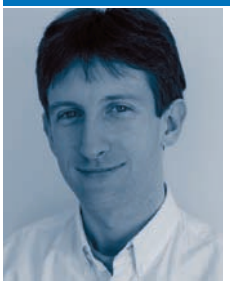
AUSGEREIFTES MODELLMANAGEMENT: MEHR ALS NUR XML UNTER VERSIONSKONTROLLE

Die Modellierung boomt und alle versuchen, Modelle mit bestehenden Versionierungswerkzeugen und Praktiken zu verwalten. Und alle leiden darunter: die Modellierer ebenso wie die Anbieter von Systemen zur Versionsverwaltung und von Modellierungswerkzeugen. In diesem Artikel gehe ich einen Schritt zurück und gehe der Frage nach, was tatsächlich für das Modellmanagement benötigt wird. Ich untersuche, wie es einem neuartigen Ansatz, der mehr für Modelle als für Text entwickelt wurde, in der Praxis ergangen ist. Dabei gehe ich darauf ein, welche alten Probleme bei der Verwaltung von Modellen, Varianten und Versionen dieser Ansatz löst – und vor allem wie. Außerdem zeige ich, welche neuen Probleme domänenspezifische Modellierungssprachen für das Modellmanagement mit sich bringen, und wie mit diesen verfahren wird.

der autor



Dr. Steven Kelly
(E-Mail: stevek@metacase.com)
ist CTO der Firma MetaCase. Er verfügt über langjährige Erfahrungen bei der Entwicklung von MetaCASE-Umgebungen und berät bei ihrer Anwendung für die domänenspezifische Modellierung.

Die in diesem Artikel vorgestellten Ergebnisse basieren auf Beispielen aus vielen Industriebranchen, darunter Telekommunikation, Medizin und Finanzen (siehe Kasten 1 am Ende des Artikels). Anhand dieser Beispiele und weiterer Erfahrungen aus den letzten 15 Jahren werden wir auch sehen, welche Ideen in der Theorie zwar großartig klingen, in der Praxis aber nicht funktionieren – und woran das liegt.

Versionierung von Text und Modellen

Von ihren bescheidenen Anfängen – als einfache Archive von Textdateien – haben sich Systeme zur Versionsverwaltung (*Version Control Systems, VCS*) so weiter entwickelt, dass sie heute eine wesentliche Rolle in der Softwareentwicklung spielen. Moderne VCS decken alle Arten der Verwaltung von Source-Artefakten ab und erfüllen drei wesentliche Aufgaben (vgl. auch [Alt09]):

- *Archivieren:* Ältere Versionen erhalten, sodass die Entwickler zu diesen zurückkehren können.
- *Zusammenarbeit:* Die Arbeit von verschiedenen Entwicklern zusammenführen.
- *Verzweigung:* Handhaben langfristig paralleler Varianten und Konfigurationsmanagement

Diese Aufgaben und die Fähigkeit der VCSs, diese effektiv zu bedienen, haben sich über die Jahrzehnte kontinuierlich weiter entwickelt.

Etwa alle 10 Jahr kommen wichtige neue VCSs auf den Markt und werden auch angenommen. Die erste Software zur Versionsverwaltung, *Source Code Control System (SCCS)* aus dem Jahr 1972, verfügte noch über keine Möglichkeiten zur Zusammenarbeit und effektiven Verzweigung. Das System wurde 1982 vom *Revision Control System (RCS)* abgelöst, das diese Bereiche abdeckte. Dessen Nachfolger, *Concurrent Versions System (CVS)* aus dem Jahr 1990 und *Subversion* aus dem Jahr 2000, bieten durchgängige inkrementelle Verbesserungen, aber keine grundlegenden Änderungen mehr. Verteilte VCSs, wie beispielsweise „Mercurial" und „Git", sind vor allem in der Open-Source-Entwicklung populär. In der eher zentralisierten Umgebung der kommerziellen hausinternen Entwicklung, auf die ich mich hier konzentriere, wurden sie bisher nicht so gut angenommen.

Text versus Objekte

Allen VCSs ist gemeinsam, dass sie sich auf Textdateien fokussieren – andere Arten von Dateien können gespeichert werden, aber wichtige Funktionen sind nur bei Text möglich. Sie funktionieren daher bei textorientiertem Quellcode wie C oder Java gut, aber sie besitzen kein Verständnis für die Semantik. Eine Änderung des Textes „Employee" innerhalb eines Strings wird beispielsweise genauso behandelt wie die Änderung des Klassennamens „Employee" in einer Methodensignatur. Die letztgenannte Änderung würde normalerweise dazu führen, dass überall dort, wo die Klasse „Employee" und der Name der Klassendatei „Employee" verwendet werden, eine entsprechende Änderung durchgeführt wird. Solche umfassenden Änderungen können mit Refaktorisierungswerkzeugen in einer Operation durchgeführt werden, aber aus der Sicht eines VCS handelt es sich um multiple unabhängige Änderungen über viele Dateien hinweg.

Die Wurzel des Problems liegt nicht beim VCS, sondern in der Benutzung von Text als Datenstruktur. Text ist ein einfacher eindimensionaler Bereich von Zeichen, während ein Programm ein komplexes Gebilde aus Knoten, Verbindungen und Eigenschaften ist. Anstatt einfach auf alle Referenzen auf die Klasse „Employee" zu verweisen, wie in einer objektorientierten Datenstruktur, müssen Textdateien die Buchstabenfolge „E", „m", „p" usw. an jeden Ort kopieren, der auf „Employee" verweist (siehe Abbildung 1). Der Compiler analysiert und verbindet diese Kopien später zurück zu echten Referenzen, aber das VCS muss mit der Duplizierung, die mit der Textrepräsentation einhergeht, umgehen.

Text versus Mehrbenutzer-Editierung

Die meisten Programmiersprachen speichern multiple Funktionsdefinitionen in einer einzigen Datei, die dann die Einheit der Granularität für das Editieren und die Versionsverwaltung bildet. Da das VCS die Struktur der Textdatei nicht kennt, gibt es keinen Weg, nur den Bereich zu sperren, der eine einzelne Funktion enthält, um anderen Benutzern zu erlauben, parallel an anderen Funktionen in der gleichen Datei zu arbeiten. Selbst wenn wir wissen, dass eine Funktion addEmployee() bei Zeichen 500 beginnt und bei Zeichen 600 endet, können wir diesen Bereich der Datei nicht sperren, da Einträge in Funktionen zuvor die Anfangs- und Endposition verändern können. Entweder muss die gesamte Datei gesperrt werden oder es muss für jeden Benutzer eine eigene Kopie erstellt werden, die dann später Zeichen für Zeichen wieder zusammengeführt werden müssen. Die Tatsache, dass die meisten dieser Zusammenführungen ohne Probleme funktionieren, ist kein Beweis für die Genialität der VCS-Algorithmen, sondern für die Tatsache, dass Entwickler meist unterschiedliche Funktionen bearbeiten.

Datei versus Repository

Während das Textformat der Programmiersprachen wegen der Trägheit der installierten Basis- und Hilfswerkzeuge nur

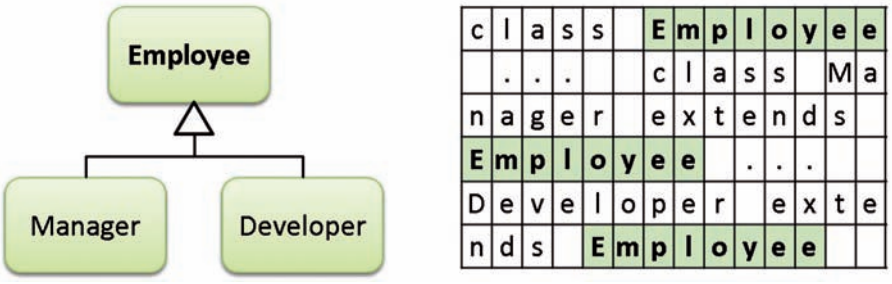


Abb. 1: Verbindung durch direkte Referenz vs. Verbindung durch String-Matching.

zu ändern ist, können Modellierungssprachen ihre eigene Datenstruktur für die Bearbeitung und Speicherung wählen. Alle Modellierungswerkzeuge benutzen eine speicherinterne Repräsentation, die eine Art von Abbildung darstellt (obwohl einige versuchen, das Modell in einen Baum zu zwingen, mit den damit verbundenen Problemen). Auch textorientierte Entwicklungsumgebungen (IDEs) halten heute oft den geparsten Graphen oder den abstrakten Syntaxbaum im Speicher – parallel zu der textuellen Repräsentation. Einige Modellierungswerkzeuge, z. B. „Simulink" und „Rational Rose", haben Textdateien als natives Speicherformat verwendet – auch XML-Dateien fallen in diese Kategorie. Die Dateien sind für Menschen schlechter lesbar, aber oft ist es einfacher, eigene Werkzeuge zu schreiben, um diese Dateien zu verarbeiten – obwohl das XML-Schema teilweise unlogisch ist, z. B. XMI.

Einige Tools, z. B. „LabVIEW", haben sich von den textorientierten Speicherformaten verabschiedet und verwenden binäre Dateien. Interaktion mit den Modellen erfolgte ausschließlich über das Werkzeug oder dessen Programmierschnittstelle. Angesichts der komplizierten Strukturen und Abhängigkeiten der Modelldaten, ist dies eher ein willkommenes Sicherheitsnetz als eine ernsthafte Einschränkung.

Dateibasierte Darstellungen haben generell ein Problem mit der Granularität. Auf der einen Seite enthält jede Datei bereits zu viele Informationen (z. B. verschiedene Funktionen), auf der anderen Seite enthält sie zu wenig: Der Inhalt bezieht sich auf etwas, das in einer anderen Datei gespeichert ist. Für einen einzelnen Entwickler und eine einzelne Version können ein einfacher Textabgleich oder eine Include-Anweisung ausreichend sein, aber über mehrere Versionen hinweg wird dies problematisch. Erst einmal erzeugt, hat eine Referenz für eine bestimmte Version eines bestimmten Elements in einer bestimmten Datei Gültigkeit, aber der Name dieses Elements oder Datei kann sich bei nachfolgenden Versionen ändern oder ein anderes Element kann es ersetzen. Führt der Link über Dateien hinweg und die anderen Versionen dieser Datei sind in der Versionsverwaltung im Archiv verborgen, gibt es keine einfache Möglichkeit, um sicherzustellen, dass der Link erhalten bleibt. Manchmal ist diese Art von Indirektion erwünscht, aber wenn nicht, wird das Problem durch die Aufspaltung in Dateien dem Entwickler aufgebürdet. Die Granularität von Referenzen wird künstlich gezwungen, genauso zu sein wie die Granularität der Sicherung, obwohl sie idealerweise erheblich größer wäre.

Um dieser Problematik zu begegnen, haben viele Werkzeuge einen Wechsel von Dateien zu Repositorys vorgenommen. In einigen wenigen Werkzeugen gibt es relationale Datenbanken, aber Relationen geben graphische Strukturen nur unzureichend wieder und Tabellen sammeln alle Dinge gleichen Typs, erlauben aber nicht die Modularisierung in Modelle und Untermodelle. In einigen Werkzeugen gibt es Objekt-Datenbanken, die den Graphen des Modells direkt als Objektgraph repräsentieren. Das ist sicherlich das natürlichste Format: Weil es am weitesten von der traditionellen, textorientierten Speicherung entfernt ist, weist es die größte Nähe zu den speicherinternen Strukturen auf, mit denen die Entwickler zu arbeiten gewohnt sind.

Modelle versus Versionskontrolle

Modelle besitzen – im Gegensatz zu Textdateien – naturgemäß eine Struktur, die sich an Graphen orientiert. Das hat sich bei den derzeitigen Versionskontroll-Systemen und -Prozessen als problematisch erwiesen, besonders wenn die Modelle als Text abgespeichert werden. Reine Textvergleiche von Modellversionen legen nur wenig verständliche Informationen über die Änderungen offen – außer bei einfachen Textänderungen. Der Textvergleich und die *Merge*-Funktionen von VCSs werden nutzlos oder sogar gefährlich, wenn sie auf Modelle angewendet werden – das ist kaum überraschend, weil diese ja nicht für diesen Zweck entworfen wurden.

Angesichts dieser Schwierigkeiten haben einige ältere und teurere Modellierungswerkzeuge eine eigenen Vergleichs- und sogar Versionierungsfunktionalität innerhalb ihrer Werkzeuge implementiert. Heutzutage ist diese Lösung weniger attraktiv: Entwicklungsprozesse sind streng an die Verwendung eines modernen VCS gebunden und Entwickler vermeiden Daten in Datenspeichern der verschiedenen Werkzeuge. Auch andere Faktoren sprechen gegen deren Verwendung, zum Beispiel das Minenfeld „Patente" rund um die Unterscheidung der Modelle oder das Fehlen eines aktuellen Erfolgs der Vergleichswerkzeuge selbst. Häufig werden die Ergebnisse von Vergleichen in einem einfachen Baum dargestellt. Manche fortschrittlichere Werkzeuge oder *Add-Ons* können die geänderten Bereiche der graphischen Modelle auch selbst hervorheben.

Ein anderes Problem für VCSs ist, dass viele Modellierungswerkzeuge mehrere Dateien für jedes Modell verwenden. Repository-basierte Werkzeuge haben üblicherweise mehrere Dateien, die gemeinsam behandelt werden müssen. Da VCSs generell einzelne Dateien bearbeiten, werden diese in der Regel als zip-Dateien versioniert. Auch dateibasierte Werkzeugen haben oft mehrere Dateien, z. B. um den konzeptionellen Inhalt des Modells von seinem graphischen Layout zu trennen. Bei dateibasierten Werkzeuge, bei denen die Modellierungssprache nicht festgelegt ist (das ist heute die Mehrheit), können andere Dateien erforderlich sein, die das Metamodell oder Profil beschreiben, um den Modelldateien einen Sinn zu geben. Diese sollten natürlich gesondert versioniert werden, aber eine versionierte Modelldatei sollte die Version der Modellierungssprache, mit der sie erzeugt wurde, aufzeichnen. Bei weniger ausgereiften Modellierungswerkzeugen, die nicht in der Lage sind, Dateien zu laden, die mit vorangegangenen Versionen erzeugt wurden, kann es außerdem erforderlich sein, die Version des Werkzeugs selbst zu speichern.

Unterstützung durch ausgereifte Modellierungswerkzeuge

Die Versionierung von Modellen in aktuellen VCSs ist folglich ein Alptraum. Einige der Probleme treten auch bei textorientierten Codedateien auf und Modelle verschärfen diese noch, andere Probleme treten speziell bei Modellen auf. Die Speicherung von Modellen als Text ist als Lösung gescheitert: Die Einführung mag einfacher erscheinen, aber im praktischen Einsatz zeigen sich die gleichen Probleme. Noch schlimmer ist: Je mehr man Modelle so ähnlich wie Text verwendet, umso weniger Nutzen kann man aus ihren Vorteilen ziehen, wie z. B. Skalierbarkeit, Vermeidung von Duplikaten und Kommunikation.

Wenn die Modelle lediglich als Ergänzung zu textbasiertem Quellcode, beispielsweise in Form von Zeichnungen und Dokumentation, verwendet werden, ist das kein großes Problem. Aber in der modellgetriebenen Entwicklung (*Model Driven Development, MDD*) sind Modelle die wichtigsten Source-Artefakte – hier benötigt man effektivere Lösungen, insbesondere bei zunehmender Verwendung. Viele MDD-Werkzeuge stecken noch in den Kinderschuhen – aber es gibt auch schon ausgereifte Werkzeuge, die seit 15 Jahren und länger im Einsatz sind, wie zum Beispiel „LabVIEW", „Simulink" und „MetaEdit+". Sie alle wurden bereits in Projekten unterschiedlicher Größe und Dauer eingesetzt – dabei sind auch Lösungen für die oben genannten Problemstellungen entstanden. Mit MetaEdit+ können Firmen außerdem ihre eigenen spezifischen Modellierungssprachen entwickeln, was ein immer wichtigeres Thema wird. Es ist auch das vom textbasierten Formaten am weitesten entfernte Werkzeug: MetaEdit+ verwendet ein Mehr-Benutzer-Objekt-Repository und sein Einsatzgebiet deckt die ganze Bandbreite ab – von eingebetteten Anwendungen bis hin zu Unternehmensanwendungen. Im weiteren Verlauf dieses Artikels liegt der Schwerpunkt auf dem Tool MetaEdit+, die Aussagen lassen sich aber auch auf andere Modellierungswerkzeuge übertragen.

Grundlagen von MetaEdit+

MetaEdit+ ist ein Modellierungswerkzeug, das auf verschiedenen Plattformen läuft und das verschiedene Modellierungssprachen (einschließlich von Anwendern neu definierten) sowie das parallele Arbeiten mehrere Anwender unterstützt. Außerdem kann man mit MetaEdit+ dasselbe Modell unterschiedlich darstellen, z. B. als Diagramm, Matrix, Tabelle oder verlinkten Text. Modelle und Metamodelle werden alle in einem Objekt-Repository gespeichert – entweder auf der lokalen Festplatte für einen einzelnen Nutzer oder auf einem Server als Mehrbenutzer-Repository. Im Folgenden betrachten wir den gesamten Anwendungsprozess. Wir sehen uns an, wie die drei Hauptfunktionen der Versionskontrolle – Archivierung, Zusammenarbeit und Verzweigung – umgesetzt werden.

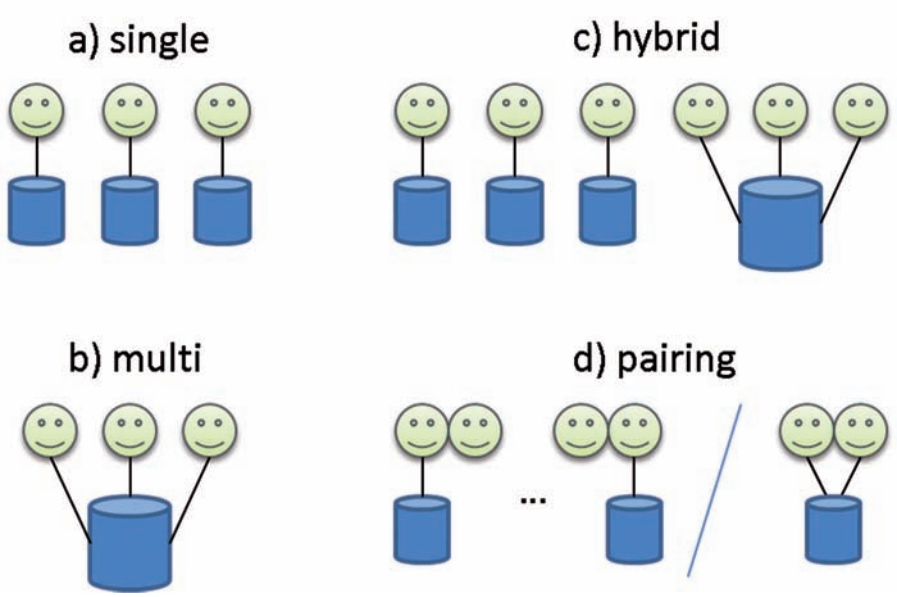


Abb. 2: Szenarien für die Nutzung von Repositorys.

Der Anwendungsprozess im Überblick

Ein Bestandteil von MetaEdit+ ist eine Bibliothek, die über 50 existierende Modellierungssprachen umfasst, aber die meisten Kunden verwenden das Werkzeug für eine domänenspezifische Modellierung (vgl. [Kel08]): Dabei erzeugt ein Experte der Anwendungsdomäne eine für sein Problemfeld spezifische Modellierungssprache und Generatoren, die den passenden Code erzeugen. Die anderen Anwender erzeugen Applikationen, die sie mit der neu definierten Sprache modellieren. Der Schwerpunkt von MetaEdit+ ist die Einfachheit der Sprachentwicklung und -evolution: In der Regel benötigt man nur ein bis drei Personenwochen, um die Sprache und die Generatoren zu entwickeln. Die volle Unterstützung des Modellierungswerkzeugs ist dann ohne zusätzlichen Aufwand verfügbar.

Integration und Wiederverwendung sind für den Modellierer sehr wichtig. Während andere Tools denselben Text an zwei Stellen platzieren, um einen Link zu erzeugen, wird in MetaEdit+ der Link normalerweise als eine direkte Referenz von einem Element zum anderen erstellt. Wenn ein weiterer Indirektionsschritt erwünscht ist, kann der Link natürlich durch Kopieren der Zeichenkette erzeugt werden und die gewünschte Abfrage wird im Generator oder während der Kompilierung durchgeführt. Dasselbe gilt für Links von einem Repository zum anderen. Objekte können über mehrere Diagramme hinweg wiederverwendet werden und verschiedene Layer von Modellen und Modellierungssprachen können miteinander kombiniert werden, um alle Aspekte und Ebenen eines Systems abzudecken.

Die Einzelbenutzer- und Mehrbenutzer-Repositorys können auf mehrere Arten miteinander kombiniert werden, um den Bedürfnissen des jeweiligen Unternehmens bzw. Projekts gerecht zu werden (**siehe auch Abbildung 2**):

- *Verschiedene Einzelbenutzer-Repositorys:* Beispielsweise für einzelne kleine Produkte oder für Module, die zu einem größeren Produkt verbunden werden.
- *Ein einzelnes Mehrbenutzer-Repository*
- *Hybrid:* Einige einzeln, einige mehrfach, beispielsweise wenn ein einzelnes Produkt oder Modul als Aufgabe für einen einzelnen Entwickler zu groß wird.
- *Paarweise:* Zwei Entwickler, die sich einen PC oder ein Mehrbenutzer-Repository teilen, beispielsweise High-Level-UI-Struktur und Verhalten durch einen Interaktionsdesigner. Details auf unterer

Ebene werden von einem Entwickler zum selben Modell hinzugefügt.

Archivierung

Die Archivierung funktioniert wie üblich: Zip-Archive des Repositorys werden in einem beliebigen VCS gespeichert. Dadurch kann man auch auf einer alten Version aufsetzen, z. B. für die Wiederherstellung nach einer Havarie oder für die Behebung eines Fehlers in einer veröffentlichten Produktversion. Die Auswahl der Versionen kann manuell mit den normalen Betriebssystem-Operationen erfolgen oder – komfortabler – aus MetaEdit+, dem VCS oder der Entwicklungsumgebung heraus: MetaEdit+ enthält Skript-Funktionen, um diese Aufgaben zu automatisieren und eine Anpassung an den firmenspezifischen Werkzeugsatz und die jeweiligen Prozesse vorzunehmen.

Für das Ein- und Auschecken ganzer Repositorys empfehle ich, die Abläufe zu automatisieren, sodass der Entwickler sich nicht um die Einzelheiten kümmern muss. Eine kurze Skript-Datei wird vom VCS als Makro aufgerufen, um das Repository im richtigen Verzeichnis zu packen bzw. zu entpacken. Werden Rechte außerhalb von MetaEdit+ behandelt (z. B. im VCS oder im Betriebssystem), kann das Skript so erweitert werden, dass es sich in das Repository einloggt und auf Wunsch ein spezifisches Projekt oder einen bestimmten Graphen öffnet. Diese Funktionalität kann in die Entwicklungsumgebung integriert werden, sodass zum Beispiel der Nutzer einen Repository-Baum, Projekte und Graphen in Eclipse sehen und einen Graphen mit einem Doppelklick für die Bearbeitung auswählen kann. Solche Funktionen, wie das unkomplizierte Öffnen eines Graphen, mögen die Entwickler. Wenn zu viele manuelle Schritte erforderlich sind, hat man schließlich, wenn dieser endlich aufgeht, bereits vergessen, warum man den Graphen eigentlich öffnen wollte.

Nach meinen Erfahrungen sinkt durch den Einsatz domänenspezifischer Modelle und eines Mehrbenutzer-Repository das Bedürfnis, viele Zwischenversionen individueller Komponenten zu speichern: Im Prinzip erhält man quasi nebenbei eine kontinuierliche Integration (*Continuous Integration*) geschenkt. Anstelle viele Zwischenversionen zu erzeugen, reicht es aus, nur Produkt- und Test-Veröffentlichungen zu versionieren. Sogar bei traditionellen VCS-Systemen und textorientiertem Code kommt es selten vor, dass eine komplette Komponente um mehr als eine Version zurückgeführt werden muss. Meistens reicht es aus, einen kleinen Teil – ein oder zwei Zeilen – einer alten Version zu nehmen und diese in die aktuelle Version einzufügen.

Die meisten Aspekte im Zusammenhang mit den Themen „Vergleich" und „Verschmelzung" (*Merge*) behandle ich im nächsten Abschnitt. Aber ein Vergleich ist auch dann nützlich, wenn eine neue Version gespeichert wird: als Hilfe beim Verfassen von Versionskommentaren und um zu überprüfen, dass man nichts geändert hat, was man nicht wollte. Unsere Kunden konnten den Vergleich von Modellen erfolgreich verbessern, indem sie einen modellierungssprachenspezifischen Generator geschrieben haben, der das Modell in einem menschenlesbaren Textformat ausgibt. Da sie sich ihre eigenen graphischen Sprachen und Codegeneratoren erzeugen, gab es kaum Probleme, den benötigten einfachen Textgenerator hinzuzufügen. Jede Version enthält auch ihre Textzusammenfassung, sodass die neue Version einfach mit ihrem Vorgänger verglichen werden kann. Anders als beim Textformat für die Speicherung von Modellen muss hier nicht jedes kleine Detail enthalten sein. Das Textformat kann bezüglich Lesbarkeit und Textvergleich optimiert werden und muss nicht eine möglichst einfache Konvertierung in und aus einem Graphen unterstützen.

Zusammenarbeit

Anstelle eines nachträglichen *Mergens* oder der pessimistischen Sperrung ganzer Modelle haben wir eine feine Granularität der Kapselung. Die Informationen über das graphische Layout eines Diagramms werden zum Beispiel separat von jedem Objekt des Diagramms gesperrt, sodass eine Person das Layout verschönern kann, während andere individuelle Objektdaten bearbeiten. Die Daten jedes Objekts werden separat gespeichert und gesperrt – bis hinunter zu den individuellen Attributen.

Manche Designvorgänge lassen sich mit der Eigenschaft *ACID (Atomicity, Consistency, Isolation und Durability)* charakterisieren – sie dauern sehr lange, d. h. Minuten bis Stunden. Wir ignorieren Lese-Schreib-Konflikte; Entwickler sind es gewöhnt, nur die letzte Version der Arbeit eines anderen Entwicklers zu sehen, sodass es keine Rolle spielt, ob der Besitzer einer Arbeit diese in einer parallelen Transaktion aktualisiert hat. Die Forschung hat gezeigt, dass in Design-Repositorys weniger als 3 von 1.000 Operationen zu Schreib-Schreib-Konflikte führen. Wenn die Granularität fein gehalten wird, ist das sowieso kein Problem. Wenn die Granularität aber auf ganze Modelle ausgeweitet wird (wie bei traditionellen Repository-Systemen oder bei dateibasierten *Merge*-Systemen), gibt es oft Änderungen von zwei Entwicklern innerhalb derselben Einheit. (Natürlich können – unabhängig von der Granularität – immer semantische Konflikte auftreten. So kann beispielsweise Objekt A auf der einen Seite der Modellwelt irgendwie semantisch mit Objekt B auf der anderen Seite verbunden sein, obwohl keine derartigen Links im Modell selbst bestehen. Das deutet auf ein Problem in der Modellierungssprache hin, die eine wichtige Verbindung nicht darstellen konnte.)

Verzweigung

In textorientierten VCSs ist ein Zweig (*Branch*) in der Regel die Kopie einer Datei, der – häufig aus Gründen der Speichereffizienz – einen Vergleich zur Grundversion darstellt. In einem VCS-Zweig kann sich jeder Teil der Datei beliebig ändern, sodass das Werkzeug nur geringe echte Unterstützung bietet. In der domänenspezifischen Modellierung (*Domain Specific Modelling – DSM*) wird anhand des Software-Produktlinien-Engineerings, die Notwendigkeit von Zweigen überprüft: Welche Produktvarianten gibt es, worin können sie sich voneinander unterscheiden und welche Teile sind immer gleich? Die Gemeinsamkeit und die Variabilität werden getrennt voneinander behandelt, sodass es kein Kopieren von großen, gleich bleibenden Teilen gibt. Die Variabilität wird oft dadurch erreicht, dass eine Modellierungssprache entwickelt wird, die es den Entwicklern erlaubt, diese auszudrücken. Nur die Variationen, die für diese Produktfamilie erlaubt sind, können spezifiziert werden. Die Sprache deckt nur die Punkte ab, die sich ändern können – das ist ein großer Unterschied im Vergleich zu normal „kopierten" Zweigen. Jede Produktvariante hat in dieser Variabilitätssprache ihr eigenes Modell und alle Produktvarianten haben dieselben Gemeinsamkeiten (entweder explizit als Modell oder implizit, z. B. in den Generatoren oder dem Framework-Code). Ein ähnlicher Ansatz kann benutzt werden, um die Konfigura-

In allen hier beschriebenen Fällen wurden die Modelle mit MetaEdit+ editiert und die Repositorys wurden im aktuellen VCS des Kunden gespeichert.

Telecom: 15 Jahre, hunderte von Entwicklern, Dutzende Standorte weltweit
Hauptsächlich kommen Einbenutzer-Repositorys zum Einsatz, jedes mit einem Feature, das von einem Entwickler bearbeitet wird. Das funktionierte 10 Mal so gut, weil die Features gut modularisiert und mit den bestehenden VCS-Praktiken vernetzt waren. Als die Features wuchsen, erlaubte die Mehrbenutzer-Version, dass mehrere Nutzer am selben Repository arbeiteten. Updates des Metamodells wurden an jedes Repository weitergegeben, da MetaEdit+ die bestehenden Modelle automatisch ohne Datenverlust aktualisiert.

Finanzwesen: Versicherungsprodukte, Mehrbenutzer, ein Standort
Die Modellierungssprache und die Generatoren wurden innerhalb von zwei Wochen nach der Spezifikation des Kunden implementiert. Mit ihnen können Versicherungsexperten anstelle von Java-Entwicklern Modelle und kompletten Code erzeugen. Das Werkzeug arbeitete 3–5 mal so gut, aber der Modellierungssprache fehlte die Unterstützung für Variabilität. Dies hätte es ihnen ermöglicht, ein Kernmodell für eine generische Versicherungspolice, z. B. für Gebäude, zu erstellen, und hätte sie in die Lage versetzt, kleinere Erweiterungen oder Differenzmodelle zu spezifizieren, für jede aktuelle Gebäudeversicherungspolice, die von verschiedenen Unternehmen angeboten wurde, anstatt die Modelle zu kopieren und zu ändern.

Medizinsektor: Konfiguration, Standorte auf verschiedenen Kontinenten
Die Verwendung eines direkten Mehrbenutzer-Repositorys über den Atlantik hinweg war aufgrund der Latenz des Netzwerkes mühsam. Viel bessere Ergebnisse wurden durch eine ferngesteuerte Desktop-Verbindung zu einem Client, der im selben Netzwerk wie der Server betrieben wurde, erzielt: Die Datenmengen, die an Mausbewegungen und Bildschirm-Aktualisierungen beteiligt waren, war wesentlich kleiner als von den Modellen selbst gefordert. Dieses Projekt versuchte, Modelle im XML-Format zu speichern, stieß aber auf die in diesem Artikel beschriebenen Probleme der Modellversionierung und auf Beziehungen zwischen den Modellen und machte mit dem Mehrbenutzer-Repository weiter.

Verteidigung: Luft- und Raumfahrt, kleines Team an einem Standort
Der Kunde hatte die Anforderungen der Bundesluftfahrtbehörde der USA (FAA) zu erfüllen, dass jede Anwendung in zwei verschiedenen Modellierungssprachen modelliert werden soll. Sie benutzten FUSION (objektorientiert) und SA/SD (strukturiert) mit beiden Sprachen und deren Modellen im selben Mehrbenutzer-Repository, was die simultane und parallele Entwicklung vereinfachte. Wegen der kleinen Teamgröße und der mangelnden Erfahrung mit Datenbanken war jedoch die Hilfe des Anbieters für die Repository-Administration erforderlich.

Kasten 1: *Anwendungsbeispiele mit MetaEdit+.*

tion von Komponenten für verschiedene Versionen zu verwalten, z. B. eine Produktionsversion parallel zu einer Testversion.

Ausblick

Beim Thema Modell-Vergleich und -Verschmelzung gibt es noch einiges zu tun. Die wichtigsten Anforderungen aus der textorientierten Versionskontrolle werden für Modelle dadurch gelöst, dass sie vermieden werden: Diesen Zweck erfüllen Mehrbenutzer-Versionen und der Einsatz einer DSM-Sprache für die Variabilität. In der Tat ist die Lösung für Modelle in vielerlei Hinsicht besser als für Text. Der Vergleich mit einer Vorgängerversion wird mit der textorientierten Modellzusammenfassung umgesetzt. Trotzdem bleiben noch einige Fälle, bei denen es praktisch wäre, zwei Modelle miteinander zu vergleichen und sie zu verschmelzen, insbesondere wenn Objekte in den Modellen massiv wiederverwendet werden. Es genügt nicht, ein Modell zu erhalten, das richtig aussieht, und bei dem irgendein Foo auf irgendein Bar verweist: Es müssen auch das richtige Foo und das richtige Bar sein. Das kann man erreichen, indem man beispielsweise eindeutige interne IDs mit dem Vergleich von Attributwerten kombiniert. Das Hauptproblem besteht dabei darin, eine Benutzungsschnittstelle zu erstellen, die der Modellierer sinnvoll verwenden kann. Die Vielzahl von Patenten zum graphischem Vergleich und Merge, die von der Firma National Instruments entwickelt wurden, dürfte viele Ansätze ausschließen. Wie viele Werkzeuge (wie z. B. „EMF Compare") gegen diese Patente verstoßen, ist unklar.

Eine automatische Unterstützung wurde für einen speziellen und recht häufigen Fall der Verschmelzung hinzugefügt: Es existieren verschiedene unverbundene Repositorys und ein Modellierer erzeugt einige Kernmodelle, die alle anderen importieren sollen, um dann darauf zu verweisen und sie in ihren eigenen Modellen wiederzuverwenden. Die Kernmodelle können zurück exportiert und später importiert werden und werden die zuvor importierten überschreiben, ohne Dopplungen oder Schäden an anderen Modellen zu erzeugen.

Fazit

Bestehende VCS wurden weiter entwickelt und bieten heute gute Unterstützung für textorientierte Sprachen. Aber diese Unterstützung lässt sich nicht einfach auf Modelle übertragen, auch wenn sie als Text gespeichert sind. Das Speichern der Modelle im Textformat bringt eine Reihe weiterer Probleme mit sich. Durch die Rückkehr zu den grundlegenden Prinzipien und Funktionen, die wir von der Versionskontrolle benötigen, finden wir geeignete Wege, um die Entwicklung von mehrfach integrierten Modellen durch mehrere Nutzer zu verwalten. Vorhandene VCS spielen nach wie vor eine wichtige Rolle beim Modellmanagement, aber einige Funktionen sollten besser den Modellierungssprachen, Werkzeugen und Repositorys selbst überlassen bleiben. Direkte Verbindungen zwischen Modellelementen, Mehrbenutzer-Repositorys und domänenspezifischen Sprachen für Produkte und deren Änderung tragen gemeinsam dazu bei, DSM-Werkzeuge sinnvoll einsetzen zu können. ■

Literatur

[Alt09] K. Altmanninger, M. Seidl, W. Wimmer, A Survey on Model Versioning Approaches, in: International Journal of Web Information Systems (IJWIS), Vol. 5 Nr. 3, 2009

[Kel08] S. Kelly, J.-P. Tolvanen, Domain-Specific Modeling, Wiley 2008