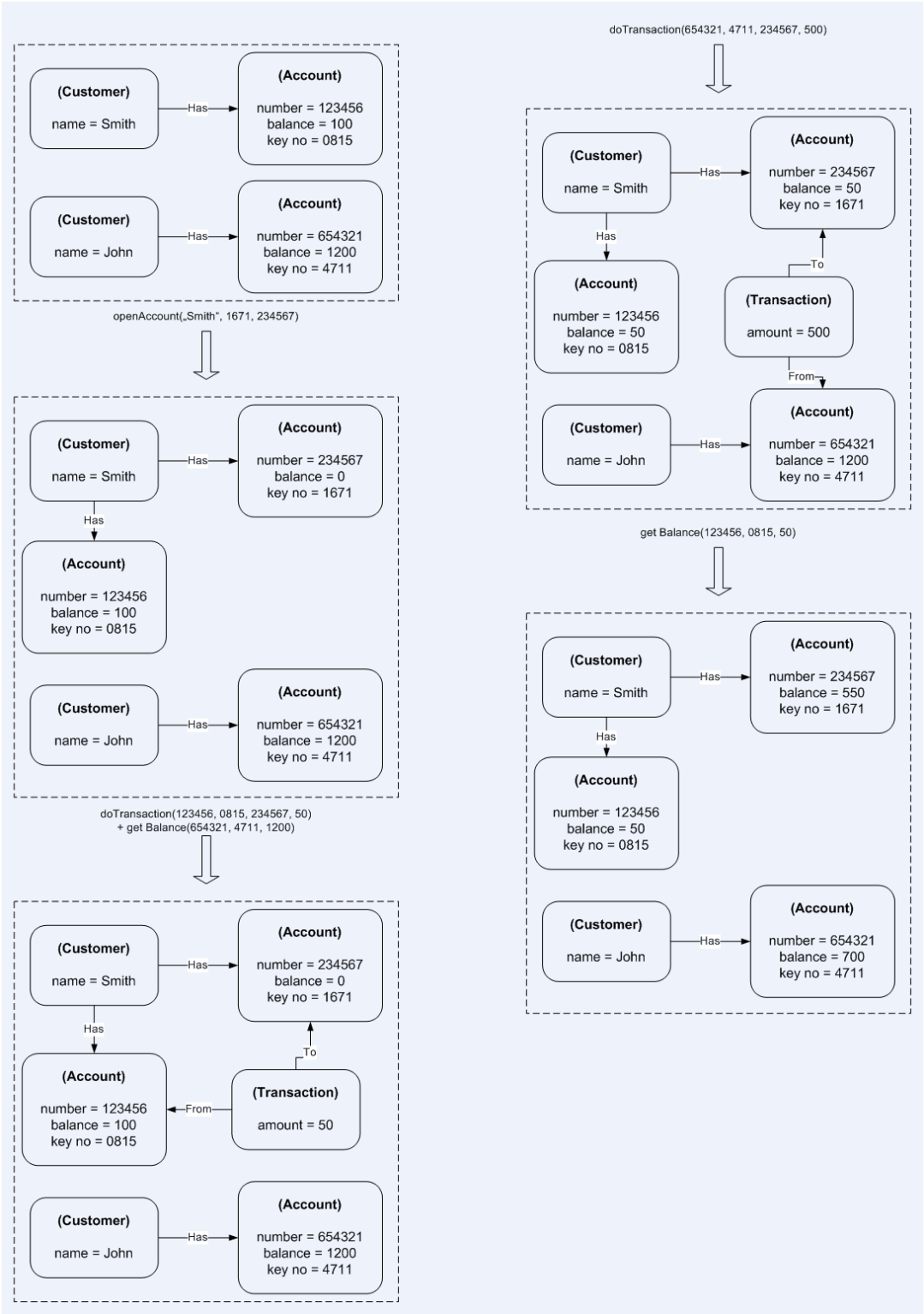

Abbildung 6: Beispiel für eine Transformationssequenz (aus der Sicht eines Bankkunden)

Konkret dargestellt werden die Eröffnung eines Kontos und die Ausführung verschiedener Überweisungen. Wohlgemerkt werden die Überweisungen durch einen Bankangestellten durchgeführt, was in dieser Sequenz zur Folge hat, dass die Transaktionsknoten von einem Ableitungsschritt zum nächsten gelöscht werden und sich die Kontostände entsprechend ändern. Dies sind genau die unspezifizierten Nebeneffekte, die wir durch unsere Rahmenbedingung zulassen.

5 Sichten in Form von OGTS und Beziehungen zwischen Sichten

Wir verwenden OGTS, um Sichten darzustellen. Die Knoten sind dabei Objekte des Systems und die Kanten die Beziehungen zwischen diesen Objekten. Operationen des Systems werden durch die Produktionen des OGTS dargestellt. Das Referenzmodell und das Systemmodell sind formal nichts anderes als Sichten und werden deswegen ebenfalls durch OGTS modelliert.

Aufgrund der Tatsache, dass Designsichten das Referenzmodell erweitern, liegt es nahe, diesen Zusammenhang durch eine *Erweiterungsrelation* zu formalisieren. Der Frage, wann sich zwei Sichten entsprechen, gehen wir nach, indem wir eine *Umbenennungsrelation* einführen. Schließlich werden wir aus Komposition von Erweiterungs- und Umbenennungsrelation die *Sichtrelation* erhalten, welche bestimmt, ob ein OGTS eine Sicht auf ein anderes OGTS darstellt.

5.1 Umbenennungsrelation

Die Umbenennungsrelation erlaubt die Benutzung verschiedener Namen für semantisch identische Elemente in zwei verschiedenen Sichten. Zwei OGTS stehen über eine Umbenennung in Relation genau dann zueinander ($\mathcal{O} \leftrightarrow \mathcal{O}'$), wenn für jeden Typen, jede Regel und jeden Regelnamen von $\mathcal{O}$ ein korrespondierender Element in $\mathcal{O}'$ existiert. Anders formuliert: $\mathcal{O} \leftrightarrow \mathcal{O}'$ genau dann, wenn ein Isomorphismus von $\mathcal{O}$ nach $\mathcal{O}'$ existiert. Für ein konkretes Element x (Typ, Regelbezeichner, etc.) von $\mathcal{O}$, schreiben wir dann $x \overset{ren}{\leftrightarrow} x'$, wenn es ein entsprechendes Element in $\mathcal{O}'$ gibt. Repräsentiert werden die Korrespondenzen von Elementen der OGTS durch ein Verzeichnis $\mathcal{O} \overset{ren}{\leftrightarrow} \mathcal{O}'$, in welchem alle Einträge $x \overset{ren}{\leftrightarrow} x'$ (bzw. $x \overset{ren}{\leftrightarrow} x$ für ungeänderte Namen aus $\mathcal{O}$) eingetragen werden. Falls $\mathcal{O}'$ Elemente enthält, die keine Entsprechung in $\mathcal{O}$ haben, werden diese nicht in das Verzeichnis ren aufgenommen.

5.2 Erweiterungsrelation

Die Extension einer Produktion p_0 durch eine andere Produktion p_1 wird modelliert durch die Teilregelrelation. $p_0 : L_0 \to R_0$ ist eine Teilregel von $p_1 : L_1 \to R_1$, geschrieben $p_0 \subseteq p_1$, wenn die Folgen der Anwendung von p_1, die der Anwendung von p_0 erweitern. Das ist genau dann der Fall, wenn $L_0 \subseteq L_1, R_0 \subseteq R_1$ (die Vor- und Nachbedingungen werden erweitert) und $delete(p_0) \subseteq delete(p_1)$, sowie $add(p_0) \subseteq add(p_1)$.

Bezogen auf OGTS folgt, das ein OGTS $\mathcal{O}_1$ ein anderes $\mathcal{O}_0$ erweitert, wenn sowohl Typgraph als auch Regelbezeichner von $\mathcal{O}_0$ in $\mathcal{O}_1$ erweitert werden. In diesem Fall schreiben wir $TG_0 \subseteq TG_1$ und $P_0 \subseteq P_1$. Außerdem muss für jeden Regelbezeichner $p \in P_0$ gelten, dass die mit p assoziierte Produktion aus $\mathcal{O}_0$ eine Unterregel der mit p assoziierten Produktion aus $\mathcal{O}_1$ ist, formal $\pi_0(p) \subseteq \pi_1(p)$. Bezüglich der expliziten Rahmenbedingung muss für die offenen Typen $O_0 = \langle O_0^-, O_0^+ \rangle$ von $\mathcal{O}_0$ gelten:

- für neue Regeln: Für jede neue Regel $p_1 \in P_1$ müssen die Knoten und Kanten von TG_0, welche durch Anwendung von p_1 gelöscht werden, formal $delete(p_1)|_{TG_0}$

$(add(p_1)|_{TG_0})$ Instanzen von O_0^- (O_0^+) sein. Neue Regeln von $\mathcal{O}_1$ sind nicht sichtbar in $\mathcal{O}_0$, d.h. sie gehören nicht zu dieser Sicht.

- Erweiterungen von Regeln: Für jede Regel $p_1 \in P_1$ und ihre Unterregel in $p_0 \in P_0$ müssen die Knoten und Kanten von $delete(p_1)|_{TG_1} \setminus delete(p_0)$ $(add(p_1)|_{TG_1} \setminus add(p_0)))$ Instanzen von O_0^- (O_0^+) sein.
- offene Typen: Keine Typen von TG_0, die in $\mathcal{O}_0$ geschützt sind, werden in $\mathcal{O}_1$ offen deklariert, also $O_1^- \cap TG_0 \subseteq O_0^-$ und $O_1^+ \cap TG_0 \subseteq O_0^+$. Neue Typen von $\mathcal{O}_1$ können beliebig instanziiert und gelöscht werden.

5.3 Sichtrelation

Die Sichtrelation ist eine Komposition der Umbenennungs- und Erweiterungsrelation. Eine Sichtrelation $v = (\mathcal{O}_0 \overset{ren}{\hookleftarrow} \mathcal{O}_0^1 \subseteq \mathcal{O}_1)$ zwischen $\mathcal{O}_0$ und $\mathcal{O}_1$ ist eine Umbenennung von $\mathcal{O}_0$ zu $\mathcal{O}_0^1$, so dass $\mathcal{O}_1$ eine Erweiterung von $\mathcal{O}_0^1$ ist. Abstrakter kann man auch $v : \mathcal{O}_0 \to \mathcal{O}_1$ schreiben und sagt, dass $\mathcal{O}_0$ eine Sicht auf $\mathcal{O}_1$ ist. Die Sichtrelation ist transitiv, d.h. wenn $\mathcal{O}_0 \to \mathcal{O}_1$ und $\mathcal{O}_1 \to \mathcal{O}_2$, dann gilt auch $\mathcal{O}_0 \to \mathcal{O}_2$. Eine Sichtrelation $v : \mathcal{O}_0 \to \mathcal{O}_1$ beschreibt eine Projektion eines Zustandsgraphen von $\mathcal{O}_1$ auf $\mathcal{O}_0$ und damit eine abstraktere Darstellung des Verhaltens von $\mathcal{O}_1$.

6 Integration zweier Sichten

Mit Hilfe der Sichtrelation sind wir in der Lage, Beziehungen zwischen Referenzmodell, Systemmodell und einzelnen Sichten zu beschreiben. Folgerichtig wenden wir uns nun der Integration von Sichten zu.

6.1 Idee

Ausgehend vom Referenzmodell werden individuelle Designsichten entwickelt, welche dann später zum finalen Systemmodell integriert werden. Dies geschieht, indem das Referenzmodell um die neu hinzugekommenen Elemente der Sichten erweitert wird. Ein Problem stellt sich dabei in der Frage, ob zwei Elemente in zwei verschiedenen Sichten dieselben Typen oder Operationen repräsentieren. Der Name einer Operation oder eines Typs macht keine verbindliche Aussage über die semantische Funktion des bezeichneten Elements. Aufgrund dessen ist ein Name alleine nicht geeignet, um festzustellen ob zwei Elemente dasselbe Konzept repräsentieren. Deswegen zieht man das Referenzmodell zu Rate und sagt, dass zwei Elemente, die dort denselben Ursprung haben, miteinander korrespondieren. Die Elemente, die keinen Ursprung im Referenzmodell und auch keine Entsprechung in einer anderen Sicht haben, können problemlos dem Referenzmodell hinzugefügt werden.

6.2 Formalismus

Im Folgenden betrachten wir die Designsichten $\mathcal{O}_1$, $\mathcal{O}_2$ auf das Referenzmodell $\mathcal{O}_0$ und die Sichtrelationen $v_i : \mathcal{O}_0 \to \mathcal{O}_i$ gegeben durch $(\mathcal{O}_0 \overset{ren_i}{\hookleftarrow} \mathcal{O}_0^i \subseteq \mathcal{O}_i)$ für $i = 1, 2$.

6.3 Definiertheit der Sichtintegration

Als $affected^-(v_i)$ $(affected^+(v_i))$ bezeichnen wir die Menge der Typen, d.h. Knoten oder Kanten von TG_0, dessen Instanzen von einer neuen Produktion in $\mathcal{O}_i$, oder der Erweiterung einer $\mathcal{O}_0$-Regel in $\mathcal{O}_i$ entfernt oder hinzugefügt werden. Offensichtlich gilt $affected^-(v_i) \subseteq O_0^-$ und $affected^+(v_i) \subseteq O_0^+$.

Die Konstruktion einer Sichtintegration ist definiert, wenn die offenen Typen des Referenzmodells, welche betroffen werden von einer neuen oder erweiterterten Produktion einer der Designsichten, auch in der anderen Sicht offen bleiben:

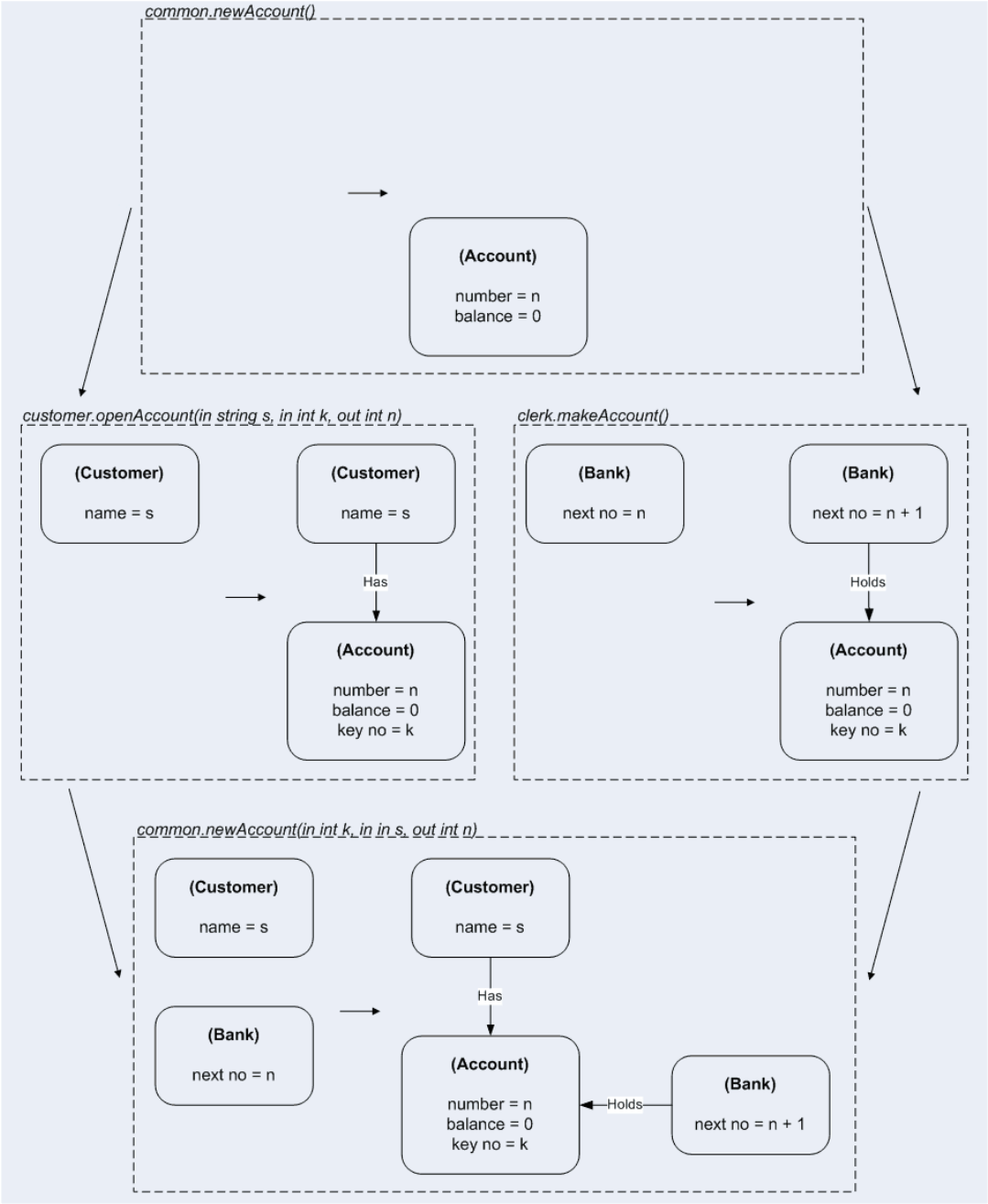

Abbildung 7: Integration der Produktionen *customer.openAccount* und *clerk.makeAccount* zu *common.newAccount*

$$ren_2(affected^-(v_1)) \subseteq O_2^- \quad \text{und} \quad ren_1(affected^-(v_2)) \subseteq O_1^-$$

und analog für O_1^+ und O_2^+.

Wenn diese Bedingung gilt, kann die Integration von $\mathcal{O}_1$ und $\mathcal{O}_2$ über $\mathcal{O}_0$ in zwei Schritten erfolgen.

1. Umbenennung von $\mathcal{O}_1$ und $\mathcal{O}_2$ (Ausgangspunkt ist die Annahme, dass die Bezeichner von $\mathcal{O}_1$ und $\mathcal{O}_2$ disjunkt sind; garantiert werden kann dies, indem jedem

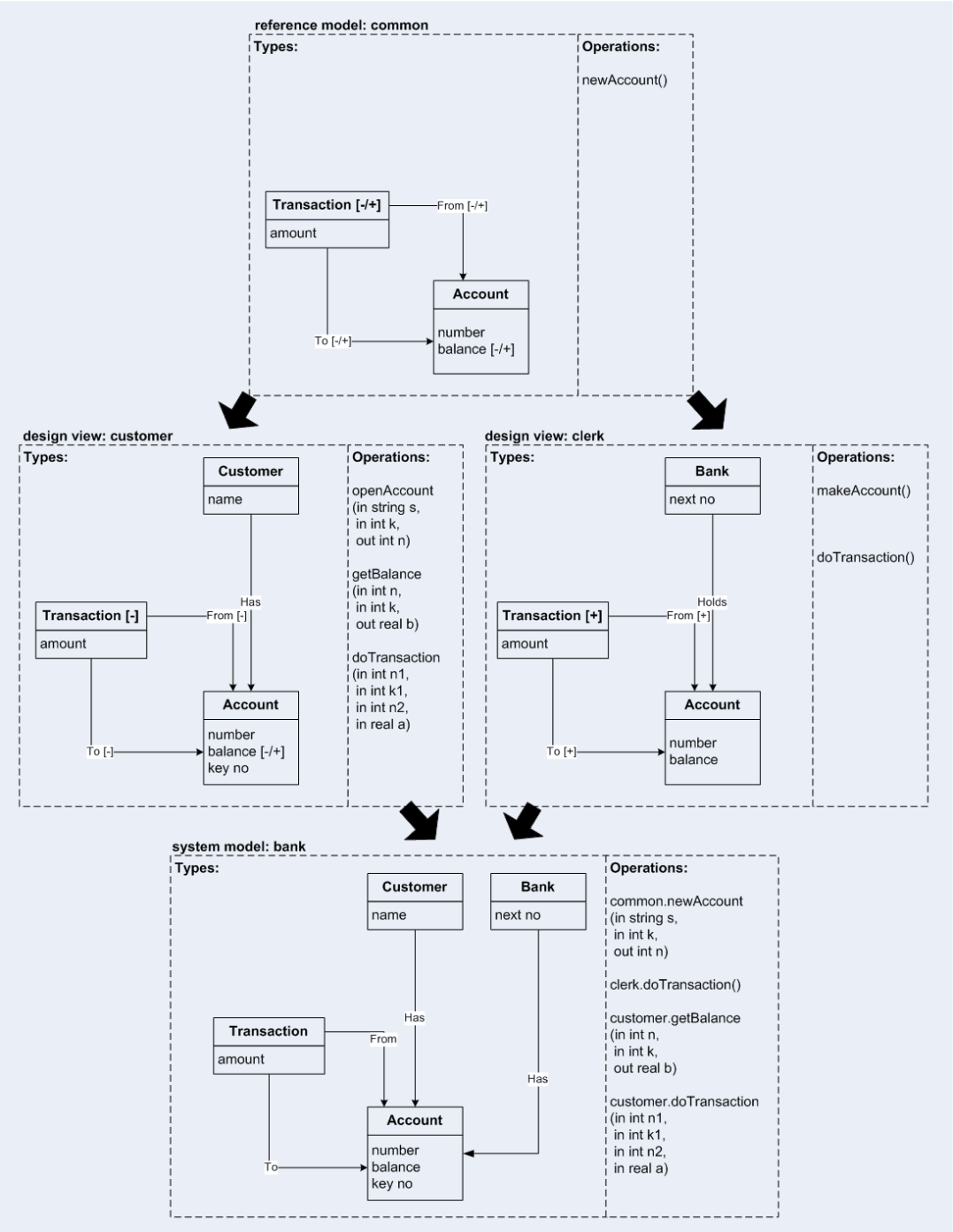

Abbildung 8: Integration der Kunden- und Angestelltensichten

Bezeichner der Name der jeweiligen Sicht voranstellt wird, z.B. ($\mathcal{O}_i$.Bezeichner), so dass Elemente genau dann einen gemeinsamen Namen tragen, wenn sie den selben Ursprung in $\mathcal{O}_0$ haben. Als Ergebnis dieses Schritts erhalten wir $\mathcal{O}'_1$ und $\mathcal{O}'_2$.

2. Konstruktion des Systemmodells durch komponentenweise mengentheoretische Vereinigung von $\mathcal{O}'_1$ und $\mathcal{O}'_2$.

6.4 Schritt 1 - Umbenennung

Wir erhalten $\mathcal{O}_1'$ und die Umbenennung $\mathcal{O}_1 \overset{ren_1'}{\leftrightarrow} \mathcal{O}_1'$ durch Erweitern der Umbenennung $\mathcal{O}_0^1 \overset{ren_1}{\leftrightarrow} \mathcal{O}_0$ auf $\mathcal{O}_1$. Genauer, $ren_1'|_{\mathcal{O}_0^1} = ren_1$ und ist minimal in dem Sinne, dass sonst keine Elemente umbenannt werden. Die Umbenennung ist also die Identität auf $\mathcal{O}_0 \setminus \mathcal{O}_0^1$. $\mathcal{O}_1'$ wird zu einer Erweiterung von $\mathcal{O}_0$. Analog erhalten wir $\mathcal{O}_2'$ und ren_2'. Der Umbenennungsschritt ist immer durchführbar und kann deswegen auch automatisiert werden.

6.5 Schritt 2 - Konstruktion

Das Systemmodell $\mathcal{O}_3' = \langle TG_3', P_3', \pi_3', O_3' \rangle$ erhalten wir dann schließlich, indem wir die Vereinigungen der Typgraphen $TG_3' = TG_1' \cup TG_2'$ und der Bezeichner der Produktionen $P_3' = P_1' \cup P_2'$ von $\mathcal{O}_1'$ und $\mathcal{O}_2'$ bilden. Für eine konkrete Regel $\pi_3'(p')$ mit $p' \in P_3'$ unterscheiden wir drei Fälle:

- wenn $p' \in P_1' \setminus P_2'$, dann ist $\pi_3'(p') = \pi_1'(p')$, d.h. die Produktion von $\mathcal{O}_1'$ wird übernommen.
- wenn $p' \in P_2' \setminus P_1'$, dann ist $\pi_3'(p') = \pi_2'(p')$, d.h. die Produktion von $\mathcal{O}_2'$ wird übernommen.
- wenn $p' \in P_1' \cap P_2'$, dann ist $\pi_3'(p') = L_1' \cup L_2' \to R_1' \cup R_2'$ die komponentenweise nicht-disjunkte Vereinigung der linken und rechten Regelseiten von $\pi_i'(p') = L_i' \to R_i'$ für $i = 1, 2$.

Die offenen Typen $O_3' = \langle O'^-_3, O'^+_3 \rangle$ erhalten wir durch Schnitt der offenen Typen von $\mathcal{O}_1$ und $\mathcal{O}_2$, also $O'^-_3 = O_1^- \cap O_2^-$ und $O'^+_3 = O_1^+ \cap O_2^+$.

Die Abbildungen 7 und 8 zeigen beispielhaft je eine Integration von zwei Produktionen und zwei Sichten. In Abbildung 7 wird vorgeführt, wie die beiden Produktionen *customer.openAccount* und *clerk.makeAccount* durch Vereinigung der respektiven linken und rechten Regelseiten integriert werden. Die Integration ist erlaubt, weil beide Produktionen in der Regel *common.newAccount* den selben Ursprung im Referenzmodell haben. Sie respektiert außerdem die Vor- und Nachbedingungen von *customer.openAccount* und *clerk.makeAccount*. So setzt etwa das aus der Integration resultierende *common.newAccount* die Existenz eines Kunden- und Bankobjekts voraus.

Abbildung 8 illustriert die Integration der Kunden- und Angestelltensichten (Abbildung 5). Das Resultat dieser Integration ist das Systemmodell Bank. Auffällig ist hierbei, dass die beiden Produktionen *customer.doTransaction* und *clerk.doTransaction* in ihren jeweiligen Sichten zwar denselben Namen tragen, aber nicht integriert werden, weil sie keinen gemeinsamen Ursprung im Referenzmodell haben. Die Folge davon ist, dass beide Produktionen (inklusive vorgestellter Qualifikation durch den jeweiligen Namen der Herkunftssicht) in das Systemmodell übernommen werden. Bemerkenswert ist auch, dass alle Typen des Systemmodells geschützt sind. Grund hierfür ist die Tatsache, dass das Systemmodell das Endprodukt unseres Spezifikationsprozesses ist und keine weiteren Änderungen vorsieht. Generell kann man sich aber auch andere Situationen vorstellen, etwa wenn noch Interaktionen mit anderen Banken modelliert werden sollen.

6.6 Ansätze zur Integration mehrerer Sichten

Bei mehr als zwei Sichten ist es ungünstig, das Referenzmodell als alleinige Ableitungsbasis zu benutzen, weil etwa bei der Integration von zwei Sichten $\mathcal{O}_1$ und $\mathcal{O}_2$ dem Referenzmodell möglicherweise neue Elemente hinzugefügt werden. Diese

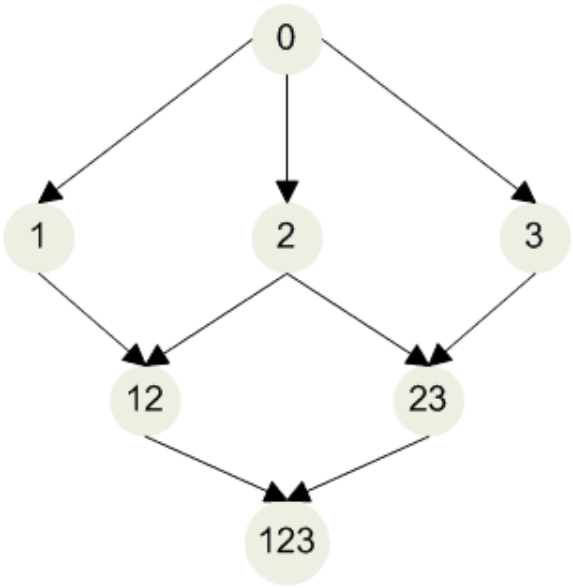

Abbildung 9: Triviale Integration von drei Sichten

Elemente wären dann aber auch direkt in einer dritten Sicht $\mathcal{O}_3$ sichtbar, was nicht notwendigerweise erwünschenswert ist. Sinnvoll ist es deswegen für je zwei Sichten, die semantische Konzepte teilen sollen, zusätzlich eine *abstrakte Sicht*, welche ebenfalls auf dem Referenzmodell basiert, zwischenzuschalten. Das Referenzmodell selbst bleibt unverändert. Dieses Vorgehen resultiert in einer Hierarchie von Sichten, die bei Integration auf mehrere Arten aufgelöst werden kann. Der naheliegendste Ansatz konstruiert iterativ über paarweise Integration von je zwei Sichten das Systemmodell. Andere Strategien sind ebenfalls denkbar, von einer detaillierten Betrachtung sehen wir an dieser Stelle aber ab.

Abbildung 9 veranschaulicht den trivialen Integrationsvorgang (durch paarweise Integration) bei drei vom Referenzmodell 0 abgeleiteten Sichten, ohne Zwischenschaltung von abstrakten Sichten. Erst werden die Sichten 1 und 2 (zu 12) sowie 2 und 3 (zu 23) integriert, um dann im nächsten Schritt durch Integration von 12 und 23 das Systemmodell 123 zu erhalten.

7 Zusammenfassung

Der hier vorgestellte Ansatz der Sicht-basierten Systemspezifikation beruht auf Offenen Graphtransformationssystemen und hat die folgenden charakteristischen Eigenschaften:

- Einzelne Sichten eines Systems teilen ein gemeinsames Referenzmodell.
- Sichten, Referenzmodell und Systemmodell werden durch OGTS repräsentiert.
- Eine Sicht ist durch eine lose Semantik charakterisiert, möglicherweise eingeschränkt durch eine explizite Rahmenbedingung.
- Der Modell-Manager erweitert das Referenzmodell, wenn neue Abhängigkeiten im Entwicklungsprozess offensichtlich werden. Im Fall von mehr als zwei Sichten können abstrakte Sichten eingeführt werden. Die daraus resultierende Sichthierarchie wird dann über eine geeignete Strategie auf ein Systemmodell zusammengeführt.

8 Quellenangaben
Literatur

[1] R. Heckel, G. Engels, H. Ehrig und G. Taentzer. A view-based approach to System Modeling based on Open Graph Transformation Systems. In H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools*. World Scientific, 1999.

[2] M. Andries und G. Engels. A hybrid query language for the extended entity relationship model. *Journal of Visual Languages and Computing*, 7(3):321-352, 1996. Special Issue on Visual Query Systems.

[3] C. Batini, M. Lenzerini und S. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323-364, 1986

[4] P. Böhm, H.-R. Fonio und A. Habel. Amalgamation of graph transformations: a synchronization mechanism. *Journal of Computer and System Science*, 34:377-408, 1987

[5] A. Corradini, U. Montanari und F. Rossi. Graph processes. *Fundamenta Informaticae*, 26(3,4):241-266, 1996

[6] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel und M. Löwe. Algebraic approaches to graph transformation, Part I: Basic concepts and double pushout approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*, pages 163-245. World Scientific, 1997.

[7] C.J. Date. *An Introduction to Database Systems*. Addison Wesley, 1979.

[8] G. Engels, R. Heckel, G. Taentzer und H. Ehrig. A combined reference model- and view-based approach to system specification. *Int. Journal of Software and Knowledge Engineering*, 7(4):457-477, 1997

[9] G. Engels, R. Heckel, G. Taentzer und H. Ehrig. A view-oriented approach to system modeling using graph transformation. In *Proc. of ESEC/FSE'97, Zürich, LNCS 1301*, pages 327-343. Springer Verlag, 1997.

[10] A Finkelstein, J. Kramer, B. Nuseibeh, M. Goedicke und L. Finkelstein. Viewpoints: A framework for integrating multiple perspectives in system development. *International Journal of Software Engineering and Knowledge Engineering*, 2(1):31-58, March 1992

[11] T. Fischer, J. Niere, L. Torunski und A. Zündorf. Story diagrams: A new graph transformation language based on UML and Java. In *Proc. 6th Int. Workshop on Theory and Application of Graph Transformation (TAGT'98), Paderborn*, November 1998.

[12] R. Heckel. *Open Graph Transformation Systems: A New Approach to the Compositional Modelling of Concurrent and Reactive Systems.* PhD thesis, TU Berlin, 1998.

[13] R. Heckel, H. Ehrig, U. Wolter und A. Corradini. Double-pullback transistions and coalgebraic loose semantics for graph transformation systems. *Applied Categorical Structures*, 1999. To appear; also available as technical report TR 97-07, Department of Computer Science, TU Berlin.

[14] D. Jackson. Structuring Z specifications with views. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 4(4):365-389, October 1995

[15] T. Lewis, L. Rosenstein, W. Pree, A. Weinhand, E. Gamma, P. Calder, G. Andert, J. Vlissides und K. Schmucker. *Object-oriented Application Frameworks*. Prentice-Hall, 1996.

[16] M. Nagl, editor. *Building Tightly Integrated Software Development Environments: The IPSEN Approach, LNCS 1170*. Springer Verlag, 1996.

[17] L. Ribeiro. *Parallel Composition and Unfolding Semantics of Graph Grammars*. PhD thesis, TU Berlin, 1996.

[18] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.

[19] A. Schürr und A.J. Winter. UML packages for PROgrammed Graph REwrite Systems. In *Proc. 6th Int. Workshop on Theory and Application of Graph Transformation (TAGT'98), Paderborn*, November 1998.

[20] A. Schürr, A.J. Winter und A. Zündorf. Graph grammar engineering with PROGRES. In *5th European Software Engineering Conference (ESEC'95), Sitges, LNCS 989*, pages 219-234. Springer Verlag, 1995.

[21] C. Tuijn und M. Gyssens. CGOOD, a categorical graph-oriented object data model. *TCS*, 160:217-239, 1996.

[22] B. Westfechtel. Structure-oriented merging of revisions of software documents. In P. Feiler, editor, *Proc. 3rd Int. Workshop on Software Configuration Management (SCM3) New York*, pages 68-79. ACM Press, 1991.

[23] A. Zamperoni. GRIDS - graph-based integrated development of software: Integrating different perspectives of software engineering. In *Proc. 18th International Conference on Software Engineering (ICSE)*, pages 48-59. IEEE CS Press, March 25-29 1996.