Graphtransformation als Konzeptionelles und Formales Rahmenwerk für Systemmodellierung und Modellevolution

Tim Dahmen, 2018769
im Rahmen des Seminars Evolution von Softwaresystemen

07.05.2002

Zusammenfassung

Im Umfeld großer, dynamischer und verteilter Softwaresysteme stellt das effiziente Finden und konsistente Beschreiben geeigneter Systemmodelle ein nach wir vor ungelöstes Problem dar. Die Unified Modelling Language hat sich als Quasi-Standard durchgesetzt und unterstützt den Leser mit einfach zu verstehenden Diagrammen, versagt jedoch weitestgehend wenn es darum geht, den Designer solcher Modelle durch geeignete Sprachstrukturen ebenfalls zu unterstützen und zu leiten. Dieser Artikel versucht, aufbauend auf UML, ein formales Rahmenwerk zu schaffen, das eben diese Lücke von UML schließt, und die Sprache so auf eine saubere formale Basis zu stellen. Graphtransformationssysteme werden als formales Rüstzeug zur Beschreibung dynamischer Aspekte des Systemmodells eingeführt und ihre Anwendung wird anhand von praktischen Beispielen erklärt.
Keywords: *Graphtransformation, Software Design, UML, Systemmodellierung*

1 Einführung

Die Formulierung eines geeigneten Systemmodells als abstrakte Zwischenschicht zwischen realer Welt auf der einen und Softwareimplementierung auf der anderen Seite gilt als unverzichtbar für die Beherrschbarkeit moderner verteilter Applikationen. Die Formulierung eines solchen Modells erfolgt typischerweise in der Standardsprache UML, das Ausarbeiten dieses Modells ist jedoch ein kaum standardisierter Vorgang, der generell als schlecht beherrschbar und fehleranfällig gilt. Besonders kritisch sind Fehler in dieser frühen Projektphase insbesondere deshalb, weil sie oftmals erst sehr spät erkannt werden, so daß in der Zwischenzeit geleistete Entwicklungsarbeit ganz oder teilweise vergeblich ist.

2 Konzeptionelles Rahmenwerk

2.1 Voraussetzungen

Im Folgenden werden wir uns mit der Entwicklung eines Systemmodells für moderne Anwendungen befassen, was drei wesentliche Punkte umfaßt. Das entwickelte Modell soll Software beschreiben , die folgende Eigenschaften ausweist:

- **groß:** der Umfang der Projekte soll sowohl in beteiligten Entwicklern, als auch in technischer Komplexität und Zeitaufwand dem realer Großprojekte entsprechen.
- **verteilt:** die Hardware soll ein räumlich oder logisch verteiltes Netzwerk aus Knoten sein, die jeweils über mindestens einen Prozessor sowie eigenen Speicher verfügen. Die Kommunikation zwischen den Knoten erfolgt über Schnittstellen.
- **dynamisch:** die Software kann erweitert werden, in dem zur Laufzeit des Systems Komponenten erweitert, hinzugefügt oder entfernt und entsprechend gelinkt werden.

2.2 Ziele

Als sinnvolle Ergänzung von UML streben wir ein standardisiertes Systemmodells an. Dieses Modell soll den Designer eines Systems unterstützen, indem es relevante Dimensionen des Systemmodells vorgibt, und so einen praktischen Leitfaden zum Design bereitstellt. Auch soll es ein in sich geschlossenes, formal sauberes

Rahmenwerk liefern, dessen Inhalt dann wieder mittels ausgesuchter UML Diagramme beschrieben werden kann.

2.3 Gliederung

Das formale Rahmenwerk gliedert sich in drei relevante Dimensionen (s. Abbildung 1). Jede dieser drei Dimensionen ist wiederum hierarchisch in drei Ebenen gegliedert (s. Abbildung 2). Die drei Dimensionen sind so gewählt, dass sie die wichtigsten Aspekte des Systemmodells abdecken, dabei jedoch vollständig orthogonal zueinander stehen, d.h. sie weisen keinerlei Redundanz auf.

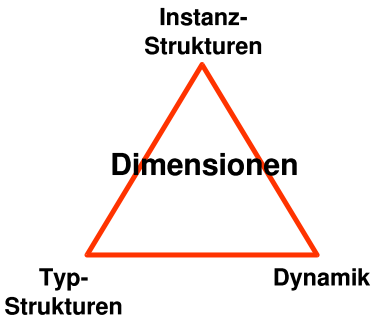

Abbildung 1: Gliederung des Systemmodells in drei Dimensionen

2.4 Strukturen der Instanzen

Die erste Dimension ist die der Instanzstrukturen. Diese Dimension beschreibt die hierarchische Gliederung und das Verhältnis der einzelnen Objekte zueinander. Die Strukturen sind hierarchisch in drei Ebenen untergliedert: auf unterster Ebene die Hardwarestrukturen, bestehend aus Knoten und Schnittstellen. Darüber finden wir die Komponentenebene mit Softwarekomponenten und deren Verknüpfungen untereinander. Auf oberster Ebene schließlich stehen die sogenannten *Problem Domain Objects*, konkrete Strukturen wie etwa Funktionen, Variablen oder Objekte, die ihre Entsprechung direkt in objektorientierten Programmiersprachen finden.

Strukturen auf Instanzebene werden beschrieben mit UML Diagrammen, wobei die Art des Diagramms vom zu beschreibenden Level abhängt. Für Hardware Strukturen verwendet

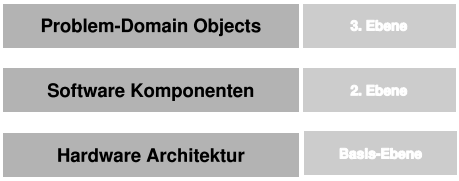

Abbildung 2: Hierarchische Gliederung des Systemmodells in drei Ebenen

man *deployment diagrams*. Für die darüber liegenden Ebenen werden diese Diagramme dann hierarchisch erweitert und verfeinert und um *Komponenten* und *Problem Domain Objects* ergänzt.

Da diese *deployment diagrams* eine sehr anschauliche und intuitive Darstellung von Graphen sind, entsteht hier die Verbindung zu Graphtransformationen: alle Objekte und Schnittstellen zwischen diesen bilden zusammen einen Graphen, in der Form, dass die Objekte die Knoten bilden und die Schnittstellen die Kanten.

Bereits an diesem einfachen Fall kann man sehen, wie unser System Modell mit UML zusammenarbeitet: die formale und saubere Struktur wird durch das Modell gegeben und geht weit über die Möglichkeiten von UML hinaus, zur Notation und Kommunikation des Modells nach außen kommt jedoch nach wie vor UML zum Einsatz.

2.5 Strukturen der Typen

Als zweite Dimension betrachten wir die Typisierung der Objekte. Diese Dimension ist vollkommen analog zur Dimension der Instanzstrukturen in die selben drei Ebenen gegliedert. Für die Dimension der Typstrukturen sollen alle Eigenschaften gelten, die man von Typisierung erwartet. Diese sind wohlverstanden und werden in reichlich vorhandener Literatur zum Thema Programmiersysteme ausführlich behandelt. Für uns wichtig soll nur die Tatsache sein, dass zwischen der Menge der Instanzen I und der Menge der Typen T ein Homomorphismus $g \subseteq I \times T$ existiert, der jeder Instanz genau einen Typ zuordnet.

Interessanter, ist die Darstellung der Typstrukturen. Für die Notation von Typstrukturen auf der Ebene der *Problem Domain Objects* verwenden wir UML Klassendiagramme. Die Notation auf Ebene der Software

Komponenten entfällt, da hier eine Typisierung zwar denkbar und vom theoretischen Standpunkt aus auch zu erwarten wäre, jedoch keine praktisch sinnvolle Interpretation für die Softwareentwicklung erkennbar ist. Auf Ebene der Hardware hingegen existieren sehr wohl wieder Typstrukturen. Diese werden durch Diagramme notiert, welche den UML Klassendiagrammen sehr ähnlich sind, jedoch anstelle von Klassen-Symbolen sogenannte Knoten-Symbole verwenden.

2.6 Dynamische Strukturen

Wirklich interessant wird das Rahmenwerk jedoch erst bei Betrachtung der dritten Dimension, der Dynamik. Diese beschreibt die Veränderung von Instanzstrukturen und Typstrukturen über die Zeit.

2.6.1 Instanzlevel Dynamik

Die dynamische Veränderung des Instanzlevels, also die Veränderung des Systems zur Laufzeit, wird durch Graphtransformationsregeln beschrieben. Eine solche Regel, bestehend aus zwei Graphen, genannt linksseitiger Graph und rechtsseitiger Graph versteht sich als abstrakte Spezifikation einer Funktion. In dieser Interpretation der Graphtransformationsregel stellt der linksseitige Graph den Zustand eines Teiles des Gesamtsystems **vor** Ausführung der Funktion dar, der rechtsseitige Graph den Zustand **danach**. Das Interessante an dieser Spezifikation ist, dass sie sich ausschließlich auf die *Problem Domain Objects* bezieht. Die Spezifikation der hierfür erforderlichen Aktionen auf den tiefer liegenden Ebenen Softwarekomponenten und Hardware, fällt der Abstraktion zum Opfer. Gerade dieser hohe Grad der Abstraktion aber macht die Graphtransformationsregeln für die Spezifikation interessant.

2.6.2 Typlevel Dynamik

Der letzte Teil unseres Systemmodells, die Dynamik der Strukturen auf Typebene, also die Veränderung der Softwarestruktur über die Zeit, deckt sich fast vollständig mit dem Begriff der Softwareevolution. Offensichtlich ist es nicht sinnvoll möglich diese in der Design Phase (oder zu irgend einem anderen Zeitpunkt **bevor** sie vollzogen wurde) zu spezifizieren. Der Grund hierfür ist trivial: hätte man bereits früher gewußt, dass man einen anderen Zustand der Software benötigt, hätte man sie direkt so entworfen, und die Anpassung wäre nicht erforderlich.

3 Beispiele

Im Folgenden werde ich zwei kleine Beispiele heranziehen, um die Anwendung unseres Systemmodells anhand fiktiver Probleme zu verdeutlichen. Das Beispiel stammt hierbei aus dem zugrunde liegenden Papier.

3.1 Smart Card

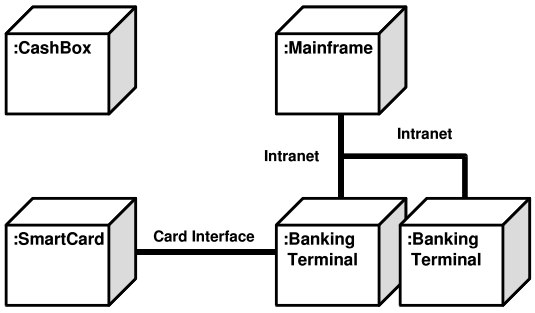

Abbildung 3: Diagramm der Instanzlevelstrukturen auf Hardwareebene, entspricht einer Momentaufnahme des Systemzustands

Das Beispiel ist eine fiktive Anwendung aus dem Finanz Sektor. Zu implementieren ist eine *Smart Card*. Der Grundgedanke hierbei ist, dass ein Kunde über eine solche Smart Card verfügt. Diese Smart Card kann, ausgestattet mit einem kleinen Prozessor und eigenem Speicher via eine Schnittstelle (*Cardinterface*) mit einer Komponente in Geschäften (*Cashbox*) oder aber mit Bankterminals kommunizieren. Die Bankterminals ihrerseits sind via Intranet mit einem *Mainframe* der Bank verbunden (s. Abbildung 4). Die Bezahlung einer Rechnung (*Bill*) erfolgt, indem das *Bill*-Objekt von der *CashBox* auf die *SmartCard* übertragen wird und dort zwischengespeichert. Später wird das *Bill*-Objekt dann von der *SmartCard* aus über das *BankTerminal* auf den *Mainframe* der Bank übertragen und dort eine entsprechende Überweisung zum Begleichen der Rechnung veranlaßt.

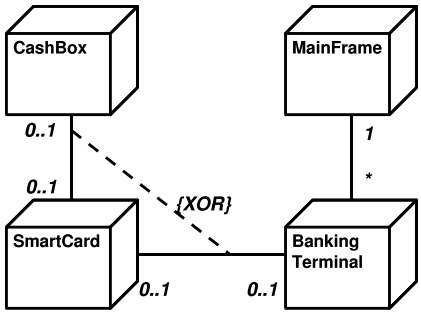

Abbildung 4: Klassendiagramm der Typlevelstrukturen auf Hardwareebene, ergänzt Quantifizierungen der Beziehungen

Analog hierzu betrachten wir direkt die Typstrukturen auf der Ebene der Hardware. Diese werden durch ein UML Klassendiagramm für Hardware notiert. Die **{xor}** Beziehung soll ausdrücken, dass die *Smartcard* entweder mit der *CashBox* oder aber mit dem BankTerminal verbunden sein kann, nicht aber gleichzeitig mit beiden.

Als Beispiel für die Dynamik des Instanzlevels auf Ebene der Hardware seien hier die Methoden *insertCard()* (Abb. 5) angegeben, die zur Verbindung der Hardwarekomponenten *CashBox* und *SmartCard* dienet. Die Regel weist keine relevanten Randbedingungen auf.

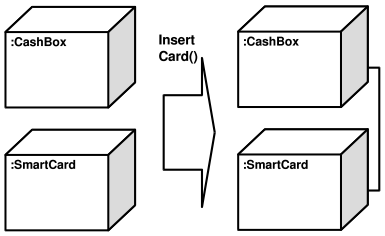

Abbildung 5: Graphtransformationsregel zur Spezifikation der Methode *InsertCard()*.

Interessanter ist die Methode *ConnectBilling()* (Abb. 6), die zur Verbindung der Softwarekomponenten *Billing* und *BillCard* dient. Im Diagramm dieser Graphtransformationsregel kann man auch deutlich erkennen, wie die Spezifikation von Lowlevel Erfordernissen an Methoden durch die Abstraktion der Regel versteckt werden. In diesem Beispiel ist die Information, dass zur Verbindung der Komponenten *Billing* und *BillCard* zunächst die entsprechenden Hardwareknoten verbunden werden müssen nicht mehr direkt sichtbar.

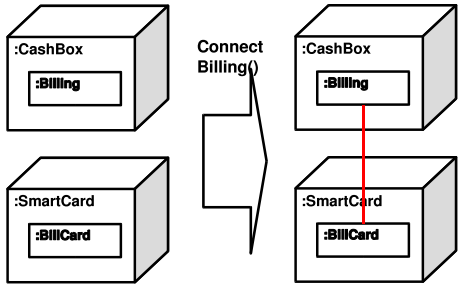

Abbildung 6: Graphtransformationsregel zur Spezifikation der Methode *ConnectBilling()*. Erfordernisse auf Ebene der Hardwarestrukturen sind nur implizit enthalten.

Als letztes Beispiel betrachten wir nun noch die Methode *payBill()* (Abb. 7), die zum Begleichen einer Rechnung dient und als Beispiel für die Spezifikation einer wirklichen Systemfunktion angesehen werden kann.

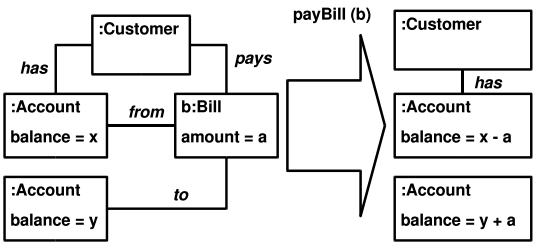

Abbildung 7: Graphtransformationsregel zur Spezifikation der Methode *PayBill()*. Die etwas komplexere Spezifikation enthält eine Reihe von Nebenbedingungen, die die Anwendbarkeit der Regel auf gültige Fälle einschränken.

4 Formalisierung mittels Graphtransformationen

Nachdem im vorangegangenen Kapitel der Fokus der Betrachtung auf der Entwicklung eines Systemmodells lag, soll nun die theoretische Basis dieses Rahmenwerkes, die Graphtransformationssystem (GTS) genauer untersucht werden.

Der nun folgende, formale Teil dieses Artikels basiert stark auf [2]

4.1 Formale Definition

Unter einer Graphtransformationsregel (GTR) r versteht man ein 5-Tupel $r = (L, R, K, glue, emb, appl)$ wobei

- $L := (V_l, E_l)$ heißt linksseitiger Graph
- $R := (V_r, E_r)$ heißt rechtsseitiger Graph
- $K := (V_k, E_k)$ heißt Interfacegraph, wobei $V_k \subseteq V_l \wedge V_k \subseteq V_r$ und $E_k \subseteq E_l \wedge E_k \subseteq E_r$
- *glue* ist ein Auftreten von K in R
- $emb \subseteq V_l \times V_r$
- *appl* eine Menge von Randbedingungen

Graphisch dargestellt werden GTR meist in Form von zwei Graphen, die mit einem Pfeil verbunden sind (s. Abbildung 7). Die Intuition hinter der Anwendung einer solchen Regel auf einen Graphen G ist, dass man ein Auftreten von L in G sucht, um dann L durch R zu ersetzen.

Etwas genauer betrachtet sind zur Anwendung einer Regel r auf einen Graphen $G = (V, E)$ folgende Schritte erforderlich

- **Wähle** ein Auftreten von L in G
- **Überprüfe** die Anwendungsbedingung *appl*, so daß diese zu wahr evaluiert [1]
- **Entferne** L ohne K aus g, ebenso wie alle losen Kanten. Eine Kante (a, b) heißt lose gdw. $a \in V_l \wedge b \notin V_l \vee b \in V_l \wedge a \notin V_l$. Der aus diesen Entferne Operationen resultierende Graph heißt Kontextgraph D.
- **Klebe** R in D. Dies geschieht mittels des Auftretens von K in D und R, indem zunächst die disjunkte Vereinigung von D und R gebildet wird, und dann für jeden Knoten $v \in K$ die entsprechenden Knoten in D und R gesucht werden. Der so entstandene Graph heißt Klebegraph E.
- **Bette** R in E gemäß der Einbettungsrelation *emb*.

[1] Sollten die Bedingungen in appl nicht zu falsch evaluieren, so kann die Regel in diesem Kontext nicht angewendet werden. Im Rahmen eines ganzen Graphtransformationssystems bleibt nichts anderes übrig, als ein anderes Auftreten von L in G zu wählen und von vorne zu beginnen. Auf welche Art und Weise dies geschieht wird durch den verwendeten Algorithmus bestimmt und spielt eine große Rolle für die Effizienz von Systemen, die Graphtransformationen als Berechnungskalkulus verwenden. Für unsere Anwendung zur Spezifikation ist diese Frage jedoch unerheblich.

4.2 Graphtransformations Systeme

Ein GTS ist eine Menge von GTR. GTS entsprechen den kontextsensitiven Grammatiken über Strings und sind zu diesen äquivalent. Dies folgt aus folgender Überlegung: jeder Graph läßt sich in geeigneter Notation durch einen String beschreiben läßt (z.B. indem man einfach V und E auflistet) und jeder String als Graph (z.B. durch eine Liste von Knoten, die den Buchstaben entsprechen). Somit gelangt man mit wenig Aufwand zu der Erkenntnis, dass GTS Turing universell sind (da kontextsensitive Grammatiken diese Eigenschaft ebenfalls besitzen).

4.3 Eigenschaften von Graphtransformations Regeln

Im Folgenden werden wir einige interessante Eigenschaften von GTS untersuchen.

- **Lokalität** Die Anwendung einer GTR r auf einen Graphen G verändert nur das unmittelbare Umfeld der in in der Regel explizit auftretenden Teilmenge von G. Diese Eigenschaft der Lokalität ergibt sich aus der Definition der Regelanwendung, sie kann also ohne weiteren Aufwand in den auf GTS basierenden Systemmodellen vorausgesetzt werden.
- **Kontakteigenschaft** Eine GTR besitzt die Kontakteigenschaft gdw. im Schritt *Entferne* keine losen Kanten entstanden sind.
- **Identifizierbarkeit** eine GTR heißt identifizierbar, gdw. das Auftreten von L in G nur Knoten aus K identifiziert.
- **Invertierbarkeit** Eine GTR $r$ist invertierbar, gdw. folgende Bedingungen gelten:
  - r besitzt die Kontakteigenschaft
  - r ist identifizierbar
  - *glue* ist injektiv
  - *emb* ist leer

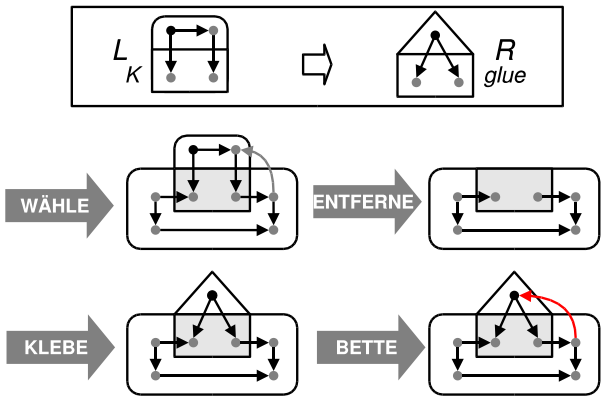

Abbildung 8: schematische Skizze der Anwendungsschritte einer Graphtransformationsregel Regel

- **Konsistenzbedingungen** Für viele praktische Anwendungen ist das Formulieren und Beweisen von Konsistenzbedingungen eine der wichtigsten Anwendungen des Systemmodell. Wie schwierig dies ist und ob solche Beweise gelingen hängt natürlich von der Art des Modells und der Bedingung ab. Die Vorzüge, die GTS an dieser Stelle bieten liegen darin, daß Graphentheorie eine der am meisten untersuchten und am besten verstandenen Teilgebiete der Informatik ist. Durch die Verwendung von GTS als Basis des Systemmodells erhält man also ein weit entwickeltes theoretisches Werkzeug an die Hand, das viele Beweise stark vereinfacht, da man sich oftmals auf bereits bewiesene Eigenschaften bestimmter Graphen stützen kann.

- **Terminierung** Allgemeine Aussagen über das Terminieren von GTS sind offensichtlich nicht möglich, da GTS Turing universell sind (siehe Absatz 4.2). Für Sonderfälle gelingen solche Terminierungsbeweise jedoch einfach. Gilt $|V_l| \geq |V_r| \forall r \in GTS$, so reduziert jede Regelanwendung die Größe des Graphen, nach endlich vielen Schritten ist keine weitere Anwendung mehr möglich.

5 Fazit

Es wurde ein universell anwendbares Systemmodell für große, verteilte und dynamische Softwaresysteme entwickelt. Relevante Dimensionen wurden identifiziert und systematisch gegliedert. Es wurde gezeigt, wie die statischen Teile eines konkreten Systemmodells mit Hilfe einer Teilmenge von UML beschrieben werden können. Für die Spezifikation der dynamischen Teile wurden Graphtransformationen vorgestellt und erst anhand von Beispielen, dann formal eingeführt. Wichtige Eigenschaften von GTS wurden vorgestellt und deren Bedeutung für die Analyse von darauf basierenden Systemmodellen aufgezeigt.
Die Evolution von Softwaresystemen, für dieses Seminar der interessanteste Aspekt des Softwaredesign, wird von dem vorgestellten universellen Systemmodells durch die dynamischen Aspekte der Typstrukturen erfasst. Leider lässen sich jedoch genau diese Aspekte mit den vorgestellten Mitteln nicht beschreiben.

Literatur

[1] Gregor Engels, Reiko Heckel *Graph Transformation as a Conceptual and Formal Framework for System Modelling and Model Evolution* LNCS 1853, p. 127ff University of Paderborn

[2] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, Gabriele Taentzer *Graph Transformation for Specification and Programming*