Generative Softwarekonstruktion auf Basis typisierter Komponenten

Frank Griffel, Christian Zirpins und Stefan Müller-Wilken

Arbeitsgruppe Verteilte Systeme – Fachbereich Informatik
Universität Hamburg
Vogt-Kölln-Str. 30, D-22527 Hamburg
{griffel|zirpins|smueller}@informatik.uni-hamburg.de

Zusammenfassung Die komponentenbasierte Entwicklungssicht erhält zunehmend Aufmerksamkeit, insbesondere auf Grund der von ihr in Aussicht gestellten vielversprechenden Vorteile wie besserer Produktivität, Test- und Wartbarkeit, Wiederverwendung und Qualität. Andererseits fehlt immer noch ein solides Fundament, auf dem sich solche Erwartungen gründen können. Zumindest bildet jedoch ein kompositorisches Vorgehen zweifellos den Kern kompentenbasierter Entwicklung. Dieser Beitrag stellt daher einen als *Generative Softwarekonstruktion* bezeichneten Entwicklungsprozess vor, der auf Basis eines *interaktionsorientierten* Typmodells eine hochgradig automatisierte Komposition verteilter Anwendungssysteme erlaubt. Das semantisch reiche Typmodell wird dabei als Erweiterung des aktuellen CORBA Component Model vorgestellt, unterstützt eine typkorrekte und bei Bedarf automatisch adaptierende Komposition und wird anhand seiner Anwendung in der Finananzdienstleistungsdomäne illustriert.

1 Generative Softwarekonstruktion

Das Komponentenparadigma ist vielleicht der aussichtsreichste Kandidat auf dem Weg zu einer industriellen Softwareproduktion. Von deren typischen Merkmalen (nach [14]) wie

- Fertigung auf Abruf / bei Bedarf aus vordefinierten Produktbestandteilen
- Auftreten von Drittanbietern die nach gemeinsamen Spezifikationen arbeiten
- Produktpflege/-wartung durch Teile-Austausch
- Unterstützung kompletter Lieferketten von der Fertigung bis zur Auslieferung

ist jedoch auch die heutige Komponentendiskussion noch weit entfernt und vielfach von Wunschdenken geprägt. Um diese Erwartungen erfüllen zu können, reicht eben nicht allein die Betonung des Bausteinaspekts in der Softwareentwicklung, sondern der gesamte Entwicklungsprozess muss überdacht werden. Die Programmierung „Zeile-für-Zeile“ ist für die heute rasch und in guter Qualität zu liefernden, zunehmend komplexeren Anwendungen nicht mehr adäquat. Ein Zusammensetzen aus vorgefertigten Bausteinen erscheint deutlich angemessener. Je

stärker eine solche Komposition durch entsprechende Entwicklungsumgebungen unterstützt und automatisiert werden kann, um so leichter können robuste und fehlerfreie Lösungen produziert werden. Daher werden Softwareentwicklungsumgebungen und -werkzeuge zukünftig immer stärker von *generierenden* Konzepten durchdrungen werden, die ausgehend von überschaubaren und anwendungsnahen (problembezogenen) Architekturbeschreibungen komplexe technische Anwendungssysteme erstellen zu können.

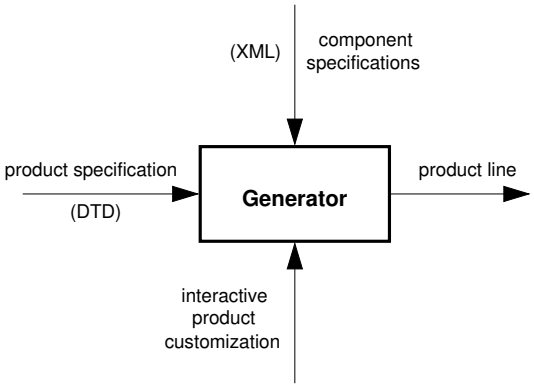

Abbildung 1. Grundkonzeption einer generativen Systemerstellung

Abbildung 1 illustriert das Prinzip einer Generator-basierten Anwendungserstellung aus bereitstehenden Komponenten. Grundidee ist die Parametrisirung des Generators mit einer Spezifikation der gewünschten Anwendung. Die Analyse dieser Architekturspezifikation erlaubt dem Generator dann die Auswahl passender Komponenten. Deren individuelle Konfigurationen sind zum Teil durch die gewählte Kombination mit anderen Komponenten festgelegt, zum Teil kann jedoch noch eine manuelle Einstellung erwünschter Eigenschaften durch den Entwickler erfolgen. Ergebnis der Generierung ist dann eine konkrete, ausführbare Applikation. Genau genommen kann der Output des Generators als *Produktlinie* bezeichnet werden, da eine ganze Reihe von ähnlichen Anwendungen erzeugt werden kann, die sich in wesentlichen Elementen oder Eigenschaften gleichen und ein identisches Anwendungsproblem adressieren. Wird der Generator mit unterschiedlichen, aber einer gemeinsamen Domäne zugehörigen Architekturspezifikationen parametrisiert, kann insbesondere auch von einer entstehenden *Produktfamilie* gesprochen werden, innerhalb derer beispielsweise ein hoher Wiederverwendungsgrad einzelner Komponenten möglich wird.

Der Beitrag gliedert sich im Weiteren wie folgt. Abschnitt 2 führt den zu Grunde gelegten Entwicklungsprozess ein und stellt die Arbeitsweise des entwickelten Generators vor. Im Abschnitt 3 werden aktuelle verwandte Ansätze diskutiert. Abschnitt 4 gibt einen Überblick über das konzipierte Typmodell

und stellt seine Verwendung vor. Der zusammenfassende Ausblick in Abschnitt 5 schließt den Artikel ab.

2 Der Entwicklungsprozess

Der Erfolg und die Mächtigkeit des beabsichtigten generativen Szenarios steht und fällt mit der Exaktheit und den Ausdrucksmöglichkeiten der Eingabespezifikationen - und hier zunächst der Architekturspezifikation. Diese basiert ihrerseits wiederum auf einer sorgfältigen und vollständigen Analyse der zu unterstützenden Anwendungsdomäne. Wichtigste Voraussetzung ist also die Domänenanalyse (bzw. ein „domain engineering"), die damit auch den ersten Schritt im gesamten Entwicklungsprozess darstellt. Methoden der Domänenanalyse sind ein rasch an Bedeutung gewinnendes Forschungsfeld (vgl. etwa [1]) und leisten einen erheblichen Beitrag zu einer problem- bzw. anwendungsnahen Softwareentwicklung, indem sie Fachkonzepte bereitstellen, die unabhängig von konkreten technologischen Aspekten sind. Insbesondere verlagern sich Betrachtungen, wie sie sich meist in einer Modellierungsphase finden, dabei zum Teil in die Aufbereitung eines Anwendungsbereichs, etwa die Gruppierung oder Zuordnung einzelner Merkmale zu Entitäten der Domäne.

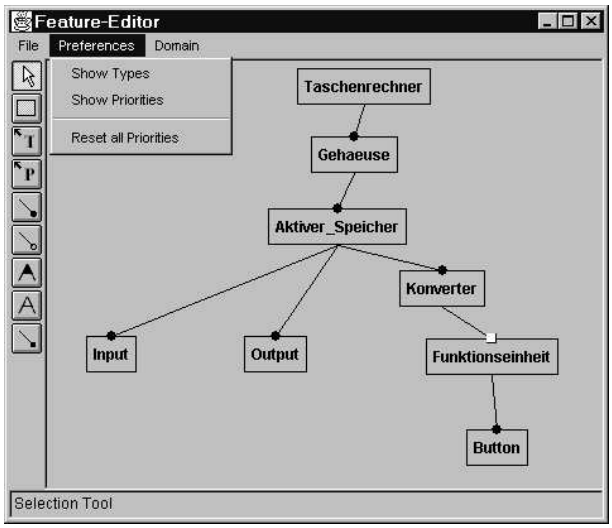

Abbildung 2. FeatureEditor zur Architekturspezifikation

Dies funktioniert allerdings erst dann zufriedenstellend, wenn eine entsprechende Formalisierung der Ergebnisse der Domänenanalyse erreicht werden kann. Hinsichtlich des Ziels einer generativen Konstruktion ist dabei zugleich das Zusammenspiel der Architektur mit den Spezifikationen bereitstehender Komponenten zu beachten. Der hier vorgestellte Ansatz setzt dabei nach erfolgter

Domänenanalyse ein und überlappt erst mit dem sich anschließenden Formalisierungsprozess und zwar in Form eines Werkzeugs zur Erstellung sogenannter *Merkmalsdiagramme* wie sie aus der „Feature-Oriented Domain Analysis" [2] bekannt sind. Dieses in Abbildung 2 illustrierte Werkzeug erlaubt zum einen die interaktive Erstellung von Merkmalsstrukturen („feature structures") [5], zum anderen die Erstellung der architekturellen Struktur einer zu entwickelnden Applikation. Die Entscheidung zugunsten solcher Merkmalsdiagramme anstelle von UML fiel hier aufgrund der wesentlich einfacheren und weniger mit technischen Details überfrachteten Darstellung, die damit die fachorientierte Kommunikation mit dem späteren Anwender / Kunden erleichtert. Andererseits verfolgt unser Ansatz ja gerade die Abstraktion von der Programmierebene, so dass viele Details gar nicht in Erscheinung treten müssen und die auftretenden Merkmale einer Anwendung eine direkte Zuordnung vorhandener Komponenten erlauben. Insbesondere entfällt damit die frühe Einschränkung des Variabilitätsgrades durch entsprechende UML-Konstrukte (vgl. [10]), so dass variantenreichere Produktlinien generiert werden können.

Der Diagrammeditor liefert die grafisch erstellten Spezifikationen in Form von DTD-Dateien sowie Merkmalsstrukturen in Form von Klassenspezifikationen. Während erstere die eigentliche Eingabe für den Generator darstellen, tragen letztere zur Typrepräsentation innerhalb des Typmanagers bei (vgl. Abschnitt 4). Die DTD-Dateien folgen dabei einem erweiterten CORBA Component Metamodell (CCM) [3], da eine konkrete Festlegung auf eine Implementationstechnik bzw. ein konkretes (programmiersprachliches) Komponentenmodell vermieden werden soll. Auf diese Weise erschließt sich dem Generator ein maximaler Pool von Komponenten. Abbildung 3 zeigt das resultierende Zusammenspiel der einzelnen Werkzeuge innerhalb der implementierten Entwicklungsumgebung.

2.1 Phasen der Generierung

Die Auswertung des Metamodells durch den Generator führt in einer ersten Phase (Spezifikationsphase) zur Auswahl passender Komponenten, die die Architekturspezifikation korrekt zu erfüllen vermögen. Die Spezifikationen der Komponenten bedienen sich dabei einer erweiterten Instanziierung des CCM-Metamodells in Form des in Abschnitt 4 vorgestellten Typmodells bzw. dessen XML-Deskriptoren. Auf diese Weise stellt sich das „Ausfüllen" der Architekturspezifikation für den Generator als ein konstruktives Parsing der XML-Spezifikationen dar, indem ein schrittweiser Test gegen die zugehörige DTD erfolgt[1].

Die zweite Arbeitsphase des Generators umfasst die Konfiguration der ausgewählten Komponenten, wobei diese Phase zeitlich mit der ersten verwoben ist, da die schrittweise Komponentenauswahl möglicherweise bereits von bestimmten Konfigurationen abhängt oder umgekehrt diese erfordert. Verbleibende Freiheitsgrade werden dem Anwendungsentwickler in Form einer angeleiteten Konfiguration („Wizard") angeboten.

[1] Dies stellt eine Umkehrung des üblichen Verhältnisses zwischen DTD und XML-Datei dar und erfordert einen speziellen Parser!

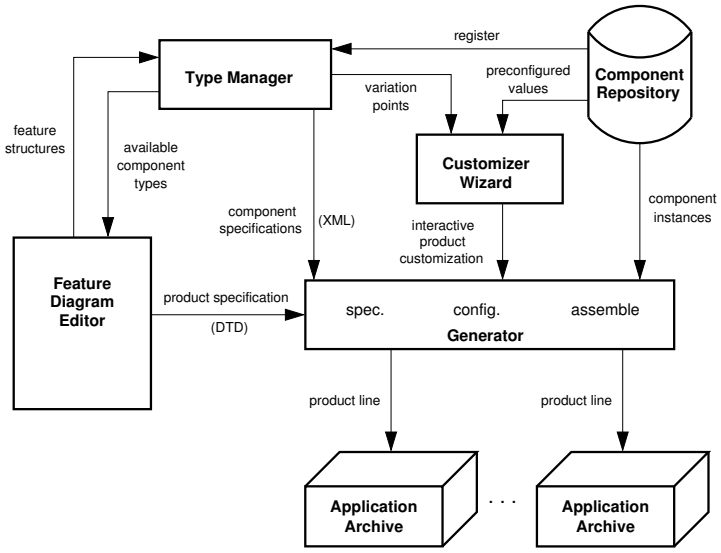

Abbildung 3. Prinzipsicht der generativen Entwicklungsumgebung

Ist dieses Customizing abgeschlossen (das unter Umständen auch zur Auswahl zusätzlicher Komponenten durch den Generator geführt hat), erstellt der Generator ein Anwendungsskelett zur Verknüpfung aller ausgewählten Komponenten gemäß den Architekturvorgaben. Die zur Verknüpfung benötigten Informationen können dabei vollständig den eingelesenen Komponentenspezifikationen entnommen werden, da diese alle Verbindungspunkte explizit beschreiben. Wurde bisher von „Komponentenauswahl“ gesprochen handelte es sich genau genommen um die Auswahl von Komponententypen.

Erst in der dritten Phase - der Konstruktionsphase - selektiert der Generator konkrete Komponenteninstanzen gemäß den im Skelett angegebenen Typen und setzt entsprechende Variationspunkte gemäß seinen Konfigurationsvorgaben. Die Konstruktion schließt mit dem „Verpacken“ aller Anwendungsbestandteile zu einem Anwendungsarchiv ab, um für die anschließenden Deployment- und Installationsphasen eine handhabbare Einheit bereitzustellen.

Die ebenfalls zum Komponentenmodell der OMG gehörenden Deployment- und Assembly-Deskriptoren erlauben die Festlegung bzw. automatisierte Unterstützung der Einsatzkonfiguration („deployment“) bzw. der Auslieferung und Installation der Anwendungsbestandteile in der Zielsystemumgebung. Die Einsatzkonfiguration kann dabei beispielsweise die Auswahl von Transaktionsmodi oder Sicherheitsattributen umfassen, während die Installation z.B. Informationen benötigt, welche programmiersprachlichen Klassen auf welchen Rechnerknoten vorliegen müssen. Unser Ansatz erlaubt somit nicht nur die server-zentrierte Komponentensicht, die das unterliegende CCM nahe legt, sondern auch die erwünschte „freie“ Komposition von Anwendungsbausteinen.

3 Verwandte Ansätze

Der Wunsch nach einer automatischen Softwareerstellung im Sinne einer Generierung ist nicht neu, sondern besitzt seine Anfänge spätestens in den Bemühungen der frühen KI, die die Vorstellung sich selbst programmierender Systeme u.Ä. hervorbrachte, diese visionären Ideen jedoch nicht umzusetzen vermochte. Weit pragmatischerer sind die neueren Ansätze, die etwa im Rahmen der Generischen (GP), der Adaptiven (AP) [12] und der Aspekt-orientierten (AOP) [11] sowie der Subjekt-orientierten Programmierung (SOP) [9] betrachtet werden. Gemeinsam ist solchen Ansätzen die Betonung der Entkopplung von strukturellen („funktionsblinde" Architektur) und verhaltensbezogenen („strukturblinder" Code) Systemeigenschaften, die getrennte Bearbeitung einzelner Systemaspekte („separation of concerns" [10]) und die fachliche Anwendungsnähe durch Betonung der Domänenanalyse.

Den Leitlinien „Trennung" und „Entkopplung" stehen dann automatische Mechanismen zur Rekombination der zunächst separat modellierten und eventuell programmierten Systemaspekte gegenüber. Diese Mechanismen haben dann einen entsprechenden transformativen oder generativen Charakter, um ein reales System zu produzieren. Ansätze wie GenVoca bzw. Jakarta [6] zielen dann explizit auf die Erzeugung eines problemspezifischen Generators, der dann wiederum das gewünschte Endprodukt erstellen kann.

Überhaupt sind diese Ansätze geprägt von der Vorstellung aus einer domänenspezifischen „Eingabesprache" - eventuell unter Kombination mit einer Konfigurationssprache - eine problembezogene Zielsprache zu erzeugen, in der dann die eigentliche Anwendung formuliert wird. Microsofts „Intentionelles Programmieren" (IP) [10] verdeutlicht diesen Trend besonders markant, indem die eigentlichen Ausdrucksmöglichkeiten der Sprache zur Systemprogrammierung (eigentlich besser: „-formulierung") völlig frei definierbar und damit beliebig problemnah werden. Alle Ansätze sind jedoch nach wie vor den „lines-of-code" sehr nahe und ihre Endprodukte besitzen wiederum keinen kompositorischen Charakter.

Die Komponentensicht stellt nun eine ideale Kombination mit den grundlegenden Ideen der angeführten Ansätze dar, indem sie klar abgegrenzte Bausteine bereitstellt, die zum einen direkt die Rolle einzelner Systemaspekte übernehmen können, zum anderen die notwendigen Mechanismen zur Verknüpfung und Konfiguration vereinfachen, da sie per se von der Codezeilenebene abstrahieren. Ein „Verweben" systemdurchdringender („horizontaler") Aspekte (z.B. Sicherheit) wie sie die AOP vorsieht, ist in der Praxis komplexer Anwendungssysteme beispielsweise schwierig zu automatisieren.

Nehmen wir jedoch eine Dienstleistungssicht ein, d.h. eine Komponente ist für genau diesen Aspekt „zuständig" und stellt ihn entsprechend den anderen Systembestandteilen bereit, kann die wünschenswerte Trennung der Belange aufrechterhalten werden bei gleichzeitiger Vereinfachung der Systemerstellung. Letztlich liegt diese Sicht Buskonzepten wie dem CORBA ORB mit angegliederten Diensten oder einem EJB-Container mit den Dienstfunktionen für die in ihm enthaltenen Komponenten ohnehin zugrunde.

Unser hier vorgestellter Ansatz ist also klar als ein Komponenten-basierter Generator positioniert, d.h. geht von der Zusammensetzung einer Softwareapplikation aus bereits vorhandenen Komponenten aus. Es steht weder die Erweiterbarkeit noch die Spezialisierung sprachlicher Ausdruckmittel im Vordergrund, sondern die automatisierte Konstruktion (in Abgrenzung zum Begriff der Programmierung) des gewünschten Endprodukts. Dann müssen allerdings an die Beschreibungen bzw. Spezifikationen der vorhandenen Bausteine hohe Ansprüche gestellt werden, um einem Generator ausreichende Informationen zur korrekten Anwendungserstellung bereitzustellen. Ferner sollte der Generator nicht auf einem bestimmten Komponentenmodell basieren, um möglichst flexibel und technologieunabhängig einsetzbar zu sein. Stattdessen liegt es nahe von entsprechenden Meta-Modellen für Komponenten auszugehen. Unser zugehöriges Typmodell wird im Folgenden vorgestellt.

4 Klassifikation von Komponenten

Komponenten-Metamodelle wie das der CORBA Components führen eine ganze Reihe von abstrakten Konstrukten und Informationen ein, die eine sinnvolle Obermenge zu den Eigenschaften konkreter Komponententechnologien bieten und somit die Definition von Komponententypen erlauben. Eine Klassifikation von Komponenten ist für den oben beschriebenen generativen Konstruktionsprozess unabdingbar, da nur sie Entscheidungskriterien während der Komponentenauswahl bzw. -suche des Generators bereitstellen kann. Eine Klassifikation durch Typisierung ist naheliegend, allerdings nicht unproblematisch: Der Generator muss sich auf (mindestens) zwei Aussagen beziehen können. Zum einen wann „passt“ ein Komponententyp in einen „Slot“ der Architekturspezifikation - wir bezeichnen dies als die Frage nach der *Konformität*, zum anderen wann kann eine Komponente mit einer anderen kombiniert werden - wir bezeichnen dies als die Frage nach der *Kompatibilität*. Modelle wie das CCM sehen zwar explizit einen „componenttype“ (in Form einer XML-Beschreibung) vor, beantworten jedoch nicht die Fragen nach potentieller Konformität bzw. Kompatibilität - diese beschränkt sich auf die Überprüfung der Namensgleichheit entsprechender XML-Elemente. Wir haben daher zum einen zunächst Mechanismen zur praktischen Verarbeitung des CCM-Metamodells in Form eines Typmanagements geschaffen, zum anderen das Modell erweitert, um eine flexible Klassifikation und zugehörige Vergleichsmöglichkeiten bereitstellen zu können.

Abbildung 4 zeigt zur Illustration Ausschnitte aus den ensprechenden XML-Repräsentationen einer Hexidezimal-Koverter-Komponente für einen Taschenrechner. Gezeigt sind Teile der Komponentenspezifikation selbst, ein Ereignis auf semantischer Ebene (Business Event) sowie eine Nachricht auf syntaktischer Ebene. Die Komponente selbst wird im gezeigten Fall lose und verteilt über das Java Message Queue System (JMS) an andere Bausteine gekoppelt. Die Modellbeschreibung und der Generator erlauben die Einbeziehung weitgehend beliebiger Technologien, insbesondere lokale wie verteilte.

```
<gencomponent>
<classname name="HexConverter"/>
<description>converts from hexadecimal to decimal</description>
<feature name="number system" value="hexadecimal"/>
</gencomponent>
<componentfeatures>
<corbacomponent>
  <repositoryid/>
  <transaction use="not-supported"/>
  <threading policy="serialize"/>
  <configurationcomplete set="true"/>
  <componentfeatures name="" repid= "F1">
    <ports>
      <emits emitsname="NumberCalculated" eventtype="JMS Event"
        eventname="NumberCalculated"/>
      <emits emitsname="NumberChanged" eventtype="JMS Event"
        eventname="NumberChanged"/>
      <consumes consumesname="NumberCalculated" eventtype="JMS Event"
        eventname="NumberCalculated"/>
      <consumes consumesname="NumberChanged" eventtype="JMS Event"
        eventname="NumberChanged"/>
    </ports>
  </componentfeatures>
</corbacomponent>
</componentfeatures>

<semBEType typename = "NumberChanged">
   <feature name = "NUMBER_CHANGED" datatype = "tuple_of_1" >
     <feature name = "NUMBER" datatype = "String" />
     </feature>
</semBEType>

<syntMessageType typename="NumberChanged">
   <OperationTypes>
      <OperationType typename="numberChanged">
         <parameter>
            <datatype type="NumberChangedEvent"></datatype>
         </parameter>
         <returnvalue></returnvalue>
      </OperationType>
   </OperationTypes>
   <CommunicationType value="Synchron"></CommunicationType>
   <CommunicationModelType value="MessageQueueModel">
   </CommunicationModelType>
</syntMessageType>
```

Abbildung 4. Illustration der XML-Repräsentationen des Typmodells

Die exakt definierte Typisierung von Komponenten stellt zudem eine ausgezeichnete Grundlage zur zukünftigen Realisierung von Qualitäts- und Veri-

fikationskriterien dar, da gegen die entsprechenden Typspezifikationen *getestet* werden kann.

4.1 Konzeption des Typmodells

Das Typmodell ist ebenfalls ein Metamodell, um eine Vielzahl konkreter Typsysteme für Komponenten abbilden zu können. Der Typ einer Komponente ist dabei als eine in drei Ebenen untergliederte Baumstruktur repräsentiert (vgl. Abb. 5). Er ist definiert über eine Anzahl von Mengen sogenannter „Business Events“ (BEs). Die Wahl dieser Bezeichnung rührt aus der gewünschten Anwendungsnähe der Komponentensicht her und stellt keine Einschränkung der Allgemeinheit des Typmodells dar.

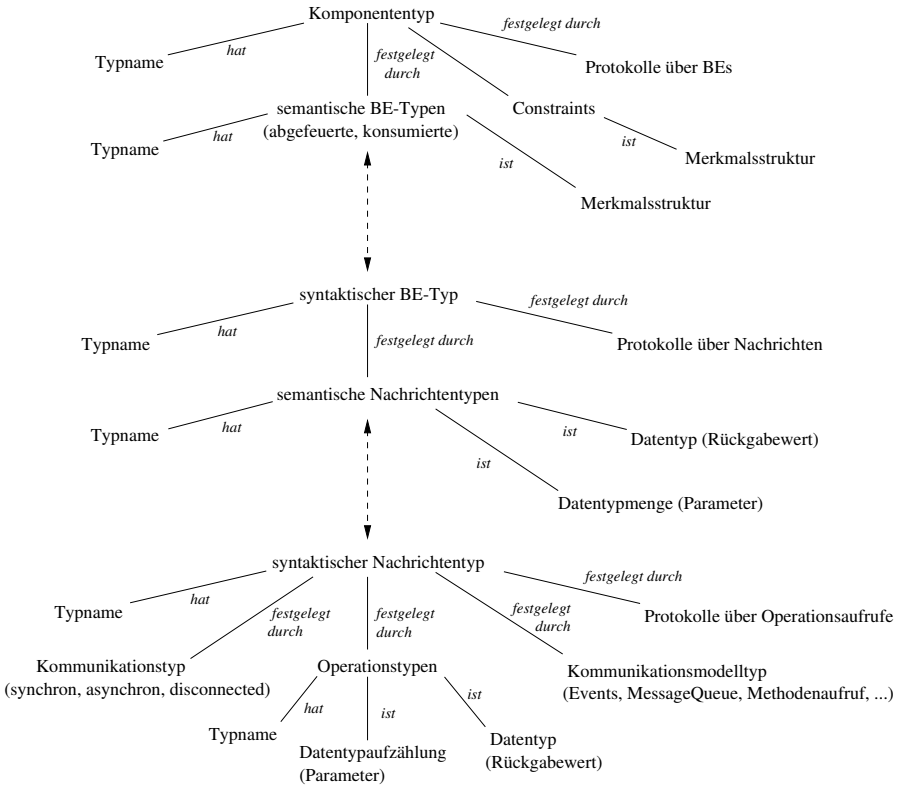

Abbildung 5. 3-stufiges Typmodell

Das Typmodell ist damit *interaktionsorientiert*, d.h. wir gehen davon aus, dass die an der Oberfläche einer Komponente beobachtbaren Vorgänge, die Komponente vollständig definieren und im Rahmen einer Anwendungsdomäne Klassen von Komponenten gebildet werden können, die alle eben jene Vorgänge unter-

stützen. Wir bezeichnen daher diese Ebene des Typmodells auch als *semantische* Ebene. Diese Business Event Sicht korrespondiert zugleich mit den UML (2.0) Konzepten „Kollaboration“ und „Aktivität“ und ist in der Vorgehensweise z.B. auch vergleichbar mit der „Typbildung“ in Smalltalk in Form sogenannter Nachrichtenprotokolle. Ebenso geht sie auf natürliche Weise konform zu „Business Component“ -Konzepten [14].

Die BEs selbst drücken also zunächst nur eine Semantik im Sinne einer gewählten Anwendungsdomäne aus, bedürfen zu ihrer Materialisation jedoch einer Syntax, die in Form der zweiten Ebene - der Nachrichtenebene - definiert ist. Jedes Business Event setzt sich aus einer Menge von Nachrichten zusammen, deren Austausch zwischen Kollaborationspartnern zur Realisierung der gewünschten Anwendungssemantik führt. Die Konkretisierung der Nachrichten erfolgt dann in Ebene drei in Form von Operationstypen, die letztlich der klassischen signaturbasierten Schnittstellenbeschreibung gleicht, allerdings nicht auf ein spezifisches Kommunikations- oder Technologiemodell festgelegt ist.

Alle drei Ebenen spezifizieren ihre jeweiligen Konstrukte zum einen strukturell über Merkmalsstrukturen, zum anderen verhaltensmäßig über als deterministische endliche Automaten (DEAs) repräsentierte Protokolle, um ein semantisch reiches Modell zu erhalten. Die Repräsentation als DEAs ist ausreichend ausdrucksstark, da das zu Grunde gelegte Komponentenmodell Nebenläufigkeit *innerhalb einer* Schnittstelle ausschliesst. Die Merkmalsstrukturen selbst können dabei auch Richtlinien in Form sogenannter Policies repräsentieren, die dann Randbedingungen über die Eigenschaften bzw. das Verhalten festlegen können. Wir verwenden hier nicht die Object Constraint Language (OCL) wie sie etwa die Catalysis-Notation zur semantischen Absicherung von Komponenten vorsieht, da unser eigener Policy-Formalismus [15] eine leichtere automatische Verarbeitung und insbesondere die Unifikation mehrerer Randbedingungen erlaubt, was wir für eine konsistenzerhaltende Komposition während der Generierung ausnutzen.

4.2 Typarten und -beziehungen

Die Einteilung der Typrepräsentation in mehrere Ebenen erlaubt zum einen getrennte Vergleiche syntaktischer und semantischer Natur und damit jeweils die Betrachtung nur einer problembezogenen Informationsmenge („Tun zwei Komponenten das gleiche?“, „Passen ihre Kommunikationsmodelle zueinander?“, etc.), zum anderen bildet sie die Grundlage für die Definition flexibler Typrelationen, indem alle drei Ebenen jeweils mehr oder weniger vollständig berücksichtigt werden.

So sprechen wir zunächst von verschiedenen Typarten. Typart-1 kennzeichnet dabei zunächst Typen für die der klassische Subtypbegriff einhergeht mit der Substitutionsfähigkeit des Supertypen durch den Subtyp. Zur Vermeidung von Vererbungsanomalien modellieren wir allerdings dabei explizit getrennt reduzierte Subtypen und erweiterte Subtypen - erweiterte Subtypen tragen im Falle der Substitution das Risiko semantischer Unverträglichkeiten und das Typmanagement generiert entsprechende Hinweise (vgl. etwa auch „join“ vs. „subtype“ in [7]).

Typart-2 bezeichnen wir als Verhaltenstyp. Er erlaubt eine (Unter-) Klassifikation von Komponenten die in einer Typart-1 Beziehung stehen durch Kennzeichnung unterschiedlichen Verhaltens. Hierzu wird auf die abgelegten Protokollspezifikationen und Constraints zurückgegriffen. Die Typart-3 schließlich wird als technischer Typ bezeichnet und kennzeichnet ansonsten gleiche Typen durch Unterscheidung ihrer Operationstypen und Kommunikationsmodelle. Diese Typarten können dann in verschiedene Beziehungen zueinander stehen, die jeweils das Auswertungsergebnis bei einer Kompatibilitäts- oder einer Konformitätsprüfung bestimmen.

Abbildung 6 zeigt exemplarisch das sich ergebende Assoziationsgeflecht[2] des Typmanagers am Beispiel einer ableitbaren Subtypbeziehung zwischen einer „Sparkassen" - und einer „Bank" -Komponente.

4.3 Typmanagement

Der praktische Einsatz des Typmodells erfolgt über eine als Typmanager bezeichnete Komponente, die die Registrierung entsprechender Komponententypen erlaubt. Die vom Typmanager bereitgestellten Funktionen gliedern sich dabei grob in vier Kategorien: Die eigentlichen Typvergleiche, das Anzeigen der Wissensbasis (z.B. manuelle Suche, Browsing), die (automatische) Suche nach bestimmten Komponententypen, sowie die Verwaltung von Domänen, da die Wissenbasis selbst (hierarchisch) in Domänen separiert ist gemäß der Gültigkeit (Sinnhaftigkeit) einzelner Komponententypen für bestimmte Anwendungsdomänen. Abbildung 7 illustriert die als Typexplorer bezeichnete Benutzeroberfläche des Typmanagers während eines Vergleichs der schon oben dargestellten Bank-, Sparkassen- und Kundentypen.

Adaption und Rollen Eine bemerkenswerte Eigenschaft des Typmanagers ist die Fähigkeit zur Bereitstellung von Komponentenadaptern, im Falle der Kopplung zunächst inkompatibler Komponenten. Hierzu können einerseits explizit als Adapter gekennzeichnete manuell erstellte Komponenten registriert werden, die dann bei Bedarf automatisch korrekt zugeordnet und verwendet werden. Andererseits ist der Typmanager in der Lage automatisch Adapter für alle syntaktischen Ebenen der Typhierarchie zu generieren. Hierfür haben wir einen Adaptionsalgorithmus in Anlehnung an [16] implementiert. Die für diesen notwendigen Äquivalenzregeln leitet der Typmanager aus dem Assoziationsgeflecht der zu adaptierenden Komponenten ab. Die resultierenden Adaptertypen sind konzeptuell zunächst technologieunabhängig; ihre konkreten Instanzen sind dann allerdings jeweils spezifisch für eine konkrete Komponententechnologie (z.Z. für JavaBean- und COM-Komponenten).

Der Einsatz solcher Adapter geht auf unsere Arbeiten [8] zurück (dort als Interzeptoren bezeichnet) und harmoniert mit der heutigen Forderung der Komponentendiskussion nach aktiven Konnektoren [7, 12] und subjektivistischen -

[2] Die entsprechende Struktur ist analog zu den *TopicMaps* des Standard ISO/IEC 13250:1999.

Abbildung 6. Assoziationsgeflecht einer Subtypbeziehung im Typmanager

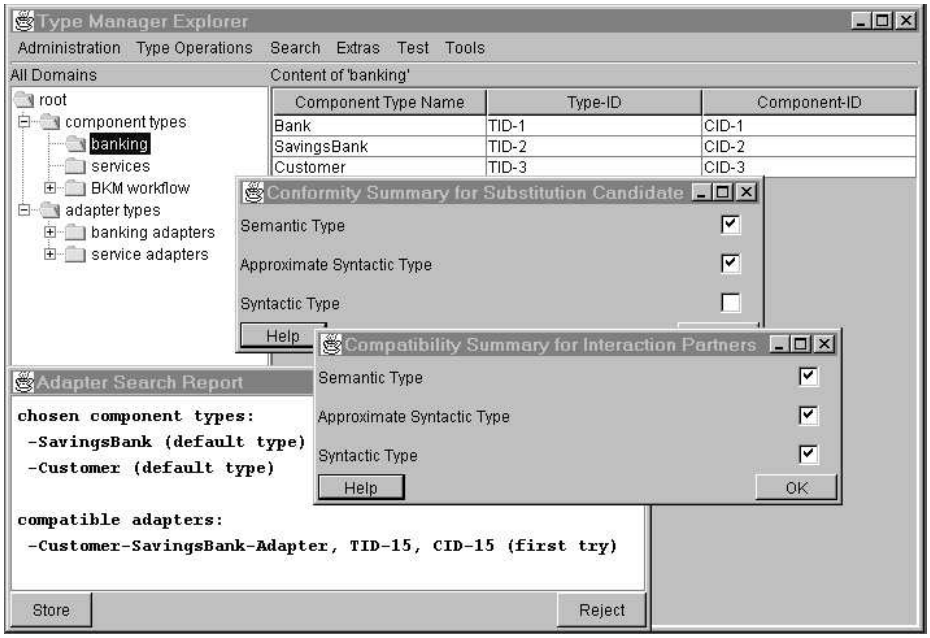

Abbildung 7. Interaktive Benutzung des Typmanagers

d.h. rollen- und situationsbezogen dynamisch generierten - Schnittstellen [6]. Der Bedeutung bzw. Mächtigkeit von Rollen- bzw. Personality-Konzepten [4] für die Modellierung wird zunehmend Aufmerksamkeit zuteil, da sie zu kompakteren, flexibleren und wiederverwendbareren Modellen führen. Wir tragen diesem Umstand in unserem Typmodell durch die explizite Definition von Rollen Rechnung: Die oberste Ebene des Typmodells kann mehr als eine semantische Spezifikation umfassen. Im Ergebnis kann eine Komponente eines solches Typ dann verschiedene Rollen durch die Kennzeichnung der jeweils gültigen semantischen Spezifikation annehmen bzw. in ihnen in der Anwendungsdomäne auftreten.

5 Resümee und Ausblick

Wir haben in diesem Beitrag einige Aspekte der Konzeption und Implementation einer Umgebung zur Etablierung einer Generativen Softwarekonstruktion aus vorgefertigten Softwarekomponenten vorgestellt. Herzstück des generativen Prozesses ist dabei der - konstruktive - Test der Modelle zur Verfügung stehender Einzelbausteine gegen ein zugehöriges Architekurmodell, dessen Ausdrucksmittel direkt zur Spezifikation einer Anwendungsarchitektur verwendet werden. Das dabei verwendete Typ-Metamodell setzt auf den MOF-Beschreibungen des „CORBA Component Models“ auf, erweitert dieses jedoch um exakt definierte, semantisch reiche Typbeziehungen, um den Anforderungen einer automatischen Verarbeitung durch einen Generator gerecht zu werden.

Neben dem Einsatz des Typmanagements im Rahmen einer Softwarekonstruktion, arbeiten wir derzeit an dessen Integration mit einem erweiterten JINI-Lookup-Service (LUS), um auch auf der Ebene systemnaher Dienste komplexere Such- und Auswahlstrategien realisieren zu können. Die Universialität des vorgestellten Typmodells zeigt sich auch in unseren aktuellen Bemühungen zur Realisierung eines E-Publishing-Portals, das einem Benutzer die Zusammenstellung unterschiedlicher Anwendungsdienste aus dem Publikationswesen zu funktionalen Ketten im Sinne von Gesamtprozessen unter Beteiligung mehrerer Dienstanbieter ermöglicht. Die dafür notwendige Dienstklassifikation erfolgt wiederum mit Hilfe des vorgestellten Typmanagements.

Danksagung

Wir danken der Deutschen Forschungsgemeinschaft für die Unterstützung der hier vorgestellten Arbeiten zum „Basic Research Component-based Develoment Environment“ (BARCODE) im Rahmen des Projekttitels DFG 1061-2 („Dynamically Configurable Software“).

Literatur

1. http://www.sei.cmu.edu/domain-engineering.
2. http://www.sei.cmu.edu/domain-engineering/FODA_bib_ref.html.
3. http://www.omg.org: Dokument orbos/99-07-01.
4. http://www.ccs.neu.edu/home/lblando/personalities.
5. B. Carpenter. *The logic of typed feature structures.* Cambridge Univ. Press, 1992.
6. D. Batory. http://www.cs.utexas.edu/users/schwartz.
7. D.F. D'Souza and A.C. Wills. *Objects, Components, and Frameworks with UML.* Addison Wesley, 1999.
8. F. Griffel, K. Müller–Jones, and W. Lamersdorf. Komponentenbasierte Entwicklung interoperabler Software auf heterogenen Middleware–Plattformen. In H. C. Mayr, editor, *Beherrschung von Informationssystemen, Tagungsband der Informatik'96*, number 88 in Schriftenreihe der Österreichischen Computer Gesellschaft, pages 327–342. R. Oldenbourg, 1996.
9. W. Harrison and H. Ossher. http://www.research.ibm.com/sop.
10. K. Czarnecki and U.W. Eisenecker. *Generative Programming – Methods, Tools, and Applications.* Addison Wesley, 2000.
11. G. Kiczales. http://www.parc.xerox.com/csl/projects/aop.
12. K. Lieberherr. http://www.ccs.neu.edu/research/demeter.
13. O. Nierstrasz and D. Tsichritzis, editors. *Object–Oriented Software Composition.* The Object–Oriented Series. Prentice–Hall International Ltd., 1995.
14. P. Herzum and O. Sims. *Business Component Factory.* Wiley, 2000.
15. M. T. Tu, F. Griffel, M. Merz, and W. Lamersdorf. Generic Policy Management for Open Service Markets. In H. König, K. Geihs, and T. Preuß, editors, *Distributed Applications and Interoperable Systems, DAIS'97 Cottbus, Germany*, pages 211–222. IFIP, Chapman & Hall, Oktober 1997.
16. D. M. Yellin and R. E. Strom. Collaboration Specifications and Component Adaptors. Forschungsbericht RC 20054 (88710) 5/4/97, T.J. Watson Research Center, Mai 1995.