Repository-Dienste für die modellbasierte Entwicklung

Udo Kelter
Fachbereich Elektrotechnik und Informatik
Universität Siegen
kelter@informatik.uni-siegen.de

Abstract: Viele langlebige Systeme existieren in mehreren Varianten, die parallel weiterentwickelt werden müssen. Hierzu werden unterstützende Repository-Dienste benötigt, neben dem klassischen Mischen von Dokumenten auch Historienanalysen. Bei Systemen, die mit modellbasierten Methoden (weiter-) entwickelt werden, ergeben sich besondere Probleme, weil auch die Modelle mitversioniert und in die Analysen einbezogen werden müssen. Dieser Workshopbeitrag beschreibt dieses Problemfeld genauer und skizziert Lösungsansätze, die im Rahmen des SiDiff-Projekts entwickelt wurden.

1 Einführung

Große Systeme, die über viele Jahre existieren und während dieser Zeit immer wieder erweitert und umstrukturiert werden, werfen große Wartungsprobleme auf. Gewartet werden müssen typischerweise nicht nur die aktuelle Version des Systems, sondern auch ältere Releases, die noch bei Kunden im Einsatz sind. Neben historisch entstandenen Varianten können auch aus produktbezogenen Gründen Varianten entstehen. Im Endeffekt müssen mehrere Systemvarianten, die zwar erhebliche Gemeinsamkeiten, aber auch signifikante Unterschiede aufweisen können, parallel weiterentwickelt werden.

Ein heute stark favorisierter Ansatz zur Reduktion der Entwicklungs- und Wartungsaufwände ist die modellbasierte Systementwicklung [10], also die Generierung von Teilen des Systems aus Modellen. In diesem Kontext werden auch domänenspezifische Sprachen diskutiert, die meist eine vorhandene Modellierungssprache erweitern[1]. Insb. die adaptive Wartung infolge von neuen Versionen unterliegender Systeme (DBMS, BS u.a.) kann so erleichtert werden. Die eingesetzten Modelltypen hängen natürlich von den Merkmalen der Anwendungsdomäne ab. Häufig eingesetzt werden Datenmodelle, meist eine Variante der UML-Klassendiagramme. Für eingebettete Systeme werden vor allem Zustandsmodelle und Funktions- bzw. Aktivitätsmodelle eingesetzt. Die Anwendungsdomänen unterscheiden sich auch darin, wie große Teile eines Systems aus den Modellen generiert werden können: im Idealfall alles, oft müssen jedoch umfangreiche Teile manuell entwickelt werden.

[1] Die Frage, ob die notwendigen komplexen Generierungsframeworks mehr Wartungsprobleme schaffen als lösen, ist für dieses Papier nicht relevant. Festgehalten werden kann, daß domänenspezifische Sprachen die hier diskutierten technologische Herausforderungen eher vergrößern.

Die modellbasierte Systementwicklung hat die Konsequenz, daß die Modelle zum integralen Teil der Software werden. Die Modelle müssen immer völlig konsistent sein mit manuell entwickeltem Code, Konfigurationsdaten und sonstigen Ressourcen, die oft in XML-Dateien gespeichert werden. Alle zusammengehörigen Dokumente – hierzu gehören natürlich auch Testdaten, Dokumentationen und sonstige Begleitdokumente – müssen gemeinsam versioniert werden.

Die Versionierung von Software wird klassischerweise durch Repository-Systeme wie CVS oder SVN unterstützt. Diese sind allerdings für Texte ohne spezielle Struktur konzipiert worden und arbeiten mit strukturierten Dokumenten, namentlich Modellen, nicht zufriedenstellend. Im Kern sind die üblichen Repository-Dienste auch für Modelle erforderlich; Systeme, die dies leisten, bezeichnen wir i.f. als **Modell-Repositories**. Modell-Repositories müssen natürlich nicht nur Modelle, sondern auch beliebige andere Dokumente integriert mitverwalten können.

Wir konzentrieren uns i.f. auf solche Modell-Repositories, die Funktionen anbieten, die die Weiterentwicklung langlebiger Software mit langen Versionshistorien unterstützen.

2 Modell-Repositories

Hauptfunktionen von Modell-Repositories. Zur Unterstützung von Entwicklungsprozessen werden mit hoher Priorität folgende Dienste bzw. Funktionen von Repositories benötigt. Diese bauen auf Basisfunktionen zur Verwaltung von Versionsgraphen und anderer administrativer Daten auf. Man kann alle Funktionen in einem einzigen System realisieren oder einige in Form autarker Analysewerkzeuge, solche Implementierungsentscheidungen interessieren uns an dieser Stelle nicht.

1. das *Vergleichen und Mischen* von Modellen: In großen Projekten ist arbeitsteilige Entwicklung und das gemeinsame Bearbeiten von Dokumenten unvermeidlich. Das exklusive Sperren von Dokumenten führt zu praktischen Problemen, daher hat sich in der Praxis weitgehend die Strategie durchgesetzt, nicht zu sperren, sondern automatisch zu mischen (3-Wege-Mischen).
2. *Suchfunktionen*, die "gleiche" Modellelemente oder -Fragmente in unterschiedlichen Varianten finden: Wir unterstellen hier, daß bei langlebigen Systemen mehrere Releases bei Anwendern installiert sind. Wenn in irgendeinem der Releases ein kritischer Fehler gefunden wird, muß für alle anderen Releases geprüft werden, ob der Fehler dort auch auftritt. In einen Fehler sind in der Regel mehrere Modellelemente involviert. Dieses Modellfragment kann in anderen Releases identisch oder in modifizierter Form auftreten. In diesem Zusammenhang ist es auch wichtig zu wissen, in welcher Version – die nicht notwendig ein Release ist – der Fehler entstanden ist und in welchem Kontext die damaligen Änderungen standen. Generell ist es für das Verständnis des Systems oft hilfreich zu wissen, welche Teile warum und zusammen mit welchen anderen Teilen eingeführt wurden und wie diese Systemteile weiterentwickelt wurden.
3. *historienbasierte Modellanalysen*: Ein System, das oft geändert wird, degeneriert insofern, als die Qualität der Entwurfsentscheidungen immer suboptimaler wird. Im Endef-

fekt sinkt die Verstehbarkeit des Systems und damit auch die Wartbarkeit, ferner wird die Einschätzung, wie aufwendig bzw. risikoreich weitere Änderungen sind, immer schwieriger. Der Qualitätsverlust muß durch Maßnahmen zur Strukturverbesserung kompensiert werden. Für die Planung solcher Reengineering-Maßnahmen benötigt man Analysefunktionen, die kritischsten Defizite des Systems zu finden helfen. Derartige Analysefunktionen werden auch im Rahmen des Softwarequalitätsmanagements für die Bewertung von Entwicklungsprozessen, zur Vorbereitung von Audits etc. benötigt.

Ein Referenzmodell für Repository-Dienste. Die vorstehenden Funktionen werden im Prinzip direkt von Entwicklern genutzt. Intern weisen sie begriffliche Überschneidungen und Abhängigkeiten auf. Es liegt nahe, ein Repository-System in dementsprechende Subsysteme zu zerlegen. Im folgenden Referenzmodell werden die konzeptuellen Abhängigkeiten und mit den Konzepten direkt korrespondierende Dienste in Form von Schichten dargestellt:

Schicht 0: Dokumentverwaltung

Diese Schicht beinhaltet Dienste zur Speicherung von Dokumenten beliebigen Typs. Eingeschlossen ist die Verwaltung von Nachfolger-Beziehungen zwischen Revisionen, Benutzern und sonstigen administrativen Daten. Diese Schicht wird man in der Regel durch ein etabliertes Repository-Produkt realisieren.

Schicht 1: Differenz- und Ähnlichkeitsberechnung von Modellen

Dieser Schicht liegen Begriffsdefinitionen für die Ähnlichkeit von einzelnen Modellelementen bzw. Modellfragmenten zugrunde. Diese Definitionen gehen direkt in die Berechnung von Modelldifferenzen ein. Allerdings gilt bei manchen Ähnlichkeitsbegriffen auch die Umkehrung. Daher kann man die Berechnung von Ähnlichkeiten und Differenzen nicht trennen. Aus Dokumentdifferenzen kann man direkt Differenzmetriken ableiten, die typischerweise in der Differenz angegebene Korrespondenzen zwischen Modellelementen oder Änderungsoperationen zählen, ggf. gewichtet und/oder selektiert.

Im Gegensatz zu Schicht 0 ist diese Schicht *abhängig vom Modelltyp*, d.h. pro Modelltyp sind eigene Implementierungen oder zumindest Anpassungen erforderlich. Dies gilt auch für die aufbauenden Schichten.

Schicht 2a: Mischfunktionen

Eine Mischung basiert in der Regel auf einer vorher bestimmten Differenz, da die gemeinsamen Teile nur einmal in das Ergebnis übernommen werden. Die speziellen Teile der zu mischenden Dokumente können unverträglich sein. Zentrale Begriffe dieser Ebene sind daher Konflikte und Strategien zur Konfliktbehandlung.

Schicht 2b: Historienanalysen

Diese Schicht realisiert die oben erwähnten Suchfunktionen und historienbasierte Modellanalysen. Sie baut direkt auf Schicht 1 auf und liegt daher parallel zu den Mischfunktionen. Neben der Definition von Suchfunktionen sind hier Verfahren zur Vorverarbeitung und Indexierung von Versionshistorien angeordnet.

3 Technologische Herausforderungen

Dieser Abschnitt diskutiert den Stand der Technik in den vorstehende benannten Schichten des Referenzmodells. Auf Schicht 0 gehen wir nicht ein, denn hier ist etablierte Technologie verfügbar.

3.1 Differenzberechnung

Die entscheidende interne Funktion ist hier Bestimmung korrespondierender Dokumentteile. Für textuelle Dokumente sind hinreichend gute und effiziente Algorithmen bekannt, für graphstrukturierte Modelle ist das Problem deutlich komplizierter. Persistente Darstellungen sind keine geeignete Basis, sondern nur abstrakte Syntaxbäume. Besonders schwierig ist die Bestimmung von Korrespondenzen bei Modelltypen, in denen wichtige Modellelemente keine markanten lokalen Eigenschaften haben, sondern deren Nachbarschaft entscheidend ist. Ein zusätzliches Problem bei langlebigen Systemen sind neue Sprachversionen, die zu veränderten Metamodellen führen.

Der in [6] publizierte Vergleich von Algorithmen zeigt die Spannbreite der aktuell bekannten Lösungen und die jeweiligen Kompromisse auf:

- Verbreitet sind Verfahren, die auf Basis persistenter Identifizierer von Modellelementen arbeiten. Sie sind sehr effizient und leicht implementierbar, basieren aber auf sehr einschränkenden Voraussetzungen und Annahmen, die bei langlebigen, großen Systemen praktisch nicht erfüllbar sind. Sie bieten wenig bzw. keine Unterstützung für die Konfliktbehandlung und ähnlichkeitsbasierte Suchverfahren, weil die Semantik der Dokumenttypen nicht bekannt ist.
- Auf der anderen Seite stehen dedizierte Algorithmen für einzelne Sprachen bzw. Dokumenttypen, die besonders hochwertige Vergleichs- bzw. Mischergebnisse liefern, dafür aber relativ ineffizient sind und einen sehr hohen (um nicht zu sagen prohibitiven) Implementierungsaufwand verursachen, weil sie für jeden Modelltyp weitgehend neu entwickelt werden müssen.
- Frameworks wie das System SiDiff [5] zielen mittels einer “Sprache” zur Spezifikation von Ähnlichkeiten auf einen tragfähigen Kompromiß zwischen Laufzeiteffizienz, Qualität der Ergebnisse und Implementierungsaufwand der Algorithmen.

Qualitativ gute Algorithmen sind aktuell nur verfügbar für Klassendiagramme (Datenmodelle) in diversen Varianten, insb. für reverse engineerte Java-Quellprogramme, ferner für einfachere Varianten von Aktivitätsdiagramme und Zustandsautomaten. Für andere Modelltypen und domänenspezifischen Sprachen ist wenig oder nichts verfügbar. Weiterer Forschungs- und Entwickungsbedarf besteht hinsichtlich der Gestaltung der Metamodelle, der Qualitätsbeurteilung bzw. Optimierung von Differenzen und der Evolution der Metamodelle.

3.2 Mischfunktionen

Beim Mischen von Dokumenten können Fehler entstehen, und zwar hinsichtlich kontextfreier bzw. kontextsensitiver Syntax, Programmierstil und Semantik. Paare von Modellteilen (bzw. die sie erzeugenden Änderungen), die solche Fehler erzeugen, stehen in Konflikt zueinander. Mischfunktionen haben daher zwei wesentliche Teilfunktionen, nämlich Konflikterkennung und Konfliktbehandlung, also Mischentscheidungen. Beide Teilfunktionen hängen von der Semantik des Modelltyps ab und sind schwieriger zu realisieren, wenn

- Modelle dieses Typs komplexe Konsistenzkriterien aufweisen und Modelleditoren nur Modelle verarbeiten können, die einen hohen Korrektheitsgrad einhalten;
- eventuelle falsch positive Mischentscheidungen von nachfolgenden Entwicklungsschritte nicht oder nur mit hohem Aufwand erkannt werden.

Der Stand der Technik kann hier als rudimentär bezeichnet werden. Für viele Modelltypen gibt es keine Mischwerkzeuge, viele vorhandene Mischwerkzeuge unterstützen nur das 2-Wege-Mischen auf Syntaxbaumsdarstellungen und bieten nur sehr wenig Unterstützung.

3.3 Suchfunktionen

Suchfunktionen verallgemeinern die paarweise Ähnlichkeit, die schon beim Vergleichen von zwei Modellen benötigt wurde, auf beliebige Revisions- und Varianten-Ketten oder allgemeine Sammlungen von Modellen. Die Existenz bzw. Qualität von Suchfunktionen hängt daher direkt von den Funktionen ab, die Ähnlichkeiten berechnen. Einzelne Lösungsansätze werden in [2, 13, 14] diskutiert, von einem flächendeckenden Angebot praxiserprobter Lösungen ist man aber noch weit entfernt.

3.4 Analysefunktionen

Hauptzweck dieser Funktionen ist, die Qualität eines Systems zu beurteilen und insb. Systemteile (also u.a. Modellfragmente) zu finden, an denen Strukturverbesserungen notwendig sind. Systemteile mit einer geringen Größe sind in diesem Sinne leichter handhabbar, weil man in vielen Fällen die Defekte formal beschreiben kann (z.B. zu große Klassen oder Zyklen in benutzt-Beziehungen); auf dieser Basis kann man Suchverfahren implementieren, die die entsprechenden Stellen finden. Ferner ist aufgrund der geringen Größe der Aufwand für die Reparatur gering, namentlich wenn sie durch Refactorings, ggf. in Verbindung mit Design-Patterns, z.T. automatisierbar ist bzw. durch Werkzeuge unterstützt wird.

Strukturverbesserungen in größeren Systemteilen werfen deutlich mehr Probleme auf: die Definition, wann ein Defekt vorliegt, ist nicht formalisierbar, und vielfach wird nicht alleine der Zustand einer Version zur Beurteilung herangezogen, sondern Merkmale der

Änderungshistorie, d.h. es wird vom Entwicklungsprozeß auf die Qualität des Produkts geschlossen. Beispielsweise sind Defizite in Systemteilen, in denen wiederholt größere Änderungen stattfanden, wahrscheinlicher. Das Auffinden von suspekten Systemteilen ist eher als ein Information-Retrieval-Problem anzusehen, bei dem es darum geht, unter den vielen möglichen Verbesserungsmaßnahmen diejenigen mit dem größten Nutzen und den geringsten Kosten herauszufinden. Wegen der höheren Umbaukosten können nämlich i.d.R. nur wenige derartige Maßnahmen durchgeführt werden.

Visualisierung von Historien. Die geforderten Analysen ganzer Versionshistorien stehen vor dem Problem der Informationsüberflutung: einzelne Versionen sind i.d.R. schon sehr umfangreich, das Datenvolumen steigt infolge der Versionen um 1 - 2 Größenordnungen.

Viele Analyseverfahren nutzen daher Metriken, einzelne Versionen werden also nicht mehr in allen Details dargestellt, sondern auf ihre Metrikwerte reduziert. "Auffällige" Versionen bzw. Vorkommnisse in der Versionshistorie können nur noch anhand der Metrikwerte bzw. der numerischen Differenzen der Metrikwerte erkannt werden.

Auf Metriken basieren auch fast alle Methoden zur Visualisierung von Historien. Die bekannten Methoden zur Visualisierung von großen naturwissenschaftlichen Datenmengen versagen aber bei Modellen weitgehend, weil hier die Grundstruktur des Datenraums nicht ein homogenes 3D-Gitter ist, sondern durch die wesentlich komplizierteren Strukturen in Modellen geprägt ist. Es sind diverse Vorschläge für 2- oder 3-dimensionale Visualisierungen von Versionshistorien gemacht worden, die teilweise auf einer 2-dimensionalen Darstellung einzelner Versionen basieren, u.a. die Evolution Matrix [7], die auf polymetrischen Sichten [8] basiert, Evo Spaces [16], die sich optisch an Stadtbilder anlehnen, Gevol [3], das Evolution Radar [1] und weitere Systeme.

Ein genereller Nachteil der vorstehenden Ansätze ist, daß die gewohnte graphische Darstellung der Modelle, in der die Systemstrukturen gut dargestellt werden, nicht mehr eingesetzt wird. Wegen der geometrischen Eigenschaften ist es sogar prinzipiell fraglich, ob man die gewohnten Darstellungsformen überhaupt für Historien einsetzen kann; sie stoßen bei großen Modellen ohnehin an ihre Grenzen und sind nicht für die Darstellung von Historien konzipiert worden. Veränderungen an den Strukturen eines Systems zählen indes zu den interessantesten Veränderungen.

4 Lösungsansätze zur Visualisierung von Modell-Historien

Dieser Abschnitt skizziert einige Lösungsansätze, die für die Visualisierung von Modell-Historien im Kontext des SiDiff-Projekts [11] entwickelt wurden.

4.1 Metriken von Differenzen statt Differenzen von Metriken

Übliche Metriken für Modelle sind Zählungen von Strukturelementen, z.B. die Zahl der Attribute einer Klasse in einem Klassendiagramm oder die Zahl der ausgehenden Transitionen eines Zustands in einem Zustandsmodell. Wenn nun die ein Attribut durch ein anderes ersetzt wird oder eine Transition durch eine andere, ändern sich die Metrikwerte nicht, obwohl signifikante Änderungen stattgefunden haben. Anders gesagt sind die numerischen Differenzen der Metrikwerte nur ein unzuverlässiger Indikator für den Umfang der Änderungen.

Die naheliegende Lösung besteht darin, zunächst eine vollständige (korrekte) Differenz [12] zwischen den Versionen zu berechnen. Aus dieser Differenz geht hervor, welche Editieroperationen zum Nachvollziehen der Veränderungen notwendig sind. Metriken, die sich auf Differenzen beziehen und z.B. die darin enthaltenen Operationen zählen, bezeichnen wir als **Differenzmetriken**. Differenzmetriken sind offensichtlich viel genauere Indikatoren für den Umfang der Änderungen als Differenzen von Metrikwerten.

Differenzmetriken haben den Vorteil, daß Typen von Änderungen hinsichtlich ihres Risikos kategorisiert werden können. Ferner können tabellarische oder graphische Darstellungen von Änderungshistorien einzelne Metriken isoliert darstellen (m.a.W. sollten Analysewerkzeuge dies erlauben).

Differenzmetriken und Werkzeuge, die diese unterstützen, müssen spezifisch für einzelne Modelltypen entwickelt werden, da jeder Modelltyp eigene Editieroperationen und Konsistenzkriterien hat. Das reine Graphiksystem eines Werkzeugs kann weitgehend unabhängig von den Modelltypen entwickelt werden (s. z.B. [4]), muß also nicht für jeden Modelltyp neu entwickelt werden. Das relevanteste Problem ist auch hier wieder die Differenzberechnung (vgl. Abschnitt 3.1).

4.2 3D-Darstellungen und Animation

3-dimensionale Darstellungen von Versionshistorien können grob eingeteilt werden in solche, die Strukturen der Modelle direkt anzeigen, auf dieser Basis natürlich auch Veränderungen der Strukturen, und andere, die i.w. nur Metrikwerte anzeigen.

Anzeige von Modellstrukturen. Ein Beispiel für die erste Kategorie ist der StructureChangesView im Werkzeug Evolver [4, 15], s. Bild 1. Diese Ansicht adressiert vor allem strukturelle Änderungen, z.B. wenn Klassen ihre Assoziationen zu anderen Klassen ändern. Die Darstellung besteht aus mehreren hintereinanderliegenden "Scheiben", von denen jede eine Version darstellt. Jede Versionsdarstellung besteht aus Würfeln, die z.B. Klassen eines Klassendiagramms oder Zustände eines Zustandsdiagramms repräsentieren. Linien zwischen diesen Würfeln stellen Beziehungen, Transitionen o.ä. dar. Die Grunddarstellung kann mit diversen Metriken angereichert werden, z.B. kann je eine Metrik auf die Höhe, Breite und Farbe der Quader und die Dicke und Farbe der Linien abgebildet werden.

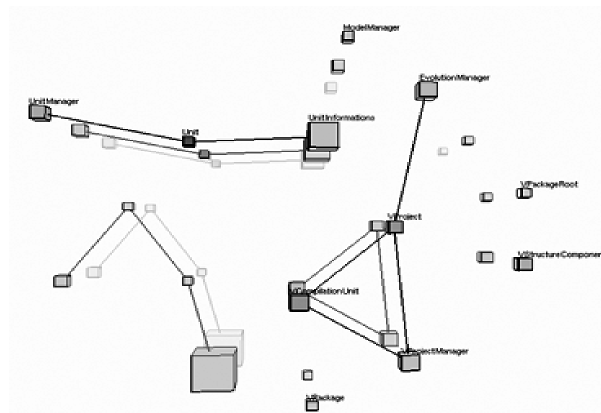

Abbildung 1: StructureChangesView im Werkzeug Evolver

Die Kameraposition kann beliebig zwar in Realzeit verändert werden ("Ego-Shooter"), normalerweise ist sie vor der vordersten Scheibe, so auch in Bild 1. Nur diese vorderste Version kann man also unbehindert erkennen, die dahinterliegenden Versionen werden von den davorliegenden teilweise verdeckt. Die vorneliegende Version ist die am meisten interessierende; um diese von den anderen optisch abzuheben, werden die hinteren Versionen immer transparenter gezeichnet. Die Knoten der Graphen werden vereinfacht dargestellt; würde man alle Details einer Klasse oder eines Zustands darstellen, wären diese nicht mehr lesbar.

Diese Darstellung ist gut geeignet, um bestimmte Typen von Änderungen zu erkennen, allerdings hat sie durchaus Limitationen:

- Die Zahl der angezeigten Entitäten muß klein sein. Für eine erste Gesamtübersicht über ein unbekanntes System ist diese Darstellung daher wenig geeignet, sie ist eher nach einer Einschränkung der Menge der zu untersuchenden Entitäten sinnvoll. Gute Möglichkeiten zur Selektion der angezeigten Entitäten sind daher sehr wichtig.
- Es kann nur eine beschränkte Zahl von Versionen sinnvoll angezeigt werden, ca. 5 Versionen sind noch gut erkennbar, ab ca. 10 Versionen wird die Darstellung trotz transparenter Darstellung unbrauchbar. Die Zahl der anzeigten Versionen sollte in Realzeit veränderbar sein, ebenso die vorne angezeigte Version. Wünschenswert ist ferner eine Funktion, die aus der gesamten Revisionskette besonders interessante Versionen anhand bestimmter Kriterien selektiert.

Metrikbasierte Darstellungen. Ein Beispiel für die zweite o.g. Kategorie ist der EvolutionView im Werkzeug Evolver, s. Bild 2. Diese zeigt für jede Version und jede Entität eine Säule. Die Höhe einer Säule ergibt sich anhand einer Metrik, angewandt auf diese Entität in dieser Version. Bei der Kameraposition, die in Bild 2 gewählt ist, verläuft die "Zeit" von hinten nach vorne. Von links nach rechts sind die angezeigten Entitäten angeordnet. In Bild 2 ist eine der Entitäten selektiert und alle zugeordneten Säulen über alle Versionen hinweg sind dunkler gezeichnet.

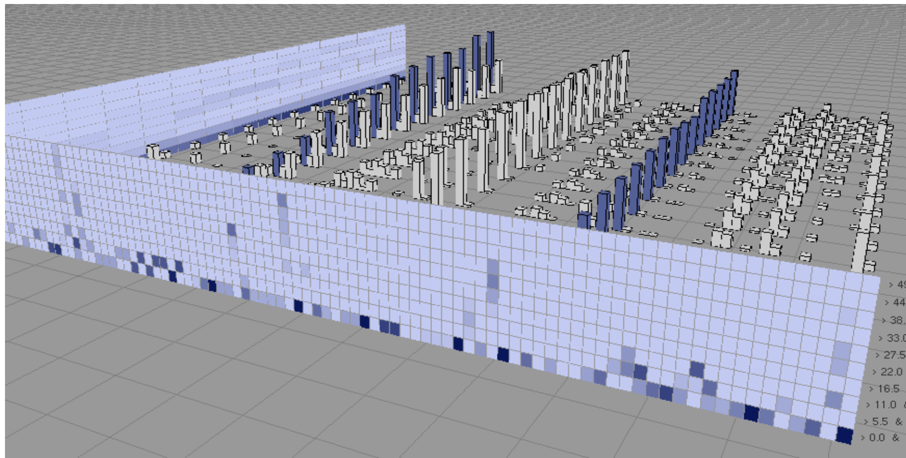

Abbildung 2: EvolutionView

An den Seiten befinden sind zusätzlich noch zwei Spektrographen: diese zeigen die Häufigkeitsverteilungen der Metrikwerte an. Hierzu wird der Wertebereich der Metrik in ca. 10 Intervalle eingeteilt, entsprechend viele kleine übereinanderliegende Rechtecke stehen auf einer Spektrographen-Wand zur Verfügung. Mit einer Farbcodierung wird die Zahl der Säulen, deren Größe im jeweiligen Intervall liegt, angezeigt. Die in Bild 2 vorne sichtbare Spektrographen-Wand zeigt die Häufigkeitsverteilungen der Metrikwerte *pro Entität* über die Systemversionen hinweg an, die seitlich sichtbare Wand die Häufigkeitsverteilungen *pro Systemversion*.

Animationen. Die beiden vorigen Darstellungsformen haben die zeitliche Reihenfolge, die durch eine Folge von Revisionen eines Systems entsteht, auf eine Dimension eines 3-dimensionalen graphischen Objekts abgebildet. Ein völlig anderer Ansatz besteht darin, die zeitliche Reihenfolge durch eine Animation darzustellen. Basis kann eine geeignete 2- oder 3-dimensionale Darstellung einzelner Versionen sein. Dies muß allerdings so gewählt sein, daß die Bewegungen gut erkennbar sind, also nicht zu geringfügig und nicht zu heftig sind.

Ein Beispiel findet sich im EvolutionView des Werkzeugs Evolver, s. Bild 3, das nur *ein* Bild einer Animation zeigt. Komplette Animationen können auf der WWW-Seite des Projekts [4] als Video angesehen werden. Entitäten werden hier durch Ellipsen dargestellt, die kreisförmig um einen Mittelpunkt angeordnet sind. Jeweils eine Metrik wird dargestellt durch (a) den Abstand der Ellipsen vom Mittelpunkt, (b) die Größe der Ellipse und (c) die Richtung der Längsachse der Ellipse. Änderungen in diesen Metriken führen zu entsprechenden mehr oder wenig schnellen Bewegungen der Ellipsen; die Fähigkeiten des menschlichen Sehsystems können hier besonders gut ausgenutzt werden.

Zusätzlich kann *eine* der Entitäten ausgewählt werden, deren Beziehungen zu anderen Entitäten dargestellt werden.

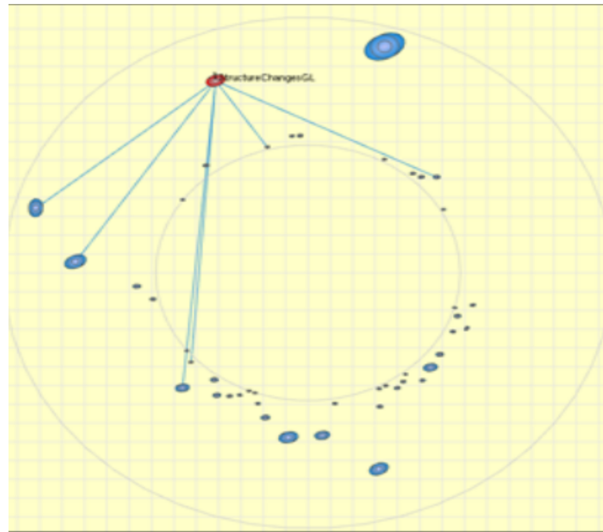

Abbildung 3: AnimationView

Integration und Evaluation der Darstellungsformen. Die vorstehenden Darstellungsformen haben jeweils eigene Stärken und Schwächen und müssen in einer integrierten Form verfügbar sein, in der man nahtlos zwischen diesen und weiteren Darstellungen, auch von Einzelversionen, wechseln kann.

Erste kontrollierte Evaluationen der oben vorgestellten Darstellungsformen zeigten, daß die EvolutionView aus Anwendersicht den größten Nutzen brachte, gefolgt von der AnimationView. Am schlechtesten schnitt die StructureChangesView ab.

5 Resümee

Die modellbasierte Systementwicklung erfordert für Modelle die gleichen Repository-Dienste, die man bei textuellen Dokumenten gewohnt ist. In der Praxis und hinsichtlich der technologischen Grundlagen ist man hiervon noch weit entfernt.

Das aus Sicht der Praxis drängendste Problem stellen Misch- und Vergleichswerkzeuge dar. Für viele Modelltypen sind keine brauchbaren Werkzeuge verfügbar, die verfügbaren Werkzeuge unterstützen nur das zeitaufwendige manuelle Mischen und bieten nur beschränkte Unterstützung bei der Konflikterkennung.

Analysefunktionen, die die Weiterentwicklung von Systemen mit vielen Revisionen bzw. Varianten unterstützen, existieren nur als Forschungsprototypen. Hier ist noch viel Raum für Verbesserungen vorhanden, sowohl bei der Entwicklung weiterer Darstellungsformen als auch bei der Integration verschiedener Darstellungsformen und Optimierung hinsichtlich der praktischen Nutzung.

Literatur

[1] d'Ambros, Marco; Lanza, Michele; Lungu, Mircea: Visualizing Integrated Logical Coupling Information; in: Proc. International Workshop on Mining Software Repositories 2006 (MSR 2006); ACM Press; 2006

[2] Bildhauer, Daniel; Horn, Tassilo; Ebert, Jürgen: Similarity-Driven Software Reuse; p.31-36 in: Proc. 2009 ICSE Workshop on Comparison and Versioning of Software Models; IEEE Catalog Number CFP0923G; 2009

[3] Collberg, Christian; Kobourov, Stephen; Nagra, Jasvir; Pitts, Jacob; Wampler, Kevin: A System For Graph-based Visualization of the Evolution of Software; p.77ff in: Proc. 2003 ACM Symposium on Software Visualization SoftVis'03; ACM; 2003

[4] Evolver: Analyzing Software Evolution with Animations and 3D-Visualizations (Project homepage); `http://pi.informatik.uni-siegen.de /projects/evolver`; 2009

[5] Kelter, Udo; Wehren, Jürgen; Niere, Jörg: A Generic Difference Algorithm for UML Models; p.105-116 in: Software Engineering 2005. Fachtagung des GI-Fachbereichs Softwaretechnik, 8.-11.3.2005, Essen; LNI 64, GI; 2005

[6] Kolovos, Dimitrios S.; Ruscio, Davide Di; Pierantonio, Alfonso; Paige, Richard F.: Different Models for Model Matching: An Analysis Of Approaches To Support Model Differencing; p.1-6 in: Proc. 2009 ICSE Workshop on Comparison and Versioning of Software Models; IEEE; 2009

[7] Lanza, M.: Recovering Software Evolution Using Software Visualization Techniques; p.37-42 in: Proc. 4th Intl. Workshop Principles Software Evolution IWPSE; ACM; 2001

[8] Lanza, Michele; Ducasse, Stéphane: Polymetric Views - A Lightweight Visual Approach To Reverse Engineering; IEEE Trans. Softw. Eng., 29:9, p.782ff; 2003

[9] Lungu, Mircea; Lanza, Michele: Softwarenaut: Exploring Hierarchical System Decompositions; p.351-354 in: Proc. Conference on Software Maintenance and Reengineering (CSMR '06), Washington, DC, USA, 2006; IEEE Computer Society; 2006

[10] Miller, Joaquin; Mukerji, Jishnu (eds.): MDA Guide Version 1.0.1; OMG, Document Number: omg/2003-06-01; 2003-06-12; `http://www.omg.org/docs/omg/03-06-01.pdf`

[11] SiDiff Differenzwerkzeuge; http://www.sidiff.org; 2008

[12] Treude, Christoph; Berlik, Stefan; Wenzel, Sven; Kelter, Udo: Difference Computation of Large Models; p.295-304 in: 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Sep 3 - 7, Dubrovnik, Croatia; 2007

[13] Wenzel, Sven; Kelter, Udo; Hutter, Hermann: Tracing Model Elements; p.104-113 in: 23rd IEEE International Conference on Software Maintenance (ICSM 2007), October 2-5, 2007, Paris, France; IEEE ; 2007

[14] Wenzel, Sven; Kelter, Udo: Analyzing Model Evolution; p.831-834 in: Proc. 30th International Conference on Software Engineering, Leipzig, Germany, May 1018, 2008 (ICSE'08); ACM Press; 2008

[15] Wenzel, Sven; Koch, Jens; Kelter, Udo; Kolb, Andreas: Evolution Analysis with Animated and 3D-Visualizations; p.475-478 in: Proc. 25th IEEE International Conference on Software Maintenance (ICSM 2009), 2009, Edmonton, Canada; IEEE; 2009

[16] Wettel, R.; Lanza, M.: Visual exploration of large-scale system evolution; p.219-228 in: Proc. 15th Working Conference on Reverse Engineering (WCRE), Washington DC, USA; IEEE Computer Society; 2008

[17] Wu, J.; Holt, J.; Hassan, A.: Exploring Software Evolution Using Spectrographs; p.80-89 in: Proc. Working Conference on Reverse Engineering (WCRE 2004); 2004